RESUM ADA

Pràctica 1 - Algoritmo de Dijsktra

Teniendo un grafo ponderado de N nodos, sea x el nodo inicial, un vector D de tamaño N guardará al final del algoritmo las distancias desde x al resto de los nodos.

1. Inicializar todas las distancias en D con un valor infinito relativo ya que son desconocidas al principio, exceptuando la de x que se debe colocar en 0 debido a que la distancia de x a x sería 0.
2. Sea a = x (tomamos a como nodo actual).
3. Recorremos todos los nodos adyacentes de a, excepto los nodos marcados, llamaremos a estos nodos no marcados vi.
4. Para el nodo actual, calculamos la distancia tentativa desde dicho nodo a sus vecinos con la siguiente fórmula: dt(vi) = Da + d(a,vi). Es decir, la distancia tentativa del nodo 'vi' es la distancia que actualmente tiene el nodo en el vector D más la distancia desde dicho el nodo 'a' (el actual) al nodo vi. Si la distancia tentativa es menor que la distancia almacenada en el vector, actualizamos el vector con esta distancia tentativa. Es decir: Si dt(vi) < Dvi ⟶ Dvi = dt(vi)
5. Marcamos como completo el nodo a.
6. Tomamos como próximo nodo actual el de menor valor en D que no se haya visitado y volvemos al paso 3 mientras existan nodos no marcados.

Una vez terminado al algoritmo, D estará completamente lleno con las distancias de x a cualquier otro nodo del grafo.
Complexitat Dijsktra 1 = O(V^2)
Complexitat Dijsktra 2 = O(A log(V)) Cua de prioritat

Representación del Grafo

El grafo con el que trabajaremos será un grafo cuyos vértices son puntos del plano real. El peso de una arista que conecta dos vértices será la distancia euclídea entre los dos vértices.

Clases

Los puntos del plano se representan con la clase CPoint:

```
public class CPoint {
        public double m_X,m_Y;
        public CPoint(double x,double y);
        public CPoint Neg();
        public CPoint Add(CPoint b);
        public CPoint Sub(CPoint b);
        public double Product(CPoint b);
        public CPoint Div(double b);
        public double Module();
        }
```

Esta clase implementa puntos del plano con coordenadas m_X y m_Y. Además implementa las operaciones aritméticas y módulo entre puntos.

La clase CVertex representa un vértice del grafo. Esta contiene el punto del plano donde se encuentra el vértice y una lista de los vértices vecinos. Además tiene los campos m_Distance y m_Visit necesarios para implementar el algoritmo de Dijkstra.

```
public class CVertex {
        public CPoint m_Point;
        public LinkedList<CVertex> m_Neighbords;
        public double m_Distance;
        public boolean m_Visit;
        public CVertex(double x, double y);
        }
```

La clase CGraph representa todo el grafo como un array de vértices (m_Vertices) y nos da todas las funcionalidades necesarias para trabajar con el grafo.

```
public class CGraph {
        public ArrayList<CVertex> m_Vertices;
        public CGraph();
        public void Clear();
        // Vertices -----------------------------------------------------
        public CVertex FindVertex(double x,double y);
        public CVertex GetVertex(double x,double y) throws Exception;
```

```
        public CVertex GetVertex(CPoint p) throws Exception;
        public boolean MemberP(CVertex v);
        // Edges ------------------------------------------------------
        public void Add(double x1, double y1, double x2, double y2);
        // Draw -------------------------------------------------------
        public void Draw(Graphics g,double esc, Color c);
        public void AddRectHull(CPoint min, CPoint max);
        // Files ------------------------------------------------------
        public void Write(String filename) throws Exception;
        public void Read(String filename) throws Exception;
        // Dijkstra ---------------------------------------------------
        public void Dijkstra1(CVertex start) throws Exception;
        public void Dijkstra2(CVertex start) throws Exception;
        }
```

