Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI2692 - Laboratorio de Algoritmos y Estructuras II
Trimestre abril-julio 2023

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

1. Introducción

El objetivo de este laboratorio es el de aplicar la técnica de divide-and-conquer en la solución de un problema de optimización. En específico, se quiere resolver el **problema del agente viajero**, en inglés *traveling salesman problem* (TSP).

2. El problema del agente viajero

El problema del agente viajero, que llamaremos TSP por sus siglas en inglés, es uno de los problemas de optimización más ampliamente estudiado. El TSP consiste en que dado un conjunto de ciudades, se quiere encontrar un tour que comience y finalice en cualquier ciudad, tal que solo es posible atravesar cada ciudad una sola vez, y la distancia recorrida debe ser la mínima posible. Hay varios tipos de problemas TSP. En este proyecto se va a trabajar en la versión de TSP, en donde las ciudades están dadas como puntos en el plano cartesiano. Es decir, para cada ciudad se va indicar las coordenadas en donde está ubicada. A esta versión del TSP se le conoce como **TSP euleriano**, ya que la distancias entre las ciudades corresponde a las distancias entre los puntos en el plano. La Figura 1 muestra una solución de una instancia de TSP con ocho ciudades y en donde cada ciudad es un punto en el plano. Usando la fórmula de distancia [1] de TSPLIB [2] [2] para este tipo de problemas, que se denominan en TSPLIB como **EUC_2D**, se tiene que la distancia de tour es 31. Como en este tipo de TSP, la distancia desde una ciudad A hasta una ciudad B, es igual a la distancia desde B hasta A, entonces eso hace que el problema de **TSP euleriano** sea de la categoría de los **TSP simétricos**.

3. Preliminares

Se tiene que en este contexto una **ciudad** es un punto el plano cartesiano. Las ciudades se representan con un par que corresponden a las coordenadas x y y en el plano cartesiano. Por ejemplo, las ciudades de la Figura 1 corresponden a los puntos $(1, 1)$, $(0, 9)$, $(3, 8)$, $(4, 0)$, $(6, 6)$, $(10, 7)$, $(5, 1)$ y $(7, 3)$.

Llamamos **lado** al recorrido desde una ciudad A hasta una ciudad B. Denotamos un lado con un par $(\!| \ a, b \ |\!)$ que contiene a dos ciudades a y b. Los lados de la Figura 1 son $(\!| \ (1, 1), (0, 9) \ |\!)$, $(\!| \ (1, 1), (4, 0) \ |\!)$, $(\!| \ (4, 0), (5, 1) \ |\!)$, $(\!| \ (5, 1), (7, 3) \ |\!)$, $(\!| \ (7, 3), (10, 7) \ |\!)$,

[1] La función `nint` corresponde al método `roundToInt` de Kotlin.
[2] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf`

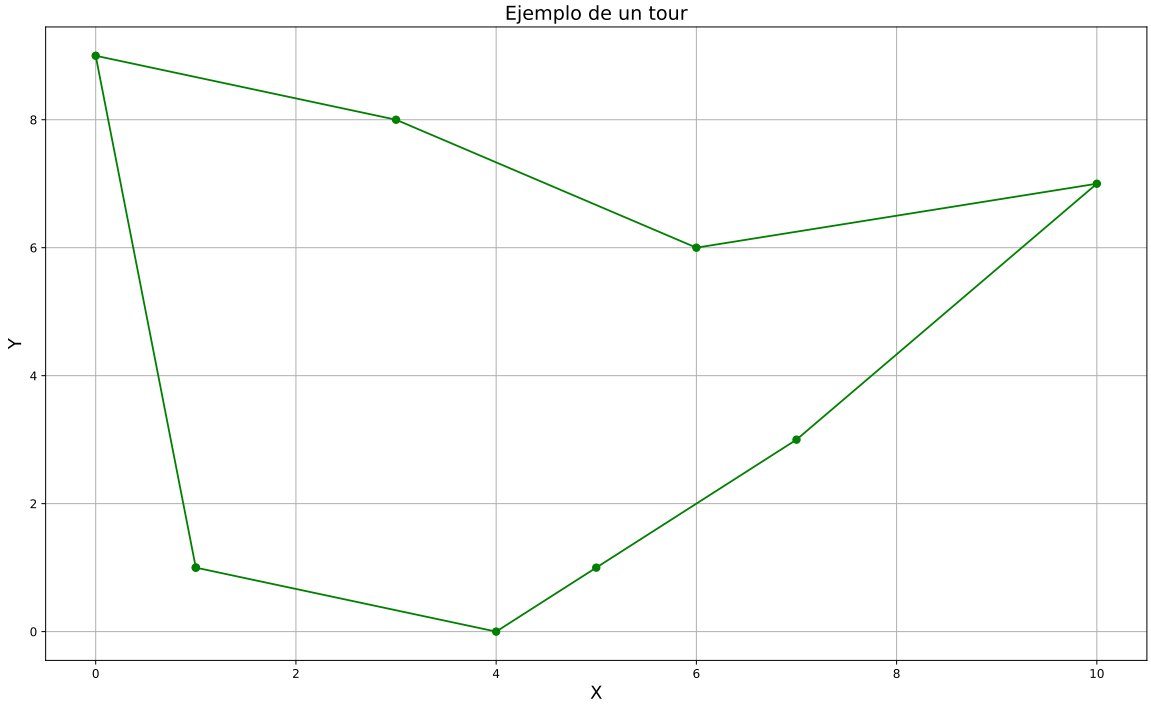


Figura 1: Tour que es solución de una instancia TSP euleriano de ocho ciudades. Los puntos en donde encuentran las ciudades son: $(1,1)$, $(0,9)$, $(3,8)$, $(4,0)$, $(6,6)$, $(10,7)$, $(5,1)$ y $(7,3)$

