Algoritmos e Estructuras de Datos

Pedro Gamallo Fernández

2018-2019

Abstract

Apuntes sobre los temas de la asignatura *AED* centrados especialmente **en el examen de enero**, por lo que solo se mencionan muy por encima los aspectos prácticos, que ya fueron evaluados en los tests a lo largo del cuatrimestre. Aún así, son unos apuntes bastantes extensos, todavía reducibles al gusto de cada uno.

Por supuesto, seguro que contendrán algún error ortográfico o incluso de conceptos, por lo que no espero que se tomen como una verdad absoluta en la que confiar ciegamente de cara a la preparación del examen, sino un material auxiliar para repasar los contenidos una vez estudiada la materia.

1 Tema 1: Introducción

Definiciones básicas:

Algoritmo: Conjunto de reglas para resolver un problema. Sus propiedades más importantes son la *definibilidad*, es decir, dicho conjunto debe estar bien definido sin producir ningún tipo de duda o ambigüedad, y la *finitud*, lo que significa tener un número finito de pasos que se completen en un tiempo finito.

Algoritmia: Ciencia que estudia técnicas para construir algoritmos eficientes y técnicas para medir la eficiencia de dichos algoritmos, y así poder compararlos y ver cual es más adecuado.

Estructuras de datos: Es la parte *estática* de un problema a resolver, ya que consta de los datos y las variables que se almacenan del problema (el algoritmo es la parte dinámica). Es muy importante seleccionar la estructura de datos que mejor se adapte a cada problema, ya que puede simplificar enormemente tanto el número como la complejidad de las operaciones a realizar.

TADs: Tipos Abstractos de Datos. Proporcionan una abstracción (visión de alto nivel) de las estructuras de datos, ofreciendo al programador simplemente las operaciones disponibles, liberándolo de la implementación.

2 Tema 2: Árboles I. Árboles binarios

Un **árbol** es una estructura de datos *no lineal* con una *organización jerárquica* de *elementos homogéneos* (es decir, todos del mismo tipo), donde cada elemento tiene *un único padre*, pero puede generar *varios elementos sucesores o hijos*.

Definiciones generales:

Nodo: Cada elemento componente del árbol.

Ascendiente: Elementos antecesores de uno dado. Se encuentran en los niveles superiores y de ellos se deriva el elemento actual.

Descendiente: Elementos sucesores de uno dado. Se encuentran en los niveles inferiores y derivan del actual.

Nodo raíz: Nodo original del árbol. Se encuentra en el nivel más alto y es el único que no se deriva de ningún nodo padre.

Nodo hoja: Nodo terminal del árbol. Es aquel que no tiene ningún descendiente.

Subárbol: Subconjunto de elementos de un árbol, que a su vez mantienen la estructura de árbol.

Nodo padre: Nodo ascendiente directo del elemento actual. Cada nodo tiene un único padre (o 0 en el caso del nodo raíz).

Nodo hijo: Nodo descendiente directo del elemento actual. Los nodos sin ningún hijo son los ya mencionados *nodos hoja*.

Nodos hermanos: Nodos que se encuentran en el mismo nivel y se derivan del mismo nodo padre.

Nodo interno: Nodo que cuenta con un nodo padre (por lo tanto no es el *nodo raíz*) y con uno o más nodos hijos (por lo que tampoco es *nodo hoja*).

Camino: Secuencia de nodos entre un nodo origen y un nodo destino, donde cualquier par de nodos mantienen la relación padre-hijo.

Rama: Camino desde el nodo raíz hasta un nodo hoja.

Nivel: (Término ya mencionado anteriormente) Número de nodos que tiene el camino desde la raíz hasta el elemento actual. Por tanto, *el nivel del nodo raíz es 1* (para llegar del nodo raíz al nodo raíz hay que recorrer un nodo, que es él mismo).

Grado: Número de hijos de un nodo dado.

Peso: Número de nodos hoja que tiene un árbol.

Altura: Nivel más alto de un árbol. Dicho de otra forma, es el número de nodos de la rama más larga.

Dentro de los árboles existen varios tipos de estructuras de datos con diferentes características. En este tema se estudian los **árboles binarios**.

Árbol binario: Árbol donde cada nodo tiene como máximo grado 2, es decir, dos nodos hijos.

Árbol binario equilibrado: Árbol binario donde la diferencia de altura entre los subárboles de cada nodo (subárbol izquierdo y subárbol derecho) es como máximo de una unidad.

Árbol binario totalmente equilibrado: Árbol binario equilibrado donde la altura de los subárboles de cada nodo es igual (es decir, todos los nodos hoja de cada uno de los subárboles de un nodo dado están en el mismo nivel).

Árbol binario lleno: Árbol binario donde todos los nodos hoja se encuentran en el mismo nivel, y sus padres tienen todos 2 hijos (y por tanto ya no aceptan más). Por tanto, el árbol binario lleno es un árbol binario totalmente equilibrado con la restricción de que además los subárboles no pueden admitir más hijos en el último nivel.

Árbol binario completo: Árbol binario equilibrado donde en el último nivel todos los nodos hoja se distribuyen de izquierda a derecha sin dejar huecos. Si además no caben más hijos en el último nivel, tenemos un *árbol binario lleno*.

Existen dos posibles implementaciones del TAD Árbol binario: con *memoria dinámica (punteros)* o con *memoria estática (vectores)*. Pero en cualquier caso, cada elemento tiene que contener la información sobre el nodo que representa y enlaces al hijo izquierdo y el hijo derecho (mediante punteros a los elementos correspondientes en el caso de la implementación con memoria dinámica). La implementación con memoria dinámica suele ser la escogida porque además de facilitar la localizacion de los hijos con el uso de punteros, soluciona el problema de no disponer de memoria contigua del tamaño necesario para la representación de todos los

elementos.

Para enumerar los elementos de un árbol se utilizan los **recorridos**. Existen dos tipos principales de recorridos: los *recorridos en anchura* y los *recorridos en profundidad*. A continuación pasaremos a explicarlos.

2.1 Recorrido en anchura

Consiste en recorrer los distintos niveles del árbol de forma ordenada, del menor al mayor, imprimiendo los elementos de cada nivel de izquierda a derecha.

Su implementación es evidentemente no recursiva y se usa una *cola* como estructura de datos auxiliar. Para su ejecución se siguen los siguientes pasos:

1. Se empieza en el nivel más bajo (el 1, en el que se encuentra el nodo raíz).
2. Se coge el primer elemento de la lista, sacándolo de ésta.
3. Se mira si tiene algún descendiente, y si los tiene, se introducen de izquierda a derecha.
4. Se repiten los pasos 2 y 3 hasta que la lista esté vacía.

2.2 Recorrido en profundidad

Existen tres recorridos en profundidad, cada uno de ellos con una implementación recursiva y otra no recursiva.

Inorden: Izquierda-Raíz-Derecha

Se recorre el árbol izquierdo, luego la raíz y por último el árbol derecho, para cada subárbol de cada nodo.

Preorden: Raíz-Izquierda-Derecha

Para cada nodo se recorre primero el elemento actual, luego su subárbol izquierdo y por último el derecho.

Postorden: Izquierda-Derecha-Raíz

Para cada nodo se recorre primero su subárbol izquierdo, luego el derecho y por último el elemento actual.

