Universidad Simón
Bolívar

Proyecto I

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

Autor:
Christopher Gómez 18-10892

Profesor:
Guillermo Palma

Laboratorio de Algoritmos y Estructuras de Datos II (CI2692)

20 de junio de 2021

1. Introducción

El siguiente informe trata de la resolución mediante un algoritmo aproximado basado en *Divide-and-conquer* del famoso problema de optimización llamado *problema del agente viajero*, o TSP por las siglas en inglés de *traveling salesman problem*. El problema consiste en hallar, dado un conjunto de puntos (llamados ciudades), el camino más corto que conecte a todos ellos, pasando por cada uno una sola vez y volviendo al punto inicial. El tipo de TSP que se quiere resolver es el **euleriano** y **simétrico**; **euleriano** porque la distancia entre ciudades corresponde a su distancia euclideana en el plano, y simétrico porque dadas dos ciudades A y B, la distancia entre A y B es la misma que entre B y A.

Se explican las estructuras usadas para representar los datos del problema, los detalles de implementacíón, las dificultades del proceso, los resultados obtenidos junto con el análisis de los mismos, y la conclusiones a las que llegaron.

2. Diseño de la solución

Para resolver el *problema del agente viajero* se utilizó un algoritmo heurístico basado en el paradigma *Divide-and-conquer* proveído por el profesor del curso, cuya idea principal es dividir el problema inicial (encontrar el tour más corto que pase por todas las ciudades una sola vez, y que comience y termine en la misma ciudad) en dos más pequeños de aproximadamente la mitad del tamaño del original, de forma recursiva hasta obtener entradas de 1, 2 o 3 ciudades, donde el tour solución es trivial, para luego combinar ciclos por pares mediante la eliminación un lado de cada uno y su reemplazo por un par de lados que los conecte de forma que se obtenga el recorrido más óptimo posible.

La implementación de los algoritmos se realizó en el lenguaje de programación **Kotlin**. La primera decisión de diseño tomada fue con qué estructuras se representarían los elementos del algoritmo dado, y de esta forma, el marco de trabajo escogido fue el siguiente:

- **Ciudades:** Representadas como pares de números reales, con la estructura **Pair<Double, Double>**, para la cual se usó el alias de **Point**.
- **Particiones:** Representadas como arreglos de ciudades, usando la estructura **Array<Point>**, para la cual se definió el alias de **CityArray**.
- **Lados:** Representados como pares de ciudades, mediante la estructura **Pair<Point, Point>**.
- **Ciclos:** Representados como arreglos de lados, usando la estructura **Array<Pair<Point, Point>>**, para la cual se definió el alias de **Cycle**.
- **Rectángulos:** Se decició representarlos usando sólo dos coordenadas, aquellas correspondientes a las esquinas inferior izquierda y superior derecha que delimitan el área del mismo. La estructura usada fue **Pair<Point, Point>**, a la cual se le dio el alias de **Rectangle**. Cabe destacar que aunque la estructura usada fue la misma que para los lados, en el código fuente se distingue entre **Pair<Point, Point>** como un lado y **Rectangle** con un rectángulo.

Con este marco de trabajo es que se procede a la implementación de los algoritmos.

Para el algoritmo de `obtenerPuntoDeCorte` se implemento una modificación de *Introsort* que recibe un **CityArray** y un caracter ('X' o 'Y') indicando el eje respecto al cual se ordenarían los pares de la entrada; la implementación usa el algoritmo de partición de Hoare con pivote en el último elemento, y luego de varias pruebas se decidió limitar el tamaño de las secuencias a ordenar con *Insertion Sort* a 16, que dio mejores resultados que tamaños como 8 o 32. Se usa *Introsort* debido a su notable mejora de los tiempos en la práctica con respecto a otros algoritmos como *Merge Sort*, y porque asegura un rendimiento destacable para cualquier secuencia a ordenar, independientemente de su forma.