$\langle\!| (10,7),(6,6) |\!\rangle$, $\langle\!| (6,6),(3,8) |\!\rangle$, $\langle\!| (0,9),(3,8) |\!\rangle$ y $\langle\!| (1,1),(0,9) |\!\rangle$. Es indiferente el orden en que estén las ciudades en los lados.

Se define un **tour** como una secuencia de ciudades, que comienza y termina en una misma ciudad. La Figura 1 muestra un tour con ocho ciudades. El tour lo podemos representar como un arreglo, donde la posición en el arreglo corresponde al orden en que la ciudad es visitada. Un tour de la figura 1 es $[(1,1),(4,0),(5,1),(7,3),(10,7),(6,6),(3,8),(0,9),(1,1)]$. También se puede expresar un tour como un arreglo con una secuencia de lados. Por ejemplo, en la Figura 1 el tour se puede escribir como $[\langle\!| (1,1),(4,0) |\!\rangle, \langle\!| (5,1),(4,0) |\!\rangle, \langle\!| (5,1),(7,3) |\!\rangle, \langle\!| (10,7),(7,3) |\!\rangle, \langle\!| (10,7),(6,6) |\!\rangle, \langle\!| (6,6),(3,8) |\!\rangle, \langle\!| (3,8),(0,9) |\!\rangle, \langle\!| (0,9),(1,1) |\!\rangle]$. A un tour de ciudades, compuesto con puntos o lados, lo llamaremos **ciclo**.

Para este proyecto se va a tener como cierto, que es posible **un ciclo con una ciudad**. La interpretación es que es un tour comienza y finaliza en la misma ciudad. Por ejemplo, sea la ciudad $(1,1)$, entonces $[\langle\!| (1,1),(1,1) |\!\rangle]$ es un ciclo válido con una sola ciudad y un solo lado. La Figura 2 muestra la representación gráfica de un ciclo con una ciudad.

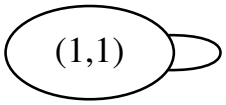
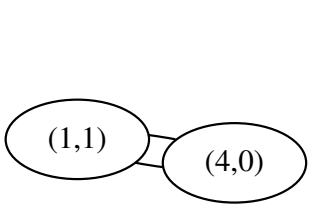


Figura 2: Ejemplo de un ciclo con la ciudad $(1,1)$.

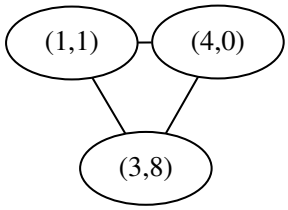
La librería TSPLIB [3] es una librería que contiene instancias de las diferentes variantes del TSP, incluidas instancias para TSP euleriano simétrico [4]. Las mejores soluciones conocidas para las instancias de TSP euleriano simétrico en TSPLIB, se encuentran publicadas [5]. Las instancias de TSPLIB y los archivos de las soluciones de las instancias, tienen un formato específico. El cálculo de la distancia entre dos ciudades de una instancia de TSP euleriano simétrico, está definida en la documentación de TSPLIB [6].

4. Algoritmo divide-and-conquer para TSP

El Algoritmo 1 resuelve el TSP euleriano simétrico, haciendo uso de la técnica divide-and-conquer. La entrada del Algoritmo 1 es un arreglo con las ciudades. La salida es un ciclo, representado como un arreglo de ciudades. La solución debe ser un ciclo que sea una solución válida del TSP. Desde la línea tres hasta la línea diez del Algoritmo 1, la idea es que si la entrada tiene menos de cuatro ciudades, entonces se construye un ciclo con algoritmos especializados. La Figura 3a muestra un ciclo con dos ciudades, que puede ser representado como $[(\!|\ (1,1),(4,0)\ |\!),(\!|\ (1,1),(4,0)\ |\!)]$. La Figura 3b tiene un ciclo de tres ciudades, que puede ser representado como $[(\!|\ (1,1),(4,0)\ |\!),(\!|\ (1,1),(3,8)\ |\!),(\!|\ (4,0),(3,8)\ |\!)]$. En caso de que se tengan como entrada cuatro o más ciudades, entonces en la línea 12 se divide el arreglo de ciudades en dos particiones. En la línea 13 y 14 se hace la llamada recursiva a `divideAndConquerTSP` con las particiones. Una vez obtenidos los ciclos por el enfoque de divide-and-conquer, estos ciclos se combinan en la línea 15 para obtener un solo ciclo.