2.2.1 Recorridos recursivos

En cada nodo, en función del recorrido seleccionado, se imprime el elemento actual o se sigue descendiendo por el subárbol hasta llegar a un nodo hoja, el cual se imprime, y se empieza a subir para imprimir los demás. La recursividad se refleja en el hecho de ir descendiendo por el árbol hasta llegar al nodo hoja. *(Diapositivas 32 a 34 en la presentación del Tema 2)*

2.2.2 Recorridos no recursivos

Se utiliza una estructura de datos auxiliar, generalmente una pila, para ir almacenando los nodos antes de imprimirlos. En el caso del *recorrido inorden*:

1. Se guardan los elementos izquierdos en la pila hasta llegar a un nodo sin hijo izquierdo.
2. Se desapila el tope, se imprime y se inserta en la pila el hijo derecho.
3. Se repiten los pasos 1 y 2 hasta que la pila esté vacía.

2.3 Árboles de expresión

Se utilizan para representar expresiones matemáticas en memoria. Cada nodo representa un operando o un operador. De hecho, la raíz de cada subárbol es un operador. La operación **A + B** se representaría como un árbol con + como nodo padre, *A* como hijo izquierdo y *B* como nodo derecho.

Si a los árboles de expresión se les realiza el recorrido inorden, obtenemos la notación natural de la operación matemática.

Para la construcción de árboles de expresión a partir de una operación en notación convencional se usan dos pilas comos estructuras de datos auxiliares, una para operadores y otra para operandos. Los operadores y los operandos se van apilando en sus respectivas pilas hasta que llega un operador cuya prioridad es menor o igual que el que se encuentra en el tope de la pila de operadores, lo que produce que se desapile en dicha pila, y se vaya formando la expresión en la pila de operandos con los operadores desapilados. Como dato importante, cabe destacar que el paréntesis izquierdo se apila siempre, y el derecho produce siempre que se desapile hasta llegar al paréntesis izquierdo. *(Diapositivas 46 y 47 en la presentación del Tema 2)*

3 Tema 3: Árboles II. Estructuras para búsqueda

Un uso importante de los árboles, especialmente de los árboles binarios, es en algoritmo de búsqueda. Para ello, se han desarrollado los *árboles binarios de búsqueda.*

3.1 Árbol binario de búsqueda:

Árbol binario en el que, para todo nodo, todos los elementos de su subárbol izquierdo tienen un valor menor que él, y todos los elementos de su subárbol derecho tienen un valor mayor valor que él.

Presenta la clara ventaja de que localizar un elemento es muy fácil y rápido, al estar los datos ordenados.

Además, al realizarle el recorrido inorden, los elementos se imprimen ordenador de menor a mayor.

3.1.1 Insercción

En un **ABB** la inserción no se puede realizar en el primer hueco libre que se vea, sino que hay que respetar la estructura, buscando desde la raíz el hueco correspondiente al nodo que queremos insertar. Por tanto, la inserción es *recursiva*, ya que se parte del nodo raíz y se va mirando si el elemento que queremos insertar es menor o mayor, bajando por la rama correspondiente hasta encontrar su sitio.

Como clara desventaja de este tipo de árbol tenemos el caso en el que la mayor parte de los elementos que se inserten sean mayores o menores que el nodo raíz, provocando un árbol claramente desbalanceado, lo que además aumenta el tiempo de búsqueda al tener que recorrer muchos elementos, casi como si se tratase de una lista.

3.1.2 Eliminación

La desventaja en el caso de la eliminación reside en tener que reestructurar el árbol si se eliminan ciertos nodos, para mantener la estructura que cumpla las restricciones de los ABB. Existen tres posibles situaciones de eliminación:

1. Si el nodo es hoja, se suprime del árbol sin más.

2. Si el nodo tiene un único hijo, se intercambia de posición con su hijo y se elimina del árbol.

3. Si el nodo tiene dos hijos, buscamos el menor de todos los descendientes del hijo derecho, sustituimos por dicho nodo, y lo eliminamos del árbol. Otra opción es sustituir por el nodo de mayor valor de los descendientes del subárbol izquierdo.

3.2 Montículo binario

Árbol binario completo (aunque a veces llamado s*emicompleto*) que da soporte eficiente a las operaciones del TAD *cola de prioridad*. Por tanto, sus aplicaciones son las mismas que las aplicaciones de una cola con prioridad: Simulación de eventos discretos (planificación de procesos en un sistema multiusuario, gestión de enfermos en un servicio de urgencias, etc.) y algoritmos voraces (ordenando los candidatos según maximicen nuestra puntuación o función objetivo).

Pueden ser *montículos de mínimos*, de forma que los elementos con menor valor tienen mayor prioridad, o *montículos de máximos*, en los que los elementos de mayor valor tienen mayor prioridad. Para nuestros ejemplos usaremos un *montículo de mínimos*, por lo que siempre se tendrá que cumplir la condición de que cada nodo tenga un valor menor que el de cualquiera de sus hijos. Por tanto, se dice que los montículos mantienen un *orden parcial*, ya no es tan estricto como los *ABB*, pero lo es más que una ordenación aleatoria.

Tienen la gran ventaja de que facilitan la inserción, no teniendo que reestructurar todo el árbol cada vez que llega un elemento nuevo.

En este tipo de aplicaciones hay que buscar siempre el elemento de *mayor prioridad*, que *será siempre la raíz*.

3.2.1 Implementación

Existen dos posibles implementaciones: *dinámica* y *estática*:

Implementación dinámica: Igual que en el resto de los árboles binarios, cada elemento consta de su valor y un puntero a cada uno de sus hijos.

Implementación estática: Se representan los elementos del árbol dentro de un array, usando el índice de cada elemento como localizador (puntero) de su padre o de sus hijos. De esta forma, el hijo izquierdo de k se sitúa en la posición $2k$ y el hijo derecho en la posición $2k+1$, mientras que su padre está en la posición $k/2$ *(división entera.*

Esta representación facilita las operaciones de búsqueda tanto en simplicidad como en tiempo de ejecución, pero se requiere conocer el número máximo de elementos de antemano. Además, el hecho de reestructurar el árbol se hace más difícil en este tipo de implementaciones.

3.2.2 Inserción

Como árbol binario completo que es, la inserción se hace en el hueco libre más a la izquierda del último nivel. Sin embargo, si se inserta un elemento con un valor (para nuestro ejemplo de montículo de mínimos) menor que su padre, es necesario reorganizar el árbol intercambiando el elemento con su padre hasta llegar una posición en la que se cumpla el criterio.

3.2.3 Eliminación

En este tipo de árboles, el elemento que se quiere suprimir es siempre el de mayor prioridad, es decir, la raíz. Para llevar a cabo esta operación, el nodo raíz se intercambia de posición con

el nodo hoja más a la derecha (el nodo más a la derecha del último nivel), y una vez allí, se elimina. A continuación, se procede a la reorganización del nuevo nodo raíz hasta encontrar su sitio, intercambiándolo con (en nuestro caso) el hijo menor en cada caso.

4 Tema 4: Árboles III. Árboles equilibrados AVL

Con árboles desbalanceados se pierde la eficiencia de búsqueda, acercándose a la de una lista, es decir *O (n)* para la búsqueda. Sin embargo, con árboles equilibrados se consigue una eficiencia de *O log(n)* en la búsqueda, inserción o eliminación.

Aquellos árboles que se mantienen siempre equilibrados debido a constantes reordenaciones se conocen como **árboles AVL** (*Adelson-Velskii y Landis*).

