UNIVERSIDAD SIMÓN BOLÍVAR

Dpto. de Computación y Tecnología de la Información
CI-2692 - Laboratorio de Algoritmos y Estructuras de Datos II

Proyecto 1

Un algoritmo divide-and-conquer para el problema del agente viajero euleriano

---

Alumna:
Ka Man Fung
Carné: 18-10492

Profesor:
Guillermo Palma

Junio 2021

Índice

1. Introducción

El problema del agente viajero (*TSP* por sus siglas en inglés, Travelling Salesman Problem) es uno de los algoritmos de optimización más estudiados en las ciencias de la computación. Dicho problema consiste en que, dado un conjunto de ciudades, se quiere encontrar un tour o recorrido completo que conecte todas las ciudades, visitándolos tan solo una vez y volviendo al punto de partida, y que además minimice la distancia total del tour. En esta ocasión, se estudiará la versión del TSP euleriano, en la que las ciudades son puntos del plano cartesiano y las distancias entre las ciudades corresponde a las distancias entre los puntos en el plano. Dado que la distancia entre una ciudad A hasta una ciudad B es igual a la distancia entre una ciudad B hasta una ciudad A, hace que el problema del TSP euleriano sea de la categoría TSP simétrico. Para ello, se implementará la solución de este problema de optimización con la técnica de divide-and-conquer. El algoritmo consiste principalmente en dividir el arreglo de ciudades en pequeñas instancias, para construir y combinar los ciclos conformados por lados (recorrido entre dos ciudades) de manera recursiva. Para construir los ciclos, se revisa cada combinación de pares de lados de los ciclos y se sustituye por dos nuevos lados que unan a los ciclos, tal que la distancia sea la menor posible.

2. Diseño de la solución

2.1. Estructuras de datos

Para construir la solución TSP euleriano, las estructuras de datos utilizadas para representar los elementos más importantes del proyecto fueron:

Ciudades: cada ciudad consta de un punto, es decir, de un par ordenado de números flotantes. Ejemplo: (1.0, 9.0).

Lados: es el recorrido de una ciudad A hasta otra ciudad B, denotado como un par ordenado de dos ciudades. Ejemplo: ((1.0, 9.0), (4.0,1.0)).

Ciclos: es una secuencia de lados, que comienza y termina en la misma ciudad. Ejemplo: [((1.0, 1.0), (4.0, 0.0)),((1.0, 1.0), (3.0, 8.0)),((4.0, 0.0), (3.0, 8.0))].

Particiones: consta de un arreglo de ciudades (arreglo de coordenadas o puntos) que se encuentran en un rectángulo determinado.

Rectángulos: consta de un arreglo de las coordenadas de números flotantes de un rectángulo, donde el primer elemento es la coordenada (x mínima, y mínima), la segunda es la coordenada (x máxima, y mínima), la tercera es la coordenada (x mínima, y máxima) y la cuarta es la coordenada (x máxima, y máxima).

2.2. Detalles de implementación

Para la implementación del algoritmo divide-and-conquer del problema TSP, además del pseudocódigo proporcionado, fueron introducidos varios elementos de diseño de la solución:

En el Algoritmo 2, ***obtenerParticiones***, se utilizó dos funciones añadidas, rectanguloCoords y rectanguloDim, funciones que retornan las coordenadas de un rectángulo que abarque todos los puntos de las ciudades y las dimensiones de los lados del rectángulo. Para ordenar el arreglo de ciudades en la función ***obtenerPuntoDeCorte***, se implementó la versión modificada de *QuickSort (IntroSort)*. Se escogió dicho algoritmo, ya que como vimos en la práctica, presenta mejores tiempos de ejecución que el resto de algoritmos como *MergeSort* y las demás variaciones de *QuickSort*. Para ello, a la función de ordenamiento se agregó como argumento adicional el parámetro del eje de coordenada al que se quiere ordenar (X o Y). Por ejemplo, si se trataba del eje X, se ordenaba con respecto a la primer valor de cada punto y si existen valores repetidos en ese eje X, se ordena con respecto al otro eje. Así, para ordenar el arreglo de ciudades considerando ambas componentes de las coordenadas se agregó la función relacionDeOrden, que recibe dos ciudades y un eje a comparar, si las ciudades comparten un mismo valor en la componente del eje dado, se compara con respecto al eje contrario.

