Apuntes

Tema ***Algoritmos avanzados***

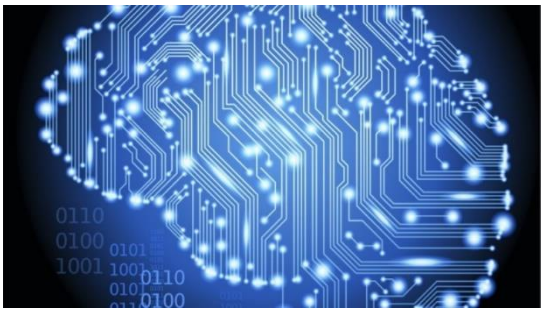
"Un diseñador sabe que ha alcanzado la perfección no cuando ya no tiene nada mas que añadir, sino cuando ya no le queda nada mas que quitar"
Antoine de Saint Exupéry

Algoritmos avanzados

Introducción

Muchas veces los programadores se ven tentados a codificar el primer algoritmo que se les viene a la mente, sin ponerse a pensar en el costo del mismo. Hemos visto diferentes algoritmos que se aplican a diferentes problemas, y que tienen costos diferentes de ejecución, ya sea en memoria utilizada, o en tiempo necesario para completar.

Un buen algoritmo puede ser la única forma que tengamos de que un algoritmo complete la solución; ya que la solución obvia, la primera que se nos viene a la mente, es tan costosa que necesita más memoria de la que disponemos, o llevaría varios años de ejecución, aún utilizando los servidores más potentes del planeta. En ésta lectura, estudiaremos algunos problemas reales, y algoritmos inteligentes que los resuelven.

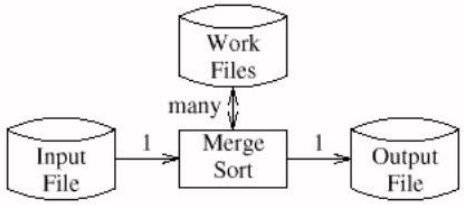


Ordenación de archivos a través de mapas de bits (bitmap sort)

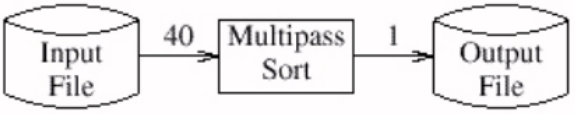
El bitmap sort es el método mas rápido y mas eficiente para ordenar un conjunto determinado de enteros. Fue publicado por por primera vez por Jon Bentley, en el libro de "Programming Pearls.".

Funciona al pensar en una porción de memoria como un conjunto de bits numerados. Cada número que se va a ordenar se debe considerar como un bit en esa porción de memoria. Después de que se hayan establecido todos los bits, es una cuestión simple el obtener los números ordenados comenzando desde el principio (o el final si se requiere el orden inverso) y verificar si se ha establecido un bit. Si es así, se mueve al buffer de salida.

Supongamos que tenemos un archivo que contiene a lo sumo 27000 **enteros** en el rango 1..27000, sin duplicados, y que no hay otros datos asociados con dichos enteros, y que deseamos ordenarlos, considerando que tenemos muy poco espacio en memoria como para cargar el archivo y hacer la ordenación en memoria. Para ser más precisos, asumimos que podemos cargar unos 1000 números en una memoria de 16 bits.



Una solución obvia es



la de usar los algoritmos de ordenamiento externo vistos anteriormente.
Pero una mejor solución sería la tener un bucle principal que hace 27 pasadas sobre el archivo principal. En la primera pasada lee en memoria cualquier entero entre 1 y 1000, los ordena, y los escribe en el archivo de salida. La segunda pasada ordena los enteros que van entre 1001 y 2000, y así sucesivamente.
La solución ideal haría una sola lectura del archivo de entrada, ordenaría todos los elementos en memoria, y los escribiría en el archivo de salida. ¿Pero cómo podemos hacer esto si el archivo contiene 27000 enteros y solo podemos almacenar 2000 enteros de 16-bits en memoria?

Si usamos un bitmap (o vector de bits)(*bitmap bit vector representation*), podríamos representar el archivo por un string de 27000 bits en el cual el n-ésimo bit es 1 sólo si n está en el archivo. Como podemos alojar 2000 enteros de 16 bits en memoria, esto nos deja disponibles unos 32000 bits, más que suficiente para representar los 27000 enteros del archivo.

```
/* phase 1: initialize set to empty */
        for i = [0, n)
        bit[i] = 0
/* phase 2: insert present elements into the set */
        for each i in the input file
        bit[i] = 1
/* phase 3: write sorted output */
        for i = [0, n)
        if bit[i] == 1
        write i on the output file
```