En este tipo de árboles, cada nodo cuenta con un campo llamado **factor de equilibrio**, que almacena la *diferencia entre la altura del subárbol derecho y la altura del subárbol izquierdo* para dicho nodo. Como se estudió en el tema 2, el árbol será equilibrado mientras el valor absoluto de todos los factores de carga sea **menor que 2**.

4.1 Inserción

Se pueden dar varios casos en la inserción:

1. Si las ramas izquierda y derecha tienen la misma altura, da igual donde se inserte, que al insertar solo se producirá un nuevo factor de equilibrio de -1 o 1, con lo que el árbol sigue estando equilibrado.

2. Si las ramas izquierda y derecha difieren en su altura en 1 unidad:

   (a) Si el factor de equilibrio era -1 y se inserta por la derecha, o si el factor de equilibrio era 1 y se inserta por la izquierda, el nuevo factor de equilibrio pasa a ser 0, por lo que se mejora el equilibrio del árbol y no hay ningún problema.

   (b) Si el factor de equilibrio era -1 y se inserta por la izquierda, o si el factor de equilibrio era 1 y se inserta por la derecha, el nuevo factor de equilibrio pasa a ser -2 o 2 (respectivamente), desequilibrando el árbol, y siendo necesaria una reestructuración.

4.2 Reestructuración

Tras una inserción, el nodo insertado pasa a ser un nodo hoja, por lo que su factor de equilibrio es 0. Posteriormente vamos subiendo por el árbol recalculando los factores de equilibrio hasta encontrar uno cuyo valor absoluto sea 2, a partir del cual habrá que reestructurar el árbol.

Este proceso se repite hasta llegar al nodo raíz.

Existen 4 tipos de estructuraciones, que no son más que *rotaciones* del árbol: *I significa izquierda y D derecha*

1. **Rotación simple II:** Se produce cuando un nodo tiene factor -2 y su hijo izquierdo, factor -1. Se rota hacia la derecha, dejando el nodo con factor -1 como nuevo padre, el nodo con factor -2 como hijo derecho, y el nodo que originó el desequilibrio, como hijo izquierdo.

2. **Rotación simple DD:** Se produce cuando un nodo tiene factor 2 y su hijo derecho, factor -1. Es lo contrario al caso anterior: Rotación a la izquierda, nodo con factor 1 como padre, nodo con factor 2 como hijo izquierdo, y nodo que originó el desequilibrio como hijo derecho.

3. **Rotación compuesta DI**: Se produce cuando un nodo tiene factor 2 y su hijo derecho, factor -1. Se rota el hijo con factor a la derecha y se sube el nodo que originó el desequilibrio a su posición, y se tiene una rotación simple DD.

4. **Rotación compuesta ID**: Se produce cuando un nodo tiene factor -2 y su hijo izquierdo, factor 1. Se rota el hijo con factor 1 a la izquierda y se pone el nodo que causó el desequilibrio en su posición, teniendo un rotación simple II.

4.3 Eliminación

Se sigue el **mismo algoritmo que para ABB**, pero **incluyendo las reestructuraciones necesarias**.

Recordemos los posibles casos:

1. Si el nodo a suprimir es un nodo hoja, se suprime sin más.

2. Si el nodo a suprimir solo tiene un descendiente, se sustituye por su descendiente y se elimina.

3. Si el nodo tiene dos subárboles, se busca el nodo más a la derecha del subárbol izquierdo, o el nodo más a la izquierda del subárbol derecho, se sustituye, se elimina, y se comprueba si es necesaria reestructuración.

5 Tema 5: Árboles IV. Árboles equilibrados B y B+

Si queremos almacenar una cantidad muy grande de datos, podemos llegar al caso de que ocupe varios bloques de disco, lo que supone tener que hacer varios accesos, ralentizando mucho el tiempo de búsqueda. Para minimizar el problema del tiempo de acceso al disco (u otros medios externos), se desarrollaron los **árboles B** y **árboles B+**.

5.1 Árboles B

Árboles siempre equilibrados que minimizan el tiempo de búsqueda y borrado de elementos. En ellos, los nodos se agrupan en **páginas**, a las que se acceden el bloques.

El **orden** del árbol B indica el número máximo de ramas que puede tener cada páginas interna.

Sea m el orden del árbol B, cada página interna debe tener un mínimo de $m/2$ ramas y un máximo de m. Además, el número de elementos de cada página es igual a su número de ramas menos 1. Por otra parte, todas las páginas hoja se encuentra en el mismo nivel, por lo que se trata de un árbol **totalmente equilibrado**.

Se puede concluir, por tanto, que cada página es un vector de, como máximo, *m-1 elementos*, y entre *m/2* y *m ramas* (a no ser que sea una página hoja).

5.1.1 Inserción

Se busca la página correspondiente, descendiendo por el árbol hasta llegar la página hoja adecuada. Al querer insertar en dicha hoja, se pueden dar 2 casos:

1. Si la página no está llena, se inserta sin problema el elemento.

2. Si la página está llena, ésta se divide en 2 (insertando ya el elemento en su sitio correspondiente) y el elemento *mediana* (el que ocupe la posición del medio) asciende en el árbol al nivel superior, insertándose en la página padre, y de dicho elemento saldrán dos ramas, una a cada página generada en el elemento inferior (eliminándose, evidentemente, la rama que apuntaba a la página que se acaba de dividir, y que por tanto dejó de existir).

5.1.2 Búsqueda

Para buscar una clave en el árbol B se necesita partir de la página raíz y, evidentemente, especificar la clave a buscar. Por tanto, estos serán los dos parámetros que se le pasen a la función *Buscar()*. Se necesita un procedimiento auxiliar llamado *Buscarnodo()* que examina para cada página dada si la clave se encuentra en ella o, en caso contrario, entre que dos claves iría. En el segundo de los casos, se desciende a la página correspondiente del nivel inferior por la rama que se encuentra entre los elementos, y se repite la búsqueda.

5.1.3 Eliminación

En las eliminaciones también se pueden dar varios casos:

1. **La página donde se va a insertar tiene más elementos que el mínimo:** Si es una página hoja, se elimina sin problema. Si no es hoja, se sustituye el elemento por el predecesor en la hoja que salga de su rama derecha, y se elimina el elemento ahora en la página hoja.

2. **La página donde se va a eliminar tiene el número mínimo de elementos:** Se pueden dar 2 casos:

   (a) Si una de las hojas hermanas tiene más elementos que el mínimo, se toma el elemento del extremo de dicha página, se sube a la padre, y de la padre se baja el correspondiente a la página en la que queremos hacer la eliminación, para tener más elementos que el mínimo y poder hacer la eliminación simple.

   (b) Si todas las páginas hermanas tienen el mínimo de elementos, se unifican las dos páginas y se trae la mediana de ambas que está en la página padre. De esta forma la página padre reduce en uno sus elementos, y también en una sus ramas.

5.1.4 Aplicaciones

Como se ha visto, al agrupar los nodos en páginas se reduce considerablemente la altura del árbol, ahorrando bloques en disco. Las búsquedas y las inserciones y eliminaciones sin modificación de la estructura se realizan de forma más rápida que en un árbol normal. Por otra parte, cuando hay subdivisiones o unificaciones, se requieren más accesos a disco (aunque pueden seguir siendo menos que con el árbol convencional).

Por tanto, los árboles B pueden suponer una buena opción para almacenamiento masivo de datos en disco.

Aquí se muestran algunos ejemplos en los que se usan árboles B:

- Creación de bases de datos
- Gestión de sistemas de archivos de sistemas operativos
- Sistemas de compresión de datos (para buscar datos comprimidos por medio de una clave).

