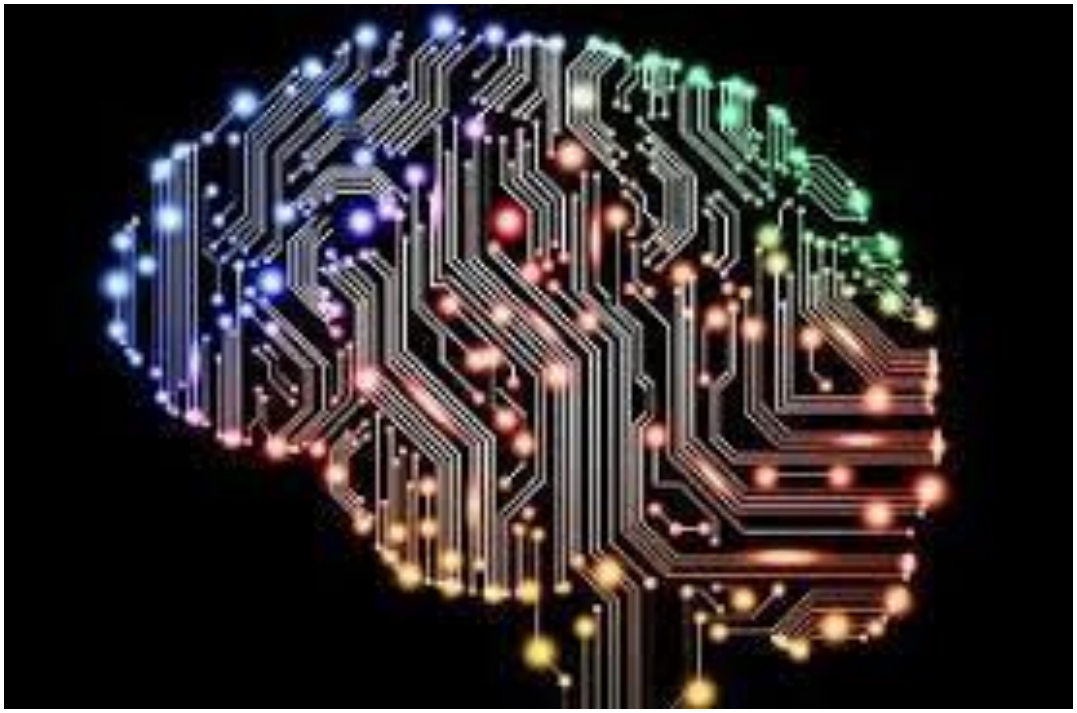
Algorítmica y Modelos de Computación

Práctica 1



Cristian Delgado Cruz
49731061R

Universidad
de Huelva

Índice

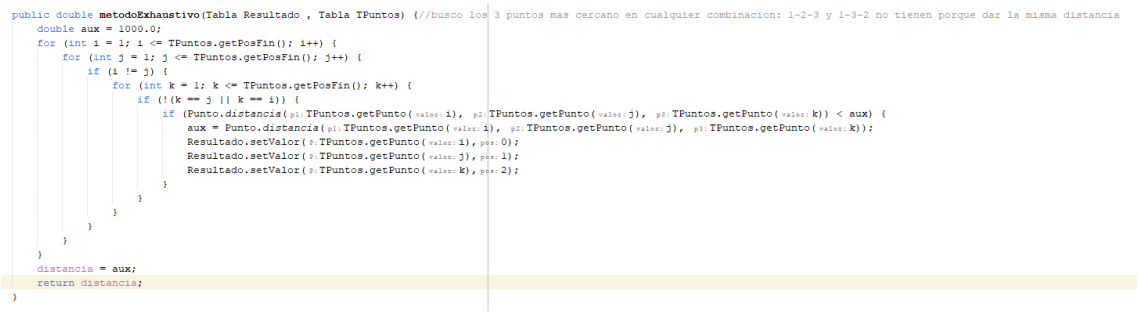
Análisis Teórico

En este Análisis Teórico utilizaré mi código implementado para hacer el cálculo de los costes, para ello utilizaré los cálculos vistos en clase.

Además, compararé el caso exhaustivo con el divide y vencerás, primero teóricamente y luego empíricamente en un caso de 14 puntos y de 1000 puntos aleatorios gracias a una función que yo mismo implementé.