(a) Ejemplo de un ciclo con dos ciudades.



(b) Ejemplo de un ciclo con tres ciudades.

Figura 3: Ejemplo de ciclos simples de dos y tres ciudades.

El Algoritmo 2 divide un arreglo de ciudades, en dos particiones de ciudades. La idea es dividir en dos el espacio rectangular generado por los puntos de entrada, a través de un punto de corte. Luego se obtienen los puntos de cada partición y se retorna un par con dos arreglos que contienen las ciudades que se encuentran incluidas en cada partición. La Figura 4 muestra el rectángulo generado por el grupo de ciudades de la Figura 1. El lado más largo del rectángulo es el que está paralelo al eje X, por lo que ese será el lado usado para obtener las particiones. El punto de corte viene dado por el punto $(4, 0)$, usando la Función `obtenerPuntoDeCorte`. Luego se traza una recta perpendicular al eje X desde el punto de corte. Esto da como resultado que el

[3] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`
[4] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/`
[5] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html`
[6] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf`

Algoritmo 1: divideAndConquerTSP

Entrada: Un arreglo P con coordenadas de ciudades
Salida : Un ciclo C solución válida del TSP

```
1  inicio
2      n ← P.length ;
3      si n = 0 entonces
4          devolver [] ;
5      si no, si n = 1 entonces
6          devolver cicloUnaCiudad(P) ;
7      si no, si n = 2 entonces
8          devolver cicloDosCiudades(P) ;
9      si no, si n = 3 entonces
10         devolver cicloTresCiudades(P) ;
11     en otro caso
12         (pderecha, pizquierda) = obtenerParticiones(P) ;
13         c1 ← divideAndConquerTSP(pderecha) ;
14         c2 ← divideAndConquerTSP(pizquierda) ;
15         devolver combinarCiclos(c1, c2) ;
```

rectángulo original sea divido en dos rectángulos. Cada uno de esos rectángulos corresponde a una partición de las ciudades originales. Se puede ver en la Figura 4 el rectángulo de la izquierda contiene una partición con las ciudades $[(1,1),(0,9),(3,8),(4,0)]$ y el rectángulo de la derecha a la partición $[(6,6),(10,7),(5,1),(7,3)]$. No siempre es posible obtener una partición balanceada de las ciudades. Por ello, la línea 18 del Algoritmo 2 se escoge como punto de corte, un punto medio del lado más largo del rectángulo que contiene las ciudades. La Función `obtenerPuntoDeCorteMitad` obtiene ese punto de corte, de esta manera se quiere obtener dos particiones y que una de ellas tenga al menos un punto.

El Algoritmo 3 presenta como se realiza la combinación de dos ciclos. La idea es revisar cada combinación de pares de lados de los ciclos y sustituirlos por dos nuevos lados que unan a los ciclos. Se quiere encontrar la combinación de lados a agregar y a eliminar que proporcione la generación de un nuevo ciclo con la menor distancia posible. La Figura 5 presenta dos particiones. Al aplicar el Algoritmo 3 a esas dos particiones, se obtiene que se deben eliminar los lados $\langle\!| \ (3,8),(4,0) \ |\!\rangle$ y $\langle\!| \ (5,1),(6,6) \ |\!\rangle$, y se deben agregar los lados $\langle\!| \ (3,8),(6,6) \ |\!\rangle$ y $\langle\!| \ (4,0),(5,1) \ |\!\rangle$, tal como se muestra en la Figura 6. Una vez finalizado el Algoritmo 3 se obtiene el ciclo de la Figura 1, el cual es una solución válida del TSP.

5. Mejorando una solución de TSP con búsqueda local

Una vez que se obtiene una solución factible para el TSP, es posible mejorar esta por medio de algoritmos de búsqueda local especializados. Uno de los algoritmos para mejorar soluciones