5.2 Árboles B*

Optimización de los árboles B. En la **inserción**, si la página está llena, se mueven elementos a una página hermana, *posponiendo la división hasta que los hermanos estén completos*. Con esto se reducen el número de divisiones.

5.3 Árboles B+

Optimización de los árboles B para conseguir un recorrido secuencial más rápido.

Todos los elementos se encuentran en páginas hoja, de forma que algunos de estos elementos aparecen duplicados en las páginas de niveles superiores únicamente con la función de actuar como índices. Debido a esto, ocupan algo más de espacio en memoria, pero para árboles que se modifican frecuentemente supone también un aspecto positivo, ya que evita la reorganización del árbol, que era la operación más costosa en los árboles B.

5.3.1 Inserción

Por lo general es similar a la inserción en los árboles B, salvo por el caso en el que la página este llena. En estos casos, el elemento *mediana* sube a la página padre. pero también se mantiene en la página de la derecha resultante de la división en 2 de la página hoja original.

5.3.2 Eliminación

La eliminación es mucho más simple que en árboles B, ya que los elementos siempre están en las páginas hojas.

Se pueden dar dos casos:

1. Si tras eliminar el elemento la página todavía tiene el mínimo de elementos o más, no hay que hacer ninguna modificación.

2. Si tras eliminar el elemento la página queda con menos elementos que el mínimo, hay redistribuir los elementos, borrando además un índice en la página de nivel superior, o cambiándolo.

6 Tema 6: Grafos I

Mientras que los árboles representan relaciones entre objetos entre los cuales existe una jerarquía, los grafos son modelos naturales de **relaciones arbitrarias entre objetos**.

Algunas definiciones generales importantes son las siguientes:

Grafo G=(V,A): Conjunto de vértices o nodos **V** y un conjunto de arcos **A** mediante los que se relacionan.

Arco/arista (u,v): Relación entre un par de nodos **u** y **v**.

Grafo dirigido (digrafo: Grafo en el que los pares de nodos forman arcos en una sola dirección. Es decir, un arco va del primer nodo al segundo, pero no al revés.

Grafo no dirigido: Grafo en el que los pares de nodos forman arcos bidireccionales. Un mismo arco va del primer nodo al segundo y del segundo al primero.

Nodos adyacentes: Par de arcos unidos por un arco en los grafos no dirigidos. Por contra, en los grafos dirigidos, solo el primer nodo del arco es adyacente del segundo, pero no viceversa.

Grafo valorado: Grafo en el que los arcos tienen un factor de peso asociado.

Grado de un nodo: En los grafos no dirigidos es el número de arcos que se comunican con el nodo. Sin embargo, en los digrafos tenemos que diferencias entre el **grado entrante** (arcos que llegan al nodo) y **grado saliente** (arcos que salen del nodo.

Camino: Secuencia de nodos comunicados por medio de arcos que se recorren para llegar de un nodo origen a un nodo final.

Longitud del camino: Número de arcos que forman un camino.

Bucle: Arco de un nodo a si mismo.

Grafo conexo: Grafo en el que existe camino entre cualquier par de nodos que lo forman. En el caso de los digrafos, se diferencia entre **fuertemente conexo** (si existe un camino entre cualquier par de nodos) y **conexo** (si simplemente existe una cadena que una cualquier par de vértices, aunque implique arcos en sentido contrario).

Grafo completo: Grafo que contiene un arco entre cualquier par de vértices.

6.1 Aplicaciones

Los grafos son estructuras de datos muy usadas para solucionar problemas cotidianos o técnicos. Algunos de los ejemplos más conocidos son:

- Redes de alcantarillados.
- Redes de comunicaciones.
- Circuitos eléctricos.
- Diagramas de flujo de algoritmos.

6.2 Representación

Hay que representar los vértices y los arcos que los unen. Existen dos posibles representaciones: *representación secuencial* por medio de **matrices de adyacencia**, o *representación dinámica* con **listas de adyacencia**. Cada representación tiene sus ventajas y desventajas en función de las operaciones que se apliquen sobre los vértices y arcos.

6.2.1 Matriz de adyacencia

Matriz cuadrada de orden n, siendo *n el número de nodos* del grafo. Cada fila y columna se corresponden con un nodo, por lo que es necesario establecer un orden entre éstos. En cada posición *i,j* de la matriz hay un 0 si no existe arco entre dichos nodos, o un 1 si sí lo hay.

En **no dirigidos** esta matriz es **simétrica**, mientras que en los digrafos no tiene porqué, y generalmente no lo es.

Si es un grafo valorado, en cada entrada de la matriz se pone el valor del peso del arco, en vez de un simple 1, formando así la **matriz de pesos**.

Como **ventajas** tenemos:

- Eficiencia a la hora de obtener los costes asociados a un arco.
- La comprobación de adyacencia entre 2 nodos cualesquiera es inmediata e independiente del número de nodos.

Como **desventajas** están:

- No se permite la eliminación de nodos del grafo, ya que no se pueden suprimir filas ni columnas en la matriz.

- En grafos dispersos, con pocos arcos, se producen matrices con muchos 0, lo que es un ejemplo de espacio desaprovechado.

6.2.2 Lista de adyacencia

Lista directorio donde cada elemento se corresponde con un nodo del grafo. A su vez, cada uno de estos elementos cuenta con una *lista enlazada de adyacencia* en la que aparecen aquellos nodos con los que es adyacente.

Ventajas:

- Ocupa menos memoria que la matriz de adyacencia cuando el número de arcos es mucho menor que el número de nodos, al evitar tener que almacenar todos los 0 de la matriz.

Desventajas:

- La representación es más compleja, al contar con varios punteros
- Es ineficiente para encontrar los arcos que llegan a un nodo, pues hay que recorrer todas las listas viendo en cada una si hay un arco al elemento buscado.

6.3 Recorridos

Consiste en visitar todos los nodos, cada uno a partir de otro visitado anteriormente.

6.3.1 Recorrido en anchura

Consiste en partir de un nodo dado e ir visitando los nodos adyacentes que no estuvieran ya visitados, repitiendo estos pasos hasta visitarlos todos.

Se trata de un recorrido **no recursivo**, que utiliza una cola como estructura de datos auxiliar para recoger los nodos adyacentes al actual que no estén visitados.

6.3.2 Recorrido en profundidad

Se parte de un nodos dado y se recorrer en profundidad cada vertice adyacente a éste que no estuviera visitado, repitiendo estos pasos hasta que no queden más nodos por visitar.

Como se puede imaginar, es un algoritmo que acepta una definición recursiva, pero también una no recursiva.

- **Recorrido en profundidad recursivo:** A partir de un nodo, para cada uno de sus adyacentes, se marca como visitado y se realiza sobre cada uno el mismo recorrido recursivo. Una vez se ha realizado el recorrido recursivo de todos los nodos adyacentes al dado, se prosigue con los demás nodos del grafo que no hayan sido visitados.
- **Alternativa no recursiva:** Emplea una pila como estructura de datos auxiliar, de forma que a partir de un nodo dado, se inserta en la pila, se desapila el tope y se insertan sus nodos adyacentes no visitados, repitiendo estos dos pasos hasta tener la pila vacía.

6.4 Componentes conexas de un grafo

La determinación de si un grafo es conexo (o fuertemente conexo en el caso de los digrafos) es de gran importancia a la hora de solucionar problemas de redes de comuniaciones o transportes para determinar si es posible llegar de un nodo a cualquier otro.