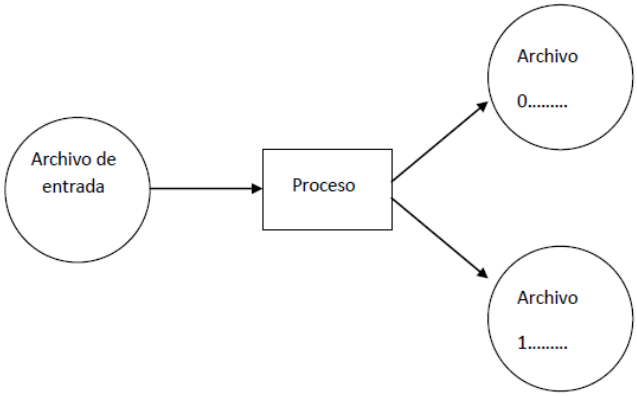
El algoritmo debiera inicializar el conjunto en vacío, poniendo todos los bits en 0. Luego debiera construir el conjunto a través de la lectura de cada entero del archivo, poniendo en 1 el bit apropiado, y luego debiera escribir cada entero en el archivo de salida, si su correspondiente bit está en 1. En la Figura, observamos el seudocódigo de este algoritmo.

Busqueda en archivos de gran tamaño

Supongamos que tenemos una cinta que contiene a lo sumo un millón de enteros en orden aleatorio, y que deseamos encontrar un entero de 20 bits que no está en la cinta. Sabemos que existen 2^20=1.048.576 enteros de 20 bits, por lo que seguramente existirán enteros que no estén en la cinta.
¿Cómo resolveríamos este problema si tuviéramos cantidades ilimitadas de memoria?
¿y si la memoria fuese limitada?

Si asumimos una cantidad ilimitada de memoria, podríamos implementar una solución similar a la del problema anterior, y usar un bitmap de 1.048.576 posiciones para representar los enteros que hayamos leído. Luego de finalizar la lectura del archivo de entrada recorremos el bitmap, y cualquier posición en 0 indicará un elemento que no estaba en el archivo original.

Si la cantidad de memoria fuera limitada, entonces probablemente no podremos usar un bitmap. Ahora atacamos el problema asumiendo que no disponemos de esta memoria para usar un bitmap.
Es útil ver este problema en término de los 20 bits que representan cada entero. En la primer pasada leemos el archivo de entrada, y escribimos aquellos que empiezan con un bit 0 a un archivo, y aquellos que comienzan con un bit 1 a otro, como se ve en la Figura .



El archivo que contiene todos los enteros que comienzan con 0, contiene los números 0-524288 (0 - 219), mientras que el otro archivo contiene los números que van desde 524289 al 1048576 (219+1 - 220).
El archivo de menor tamaño contiene menor cantidad de enteros, y por lo tanto seguramente contiene números que faltan en la secuencia.
El algoritmo toma este archivo, y vuelve a repetir el procedimiento descrito en la Figura.

Si el archivo original contiene N elementos, la primera pasada lee N enteros, la segunda lee a lo sumo N/2 enteros, la tercera lee a lo sumo N/4, y así sucesivamente, **por lo que el tiempo total insumido es proporcional a N**.

Rotación de vectores

Supongamos que deseamos rotar un vector de N elementos en I posiciones. Por ejemplo, si N=8 e I=3, entonces el vector ABCDEFGH se rotaría a DEFGHABC.
Una solución trivial a este problema usaría un vector intermedio, en el que copiaríamos los primeros I elementos del vector (ABC), luego moveríamos los restantes elementos en I posiciones, y finalmente copiaríamos el vector temporario en las ultimas I posiciones del vector, como se ve en la Figura 3. Sin embargo, esta solución es costosa en términos de espacio.

```
char temp[I];
for (i=0; i < I; i++)
    temp[i] = V[i];
for (i=0; i < N-I; i++)
   V[i]=V[i+3];
for (i=0, i < I; i++)
   V[N-I+i] = temp[i];
```

Una solución diferente podría ser tener una subrutina que rota el vector en una sola posición (en tiempo lineal), y llamarla I veces, pero esto es muy costoso también, esta vez en términos de performance.

Un algoritmo inteligente invita a ver el problema de manera diferente. Rotar el vector de entrada es simplemente el intercambio de dos segmentos del vector AB para convertirlo en el vector BA, donde A representa los primeros I elementos del vector original. Supongamos que A es más pequeño que B. Entonces podemos dividir B en BI y BD de modo que BD tenga la misma longitud que A. Luego intercambiamos A y BD para transformar ABIBD en BDBIA. En este punto, la secuencia A está en su posición final, así que podemos enfocarnos en intercambiar

las dos partes de B. Pero este problema tiene la misma forma que el original, así que podemos resolverlo de manera recursiva.

