Algoritmos y Programación II

María Inés Parnisari

30th July 2013

Contents

Part I

Master Theorem

1 Introduction

In the analysis of algorithms, the **master theorem** provides a cookbook solution in asymptotic terms (using Big O notation) for recurrence relations of types that occur in the analysis of many divide and conquer algorithms. Nevertheless, not all recurrence relations can be solved with the use of the master theorem; its generalizations include the Akra–Bazzi method.

Consider a problem that can be solved using recurrence algorithm such as below:

```
procedure T( n :  size of problem ) defined as:
  if (n < k):
    exit
  Do work of amount f(n)
  For a total of $a$ times:
    T(n/b)
end procedure
```

In the above algorithm we are dividing the problem into a number of sub problems recursively, each sub problem being of size n/b. This can be visualized as building a call tree with each node of the tree as an instance of one recursive call and its child nodes being instances of subsequent calls. In the above example, each node would have a number of child nodes. Each node does an amount of work that corresponds to the size of the sub problem n passed to that instance of the recursive call and given by $f(n)$. The total amount of work done by the entire tree is the sum of the work performed by all the nodes in the tree.

2 Generic form

The master theorem concerns recurrence relations of the form: $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ where $a \geq 1,\ b > 1$.

In the application to the analysis of a recursive algorithm, the constants and function take on the following significance:

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- $f\ (n)$ is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.

3 Case 1

If it is true that $f(n) = O\left(n^{log_b(a)-\epsilon}\right)$ for some constant $\varepsilon > 0$ (using Big O notation) it follows that:

$$T(n) = O\left(n^{log_b(a)}\right)$$

4 Case 2

If it is true, for some constant $k \geq 0$ that: $f(n) = O\left(n^{log_b(a)} log_k(n)\right)$ it follows that:

$$T(n) = O\left(n^{log_b(a)} log_{k+1}(n)\right)$$

5 Case 3

If it is true that: $f(n) = \Omega\left(n^{log_b(a)+\epsilon}\right)$ for some constant $\varepsilon > 0$ and if it is also true that: $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for some constant $c < 1$ and sufficiently large n it follows that:

$$T(n) = O\left(f(n)\right)$$

6 Inadmissible equations

The following equations cannot be solved using the master theorem:

- $T(n) = 2^n \cdot T\left(\frac{n}{2}\right) + n^n$
  a is not a constant
- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \frac{n}{log(n)}$
  non-polynomial difference between f(n) and $n^{log_b(a)}$
- $T(n) = \frac{1}{2} \cdot T\left(\frac{n}{2}\right) + n$
  $a < 1$ (cannot have less than one sub problem)
- $T(n) = 64 \cdot T\left(\frac{n}{8}\right) - n^2 \cdot \log(n)$
  f(n) is not positive
- $T(n) = T\left(\frac{n}{2}\right) + n\left(2 - \cos(n)\right)$
  case 3 but regularity violation.

Part II

Data structures

7 Stack

7.1 Introduction

A stack is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: *push* and *pop*. The push operation adds to the top of the list, hiding any items already on the stack, or initializing the stack if it is empty. The pop operation removes an item from the top of the list, and returns this value to the caller. A pop either reveals previously concealed items, or results in an empty list.

A stack is a *restricted data structure*, because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are typically those that have been in the list the longest.

7.2 Operations

```
initialize -> Stack
push (N, Stack) -> Stack
top (Stack) -> (N U ERROR)
remove (Stack) -> Stack
isEmpty (Stack) -> Boolean
```

7.3 Implementations

A stack can be easily implemented either through an array or a linked list. What identifies the data structure as a stack in either case is not the implementation but the interface: the user is only allowed to pop or push items onto the array or linked list, with few other helper operations.

The **array implementation** aims to create an array where the first element (usually at the zero-offset) is the bottom. That is, array[0] is the first element pushed onto the stack and the last element popped off. The program must keep track of the size, or the length of the stack.

The **linked-list** implementation is equally simple and straightforward. In fact, a stack linked-list is much simpler than most linked-list implementations: it requires that we implement a linked-list where only the head node or element can be removed, or popped, and a node can only be inserted by becoming the new head node.

7.4 Applications

7.4.1 Expression evaluation and syntax parsing

Calculators employing reverse Polish notation use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form may be accomplished using a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

8 Queue

8.1 Introduction

A **queue** is a particular kind of collection in which the entities are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that whenever an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.

Common implementations are circular buffers and linked lists.

8.2 Operations

```
initialize -> Stack
pop (Stack) -> Stack
push (Stack) -> Stack
front (Queue) -> (N U ERROR)
remove (Stack) -> Stack
isEmpty (Queue) -> Boolean
```

9 Linked list

9.1 Introduction

A **linked list** is a data structure that consists of a sequence of data records such that in each record there is a field that contains a reference (i.e., a *link*) to the next record in the sequence. Linked lists are among the simplest and most common data structures; they provide an easy implementation for several important abstract data structures, including stacks, queues, associative arrays, and symbolic expressions.

The principal benefit of a linked list over a conventional array is that the order of the linked items may be different from the order that the data items are stored in memory or on disk. For that reason, linked lists allow insertion and removal of nodes at any point in the list, with a constant number of operations.

On the other hand, linked lists by themselves do not allow random access to the data, or any form of efficient indexing. Thus, many basic operations — such as obtaining the last node of the list, or finding a node that contains a given datum, or locating the place where a new node should be inserted — may require scanning most of the list elements.

9.2 Basic concepts and nomenclature

Each record of a linked list is often called an **element** or **node**. The field of each node that contains the address of the next node is usually called the ***next* link** or ***next* pointer**. The remaining fields are known as the **data**, **information**, **value**, **cargo**, or **payload** fields. The **head** of a list is its first node, and the **tail** is the list minus that node (or a pointer thereto).

9.2.1 Linear and circular lists

In the last node of a list, the link field often contains a **null** reference, a special value that is interpreted by programs as meaning "there is no such node". A less common convention is to make it point to the first node of the list; in that case the list is said to be **circular** or **circularly linked**; otherwise it is said to be **open** or **linear**.

9.2.2 Singly-, doubly-, and multiply-linked lists

Singly-linked lists contain nodes which have a data field as well as a *next* field, which points to the next node in the linked list.

In a **doubly-linked list**, each node contains, besides the next-node link, a second link field pointing to the *previous* node in the sequence. The two links may be called **forward**(**s**) and **backwards**, or **next** and **prev**(**ious**).

The technique known as XOR-linking allows a doubly-linked list to be implemented using a single link field in each node. However, this technique requires the ability to do bit operations on addresses, and therefore may not be available in some high-level languages.

