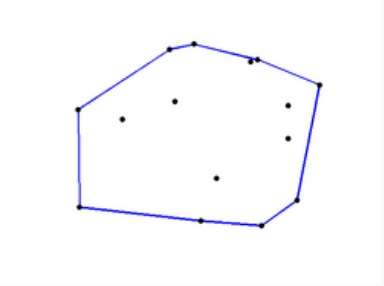
Ayuda para entender: Stanford University ACM Team Notebook (2013-14)

8. Convex Hull

Envolvente convexa



Envoltura convexa de un conjunto de 15 puntos en el plano.

En matemática se define la **envolvente convexa**, **envoltura convexa** o **cápsula convexa** de un conjunto de puntos ***X*** de dimensión ***n*** como la intersección de todos los conjuntos convexos que contienen a ***X***.[1]

Dados k puntos $x_1,\ x_2,\ \ldots,\ x_k$ su envolvente convexa *C* viene dada por la expresión:

$$C(X) = \left\{ \sum_{i=1}^{k} \alpha_i x_i \;\middle|\; x_i \in X,\ \alpha_i \in \mathbb{R},\ \alpha_i \geq 0\,,\ \sum_{i=1}^{k} \alpha_i = 1 \right\}$$

En el caso particular de puntos en un plano, si no todos los puntos están alineados, entonces su envolvente convexa corresponde a un polígono convexo cuyos vértices son algunos de los puntos del conjunto inicial de puntos.

Una forma intuitiva de ver la envolvente convexa de un conjunto de puntos en el plano, es imaginar una banda elástica estirada que los encierra a todos. Cuando se libere la banda elástica tomará la forma de la envolvente convexa.

12. Slow Delaunay triangulation

Triangulación de Delaunay

Una **triangulación de Delaunay**, es una red de triángulos que cumple la **condición de Delaunay**. Esta condición dice que la circunferencia circunscrita de cada triángulo de la red no debe contener ningún vértice de otro triángulo.

Condición de Delaunay

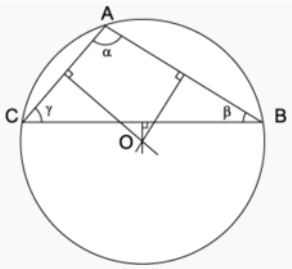


Fig. 4. Los tres vértices A, B, C del triángulo ABC están a la misma distancia del circuncentro O.

La circunferencia circunscrita de un triángulo es la circunferencia que contiene los tres vértices del triángulo.

Según la definición de Delaunay la circunferencia circunscrita es *vacía*, si no contiene otros vértices aparte de los tres que la definen.

La condición de Delaunay dice que una red de triángulos es una *triangulación de Delaunay* si todas las circunferencias circunscritas de todos los triángulos de la red son vacías. Esa es la definición original para espacios bidimensionales. Es posible ampliarla para espacios tridimensionales usando la esfera circunscrita en vez de la circunferencia circunscrita. También es posible ampliarla para espacios con más dimensiones pero no se usa en la práctica.

Esa condición asegura que los ángulos del interior de los triángulos son lo más grandes posible. Es decir, maximiza la extensión del ángulo más pequeño de la red.

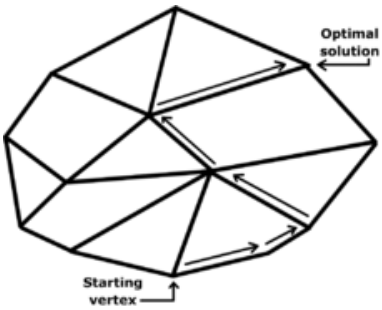
Propiedades

Triangulaciones de Delaunay tienen las propiedades siguientes:

- La triangulación forma la envolvente convexa del conjunto de puntos.
- El ángulo mínimo dentro de todos los triángulos está maximizado.
- La triangulación es unívoca si en ningún borde de circunferencia circunscrita hay más que tres vértices.

