Análisis y desarrollo de Algoritmos (ADA) Diagramas de Voronoi

Fabrizio Franco
School of Computer Science
University of Engineering and Technology UTEC
Email: fabrizio.franco@utec.edu.pe

***Resumen*—En el presente trabajo, se realiza una introducción a los Diagramas Voronoi, se explican sus aplicaciones y se hace un análisis asintótico profundos sobre los principales algoritmos para obtener estas estructuras geométricas.**

***Index Terms*—Polígonos de Thiessen, Diagramas Voronoi, Algoritmo de Fortune, Análisis asintótico.**

I. Definición

Los diagramas de Voronoi o polígonos de Thiessen son estructuras matemáticas de carácter geométrico. Se basan en crear particiones en un plano. Estos diagramas utilizan la distancia euclidiana para interpolar. Es decir, a partir de un conjunto de puntos conocidos, obtener nuevos puntos; este proceso es acumulativo y se puede dividir en intervalos, por ejemplo existe la interpolación lineal, un método que recibe los puntos (I_x, I_y) y (J_x, J_y) para obtener más puntos, este método en particular, es el más sencillo para interpolar, por lo que la precisión puede no ser lo suficientemente buena para algunas aplicaciones en concreto.

Con la interpolación se generan una serie de líneas rectas, para cada una de dichas líneas existe una mediatriz (una línea recta perpendicular). Es posible, a partir de las intersecciones de las mediatrices generar particiones al plano con la siguientes siguientes propiedades:

- Para todo punto que pertenece al perímetro de alguna partición, este equidistante a los puntos originales que sean vecinos
- Para todo punto que no pertenece al perímetro, este se encuentra en la superficie de la partición del punto original más cercano.

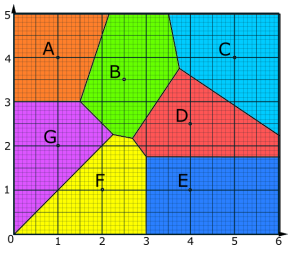

Figura 1: Ejemplo de un diagrama de Voronoi - [10]

II. Trasfondo matemático

II-A. Distancia Euclidiana

Ya que esta estructura se aplica sobre un plano euclídeo, la distancia más apropiada a utilizarse es la Distancia Euclidiana. Sean dos puntos (I_x, I_y) y (J_x, J_y) , la distancia entre ellos está dada por:

$$d(I, J) = \sqrt{(J_x - I_x)^2 + (J_y - I_y)^2}$$

II-B. Definición formal

Para un conjunto X de puntos en el plano con n elementos, podemos definir a su Diagrama Voronoi como al conjunto de n divisiones en el plano, cada una asociada a un elemento $x_i \in X$. Para cada punto en el plano k, este se ubica en la superficie asociada a $x_i \leftrightarrow [d(k, x_i) < d(k, x_j) \forall x_j \in \{X \backslash x_i\}]$

II-B1. Notación: El diagrama Voronoi de X se denota como $Voronoi(X)$, Las diferentes regiones para cada x_i se denotan como $V(x_i)$. Formalmente, esta región se construye de las intersecciones de las bisectrices de las rectas de x_i hacia todo x_j tal como se definió anteriormente. Esta operación hace que para cada par de puntos se divida el plano en 2 semiplanos, el que contiene a x_i se denota como $p(x_i, x_j)$ por lo que $V(x_i)$ es la intersección de todos los semiplanos generados, resultando en lo siguiente:

$$V(x_i) = \bigcap_{j=1}^{n} p(x_i, x_j)$$

Un ejemplo gráfico es el siguiente:

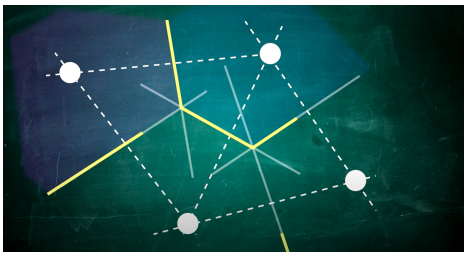

Figura 2: Ejemplo - [9]