Para crear un grafo se utiliza el constructor y la función Add nos permite añadir dos puntos que forman una arista del grafo. Esta función ya se encarga de crear automáticamente los vértices y actualizar sus listas de vecinos.
Finalmente la clase CGraphView implementa una ventana donde se pueden visualizar los resultados. Esta ventana puede visualizar objetos CGraph. Es especialmente útil para ver los resultados del algoritmo que implementareis.

```
public class CGraphView extends JFrame {
        private static final long serialVersionUID = 1L;
        private static final int HEIGHT = 400;
        private static final int WIDTH = 700;
        private ArrayList<Object> m_Elements;
        private ArrayList<Color> m_Colors;
        public CGraphView();
        public void paint(Graphics g);
        public void ShowGraph(CGraph graph,Color color);
        }
```

Práctica 2: Algoritmos Greedy

La solución que buscaremos será aproximada y se basará en la heurística de que cuando estamos en una ciudad, iremos a la ciudad más cercana de las que tengamos que visitar. Esta heurística permite utilizar el siguiente algoritmo greedy:

1. Empezar por la ciudad de origen.
2. Mientras hayan ciudades por visitar repetir
    a. Buscar cual es la ciudad de las que tenemos que visitar más cercana a la que estamos.
    b. ir a la ciudad más cercana
    c. Quitar la ciudad donde estamos de las que tenemos que visitar.
3. Ir a la ciudad de origen.

Para representar las ciudades y la red de carreteras utilizaremos un grafo donde cada vértice es un punto del plano y las aristas son las rectas que unen estos puntos. Las distancias a recorrer serán las distancias euclideas entre los puntos del plano (rectas entre vértices). Como tenemos un grafo, podremos utilizar para calcular el camino más corto entre dos ciudades el algoritmo de Dijkstra para el paso 2.a del algoritmo.

La clase CGraph representa todo el grafo como un array de vértices (m_Vertices) y nos da todas las funcionalidades necesarias para trabajar con el grafo.

```
public class CGraph {
        public ArrayList<CVertex> m_Vertices;
        public CGraph();
        public void Clear();
        // Vertices ---------------------------------------------------
        public CVertex FindVertex(double x,double y);
        public CVertex GetVertex(double x,double y) throws Exception;
        public CVertex GetVertex(CPoint p) throws Exception;
        public boolean MemberP(CVertex v);
        // Edges ------------------------------------------------------
        public void Add(double x1, double y1, double x2, double y2);
        // Draw ------------------------------------------------------- public void Draw(Graphics
        g,double esc, Color c);
        public void AddRectHull(CPoint min, CPoint max);
```

```
        // Files ------------------------------------------------------------
        public void Write(String filename) throws Exception;
        public void Read(String filename) throws Exception;
        // Track ------------------------------------------------------------
        public CTrack SalesmanTrack(CVisits visits) throws Exception;
        }
```

Para poder indicar un conjunto de ciudades a visitar se utiliza la clase CVisits. Esta es una lista de puntos del plano que han de coincidir con los vértices del grafo. El primer punto de la lista es el punto de origen.

```
public class CVisits {
        public LinkedList<CPoint> m_Points;
        public CVisits();
        public void Add(double x, double y) throws Exception;
        public void Clear();
        // Draw -------------------------------------------------------------
        public void Draw(Graphics g,double esc, Color c);
        public void AddRectHull(CPoint min, CPoint max);
        // Files ------------------------------------------------------------
        public void Write(String filename) throws Exception;
        public void Read(String filename) throws Exception;
        // Print ------------------------------------------------------------
        public String toString();
        }
```

La clase CTrack se utiliza para representar el camino que tendrá que seguir el viajante. Es una lista de vértices del grafo.

```
public class CTrack {
        public LinkedList<CVertex> m_Vertices;
        public CGraph m_Graph;
        public CTrack(CGraph graph);
        public void AddLast(double x, double y) throws Exception;
        public void AddLast(CPoint p) throws Exception;
        public void AddLast(CVertex v) throws Exception;
        public void AddFirst(CVertex v);
        public void Clear();
        public void Append(CTrack t);
        // Draw -------------------------------------------------------------
        public void Draw(Graphics g,double esc, Color c);
        public void AddRectHull(CPoint min, CPoint max);
        // Files ------------------------------------------------------------
        public void Write(String filename) throws Exception;
        public void Read(String filename) throws Exception;
        // Print ------------------------------------------------------------
        public String toString();
        }
```

