UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES

FACULTAD DE CIENCIAS EXACTAS

INGENIERIA DE SISTEMAS

PROBLEMA DE LA GALERIA DE ARTE

ANÁLISIS Y DISEÑO DE ALGORITMOS II

TRABAJO FINAL

Profesoras: Liliana Martínez - Claudia Pereira

Alumnos: Báez, Brenda
Bertino, Ariel Eugenio

Emails: baezbrenda0@gmail.com
eugenioingenio10@gmail.com

INDICE

RESUMEN

El presente informe forma parte del trabajo final de la cátedra Análisis y Diseño de Algoritmos I, correspondiente a la carrera Ingeniería de Sistemas de la Facultad de Ciencias Exactas de la Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN).

El objetivo del trabajo es desarrollar una serie de algoritmos en C++ que permita resolver el problema de la galería de arte. Es decir, dado un polígono o un conjunto de puntos (que representan la sala de galería de arte), el problema tiene como objetivo encontrar la menor cantidad guardias o cámaras (puntos) de modo que vigilen toda la sala.

El problema pertenece a NP-Hard para un conjunto de puntos, esto nos dice que no se puede encontrar una solución en un tiempo polinomial y que además no se puede encontrar un algoritmo de aproximación con factor constante.

En esencia nuestro algoritmo se basa en las ideas de Chvátal (1975) y Fisk (1978). Quienes respectivamente demuestran que son suficientes hasta $\left\lfloor \frac{n}{3} \right\rfloor$ guarias para vigilar cualquier polígono no convexo (con n cantidad de vértices del polígono) y que, para llegar a esto a partir de un polígono P, se lo debe triangular (descomponer en triángulos), colorear el grafo de esa triangulación con tres colores y ubicar los guardias en los puntos con el color que menos se repita.

Cabe aclarar que se utilizó el framework Visual Studio para el desarrollo del programa y MatLab para implementar la interfaz gráfica

OBSERVACION GENERALES

Cuando de grafos hablamos, nos referimos a un modelo conformado por una serie de nodos o vértices conectados o no entre ellos mediante una serie de arcos o aristas. Existen varios tipos de grafos, con diferentes funcionalidades y características. Nosotros utilizaremos un grafo llamado grafo no dirigido.
Formalmente:
G = (V, E), donde
V: es un conjunto de vértices
E: es un conjunto de aristas, donde cada arco {i, j} representa una unión del vértice v_i al v_j donde {i, j} = {j, i}.
Cabe destacar que este grafo posee varias formas de ser representado, más adelante especificaremos que forma utilizamos.
Cabe aclarar que cuando nos referimos a un grafo que forma un árbol, Hacemos alusión a un grafo especial que no posee ciclos o bucles mediante una serie de arcos.
Llamamos grado de un nodo a la suma de arcos que salen y entran desde y hacia él.

ANALISIS DE LAS POSIBLES SOLUCIONES

Para poder generar una solución factible consideramos los siguientes pasos, a partir del polígono que representa la galería de arte. Anexaremos documentos para cada paso con el fin de tener la información mejor organizada. Para ir a un anexo en concreto presionar la tecla Ctrl+Click en el anexo que desee acceder

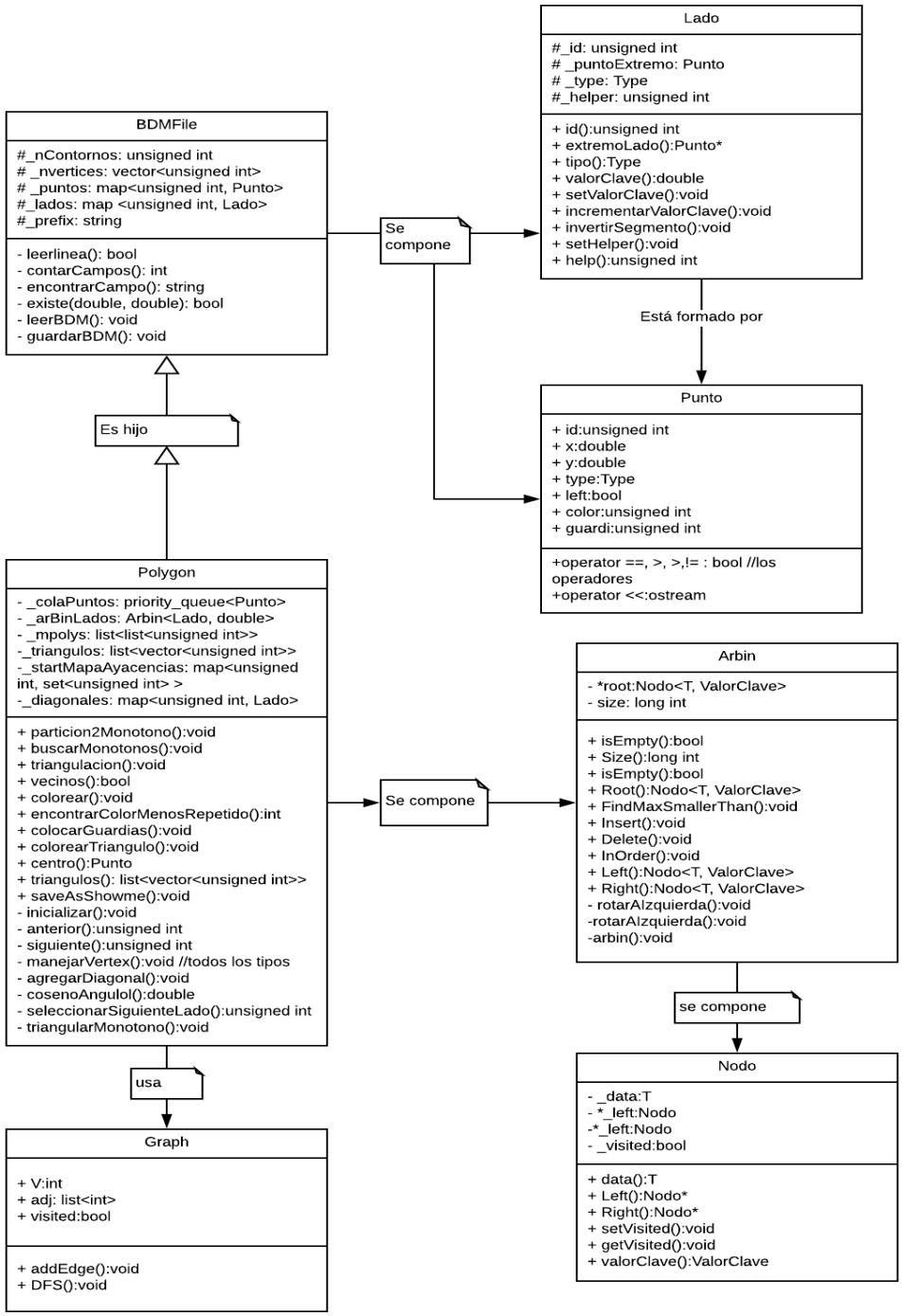
I. Triangular el polígono inicial. (Anexo I)
II. Generación del grafo dual (árbol de recorrido). (Anexo II)
III. Generar una coloración triangulación anterior. (Anexo III)
IV. Ubicar los guardias en el color que menos repeticiones.