Si un grafo no es conexo, está formado entonces por **componentes conexas**.

6.4.1 Algoritmo para determinar las componentes conexas en un grafo no dirigido

1. Realizar un recorrido del grafo a partir de cualquier vértice, almacenando los vértices visitados.
2. Si el conjunto de los vértices visitados es igual al conjunto de todos los nodos del grafo, el grafo es conexo
3. En caso contrario, el conjunto de vértices visitados en el recorrido anterior es una **componente conexa**.
4. Se toma un vértice no visitado y se repiten los pasos 1 y 3.
5. El algortimo finaliza cuando no quedan nodos por visitar

6.4.2 Algoritmo para determinar las componentes conexas en un digrafo

1. A partir de un nodo, obtener el conjunto de todos sus descendientes
2. A partir del mismo nodo, obtener el conjunto de todos sus ascendientes
3. Calcular la intersección de los dos conjuntos. Si es igual al conjunto de todos los nodos del grafo, el grafo es fuertemente conexo. En caso contrario no lo es.
4. Si no es fuertemente conexo, repetir los pasos 1, 2 y 3 con nodos no visitados hasta encontrar todas las componentes fuertemente conexas del grafos.

6.5 Matriz de caminos

Si multiplicamos la matriz de adyacencia por si misma, obtenemos la matriz de caminos de longitud 2, que tiene un 1 en aquellas posiciones *i,j* entre las que exista un camino de longitud 2 entre el nodo *i* y el nodo *j*. Si esta matriz la multiplicamos de nuevo por la matriz de adyacencia, obtenemos la matriz de caminos de longitud 3, y así sucesivamente.

La **matriz de caminos** genérica es aquella que resulta de sumar todas las matrices de caminos de longitudes 1 hasta n, siendo *n el número de nodos*.

En un grafo fuertemente conexo, todas las entradas de la matriz de caminos tienen un 1 excepto las de la diagonal principal.

6.6 Punto de articulación

Se llama **punto de articulación** a aquel nodo que si se elimina (junto con sus arcos), divide la componente conexa en la que se encuentra en 2 o más componentes conexas.

6.6.1 Algoritmo de búsqueda de puntos de articulación

Se realiza un *recorrido en profundidad recursivo* del grafo, que se puede representar con un *árbol de expansión*. Cada arco del grafo será una arista del árbol, produciendo *aristas hacia atrás* si un vértice de los que une el arco ya estaba visitado anteriormente.

Se realizan los siguientes pasos:

1. Se numeran los nodos en el orden en el que se visitaron en el recorrido en profundidad. A esta numeración se le llama **Num(v)**

2. Para cada vértice del árbol determinamos **Bajo(v)**, que es el mínimo entre 3 números:

   - *Num(v)*
   - El menor valor de *Num(w)* para los vértices w de las aristas hacia atrás del vértice v.
   - El menor valor de *Bajo(w)* para sus descendientes w.

3. Determinamos los puntos de articulación:

   - La raíz es punto de articulación si y sólo si tiene 2 hijos.
   - Cualquier vértice w es punto de articulación si y sólo si tiene al menos un hijo u tal que *Num(w) >= Bajo(u)*

7 Tema 7: Grafos II. Algoritmos fundamentales con grafos

7.1 Ordenación topológica

Es importante el término **GDA *Grado Dirigido Acíclico***, que es el nombre que reciben aquellos digrafos que no presentan ciclos, es decir, caminos desde un vértice hasta si mismo (por tanto siempre hay ceros en la diagonal de la matriz de caminos).

Este tipo de grafos son útiles para representar estructuras sintáticas de expresiones ariméticas, y ordenaciones parciales.

Las **ordenaciones parciales** son un tipo de relación entre un conjunto de elementos de formas que uno nunca se relaciona con si mismo (*no reflexiva*) y si un elemento a se relaciona con uno b, y a su vez éste se relaciona con un elemento c, se puede decir que a y b están relacionados (*transitiva*).

Una vez sabido esto, podemos pasar a la definición de **ordenación topológica**: ordenación lineal tal que si hay un camino de un vértice a otro, entonces el vértice origen aparece antes que el vértice destino.

Por tanto, solo existe ordenación topológica en GDAs.

7.1.1 Algoritmo para la obtención de una ordenación topológica

1. Buscamos los nodos cuyo **grado entrante sea 0** y los metemos en una **cola**, que usaremos como estructura de datos auxiliar.

2. Sacamos el vértice del **frente de la cola** y lo añadimos a la ordenación.

3. **Eliminamos los arcos** que salen del vértice con el que acabamos de trabajar, recalculando los grados entrantes de los demás vértices.

4. Repetimos los pasos 1, 2 y 3 hasta que la cola esté vacía.

7.2 Calculo de la matriz de caminos: Algoritmo de Warshall

Wharsall encontró una forma de calcular las matrices de caminos de distintas longitudes sin tener que realizar multiplicaciones, solo **fijándose en la matriz anterior**.

La operación a realizar es la siguiente:

$$P_k(i,j) = min[1, P_{k-1}(i,j) + (P_{k-1}(i,v) * P_{k-1}(v,j))]$$

Es decir. el campo *(i,j)* tendrá un 1 solo si ya había un 1 en ese mismo campo de la matriz anterior, o si en la matriz anterior había un camino desde el vértice i hasta uno intermedio, y otro desde ese intermedio hasta el vértice j.

7.3 Camino más corto entre un par de vértices: Algoritmo de Dijkstra

Un problema común es querer obtener el camino más corto entre dos vértices en un grafo valorado. Dicho camino recibe el nombre de **camino mínimo**.

Un ejemplo en los que se presenta este problema es: Obtener el camino más rápido o barato (si hay que coger metro o bus) entre dos localizaciones.

El **Algoritmo de Dijkstra** calcula el camino de longitud mínima entre un vértice origen y todos los demás vértices del grafo.

Se trata de un **algoritmo voraz**, ya que va seleccionando la mejor solución posible en cada paso que realiza. Para dicho algoritmo tenemos:

- **C:** Conjunto de vértices candidatos (todavía sin estudiar).
- **S:** Conjunto de vértices ya escogidos.
- **Camino especial:** Camino que sale del vértice origen (a partir del cual se estudian los caminos más cortos) y que pasa por todos los vértices de **S** excepto probablemente el último.
- **D:** Vector de distancias, que mantiene en cada uno de sus campos la longitud del camino especial más corto entre el vértice origen y el vértice del grafo al que represente la posición del campo del vector.
- **A:** Matriz de pesos, necesaria para poder calcular los caminos.
- **P:** Vector auxiliar que contiene en cada uno de sus campos el predecesor inmediato del camino especial mínimo para el vértice que represente la posición del campo en el vector.

Para realizar el algoritmo, se parte de un vértice origen, y a partir de él se establece en D el peso de los arcos con sus vértices adyacentes, poniendo en P (en el campo que corresponda a estos vértices) el número del vértice inicial. Se marca el vértice original como ya estudiado (se añade a S), y se selecciona el vértice con el que se estableció el camino más corto de todos los caminos establecidos hasta este punto. Se repiten estos pasos sobreescribiendo solo el valor de aquellos caminos que tuvieran una longitud mayor a la obtenida por medio del vértice de estudio actual.

Como se ve, tiene la desventaja de que si queremos saber el camino mínimo entre cualquier par de vértices, tenemos que realizar el algoritmo partiendo de o todos los vértices del grafo y luego comparar.