Otra solución posible emplea la rutina Reverse. Comenzando con AB, usamos Reverse sobre A para obtener ARB, luego usamos Reverse sobre B para obtener ARBR, y finalmente usamos Reverse sobre todo el vector para obtener (ARBR)R, que es exactamente BA. En la Figura vemos la ejecución de este algoritmo.

```
Reverse(1,I);          // CBADEFGH
Reverse(I+1,N);        // CBAHGFED
Reverse(1,N);          // DEFGHABC
```

Búsqueda de anagramas

Dado un diccionario de palabras, queremos encontrar todos los conjuntos de anagramas. Por ejemplo, “caso”, “saco” y “cosa” son anagramas entre sí, por que cada uno puede ser formado a partir de permutar las letras de los otros.
Este problema puede ser muy complicado de resolver si intentamos usar un algoritmo trivial. Uno podría pensar en tomar cada palabra del diccionario, y recorrer el resto del diccionario intentando encontrar aquellas que sean anagramas de esa, y así sucesivamente. Pero este procedimiento sería demasiado costoso.

La mejor solución consiste en computar las firmas (*signatures*) de cada palabra del diccionario, donde la firma de una palabra consiste simplemente en ordenar las letras de cada palabra. Por ejemplo, la firma de “caso”, “saco” y “cosa” es “acos”. Otros ejemplos de firmas son:

Casa -> aacs
perro -> eoprr
firma -> afimr

Los anagramas serán entonces todas aquellas palabras que tengan la misma firma.
El algoritmo para encontrar todos los anagramas posibles realiza una primera pasada sobre el archivo de entrada, computando la firma de cada palabra. Luego ordena el archivo por las firmas, y finalmente recorre el archivo ordenado, imprimiendo todos los conjuntos de anagramas correspondientes a aquellas palabras que tengan la misma firma, que estarán ordenadas.

Algoritmos Evolutivos

Los algoritmos evolutivos son **métodos de optimización** y búsqueda de soluciones basados en los postulados de la evolución biológica. En ellos se mantiene un conjunto de entidades que representan posibles soluciones, las cuales se mezclan, y compiten entre sí, de tal manera que las más aptas son capaces de prevalecer a lo largo del tiempo, evolucionando hacia mejores soluciones cada vez.
Los algoritmos evolutivos, y la computación evolutiva, **son una rama de la inteligencia artificial**. Son utilizados principalmente en problemas con **espacios de búsqueda extensos y no lineales**, en donde otros métodos no son capaces de encontrar soluciones en un tiempo razonable.

Siguiendo la terminología de la teoría de la evolución, las entidades que representan las soluciones al problema se denominan individuos o cromosomas, y el conjunto de éstos, población. Los individuos son modificados por operadores genéticos, principalmente el cruce, que consiste en la mezcla de la información de dos o más individuos; la mutación, que es un cambio aleatorio en los individuos; y la selección, consistente en la elección de los individuos que sobrevivirán y conformarán la siguiente generación. Dado que los individuos que representan las soluciones más adecuadas al problema tienen más posibilidades de sobrevivir, la población va mejorando gradualmente.
Suele hablarse de tres paradigmas principales de algoritmos evolutivos:

- Programación evolutiva
- Estrategias evolutivas
- Algoritmos genéticos

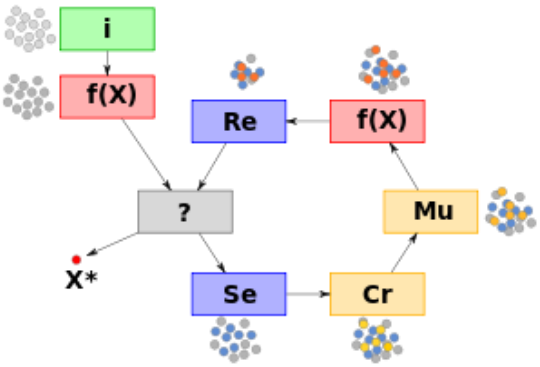
Antena de la nave espacial ST5 de la NASA. Esta intrincada forma fue encontrada usando un algoritmo evolutivo, con el propósito de conseguir el mejor patrón de radiación para la nave.

Algoritmos Genéticos

Un algoritmo es una serie de pasos organizados que describe el proceso que se debe seguir, para dar solución a un problema específico. En los años 1970, de la mano de **John Henry Holland**, surgió una de las líneas más prometedoras de la inteligencia artificial, la de los algoritmos genéticos. Son llamados así porque se inspiran en la evolución biológica y su base genético-molecular. Estos algoritmos hacen evolucionar una población de individuos sometiéndola a acciones aleatorias semejantes a las que actúan en la evolución biológica (mutaciones y recombinaciones genéticas), así como también a una selección de acuerdo con algún criterio, en función del cual se decide cuáles son los individuos más adaptados, que sobreviven, y cuáles los menos aptos, que son descartados. ***Los algoritmos genéticos se***

enmarcan dentro de los algoritmos evolutivos, que incluyen también las estrategias evolutivas, la programación evolutiva y la programación genética.