Una de las primeras dificultades presentadas en la implementación se encontró en el algoritmo de `obtenerPuntos Rectangulo`, ya que se buscaba que si al momento de realizar un corte habían puntos que quedaban en una

arista compartida por ambos rectángulos resultantes, estos puntos quedasen en solo una de las dos particiones, pero no en ambas. Dado que la función no se podía modificar para que aceptara parámetros que indicasen si la partición a realizar era la izquierda o derecha, el primer acercamiento fue tratar de inferir de cuál partición se trataba, usando información sobre la partición y el rectángulo de entrada, y el hecho de que esta partición siempre está ordenada al momento de ser pasada como argumento de `obtenerPuntosRectangulo`.

Se intentó implementar varios condicionales que indicarían si el rectángulo estaba arriba, abajo, a la izquierda o a la derecha, basados en el primer y último punto de la partición y su pertenencia o no al área delimitada por el rectángulo, lo cual funcionó para varias instancias, excepto en algunas en las que resultaba un rectángulo en el que ambas esquinas eran colineales y paralelas a uno de los ejes (es decir, un rectángulo sin área), en cuyo caso los condicionales fallaban y agregaban el punto a ambas particiones. Luego, como la implementación se tornaba innecesariamente compleja, se desechó la idea y se decidió abordar el problema desde otro acercamiento, el cual fue dividir los rectángulos de forma distinta en la función de `aplicarCorte`, de modo que estos no compartieran ninguna arista, al mismo tiempo que tampoco se descartase ningún punto que pudiese haber en medio de ambos.

Así, la solución al problema fue, desde la función de `aplicarCorte`, generar un rectángulo izquierdo y derecho, o superior e inferior, de tal forma que la arista que ambos comparten esté separada por un pequeño salto y no se excluyan posibles puntos en ese salto. Se decidió que el salto sea de 0.0001 unidades, lo suficiente como para evitar dejar de excluir algún punto; se realizaron intentos con saltos de 0.2, los cuales fallaban en instancias donde habían ciudades separadas entre sí por menos distancia, y también fallaron los intentos con saltos muy pequeños de 1e-10 a 1e-6, debido a que la diferencia entre el exponente de las coordenadas del rectángulo y la distancia del salto provocaban redondeos en los cálculos que terminaban dejando las coordenadas sin ningún cambio y no se eliminaba el problema de aristas que se solapan. De tal forma, el salto escogido no presentó problemas para ninguna de las instancias probadas.

Con dicha implementación de `aplicarCorte`, el algoritmo de `obtenerPuntosRectángulo` consistió en agregar todos los puntos contenidos en el área delimitada por el rectángulo dado, incluidas todas sus aristas. Dado que se usó la estructura **Array** para implementar las particiones, y estas son de tamaño fijo (es decir, no se podía hacer *append* de las ciudades que pertenecieran al rectángulo), se buscaron en el algoritmo los índices a y b de la partición de entrada tales que todos las ciudades en $[a..b)$ estén dentro del rectángulo, para calcular el tamaño del arreglo de salida y retornarlo. Esto es válido ya que el arreglo de entrada estará siempre ordenado con respecto al eje de corte, por lo que todos los puntos dentro de un rectángulo siempre conforman una sección continua del **CityArray** de entrada.

Posteriormente, el algoritmo de `combinarCiclos` también presentó complicaciones al momento de ser implementado. La primera de ellas fue la de mantener los lados en el orden correspondientes al efectuar la combinación, tal que en el ciclo resultante los lados adyacentes siempre compartan una de sus ciudades. Se idearon algoritmos para lograr hacer esto en un solo procedimiento, agregando los lados al `ciclo3` a la vez que se mantenía el orden en el proceso, sin embargo, el algoritmo era complicado y fallaba en varias de las instancias. De esa manera, se separó el problema de combinar en dos partes: primero se concatenan el `ciclo1` y el `ciclo2`, mientras se reemplan los lados a elminar por los lados a agregar, y luego se permutaría el arreglo `ciclo3` para obtener el ciclo deseado, en el que cada lado comparta una ciudad con su adyacente en el arreglo.

Para lograr permutar el ciclo resultante de esta manera se implementó un algoritmo basado en `Selection Sort`, en el que por cada lado del arreglo, se busca el lado que comparte una ciudad con él y se intercambia el elemento de adelante por el correspondiente. Como en todo arreglo siempre habrán solamente dos lados que compartan una misma ciudad, el algoritmo implementado fue lo suficientemente eficiente para quedarse en el programa de solución, ya que al buscar el lado adyacente no se necesitaba escanear linealmente el arreglo entero por cada iteración (a diferencia de `Selection Sort`), sino que se podía romper el ciclo al encontrar el elemento buscado.