In a **multiply-linked list**, each node contains two or more link fields, each field being used to connect the same set of data records in a different order (e.g., by name, by department, by date of birth, etc.). (While doubly-linked lists can be seen as special cases of multiply-linked list, the fact that the two orders are opposite to each other leads to simpler and more efficient algorithms, so they are usually treated as a separate case.)

In the case of a doubly circular linked list, the only change that occurs is the end, or "tail" of the said list is linked back to the front, "head", of the list and vice versa.

9.3 Tradeoffs

As with most choices in computer programming and design, no method is well suited to all circumstances. A linked list data structure might work well in one case, but cause problems in another. This is a list of some of the common tradeoffs involving linked list structures.

9.3.1 Linked lists vs. dynamic arrays

A *dynamic array* is a data structure that allocates all elements contiguously in memory, and keeps a count of the current number of elements. If the space reserved for the dynamic array is exceeded, it is reallocated and (possibly) copied, an expensive operation.

Linked lists have several advantages over dynamic arrays. Insertion of an element at a specific point of a list is a constant-time operation, whereas insertion in a dynamic array at random locations will require moving half of the elements on average, and all the elements in the worst case. While one can "delete" an element from an array in constant time by somehow marking its slot as "vacant", this causes fragmentation that impedes the performance of iteration.

Moreover, arbitrarily many elements may be inserted into a linked list, limited only by the total memory available; while a dynamic array will eventually fill up its underlying array data structure and have to reallocate — an expensive operation, one that may not even be possible if memory is fragmented. Similarly, an array from which many elements are removed may have to be resized in order to avoid wasting too much space.

On the other hand, dynamic arrays (as well as fixed-size array data structures) allow constant-time random access, while linked lists allow only sequential access to elements. Singly-linked lists, in fact, can only be traversed in one direction. This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly, such as heapsort. Sequential access on arrays and dynamic arrays is also faster than on linked lists on many machines, because they have optimal locality of reference and thus make good use of data caching.

Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values, because the storage overhead for the links may exceed by a factor of two or more the size of the data. In contrast, a dynamic array requires only the space for the data itself (and a very small amount of control data). It can also be slow, and with a naive allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using memory pools.

9.4 Speeding up search

Finding a specific element in a linked list, even if it is sorted, normally requires $O(n)$ time (linear search). This is one of the primary disadvantages of linked lists over other data structures.

In an unordered list, one simple heuristic for decreasing average search time is the *move-to-front heuristic*, which simply moves an element to the beginning of the list once it is found. This scheme, handy for creating simple caches, ensures that the most recently used items are also the quickest to find again.

Another common approach is to "index" a linked list using a more efficient external data structure. For example, one can build a red-black tree or hash table whose elements are references to the linked list nodes. Multiple such indexes can be built on a single list. The disadvantage is that these indexes may need to be updated each time a node is added or removed (or at least, before that index is used again).

10 Hash table

10.1 Introduction

A **hash table** is a data structure that uses a hash function to map identifying values, known as keys (e.g., a person's name), to their associated values (e.g., their telephone number). The hash function is used to transform the key into the index (the *hash*) of an array element (the *slot* or *bucket*) where the corresponding value is to be sought.

Ideally, the hash function should map each possible key to a unique slot index, but this ideal is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after creation). Most hash table designs assume that *hash collisions* - the situation where different keys happen to have the same hash value—are normal occurrences and must be accommodated in some way.

In a well-dimensioned hash table, the average cost for each lookup is independent of the number of elements stored in the table.

10.2 Load factor

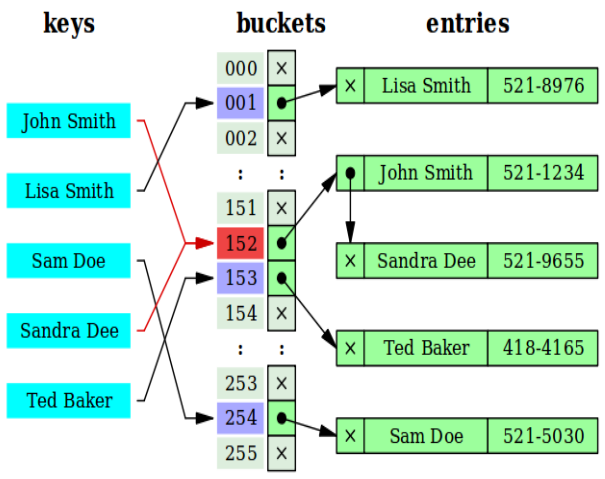
The performance of most collision resolution methods does not depend directly on the number n of stored entries, but depends strongly on the table's *load factor*, the ratio n/s between n and the size s of its bucket array. With a good hash function, the average lookup cost is nearly constant as the load factor increases from 0 up to 0.7 or so. Beyond that point, the probability of collisions and the cost of handling them increases.

On the other hand, as the load factor approaches zero, the size of the hash table increases with little improvement in the search cost, and memory is wasted.

10.3 Open hashing

In the strategy known as *open hashing* each slot of the bucket array is a pointer to a linked list that contains the key-value pairs that hashed to the same location. Lookup requires scanning the list for an entry with the given key. Insertion requires adding a new entry record to either end of the list belonging to the hashed slot. Deletion requires searching the list and removing the element.

Figure 1: Open hashing



The cost of a table operation is that of scanning the entries of the selected bucket for the desired key. If the distribution of keys is sufficiently uniform, the *average* cost of a lookup depends only on the average number of keys per bucket—that is, on the load factor.

Chained hash tables remain effective even when the number of entries n is much higher than the number of slots. Their performance degrades more gracefully (linearly) with the load factor. For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list, and possibly even faster than a balanced search tree.

For separate-chaining, the worst-case scenario is when all entries were inserted into the same bucket, in which case the hash table is ineffective and the cost is that of searching the bucket data structure. If the latter is a linear list, the lookup procedure may have to scan all its entries; so the worst-case cost is proportional to the number n of entries in the table.

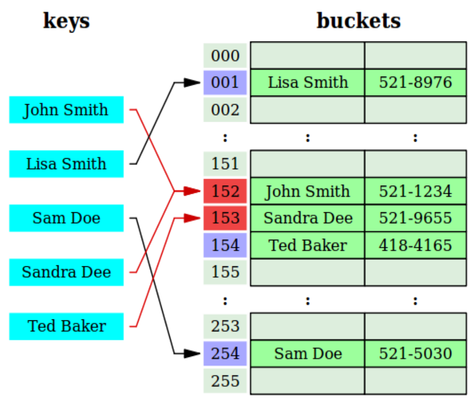
The bucket chains are often implemented as ordered lists, sorted by the key field; this choice approximately halves the average cost of unsuccessful lookups, compared to an unordered list. However, if some keys are much more likely to come up than others, an unordered list with move-to-front heuristic may be more effective.