7.4 Camino más corto entre cualquier par de vértices: Algoritmo de Floyd

Para evitar el problema anterior, se implementó en **algoritmo de Floyd**. Este algoritmo en vez de un vector, como en el caso anterior, devuelve una matriz **D** donde cada campo *(i,j)* contiene el coste mínimo de los caminos que van desde i hasta j.

Para realizar el algoritmo, se siguen los mismos pasos que con el **Algoritmo de Wharsall** para encontrar las distintas matrices de caminos de las distintas longitudes. En este caso la matriz D_1 muestra los caminos más cortos entre cualquier par de vértices con longitud 1, la matriz D_2 muestra los caminos más cortos entre cualquier par de vértices con **longitud 2 o inferior**, y así sucesivamente. Por tanto, la matriz D_n, siendo n el número de nodos, nos mostrará los caminos más cortos entre cualquier par de vértices, independientemente de la longitud del camino.

7.5 Control de flujo

El control de flujo supone la forma de controlar la cantidad de objetos o elementos que se transportan de un lugar a otro.

Algunos ejemplos son:

- Flujo de mercancías entre el lugar donde se producen y donde se reciben.
- Flujo de personas que realizan un desplazamiento entre dos puntos señalados.
- Flujo de agua por medio de un sistema de tuberías.

Muchas veces lo que se pretende es maximizar el flujo, es decir, transportar la mayor cantidad posible de elementos desde el punto de partida (**fuente s**) hasta un punto final (**sumidero t**).

Para este tipo de problemas, un **grafo valorado** es la estructura perfecta. En muchos casos, incluso un *grafo dirigido valorado*, pero eso es algo que ya depende de las especificaciones concretas.

Para resolver estos problemas de maximización de flujo se emplea el **Algoritmo de Ford-Fulkerson**, el cual va buscando los caminos entre el nodo fuente y el sumidero en los que se pueda incrementar el flujo todavía más, para obtener en el sumidero la mayor cantidad de elementos posibles.

La explicación de los pasos del algoritmo es algo más compleja *(diapositivas 30 a 37 de la presentación del tema 7)*, pero a nosotros lo que nos interesa es básicamente su funcionalidad.

7.6 Árbol de expansión de coste mínimo

A veces, a partir de un **grafo no dirigido**, se pretenden modelar relaciones simétricas entre elementos. En esta situación, tenemos varias definiciones importantes:

- **Red conectada:** Red (grafo que representa las relaciones entre elementos) en la que desde un vértice se puede llegar a cualquier otro por medio de un camino.
- **Árbol:** Red que es subconjunto de la red original, siendo también una red conectada, con **n** nodos y *n-1* vértices, y a la que si se le añade un arco más, se forma un ciclo.
- **Árbol de expansión:** Árbol que contiene **todos los vértices** de la red original.

- **Árbol de expansión de coste mínimo:** Árbol de expansión en el cual el paso de sus arcos suman el menor valor posible.

Buscar un árbol de expansión es una forma de saber si una red es conectada.

Los árboles de expansión de coste mínimo tienen muchas funcionalidades:

- Diseño de redes de telecomunicaciones, para saber la forma más barata o corta de conectar todos los nodos.
- Diseño del menor número de caminos que unan todos los pueblos de una localidad de la forma más corta posible.

Para encontrar estos árboles de expansión existen 2 algoritmos: el **Algoritmo de Prim** y el **Algoritmo de Kruskal**.

7.6.1 Algoritmo de Prim

Es un **algoritmo voraz**, ya que en cada paso se añade el arco más corto disponible al árbol (mejor solución en cada paso). Se parte de un vértice inicial y se añade el vértice adyacente cuyo arco entre ambos tenga el valor de menor peso. A continuación, entre los vértices ya conectados, se busca el arco de menos peso que los conecten sus vértices adyacentes que no hayan sido conectados todavía. Se repiten estos pasos hasta haber conectado todos los vértices.

7.6.2 Algoritmo de Kruskal

Se parte del conjunto de vértices del grafo y se selecciona el arco de menor peso, uniendo dos vértices y reduciendo en uno el número de componentes conexas. A continuación se selecciona el siguiente arco de menor peso que conecte también vértices de dos componentes conexas distintas, reduciendo otra vez el número total en uno. Si no se respetara esta condición, se producirían ciclos. El algoritmo finaliza cuando solo hay una componente conexa.

8 Tema 8. Tablas Hash

El problema de búsqueda es uno de los problemas más importantes y repetidos. Se quiere buscar un elemento entre un conjunto. En función de si el conjunto está ordenado o no, se puede realizar una **búsqueda lineal** (recorre todos los elementos del primero al último buscando el elemento buscado, por lo que el orden de complejidad es de $O(n)$, una **búsqueda binaria** (solo posible para arrays ordenados, pero reduciendo el orden de complejidad a $O\ log_2(n)$, o una **búsqueda por clave** (con la que se accede a cualquier elemento por medio de su clave, teniendo una complejidad $O(1)$).

Siguiendo este último método, se han implementado las **tablas hash**, que proporcionan un tiempo de búsqueda constante, independiente del número de elementos y aunque éstos no estén ordenados, al acceder a cada uno a través de su clave.

Además, la inserción y el borrado son también constantes.

Esto se consigue obteniendo la posición de cada elemento por medio de una fórmula matemática llamada **función hash**. Ejecutar dicha fórmula para obtener la posición se hace de manera constante, independientemente del tamaño de la tabla, y una vez se sabe la posición, se puede acceder al elemento también de forma constante. Estas son claras **ventajas** sobre el resto de estructuras de datos para almacenar conjuntos de datos.

Sin embargo, como **desventajas** tenemos:

- Los datos se almacenan en un array (estructura estática) por lo que debe fijarse un tamaño máximo desde el principio, que no se podrá sobrepasar. Si el tamaño es demasiado grande, se desperdiciará mucho espacio.
- No hay forma de recorrer los elementos directamente, ya que no tienen porque encontrarse en posiciones consecutivas, pudiendo haber varias posiciones vacías de array entre cada par de elementos.
- Tampoco se pueden almacenar los elementos ordenados, ya que su posición viene dada por el resultado devuelto por la fórmula matemática.
- Se puede dar el caso en el que la función hash asigne a un elemento una posición ya ocupada en el array. A esto se le conoce como**colisión**, y es necesario plantear una solución para redirigir el elemento a otra posición, y tener forma localizarlo después.

Las **claves** (elemento identificativo de los datos que queremos ordenar) pueden ser números o cadenas de caracteres.

8.1 Funciones Hash

Existen diversas funciones hash con sus ventajas e inconvenientes, que se muestran a continuación.

8.1.1 Función hash por módulo

Es muy simple de entender y de implementar: $\boldsymbol{h(K) = K \bmod N}$, siendo **h** la función hash, **K** la clave del dato, **mod** el operando módulo (resto de la división), y **N** el tamaño de la tabla (tamaño del array en el que se insertan los datos).

Sin embargo, a pesar de su sencillez, tiene el gran inconveniente de que ciertos valores de N pueden producir muchas colisiones para muchas claves (aquellos que tengan una factorización con muchos factores).

Una forma de solucionarlo es seleccionar para el valor de N un número primo.

Solo sirve para **claves** formadas por **números enteros**.

