UNIVERSIDAD SIMÓN BOLÍVAR
VICERRECTORADO ACADÉMICO

DEPARTAMENTO DE COMPUTACIÓN Y TECNOLOGÍA DE LA INFORMACIÓN CI-2692 – LABORATORIO DE ALGORITMOS Y ESTRUCTURAS II

Informe Proyecto 1

Autores:

Franco Murillo - 1610782

Fredthery Castro - 1810783

Profesor:

Guillermo Palma

Sartenejas, febrero de 2024

Diseño

Se presenta el análisis del diseño que se usó para el programa que resuelve el problema de la ruta de máximo beneficio (PRMB).

El programa hace uso de las siguientes clases de Java:

- **java.io.BufferedReader,** para leer texto de una entrada de caracteres de manera eficiente.
- **java.io.File,** para representar una ruta de acceso de archivo o directorio.
- **java.io.FileReader,** para leer datos basados en caracteres de un archivo.
- **java.text.DecimalFormat,** para formatear números en cadenas con un formato específico.
- **java.text.DecimalFormatSymbols,** para definir los símbolos que se utilizan al formatear números con la clase DecimalFormat.

El programa recibe como entrada un archivo con la instancia, que representa la lista de ciudades. El archivo contiene en la primera línea, un número n, que representa el número de ciudades contenidas en el archivo. Luego, tiene n líneas con 3 números naturales separados por coma que representa cada ciudad. El primer número representa la coordenada X, el segundo número la coordenada Y, y el tercer número el beneficio de visitar dicha ciudad. Para trabajar de forma más cómoda con las ciudades, se decidió declarar un data class llamado *Ciudad*, que tiene 4 valores de tipo Int:

- El índice, que identifica la ciudad
- La coordenada X de la ciudad
- La coordenada Y de la ciudad
- El beneficio de visitar la ciudad

Así, cada ciudad se puede representar como una 4-tupla de enteros.

En el método *main* se asigna a una variable el archivo de entrada para poder trabajar con él. Además, se declara el objeto lector usando las clases *BufferedReader* y *FileReader*. Posteriormente se lee la primera línea del archivo de entrada y se almacena en una variable *n*, para luego crear una lista de tamaño n de elementos de tipo *Ciudad*. Después, se lee el resto de datos del archivo, almacenando cada ciudad en la lista de ciudades. Asimismo, se procede a establecer el formato que tendrá el beneficio obtenido, la cual es con cuatro decimales. Luego se hace la llamada a la función que resuelve el problema, haciendo uso de la técnica Divide-And-Conquer. Finalmente, se almacenan los índices de las ciudades de la ruta solución, junto con el beneficio de la ruta, y se imprimen los resultados.

A continuación, se presentan las funciones utilizadas en el programa:

- **distancia**: recibe dos ciudades (representadas como 4-tuplas), y calcula la distancia euclidiana entre las dos ciudades. Retorna la distancia obtenida como un Double.
- **ganancia**: recibe una ruta (representada como una lista de Ciudades), y calcula la ganancia total de recorrer las ciudades de la ruta. Retorna la ganancia como un Double.
- **partition y quicksort**: reciben una lista de ciudades y realizan su ordenamiento en función de la coordenada X de la ciudad.

- **divideAndConquerPRMB:** recibe una lista de ciudades y devuelve un par que contiene una lista de ciudades y un Double. Este par representa la ruta óptima (lista de ciudades) y la ganancia total (valor Double) obtenida al seguir esa ruta.

La función sigue un enfoque de divide y vencerás para resolver el problema. Si el tamaño de la lista de ciudades es menor o igual a 3, la función resuelve el problema de manera ad-hoc utilizando la función *resolverMiniPRMB*. Si el tamaño de la lista de ciudades es mayor que 3, la función divide la lista de ciudades en dos sublistas, *ciudadesIzquierda* y *ciudadesDerecha*, utilizando el índice de la mitad de la lista.

Luego, la función resuelve los subproblemas de manera recursiva llamando a sí misma con las sublistas antes mencionadas, y guarda las rutas óptimas y las ganancias totales obtenidas de estos subproblemas en *rutaIzquierda* y *rutaDerecha*, respectivamente.

Finalmente, la función combina las rutas óptimas y las ganancias totales de los subproblemas utilizando la función *combinarRutas* y devuelve el resultado.