Los algoritmos genéticos funcionan entre **el conjunto de soluciones de un problema llamado fenotipo**, y **el conjunto de individuos de una población natural**, codificando la información de cada solución en una cadena, generalmente binaria, **llamada cromosoma**. Los símbolos que forman la cadena son llamados los genes. Cuando la representación de los cromosomas se hace con **cadenas de dígitos binarios se le conoce como genotipo**. Los cromosomas evolucionan a través de iteraciones, llamadas **generaciones**. En cada generación, los cromosomas son evaluados usando alguna medida de aptitud. Las siguientes generaciones (nuevos cromosomas), son generadas aplicando los operadores genéticos repetidamente, siendo estos **los operadores de selección, cruzamiento, mutación y reemplazo**.

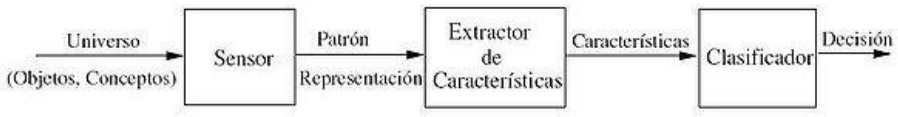


Los algoritmos genéticos son de probada eficacia en caso de querer **calcular funciones no derivables** *(o de derivación muy compleja)* aunque su uso es posible con cualquier función.
Deben tenerse en cuenta también las siguientes consideraciones:

- Si la función a optimizar tiene muchos máximos/mínimos locales se requerirán más iteraciones del algoritmo para "asegurar" el máximo/mínimo global.
- Si la función a optimizar contiene varios puntos muy cercanos en valor al óptimo, solamente podemos "asegurar" que encontraremos uno de ellos (no necesariamente el óptimo).

Reconocimiento de patrones

El reconocimiento de patrones se ocupa de los procesos sobre ingeniería, computación y matemáticas relacionados con objetos físicos o abstractos, con el propósito de extraer información que permita establecer propiedades de entre conjuntos de dichos objetos.
El reconocimiento de patrones (*también llamado lectura de patrones, identificación de figuras y reconocimiento de formas*) consiste en el reconocimiento de patrones de señales. Los patrones se obtienen a partir de los procesos de **segmentación**, **extracción de características** y **descripción** donde cada objeto queda representado por una colección de descriptores. El sistema de reconocimiento debe asignar a cada objeto su categoría o clase (*conjunto de entidades que comparten alguna característica que las diferencia del resto*). Para poder reconocer los patrones se siguen los siguientes procesos:



- Adquisición de datos
- Extracción de características
- Toma de decisiones

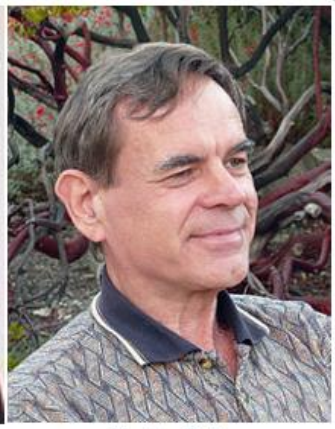
El punto esencial del reconocimiento de patrones es la clasificación: se quiere clasificar una señal dependiendo de sus características. Señales, características y clases pueden ser de cualquiera forma, por ejemplo se puede clasificar imágenes digitales de letras en las clases «A» a «Z» dependiendo de sus píxeles o se puede clasificar ruidos de cantos de los pájaros en clases de órdenes aviares dependiendo de las frecuencias.

Los sistemas de reconocimiento de patrones tienen diversas aplicaciones. Algunas de las más relevantes y utilizadas actualmente son:
Meteorológica: poder clasificar todos los datos meteorológicos según diversos patrones, y con el conocimiento a priori que tenemos de las diferentes situaciones que pueden aparecer nos permite crear mapas de predicción automática.
Reconocimiento de caracteres escritos a mano o a máquina: es una de las utilidades más populares de los sistemas de reconocimiento de patrones ya que los símbolos de escritura son fácilmente identificables.
Reconocimiento de voz: el análisis de la señal de voz se utiliza actualmente en muchas aplicaciones, un ejemplo claro son los teleoperadores informáticos.
Aplicaciones en medicina: análisis de biorritmos, detección de irregularidades en imágenes de rayos-x, detección de células infectadas, marcas en la piel...
Reconocimiento de huellas dactilares: utilizado y conocido por la gran mayoría, mediante las huellas dactilares todos somos identificables y con programas que detectan y clasifican las coincidencias, resulta sencillo encontrar correspondencias.
Reconocimiento de rostros: utilizado para contar asistentes en una manifestación o simplemente para detectar una sonrisa, ya hay diferentes cámaras en el mercado con esta opción disponible.
Interpretación de fotografías aéreas y de satélite: gran utilidad para propuestas militares o civiles, como la agricultura, geología, geografía, planificación urbana...
Predicción de magnitudes máximas de terremotos.
Reconocimiento de objetos: con importantes aplicaciones para personas con discapacidad visual.
Reconocimiento de música: identificar el tipo de música o la canción concreta que suena.