8.1.2 Función hash cuadrado

Se eleva la clave al cuadrado y del resultado se seleccionan los dígitos centrales: $\boldsymbol{h(K) = dígitos\_centrales(K^2)}$. Para determinar cuantos dígitos centrales se cogen, se puede utilizar una función matemática en función del tamaño de la tabla, por ejemplo: *dígitos_centrales = log(N).*

En este caso el número de colisiones no va a depender de los factores del valor del tamaño de la tabla tan directamente, sino que toma un carácter un poco más aleatorio, aunque por lo general se reducen para tamaños de tabla grandes.

Solo sirve para **claves** formadas por **números enteros**.

8.1.3 Función hash por plegamiento

Se divide la clave en partes con el mismo número de dígitos (salvo la última, que puede tener menos si la división no es exacta) y se realiza una operación de suma o multiplicación sobre las

partes, y sobre la solución, se queda con las cifras menos significativas.

El número de cifras para realizar las divisiones en partes y el número de cifras menos significativas que se escogen de la solución dependen también del tamaño de la tabla.

Solo sirve para **claves** formadas por **números enteros**.

8.1.4 Función hash por el método de la división

Para claves formadas por **cadenas de caracteres**, una opción es sumar los valores ascii de cada uno de los caracteres que forman las cadena y al valor entero obtenido calcularle la función hash por módulo.

Como en el caso de la función hash por módulo, uno de los **inconvenientes** puede ser el valor del tamaño de la tabla escogido. Además, también presenta problemas con valores de tamaño muy grandes si las cadenas tienes todas pocos caracteres, ya que no se llenará gran parte de la tabla, desperdiciando espacio mientras se pueden estar provocando colisiones en un número reducido de posiciones.

8.1.5 Función hash por suma ponderada

Otro método para calcular la posición cuando las claves está formadas por **cadenas de caracteres** es obtener los valores ascii de cada carácter, multiplicarlo por un número en función de su posición en la cadena y luego sumar los valores. Para evitar obtener valores no válidos, hay que realizar el módulo del valor del tamaño de la tabla para cada valor de cada carácter.

Como el código ascii recoge 256 caracteres, usar una numeración en base 256 puede ser una buena opción. En ese caso, el último carácter de la cadena tendrá simplemente su valor ascii, el siguiente tendrá su valor ascii más 256, el posterior tendrá su valor ascii más 256*2, y así sucesivamente, sumando después todos esos valores.

8.2 Resolución de colisiones

Como se explicó antes, cuando una clave es asignada a una posición ya ocupada se produce una **colisión**, que es necesario resolver para poder asignar una posición válida a dicha clave y poder localizarla después.

Una forma de **evitar las colisiones** primeramente es seleccionar una **buena función hash**. Sin embargo, siempre hay colisiones que son inevitables. Para resolver estas existen dos alternativas: **recolocación** y **encadenamiento**.

8.2.1 Recolocación

Si la posición asignada al elemento está ya ocupada, se busca otra posición.

Hay 3 tipos de recolocaciones:

- **Recolocación simple:** Si la posición asignada está ocupada se prueba en la siguiente, y si también está ocupada, en la siguiente, y así hasta llegar a una posición vacía.

  A la hora de buscar elementos, se sigue el mismo proceso: si no está en la posición correspondiente, se recorren las siguientes hasta encontrarlo, o en el caso de que no esté, hasta recorrer un espacio vacío, o en caso contrario, recorrer toda la tabla.

Esto supone tener que hacer una distinción entre las posiciones que están vacías porque se ha borrado un elemento y las que lo están porque todavía no se insertó ningún elemento en ellas.

Como claras **desventajas** tenemos:

- Por un lado, el coste de la inserción, búsqueda y borrado en el caso de tener que recorrer muchas posiciones.
- Por otro, se pueden forman grandes bloques de posiciones ocupadas seguidas, lo que repercute directamente en la desventaja anterior.

- **Recolocación simple:** Similar a la recolocación simple, pero en vez de avanzar por las posiciones de una en una, se avanza de ***a* en *a*, siendo *a* un número entre 2 y N-1** (con N como valor del tamaño de la tabla).

  Un gran **inconveniente** es que, en función del valor de a, a pesar de haber posiciones vacías, al recorrer la tabla puede no encontrarse ninguna.

  La forma de solucionarlo es seleccionar un valor ***a* primo con *N***.

  Se conoce como **factor de carga** a la relación entre el número de datos almacenados y el tamaño de la tabla. Cuando este valor se mantiene **por debajo de 1/2**, se tiene una media de colisiones constante.

- **Recolocación cuadrática:** Cada vez que se produce una colisión, se busca en la posición resultante de sumar el número de colisión elevado al cuadrado a la posición original. De esta forma se evitan formar bloques de posiciones llenar consecutivas.

  Sin embargo, también es posible no encontrar un posición libre a pesar de que si que las haya, aunque esto solo ocurre cuando la tabla está casi llena. Por ello, es necesario mantener el **factor de carga por debajo de 1/2**, lo que se consigue con la **redispersión**, que se comentará más adelante.

8.2.2 Encadenamiento

Cada posición del vector de datos contiene una lista enlazada, de forma que aunque ya haya elementos en la posición, se puede insertar uno nuevo añadiéndolo al final de la lista sin producir ninguna colisión. por tanto, la **inserción** se realiza siempre en un **tiempo constante**.

Por otra parte, la **búsqueda** y la **eliminación** requieren calcular la función hash y posteriormente recorrer la lista hasta encontrar el elemento, lo que supone un **coste lineal** en el peor de los casos.

A mayores de la reducción general en el tiempo de inserción, otra gran **ventaja** es que no existe un número máximo de elementos, ya que las listas pueden crecer tanto como se necesite.

No obstante, si todos los elementos se distribuyen en unas pocas posiciones del array, se acabarán teniendo listas muy largas en ciertas posiciones, mientras otras se mantienen vacías, reduciendo la eficiencia, al asimilarse a trabajar simplemente con arrays.

Para evitar este problema se debe mantener el **factor de carga por debajo de 3/4**, usando la **redispersión**.

8.3 Redispersión

Cuando en una tabla hash, tanto con recolocación como con encadenamiento, se supera el **factor de carga máximo establecido**, es necesario reducirlo para no empeorar la eficiencia de la tabla hash.

Para ello se realiza la operación de **redispersión**, que **aumenta el tamaño de la tabla**, generalmente al doble, y **recoloca los elementos** en sus nuevas posiciones. Esto supone que sea una **operación costosa**, pero al realizarse pocas veces y mantener la eficiencia de la tabla hash, es **aceptable**.

8.4 Aplicaciones

Algunas de las aplicaciones más típicas de las tablas hash son:

- Tablas de símbolos (variables, por ejemplo) de intérpretes y compiladores.
- Gestión de un conjunto de personas, identificados por su DNI o número de la seguridad social.

9 Tema 9. Estrategias algorítmicas

A la hora de elaborar algoritmos se pretende **maximizar la eficiencia** pero **minimizando los recursos consumidos**. Además, queremos puedan tener el **menor tiempo de ejecución**.

Existen diversos tipos de **estrategias algorítmicas**, que marcan una serie de ideas sobre la elaboración de algoritmos para resolver ciertos problemas siguiendo una filosofía específica.

9.1 Divide y vencerás

Consiste en descomponer un problema en un conjunto de problemas más pequeños que se puedan resolver de forma independiente y combinar sus soluciones para obtener la solución al problema original. Los subproblemas se pueden ir dividiendo a su vez en problemas más pequeños hasta llegar a una solución inmediata.