Este apartado consiste en agregar una marca a los vértices con el color de menos repeticiones, indicando que tiene un guardia. Estos guardias desde ese vértice vigilas todos los triángulos que contenga/n ese/os vértice/s.

V. Variantes del problema de la galería de arte. (Anexo V)

TIPO DE DATOS Y ESTRUCTURAS

A continuación, describiremos los diferentes tipos de datos involucrados en la solución como así también las estructuras subyacentes de los mismo con su funcionalidad como también la complejidad temporal de los algoritmos que dan paso a la solución. Para tener una visión general de lo de estos aspectos mostramos inicialmente el diagrama de clases con todas las clases involucradas en el proyecto. Cabe aclarar que algunas de los tipos de datos utilizados son parametrizados, esto significa que se puede definir tipo que contenga otros tipos de datos en su interior con los cuales trabajar. Los tipos parametrizados del proyecto son: Nodo, Arbin



- Nodo: Este nodo posee como estructura un dato(dato), dos apuntadores dos nodos del mismo tipo; uno izquierdo y otro derecho (izq y der), como así también una variable booleana(visitado) que nos indica si ese nodo esta visitado o no. La funcionalidad que posee es la siguiente:
  - Nodo (const T & data, Nodo *lt, Nodo *rt): este el constructor de la clase permite crear nodos de este tipo, y como parámetro posee, un dato, un puntero a nodos izquierdo y derecho.
  - T& data (): este método retorna el dato que tiene el nodo
  - Nodo* Left () y Nodo* Right (): retornan como resultado los punteros a nodo izquierdo y derecho respectivamente.
  - void SetVisitado (bool visited): cambia el estado visitado del nodo al estado pasado por parámetro 'visited';
  - bool GetVisitado (): obtiene el estado del nodo almacenado en la variable 'visitado' del nodo
  - KeyType valorClave (): este método se encarga de retornar el valor de clave que posee el dato, este método será de utilizado más adelante, cuando se lo parametriza, por ejemplo, con un punto el cual posee un valor de clave. Se detallará más adelante.

- Arbin: Se trata de un árbol binario compuesto en su estructura por un puntero a su raíz que es un nodo de tipo Nodo. A demás posee una variable que nos brinda su tamaño o cantidad de nodos llamada size, que aumenta y disminuye con cada alta y baja de nodos en el árbol. En lo que respecta a la funcionalidad tenemos lo siguiente:
  - Formas de recorrer en árbol que son InOrder (), PreOreder () y PostOrder (), los cuales lo recorren de diferentes formas. Todos ellos poseen una complejidad temporal perteneciente al orden de O(n), con n=cantidad de nodos.
  - void vaciar (): este método es invocado por el destructor de la clase, recorre todo el árbol liberando la memoria ocupada por cada nodo, su complejidad: O(n), n=cantidad de nodos.
  - bool estaVacio () const: consulta si el árbol está vacío, preguntando si la raíz es esta vacía y retornando esa respuesta. Posee complejidad O (1) complejidad constante ya que se verifica directamente mediante una variable.
  - long int Size (): retorna la cantidad de accediendo a la variable privada size, por lo tanto, tiene complejidad O (1).
  - Nodo<T, KeyType>* Root (): retorna el puntero a la raíz del árbol accediendo a la variable privada root, haciendo que sea una operación de complejidad constante O (1).
  - void hallar (const KeyType keys, Nodo<T, KeyType>* & res): encuentra el nodo con la clave keys dada por parámetro, si no existe retorna null, este lo devuelve en el parámetro res. En el peor de los casos deberá recorrer la rama más larga, por consiguiente, la complejidad temporal de este método es O (log n), con n cantidad de nodos del árbol.