Chained hash tables also inherit the disadvantages of linked lists. When storing small keys and values, the space overhead of the next pointer in each entry record can be significant. An additional disadvantage is that traversing a linked list has poor cache performance, making the processor cache ineffective.

10.4 Closed hashing

In another strategy, called closed hashing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.

Figure 2: Closed hashing



Well-known probe sequences include:

- Linear probing, in which the interval between probes is fixed (usually 1)
- Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
- Double hashing, in which the interval between probes is computed by another hash function

A drawback of all these closed hashing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance seriously degrades when the load factor grows beyond 0.7 or so. For many applications, these restrictions mandate the use of dynamic resizing, with its attendant costs.

Closed hashing schemes also put more stringent requirements on the hash function: besides distributing the keys more uniformly over the buckets, the function must also minimize the clustering of hash values that are consecutive in the probe order.

Open addressing only saves memory if the entries are small (less than 4 times the size of a pointer) and the load factor is not too small. If the load factor is close to zero (that is, there are far more buckets than stored entries), open addressing is wasteful even if each entry is just two words.

Open addressing avoids the time overhead of allocating each new entry record, and can be implemented even in the absence of a memory allocator. It also avoids the extra indirection required to access the first entry of each bucket (which is usually the only one). It also has better locality of reference, particularly with linear probing. With small record sizes, these factors can yield better performance than chaining, particularly for lookups.

Generally speaking, closed hashing is better used for hash tables with small records that can be stored within the table. They are particularly suitable for elements of one word or less. In cases where the tables are expected to have high load factors, the records are large, or the data is variable-sized, chained hash tables often perform as well or better.

10.5 Dynamic resizing

To keep the load factor under a certain limit e.g., under 3/4, many table implementations expand the table when items are inserted. Resizing is accompanied by a full or incremental table **rehash** whereby existing items are mapped to new bucket locations.

To limit the proportion of memory wasted due to empty buckets, some implementations also shrink the size of the table - followed by a **rehash** - when items are deleted.

10.5.1 Resizing by copying all entries

A common approach is to automatically trigger a complete resizing when the load factor exceeds some threshold r_{max}. Then a new larger table is allocated, all the entries of the old table are removed and inserted into this new table, and the old table is returned to the free storage pool. Symmetrically, when the load factor falls below a second threshold r_{min}, all entries are moved to a new smaller table.

If the table size increases or decreases by a fixed percentage at each expansion, the total cost of these resizings, amortized over all insert and delete operations, is still a constant, independent of the number of entries n and of the number m of operations performed.

10.6 Performance analysis

In the simplest model, the hash function is completely unspecified and the table does not resize. For the best possible choice of hash function, a table of size n with open addressing has no collisions and holds up to n elements, with a single comparison for successful lookup, and a table of size n with chaining and k keys has the minimum $max(0, k-n)$ collisions and $O(1+k/n)$ comparisons for lookup. For the worst choice of hash function, every insertion causes a collision, and hash tables degenerate to linear search, with $\Omega(k)$ amortized comparisons per insertion and up to k comparisons for a successful lookup.

Adding rehashing to this model is straightforward. As in a dynamic array, geometric resizing by a factor of b implies that only k/b^i keys are inserted i or more times, so that the total number of insertions is bounded above by $bk/(b-1)$, which is $O(k)$. By using rehashing to maintain $k < n$, tables using both chaining and open addressing can have unlimited elements and perform successful lookup in a single comparison for the best choice of hash function.

In more realistic models the hash function is a random variable over a probability distribution of hash functions, and performance is computed on average over the choice of hash function. When this distribution is uniform, the assumption is called "simple uniform hashing" and it can be shown that hashing with chaining requires $\Theta(1+k/n)$ comparisons on average for an unsuccessful lookup, and hashing with open addressing requires $\Theta\left(\frac{1}{1-k/n}\right)$. Both these bounds are constant if we maintain $k/n < c$ using table resizing, where c is a fixed constant less than 1.

10.7 Features

10.7.1 Advantages

The main advantage of hash tables over other table data structures is speed. This advantage is more apparent when the number of entries is large (thousands or more). Hash tables are particularly efficient when the maximum number of entries can be predicted in advance, so that the bucket array can be allocated once with the optimum size and never resized.

10.7.2 Disadvantages

Hash tables can be more difficult to implement than self-balancing binary search trees. Choosing an effective hash function for a specific application is more an art than a science. In closed hash tables it is fairly easy to create a poor hash function.

Although operations on a hash table take constant time on average, the cost of a good hash function can be significantly higher than the inner loop of the lookup algorithm for a sequential list or search tree. Thus hash tables are not effective when the number of entries is very small.

The entries stored in a hash table can be enumerated efficiently (at constant cost per entry), but only in some pseudo-random order. Therefore, there is no efficient way to efficiently locate an entry whose key is *nearest* to a given key. Listing all n entries in some specific order generally requires a separate sorting step, whose cost is proportional to $\log(n)$ per entry. In comparison, ordered search trees have lookup and insertion cost proportional to $\log(n)$, but allow finding the nearest key at about the same cost, and *ordered* enumeration of all entries at constant cost per entry.

If the keys are not stored (because the hash function is collision-free), there may be no easy way to enumerate the keys that are present in the table at any given moment.

Although the *average* cost per operation is constant and fairly small, the cost of a single operation may be quite high. In particular, if the hash table uses dynamic resizing, an insertion or deletion operation may occasionally take time proportional to the number of entries.

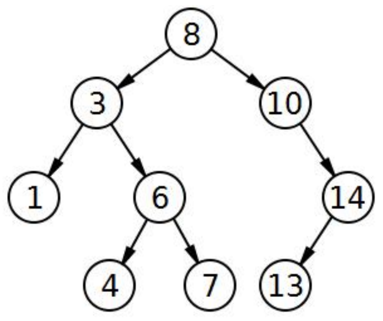
Hash tables in general exhibit poor locality of reference—that is, the data to be accessed is distributed seemingly at random in memory. Because hash tables cause access patterns that jump around, this can trigger microprocessor cache misses that cause long delays. Compact data structures such as arrays, searched with linear search, may be faster if the table is relatively small and keys are integers or other short strings.

Hash tables become quite inefficient when there are many collisions.