En la siguiente página se presenta el pseudo código de la función divideAndConquerPRMB

```
-divideAndConquerPRMB

Función divideAndConquerPRMB(ciudades)
    Si el tamaño de ciudades es menor o igual a 3 entonces
        Retornar resolverMiniPRMB(ciudades)
    Fin Si

    // Dividir el problema en dos subproblemas
    mitad ← tamaño de ciudades / 2
    ciudadesIzquierda ← sublist de ciudades desde 0 hasta mitad
    ciudadesDerecha ← sublist de ciudades desde mitad hasta el final de ciudades

    // Resolver los subproblemas de manera recursiva
    rutaIzquierda ← divideAndConquerPRMB(ciudadesIzquierda)
    rutaDerecha ← divideAndConquerPRMB(ciudadesDerecha)

    // Combinar las soluciones de los subproblemas
    Retornar combinarRutas(rutaIzquierda, rutaDerecha)
Fin Función
```

- **resolverMiniPRMB**: toma una lista de ciudades y devuelve un par quecontiene una lista de ciudades y un valor doble. Este par representa la ruta óptima (lista de ciudades) y la ganancia total (valor Double) obtenida al seguir esa ruta.

La función comienza inicializando una variable solucion con una lista vacía de ciudades y una ganancia de 0.0.

Si solo hay una ciudad en la lista, la función devuelve esa ciudad como la ruta óptima y su beneficio como la ganancia total.

Si hay dos ciudades en la lista, la función calcula la ganancia entre las ciudades y compara esta ganancia con el beneficio de cada ciudad. Si la ganancia entre las ciudades es menor o igual al beneficio de la primera ciudad, la función devuelve la primera ciudad como la ruta óptima y su beneficio como la ganancia total. De lo contrario, devuelve ambas ciudades como la ruta óptima y la ganancia entre las ciudades como la ganancia total. Este proceso se repite de manera similar para la segunda ciudad.

Si hay tres ciudades en la lista, la función prueba todas las posibles rutas, incluyendo las que tienen menos ciudades, y elige la ruta con la mayor ganancia. Primero, evalúa cada ciudad por separado y guarda la ciudad con el mayor beneficio y su beneficio como la ganancia total. Luego, evalúa las rutas entre dos ciudades y guarda la ruta con la mayor ganancia y su ganancia como la ganancia total. Finalmente, evalúa las rutas entre las tres ciudades y guarda la ruta con la mayor ganancia y su ganancia como la ganancia total.

Al final, la función elige la ruta con la mayor ganancia entre las rutas individuales, las rutas entre dos ciudades y las rutas entre tres ciudades, y devuelve esta ruta y su ganancia como la solución.

En la siguiente página se presenta el pseudocódigo de resolverMiniPRMB.

resolverMiniPRMB (parte 1)

```
-resolverMiniPRMB
Función resolverMiniPRMB(ciudades)
    solucion ← Par vacío

    Si el tamaño de ciudades es igual a 1 entonces
        solucion ← Par con la primera ciudad y su beneficio
        Retornar solucion
    Fin Si

    Si el tamaño de ciudades es igual a 2 entonces
        gananciaEntreCiudades ← beneficio de la primera ciudad + beneficio de la segunda ciudad - distancia entre las dos ciudades

        Si el beneficio de la primera ciudad es mayor o igual al beneficio de la segunda ciudad entonces
            Si el beneficio de la primera ciudad es mayor o igual al segundo elemento de maxEntreDos entonces
                maxEntreDos ← Par con la primera ciudad y su beneficio
            Fin Si
        Sino
            Si el beneficio de la segunda ciudad es mayor o igual al segundo elemento de maxEntreDos entonces
                maxEntreDos ← Par con la segunda ciudad y su beneficio
            Fin Si
        Fin Si

        solucion ← el Par con mayor segundo elemento entre maxIndividual, maxEntreDos y maxEntreTres
        Retornar solucion
    Fin Si
```