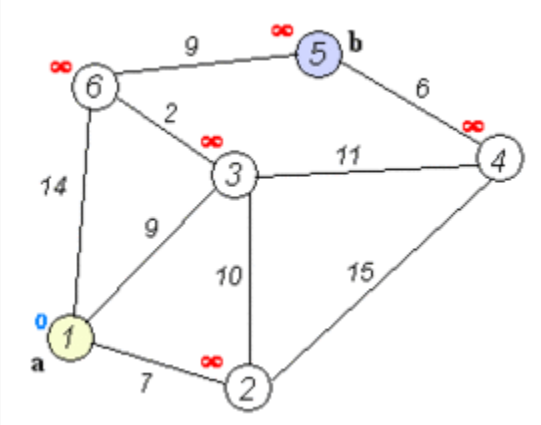
17. Simplex Algorithm

En optimización matemática, el término **algoritmo símplex** habitualmente se refiere a un conjunto de métodos muy usados para resolver problemas de programación lineal, en los cuales se busca el máximo de una función lineal sobre un conjunto de variables que satisfaga un conjunto de inecuaciones lineales. Un algoritmo simplex es un algoritmo de pivote.



Un sistema de desigualdades lineales define un poliedro como una región factible. El algoritmo simplex comienza en un vértice y se mueve a lo largo de las aristas del poliedro hasta que alcanza el vértice de la solución óptima.

18. Dijkstra

Algoritmo de Dijkstra	
	
Tipo	Algoritmo de búsqueda
Problema que resuelve	Problema del camino más corto
Estructura de datos	Grafo

El **algoritmo de Dijkstra**, también llamado **algoritmo de caminos mínimos**, es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de vértices en un grafo con pesos en cada arista.

La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más

corto desde el vértice origen, al resto de vértices que componen el grafo, el algoritmo se detiene. El algoritmo es una especialización de la búsqueda de costo uniforme, y como tal, no funciona en grafos con aristas de coste negativo (al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo).

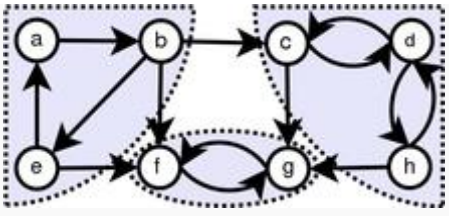
Algoritmo

Teniendo un grafo dirigido ponderado de N nodos no aislados, 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 v_i.
4. Para el nodo actual, calculamos la distancia tentativa desde dicho nodo a sus vecinos con la siguiente fórmula: $dt(v_i) = D_a + d(a,v_i)$. Es decir, la distancia tentativa del nodo 'v_i' es la distancia que actualmente tiene el nodo en el vector D más la distancia desde dicho el nodo 'a' (el actual) al nodo v_i. Si la distancia tentativa es menor que la distancia almacenada en el vector, actualizamos el vector con esta distancia tentativa. Es decir: Si $dt(v_i) < D_{vi} \rightarrow D_{vi} = dt(v_i)$
5. Marcamos como completo el nodo a.
6. Tomamos como próximo nodo actual el de menor valor en D (puede hacerse almacenando los valores en una cola de prioridad) y volvemos al paso 3 mientras existan nodos no marcados.

Una vez terminado al algoritmo, D estará completamente lleno.

19. Strongly Connected Components



Graph with strongly connected components marked

In the mathematical theory of directed graphs, a graph is said to be **strongly connected** if every vertex is reachable from every other vertex. The **strongly connected components** of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected. It is

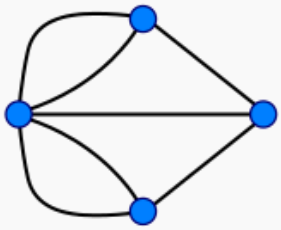
possible to test the strong connectivity of a graph, or to find its strongly connected components, in linear time.

Definitions

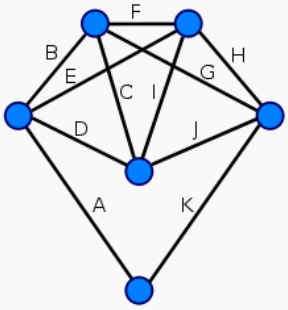
A directed graph is called **strongly connected** if there is a path in each direction between each pair of vertices of the graph. In a directed graph *G* that may not itself be strongly connected, a pair of vertices *u* and *v* are said to be strongly connected to each other if there is a path in each direction between them.