- void hallarMin (Nodo<T, KeyType>* &min): este método retorna el nodo con el menor valor, recorre el árbol siempre desplazándose hacia el nodo izquierdo, por lo tanto, su complejidad es de orden logarítmico ya que a media que se adentra en el árbol buscado por cada nodo que recorre descarta la mitad de los nodos y es O (log n).
- void hallarMax (Nodo<T, KeyType>* &max): funcionamiento análogo al método anterior, pero buscando el mayor. Siguiente su complejidad es O (log n).
- void FindMaxSmallerThan (const KeyType& keys, Nodo<T, KeyType>* &res): encuentra y retorna el nodo más grande con la clave más pequeña que la indicada por el parámetro 'keys'. La complejidad ronda el orden de O (log n)
- void Insert (const T & x): recorre el árbol en base a la clave del valor que se le pasa por parámetro. Si la clave es menor o mayor a la clave del nodo actual del árbol recorreré hacia izquierda o derecha respectivamente hasta llegar a una hoja que es donde inserta el nodo. Cada vez que se adentra el árbol deja la mitad de los nodos ya sea derechos o izquierdos de lado por lo tanto al llegar a la posición de inserción alcanzara una complejidad temporal de O (log n).
- void Delete (KeyType keys): elimina el nodo con clave dada, posee una complejidad de O (log n), con n+ numero de nodos.
- void borrarMin (Nodo<T, KeyType>* &min): este método se encarga de encontrar y eliminar el nodo con la clave más pequeña. Conlleva una complejidad temporal perteneciente a O (log n).
- void borrarMax (Nodo<T, KeyType>* &max): similar funcionamiento que el método anterior aplicado al nodo con la clave más grade, también de complejidad O (log n).
- const Arbin & operator== (const Arbin & rhs): se define un operador de asignación por copia. Realiza una copia el árbol que proviene del parámetro 'rhs' y lo asigna a la raíz del árbol actual. En el peor de los casos este método deberá copiar todo el árbol por lo tanto su complejidad temporal ronda el orden O(n), con n cantidad de nodos del árbol.
- int altura () const {return altura(root): este método devuelve la altura total de árbol, va comparando la altura de los dos subárboles, por lo tanto, la complejidad temporal es del orden de O(n). n nodos en el árbol.
- int altura (Nodo<T, KeyType> *t) const: este método en esencia funciona igual que el anterior, pero retorna la altura del árbol que se le pasa por parámetro, en el peor de los casos retorna la altura de todo el árbol. Complejidad temporal de O(n).
- Nodo<T, KeyType>* Left (Nodo<T, KeyType> *node) y Nodo<T, KeyType>* Right (Nodo<T, KeyType> *node): retornan el puntero o a lo que apuntan el hacia la izquierdo y derecha respectivamente del nodo pasado por parámetro, como solo es una sentencio de retorno su complejidad en constante, O (1).

- Nodo<T, KeyType> * clone (Nodo<T, KeyType> *t) const: clona o copia el árbol pasado por parámetro, a lo sumo clonara todo el árbol, por lo tanto, posee O(n) como complejidad temporal con n numero de nodos en el árbol 't' como parámetro.
- void rotarConHijoIzq (Nodo<T, KeyType> * & k2) const: este método intercambia en puntero del nodo pasado por parámetro con el putero de su hija izquierdo, genera el efecto de rotar cono padre con hijo izquierdo. Consta de asignaciones por lo tanto posee complejidad constante, O (1).
- void rotarConHijoDer (Nodo<T, KeyType> * & k1) const: similar al método explicado con anterioridad, pero rotando el nodo con su hijo derecho.
- void distribuir (KeyType keys, Nodo<T, KeyType> * & t) const: este método cumple la función de inserta nodo en el árbol. Para esto busca la posición del nodo a inserta mediante las claves de estos. Si la clave del nodo que yo quiero insertar es menor a otra clave pasada por parámetro avanza por la rama de los hijos izquierdos(menores) de lo contrario avanza por derecha(mayores). Cuando encuentra la posición li inserta actualizando los punteros pertinentes. Al avanzar por cada nodo del árbol, de ser necesario se debió haber rotado la posición actual con el hijo derecho o izquierdo. Para la complejidad temporal tenemos que en el peor de los casos recorre n nodos, entonces, su complejidad es O(n), n nodos en el árbol.

- Graph: es una clase que representa un grafo dirigido. Posee un entero que se refiere a la cantidad de nodos que tiene el grafo, un arreglo de listas para representar las conexiones o arcos entre los nodos y un arreglo de visitados de tipo booleano que nos dice si el nodo en cuestión ya fue visitado o no en este caso cuando se recorre de una manera específica.
  - Graph (int V): él es constructor de la clase. Toma la cantidad de nodos del parámetro 'V'. inicializa un arreglo de listas de adyacencias con esa misma cantidad de nodos. De la misma forma inicializa un arreglo de visitados de ese tamaño y por último marca todos los nodos como no visitados en el arreglo previamente nombrado. Inicializar la lista de adyacencia y asignar a todos los nodos el estado de no visitado supone una complejidad temporal de O(n), siendo n la cantidad de nodos.
  - void addEdge (int v, int w): este método se encarga de agregar un arco desde el nodo v al w, /con complejidad O (1).
  - void DFS (int v, vector<int> &V, int &i): método encargado de recorrer en grafo en profundidad, esto significa que siempre recorre tratando de avanzar por el primer adyacente de cada nodo. Inicia desde el nodo 'v' pasado por parámetro y va recorriendo todo el grafo. En general la complejidad de este recorrido es O(V+E), donde V es la cantidad de nodos y E es la cantidad de arcos.
- Punto: es una clase que representa un punto, pero con algunos agregados que lo adaptan al problema de la galería de arte. Para empezar, dispone de coordenadas x e y, un identificador del punto, un tipo de utilidad a la hora de operar en la triangulación, un atributo booleano left que dice si pertenece o no a la cadena