Por otro lado, se tomó decisión de diseño trabajar con distancias en números reales de doble precisión en el algoritmo interno de `combinarCiclos` por dos razones: i) como se busca minimizar la ganancia absoluta al reemplazar un par de lados antiguo por uno nuevo, los redondeos podían provocar que se deje de tomar en cuenta una mejor solución que solo difería de alguna anterior por unos cuantos decimales, y ii) se hicieron

distintas pruebas en las que este cambio representaba una mejora del más de 1 % en la calidad de las soluciones, por lo que se mantuvo en la versión final. A pesar de esto, tal como la especificación indica, la distancia entre cada par de ciudades al final del algoritmo (que es impresa por salida estándar e indicada en el comentario del archivo de salida) es calculada redondeando la distancia euclideana entre los puntos al entero más cercano, es decir, $distEntera = \lfloor distReal + 0{,}5 \rfloor$ para cada par de ciudades.

Por otro lado, como la especificación requiere que la primera ciudad del arreglo de entrada sea la primera y la última en ser visitada en el arreglo de lados, el ciclo que hace de salida a la función `divideAndConquer` es trasladado en la función `main` las posiciones adecuadas para que se cumpla la especificación. Por otro lado, para asegurar que la solución hallada sea una solución válida al arreglo de ciudades de entrada se implementó un verificador basado en búsqueda binaria que corre en tiempo $O(n \lg n)$, cuya eficiencia permitió que permanezca en el programa solución final, ya que no afectaba significativamente los tiempos de ejecución (de 2 a 6 segundos). El verificador de soluciones chequea que las siguiente condiciones se cumplan en el ciclo de salida:

- Su tamaño es igual al número de ciudades del arreglo de entrada.
- Comienza y termina en la primera ciudad del arreglo de entrada.
- Contiene en sus lados exactamente dos veces a cada ciudad del arreglo de entrada.
- Cada lado comparte una ciudad con los lados en posiciones adyacentes.

Con esto, se verifica que la solución presentada es totalmente operativa y produce soluciones válidas, ya que el verificador comprobó que en todas las instancias de prueba que se cumplan todas las condiciones mencionadas, sin arrojar error en ninguna.

Finalmente, también se implementan en el programa las funciones de `extraerDatos` y `escribirTSP`, cuyas funciones som, respectivamente, convertir los datos de un archivo en formato TSPLIB al arreglo de ciudades correspondiente, y escribir los resultados en un archivo de salida, mapeando cada ciudad de acuerdo a su posición en el arreglo inicial, respectivamente.

3. Resultados experimentales

La ejecución del programa se hizo en una máquina con procesador Intel(R) Celeron(R) CPU G1610 Dual-Core @ 2.60GHz con 2 GB de RAM, en el sistema operativo Linux Mint 19.3 Tricia 2-bit, usando Kotlin 1.5.0 y en el entorno de ejecución JRE 11.0.11+9; el tiempo tardado en correr las 73 instancias de prueba incluidas en `etsp_instancias.tar.xz` secuencialmente fue de aproximadamente 64 segundos. El Cuadro 1 muestra los resultados obtenidos por el programa, indicando el nombre de la instancia, la distancia del tour obtenido, y el porcentaje de desviación con respecto a la distancia de la solución más óptima conocida para esa instancia, disponible en ladocumentación de TSPLIB.

Como se puede ver en el Cuadro 1, la mayoría de las soluciones obtenidas oscilan entre un 15 % y 30 % de desviación con respecto a las soluciones óptimas. Se obtuvo que el promedio de las desviaciones entre todas las instancias ejecutadas fue de 21.35 %, por lo que se puede decir que la calidad de los resultados es aceptable tomando en cuenta la eficiencia del programa (recuérdese que el algoritmo para hallar la mejor solución por fuerza bruta es del orden de $O(n!)$).