11 Binary Search Tree

11.1 Introduction

A **binary search tree** (**BST**) is a node-based binary tree data structure which has the following properties:



- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

  Generally, the information represented by each node is a **record** rather than a single data element. However, for sequencing purposes, nodes are compared according to their **keys** rather than any part of their associated records.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

12 AVL Tree

12.1 Introduction

An **AVL tree** is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; therefore. Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

The AVL tree is named after its two inventors, G.M. Adelson-Velskii and E.M. Landis.

The **balance factor** of a node is the height of its left subtree minus the height of its right subtree (sometimes opposite) and a node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.

12.2 Operations

The basic operations of an AVL tree involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications are preceded or followed by one or more operations called tree rotations, which help to restore the height balance of the subtrees.

12.2.1 Lookup

Lookup in an AVL tree is performed exactly as in an unbalanced binary search tree. Because of the height-balancing of the tree, a lookup takes O(log n) time.

12.2.2 Insertion

After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL. For each node checked, if the balance factor remains -1, 0, or $+1$ then no rotations are necessary. However, if the balance factor becomes ± 2 then the subtree rooted at this node is unbalanced. If insertions are performed serially, after each insertion, at most two tree rotations are needed to restore the entire tree to the rules of AVL.

There are four cases which need to be considered, of which two are symmetric to the other two. Let P be the root of the unbalanced subtree. Let R be the right child of P. Let L be the left child of P.

Right-Right case and **Right-Left case**: If the balance factor of P is -2, then the right subtree outweighs the left subtree of the given node, and the balance factor of the right child (R) must be checked. If the balance factor of R is ≤ 0, a **left rotation** is needed with P as the root. If the balance factor of R is $+1$, a **double left rotation** (with respect to P) is needed. The first rotation is a right rotation with R as the root. The second is a left rotation with P as the root.

Left-Left case and **Left-Right case**: If the balance factor of P is $+2$, then the left subtree outweighs the right subtree of the given node, and the balance factor of the left child (L) must be checked. If the balance factor of L is ≥ 0, a **right rotation** is needed with P as the root. If the balance factor of L is -1, a **double right rotation** (with respect to P) is needed. The first rotation is a left rotation with L as the root. The second is a right rotation with P as the root.

12.2.3 Deletion

If the node is a leaf or has only one child, remove it. Otherwise, replace it with either the largest in its left subtree (inorder predecessor) or the smallest in its right subtree (inorder successor), and remove that node. The node that was found as a replacement has at most one subtree. After deletion, retrace the path back up the tree (parent of the replacement) to the root, adjusting the balance factors as needed.

As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

In addition to the balancing described above for insertions, if the balance factor for the tree is 2 and that of the left subtree is 0, a right rotation must be performed on P. The mirror of this case is also necessary.

The retracing can stop if the balance factor becomes -1 or $+1$ indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. If the balance factor becomes -2 or $+2$ then the subtree is unbalanced and needs to be rotated to fix it. If the rotation leaves the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

The time required is O(log n) for lookup, plus a maximum of O(log n) rotations on the way back to the root, so the operation can be completed in O(log n) time.

13 Priority Queue

13.1 Introduction

A **priority queue** is an abstract data type in computer programming that supports the following three operations:

- *insert WithPriority*: add an element to the queue with an associated priority .
- *getNext*: remove the element from the queue that has the *highest priority*, and return it.
- *peekAtNext:* look at the element with *highest priority* without removing it.

One can imagine a priority queue as a modified queue, but when one would get the next element off the queue, the highest-priority one is retrieved first.
To get better performance, priority queues typically use a heap as their backbone, giving O(log n) performance for inserts and removals (and O(1) for peekAtNext). O(n) insertions and O(n) extractions gives a total time of $O(n^2)$. Alternatively, if a self-balancing binary search tree is used, all three operations take O(log n) time.

13.2 Relationship to sorting algorithms

The semantics of priority queues naturally suggest a sorting method: insert all the elements to be sorted into a priority queue, and sequentially remove them; they will come out in sorted order. This is actually the procedure used by several sorting algorithms, once the layer of abstraction provided by the priority queue is removed. This sorting method is equivalent to the following sorting algorithms:

- Heapsort if the priority queue is implemented with a heap.
- Selection sort if the priority queue is implemented with an unordered array.
- Insertion sort if the priority queue is implemented with an ordered array.
- Tree sort if the priority queue is implemented with a self-balancing binary search tree.

A sorting algorithm can also be used to implement a priority queue.

14 Heap

14.1 Introduction

In computer science, a **heap** is a specialized tree-based data structure that satisfies the *heap property:* if B is a child node of A, then $\text{key}(A) \geq \text{key}(B)$. This implies that an element with the greatest key is always in the root node, and so such a heap is sometimes called a *max-heap*. (Alternatively, if the comparison is reversed, the smallest element is always in the root node, which results in a *min-heap*.) The several variants of heaps are the prototypical most efficient implementations of the abstract data type priority queues.

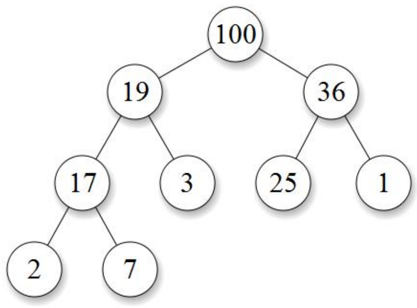


Figure 3: Heap

Heaps are usually implemented in an array, and do not require pointers between elements.

The operations commonly performed with a heap are:

- *delete-max* or *delete-min*: removing the root node of a max- or min-heap, respectively
- *increase-key* or *decrease-key*: updating a key within a max- or min-heap, respectively
- *insert*: adding a new key to the heap
- *merge*: joining two heaps to form a valid new heap containing all the elements of both.

Heaps are used in the sorting algorithm heapsort.

14.2 Applications

The heap data structure has many applications.

- Heapsort: One of the best sorting methods being in-place and with no quadratic worst-case scenarios.
- Selection algorithms: Finding the min, max, both the min and max, median, or even the k-th largest element can be done in linear time (often constant time) using heaps.
- Graph algorithms: By using heaps as internal traversal data structures, run time will be reduced by polynomial order.

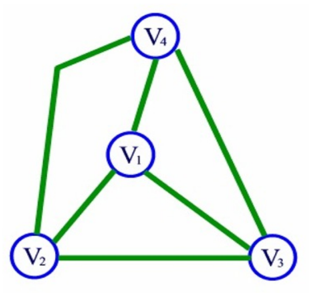
Full and almost full binary heaps may be represented in a very space-efficient way using an array alone. The first (or last) element will contain the root. The next two elements of the array contain its children. The next four contain the four children of the two child nodes, etc. Thus the children of the node at position n would be at positions 2n and 2n+1 in a one-based array, or 2n+1 and 2n+2 in a zero-based array. This allows moving up or down the tree by doing simple index computations. Balancing a heap is done by swapping elements which are out of order. As we can build a heap from an array without requiring extra memory (for the nodes, for example), heapsort can be used to sort an array in-place.