izquierda, color de tipo entero asignándose en el coloreo, y guardia que nos dice si ese punto tiene o no un guardia asignado.
  - Punto (), Punto (const Punto& pb): constructores de la clase vacío y por copia. El constructor por copia se encarga de generar un punto a partir de otro dado por parámetro, copiando los valores dados al nuevo punto.
  - Punto (double xx, double yy): constructor a partir de coordenadas dadas. Además, asigna como al tipo el valor desconocido.
  - Punto (int idd, double xx, double yy), Punto (double xx, double yy, Type ttype), Punto (int idd, double xx, double yy, Type ttype): constructores varios.
  - bool operator==(const Punto& pa, const Punto& pb) y bool operator!=(const Punto& pa, const Punto& pb): sobrecarga del operador de igualdad y desigualdad, para poder comparar puntos.
  - bool operator>(const Punto&, const Punto&) y bool operator<(const Punto&, const Punto&): comparadores mayor y menor, basados en la coordenada y. si la coordenada y es la misma en ambos putos compara la coordenada x.
  - ostream &operator<< (ostream &os, const Punto& point): es un operador de salida, sirve para mostrar el punto por pantalla.
- Lado: o segmento dirigido desde cabeza hacia la cola. Se define como dos puntos y tipo. Posee los siguientes atributos: un id entero, un tipo (si es de entrada o si es a insertar), una clave que se usa en un árbol de búsqueda, y un helper.
  - Lado (), Lado (Punto* ep1, Punto* ep2, Type type), Lado (const Lado& line): son diferentes constructores de la clase.
  - unsigned int id () const: retorna el id del segmento.
  - Punto* extremoLado (int i) const: retorna el extremo del segmento.
  - Type tipo () const: devuelve el tipo de segmento.
  - double valorClave () const: brinda la clave del segmento.
  - void setvalorClave (double y): permite setear el valor de la clave del segmento.
  - void incrementarvalorClave (const double diff): este método aumenta la clave del segmento. Esto es una forma de no tener claves repetidas en el árbol de búsqueda.
  - void invertirSegmento (): el efecto de este método es invertir el segmento. si el segmento iniciaba en a y finalizaba en b, ahora se intercambian estos valores.
  - Void setHelper (unsigned int i): establecer el helper de un lado;
  - unsigned int helper (): retornar el helper del segmento
  - ostream &operator<<(ostream &os, const Lado& line): sobrecarga del operador de salida para poder visualizar el segmento.
- BDMFile: es clase cumple el rol de archivo de contornos con el cual de crea el polígono original. Se generan en esta clase; una variable _ncontornos que guarda el número de contornos del polígono, un vector(_nVertices) con la cantidad de vértices. Un mapa con un valor asociado a un punto, otro mapa con un valor asociado a un lado, valores de x e y mínimos y máximos que representan la caja que encierra al polígono inicial, por último, un variable que cumple el rol del nombre del archivo que eventualmente se leerá o se escribirá.

- Int contarCampos (const string& source, char delim): retorna la cantidad de campos del que posee una cadena de texto ‘source’, separadas por un delimitador ‘delim’. Complejidad O(n) con n cantidad de caracteres de la línea.
- Bool leerlinea (ifstream& nombreArchivo, string& result): desde un archivo, extrae las líneas de este. Por cada línea cuenta los campos, en particular que sean números, retorna si es o no el fin del archivo. Su complejidad temporal pertenece al orden de O(n), en este caso n representa la cantidad de líneas del archivo o el tamaño de este.
- string encontrarCampo (string& source): retorna de la línea la cadena restante luego de haber salteado, caracteres innecesarios (espacios en blanco, cualquier cosa que sea un numero o comentario), eventualmente quedara un campo, que es retornado. Como consideramos que una línea puede tener n caracteres y debe ser procesada toda la línea, la complejidad temporal asciendo a O(n).
- bool existe (double x, double y): a este método se le entrega como parámetro una coordenada x e y con el fin de ver si existen puntos duplicados en el mapa de puntos generado previamente. Complejidad temporal de O(n) con n siendo en número de puntos en el mapa de puntos.
- void guardarBDM (float * in, int N): genera un archivo que proviene del exterior. Guarda números línea a línea. Se puede entender como n al tamaño de archivo exterior, entonces O(n).
- void leerBDM (const string& nombreArchivo): a partir de un archivo, guarda el número de vértice, controla que la cantidad de vértices y contornos sea adecuada para generar un polígono. Genera los lados del polígono uniendo puntos contiguos. O(n) siendo n la cantidad de líneas del archivo ‘nombre de archivo’.
- void leerBDM (const string& nombreArchivo, bool parse): funcionamiento similar al método anterior, pero sin procesar la entrada.

- Polygon: la clase polygon tiene como estructura una cola de prioridad de puntos, lo que genera un orden preestablecido para los puntos del polígono, luego un árbol binario de búsqueda de lado, una lista de polígonos monótonos en esencia es una lista de listas de enteros para contener dichos polígonos, además esta clase trabaja con una lista de triángulos que está definida como una lista de vectores de elementos enteros. Además de las 4 estructura previamente descritas, esta clase dispone de dos estructuras adicionales para descomponer los polígonos monótonos; estas son un mapa de adyacencias donde cada elemento está compuesto por un entero y un conjunto de enteros, básicamente un entero como clave(única) para cada conjunto y por último un mapa de lados donde a cada clave única(entero), se le asigna una lista de lados. Cabe aclarar que la clase polígono hereda de la clase BDMFile, con el objetivo de bridarle comporta extra relacionado al almacenamiento de los resultados en archivos. A partir de toda esta estructura interna definimos varios métodos para su manipulación, los siguientes:

- unsigned int anterior (unsigned int i): retorna el id del punto anterior o bien, dado un punto o lado retorna un lado anterior. Complejidad O(n) ya que son n vértices.
- unsigned int siguiente (unsigned int i): ídem al método previamente descrito, pero retorna un punto o lado siguiente.
- double orient2d (double* pa, double* pb, double* pc): dados tres puntos que conforman un triángulo, retorna el arreglo de este. Al ser solo asignaciones posee complejidad constante, O (1).
- Polygon (string nombreArchivo, bool parse=true): se trata del constructor de polígono, que invoca al método inicializar. Este método se encargar de recorrer el mapa de puntos clasificándolos en de 6 tipos (regular_up, reglar_down, start, end, Split o merge). Le determinación del tipo de punto se hace comparan un punto actual con el puno siguiente y anterior al susodicho punto y realizando unas verificaciones. Entonces el punto actual es:
  Regular_down: si mayor al siguiente y menor que el anterior.
  Regular_up: si es mayor al anterior y menos al siguiente
  End: si el anterior y el siguiente son mayores y además el área comprendida por los tres puntos es mayor a cero
  Merge: misma condición que la anterior, pero si el área es igual o menor acero.
  Start: si al punto actual es menos al anterior y al siguiente y además el área del triángulo que forman esos puntos es mayo a cero.
  Split: igual condición que la anterior pero que el área del triángulo sea menor o igual a cero.
  Para concluir la construcción del polígono se van agregando los puntos ya clasificados tanto a la cola de puntos como al mapa de adyacencias.
- ~Polygon (): destructor de la clase, a la vez que recorre el mapa de puntos va liberando la memoria, de manera similar libera la memoria utilizada por el mapa de puntos. Complejidad O(n) con n siendo el número de lados o punto.
- void agregarDiagonal (unsigned int i, unsigned int j): Agregar una diagonal del punto i al punto j. Esa diagonal será de tipo insert y se agregan esa diagonal como adyacente entre vértices i y j. son accesos directos a las posiciones a insertar, por lo tanto, la complejidad es de O(n) con n numero de vértices en el mapa de adyacencias '_startMapaAdyacencias'.
- double cosenoAngulo (double *A, double *B, double *C): Calcula el ángulo a partir de tres puntos A, B, C. El origen de los dos puntos es A y los extremos son B y C, forman dos segmentos a los que puede calcularse el ángulo. Solo ejecuta operaciones aritméticas, por lo tanto, su complejidad es de O (1).
- unsigned int seleccionarSiguienteLado (Lado* edge): Para cada lado, encontrar el lado siguiente que deberíamos elegir cuando buscamos una porción monótona. En este proceso recorre el mapa de adyacencias del lado al cual queremos encontrar el siguiente lado. Complejidad O(n) n número de puntos adyacentes al lado.

- Void manejarStartVertex (unsigned int i): este método se encarga de procesar vértices de tipo start. Recorre el árbol de lados de manera In Order para luego insertar el lado que posee el vértice start. O(n), n nodos en el árbol.
- void manejarEndVertex (unsigned int): primer asigna el valor del vértice anterior a i en una variable, asigna su lado a otra variable. Si el punto de ese lado es de tipo merge se agrega un diagonal desde al vértice inicial hasta el vértice de ese lado, luego se elimina la clave de ese lado del árbol de lados. O(n), n elementos de árbol.
- Void manejarMergeVertex (unsigned int i): Método para decidir qué hacer con puntos de tipo merge. A partir de un punto dado, verifica si el punto del lado anterior en de tipo también merge, si lo es traza una diagonal y borra el lado anterior del árbol de lados. También trata de trazar otra diagonal buscando el lado mayor, que es menor a la coordenada x del punto pasado por parámetro inicialmente. O(n) n nodos en el árbol de lados.
- Void manejarSplitVertex (unsigned int i): procesamiento de vértice de tipo Split. Halla mayor nodo que es menor que el vértice de entrada. Toma el lado consecutivo y agrega una diagonal, luego agrega ese lado al árbol de lados. O(n) recorre en el peor de los casos todo el árbol.
- void manejarRegularVertexUp (unsigned int): similar proceso que los anteriores, verifica que el extremo del lado izquierdo sea de tipo merge. Si es así traza una diagonal, pero esta vez no guarda el nuevo lado. O(n) proveniente del agregarDiagonal.
- void manejarRegularVertexDown (unsigned int): ídem anterior pero esta vez sí guarda el nuevo lado en el árbol de lado.
- Void particion2Monotono (): para empezar en método se lleva a cabo si el primer punto de la cola de puntos es de tipo start y a su vez no hay punto repetidos. Ejecuta determinada acción según el tipo de vértice. Se realiza en un tiempo de O(n) en peor casa debe inserta nodos en el árbol de lado.
- Punto centro (Triangulo t): Esta función calcula el centro del triángulo, este va a ser el representante del de ese triangulo, cuando se le aplique el DFS para encontrar el orden de recorrido, para su posterior coloreo. O (1) solo son asignaciones y operaciones aritméticas.
- ListaTriangulo triangulos (): retorna la lista de triángulos. O (1).
- int encontrarColorMenosRepetido (): recorre el grafo del polígono inicial y cuenta cuantos colores hay de cada uno. Retorna el que se repite más, O(n), con n cantidad de nodos en el grafo.
- void colocarGuardias (int color): después de saber qué color se repite menos recorre el grafo preguntando si el color de cada nodo es el que monos se repite, si es le asigna un guardia. O(n) n nodos en el grafo.
- bool vecinos (Triangulo t1, Triangulo t2): dados dos triángulos por parámetro, retorna un booleanos indicando si son vecinos. Solo realiza comparaciones por lo tanto su complejidad es O (1).
- void colorear (): arreglo contiene la información, el grafo contiene la estructura donde nos vamos a mover. Una vez que tenemos todos los

elementos, los agregamos al grafo (que es de int, lleva referencia al arreglo). Tenemos un grafo es de tipo int y se construye con la variable que contaba la cantidad de vértices. Una vez que colocamos los triángulos en el arreglo, buscamos los vecinos correspondientes. Ahora hay que construir un grafo que tenga los triángulos y sus centros de masa, a partir de ahí se puede crear la relación a partir de que los triángulos comparten lados, una vez que se tiene el grafo, se puede aplicar el dfs, para colorear en costo O(n), con n el número de triángulos. Creamos los lados del grafo. Podemos ejecutar el DFS desde el nodo raíz. Continuamos coloreando todos triángulos, buscamos el color que menos se repite y colocamos los guardias.

- Void colorearTriangulo (Triangulo &t): método que sirve para ir coloreando los vértices hasta con tres colores. de los triangulos, respetando que a nodos adyacente colores distintos, recorre todo el grafo su complejidad O(n) n cantidad de nodos del grafo.
- void saveAsShowme (): método que se encarga de plasmas lo resultado en los archivos de salida.
- void buscarMonotonos (): O(n), n cantidad de lados
- void triangularMonotono (PoliMonotono& mpoly): O (n log n), n vértices.