The binary relation of being strongly connected is an equivalence relation, and the induced subgraphs of its equivalence classes are called **strongly connected components**. Equivalently, a **strongly connected component** of a directed graph *G* is a subgraph that is strongly connected, and is maximal with this property: no additional edges or vertices from *G* can be included in the subgraph without breaking its property of being strongly connected. The collection of strongly connected components forms a partition of the set of vertices of *G*.

20. Eulerian Path



The Königsberg Bridgesgraph. This graph is not Eulerian, therefore, a solution does not exist.



Every vertex of this graph has an even degree, therefore this is an Eulerian graph. Following the edges in alphabetical order gives an Eulerian circuit/cycle.

In graph theory, a Eulerian trail (or Eulerian path) is a trail in a graph which visits every edge exactly once. Similarly, an Eulerian circuit or Eulerian cycle is an Eulerian trail which starts and ends on the same vertex.

The term Eulerian graph has two common meanings in graph theory. One meaning is a graph with an Eulerian circuit, and the other is a graph with every vertex of even degree. These definitions coincide for connected graphs.

21. Suffix Arrays

	Caso promedio	Caso Peor
Espacio	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Construcción	$\mathcal{O}(n)$	$\mathcal{O}(n)$

En Ciencias de la Computación un arreglo de sufijos es un arreglo ordenado de todos los sufijos de una cadena dada. Esta estructura de datos es muy simple, pero sin embargo es muy poderosa y es usada en algoritmos de compresión de datos.

Ejemplo

Consideremos el texto $S = banana\$$ a ser indexado:

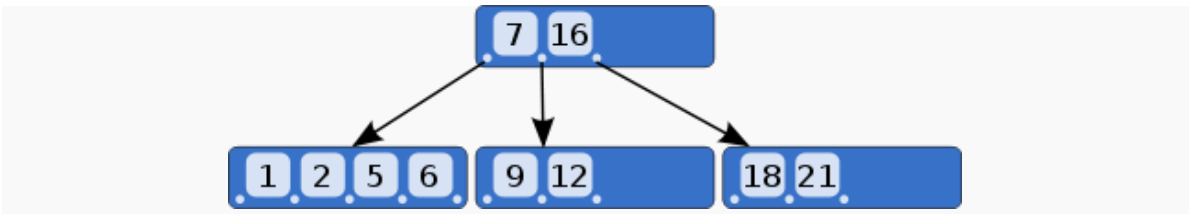
i	1	2	3	4	5	6	7
S[i]	b	a	n	a	n	a	$

El texto termina con el carácter especial $ el cual debe ser único dentro de la cadena y lexicográficamente más pequeño que cualquier otro carácter. El texto contiene los siguientes sufijos:

Suffix	**i**
banana$	1
anana$	2
nana$	3
ana$	4
na$	5
a$	6
$	7

22. Binary indexed Tree

Overview



A B-tree of order 2 (Bayer & McCreight 1972) or order 5 (Knuth 1998).

In B-trees, internal (non-leaf) nodes can have a variable number of child nodes within some pre-defined range. When data is inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may be joined or split. Because a range of child nodes is permitted, B-trees do not need re-balancing as frequently as other self-balancing search trees, but may waste some space, since nodes are not entirely full. The lower and upper bounds on the number of child nodes are typically fixed for a particular implementation. For example, in a 2-3 B-tree (often simply referred to as a 2-3 tree), each internal node may have only 2 or 3 child nodes.

Each internal node of a B-tree will contain a number of keys. The keys act as separation values which divide its subtrees. For example, if an internal node has 3 child nodes (or subtrees) then it must have 2 keys: a_1 and a_2. All values in the leftmost subtree will be less than a_1, all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2.