En la función ***combinarCiclos***, luego de encontrar los lados a eliminar y agregar, se prefirió no remover del *Ciclo1* y del *Ciclo2* los lados a eliminar, ya que esto requería de crear un nuevo arreglo (como los arreglos tienen tamaño fijo) y realizar un copia. Por ello, para crear el nuevo *Ciclo3*, se fue agregando los lados del *Ciclo1* y *Ciclo2*, y si se trataba de un lado a eliminar, se agregaba el lado nuevo que se quiere añadir. Luego, para ordenar el nuevo *Ciclo3* y mantener la propiedad de que cada lado y su lado más próximo compartiera una misma ciudad, se implementó una versión modificada de *SelectionSort*. En dicha modificación, por cada lado, se busca el siguiente lado que comparta alguna ciudad en común y se coloca en la posición más próxima al lado actual. Por otro lado, para calcular la distancia entre dos ciudades, se prefirió realizar el cálculo sin redondear a números enteros, ya que se quería tener una mayor precisión al escoger la distancia mínima entre los nuevos y viejos lados de la función ***combinarCiclos***. Se realizó pruebas con distancias redondeadas, pero estas presentaban peores resultados en la distancia total, con respecto a las no redondeadas.

Como en el Algoritmo1, ***divideAndConquerTSP***, no se pudo implementar una manera sencilla en que los ciclos obtenidos comenzaran con la primera ciudad dada en el archivo de entrada, ya que es una función recursiva (no necesariamente el ciclo que recibe contiene la primera ciudad) y no recibe como parámetro cuál es la primera ciudad dada, para mantener dicha propiedad de una solución válida de TSP, fue implementado en el ***main***. Por ello, en el ***main*** se realiza una búsqueda lineal del índice de la primera ciudad dada en el ciclo obtenido y se realiza una partición del arreglo tal que

el ciclo final comience con la primera ciudad dada. Una opción para mantener que el ciclo comenzara con la primera ciudad dada en la función ***divideAndConquerTSP***, pudo ser trabajar con tripletas en la estructura de datos de una ciudad, tal que una de las componentes sea su posición o orden en que estaba originalmente en el archivo de entrada. Pero, debido a las limitaciones de tiempo y que requería realizar cambios en numerosas funciones, se prefirió mantener las ciudades como pares.

Luego, para verificar que el ciclo obtenido es un tour válido, se agregó una función llamada verificadorTSP, función utilizada en el ***main***. Dicha función verifica que el tour tenga el mismo tamaño que el arreglo de ciudades, que comience y finalice con la misma ciudad, que comience con la primera ciudad del archivo TSP, que cada ciudad sea visitado una sola vez, que todas las ciudades dadas estén en el tour y que cada lado del tour comparta una ciudad con el lado siguiente.

Finalmente, se agregó la función extraerCoords, función que recibe un archivo de entrada que contiene los datos de una instancia TSP y retorna un arreglo con las coordenadas de las ciudades (puntos) del archivo. Si bien algunas instancias eran coordenadas de números enteros o flotantes, se prefirió convertir los enteros en tipo Double, para facilitar la ejecución del algoritmo. Asimismo, se implementó la función tourSolucion, que dado un arreglo de ciudades y un tour solución TSP, retorna un arreglo de los índices de las ciudades en el orden en que fueron visitadas. Para escribir en el archivo de salida, se creó la función escribirOutfile, que dado una dirección de un archivo (path), las distancia total del tour solución TSP y del arreglo con los índices de la ciudades según el orden en que fueron visitadas (obtenido con tourSolucion), escribe sobre el archivo la solución del tour TSP. Las anteriores funciones fueron utilizadas en el ***main***, ya sea para pasar los datos necesarios a la función ***divideAndConquerTSP*** y procesar los resultados obtenidos.