15 Graph

15.1 Introduction

In computer science, a **graph** is an abstract data structure that is meant to implement the graph concept from mathematics.

A graph data structure consists mainly of a finite (and possibly mutable) set of ordered pairs, called **edges** or **arcs**, of certain entities called **nodes** or **vertices**. As in mathematics, an edge (x,y) is said to **point** or **go from** x **to** y. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references.



A graph data structure may also associate to each edge some **edge value**, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

15.2 Operations

The basic operations provided by a graph data structure G usually include:

- adjacent(G, x,y): tests whether there is an edge from node x to node y.
- neighbours(G, x): lists all nodes y such that there is an edge from x to y.
- add(G, x,y): adds to G the edge from x to y, if it is not there.
- delete(G, x,y): removes the edge from x to y, if it is there.
- get_node_value(G, x): returns the value associated with the node x.
- set_node_value(G, x, a): sets the value associated with the node x to a.

Structures that associate values to the edges usually also provide:

- get_edge_value(G, x,y): returns the value associated to the edge (x,y).
- et_edge_value(G, x,y,v): sets the value associated to the edge (x,y) to v.

15.3 Representations

Different data structures for the representation of graphs are used in practice:

- **Adjacency list** - An adjacency list is implemented as an array of lists, with one list of destination nodes for each source node.
- **Incidence list** - A variant of the adjacency list that allows for the description of the edges at the cost of additional edges.
- **Adjacency matrix** - A two-dimensional Boolean matrix, in which the rows and columns represent source and destination vertices and entries in the matrix indicate whether an edge exists between the vertices associated with that row and column.
- **Incidence matrix** - A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The array entries indicate if both are related, i.e. incident.

15.4 Algorithms

Typical higher-level operations associated with graphs are: finding a path between two nodes, like depth-first search and breadth-first search and finding the shortest path from one node to another, like Dijkstra's algorithm. A solution to finding the shortest path from each node to every other node also exists in the form of the Floyd-Warshall algorithm.

A directed graph can be seen as a flow network, where each edge has a capacity and each edge receives a flow. The Ford-Fulkerson algorithm is used to find out the maximum flow from a source to a sink in a graph.

Part III

Design techniques

16 Divide and Conquer

16.1 Introduction

Divide and conquer (**D&C**) is an important algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g. Karatsuba), syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFTs).

The name "divide and conquer" is sometimes applied also to algorithms that reduce each problem to only one subproblem, such as the binary search algorithm for finding a record in a sorted list. These algorithms can be implemented more efficiently than general divide-and-conquer algorithms; in particular, if they use tail recursion, they can be converted into simple loops.

The correctness of a divide and conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

16.2 Implementation

Recursion: Divide-and-conquer algorithms are naturally implemented as recursive procedures. In that case, the partial sub-problems leading to the one currently being solved are automatically stored in the procedure call stack.

Explicit stack: Divide and conquer algorithms can also be implemented by a non-recursive program that stores the partial sub-problems in some explicit data structure, such as a stack, queue, or priority queue. This approach allows more freedom in the choice of the sub-problem that is to be solved next, a feature that is important in some applications — e.g. in breadth-first recursion and the branch and bound method for function optimization.

Stack size: In recursive implementations of D&C algorithms, one must make sure that there is sufficient memory allocated for the recursion stack, otherwise the execution may fail because of stack overflow.

Choosing the base cases: In any recursive algorithm, there is considerable freedom in the choice of the *base cases*, the small subproblems that are solved directly in order to terminate the recursion.

Sharing repeated subproblems: For some problems, the branched recursion may end up evaluating the same sub-problem many times over. In such cases it may be worth identifying and saving the solutions to these overlapping subproblems, a technique commonly known as memoization. Followed to the limit, it leads to bottom-up divide-and-conquer algorithms such as dynamic programming and chart parsing.

17 Dynamic programming

17.1 Introduction

Dynamic programming is a method for solving complex problems by breaking them down into simpler steps. It is applicable to problems exhibiting the properties of overlapping subproblems which are only slightly smaller and optimal substructure. When applicable, the method takes far less time than naive methods.

Top-down dynamic programming simply means storing the results of certain calculations, which are later used again since the completed calculation is a sub-problem of a larger calculation. Bottom-up dynamic programming involves formulating a complex calculation as a recursive series of simpler calculations.

17.2 Dynamic programming in computer programming

There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping subproblems which are only slightly smaller. When the overlapping problems are, say, half the size of the original problem the strategy is called "divide and conquer" rather than "dynamic programming". This is why mergesort, quicksort, and finding all matches of a regular expression are not classified as dynamic programming problems.

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its subproblems. Consequently, the first step towards devising a dynamic programming solution is to check whether the problem exhibits such optimal substructure. Such optimal substructures are usually described by means of recursion.

Overlapping subproblems means that the space of subproblems must be small, that is, any recursive algorithm solving the problem should solve the same subproblems over and over, rather than generating new subproblems. Note that the subproblems must be only **slightly** smaller than the larger problem; when they are a multiplicative factor smaller the problem is no longer classified as dynamic programming.

17.3 Example: Fibonacci

Algorithm 1 Fibonacci

```
function fib(n):
  previousFib := 0
  currentFib := 1
  if (n == 0): return 0
  if (n == 1): return 1
  repeat (n-1) times:
    newFib := previous Fib + currentFib
    previous Fib := currentFib
    currentFib := newFib
  return currentFib
```

18 Greedy algorithms

18.1 Introduction

A **greedy algorithm** is any algorithm that follows the problem solving metaheuristic of making the locally optimal choice at each stage with the hope of finding the global optimum.

For example, applying the greedy strategy to the travelling salesman problem yields the following algorithm: "At each stage visit the unvisited city nearest to the current city".

18.2 Specifics

In general, greedy algorithms have five pillars:

1. A *candidate set*, from which a solution is created.
2. A *selection* function, which chooses the best candidate to be added to the solution.
3. A *feasibility* function, that is used to determine if a candidate can be used to contribute to a solution.
4. An *objective* function, which assigns a value to a solution, or a partial solution.
5. A *solution* function, which will indicate when we have discovered a complete solution.