Algoritmo Exhaustivo

Mi código exhaustivo consiste en mirar todas las combinaciones de puntos desde el primero hasta el último y quedarme con aquella combinación de 3 que tenga la menor distancia del primero al segundo y del segundo al tercero.



```
public double metodoExhaustivo(Tabla Resultado , Tabla TPuntos) {//busco los 3 puntos mas cercano en cualquier combinacion: 1-2-3 y 1-3-2 no tienen porque dar la misma distancia
    double aux = 1000.0;
    for (int i = 1; i <= TPuntos.getPosFin(); i++) {
        for (int j = 1; j <= TPuntos.getPosFin(); j++) {
            if (i != j) {
                for (int k = 1; k <= TPuntos.getPosFin(); k++) {
                    if (!(k == j || k == i)) {
                        if (Punto.distancia(p1:TPuntos.getPunto(valor:i), p2:TPuntos.getPunto(valor:j), p3:TPuntos.getPunto(valor:k)) < aux) {
                            aux = Punto.distancia(p1:TPuntos.getPunto(valor:i), p2:TPuntos.getPunto(valor:j), p3:TPuntos.getPunto(valor:k));
                            Resultado.setValor(p:TPuntos.getPunto(valor:i), pos:0);
                            Resultado.setValor(p:TPuntos.getPunto(valor:j), pos:1);
                            Resultado.setValor(p:TPuntos.getPunto(valor:k), pos:2);
                        }
                    }
                }
            }
        }
    }
    distancia = aux;
    return distancia;
}
```

$$T(n) = 1 + \sum_{i=1}^{n}(23 + \sum_{j=1}^{n}(20 + \sum_{k=1}^{n} 14)) =$$

$$T(n) = 1 + 23n + 20n^2 + 14n^3$$

Que al final se convierte en:

$$T(n) \epsilon \boldsymbol{\theta}(\boldsymbol{n^3})$$

Cosa que tiene sentido porque son 3 bucles “for”

Empíricamente en una muestra pequeña de 14 puntos el tiempo en el que carga el resultado es de menos de medio segundo, casi incontable.
Pero con los mil puntos, llegamos a 6 segundos.

Este algoritmo no tiene casos, peor mejor y base, porque comprueba todos los puntos siempre, es decir, su caso base, caso mínimo y caso mejor es el mismo, porque siempre recorre todos los puntos para encontrar el mejor camino.

Algoritmo Divide y Vencerás

Para el estudio teórico no voy a contar el coste de ordenar los elementos y el coste de combinarlo, ya que podríamos considerarlo de orden $0 \in n$ ya que solo se combinan pocos puntos en cada llamada.

```
//Divide y venceras
    private double divide_venceras(Tabla Tpuntos, double prin, double fin, int it, Tabla Resultado) {//el algoritmo como tal

        double mitad = (prin + fin) / 2;//Partimos entre 2
        int indices[] = entrepuntos(Tpuntos, ini:prin, fin);//calcula los puntos que hay entre un indice y el otro

        if (indices[1] - indices[0] + 1 < 3) {//si no hay mas de 2 puntos no puede continuar
            return -1;
        }
        if (repetidospuntos(Tpuntos, indices[0], indices[1])) {//Si los indices son repetidos devuelve -1 para que no los vuelva a hacer
            return -1;
        }

        double izq = DivideyVenceras.this.divide_venceras(Tpuntos, prin, fin:mitad, it + 1, Resultado); //Aplicamos divide y venceras por la iquierda de la particion
        double drc = DivideyVenceras.this.divide_venceras(Tpuntos, prin:mitad, fin, it + 1, Resultado); //Aplicamos divide y venceras por la derecha de la particion

        if (izq == -1 && drc == -1) { //Si no hay conjuntos a la derecha y a la izquierda
            return calculador(Tpuntos, indices[0], indices[1], Resultado);
        }

        double dist = izq;
        if (izq == -1) {
            dist = drc;
        } else if (drc != -1 && drc < izq) {
            dist = drc;
        }

        int[] indicesNuevos = entrepuntos(Tpuntos, mitad - distancia, mitad + distancia);
        if (dist < fin - prin) {
            double aux = calculador(Tpuntos, indicesNuevos[0], indicesNuevos[1], Resultado);
            return aux < dist ? aux : dist;
        }

        double aux = calculador(Tpuntos, indicesNuevos[0], indicesNuevos[1], Resultado);
        return aux < dist ? aux : dist;
    }
}
```