2.3. Dificultades del proyecto

Las mayores dificultades que se presentaron durante la realización del proyecto fue al realizar las funciones ***obtenerPuntosRectangulo*** y ***combinarCiclos***, además de la implementación del algoritmo de ordenamiento. Para la implementación del algoritmo de ordenamiento, se probó con *MergeSort*, *Randomized QuickSort* y *QuickSort (IntroSort)*. Tras varias ejecuciones y comparaciones con respecto al tiempo de ejecución, se prefirió *QuickSort (IntroSort)*. La mayor dificultad en dicho algoritmo se presentó al tener ordenar con respecto a dos ejes, por lo que se tuvo que considerar que si dos elementos eran iguales en un eje, se tenía que ordenar con respecto al eje contrario.

Por otro lado, con la función ***obtenerPuntosRectangulo***, al sólo recibir como parámetros el arreglo de ciudades y las coordenadas del rectángulo, se intentó varias maneras de poder determinar si se trataba de un rectángulo izquierdo, derecho, su-

perior o inferior. Si bien no hubo problemas en determinar y extraer las ciudades de un rectángulo izquierdo/inferior, para el caso del rectángulo derecho/superior no fue lo mismo. Lo primero que se intentó fue que, al tener el arreglo de ciudades ordenado, si la primera ciudad se encontraba dentro del rectángulo dado, se trataba de uno izquierdo/inferior, por lo que se podía retornar el arreglo de las ciudades que se encontraban dentro de esas coordenadas sin excluir los del punto de corte. Pero, si dicha ciudad no se encontraba en el rectángulo izquierdo/inferior, se debía averiguar si era derecho o superior para realizar la exclusión de las ciudades según el eje de corte X o Y. Si bien se agregó varias instrucciones condicionales para determinar si la primera ciudad sobrepasaba del X mínimo (se trataba de un rectángulo derecho) o del Y mínimo (se trataba de un rectángulo superior) del rectángulo, no fue suficiente ya que en varias instancias en que el rectángulo dado fuera una línea, es decir, las componente X (o Y) de todas las coordenadas eran iguales, fallaba el algoritmo e incluía las ciudades incorrectas.

Así, otro intento fue que, en ambas particiones de los rectángulos izquierdo y derecho, se considerara las ciudades que se encuentran en el eje de corte. Pero, al tener que hacer la exclusión de las ciudades que tienen en común, se tendrían que modificar el pseudocódigo de la función ***obtenerParticiones***. Además, como los arreglos son de tamaño estático, se tendrían que crear nuevos arreglos para realizar la exclusión. Por lo que, por la limitación de tiempo, como opción final se modificó la función ***aplicarCorte***, tal que el rectángulo derecho fuera desplazada en el eje X (Y si es un rectángulo superior) 0.000001 decimales, para evitar incluir ciudades de más en ambos rectángulos. Si bien, no es la mejor solución ante este problema, con dicha modificación al algoritmo, el programa dio funcionamiento con las instancias proporcionadas.

Con respecto a la función ***combinarCiclos***, la mayor complejidad se presentó al tener que ordenar el *Ciclo3*, tras eliminar y agregar los nuevos lados, para que cumpliera con la propiedad de que cada lado fuera continuación del siguiente (comparten una ciudad). Como se mencionó anteriormente, al final se escogió una versión modificada de *SelectionSort*, ya que dicho algoritmo es de fácil implementación. Por un primer momento, se había implementado una búsqueda lineal en el arreglo del *Ciclo3* y en el arreglo de las ciudades P, tal que se construía un nuevo ciclo de manera ordenada, pero dicho algoritmo resultaba ineficiente ya que recorría numerosas veces ambos arreglos y requería de crear un arreglo auxiliar.

2.4. Operatividad y/o problemas

Tras numerosas correcciones y ejecuciones, el programa entregado se encuentra operativo, no presenta problemas con las instancias de TSP proporcionadas. Asimismo, con la función implementada para verificar la solución TSP generada, con las instancias probadas no presentó problemas conocidos.

3. Resultados experimentales

3.1. Datos de la plataforma

- **Sistema de operación:** GNU/Linux (Linux Mint 20).
- **CPU:** Quad Core Processor (up to 1.4 ghz).
- **Memoria RAM:** 4GB.
- **Versión de Kotlin:** kotlin-jvm version 1.5.0-release-749 (JRE 11.0.9.1+1).
- **Versión de JVM:** OpenJDK Runtime Environment (11.0.9.1) .