Most problems for which they work well have two properties:

- Greedy choice property: The choice made by a greedy algorithm may depend on choices made so far but not on future choices or all the solutions to the subproblem. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices. This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution. After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.
- Optimal substructure: A problem has optimal substructure if the best next move always leads to the optimal solution. An example of 'non-optimal substructure' would be a situation where capturing a queen in chess (good next move) will eventually lead to the loss of the game (bad overall move).

18.3 Cases of failure

For many other problems, greedy algorithms fail to produce the optimal solution, and may even produce the *unique worst possible* solution.

Imagine the coin example with only 25-cent, 10-cent, and 4-cent coins. The greedy algorithm would not be able to make change for 41 cents, since after committing to use one 25-cent coin and one 10-cent coin it would be impossible to use 4-cent coins for the balance of 6 cents. Whereas a person or a more sophisticated algorithm could make change for 41 cents change with one 25-cent coin and four 4-cent coins.

Greedy algorithms can be characterized as being 'short sighted', and as 'non-recoverable'. Despite this, greedy algorithms are best suited for simple problems (e.g. giving change). It is important, however, to note that the greedy algorithm can be used as a selection algorithm to prioritize options within a search, or branch and bound algorithm.

18.4 Applications

Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.

If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimisation methods like dynamic programming.

18.5 Algorithm

Algorithm 2 Algoritmo greedy

```
function greedy (candidates):
  solutionSet := {}
  while notEmpty (candidates):
    x := selectFrom (candidates)
    if (not x):
      break
    if feasible(solutionSet + x):
      solutionSet := solutionSet + {x}
    candidates := candidates - {x}
  return solutionSet
```

19 Backtracking

19.1 Introduction

Backtracking is a general algorithm for finding all (or some) solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution.

Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. It is useless, for example, for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate a large number of candidates with a single test.

Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, Sudoku, and many other puzzles. It is often the most convenient technique for the knapsack problem and other combinatorial optimization problems.

Backtracking depends on user-given "black box procedures" that define the problem to be solved, the nature of the partial candidates, and how they are extended into complete candidates.

19.2 Description of the method

The backtracking algorithm enumerates a set of *partial candidates* that, in principle, could be *completed* in various ways to give all the possible solutions to the given problem. The completion is done incrementally, by a sequence of *candidate extension steps.*

Conceptually, the partial candidates are the nodes of a tree structure, the *potential search tree.* Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further.

The backtracking algorithm traverses this search tree recursively, from the root down, in depth-first order. At each node c, the algorithm checks whether c can be completed to a valid solution. If it cannot, the whole sub-tree rooted at c is skipped. Otherwise, the algorithm (1) checks whether c itself is a valid solution, and if so reports it to the user; and (2) recursively enumerates all sub-trees of c.

Therefore, the *actual search tree* that is traversed by the algorithm is only a part of the potential tree. The total cost of the algorithm is basically the number of nodes of the actual tree times the cost of obtaining and processing each node. This fact should be considered when choosing the potential search tree and implementing the pruning test.

19.3 Pseudocode

In order to apply backtracking to a specific class of problems, one must provide the data P for the particular instance of the problem that is to be solved, and six procedural parameters, *root*, *reject*, *accept*, *first*, *next*, and *output*. These procedures should take the instance data P as a parameter and should do the following:

1. *root*: return the partial candidate at the root of the search tree.
2. *reject*(P,c): return *true* only if the partial candidate c is not worth completing.
3. *accept*(P,c): return *true* if c is a solution of P, and *false* otherwise.
4. *first*(P,c): generate the first extension of candidate c.
5. *next*(P,s): generate the next alternative extension of a candidate, after the extension s.
6. *output*(P,c): use the solution c of P, as appropriate to the application.

The backtracking algorithm reduces then to the call *backtracking*(*root*(P)), where *backtracking* is the following recursive procedure:

Algorithm 3 Algoritmo de backtracking

```
function backtracking(c):
  if (reject (P,c)):
    return
  if (accept (P,c)):
    output (P,c)
  s := first (P,c)
  while (s != null):
    backtracking (s)
    s := next (P,s)
```

Part IV

Graphs

20 DFS (Depth-First Search)

20.1 Introduction

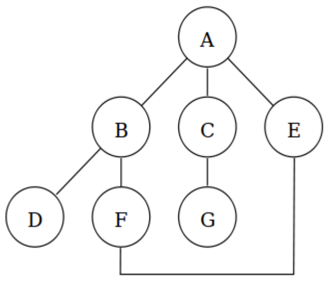
Depth-first search (**DFS**) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time $O(|V|+|E|)$, linear in the size of the graph. In these applications it also uses space $O(|V|)$ in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices. Thus, in this setting, the time and space bounds are the same as for breadth first search and the choice of which of these two algorithms to use depends less on their complexity and more on the different properties of the vertex orderings the two algorithms produce.

20.2 Example

A depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously-visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G.



Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

20.3 Output of a depth-first search

The most natural result of a depth first search of a graph is a spanning tree of the vertices reached during the search. Based on this spanning tree, the edges of the original graph can be divided into three classes: **forward edges**, which point from a node of the tree to one of its descendants, **back edges**, which point from a node to one of its ancestors, and **cross edges**, which do neither. Sometimes **tree edges**, edges which belong to the spanning tree itself, are classified separately from forward edges.

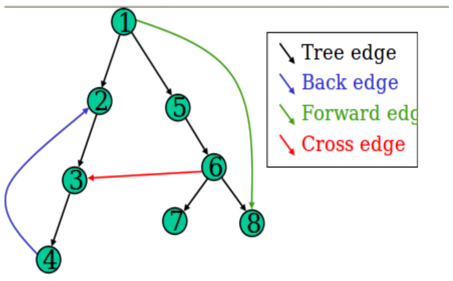


Figure 4: Output of DFS

20.3.1 Vertex orderings

It is also possible to use the depth-first search to linearly order the vertices of the original graph (or tree). There are three common ways of doing this:

- A **preordering** is a list of the vertices in the order that they were first visited by the depth-first search algorithm. This is a compact and natural way of describing the progress of the search, as was done earlier in this article.
- A **postordering** is a list of the vertices in the order that they were *last* visited by the algorithm.
- A **reverse postordering** is the reverse of a postordering, i.e. a list of the vertices in the opposite order of their last visit. When searching a tree, reverse postordering is the same as preordering, but in general they are different when searching a graph.

20.4 Algorithm

Algorithm 4 DFS

```
function dfs (nodo, origen):
  visitar (nodo)
  for v in nodo.adyacentes:
    if (v.visitado == False):
      visitar(v)
```

21 BFS (Breadth-First Search)

21.1 Introduction

In graph theory, **breadth-first search** (**BFS**) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, it explores their unexplored neighbour nodes, and so on, until it finds the goal.