Algoritmo Knuth-Morris-Pratt

El algoritmo KMP es un algoritmo de búsqueda de subcadenas simple y por lo tanto su objetivo es buscar la existencia de una subcadena dentro de una cadena. Para ello utiliza información basada en los fallos previos, aprovechando la información que la propia palabra a buscar contiene de sí *(sobre ella se precalcula una tabla de valores)*, para determinar donde podría darse la siguiente existencia, sin necesidad de analizar más de 1 vez los caracteres de la cadena donde se busca. El algoritmo originalmente fue elaborado por Donald Knuth y Vaughan Pratt


Donald Knuth


Vaughan Pratt

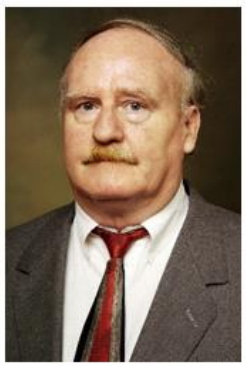
y de modo independiente por James H. Morris en 1977, pero lo publicaron juntos los tres.


James H. Morris

El algoritmo KMP, trata de localizar la posición de comienzo de una cadena, dentro de otra. Antes que nada con la cadena a localizar se precalcula una tabla de saltos *(conocida como tabla de fallos)* que después al examinar entre si las cadenas se utiliza para hacer saltos cuando se localiza un fallo.
Supongamos una tabla **'F'** ya precalculada, y supongamos que la cadena a buscar esté contenida en el array **'P()'**, y la cadena donde buscamos esté contenida en un array **'T()'**. Entonces ambas cadenas comienzan a compararse usando un puntero de avance para la cadena a buscar, si ocurre un fallo en vez de volver a la posición siguiente a la primera coincidencia, se salta hacia donde sobre la tabla, indica el puntero actual de avance de la tabla. El array **'T'** utiliza un puntero de avance absoluto que considera donde se compara el primer carácter de ambas cadenas, y utiliza como un puntero relativo *(sumado al absoluto)* el que utiliza para su recorrido el array 'P'. Se dan 2 situaciones:
Mientras existan coincidencias el puntero de avance de 'P', se va incrementando y si alcanza el final se devuelve la posición actual del puntero del array 'T'
Si se da un fallo, el puntero de avance de 'T' se actualiza hasta, con la suma actual del puntero de 'P' + el valor de la tabla 'F' apuntado por el mismo que 'P'. A continuación se actualiza el puntero de 'P', bajo una de 2 cicunstancias; Si el valor de 'F' es mayor que -1 el puntero de 'P', toma el valor que indica la tabla de salto 'F', en caso contrario vuelve a recomenzar su valor en 0.

Algoritmo de búsqueda de cadenas Boyer-Moore

El algoritmo de búsqueda de cadenas Boyer-Moore es un particularmente eficiente algoritmo de búsqueda de cadenas, y ha sido el punto de referencia estándar para la literatura de búsqueda de cadenas práctica. Fue desarrollado por Bob Boyer y J Strother Moore en 1977. El algoritmo preprocesa la cadena objetivo (clave) que está siendo buscada, pero no en la cadena en que se busca *(no como algunos algoritmos que procesan la cadena en que se busca y pueden entonces amortizar el coste del preprocesamiento mediante búsqueda repetida)*.


Robert Stephen Boyer


J Strother Moore

El tiempo de ejecución del algoritmo Boyer-Moore, aunque es lineal en el tamaño de la cadena siendo buscada, puede tener un factor significativamente más bajo que muchos otros algoritmos de búsqueda: no necesita comprobar cada carácter de la cadena que es buscada, puesto que salta algunos de ellos. Generalmente **el algoritmo es más rápido cuanto más grande es la clave que es buscada**, usa la información conseguida desde un intento para descartar tantas posiciones del texto como sean posibles en donde la cadena no coincida.

A la gente frecuentemente le sorprende el algoritmo de Boyer-Moore, cuando lo conoce, porque en su verificación intenta comprobar si hay una coincidencia en una posición particular marchando hacia atrás. Comienza una búsqueda al principio de un texto para la palabra "ANPANMAN", por ejemplo, comprueba que la posición octava del texto en proceso contenga una "N". Si encuentra la "N", se mueve a la séptima posición para ver si contiene la última "A" de la palabra, y así sucesivamente hasta que comprueba la primera posición del texto para una "A".

La razón por la que Boyer-Moore elige este enfoque está más clara cuando consideramos que pasa si la verificación falla. Por ejemplo, si en lugar de una "N" en la octava posición, encontramos una "X". La "X" no aparece en "ANPANMAN", y esto significa que no hay coincidencia para la cadena buscada en el inicio del texto o en las siguientes siete posiciones, puesto que todas fallarían también con la "X". Después de comprobar los ocho caracteres de la palabra "ANPANMAN" para tan sólo un carácter "X", seremos capaces de saltar hacia delante y comenzar buscando una coincidencia en el final en la 16ª posición del texto.

```
- - - - - - - X - - - - - - -
A N P A N M A N - - - - - - -
- A N P A N M A N - - - - - -
- - A N P A N M A N - - - - -
- - - A N P A N M A N - - - -
- - - - A N P A N M A N - - -
- - - - - A N P A N M A N - -
- - - - - - A N P A N M A N -
- - - - - - - A N P A N M A N
```

La X en la posición 8 excluye todas la 8 posibles posiciones de comienzo mostradas.