3.2. Resultados de las instancias TSP

En los siguientes cuadros se presentan las distancias obtenidas con diversas instancias de TSP EUD_2D, junto al porcentaje de desviación con respecto al valor óptimo de las soluciones TSP[1].

Nombre de la instancia	Distancia de la solución	Porcentaje de desviación (%)
berlin52	8862	17.5019888623707
bier127	143302	21.1528381325984
brd14051	565609	20.5000159783547
ch130	7117	16.481178396072
ch150	7798	19.4546568627451
d198	17645	11.8187579214195
d493	41616	18.8960630821096
d657	60905	24.5195453058554
d1291	66489	30.8812818645302
d1655	74703	20.2404712850888
d2103	107780	33.9714108141703
d15112	1908216	21.3041388762456
eil51	465	9.15492957746479
eil76	588	9.29368029739777
eil101	696	10.6518282988871
fl417	16629	40.198971418936
fl1400	27815	38.1974462165251
fl1577	31898	43.3682412692705
fl3795	41131	42.9549562074239

Cuadro 1: Soluciones TSP (Parte I)

[1]http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html

Nombre de la instancia	Distancia de la solución	Porcentaje de desviación (%)
fnl4461	219626	20.2995081230897
gil262	2821	18.6291000841043
kroA100	25090	17.8930551639883
kroA150	31722	19.5973458000302
kroA200	35223	19.9366657586489
kroB100	24910	12.5062101982747
kroB150	31551	20.7462686567164
kroC100	24614	18.6274037303003
kroD100	24792	16.4271625810087
kroE100	25214	14.2559361972086
lin105	16979	18.081925029557
lin318	51234	21.9015441718813
nrw1379	67798	19.7040855962428
p654	47985	38.5128308749242
pcb442	59198	16.5819843239198
pcb1173	70275	23.523518245096
pcb3038	167856	21.9050939038738
pr76	126818	17.2514538780869
pr107	50627	14.2744283682821
pr124	70175	18.8802303913264
pr136	107580	11.1685198197826
pr144	62714	7.13565778909066
pr152	88813	20.5355446377677
pr226	99630	23.965708171061
pr264	58841	19.7537396967538
pr299	59301	23.0540972380735
pr439	134209	25.1751121557216
pr1002	324532	25.2801636781254
pr2392	481393	27.3418652389216
rat99	1382	14.1205615194055
rat195	2716	16.9177787343952
rat575	7950	17.3778237117968
rat783	10489	19.1119691119691
rd100	8560	8.21744627054362
rd400	18190	19.0367122570512
rl1304	344839	36.328019988298
rl1323	360364	33.3698496293473
rl1889	432403	36.6046831955923
rl5915	757251	33.9011193040157
rl5934	742107	33.4616802596912
rl11849	1186288	28.4879363305129

Cuadro 2: Soluciones TSP (Parte II)

Nombre de la instancia	Distancia de la solución	Porcentaje de desviación (%)
st70	813	20.4444444444444
ts225	140454	10.9054586514849
tsp225	4575	16.8283963227783
u159	50066	18.9781368821293
u574	45229	22.5552093212302
u724	51058	21.8277260796946
u1060	278654	24.3469258436192
u1432	176626	15.4644701575472
u1817	72801	27.2722504851314
u2152	78557	22.261995548846
u2319	244270	4.27481046376614
vm1084	303186	26.6986213784544
vm1748	444794	32.1604725513733

Cuadro 3: Soluciones TSP (Parte III)