Estos últimos dos métodos los explicaremos más adelante

Como último apartado de esta sección se bridará un pequeño seguimiento de los procesos que va realizando en programa para brindar una solución a determinada entrada.

A partir de un polígono almacenado previamente en un archivo, se lo ingresa y se lo define dentro del programa. Luego se llama al método triangulación sobre ese polígono. Luego se lo va particionando o dividiendo en subpoligonos monótonos. Esto se puede realizar ya que se clasifican los vértices (explicado más arriba). Con esta clasificación se puede ir particionando el polígono. Una vez que se tiene el polígono ‘descompuesto o fraccionado’ en polígonos monótonos más pequeños. A continuación, debemos buscar esos polígonos monitos con el fin de guardarlos en una estructura de lista de listas para así tenerlos organizados. Pasamos al punto de tener que recorrer esta estructura para ir triangulando los polígonos. Una vez que se logró triangular (descomponer el polígono inicial en triángulos). Pasamos a la etapa de coloreo

Obtenemos los triángulos del polígono los empezamos a recorrer un arreglo que posee la información de los triángulos y grafo que es la estructura donde nos vamos a mover, generamos los centros de todos los triángulos. Generamos un grafo con la cantidad de vértices inicial y desde el arreglo buscamos los vecinos correspondientes de cada triangulo (son vecinos si comparten un lado). Al grafo le cargamos los triángulos y sus centros de masa. Con estos centros de masa se arma un grafo dual acíclico. Sobre este grafo aplicamos un recorrido en profundidad(DFS), que nos genera el orden o recorrido de menos longitud para el grafo. Lo que nos entregó este recorrido es el orden de coloreo de tres colores para los nodos que confirman el polígono inicial. Al iniciar este coloreo nos encontramos con un triángulo que se colorea de manera trivial (color

cualquiera para cada vértice), respetando siempre colores distintos para nodos adyacentes. Luego la coloración restante se torna rápida ya que los triángulos comparten lados, por lo tanto, si se colorea un triángulo, con verificaciones simples se determinaría el color de los demás vértices.

Con el coloreo realizado se recorre el polígono contando los colores y tomando el que menos se repite. Con esta información se recorre por última vez el polígono colocando los guardias donde esta ese color menos repetido.

RESOLUCION DEL PROBLEMA DE LA GALERIA DE ARTE

En este punto incluiremos algunos seudocódigos para poder hacer un acercamiento a la complejidad temporal de los algoritmos usados. La implementación puede sufrir unas leves modificaciones que la harán diferir de los seudocódigos, como así también los nombre que utilizamos, pero el razonamiento que seguimos se ve reflejado sin problemas aquí. Antes debemos definir algunos elementos:

V_i: vértice i.
L_i: lado i.
Cada lado L_i tiene un helper(L_i)

Crear Polígono
Triangulación
    particion2Monotono ()
    BuscarMonotonos ()
    TriangularMonotono(P)
Colorearlo
    Generar centros de masa
    Crear el grafo dual de la triangulación
    Calcular vecinos
    Generar aristas en el grafo dual
    Dfs sobre grafo dual
    Coloreo de los triangulos
    Encontrar menos repetido
    Colocar guardias

- **Crear Polígono**: a partir de un archivo de entrada con el polígono, representado con una coordenada cartesiana por línea. Convergerá a una complejidad O(n) con n nuero se puntos.
    1. Leer línea del archivo, extrayendo cantidad de vertices
    2. Por cada línea
        2.1. Se obtiene las coordenadas del vértice
        2.2. Se crea el punto, con su id

3. Recorriendo todos los puntos y en base a su id se crean los lados del polígono

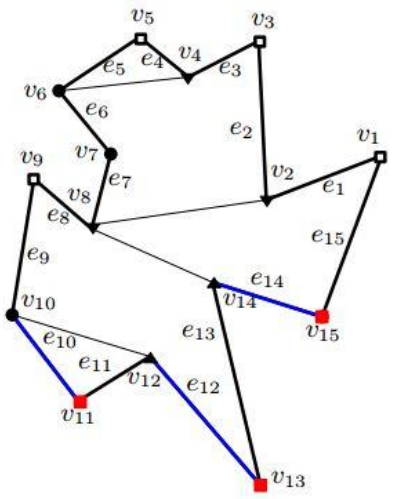
- **Triangulación**: Este algoritmo agrega un conjunto de diagonales que no se intersecan entre sí. Un polígono simple de n vértices, puede ser triangulado una complejidad temporal de O (n log n), con una serie operaciones (no precisamente en este orden) y con complejidades temporales involucradas de:

- Construir una cola de prioridad Q: O(n)
- Inicializar T: O (1)
- Para manejar un evento, realizar:
  - Una operación en Q: O (log n)
  - Al menos una consulta en T: O (log n)
  - Una inserción o borrado en T: O (log n)
  - Insertar al menos 2 diagonales en D: O (1)

Entrada: Polígono simple P (guardado una lista doblemente vinculada D)
Salida: Particiones de P, en partes monótonas guardados en D

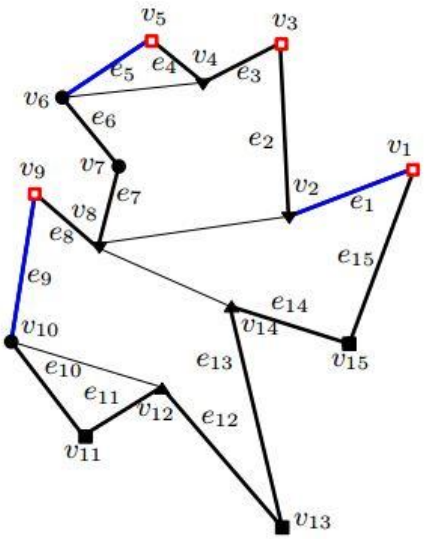
particion2Monotono ()
1. Construir una cola de prioridad Q, con los vértices de P, usando su coordenada 'y' como prioridad. Si dos puntos tienen la misma coordenada 'y', el que tenga la coordenada 'x' más chica, tendrá la prioridad más alta.
2. Se inicializa un árbol binario de búsqueda T.
3. Mientras Q tengo elementos
   I. Se elimina el vértice v_i con la mayor prioridad de Q.
   II. Se invocan los métodos apropiados para el manejo de cada vértice en base a su tipo.