Algoritmo 2: obtenerParticiones

Entrada: Un arreglo P con coordenadas de ciudades
Salida : Un par con dos arreglos con coordenadas de ciudades

```
1  inicio
2      rectangulo ← Obtenga las coordenadas del rectángulo que contiene a las ciudades
         en P ;
3      (X_dim, Y_dim) ← Obtener las dimensiones de los lados que son paralelos al eje X y
         al eje Y de rectangulo ;
4      si X_dim > Y_dim entonces ejeDeCorte ← X ;
5      en otro caso ejeDeCorte ← Y ;
6      (x_c, y_c) ← obtenerPuntoDeCorte(P, ejeDeCorte) ;
7      (rectanguloIzq, rectanguloDer) ← aplicarCorte(P, ejeDeCorte, (x_c, y_c),
         rectangulo) ;
8      particionIzq ← obtenerPuntosRectangulo(P, rectanguloIzq) ;
9      particionDer ← obtenerPuntosRectangulo(P, rectanguloDer) ;
10     si (particionIzq.length = 0 ∧ particionDer.length > 3) ∨ (particionIzq.length >
         3 ∧ particionDer.length = 0) entonces
11         si ejeDeCorte = X entonces ejeDeCorte ← Y ;
12         en otro caso ejeDeCorte ← X ;
13         (x_c, y_c) ← obtenerPuntoDeCorte(P, eje) ;
14         (rectanguloIzq, rectanguloDer) ← aplicarCorte(P, ejeDeCorte,
             (x_c, y_c), rectangulo) ;
15         particionIzq ← obtenerPuntosRectangulo(P, rectanguloIzq) ;
16         particionDer ← obtenerPuntosRectangulo(P, rectanguloDer) ;
17         si (particionIzq.length = 0 ∧ particionDer.length > 3) ∨ (particionIzq.length >
             3 ∧ particionDer.length = 0) entonces
18             (x_c, y_c) ← obtenerPuntoDeCorteMitad(rectangulo, eje) ;
19             (rectanguloIzq, rectanguloDer) ← aplicarCorte(P, ejeDeCorte,
                 (x_c, y_c), rectangulo) ;
20             particionIzq ← obtenerPuntosRectangulo(P, rectanguloIzq) ;
21             particionDer ← obtenerPuntosRectangulo(P, rectanguloDer) ;
22     devolver (particionIzq, particionDer)
```

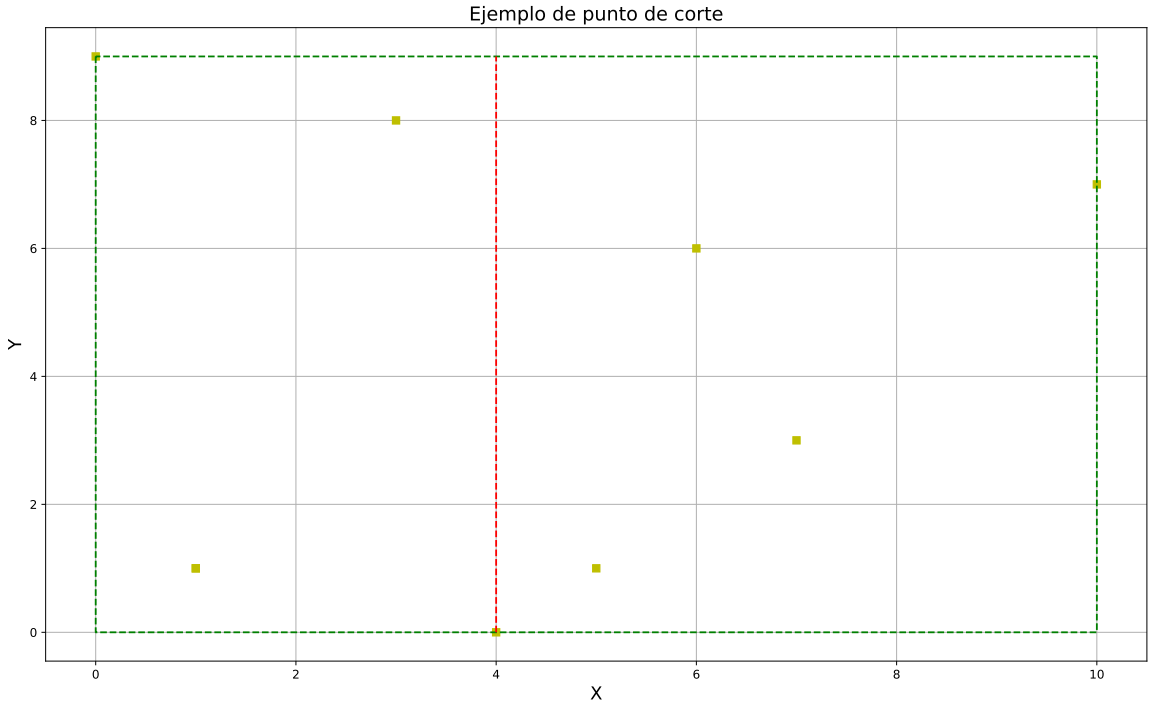


Figura 4: Ejemplo de dos particiones de la división del espacio rectangular generado por los puntos de la Figura 1. En verde el espacio rectangular, generado por las ciudades. El punto de corte es la ciudad $(4, 0)$ y la línea roja es el corte realizado.

Función obtenerPuntoDeCorte(P, *ejeDeCorte*)

1 **inicio**
2 $n \leftarrow P.length$;
3 $pos \leftarrow \lceil n/2 \rceil - 1$;
4 **si** $ejeDeCorte = X$ **entonces**
5 Ordena las ciudades de P, con respecto a la coordenada X. Si dos elementos tienen igual valor de la coordenada X, ordene por la coordenada Y
6 **en otro caso**
7 Ordena las ciudades de P, con respecto a la coordenada Y. Si dos elementos tienen igual valor de la coordenada X, ordene por la coordenada X
8 **devolver** $P[pos]$