II-C. Propiedades de los diagramas de Voronoi

III. Aplicaciones

Los polígonos de Thiessen tienen diversas aplicaciones prácticas en muchos campos que utilizan Ciencias de Computación como herramienta, así como también campos propios

de Ciencias la computación. Entre los ejemplos más notables destacan Computación gráfica, metereología, topografía, epidemiología. Aplicaciones más detallas son las siguientes:

- Epidemiología: Los polígonos de Thiessen permiten, junto a una base de datos con la información de los casos detectados y un mapa de densidad, identificar que superficie $V(x_i)$ contiene la densidad de puntos mayor, región que representa el origen de la infección. J. Snow fue quien utilizó este método por primera vez en la batalla contra el cólera.
- Location tracking: El software de geo-localización usa Diagramas de Voronoi según lo necesite, por ejemplo, si un peatón desea saber cuál es el restaurante más cercano al punto k en donde se encuentra. La ubicación del transeúnte está dentro de una superficie $V(x_i)$, por lo que x_i será la localización en donde se encuentre dicho restaurante más cercano. //Fuente derivando
- Aviación: Identificar el aeropuerto más cercano en caso de aterrizaje de emergencia. De realizarse incorrectamente, personas podrían morir de una forma evitable. Los diagramas de Voronoi son la herramienta adecuada para calcular dicho aeropuerto más cercano al punto k, actual ubicación del avión. En este caso el conjunto de puntos X del cual se construye el diagrama de Voronoi es el conjunto de aeropuertos donde un aterrizaje está permitido. Cada región del diagrama representa las zonas donde el aeropuerto x_i es el más cercano.

IV. Algoritmos

IV-A. Algoritmo Bowyer-Watson

Este algoritmo permite calcular la triangulación de Delaunay de un conjunto X de puntos en el espacio de n dimensiones. El diagrama Voronoi es el grafo dual de la triangulación, es decir, es un grafo que tiene un vértice por cada región en el grafo original, así como también una arista que une a dos regiones contiguas del grafo original. En este caso, el grafo original es la triangulación mencionada, un conjunto de triángulos con circunferencias circunscritas de tal forma que dichas circunferencias no contienen vértices de otros triángulos.

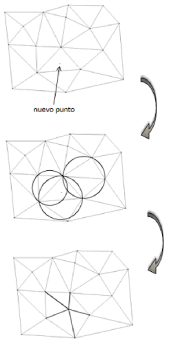

Figura 3: Ejemplol Bowyer- Watson - [6]

Este algoritmo tiene un carácter incremental, se parte de un caso base con una triangulación trivial (un triángulo), a este se le van uniendo puntos. Luego de la unión, se borran los triángulos que no cumplan la condición de circunferencias circunscritas. El pseudocódigo es el siguiente:

```
Bowyer-Watson(X){
    triangulacion= triangulo_trivial
    for x_i in X{
        trash=[];
        for triangulo in triangulacion{
            if(x_i in triangulo.cincunferencia)
                triangulacion.remove(triangulo);
                trash.insert(triangulo);
        }

        poligono=[];
        for triangulo in trash{
            for edge in triangulo{
                if (edge not in (trash - triangulo))
    {
                    poligono.insert(edge);
                }
            }
        }

        for edge in poligono{
            generateTriangulo(edge);
        }
    }

    for triangulo in triangulacion{
        if(triangulo.vertex in triangulacion)
            triangulacion.remove(triangulo);
    }
}
```

Listing 1: Algoritmo Bowyer-Watson

Intuitivamente al iterar de forma anidada a dos niveles sobre triangulación y los puntos en X, se obtiene un complejidad en tiempo cuadrática $O(n^2)$, lo cual es cierto, a menos que dispongamos de algunas herramienta que permitan reducir la complejidad. Lo que se necesita realizar es que en cada iteración del algoritmo se de un nuevo punto k, de esta forma, al insertar k encontrar los triángulos en trash creará los triángulos que contengan a k en sus circunferencias. La conectividad de triángulo nos permite saber que los triángulos en trash serán un componente conectado en el gráfo de adyacencia de la triangulación, por lo que si conoce uno de esos triángulos en trash se puede encontrar los demás realizando un recorrido del gráfo restringido a los triángulos en trash adyacentes. Sin embargo, es necesario un triángulo en trash del cual partir. Intuitivamente sería el triángulo que contiene a k. De esta forma, en el peor caso se tiene $O(n^2)$ y en el mejor caso $\Omega(n \cdot lgn)$