Esto explica por qué el rendimiento del caso promedio del algoritmo, para un texto de longitud **n** y patrón fijo de longitud **m**, es **n/m** en el mejor caso, solo uno en **m** caracteres necesita ser comprobado. Esto también explica el resultado algo contra-intuitivo de que cuanto más largo es el patrón que estamos buscando, el algoritmo suele ser más rápido para encontrarlo.
El algoritmo precalcula dos tablas para procesar la información que obtiene en cada verificación fallada: una tabla calcula cuantas posiciones hay por delante en la siguiente búsqueda basada en el valor del carácter que no coincide; la otra hace un cálculo similar basado en cuantos caracteres coincidieron satisfactoriamente antes del intento de coincidencia fallado. *(Puesto que estas dos tablas devuelven resultados indicando cuán lejos "saltar" hacia delante, son llamada en ocasiones "tablas de salto", que no deberían ser confundidas con el significado más común de tabla de saltos en ciencia de la computación.)* El algoritmo se desplazará con el valor más grande de los dos valores de salto cuando no ocurra una coincidencia.

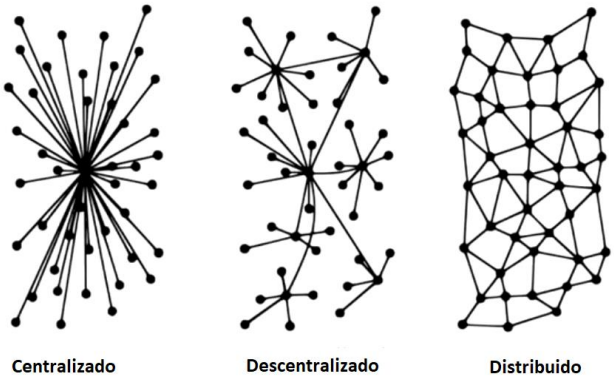
Blockchain

Una **cadena de bloques**, también conocida por las siglas BC (del inglés **Blockchain**) es una base de datos distribuida, formada por cadenas de bloques diseñadas para evitar su modificación una vez que un dato ha sido publicado usando un sellado de tiempo confiable y enlazando a un bloque anterior. Por esta razón es especialmente adecuada para almacenar de forma creciente datos ordenados en el tiempo y sin posibilidad de modificación ni revisión. Este enfoque tiene diferentes aspectos:

Almacenamiento de datos.- Se logra mediante la replicación de la información de la cadena de bloques

Transmisión de datos.- Se logra mediante peer-to-peer

Confirmación de datos.- Se logra mediante un proceso de consenso entre los nodos participantes. El tipo de algoritmo más utilizado es el de prueba de trabajo en el que hay un proceso abierto competitivo y transparente de validación de las nuevas entradas llamada minería.



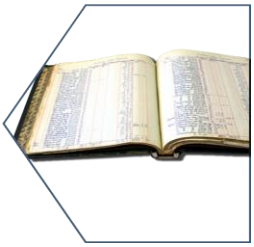
El concepto de cadena de bloque fue aplicado por primera vez en 2009 como parte de Bitcoin.

Los datos almacenados en la cadena de bloques normalmente suelen ser transacciones (Ej. financieras) por eso es frecuente llamar a los datos transacciones. Sin embargo no es necesario que lo sean. Realmente podríamos considerar que lo que se registran son cambios atómicos del estado del sistema. Por ejemplo una cadena de bloques puede ser usada para estampillar documentos y securizarlos frente a alteraciones.

Base de datos normal

Base de datos Bockchain

En una base de datos normal, cualquier información puede ser modificada, actualizada o eliminada. Blockchain es diferente: solo se puede agregar nueva información. Por su diseño, la información existente nunca se puede actualizar o eliminar.
Esto puede parecer extraño, pero esta es en realidad una de las características de seguridad clave de blockchain.
Entonces, ¿cómo puede funcionar una base de datos que no se puede cambiar o eliminar? Blockchain a menudo se conoce como un sistema basado en contabilidad. Algunos lo llaman tecnología de contabilidad distribuida. Los cambios se registran como nueva información.



En el 2009 hizo su debut el conocido Bitcoin, la criptomoneda más conocida del mundo, desarrollada por Satoshi Nakamoto (seudónimo de una persona que se mantiene en el anonimato) y, desde ese momento, el uso o la identificación que se le dio a Blockchain fue principalmente "monetaria" ligada a la transacción P2P (En la actualidad existen muchas criptomonedas).