Ejemplos:

- Multiplicación de enteros largos.
- Multiplicación rápida de matrices.
- Ordenación por mezcla.
- Ordenación rápida.

Un claro problema a resolver es la forma de dividir el problema en subproblemas de la forma más eficiente. Si no es posible hacer esta división de ninguna forma viable, significa que los subproblemas no se pueden resolver de forma independiente y por lo tanto no se puede usar la estrategia de divide y vencerás.

9.2 Algoritmos voraces

Se parte de una solución vacía y se va seleccionando la mejor solución para cada paso dentro de un conjunto de posibles candidatos. Se suelen usar en problemas de optimización.

Componentes de un algoritmo que siga la estrategia voraz:

- **C:** Conjunto de candidatos que todavía se pueden seleccionar.
- **S:** Candidatos ya seleccionados para la solución.

- **R:** Candidatos seleccionador pero rechazados después.
- **solución(S):** Función que comprueba si un conjunto de candidatos **S** es una solución al problema.
- **seleccionar(C):** Función que selecciona el *mejor* elemento de un conjunto **C** de candidatos.
- **factible(S,x):** Función que indica si a partir de un conjunto de candidatos **S**, y añadiendo otro **x** es posible llegar a una solución.
- **Insertar(S,x):** Función que añade un elemento **x** al conjunto **S** de candidatos seleccionados para la solución.
- **Objetivo(S):** Función que dada una solución **S** devuelve el coste asociado a la misma.

Este tipo de algoritmos suelen tener un **orden de complejidad polinomial**, lo que los hace bastante rápidos, sin embargo no aseguran encontrar la solución óptima, por lo que si se necesita ésta, es mejor usar otras estrategias.

9.3 Backtraking o vuelta atrás

Se puede entender como el **opuesto a la estrategia voraz**, ya que añade elementos a la solución parcial y los **elimina** si no obtiene la solución óptima. Por ello, se dice que realiza una **búsqueda exhaustiva y sistemática del espacio de soluciones**. Al probar todas las combinaciones posibles, se vuelve bastante **ineficiente**, pero garantiza encontrar la **solución óptima**. Debido a esto, es muy usado en problema de optimización

El espacio de soluciones se suele representar de forma implícita como un árbol, que puede ser binario, n-ario, permutacional o combinatorio en función de las restricciones del problema.

Componentes de un algoritmo que siga la estrategia de backtraking:

- **s:** Solución parcial hasta cierto punto.
- $s_{INICIAL}$**:** Valor al que se inicializa la solución parcial al empezar el algoritmo. Suele corresponderse con un valor inválido como solución, para evitar confundirla con una solución parcial generada por la ejecución del algoritmo.
- **nivel:** Indica el nivel actual del árbol en el que se encuentra estudiando los candidatos el algoritmo.
- **fin:** Indica si se ha encontrado una solución.
- **Generar(nivel,s):** Función que genera la solución parcial resultado de añadir el siguiente candidato del nivel actual.
- **Solución(nivel,s):** Función que comprueba si la solución parcial actual es una solución válida para el problema.
- **Criterio(nivel,s):** Función que comprueba si a partir de la solución parcial y desde el nivel actual es posible alcanzar una solución válida. En caso contrario, se rechazan los descendientes en el árbol, teniendo lugar una **poda** de esa rama.
- **MasHermanos(nivel,s):** Función que indica si hay más candidatos por estudiar en el nivel actual (*hermanos en el árbol de soluciones*).

- **Retroceder(nivel,s):** Función que a partir de una solución parcial y un nivel en el árbol, retrocede al nivel anterior eliminando de la solución parcial el candidato seleccionado del nivel que se acaba de abandonar.

- En caso de que se quieran almacenar todas las soluciones, es necesario una función **Almacenar(s)** y una variable en la que éstas se almacenen.

Este tipo de algoritmos suelen tener un **orden de ejecución factorial o exponencial**, el cual, como se dijo anteriormente, puede ser bastante ineficiente, por lo que es mejor intentar buscar otro tipo de alternativas algo más rápidas. Una mejora puede ser la de los **mecanismos de poda**, ya que en backtraking su comportamiento suele ser **impredecible**.

9.4 Ramificación y poda

Se puede interpretar como una **mejora de la estrategia de backtracking**, ya que sobre un esquema muy similar, se centra en mejorar las estrategias de ramificación y de poda. Concretamente se tienen los siguientes cambios:

- **Estrategia de ramificación:** El recorrido no tiene porque ser en profundidad como en el caso de backtracking.

- **Estrategia de poda:** La poda se realiza estimando para cada nodo del árbol unas cotas de beneficio que se pueden obtener a partir del mismo, es decir, si seguimos descendiendo por él hacia su descendientes hasta encontrar una solución.

La mejora en la poda supone tener que realizar las estimaciones de las cotas antes de explorar cada nodo, lo que añade algo de complejidad al problema. Sin embargo, estas cotas aseguran hacer mejores podas que en el caso de backtracking, reduciendo el espacio de soluciones y por tanto la cantidad de nodos a estudiar.

Se trata por tanto de compensar lo que se pierde por un lado con grandes ganancias por el otro.

A cada nodo hay que añadirle 3 campos:

- **CS:** Cota superior. Máximo valor que podemos alcanzar a partir del nodo actual.

- **CI:** Cota inferior. Mínimo valor que podemos alcanzar a partir el nodo actual.

- **BE:** Beneficio estimado. Media entre la CS y la CI. Se usa para estimar que nodo estudiar a continuación en la estrategia de ramificación (seleccionando el de mayor o menor BE entre los posibles candidatos).

Como estructura auxiliar se usa una lista denominada **lista de nodos vivos** (**LNV**) en la que se van almacenando todos los nodos generados todavía no estudiados que pueden presentar una solución factible.

Para sacar un nodo de la LNV se sigue una estrategia de ramificación seleccionando el de mayor o menor beneficio estimado, como se comentó antes. Hay cuatro posibles casos:

- **Estrategia LC-FIFO:** Se selecciona el nodo de **menor BE** y, en caso de empate, **el primero que se introdujo**.

- **Estrategia MB-FIFO:** Se selecciona el nodo de **mayor BE** y, en caso de empate, **el primero que se introdujo**.

- **Estrategia LC-LIFO:** Se selecciona el nodo de **menor BE** y, en caso de empate, **el útlimo que se introdujo**.
- **Estrategia MB-LIFO:** Se selecciona el nodo de **mayor BE** y, en caso de empate, **el último que se introdujo**.

Las funciones que intervienen en la estrategia de ramificación y poda son:

- **Selecionar(LNV):** Selecciona un nodo de la LNV siguiendo las estrategia de ramificación.
- **Solución(y):** Para un nodo **y**, comprueba si es una posible solución final del problema.
- **Valor(y):** Para un nodo **y**, devuelve el valor de su solución actual.

Generalmente, este tipo de algoritmos suelen suponer mejoras sobre la técnica de backtracking, sin embargo, en el peor caso (cuando apenas se realizan podas), se pueden generar casi tantos nodos como en el backtracking convencional, añadiéndole a mayores el coste que conlleva calcular las cotas de cada nodo y seleccionar un nodo según la estrategia de ramificación.

Por tanto, debemos escoger entre emplear más tiempo para obtener mejores cotas y así intentar podar más nodos (**estimaciones precisas de las cotas**) o ahorrar tiempo al obtener cotas menos precisas, pero podando por lo general bastantes menos nodos (**estimaciones triviales o poco precisas**).