Por ello nos queda la siguiente ecuación:

$$T(n) = \begin{cases} n\text{^}3 \\ 2T(n/2) + n \end{cases}$$

$$T(n) = 2T(n/2) + n;$$

Cambio:

$$n = 2^k \rightarrow T(2^k) = 2T(2^{k-1}) + 2^k;$$

Otro Cambio:

$$T(2^k) = T_k \rightarrow T_k = 2T_{k-1} + 2^k;$$

$$2^k = T(k) = b^k * p(k);$$

$$T_k - 2T_{k-1} = 1^k * 2^k * k^0 \rightarrow P(x) = (x-2)(x-2);$$

$$T_k = C1 * k * 2^k + C2 * 2^k;$$

Deshacemos el cambio:

$$T_k = T(2^k) \rightarrow T(2^k) = C1 * k * 2^k + C2 * 2^k;$$

$$2^k = n \rightarrow T(n) = C1 * n * \log n + C2 * n;$$

Quedándonos así que el orden es de 0 ∈ $nlogn$

Empíricamente la diferencia entre 14 puntos y 1000 puntos no tiene diferencia apreciable gracias a su orden de complejidad.

Su caso peor será cuando valga n^3, es decir el último trio de puntos que examine sea el valido.

Su caso mejor será encontrarlos instantáneamente es decir con la primera partición encontrar una distancia tan pequeña que no haya trio de puntos que al compararlos den algo menor que ese valor

Diferencias entre Exhaustivo y DyV

Como es de suponer, para los casos más pequeños la diferencia entre el Algoritmo exhaustivo y Divide y vencerás son completamente imperceptibles, ya que el funcionamiento del exhaustivo en casos pequeños en muy bueno, pero cuanto más puntos añadimos peor rendimiento da.

Por otra parte el Divide y Vencerás como va quitándose del medio puntos con cada llamada recursiva termina acabando muchísimo antes, y aunque en casos muy pequeños no se note la diferencia, en casos grandes tiene un rendimiento tan superior que deja al exhaustivo completamente fuera de poder hacer cualquier cosa.

Un ejemplo práctico podría ser el siguiente:

Si queremos calcular la distancia entre tres pueblos de una provincia, calculamos (según la provincia) la combinación de tres en tres de todos ellos, y quizá nos puede salir 20 pueblos en una provincia, eso significa que da igual que método usemos, nos valdrían los dos, por otra parte si queremos hacer lo mismo pero para toda España o aún peor para toda Europa, tendríamos que considerar todas las combinaciones haciendo casi infinito el método exhaustivo.

Por ello el DyV es muchísimo mejor algoritmo

Algoritmo Dijkstra

```
public Voraz(int NumeroNodos, ArrayList<Nodo> Nodos) {
    this.Nodos = Nodos;
    ColaNodos = new ArrayList();

    Caminos = "";
    Resultado = new int[NumeroNodos];
    Visitado = new boolean[NumeroNodos];
    Anterior = new int[NumeroNodos];

    for (int i = 0; i < NumeroNodos; i++) {
        Resultado[i] = Integer.MAX_VALUE;
        Visitado[i] = false;
        Anterior[i] = -1;
    }
}
```

```
public int[] metodoDijkstra() {
    Nodo NodoActual = null;
    ColaNodos.add( e: Nodos.get( index: 0));
    Resultado[0] = 0;

    while (!ColaNodos.isEmpty()) {
        NodoActual = ColaNodos.get( index: 0);
        ColaNodos.remove( index: 0);
        if (Visitado[NodoActual.getPosicion()]) {
            continue;
        } else {
            Visitado[NodoActual.getPosicion()] = true;
            for (int i = 0; i < NodoActual.getAristas().size(); i++) {
                int posadyacente = NodoActual.getAristas().get( index: i).getLlegada();
                int valor = NodoActual.getAristas().get( index: i).getValor();
                if (!Visitado[posadyacente]) {
                    cambio( actual: NodoActual.getPosicion(),  siguiente: posadyacente, valor);
                }
            }
        }
    }
    return Resultado;
}

public void cambio(int actual, int siguiente, int valor) {
    if (Resultado[actual] + valor < Resultado[siguiente]) {
        Resultado[siguiente] = Resultado[actual] + valor;
        Anterior[siguiente] = actual;
        if (ColaNodos.isEmpty()) {
            ColaNodos.add( index: 0,  element: Nodos.get( index: siguiente));
        } else {
            if (Resultado[siguiente] > Resultado[ColaNodos.get( index: 0).getPosicion()]) {
                ColaNodos.add( e: Nodos.get( index: siguiente));
            } else {
                ColaNodos.add( index: 0,  element: Nodos.get( index: siguiente));
            }
        }
    }
```