Función obtenerPuntoDeCorteMitad(*rectangulo*, *eje*)

```
1  inicio
2      xmin ← Obtener el menor valor en el eje X de los puntos del rectangulo ;
3      ymin ← Obtener el menor valor en el eje Y de los puntos del rectangulo ;
4      si eje = X entonces
5          xdim ← Obtener el valor del lado del rectangulo que es paralelo al eje X ;
6          puntoDeCorte ← (xmin + xdim/2, ymin)
7      en otro caso
8          ydim ← Obtener el valor del lado del rectangulo que es paralelo al eje Y ;
9          puntoDeCorte ← (xmin, ymin + ydim/2)
10     devolver puntoDeCorte ;
```

Función aplicarCorte(*ejeDeCorte*, (x_c, y_c), *rectangulo*)

```
1  inicio
2      (rectanguloIzq, rectanguloDer) ← Se divide en dos el rectangulo que contiene a
         todas las ciudades de P, trazando una recta perpendicular al ejeDeCorte. Si
         ejeDeCorte = X se traza una recta perpendicular desde x_c, y si ejeDeCorte = Y
         desde y_c. Obtenga los dos rectángulos resultantes de trazar la recta
         perpendicular ;
3      devolver (rectanguloIzq, rectanguloDer) ;
```

Función obtenterPuntosRectangulo(P, *rectangulo*)

```
1  inicio
2      particion ← Se obtienen las ciudades de P contenidos en rectangulo y se agregan
         en un arreglo ;
3      devolver particion ;
```