En la tabla se pueden notar varias secciones con resultados resaltantes, entre estas están los grupos de instancias `eil`, en las que las soluciones obtenidas fueron de menos del 11 %, `fl`, cuyos resultados son los más altos de la tabla (particularmente `fl1577`), y especialmente las instancias `u2319`, `rd100` y `rd400`, con los resultados más cercanos a la solución óptima de todas las instancias.

Nombre de la instancia	Distancia programa	Desv. (%)	Nombre de la instancia	Distancia programa	Desv. (%)
berlin52	**8862**	17.5	pr1002	**324532**	25.28
bier127	**143302**	21.15	pr107	**50627**	14.27
brd14051	**565609**	20.48	pr124	**70175**	18.88
ch130	**7117**	16.48	pr136	**107580**	11.17
ch150	**7798**	19.45	pr144	**62714**	7.14
d1291	**65971**	29.86	pr152	**88813**	20.54
d15112	**1908216**	21.3	pr226	**98780**	22.91
d1655	**74394**	19.74	pr2392	**481393**	27.34
d198	**17645**	11.82	pr264	**58841**	19.75
d2103	**107305**	33.38	pr299	**59301**	23.05
d493	**41616**	18.9	pr439	**132617**	23.69
d657	**60905**	24.52	pr76	**126818**	17.25
eil101	**696**	10.65	rat195	**2716**	16.92
eil51	**465**	9.15	rat575	**7950**	17.38
eil76	**588**	9.29	rat783	**10489**	19.11
fl1400	**27664**	37.45	rat99	**1382**	14.12
fl1577	**31530**	41.71	rd100	**8560**	8.22
fl3795	**39721**	38.05	rd400	**18190**	19.04
fl417	**16269**	37.16	rl11849	**1179874**	27.78
fnl4461	**219626**	20.3	rl1304	**340420**	34.58
gil262	**2821**	18.63	rl1323	**359122**	32.91
kroA100	**25090**	17.89	rl1889	**427708**	35.12
kroA150	**31722**	19.6	rl5915	**743577**	31.48
kroA200	**35223**	19.94	rl5934	**736284**	32.41
kroB100	**24910**	12.51	st70	**813**	20.44
kroB150	**31551**	20.75	ts225	**140454**	10.91
kroC100	**24614**	18.63	tsp225	**4575**	16.74
kroD100	**24792**	16.43	u1060	**278654**	24.35
kroE100	**25214**	14.26	u1432	**176973**	15.69
lin105	**16979**	18.08	u159	**50066**	18.98
lin318	**51210**	21.84	u1817	**72815**	27.3
nrw1379	**67798**	19.7	u2152	**78373**	21.98
p654	**47389**	36.79	u2319	**244220**	4.25
pcb1173	**70275**	23.52	u574	**45229**	22.56
pcb3038	**167856**	21.91	u724	**51058**	21.83
pcb442	**59198**	16.58	vm1084	**302504**	26.41
			vm1748	**441875**	31.29

Cuadro 1: Resultados obtenidos de la ejecución del programa sobre las instancias de prueba contenidas en `estp_instancias.tar.xz`, y su desviación con respecto a la solución óptima.

Según el Cuadro 1, no parece haber alguna relación fuerte entre el tamaño del problema y la calidad de la solucoón que el algoritmo obtiene. El mejor resultado no es obtenido de la instancia más pequeña, y la instancia con más ciudades (`brd14051`) obtiene una solución cuya desviación ronda el promedio de desviaciones señalado anteriormente (20.48 %). Así, mediante el uso de un script de Python y usando la librería **Matplotlib** se extrajo una representación de algunas de las instancias resaltadas para analizar la relación entre la forma de las mismas y los resultados que se obtienen.