IV-B. Divide and Conquer

Por otro lado, en la búsqueda de un algoritmo con una menor complejidad en tiempo, se hará uso del paradigma Divide-and-Conquer. Se divide el conjunto X original en 2 subconjuntos denominados X_1 y X_2 (Para la división de los subconjuntos, se toma en cuenta el eje de abscisas). Se procederá a calcular los Diagramas Voronoi de cada subconjunto

y luego hacer el merge correspondiente. La lógica obtenida es sumamente similar a la de un MergeSort. Obtenemos lo siguiente:

- Divide: Dividir el conjunto de puntos en 2 subconjuntos
- Conquer: Resolver recursivamente llamando a Voronoi()
- Combine: Ejecutar Merge de las subsoluciones obtenidas.

```

Voronoi(X):

mid= (size of X)/2
X_1 = X[1:mid]
X_2 = X[mid+1:sizeof(X)]

Vor1= Voronoi(X_1)
Vor_2=Voronoi(X_2)

merge(Vor_1,Vor_2)
```

Listing 2: Divide and Conquer - Voronoi

Sin embargo, qué hacer debería hacer exactamente la función merge para que funcione adecuadamente? Los dos Voronois de X_1 y X_2 tendrán un conjunto de aristas en común por alguna región en X_1 y otra región en X_2. A dicho conjunto se le denomina $\sigma(X_1, X_2)$, dicho conjunto está conformado por aristas disjuntas. Ya que se partió el plano con una recta que pasa por la mediana de las abscisas, el plano quedó dividido en π_L y π_R. Teniendo la siguiente propiedad: $Voronoi(X) = (Voronoi(X_1) \cap \pi_L) \cup (Voronoi(X_2) \cap \pi_R)$. Por lo que el merge lo que busca hacer es construir la cadena σ que separa X_1 y X_2. Toda arista de $Voronoi(X_1)$ y $Voronoi(X_2)$ que pase al otro lado de la cadena será eliminada. A partir de ello se tiene el siguiente pseudocódigo para el Merge:

```

Merge(X_1,X_2):

//Linear time convex hull implementation (Melkman's
    Algorithm)
CH1=Convex_hull(X_1)
CH2=Convex_hull(X_2)

FindBridges(CH1,CH2)
Trazar el bisector del bottomBridge
for p=bottomBridge to upperBridge{
    arista=null
    if(arista pertenece X_1){
        arista=p
    }
    else{
        arista=q
    }
}

for e in aristas{
    if(e==p and e pertenece X_2){
        discard(e);
    }
    else if(e==q and e pertenece X_1){
            discard(e);
    }
}
```

Listing 3: Merge

El merge se ejecuta en O(n), tanto por los for's como por los Convex Hulls, debido a ello la recursión queda: $T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$. Esta ecucación de recurrencia es sumamente conocida. Resulta en $O(n \cdot lgn)$

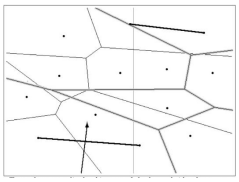

Figura 4: Ejemplol Función Merge - [4]

IV-C. Algoritmo de Fortune

Este algoritmo realiza un barrido de recta a través del plano, mientras va calculando localmente las diferentes regiones $V(x_i)$ conforme la recta se mueve de un lado del plano hacia el otro. La recta intersecta las diferentes regiones asociadas a los puntos, cuando esto sucede se guarda información al respecto. Este algoritmo se apoya de un priority-queue para almacenar los puntos donde la recta se detiene así como también un Binary Search Tree para guardar los elementos de intersección con la recta, esta información sirve para cálculo futuro.