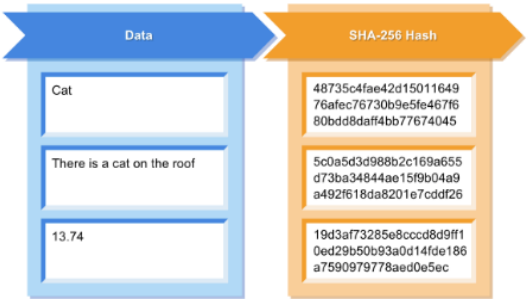
Sin embargo, esta tecnología es bastante más amplia, y se aplica muchas áreas.
Por ejemplo, Estonia es un pionero en el uso de Blockchain para gestionar sus registros en áreas como la sanitaria, judicial, legislativa o seguridad, con visos de ampliarse a otras esferas, como la medicina personal o la ciberseguridad.
Otro ejemplo en este ámbito es el del ayuntamiento (municipalidad) de Barcelona, que coordinará el proyecto europeo Decode, cuyo principal objetivo es lograr que los ciudadanos gestionen su privacidad en Internet y, al mismo tiempo, conozcan sus derechos digitales.
La industria alimentaria no está ajena a Blockchain ya que la cadena de supermercados Carrefour anunció que sus tiendas en Francia van a usar la cadena de bloques para clarificar la

procedencia de productos como la miel, huevos, queso, leche, naranjas, tomates, salmón y hamburguesas. Grandes empresas como Nestlé o Unilever han anunciado planes similares.
Existen también plataformas online que usan tecnología de cadena de bloques, como Bistspark (una plataforma para enviar y recibir dinero), Ripple (Permite a los bancos enviar pagos internacionales en tiempo real a travez de redes públicas), Slock.it (plataforma para alquilar o poner en alquiler cualquier cosa), Storj (plataforma de storage basade en cloud), etc.

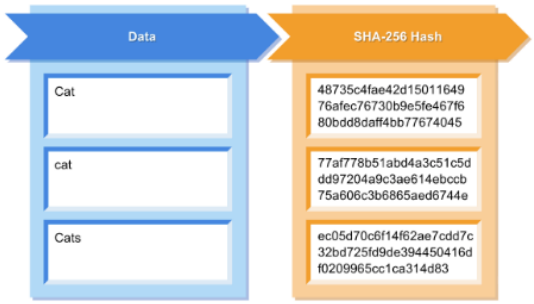
Como ves, los usos de Blockchain son muy variados por esa razón te invitamos a participar de la charla que nuestro experto, Marko Knezovic hará sobre esta interesante tecnología y sus utilidades futuras o su aplicación en casos concretos en el Blockchain Summit Latam 2018, programado para mayo próximo.

Una herramienta esencial utilizada por blockchain es el hash. Un hash es una cadena de números y letras que pueden identificar de manera única algunos textos o datos. Entonces, en muchos sentidos es como una huella dactilar.

Los valores hash a veces se llaman valores hash, códigos hash, resúmenes o simplemente hash. Hay varios algoritmos de hash o formas de crear hash. Aquí se muestran los valores hash de diferentes textos usando un algoritmo hash llamado SHA-256.