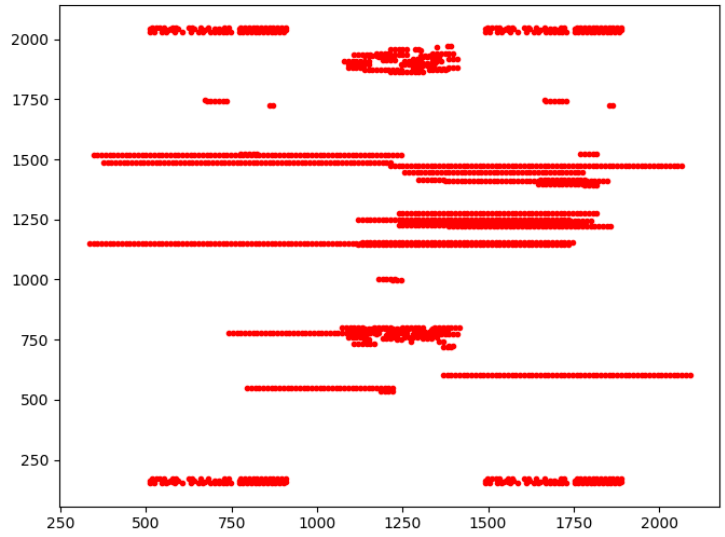

Figura 1: Representación de la instancia fl1577.

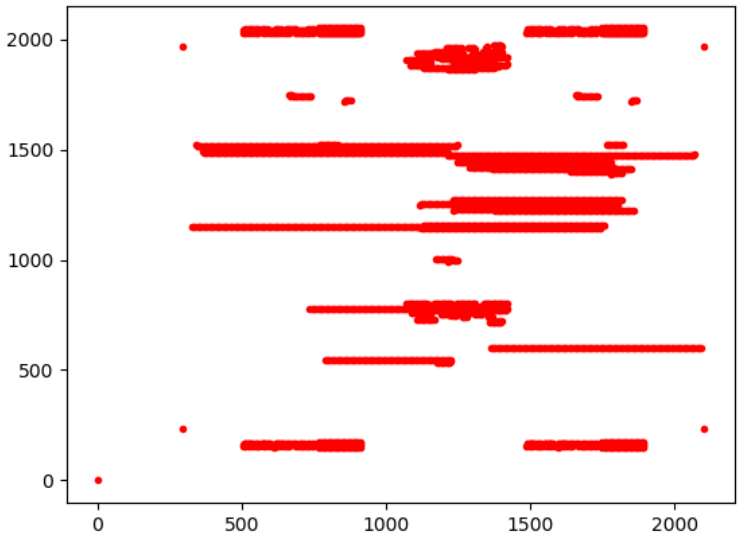

Figura 2: Representación de la instancia fl3795

De las Figuras 1 y 2, junto a los resultados del Cuadro 1 se puede inferir que el algoritmo aproximado usado da un resultado con calidad menor cuando el arreglo de entrada está conformado por ciudades no dispersas uniformemente en el plano, sino acumuladas con gran densidad en ciertas zonas, lo cual concuerda con la idea intuitiva del algoritmo, ya que siempre se busca obtener las particiones más equilibradas posibles, y si la distribución inicial de las ciudades en el plano no es uniforme en él las particiones que se obtienen con mucha probabilidad estarán desbalanceadas, produciendo peores ciclos que acumulan más distancia conforme se combinan de a pares.

Luego, se puede ver de instancias como `u2319` y `rd100`, la cuales conforman el mejor y el tercer mejor resultado de todos los casos de prueba, respectivamente, que como se espera, son las instancias distribuidas de

forma más equilibrada en el plano las que logran obtener mejores resultados. De hecho, se nota en la Figura 3 que casi todas las ciudades están separadas entre sí aproximadamente por la misma distancia, y que su distribución en el plano es casi perfectamenente balanceada, lo cual explica la desviación de la implementación de tan solo un 5 % aproximadamente.