El uso de parábolas es útil en este algoritmo porque al usar línea de barrido para cualquier punto x_i, existe una parábola que separa x_i de la línea de barrido, tal que cada punto de la parábola es equidistante a x_i y a la línea de barrido. De esta forma, cuando la línea de barrido alcance un nuevo punto x_j, se puede afirmar, en base a la parábola asociada a cada punto, cuál es el punto medio entre x_i y x_j. En este caso se hará un barrido horizontal abajo hacia arriba, creando una parábola para todo punto en X tal que cada punto sea el foco de su parábola, y la directriz de la parábola sea la línea de barrido en sí.

Así luce gráficamente el Algoritmo de Fortune:

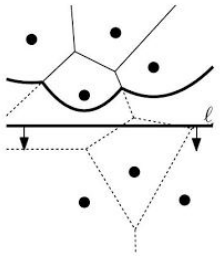

Figura 5: Ejemplo Algoritmo Fortune - [8]

Para una correcto entendimiento es necesario definir dos conceptos utilizados por Fortune.

- Evento de sitio: Corresponde a agregar una nueva arista al diagrama. Estos son nuestros eventos originales
- Evento de círculo: Corresponde a agregar un nuevo vértice al diagrama. Estos se agregan dinámicamente conforme se procesan eventos tipo sitio.

El pseudo-código del algoritmo es el siguiente:

```
Fortune(X):
    Q= empty Queue
    BST = empty Binary Search Tree
    Sort(X, X.x) //Ordenar X por la abscisa con costo O(n lg n) usando merge sort

    for x_i in X:
        Q.insert(x_i)

    While !(Q.empty()){
        q=Q.pop()

        if(q is evento sitio){
            if(T.root= null){
                T.root=q;
            }
            B = arcGenerated(x_i, parabola G)
            BST.insert(B);
            split(B); //B1 y B2
            Q.removePosibilities(q);
            Q.addPosibilities(q);
        }

        else{ //q is evento circulo
            Almacenar el vertice q del diagrama Voronoi
            BST.removeArc(q)
            Q.removePosibilities2(q);
            Q.addPosibilities2(q);
        }

    }
```

Listing 4: Fortune Algorithm

La posibilidad que se eliminan en removePosibilities son las que no consideraban el evento círculo dentro de Q con q como parte del evento círculo y las que se agregan en addPosibilities son aquellas que contemplan eventos círculo con la parábola G generada así como también las combinaciones que contienen a B_1 y B_2. Las posibilidades que se eliminan en removePosibilities2 son aquellas que contenían al vértice q dentro de una secuencia de Q, y las que se agregan en addPosibilities2 son aquella que se generan al omitir q en Q, es decir, el elemento anterior con el elemento siguiente. Como se puede visualizar, las operaciones de mayor complejidad son los procesamientos de eventos, esto se ejecuta en $O(lgn)$, e.g, eliminación del árbol binario y operaciones en la cola de prioridad. Ya que estas operaciones se ejecutan para una cantidad de n puntos/eventos, la complejidad total del algoritmo de Fortune es $O(n \cdot lgn)$

REFERENCIAS

[1] Sacristán, V. (n.d.). Intersecting half-planes and related problems.
[2] Sacristán, V. (n.d.). Diagrams, Algorithms for constructing Voronoi. Barcelona.
[3] Quach, T. (
[4] 2012). Spatial Analysis in PostGIS base on Voronoi diagram/Delaunay triangulation.
Barr, V., Siegelmann, H., Sarkozy, G., Horn, M., & Weber, J. (2004). Computational Geometry Lecture Notes Voronoi Diagrams.
[5] Bowyer, A. (1981). Computing Dirichlet tessellations. https://doi.org/0.1093/comjnl/24.2.162
[6] Watson, D. F. (1981). Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes.
[7] de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). Computational Geometry: Algorithms and Applications. Springer-Verlag Berlín Heidelberg.
[8] Fortune, S. (1986). A sweepline algorithm for Voronoi diagrams. Proceedings of the Second Annual Symposium on Computational Geometry, 313–322.
[9] Universitat Politècnica de València UPV - Derivando. Recuperado de: https://www.youtube.com/watch?v=wCmwBHfiIxQ&ab_channel=Derivando
[10] Transum.org, Voronoi Diagrams. Recuperado de: https://www.transum.org/Maths/Activity/Graph/VoronoiDiagram.asp