Trabajo Práctico Especial
Análisis y Diseño de Algoritmos I

Integrantes:

Coria, Santiago

Santos, Matías Andrés

Contacto:

santiagocoria@live.com.ar

matias_a_santos@hotmail.com

Introducción

En este informe veremos la implementación de un programa que triangula polígonos, mediante el uso de la plataforma Qt Creator, y las decisiones tomadas para atacar este problema que se nos plantea.

Problema

Problema de la triangulación: Dados un polígono convexo y la distancia entre cada par de vértices deberemos seleccionar un conjunto de cuerdas tal que:

- Dos cuerdas no se crucen entre sí.
- Todo el polígono quede dividido en triángulos.
- La longitud de todas las cuerdas solucionadas debe ser mínima.

Resolución del problema

Para la resolución del problema propuesto se implementó la técnica de programación dinámica la cual resolverá el problema, empezando por solucionar los subproblemas más simples y guardar esa información en alguna estructura para luego utilizarla al momento de resolver problemas más complejos.

Denotaremos como Sis al subproblema de tamaño s partiendo del vértice i, el problema de la triangulación mínima para el polígono formado por los s vértices que comienzan en Vi hasta $Vi+s-1$.

Para resolver el subproblema S debemos tener en cuenta dos consideraciones:

- En cualquier triangulación con más de tres vértices, cada par de vértices adyacentes es tocado por al menos una cuerda, si no es así existiría una región que no es un triángulo.
- Si (V_i , V_j)es una cuerda de una triangulación, debe haber algún Vk , tal que (V_i , V_k) y (V_k , V_j) sean aristas del polígono o cuerdas de la triangulación. Si no es así (V_i , V_j) pueden estar limitando una región que no es un triángulo.

Una vez tomadas en cuentas las consideraciones se utilizará la siguiente formulación recursiva (C_{iS}) para ir completando la tabla de soluciones:

- Si S < 4 , C_{iS} = 0
- Si S > 4, C_{iS} = Min ($1 \leq K \leq S-2$) { $C_{i\,k+1} + C_{i+k\,s-k} + D(V_i\,,\,V_{i+k}) + D(V_{i+k}\,,\,V_{i+s-1})$ }

Donde:

$D(V_p\,,\,V_q) = 0$ si Vp y Vq son adyacentes.

$D(V_p\,,\,V_q) =$ distancia entre Vp y Vq si no son adyacentes.

Una vez planteada la formulación recursiva, podremos ir dividiendo el problema en problemas de menor tamaño y ir almacenando la solución (C_{iS}) en nuestra tabla hasta llegar al problema de tamaño que incluya todos los vértices del polígono, luego el resultado de la triangulación mínima lo obtendremos en la $Tabla_{(n,0)}$.

El siguiente pseudo-código es útil para dar una idea básica del algoritmo

```
For (int s = 4; s < n ; s++)
        For (int i = 0; i < n ; i++)
                min = infinito , cuerda = 0;
                For (int k=0; k < s-2 ; k++)
                        aux= costo()
                        if (min > aux)
                                min= aux;
                                Cuerda=K;
        C[ s ][ i ]= min;
        TablaCuerdas[ s ][ i ] = Cuerda;
```

Cuerdas de la triangulación mínima

Para encontrar las cuerdas de la triangulación mínima, con la TablaCuerdas que genero de hacer la triangulación mínima, es posible con una simple recursión hallar las cuerdas de nuestro polígono. La cual estará dada (V_i, V_{i+k}) y (V_{i+k}, V_{i+s-1}) a menos que no sea cuerda. Y además las cuerdas implicadas por las soluciones $S_{i,K+1}$ y $S_{i+K,S-K}$.

Tipos de datos abstractos

Para la resolución del problema planteado anteriormente se decidieron utilizar las siguientes estructuras y tipos de datos abstractos que se explicaran a continuación.

Se implementaron las clases Punto y Polígono, la clase polígono su estructura consta de dos valores double, uno de ellos simboliza la posición x y el otro la posición y, mientras que el polígono se decidió utilizar una lista de Puntos.

A continuación describiremos la utilidad de los métodos y complejidad temporal de ellos, de cada clase.

Métodos de la clase Punto

- **Punto(double x,double y) :** Método constructor de la clase, se encarga de crear una instancia y setear los valor de x e y a partir de los que recibe, su complejidad es O(1).
- **Punto(Punto & aux) :** Método constructor de la clase, se encarga de crear una instancia y setear los valor de x e y, a partir de los valores x e y del punto que recibe como parámetro, su complejidad es O(1).
- **Punto():** Método constructor de la clase, se encarga de crear una instancia vacía, su complejidad es O(1).
- **setX(double x):** Método modificador de la clase, se encarga de reemplazar el valor actual de x por el que recibe como parámetro, su complejidad es O(1).
- **set(double y):** Método modificador de la clase, se encarga de reemplazar el valor actual de y por el que recibe como parámetro, su complejidad es O(1).
- **getX():** Método observador de la clase, cuando este método es invocado se encarga de devolver el valor x del punto, su complejidad es O(1).
- **getY():** Método observador de la clase, cuando este método es invocado se encarga de devolver el valor y del punto, su complejidad es O(1).
- **distancia(Punto aux):** Método observador de la clase,a partir del punto con el que fue llamado calcula su distancia con el punto que recibe como parámetro, su complejidad es O(1).
- **productoCruzado(Punto aux):** Método observador de la clase,a partir del punto con el que fue llamado calcula su producto cruzado con el punto que recibe como parámetro, su complejidad es O(1).