BFS is an uninformed search method that aims to expand and examine all nodes of a graph or combination of sequences by systematically searching through every solution. In other words, it exhaustively searches the entire graph or sequence without considering the goal until it finds it. It does not use a heuristic algorithm.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO (i.e., First In, First Out) queue. In typical implementations, nodes that have not yet been examined for their neighbours are placed in some container (such as a queue or linked list) called “open” and then once examined are placed in the container “closed”.

- Space complexity: When the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as $O(|E|+|V|)$ where $|E|$ is the number of edges, and $|V|$ is the number of vertices. Since it is exponential in the depth of the graph, breadth-first search is often impractical for large problems on systems with bounded space.
- Time complexity: The time complexity can be expressed as $O(|E|+|V|)$ since every vertex and every edge will be explored in the worst case.

21.2 Algorithm

Algorithm 5 BFS

```
function bfs (grafo, fuente):
  cola := Cola()
  cola.push (fuente)
  fuente.visitado := True
  while (cola):
    v := cola.pop()
    for arista e incidente en v in grafo:
      w := el otro fin de e
      if (w.visitado == False):
        w.visitado := True
        cola.push (w)
```

22 Orden topológico

22.1 Introducción

Una **ordenación topológica** de un grafo acíclico G dirigido es una ordenación lineal de todos los nodos de G que conserva la unión entre vértices del grafo G original. La condición que el grafo no contenga ciclos es importante, ya que no se puede obtener ordenación topológica de grafos que contengan ciclos. Para poder encontrar la ordenación topológica del grafo G deberemos aplicar una modificación del algoritmo de búsqueda en profundidad (DFS).

Los algoritmos usuales para el ordenamiento topológico tienen un tiempo de ejecución de la cantidad de nodos más la cantidad de aristas: $O(|V| + |E|)$.

22.2 Algoritmo[1]

Algorithm 6 Orden topológico

```
function orden_topologico (grafo):
  solucion := []
  for v in grafo.vertices:
    for w adyacente a v:
      v.grado += 1

  // Encolamos los vertices tales que gr[+](v)=0
  cola.encolar (v, v.grado == 0)
  while (cola):
    v = desencolar (cola)
    solucion.append(v)
    for w adyacente a v:
      w.grado -= 1
      if (w.grado == 0):
        cola.encolar(w)
  return solucion
```

[1]Orden topológico: http://www.youtube.com/watch?v=4qPaEfpIj5k

23 Dijkstra's algorithm

23.1 Introduction

Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. This algorithm is often used in routing.

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined.

The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and extract minimum from Q is simply a linear search through all vertices in Q. In this case, the running time is $O\left(|V|^2 + |E|\right) = O\left(|V|^2\right)$.

23.2 Algorithm[2]

Algorithm 7 Algoritmo de Dijkstra

```
function dijkstra (grafo, v_inicial):
  cola = Cola()
  distancia = {}
  camino = {}

  grafo.inicializar_vertices()
  cola.put(v_inicial)
  for v in grafo.vertices:
    distancia[v] = INFINITO
  distancia[v_inicial] = 0

  while (not cola.vacia()):
    v = cola.get()
    v.visitado = False
    for v, peso in grafo.lista_de_adyacencias[v]:
      if (distancia[v] > distancia[v] + peso):
        distancia[v] = distancia[v] + peso
        camino[v] = v
        if (not v.visitado):
          cola.put(v)
          v.visitado = True
  return (camino, distancia)
```

[2]Dijkstra: http://www.youtube.com/watch?v=6rl0ghgPfK0

24 Bellman-Ford algorithm

24.1 Introduction

The **Bellman–Ford algorithm** computes single-source shortest paths in a weighted digraph. For graphs with only non-negative edge weights, the faster Dijkstra's algorithm also solves the problem. Thus, Bellman–Ford is used primarily for graphs with negative edge weights.

If a graph contains a "negative cycle", i.e., a cycle whose edges sum to a negative value, then walks of arbitrarily low weight can be constructed, i.e., there can be no *shortest* path. Bellman-Ford can detect negative cycles and report their existence, but it cannot produce a correct answer if a negative cycle is reachable from the source.

Bellman–Ford runs in $O(|V| \cdot |E|)$ time, where $|V|$and $|E|$ are the number of vertices and edges respectively.

24.2 Algorithm[3]

Algorithm 8 Bellman-Ford

```
boolean bellmanFord (grafo, nodoOrigen):
  // Inicializamos el grafo
  for v in grafo.vertices:
    distancia[v] := INFINITO
    predecesor[v] := NULL
  distancia[nodoOrigen] := 0
  // Relajamos cada arista del grafo
  for i=1 to |V[grafo]-1|:
    for (u,v) in E[grafo]:
      if (distancia[v] > distancia[u] + peso(u,v)):
        distancia[v] := distancia[u] + peso (u,v)
        predecesor[v] := u
  // Comprobamos si hay ciclos negativos
  for (u,v) in grafo.aristas:
    if (distancia[v] > distancia[u] + peso(u,v)):
      print (‘‘Hay ciclo negativo’’)
      return False
  return True
```

[3]Bellman-Ford: http://www.youtube.com/watch?v=L6x53Vjy_HM

25 Componentes fuertemente conexas

25.1 Introducción

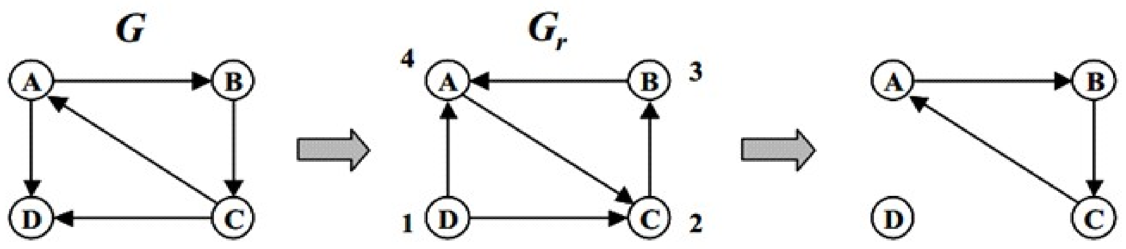
Una componente fuertemente conexa de un digrafo es un conjunto máximo de nodos en el cual existe un camino que va desde cualquier nodo del conjunto hasta cualquier otro nodo del conjunto. Si el digrafo conforma una sola componente fuertemente conexa, se dice que el digrafo es fuertemente conexo.