manejarStartVertex (V_i)
1. Insertar L_i en T
2. Asignar el helper (L_i) a V_i

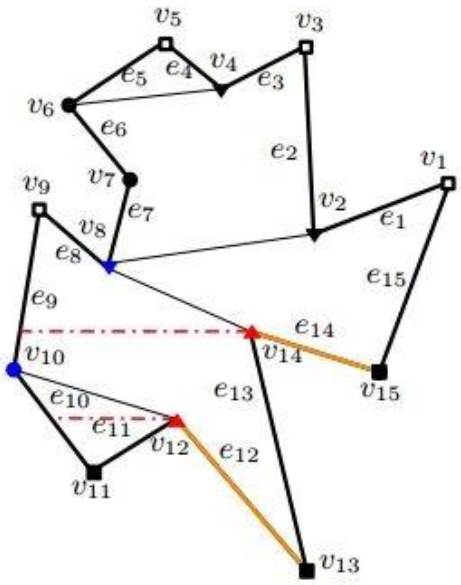


manejarEndVertex (V_i)

1. Si helper (L_{i-1}) en un vértice merge
   Entonces insertar en D una diagonal que conecte V_i con el helper(L_{i-1})
2. Borrar L_{i-1} de T

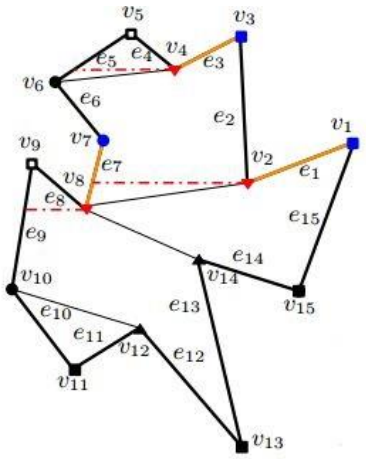


manejarSplitVertex (V_i)
1. Encontrar en T el lado L_j inmediatamente a la izquierda de V_i
2. Insertar en D la diagonal conectado V_i con el helper(L_j)
3. Asignar V_i como helper(L_j)
4. Insertar L_i en T
5. Asignar el helper(L_i) a V_i



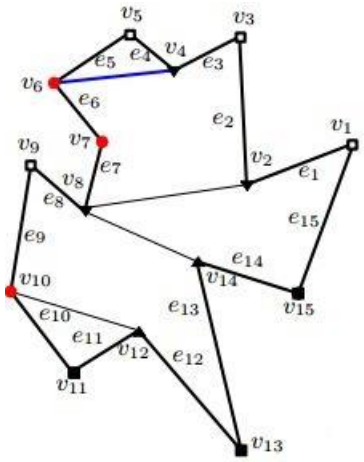
manejarMergeVertex (V_i)

1. Si el helper (L_{i-1}) es un vértice merge
   Entonces insertar en D una diagonal desde V_i a el helper (L_{i-1})
2. Borrar L_{i-1} de T
3. Encontrar en T un lado L_j inmediatamente a la izquierda de V_i
4. Si el helper(L_j) es un vértice merge
   Entonces insertar en D una diagonal desde V_i hacia el helper (L_j)
5. Asignar el helper(L_j) a V_i



manejarRegularVertexUp(V_i)
1. Si P se encuentra a la derecha de V_i
2. Encontrar en T el L_j inmediatamente a la izquierda de V_i
3. Si el helper(L_j) es un vértice merge
   Entonces inserta en D una diagonal desde V_i hacia el helper(L_j)
4. Asignar el helper(L_j) a V_i

manejarRegularVertexDown(V_i)
1. Si P se encuentra a la derecha de V_i y el helper(L_{i-1}) es un vértice merge
   Entonces inserta una diagonal desde V_i hasta el helper(L_{i-1})
2. Borrar de T el L_{i-1}
3. Insertar L_i en T
4. Asignar el Helper(L_i) a V_i



buscarMonotonos ():
Este método se encarga se generar los polígonos monótonos que obtuvo. A medida que avanza en este proceso, va eliminando los lados que hacían no monótono al polígono inicial, generando una lista de polígonos monótonos, que en su unión resultan en el polígono inicial. Estos polígonos se usarán posteriormente en la triangulación. Este proceso, se lleva a cabo con una complejidad del orden de O(n) siendo n el número de lados.

triangularMonotono(P):

Triangular cada parte monótona: un polígono simple con n vértices puede ser triangulado con una complejidad temporal en el orden de O (n log n).
El algoritmo toma lao vértices en orden decreciente en base a la coordenada 'y'. Se requiere una pila S como estructura auxiliar. Esta mantiene los puntos procesados pero que quizás necesiten más diagonales. Cuando tomamos un vértice, agregamos tantas diagonales como sea posible, desde este vértice hacia vértices en la pila.

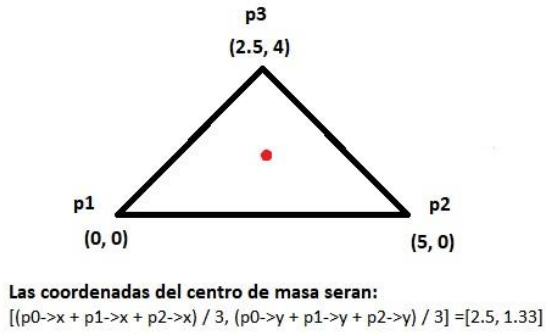
Entrada: polígono monótono P, guardado en D
Salida: una triangulación de P, almacenada en D.