Métodos de la clase Polígono

- **Poligono(list<Punto> aux):** Método constructor de la clase, a partir de la lista que recibe como parámetro se encarga de generar un polígono, su complejidad es O(n) siendo n el tamaño de la lista aux.
- **~Poligono():** Método destructor de la clase, se encarga de destruir el polígono, su complejidad es O(n) siendo n el tamaño del polígono.
- **agregar(Punto aux):** Método modificador de la clase, se encarga de llamar al método privado agregarPrivado con el punto que recibe como parámetro, su complejidad es O(1).
- **sizePoligono():** Método observador de la clase, se encarga de devolver el tamaño del polígono, su complejidad es O(1).
- **calcularArea():**Metodo observador de la clase, se encarga de devolver el area del polígono, su complejidad es O(n) siendo n el tamaño del polígono.
- **calcularPerimetro():**Método observador de la clase, se encarga de devolver el perimetro del poligono, su complejidad es O(n) siendo n el tamaño del polígono.
- **triangulacion(matrizCosto,matrizK,costoTotal):**Metodo observador de la clase, se encarga de calcular la triangulación y devolverla en los parámetros pasados por referencia, su complejidad es O(n^3) siendo n el tamaño del polígono.
- **trazarLineas(list<Punto> aux, int s,int k,int i,matrizK):** Método observador de la clase que se encarga de retornar en una lista las cuerdas obtenidas de la triangulación, su complejidad es O(n) siendo n el tamaño del polígono.
- **agregarPrivado(Punto aux):** Método modificador que se encarga de agregar un punto al polígono, su complejidad es O(1).
- **crearMatriz():** Método observador que se encarga de crear la matriz utilizada en la triangulación, su complejidad es O(n) siendo n el tamaño del polígono.
- **preCargarMatriz():**Método observador que se encarga de cargar la matriz utilizada en la triangulación, su complejidad es O(n) siendo n el tamaño del polígono.
- **buscarMayorYMenorY(Punto & may, Punto & men):** Método observador que recorre todo el polígono, y se queda con el menor y el mayor punto del mismo, su complejidad es O(n). En caso de que haya más de un punto con el mismo valor en y, se queda con el más a la izquierda, para el menor, y el más a la derecha, para el mayor.
- **inicializarPuntos(Punto p1,Punto p2, Punto puntoActual):** Método modificador que inicializa los puntos con los que se realizará el algoritmo Jarvis-March con puntos que pertenezcan al polígono, su complejidad es O(1).
- **calculaMejorPuntoMenor(Punto p1, Punto p2, Punto puntoActual):** Método modificador que analiza y encuentra cuál será el próximo punto agregado a la cadena derecha, su complejidad es O(1).
- **calculaMejorPuntoMayor(Punto p1, Punto p2, Punto puntoActual):** Método modificador que analiza y encuentra cuál será el próximo punto agregado a la cadena izquierda, su complejidad es O(1).

- **convexHull():** Método que se encarga de devolver una lista conformada por los puntos que envuelvan a todos los puntos del polígono de donde se lo llama.

Especificación en Nereus

TDA Punto

Class Punto
Imports QtMath, Real, Boolean
Basic constructors crearPunto
Effective
Type Punto
Operations

Punto : Real * Real -> Punto;	constructora
getX: Punto -> Real;	observadora
getY: Punto -> Real;	observadora
distancia: Punto * Punto -> Real;	observadora
productoCruzado: Punto * Punto -> Real;	observadora

Axioms

...

End-class

TDA Polígono

Class Poligono

Imports QList, Real, Boolean,Punto
Basic constructors Polígono
Effective
Type Polígono
Operations

Poligono: QList -> Poligono	constructora
agregar: Poligono * Punto -> Poligono;	constructora
sizePoligono: Poligono -> Integer	observadora
calcularArea: Poligono -> Real	observadora
calcularPerimetro: Poligono -> Real	observadora
convexHull: Poligono -> QList	observadora

Axioms

...

End-Class

Conclusión

Al realizar el trabajo correspondiente pudimos implementar la técnica de programación dinámica algo que nunca habíamos hecho y nos sirvió mucho para comprender mejor cómo funciona esta técnica, además al tener que hacer una interfaz gráfica, entendimos la magnitud de lo que sería hacer un programa es su totalidad y el tiempo que lleva hacer este. También debimos investigar cómo utilizar la plataforma gráfica Qt Creator, que nos complicó bastante a la hora de trabajar ya que nunca habíamos implementado una interfaz de esta magnitud, y no sabíamos cómo encarar el diseño de la aplicación.

Al ser un trabajo de este calibre entendimos que no podemos los dos trabajar en lo mismo y que no debíamos dividir tareas para así lograr optimizar el tiempo que teníamos para trabajar, para hacer dicha subdivisión de tareas utilizamos una plataforma (Trello) para así tener un esquema de cómo progresamos día a día, logrando así poder unir los códigos de una manera mucho más eficiente.