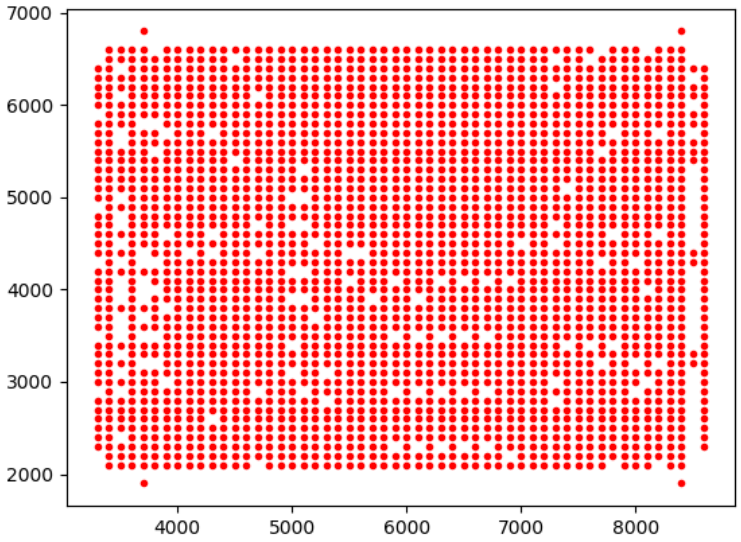

Figura 3: Representación de la instancia u2319.