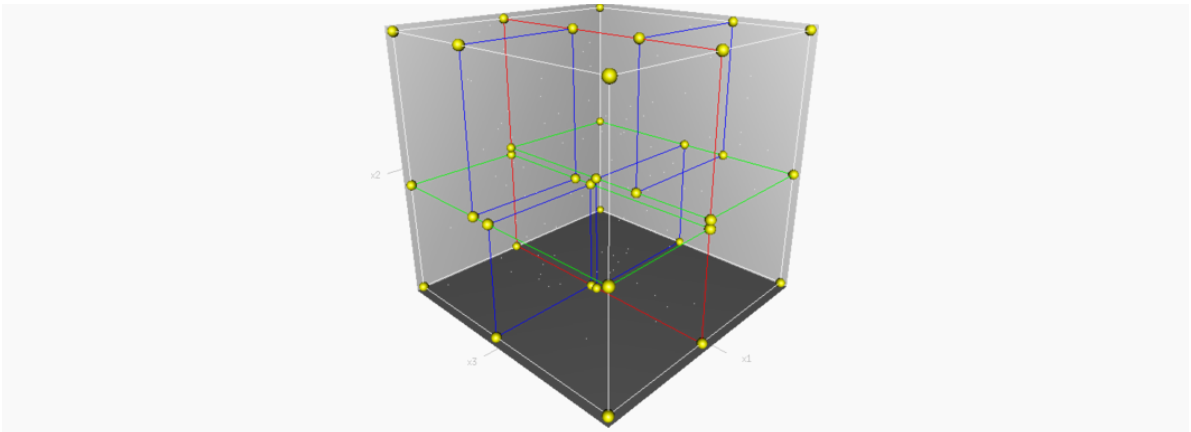
Usually, the number of keys is chosen to vary between d and $2d$, where d is the minimum number of keys, and $d + 1$ is the minimum degree or branching factor of the tree. In practice, the keys take up the most space in a node. The factor of 2 will guarantee that nodes can be split or combined. If an internal node has $2d$ keys, then adding a key to that node can be accomplished by splitting the $2d$ key node into two d key nodes and adding the key to the parent node. Each split node has the required minimum number of keys. Similarly, if an internal node and its neighbor each have d keys, then a key may be deleted from the internal node by combining with its neighbor. Deleting the key would make the internal node have $d - 1$ keys; joining the neighbor would add d keys plus one more key brought down from the neighbor's parent. The result is an entirely full node of $2d$ keys.

The number of branches (or child nodes) from a node will be one more than the number of keys stored in the node. In a 2-3 B-tree, the internal nodes will store either one key (with two child nodes) or two keys (with three child nodes). A B-tree is sometimes described with the parameters $(d + 1)$ – $(2d + 1)$ or simply with the highest branching order, $(2d + 1)$.

A B-tree is kept balanced by requiring that all leaf nodes be at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node farther away from the root.

B-trees have substantial advantages over alternative implementations when otherwise the time to access the data of a node greatly exceeds the time spent processing that data, because then the cost of accessing the node may be amortized over multiple operations within the node. This usually occurs when the node data are in secondary storage such asdisk drives. By maximizing the number of keys within each internal node, the height of the tree decreases and the number of expensive node accesses is reduced. In addition, rebalancing of the tree occurs less often. The maximum number of child nodes depends on the information that must be stored for each child node and the size of a full disk blockor an analogous size in secondary storage. While 2-3 B-trees are easier to explain, practical B-trees using secondary storage need a large number of child nodes to improve performance.

24. KD Tree



Un árbol *k*d tridimensional. La primera división (rojo) corta la celda raíz (blanco) en dos subceldas, que son divididas a su vez (verde) en dos subceldas. Finalmente, cada una de esas cuatro es dividida (azul) en dos subceldas. Dado que no hay más divisiones, las ocho finales se llaman hojas. Las esferas amarillas representan los nodos del árbol.

En ciencias de la computación, un Árbol *k*d (abreviatura de árbol *k-dimensional*) es una estructura de datos de particionado del espacio que organiza los puntos en un Espacio euclídeo de *k* dimensiones. Los árboles*k*d son un caso especial de los árboles BSP.

Un árbol *k*d emplea sólo planos perpendiculares a uno de los ejes del sistema de coordenadas. Esto difiere de los árboles BSP, donde los planos pueden ser arbitrarios. Además, todos los nodos de un árbol *k*d, desde el nodo raíz hasta los nodos hoja, almacenan un punto. Mientras tanto, en los árboles BSP son las hojas los únicos nodos que contienen puntos (u otras primitivas geométricas). Como consecuencia, cada plano debe pasar a través de uno de los puntos del árbol *k*d.