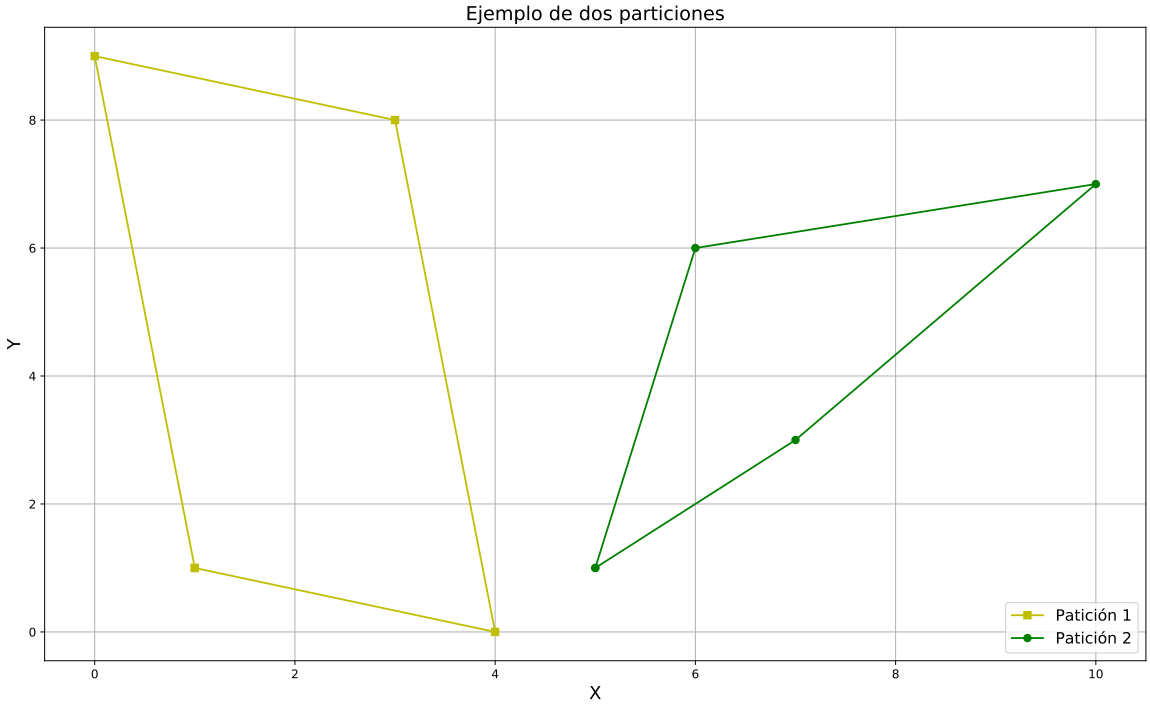


Figura 5: Se tiene dos ciclos con las ciudades de la Figura 1

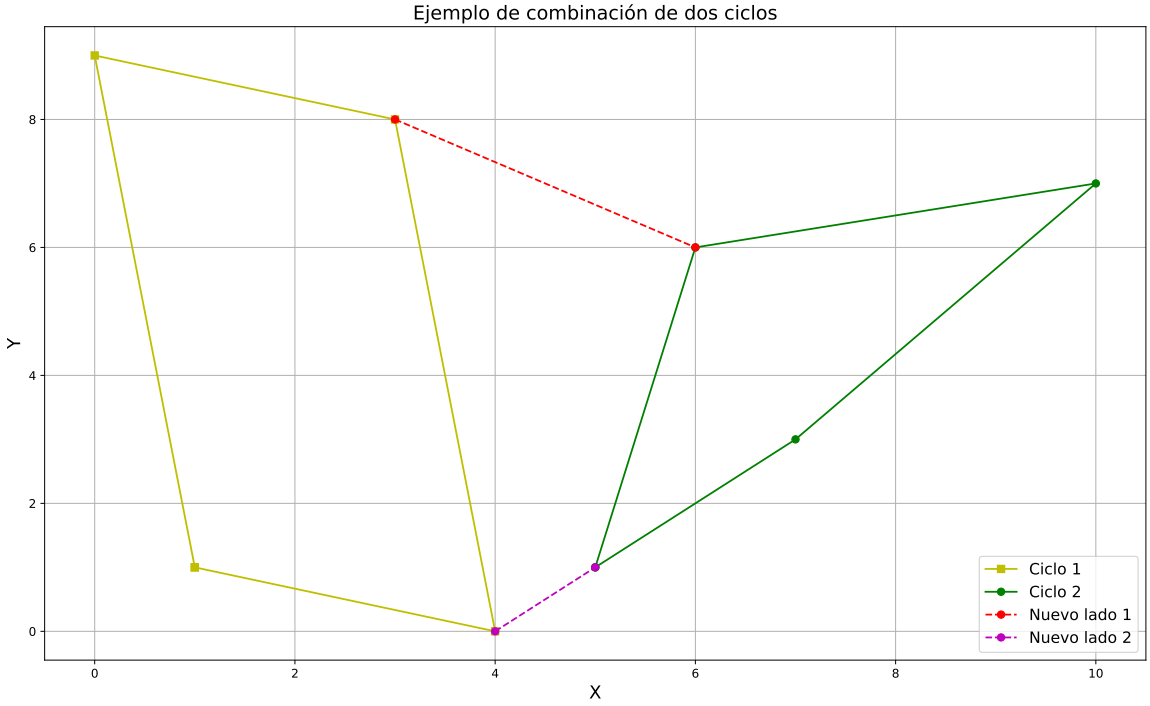


Figura 6: Se conectan las particiones de la Figura 5. Se agregan los dos lados en lineas discontinuas y se eliminan los lados $(\!| (3,8),(4,0) |\!)$ y $(\!| (5,1),(6,6) |\!)$

Algoritmo 3: combinarCiclos($Ciclo_1$, $Ciclo_2$)

Entrada: Dos secuencias de lados $Ciclo_1$ y $Ciclo_2$ que representan dos tours de ciudades

Salida : Un tour $Ciclo_3$ que es la unión de $Ciclo_1$ y $Ciclo_2$

```
1  inicio
2      si Ciclo1.length = 0 entonces devolver Ciclo2;
3      si Ciclo2.length = 0 entonces devolver Ciclo1;
4      minG ← ínf ;
5      para cada lado (a, b) ∈ Ciclo1 hacer
6          dOLD1 ← distancia(a, b) ;
7          para cada lado (c, d) ∈ Ciclo2 hacer
8              dOLD2 ← distancia(c, d) ;
9              dNEW1 ← distancia(a, c) ;
10             dNEW2 ← distancia(b, d) ;
11             dNEW3 ← distancia(a, d) ;
12             dNEW4 ← distancia(b, c) ;
13             g1 ← distanciaGanada(dOLD1, dOLD2, dNEW1, dNEW2) ;
14             g2 ← distanciaGanada(dOLD1, dOLD2, dNEW3, dNEW4) ;
15             ganancia ← mín(g1, g2) ;
16             si ganancia < minG entonces
17                 minG ← ganancia ;
18                 si g1 < g2 entonces
19                     ladosAgregarC1 ← (a, c) ;
20                     ladosAgregarC2 ← (b, d) ;
21                 en otro caso
22                     ladosAgregarC1 ← (a, d) ;
23                     ladosAgregarC2 ← (b, c) ;
24                 ladosEliminarC1 ← (a, b) ;
25                 ladosEliminarC2 ← (c, d) ;
26     Remover de Ciclo1 los lados de ladosEliminarC1 ;
27     Remover de Ciclo2 los lados de ladosEliminarC2 ;
28     Agregar a Ciclo3 los lados de ladosAgregarC1, los lados de ladosAgregarC2, los
        lados de Ciclo1 y los lados de Ciclo2 ;
29     devolver Ciclo3
```

Función distanciaGanada($dOLD1, dOLD2, dNEW1, dNEW2$)

```
1  devolver (dNEW1 + dNEW2) − (dOLD1 + dOLD2)
```

del TSP más utilizados es el operador 2-`opt`. El operador 2-`opt` consiste en intercambiar dos lados de un tour, con la esperanza de que ese cambio produzca un tour de menor distancia. Por ejemplo, en la Figura 7, se tiene un tour que es solución de TSP, y al aplicar el operador 2-`opt` se intercambian los lados $(\!| a, b |\!)$ y $(\!| c, d |\!)$, por los lados $(\!| a, d |\!)$ y $(\!| c, b |\!)$, obteniendo así un nuevo tour. El operador 2-`opt` ha probado ser eficaz para mejorar soluciones del TSP [1]. Con el operador 2-`opt` se puede realizar un algoritmo de búsqueda local, que intercambie los lados de un tour solución del TSP, para encontrar un tour de menor distancia. En este proyecto, la solución del TSP que genera el Algoritmo 1, se va a mejorar con un algoritmo de búsqueda local basado en el operador 2-`opt`. En específico, debe implementar el algoritmo de búsqueda local que se presenta en [3]. Este algoritmo debe ser implementado como una función, llamada `busquedaLocalCon2OPT`, que recibe como entrada un tour del TSP y retorna un tour del TSP. El objetivo es realizar un algoritmo que genera una solución para el TSP, y que luego la mejora aplicando `busquedaLocalCon2OPT`. En el Algoritmo 4 se muestra el resolvedor del TSP que debe implementar.