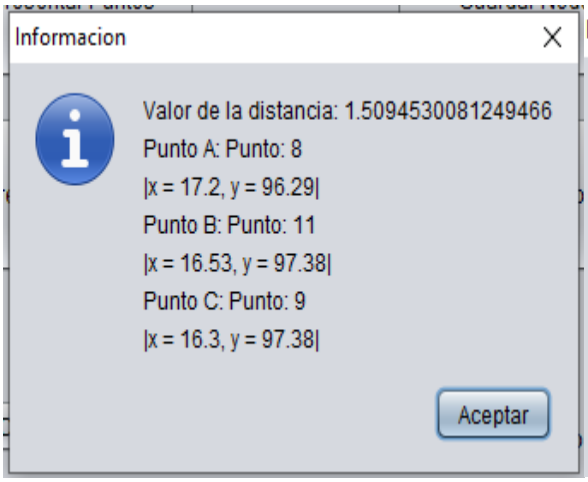
En este caso, tenemos un algoritmo donde la complejidad viene definida por el bucle for donde asignamos los pesos a la variable Resultado [], para ello habrá que recorrerla entera es decir, complejidad n. También hay que tener en cuenta el bucle donde buscamos el camino mejor, donde recorreremos toda estos nodos para hallar los caminos con los costes antes calculados. Por lo tanto, podemos afirmar que el algoritmo tiene una complejidad de n^2.

Los diferentes casos que podrían hacer el algoritmo dependen del número de intercambios que tenga que realizar para reasignar los caminos

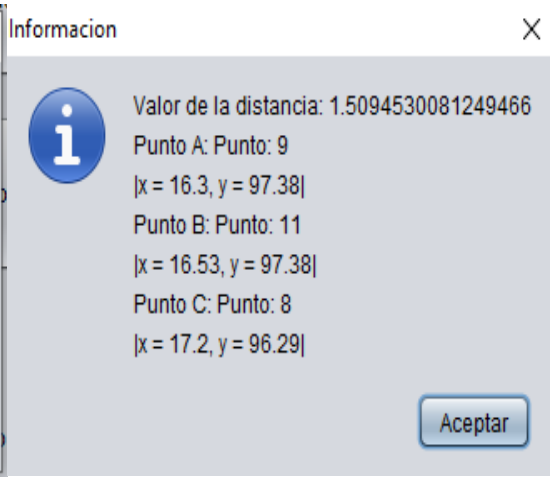
El mejor caso se produciría cuando no se añade ningún punto al camino para ir de un vector a otro y todos los caminos son directos. Aun así, tendríamos que inicializar la variable Resultado [] , buscar la arista con el coste mínimo y hacer todas las comprobaciones por lo que la complejidad en el mejor caso y peor seguirían siendo n^2.