Técnicamente, la letra *k* se refiere al número de dimensiones. Un árbol *k*d tridimensional podría ser llamado un árbol 3d. Sin embargo se suele emplear la expresión "árbol *k*d tridimensional". (También es más descriptivo, ya que un árbol tridimensional puede ser varias cosas, pero el término árbol *k*d se refiere a un tipo en concreto de árbol de particionado.) Las letras *k* y d se

escriben en minúsculas, incluso al principio de una oración. La *k* se escribe en *cursiva*, aunque son también comunes las formas "árbol KD" y "árbol Kd".

25. Segment Tree

In computer science, a segment tree is a tree data structure for storing intervals, or segments. It allows querying which of the stored segments contain a given point. It is, in principle, a static structure; that is, its content cannot be modified once the structure is built. A similar data structure is the interval tree.

A segment tree for a set *I* of *n* intervals uses O(*n* log *n*) storage and can be built in O(*n* log *n*) time. Segment trees support searching for all the intervals that contain a query point in O(log *n* + *k*), *k* being the number of retrieved intervals or segments.[1]

Applications of the segment tree are in the areas of computational geometry, and geographic information systems.

The segment tree can be generalized to higher dimension spaces as well.

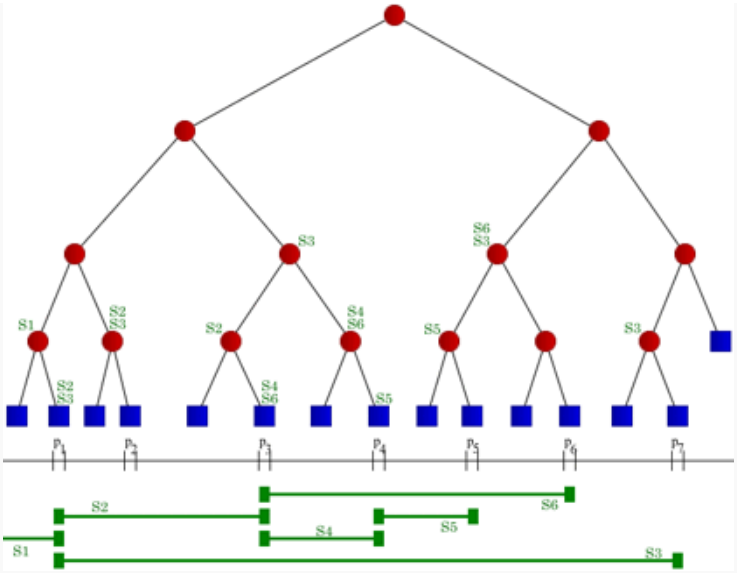
Structure description

This section describes the structure of a segment tree in a one-dimensional space.

Let *S* be a set of intervals, or segments. Let p_1, p_2, ..., p_m be the list of distinct interval endpoints, sorted from left to right. Consider the partitioning of the real line induced by those points. The regions of this partitioning are called *elementary intervals*. Thus, the elementary intervals are, from left to right:

$$(-\infty, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], \ldots, (p_{m-1}, p_m), [p_m, p_m], (p_m, +\infty)$$

That is, the list of elementary intervals consists of open intervals between two consecutive endpoints p_i and p_{i+1}, alternated with closed intervals consisting of a single endpoint. Single points are treated themselves as intervals because the answer to a query is not necessarily the same at the interior of an elementary interval and its endpoints.[2]



Graphic example of the structure of the segment tree. This instance is built for the segments shown at the bottom.

Given a set *I* of intervals, or segments, a segment tree *T* for *I* is structured as follows:

- *T* is a binary tree.
- Its leaves correspond to the elementary intervals induced by the endpoints in *I*, in an ordered way: the leftmost leaf corresponds to the leftmost interval, and so on. The elementary interval corresponding to a leaf *v* is denoted Int(*v*).
- The internal nodes of *T* correspond to intervals that are the union of elementary intervals: the interval Int(*N*) corresponding to node *N* is the union of the intervals corresponding to the leaves of the tree rooted at *N*. That implies that Int(*N*) is the union of the intervals of its two children.
- Each node or leaf *v* in *T* stores the interval Int(*v*) and a set of intervals, in some data structure. This canonical subset of node *v* contains the intervals [*x*, *x'*] from *I* such that [*x*, *x'*] contains Int(*v*) and does not contain Int(parent(*v*)). That is, each segment in *I* stores the segments that span through its interval, but do not span through the interval of its parent.