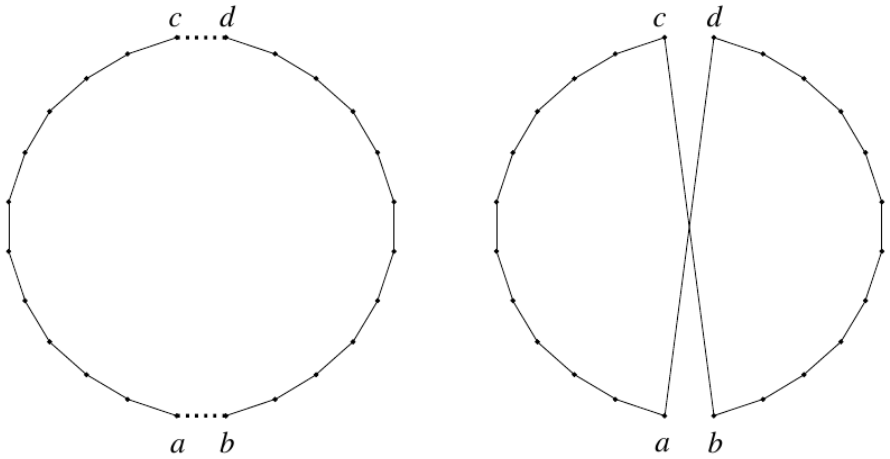


Figura 7: Ejemplo del operador 2-opt en donde se intercambian dos lados de un tour. Fuente: [1].

Algoritmo 4: divideAndConquerAndLocalSearchTSP

Entrada: Un arreglo P con coordenadas de ciudades
Salida : Un ciclo C solución válida del TSP

```
1 inicio
2     c_1 ← divideAndConquerTSP(P) ;
3     devolver busquedaLocalCon2OPT(c_1) ;
```

6. Actividades a realizar

6.1. Verificador de soluciones del TSP

Se quiere que realice un programa en Kotlin, llamado `VerificadorTSP.kt` que verifica si una solución del TSP dada, corresponde a una solución válida de una instancia especifica del TSP. Este programa se debe ejecutar con un un *script* de Bash llamado `runVerificador.sh`, que se llama de la siguiente manera:

```
>./runVerificador.sh instancia solución
```

Donde:

`instancia`**:** Es un archivo con una instancia de TSP en formato TSPLIB.

`solución`**:** Es un archivo con la solución de la instancia TSP en formato TSPLIB.

El programa debe revisar la solución de la instancia TSP, y reportar todos los problemas que pueda tener esa solución. Posibles problemas son, la falta de visita de una ciudad, visitar más de una vez una ciudad, calcular mal la distancia tour, entre otras. Si por el contrario, la solución es correcta, el programa lo debe indicar.

6.2. Resolvedor del TSP

Debe realizar un programa en Kotlin llamado `DCLSTSP.kt` que implemente el Algoritmo 4. La ejecución de `DCLSTSP.kt` se debe hacer por medio de un *script* de Bash llamado `runDCLSTSP.sh`, el cual se ejecuta con la siguiente línea de comando:

```
>./runDCLSTSP.sh archivo_entrada archivo_salida
```

Donde:

`archivo_entrada`**:** Archivo con una instancia de TSP en formato TSPLIB.

`archivo_salida`**:** Nombre del archivo que se va a generar, con la solución de la instancia TSP en formato TSPLIB.

Se tiene que en el formato de TSPLIB para los archivos que contienen soluciones del TSP, un tour **siempre debe comenzar en la ciudad identificada con el número 1**. Debe agregar en la segunda línea del archivo de salida, un comentario indicando la distancia del tour solución con el siguiente formato: `COMMENT : Length X`, donde X es la longitud del tour. Por ejemplo, dado el tour de la Figura 1, se debería agregar el siguiente comentario:

```
COMMENT : Length 31
```

La salida `DCLSTSP.kt` es por la salida estándar y debe mostrar lo siguiente:

- El nombre de la instancia TSP a resolver.
- La distancia de la solución obtenida por el algoritmo de divide-and-conquer (tour de la línea 2 del Algoritmo 4).

- La distancia del tour que es la solución final del algoritmo.

Se le proporcionará de los siguientes archivos:

`minipoints.txt`: Instancia TSP con las ciudades de la Figura 1 en formato TSPLIB.

`minipoints.out`: Contiene el tour solución que se muestra en la Figura 1, en formato TSPLIB.

`etsp_instancias.tar.xz`: Instancias de TSPLIB que deben ser ejecutadas por `DCLSTSP.kt`.