Práctica 3: Bácktrácking

Algoritmo de Backtracking Puro

En este algoritmo partiremos del primer vértice a visitar y seguiremos la primera arista que parta de él. Al llegar a un nuevo vértice miraremos si es de los que tenemos que visitar y lo quitaremos de la lista de los vértices a visitar. Luego seguiremos por la primera arista del vértice donde estamos. En el caso que una arista nos lleve a un vértice por el que ya hemos pasado consideraremos que el camino no puede seguir por él y volveremos atrás para seleccionar otra arista por la que continuar. Este proceso lo repetiremos hasta conseguir visitar todos los vértices de la lista de vértices a visitar. Finalmente seguiremos avanzando hasta conseguir llegar al vértice de partida.

Como queremos la solución óptima, cuando hemos conseguido visitar todos los vértices de la lista de visitas y hemos vuelto al vértice de partida, guardaremos el camino si es más corto que los caminos anteriores. Esto nos permite considerar que no tenemos que seguir una rama del árbol de búsqueda de soluciones si la longitud del camino que está construyendo es mayor que la del mejor camino que ya ha encontrado.

Como estamos buscando un camino en un grafo no orientado, es muy fácil caer en bucles infinitos donde vamos repitiendo siempre el mismo camino. Así parece buena idea no permitir pasar por vértices que ya se ha pasado, pero esto hace que en algunos casos no podamos llegar a una solución. Por ejemplo, tenemos que visitar un vértice al que sólo se puede llegar por una arista. En este caso tendremos que permitir repetir el vértice que conecta con el que tenemos que visitar. Así que hay que pensar cuando se puede permitir pasar dos veces por el mismo vértice y cuando no se ha de permitir para evitar entrar en bucles infinitos.

Algoritmo Combinado Backtracking y Greedy

En este algoritmo aprovecharemos el algoritmo de Dijkstra para buscar el camino más corto entre dos vértices a visitar. Así el espacio de búsqueda que exploraremos con backtracking se reducirá al orden en que visitamos los vértices de la lista de visitas. De esta forma, si tenemos que visitar n vértices, el primer nodo del árbol de búsqueda tendrá n hijos que corresponden a los n vértices a visitar. En el siguiente nivel, los nodos del árbol tendrá n-1 hijos ya que ya se ha visitado un vértice, y así, sucesivamente hasta completar los n niveles del árbol. Como explorar todo el árbol puede ser muy costoso, se puede realizar una poda basada en que si el mejor camino que hemos encontrado es más corto que la longitud del camino que estamos creando, no hace falta completar el camino que estamos creando.

Práctica 4: Branch & Bound

Se implementarán tres algoritmos basados en branch & bound. La diferencia será la heurística utilizada para dirigir el recorrido por el árbol del espacio de soluciones. Estos tres algoritmos tendrán que encontrar el camino más corto posible, o sea, la solución óptima. Las heurísticas utilizadas serán:

1. Siempre completar el camino parcial más corto de los que se hayan creado. En el caso de haber encontrado un camino completo, se podarán los caminos parciales más largos que el mejor camino completo que se haya encontrado.
2. Para cada camino parcial calcular una cota inferior de la longitud que tendría el camino completo. Completar el camino parcial cuya cota inferior para la longitud total sea la menor de los caminos parciales. En el caso de haber encontrado un camino completo, se podarán los caminos parciales cuya cota inferior sea mayor la longitud del mejor camino completo que se haya encontrado. La cota inferior para el camino completo será la suma de la longitud del camino parcial más el camino más corto de cada uno de los vértices que faltan por visitar a su vecino más cercano. Este vecino será unos de los vértices a visitar. Para reducir el tiempo de cálculo en vecino más cercano será cualquier vértice de los que se encuentra la lista visits aunque ya se haya visitado.
3. Heurística propuesta por el alumno. Esta heurística se explicará en el informe de la siguiente práctica (práctica 5).

Para la implementación del algoritmo de branch & bound se puede utilizar una cola con prioridad donde se guardarán las soluciones parciales que se pueden completar. Es importante recordar que este algoritmo ha de encontrar la solución óptima.