1. intercalar los vértices de la cadena de subida y de bajada de P en una secuencia, ordenada decreciente en base a la coordenada 'y'. tendremos una secuencia V_1, …, V_n ordenada.
2. Inicializar una pila vacía S y apilar en ella loa vértices, V_1 y V_2
3. Para j desde 3 hasta n-1
   I. Si V_j y el tope de S están en cadenas diferentes
      Entonces: quitar todos los vértices de S
      Insertar en D una diagonal desde V_j hacia cada punto quitado, excepto el ultimo
      Agregar los vértices V_{j-1} y V_J en S
      Sino: quitamos un vértice de S
      Quitamos los otros vértices de S, tanta como diagonales desde V_j hacia ellos haya dentro de P. insertar esas diagonales en D. agregar el ultimo vértice que ha sido quitado de nuevo en S.
      Agregar V_j en S.
4. Agregar diagonales desde V_n a todos vértices de la pila exceptuando el primero y el ultimo.

- **Colorearlo:** Para colorear en si el grafo se requiere realizar algunas acciones previas, Cabe aclarar que englobaremos los aspectos necesarios para ya brindar una solución al problema, generar un grafo dual de la triangulación entre otros pasos. Las estructuras que necesitamos son las siguientes;
  - Una lista con los triangulos de la triangulación, operaciones con O(n) n triangulos.
  - Una lista de nodos, con tantos elementos como triangulos haya. Recorridos con O(n), n triangulos. Un nodo posee los 4 centros de masa, uno propio y 3 de los posibles vecinos
  - Un grafo dual, recorridos y operaciones O(n), n triangulos, cada nodo representa el centro de masa de cada triangulo.
  - Arreglo, donde generar el orden para ir coloreando. O(n), n triangulos

Generar centros de masa, O(n), n triangulos

1. Para cada triangulo, elemento del arreglo de nodos
   Calcular centro de masa
   Asignamos un valor nulo a sus vecinos.



Crear el grafo dual de la triangulación, O(n), n triangulos

1. Definimos un grafo, con n número de nodos que representan los n triangulos

Calcular vecinos, O(n), n triangulos

1. Para cada triangulo
   Si comparten dos puntos
   Indicar que son vecinos, en el arreglo
   Sino proseguir con otro vértice

Generar aristas en el grafo dual, O(n), n triangulos, equivalente con n c. de masa

1. Recorremos los nodos del grafo
   Si dos centros de masa son vecinos
   Agregar arista que los una

Dfs sobre grafo dual, O(n), n nodos

Sobre el grafo dual aplicamos recorrido en profundidad, a este método le brindamos como parámetro un arreglo de enteros que corresponden con los id de los centros de masa.

DFS (origen, ordenColoreo, inicio)

1. Estado[origen]→visitado
2. ordenColoreo[inicio] → origen
3. Origen →siguiente nodo
4. Para todo nodo adyacente de origen que no esté visitado
   Realizar DFS (adyacentes de origen, ordenColoreo, inicio+1)

Coloreo de los triangulos, O(n), n triangulos

1. Para cada triangulo asociado, a su centro de más hacer
   1.1. Inicialmente todo sin colorear, por lo tanto, a los nodos del primer triangulo se la asignan colores arbitrarios
   1.2. Obtenemos cuál de los tres nodos del triángulo actual esta sin pintar
   1.3. Generamos la suma de los colores de los nodos ya pintados
   1.4. Si la suma es 3 los vecinos están pintados con 1 y 2, asignar al nodo restante el color 3, sino

1.5. Si la suma es 4 los vecinos están pintados con 1 y 3, asignar al nodo restante el color 2, sino
1.6. Si la suma es 5 los vecinos están pintados con 2 y 3, asignar al nodo restante el color 1

Encontrar menos repetido, O(n), n nodos de polígono
Recorrer los puntos de la triangulación, contando la cantidad de repeticiones de cada color y retornar el que menos se repite.

Colocar guardias, O(n), n del polígono
Se trata de recorrer por última vez el grafo de la triangulación, ya sabiendo que color es el que menos se repite y asignándole a este un 1 a la variable guardia, representado que ese nodo posee guardia.

CONCLUSIONES

ENTRE OTRAS PODEMOS MANDERLE QUE:

ALGO QUE SE TE OCURRA/QUIERAS VOS SI QUERES…. Y

Pero no nos gustaría terminar sin mencionar que el interés del teorema de la galería de arte, y resultados relacionados, no es tan solo por su belleza matemática, sino también por sus aplicaciones en ámbitos como la robótica, la vigilancia, las redes de sensores, la planificación de movimientos, el diseño por ordenador, la captura de modelos digitales, el reconocimiento de patrones, o la arquitectura (asistida por ordenadores).

BIBLIOGRAFIA/REFERENCIAS

https://es.wikipedia.org/wiki/Triangulación_de_un_polígono
https://es.wikipedia.org/wiki/Grafo#Definiciones
https://es.wikipedia.org/wiki/Algoritmo_de_triangulación_voraz
http://www.dma.fi.upm.es/personal/gregorio/grafos/web/mwt/teoria.html
http://www.tamps.cinvestav.mx/~ertello/gc/sesion11.pdf
https://es.wikipedia.org/wiki/Problema_de_la_galerÍa_de_arte
http://www.dma.fi.upm.es/personal/gregorio/grafos/web/iagraph/documents/TFC%20Coloracion.pdf
https://culturacientifica.com/2014/01/29/teorema-de-la-galeria-de-arte/
http://www.cimat.mx/~alram/analisis_algo/05_TriangulacionPoligonos.pdf
http://cs.yazd.ac.ir/farshi/Teaching/CG3952/Slides/Triang.pdf
http://www.dccia.ua.es/dccia/inf/asignaturas/RG/2009-2010/pdf/intro-triang-polig.pdf