Es sencillo desarrollar una solución de fuerza bruta para este problema. Solo se debe comprobar para cada par de nodos u y v, si u es alcanzable desde v y viceversa. Sin embargo, se puede utilizar el DFS para determinar con eficiencia las componentes fuertemente conexas de una digrafo.

Sea G un digrafo, a continuación se presenta una descripción del algoritmo de Kosaraju:

1. Se ejecuta un DFS postorden en G. Se deben enumerar los nodos en el orden de terminación de las llamadas recursivas del DFS.
2. Se construye un digrafo nuevo Gr, invirtiendo las direcciones de todas las aristas de G.
3. Luego se ejecuta un DFS en Gr, partiendo del nodo con numeración más alta de acuerdo con la numeración asignada en el paso 1. Si el DFS no llega a todos los nodos, se inicia la siguiente búsqueda a partir del nodo restante con numeración más alta.
4. Cada árbol del conjunto de árboles DFS resultante es una componente fuertemente conexa de G.

El método de Kosaraju encuentra las componentes fuertemente conexas de cualquier digrafo e tiempo y espacio lineal, mostrando una tremenda superioridad con respecto al algoritmo de fuerza bruta descrito al comienzo de esta sección.



25.2 Algoritmo

Algorithm 9 Componentes fuertemente conexas

```
function componentesFuertementeConexas (digrafo):
  pila = Pila()
  // Recorre todos los nodos en post-orden
  for v in iter_dfs_digrafo_postorden (digrafo):
    pila.apilar(v)
  // En Dt, todas las aristas apuntan en sentido contrario
  Dt := digrafo.transpuesto()
  for v in Dt.vertices:
    v.visitado := False
  cfc := []
  while (not pila.vacia()):
    v := pila.desapilar()
    if (v.visitado):
      continue
    // Agrega todos los nodos alcanzabables por DFS
    // desde ‘‘v’’, y los visita
    cfc.append(lista_dfs_nodo(Dt,v))
  return cfc
```

26 Puntos de articulación

Un punto de articulación de un grafo G es un nodo **v** cuya remoción aumenta en 1 el número de componentes conexas del grafo.

26.1 Algoritmo

- r (la raíz) es un punto de articulación si tiene más de un hijo en el árbol.
- $\forall u \in V,\ u \neq r,\ u$ es punto de articulación si al eliminarlo del árbol desde alguno de sus descendientes no se puede 'ascender' hasta alguno de los antecesores de u en el árbol.

Definimos: *másalto[u]* = el *numdfs* del nodo w más alto del árbol que se puede alcanzar desde u bajando cero ó más tree-edges y subiendo entonces como máximo un back-edge.

Por lo tanto: $u \neq r$ es punto de articulación $\Leftrightarrow$tiene un hijo x tal que *másalto[x]*$\geq$*numdfs[u]*.

$$\begin{array}{l} \qquad\qquad\qquad\qquad\qquad numdfs[u] \\ min\ numdfs[w] \forall w \in V : (u,w) \in E\ y\ (u,w) \notin T_{DFS} (\text{salimos de u subiendo por una arista back}) \\ \qquad\qquad\qquad\qquad masalto[x] \forall x \in hijos(u, T_{DFS}) \end{array}$$

másalto[u] =

Implementación: Un recorrido en profundidad calculando *numdfs* para cada vértice cuando se baja y calculando *másalto* para cada vértice cuando se sube.

Coste: con lista de adyacencia $T(n) = O(|V| + |E|)$.

Algorithm 10 Puntos de articulación

```
function puntos_de_articulacion (grafo):
  i := 1
  for v in iter_dfs_grafo_preorden (grafo):
    // Establecemos numdfs para cada vertice
    v.numdfs := i
    i += 1
  for v in iter_dfs_grafo_postorden (grafo):
    v.masalto := v.numdfs
    // Establecemos masalto para cada vertice
    for h in v.hijos_dfs:
      v.masalto := min (v.masalto, h.masalto)
    for n in v.vecinos_dfs:
      v.masalto := min (v.masalto, h.numdfs)
  puntos_articulacion := []
  for v in iter_dfs_grafo_preorden (grafo):
    // Si la raiz tiene mas de 1 hijo, es pto; sino no
    if (v == iter_preorden.raiz) and (len(v.hijos_dfs)>1):
      puntos_articulacion.append(v)
    // Para cada vertice ‘‘v’’: si existe un hijo ‘‘h’’ tal que
    // masalto(h) >= numdfs(v), v es pto de art
    else:
      for h in v.hijos_dfs:
        if (h.masalto >= v.numdfs):
          puntos_articulacion.append(v)
          break
  return puntos_articulacion
```

27 Prim's algorithm

27.1 Introduction

Prim's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Prim's algorithm is an example of a greedy algorithm.

27.2 Time complexity

Minimum edge weight data structure	**Time complexity (total)**
adjacency matrix, searching	$O(V^2)$
binary heap (as in pseudocode below) and adjacency list	$O\left((V+E)\cdot\log(V)\right) = O\left(E\cdot\log(V)\right)$
Fibonacci heap and adjacency list	$O(E+V\cdot\log(V))$

27.3 Pseudocode[4]

Algorithm 11 Algoritmo de Prim

```
function algoritmo_prim (grafo,origen):
  solucion := []
  grafo.inicializar_vertices()
  grafo.vertices[origen].visitado = True
  aristas =: []
  for (destino,peso) in grafo.lista_de_adyacencias[origen]:
    aristas.append((origen,peso,destino))
  heap := []
  for arista in aristas:
    heapq.heappush(heap,arista)
  while (heap):
    arista = heapq.heappop(heap)
    if (grafo.vertices[arista[0]].visitado != grafo.vertices[arista[2]].visitado):
    solucion.append(arista)
    for (destino,peso) in grafo.lista_de_adyacencias[arista[2]]:
      heapq.heappush(heap,(arista[2],peso,destino))
    grafo.vertices[arista[2]].visitado := True
  return solucion
```

[4]Prim: http://www.youtube.com/watch?v=BtGuZ-rrUeY

28 Kruskal's algorithm

28.1 Introduction

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, all with simple data structures.

28.2 Algorithm[5]

Algorithm 12 Algoritmo de Kruskal

```
function kruskal (graph, length):
  C(v) = {v}
  heap = Heap()
  tree = {}
  for e in graph.edges:
    heap.push(e)
  while (|tree| < |V|-1 edges):
    (u,v) := heap.desencolar()
    // Let C(v) be the cluster containing v, and let C(u) be the cluster containing u
    if (C(v)  C(u)):
      add edge (v,u) to tree
      merge C(v) and C(u) into 1 cluster
  return tree
```

[5]Kruskal: http://www.youtube.com/watch?v=OWfeZ9uDhdw