3.3. Análisis de los resultados

Como se muestra en el Cuadro 1, Cuadro 2 y Cuadro 3, el promedio del porcentaje de desviación estándar en las soluciones TSP fue de un 21.20 %. La instancia que presentó mayor desviación fue *fl1577*, con un 43.36 %, mientras la que presentó una menor desviación fue *u2319*, con un 4.27 %. Como se puede observar, en promedio, el algoritmo presentó una buena solución ante la mayoría de las instancias de TSP. Sin embargo, los resultados obtenidos dependen principalmente de los valores de cada instancia, es decir, de las coordenadas de las ciudades. Como se puede observar en los cuadros, instancias con similares tamaños de ciudades, presentaron desviaciones distintas. Asimismo, instancias con una mayor cantidad de ciudades podían presentar una menor desviación con respecto alguna de menor tamaño. Ejemplo de esto se puede ver con la instancia más grande, *brd14051*, con una desviación de 20.50 %, en comparación con la desviación de *fl1577* o de *u2319*. Particularmente, las instancias que comienzan con el prefijo "*fl*" fueron las que presentaron una mayor desviación, instancias que si se observa con detenimiento, las coordenadas de las ciudades se encuentran de manera que están agrupadas en "subgrupos", tal que las coordenadas dentro de un "subgrupo" tienen valores muy similares. Sin embargo, cada "subgrupo" se encuentra muy distanciados de los demás "subgrupos", es decir, presentan valores de coordenadas muy diferentes, por lo que la posición de las ciudades no es uniforme. Por otro lado, quienes tuvieron una menor desviación con respecto al valor óptimo, fueron instancias con coordenadas de ciudades con una distribución más uniforme, es decir, la variación entre los valores de las coordenadas de las ciudades no eran muy grandes. Por lo que, el porcentaje de desviación con respecto al valor óptimo no se debe al número de ciudades de una instancia, sino más bien, varía según la posición en que encuentran las ciudades de una instancia.

4. Conclusiones

El problema del agente viajero (TSP) es uno de los problemas de optimización más estudiados en la computación y la investigación operativa, ya que se emplea en diversos campos y aplicaciones. El concepto de "ciudad" en este problema puede representar numerosas cosas, desde clientes hasta fragmentos de ADN; el concepto de "distancia" puede representar desde el tiempo o costo de un viaje hasta la medida del ADN. Si bien este problema, siendo NP-Hard, puede abarcar numerosas soluciones y puede complicarse en numerosas maneras, con el presente proyecto se ha evidenciado que es posible implementar una buena solución TSP con los conocimientos básicos de los algoritmos que hemos estudiado en el curso.

En este sentido, la solución principal del TSP implementado pone en práctica el diseño de algoritmos *divide-and-conquer*, al tener que dividir una instancia de un problema complejo, como la construcción de los ciclos, en instancias más pequeñas. De este modo, resulta más sencillo y eficiente encontrar solución a los subproblemas, para luego combinarlas y obtener la solución original. Por otro lado, con el conocimiento de diversos algoritmos de ordenamiento, se logró solucionar pequeños subproblemas de manera eficiente, ya sea ordenando los arreglos de ciudades con *QuickSort* u ordenar los lados de un ciclo con una modificación de *SelectionSort* para que sea una solución válida del TSP. Asimismo, tras varios "StackOverflow" e "IndexOutOfBoundsException", se logró entender de mejor manera el funcionamiento de los algoritmos recursivos y el cuidado que requieren al ser implementadas. Sin estos conocimientos, lo más probable es que se hubiera implementado una solución TSP costosa con respecto al tiempo de ejecución.

Además, durante la implementación del algoritmo, se probó numerosas maneras de realizar las funciones, desde el algoritmo de ordenamiento hasta la obtención de las ciudades de un rectángulo. Como también, escoger la estructura de datos para representar los elementos como las ciudades y lados, o escoger el tipo de datos como *Int* o *Double* para las distancias, pueden influir en gran manera cómo es implementado el algoritmo y la precisión del resultado obtenido. Esto se evidenció luego de implementar el proyecto con números enteros, al calcular las distancias mínimas, no fueron tan precisas como lo fue tras cambiar los números a reales. Por lo que, no basta en buscar la solución más obvia ante un problema, sino también encontrar la mejor manera de diseñarla, tras numerosas pruebas y ejecuciones.

Finalmente, podemos concluir que, tener un buen conocimiento de los algoritmos y de las estructuras de datos, como conocer los algoritmos de ordenamiento desde las diferentes instancias y su costo de ejecución, nos permite tomar mejores decisiones de diseño ante la solución de un problema. Así como encontramos una solución a uno de los problemas más estudiados e importantes como lo es el TSP euleriano, con la misma solución del TSP podemos encontrar o construir soluciones a otros problemas muchos más complejos y cotidianos.