resolverMIniPRMB (parte 2)

```
    Si el tamaño de ciudades es igual a 3 entonces
        // Evaluamos cada ciudad por separado
        maxIndividual ← Par con la primera ciudad y su beneficio
        Para cada ciudad en ciudades hacer
            Si el beneficio de la ciudad es mayor que el segundo elemento de maxIndividual entonces
                maxIndividual ← Par con la ciudad y su beneficio
            Fin Si
        Fin Para

        // Evaluamos las rutas entre dos ciudades
        maxEntreDos ← Par con las dos primeras ciudades y la ganancia entre ellas
        Si el beneficio de la primera ciudad es mayor o igual al beneficio de la segunda ciudad entonces
            Si el beneficio de la primera ciudad es mayor o igual al segundo elemento de maxEntreDos entonces
                maxEntreDos ← Par con la primera ciudad y su beneficio
            Fin Si
        Sino
            Si el beneficio de la segunda ciudad es mayor o igual al segundo elemento de maxEntreDos entonces
                maxEntreDos ← Par con la segunda ciudad y su beneficio
            Fin Si
        Fin Si

        // Evaluamos las rutas entre las tres ciudades
        rutas ← todas las permutaciones de las tres ciudades
        rutasEntreTres ← primera ruta en rutas
        gananciaEntreTres ← ganancia de rutasEntreTres
        Para cada ruta en rutas hacer
            ganancia ← ganancia de ruta
            Si ganancia es mayor que gananciaEntreTres entonces
                rutasEntreTres ← ruta
                gananciaEntreTres ← ganancia
            Fin Si
        Fin Para
        maxEntreTres ← Par con rutasEntreTres y gananciaEntreTres

        // Elegimos la ruta con mayor ganancia
        solucion ← el Par con mayor segundo elemento entre maxIndividual, maxEntreDos y maxEntreTres
        Retornar solucion
    Fin Si
Fin Función
```

<u>Nota</u>: las dos fotos corresponden al mismo pseudocódigo, pero por temas de tamaño se separaron

- **combinarRutas**: toma dos pares, cada uno de los cuales contiene una lista de ciudades y un valor doble. Estos pares representan dos rutas y susrespectivas ganancias totales. La función devuelve un par que contiene una lista de ciudades y un valor doble, que representa la ruta combinada y su ganancia total.

La función comienza inicializando una lista *rutaConectada* con la primera mitad de la primera ruta.

Luego, si la primera mitad de la primera ruta tiene al menos una ciudad, la función añade la primera ciudad de la segunda ruta a *rutaConectada*.

Después, la función añade el resto de las ciudades de la segunda ruta a *rutaConectada*.

A continuación, la función añade una conexión desde la última ciudad de la segunda ruta a la primera ciudad de la segunda mitad de la primera ruta.

Luego, la función añade el resto de las ciudades de la segunda mitad de la primera ruta a *rutaConectada*.

Finalmente, la función calcula la ganancia total de *rutaConectada* utilizando la función *ganancia*, y devuelve *rutaConectada* y su ganancia total como la solución.

En la siguiente página se presenta el pseudocódigo de combinar

```
-combinarRutas
Función combinarRutas(ruta1, ruta2)
    // Inicializamos la ruta conectada con la primera mitad de la ruta 1
    rutaConectada ← primera mitad de ruta1

    // Añadimos la primera ciudad de la segunda ruta a la ruta conectada
    Si la mitad del tamaño de ruta1 - 1 es mayor o igual a 0 entonces
        Añadir la primera ciudad de ruta2 a rutaConectada
    Fin Si

    // Añadimos el resto de las ciudades de la segunda ruta a la ruta conectada
    Para i desde 1 hasta el tamaño de ruta2 hacer
        Añadir la ciudad i de ruta2 a rutaConectada
    Fin Para

    // Añadimos una conexión desde la última ciudad de la segunda ruta a la primera ciudad de la segunda mitad de la ruta 1
    Añadir la ciudad en la posición mitad del tamaño de ruta1 de ruta1 a rutaConectada

    // Añadimos el resto de las ciudades de la segunda mitad de la ruta 1 a la ruta conectada
    Para i desde la mitad del tamaño de ruta1 + 1 hasta el tamaño de ruta1 hacer
        Añadir la ciudad i de ruta1 a rutaConectada
    Fin Para

    // Calculamos la ganancia de la ruta conectada
    gananciaMaxima ← ganancia de rutaConectada

    // Creamos la solución con la ruta conectada y su ganancia
    solucion ← Par con rutaConectada y gananciaMaxima

    Retornar solucion
Fin Función
```

Detalles de Implementación

Antes de comenzar el proceso de resolución a través de la función *divideAndConquerPRMB* se hace un ordenamiento de los elementos de la lista, en función de su coordenada X. El motivo de este ordenamiento, es que, al momento de dividir la lista en dos partes, cada sublista resultante contiene ciudades que están más cerca entre sí, reduciendo el costo de las visitas para cada ruta e incrementando las posibilidades de crear rutas más efectivas. De hecho, se observó en la práctica que la ganancia aumentaba de forma correcta al ordenarlos, comparando el resultado con el ejemplo propuesto en las especificaciones del proyecto (que, de hecho, fue mayor parte de la referencia en la resolución del problema, al momento de hacer las pruebas).