La implementación de `DCLSTSP.kt` debe ser **razonablemente eficiente**. Es decir, se debe evitar algoritmos que aumenten innecesariamente la complejidad en tiempo de las operaciones a implementar. Por ejemplo, cuando se aplica el ordenamiento de las ciudades en la Función `obtenerPuntoDeCorte`, debe hacerse por medio de una versión modificada de Quicksort o Mergesort.

Para la implementación solo estará permitido usar las siguiente estructuras de datos de Kotlin: *arreglos*, *Pair* y *Triple*. Todo algoritmo de ordenamiento a ser usado debe ser implementado por usted.

6.3. Estudio experimental

Debe realizar un informe, **en formato PDF**, con la siguiente estructura:

Portada: Contiene el título del proyecto, la información institucional y los datos de los autores del trabajo.

Diseño de la solución: Descripción de la diseño de su solución, las estructuras de datos usadas, los detalles de implementación, entre otros.

Resultados experimentales: Presentación de los resultados obtenidos sobre las instancias TSP y el análisis de los mismos.

Lecciones aprendidas: Las lecciones aprendidas durante la realización del proyecto.

Sobre la sección *diseño de la solución*. Debe indicar las estructuras de datos usadas para representar los elementos más importantes del proyecto, tal como `ciudades`, `lados`, `ciclos`, `particiones` y `rectángulos` entre otros. Se quiere que describa los elementos del diseño de la solución introducidos por usted. Debe explicar detalles de implementación que considere relevantes. También debe describir las mayores dificultades al realizar el proyecto. En esta sección se debe indicar si su entrega está totalmente operativa o si por el contrario tiene problemas conocidos. Finalmente, es libre de agregar cualquier otro aspecto que considere relevante con la solución del proyecto.

En cuanto a la sección *resultados experimentales*. Se deben presentar los resultados de las instancias TSP incluidas en la carpeta `etsp_instancias`. Se debe indicar la plataforma donde se realizaron los experimentos: sistema de operación, modelo de CPU, cantidad de memoria RAM, versión del compilador de Kotlin y de la JVM. Debe presentar los resultados obtenidos dos tablas. La primera tabla contiene los siguientes elementos:

Nombre de la instancia	Distancia obtenida	% Desv. valor óptimo

Donde *Distancia obtenida*, es la distancia obtenida por el Algoritmo 4, mientras *% Desv. Valor Óptimo*, es el *Porcentaje de desviación con respecto al valor óptimo*. Este valor se obtiene con la fórmula $\frac{distanciaPrograma - distanciaOptima}{distanciaOptima} \cdot 100$, donde *distanciaPrograma* es el valor obtenido por un resolvedor, y *distanciaOptima* es la distancia óptima de la instancia que se reporta en TSPLIB.

El objetivo de la segunda tabla es mostrar los resultados consolidados. Esta tabla tiene las siguiente forma:

Nombre del resolvedor	Distancia total obtenida	% Desv. valor óptimo total
Divide-and-conquer		
Divide-and-conquer + 2-`opt`		

Donde *Divide-and-conquer* corresponde al resolvedor del Algoritmo 1 y *Divide-and-conquer + 2-*`opt` al Algoritmo 4. La columna *Distancia total obtenida* indica la suma total de las distancias de las soluciones del resolvedor. Finalmente, el *% Desv. valor óptimo total* es el % de desviación con respecto a la suma total de las distancias óptimas. El informe debe incluir un análisis de los resultados obtenidos en las dos tablas.

El código a entregar debe estar debidamente documentado y para todas las funciones debe indicar su descripción, las pre y post condiciones, y la explicación de los parámetros de entrada. Se debe seguir los lineamientos de la guía de estilo de Kotlin indicada en clase. Si un programa no compila o no se ejecuta, tiene cero como nota del proyecto. El código del proyecto debe estar contenido en cuatro archivos: `DCLSTSP.kt`, `runDCLSTSP.sh`, `VerificadorTSP.kt`, y `runVerificadorTSP.sh`. Opcionalmente puede proporcionar un archivo `Makefile` para compilar los códigos fuentes. El archivo `runDCLSTSP.sh` **solo debe ejecutar** correctamente el programa `DCLSTSP.kt`, no debe hacer ninguna verificación.

7. Condiciones de entrega

Debe entregar los códigos fuentes de su programa, el informe, y la declaración de autenticidad, en un archivo comprimido llamado `Proyecto1-X-Y.tar.xz` donde X y Y son los números de carné de los autores del proyecto. La entrega debe hacerse por medio de la plataforma Classroom, antes de las 11:50 pm del día domingo 18 de junio de 2023.

Referencias

[1] David S Johnson and Lyle A McGeoch. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, 1(1):215–310, 1997.

[2] Gerhard Reinelt. TSPLIB - A Traveling Salesman Problem Library. *INFORMS Journal on Computing*, 3(4):376–384, November 1991.

[3] Wikipedia contributors. 2-opt — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/2-opt`, 2023. Accedido: 01.06.2023.

Guillermo Palma / gvpalma@usb.ve / Junio de 2023