Representación de los resultados

Puntos



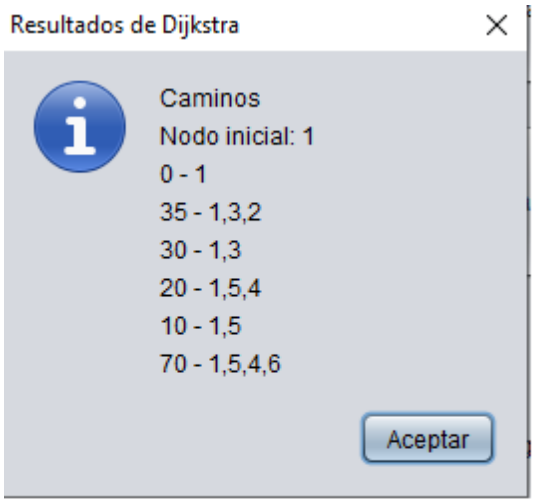
Exhaustivo

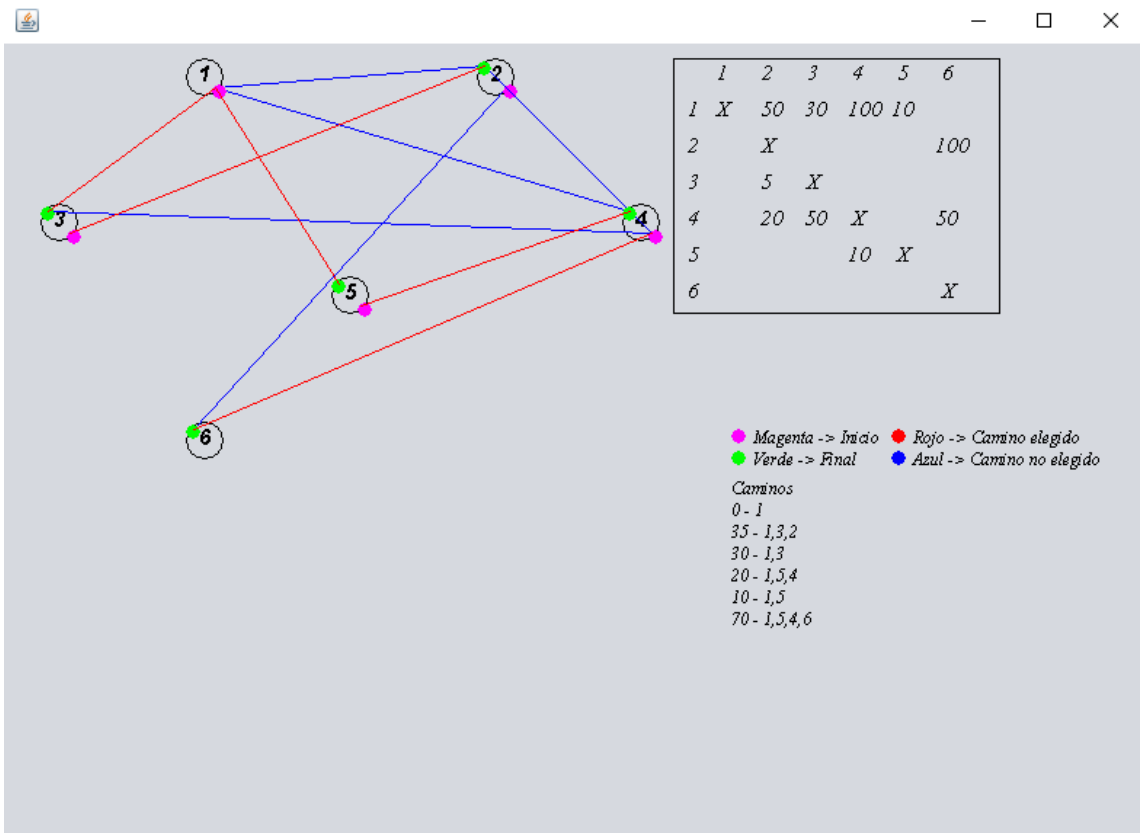


Divide y Vencerás

Los Azules son los puntos resultados.

Nodos


Resultados de Dijkstra
Caminos
Nodo inicial: 1
0 - 1
35 - 1,3,2
30 - 1,3
20 - 1,5,4
10 - 1,5
70 - 1,5,4,6
Aceptar


1
2
3
4
5
6
1 X 50 30 100 10
2 X 100
3 5 X
4 20 50 X 50
5 10 X
6 X
Magenta -> Inicio
Rojo -> Camino elegido
Verde -> Final
Azul -> Camino no elegido
Caminos
0 - 1
35 - 1,3,2
30 - 1,3
20 - 1,5,4
10 - 1,5
70 - 1,5,4,6

Explicación del Funcionamiento General

Paquetes

Algoritmo:

Contiene los métodos de los algoritmos propiamente dichos, Divide y Vencerás, Exhaustivo, QuickShort (Que ordena para el Divide y Vencerás) y Voraz.

Controlador:

Contiene las clases que se encargan de controlar el funcionamiento de las vistas, todas implementan ActionListener y para ello Sobrescriben el método actionPerformed, además de un addListener para escuchar los eventos.

Modelo:

Contiene las clases que se encargan de mantener los objetos que usamos en la práctica, además encontramos aquí Escritura y Lectura que son las clases que se encargan del uso de ficheros y son llamadas por el controlador principal.

Vista:

Contiene las clases que se encargan de mostrar por pantalla los resultados, mensajes y demás para informar al usuario de sus resultados.

Explicación de la interfaz gráfica

La interfaz gráfica es un JFrame con varios botones que son capturados como eventos en el controlador.

Por otra parte, los resultados son mostrados en JOptionpanel y la representación en Canvas montados en un Panel.

Los Botones 1 a 1 funcionan así:

Representar Puntos: Representan los puntos solo si se han cargado o creado antes, si no, salta un error en un JOptionpanel.

Representar Solución: Lo mismo que el de arriba pero solo si se han registrado unos resultados antes.

Crear Puntos: Crea 1000 puntos aleatorios, la cantidad puede ser cambiada en el código.

Cargar Puntos: Carga puntos de un fichero especifico, este fichero también puede ser cambiado en el código.

Guardar Nodos: Guarda los nodos en un fichero, que también se puede cambiar en el código.

Cargar Nodos: Crea los Nodos desde un código que también se puede retocar a gusto

Continuar: Encuentra la solución con uno de los métodos escogidos en el JComboBox, si escoges Exhaustivo o DyV sin cargar/crear los puntos o Dijkstra sin cargar los nodos dará un error en un JOptionpanel, si eliges Dijkstra también representará la solución.

Salir: Sale del programa.

El Combo Box sirve para elegir el algoritmo.