La decisión de la forma de combinar las rutas en la función *combinarRutas* se debe a que, estudiando los casos, y haciendo pruebas, llegamos a una aproximación que, si bien no es la ruta más efectiva, sigue teniendo datos cercanos a la mejor ganancia posible del problema presentado. Primeramente, se trabajó con diferentes métodos de resolución, pero algunos de éstos incluían bucles que ralentizaban la ejecución del programa, más específicamente, con el archivo d15112.prmb se hicieron exhaustivas pruebas pues era el archivo con más elemento y parte de la idea de la solución es que fuese óptima tanto en ruta como en tiempo.

Así, observamos que la implementación es bastante rápida al trabajar con archivos grandes, por lo que en términos de tiempo de ejecución, se tomó como aceptable. Además, dicha implementación se acerca un poco más al paradigma propuesto por la técnica Divide-and-Conquer, ya que hacemos divisiones e inserciones en los arreglos, sin tener que iterar sobre ellos. De la misma forma, el funcionamiento del algoritmo fue resultado de pruebas y comparación de resultados, tanto como el ejemplo de la referencia de las especificaciones del proyecto, como de resultados manuales. Ya que, al insertar de esa forma una ruta dentro de otra, estamos garantizando que se unen de forma “óptima” las rutas más efectivas, que son las calculadas por la función ad-hoc, pasando por todas las ciudades correspondientes.

Es importante notar que es óptima, pero no es la más eficaz, esto se debe a la magnitud del problema y la cantidad de escenarios posibles del mismo, lo cual lleva a algoritmos de más alto nivel y técnicas fuera del objetivo del curso (y el proyecto en sí). Sin embargo, se determinó que esta ruta es la que tiene menor “pérdida” en cuanto a acercarse al resultado más eficaz se refiere, por lo que se tomó como aceptable para solucionar el problema.

Lecciones Aprendidas

Primeramente, observamos que la técnica Divide-And-Conquer es eficaz para encontrar la ruta de máximo beneficio. Dividir el problema en subproblemas, para encontrar rutas de máximo beneficio en subconjuntos más pequeños de ciudades y luego combinarlas, resulta más eficiente que probar con todas las rutas posibles y devolver la de mayor ganancia, como se comentaba en la sección anterior, donde el tiempo de ejecución y ganancia cambia de mejor manera al implementar esta técnica.

De la misma forma, eso nos lleva también a la conclusión de que las soluciones recursivas pueden ser eficientes. Aunque la recursión puede ser costosa en términos de uso de la memoria en muchos casos, para este problema, es una forma eficiente de explorar todas las posibles combinaciones de rutas.

También se aprecia la importancia de la elección de la estructura de datos correcta. La elección de las estructuras de datos adecuadas (en este caso, las 4-tuplas para representar las ciudades y las listas para representar las rutas) es crucial para la eficiencia de la solución. Una estructura de datos adecuada puede simplificar el código, siendo más sencillo su entendimiento, y también mejorar la eficiencia al permitir operaciones rápidas y eficientes, lo cual permite tanto enfocarse en las partes más difíciles del problema como obtener resultados de prueba, de forma rápida. Fue fundamental tener código bien documentado y con buena semántica.

Respecto al problema presentado, es notable la gran dificultad que se presentó para delimitar el problema correctamente, lo que tal vez habría optimizado la solución presentada. Esto se debe a la cantidad de casos posibles al recorrer una ruta, más aún sabiendo que se puede empezar por cualquier ciudad de las ingresadas. Así, se genera un conjunto de combinaciones muy grande, (dependiendo de la cantidad de ciudades, claro) para algunos casos y de esta forma el pensar en todas las posibilidades, además de no poder comprobar la efectividad de las rutas (es decir, falta de casos de prueba) lo hace más difícil todavía. Sin embargo, esto se extrapola a escenarios de la vida real, donde a veces al solucionar un problema no existen soluciones anteriores, o conjuntos de prueba para verificar datos.