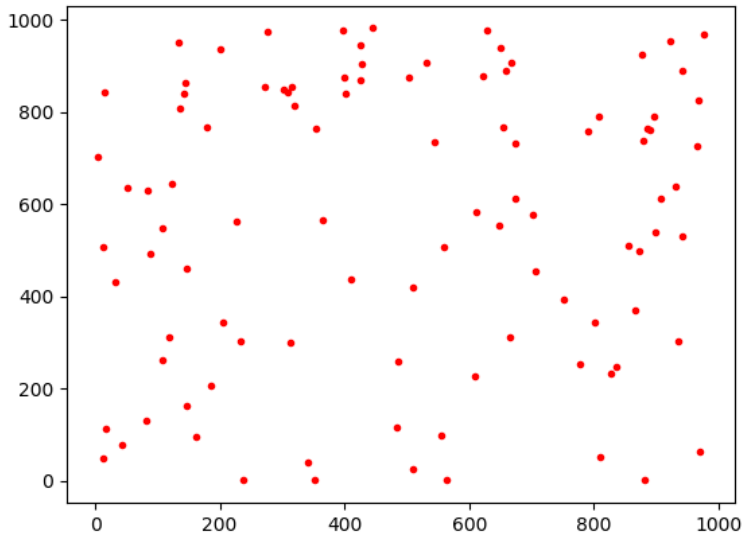

Figura 4: Representación de la instancia rd100

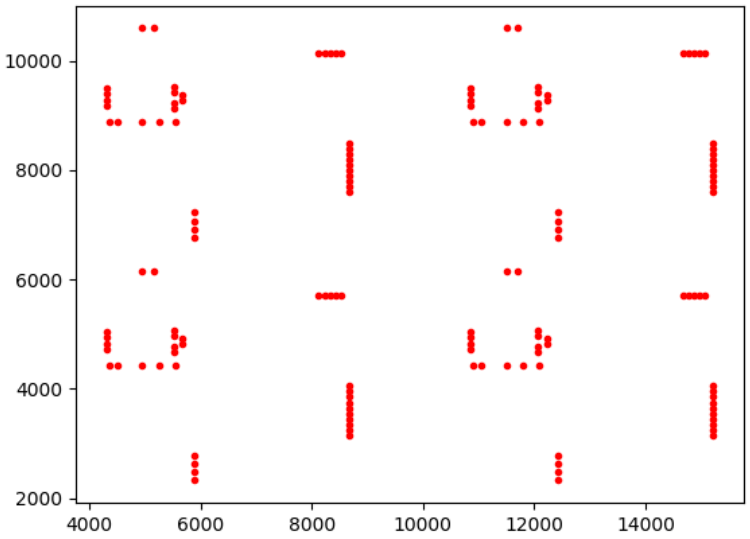

Figura 5: Representación de la instancia pr144

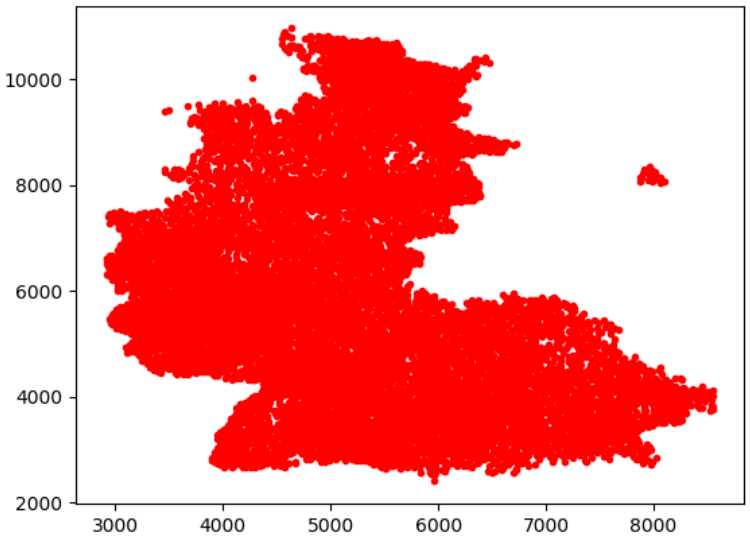

Figura 6: Representación de la instancia brd14051

De último, hay casos de instancias como la que se muestra en la Figura 5 que aunque no están uniformemente dispersas en el plano, también están equilibradas de forma aproximadamente simétrica, lo cual hace que los resultados que se obtienen no sean tan alejados de la distancia óptima que se busca. U, otro ejemplo, la instancia de la Figura 6 (la más grande de los casos de prueba), la cual está densamente distribuida en el centro del plano y tiene un pequeño cúmulo de ciudades a la derecha, y da un resultado no muy alejado del promedio. Así, se puede decir que más que por el tamaño del arreglo, los resultados del algoritmo dependerán más de la distribución específica de las ciudades de la entrada.

4. Conclusiones

Este algoritmo para resolver el *problema del agente viajero*, el proceso de implementarlo y diseñar la solución, y los resultados obtenidos a partir de él llevan a concluir que:

- La técnica de *Divide-and-conquer* para el diseño de algoritmos puede proporcionar formas interesantes de acercarse a problemas conocidos.
- Cuando se hace *Divide-and-conquer* se debe tener especial cuidado en cómo se divide el problema, y si la forma de combinar la soluciones de los subproblemas más pequeños puede llevar a la solución más óptima. Aunque pueden parecer que algoritmos como `obtenerPuntosRectangulo` u `obtenerPuntoDeCorte` sobran, es exactamente de la forma en que se particiona el arreglo de entrada en subproblemas cada vez más pequeño de lo que depende la calidad de las soluciones encontradas.
- A pesar de no hallar los mejores resultados, los algoritmos heurísticos pueden proporcionar soluciones a ciertos problemas con una calidad razonablemente buena en la mayoría de los casos si son correctamente implementados, y en un tiempo mucho menor al que se tardaría encontrando la solución más óptima posible.
- Los algoritmos de ordenamiento tienen un alcance que va más allá de ordenar secuencias de números reales o enteros, y manejar y entender cómo estos funcionan permite que se puedan implementar como parte de soluciones en las que se necesitan ordenar datos con otra forma distinta a la usual. En el proyecto se usan para ordenar pares con respecto a un eje dado (*Introsort*, *Insertion Sort*), y para permutar una secuencia de pares de tal forma que los adyacentes compartiesen una ciudad (basado en *Selection Sort*), lo cual el principio no parece tener tanta relación con los problemas de ordenamiento, sin embargo, en general, estos algoritmos de ordenamiento se podrán implementar siempre y cuando los elementos a ordenar sean comparables.
- Es importante escoger bien las estructuras a utilizar para representar cualquier conjunto de datos con el que se trabajará en el proyecto. Cada una se desempeña mejor para ciertos propósitos y de ello dependerá la comodidad con la que se manejen las implementaciones y la eficiencia del programa final.
- La limitación a solamente poder usar la estructura **Array** para guardar muchos datos agrupados enseña que gran parte de las veces se puede prescindir del uso de listas, cuando en todo momento se sabe cuántos elementos se quieren guardar, o cuando se puede hallar esta cantidad antes de inicializar el arreglo.
- Pequeños detalles en las implementaciones pueden significar una mejora notoria en (i) la eficiencia del programa, (ii) la eficacia del programa y (iii) la legibilidad y simpleza del código fuente.
- Muchas veces implementar instrucciones de alto nivel presentes en un pseudocódigo requiere de pensar con detenimiento y tomar decisiones con respecto a cómo hacerlas para que hagan lo que se especifica y en un lapso de tiempo razonable.