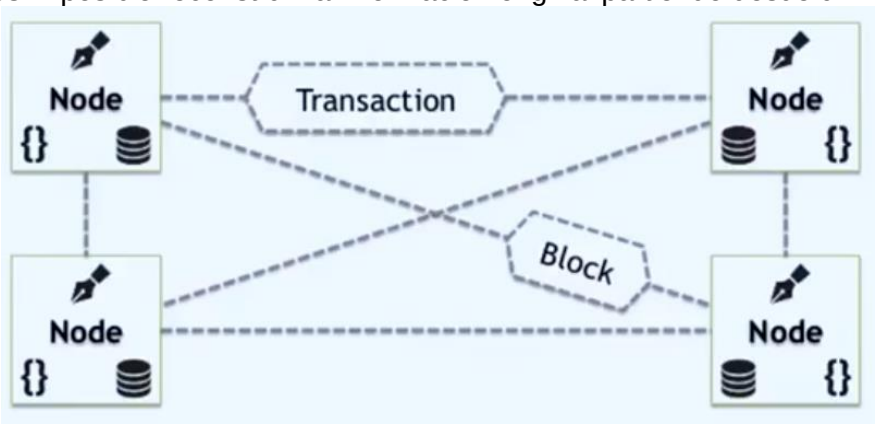


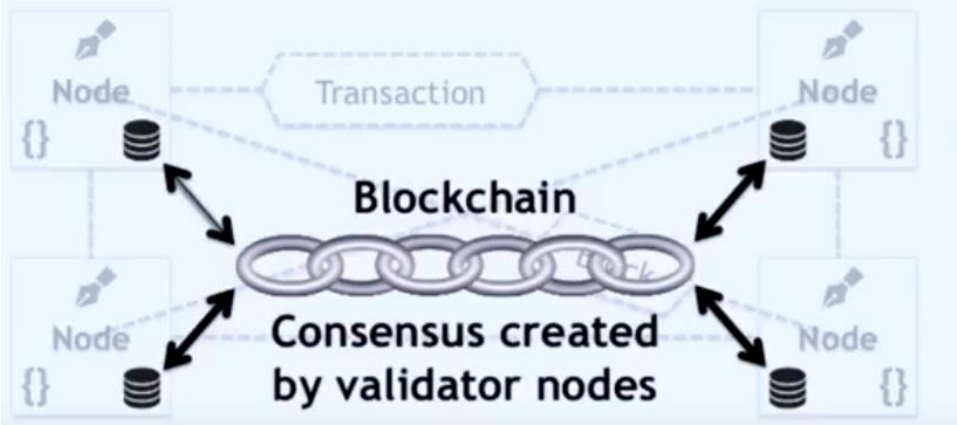
El Hash siempre tiene el mismo largo independientemente del dato que lo origine, incluso si el dato fuera una biblioteca pública complete, el hash de esos datos, seguiría teniendo el mismo largo.



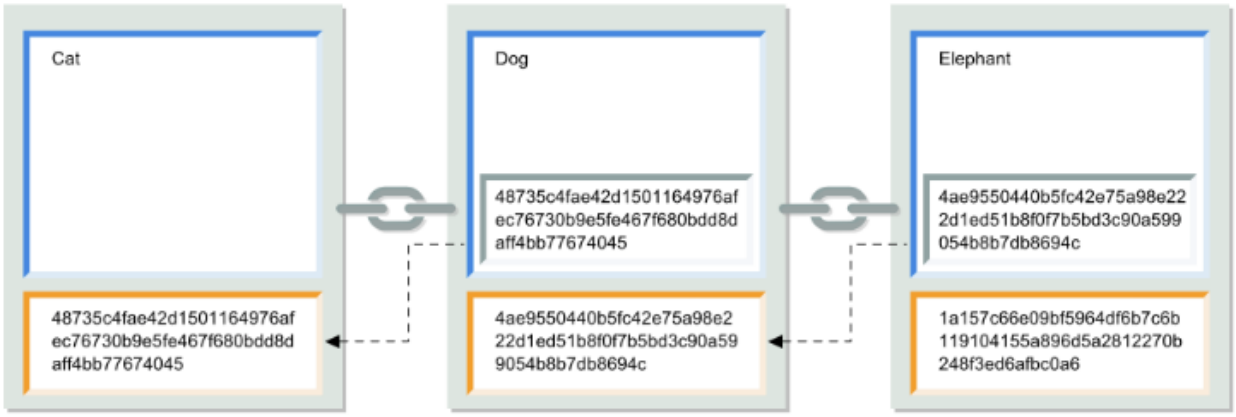
Y por el otro lado cualquier mínimo cambio en el dato original, genera un nuevo hash completamente distinto.

La probabilidad de colisión en la generación de un Hash con el algoritmo SHA-256 es de 4.3 $*10^{-60}$, es decir… casi imposible.
Por otro lado es imposible reconstruir la información original partiendo desde un valor Hash





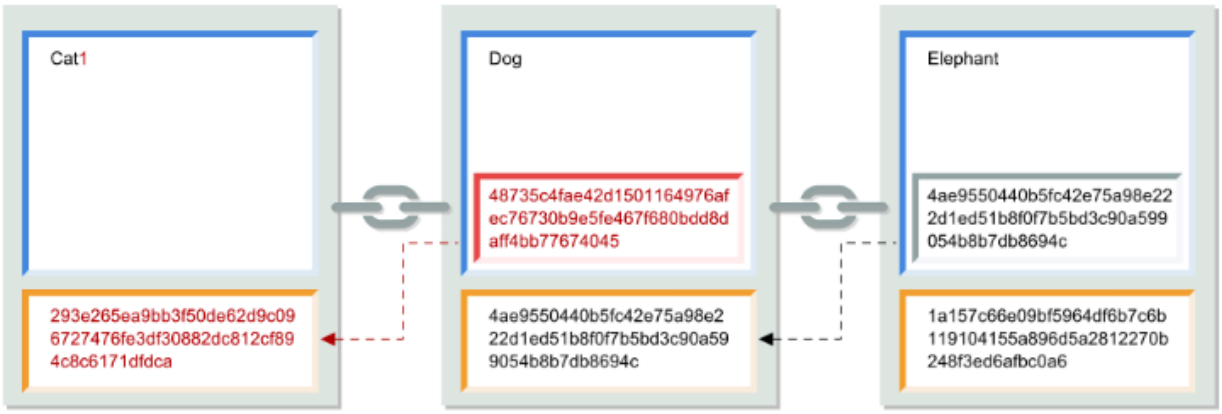
En blockchain, la información se almacena en bloques. Cada bloque incluye un hash de los datos o contenidos del bloque.
Entonces, un receptor puede confirmar que los contenidos del bloque son válidos recreando el hash del contenido del bloque y comparándolo con el hash incluido en el bloque.
Este es el primer paso en la seguridad de blockchain.



En la figura vemos una cadena de bloques, la cantidad de bloques puede ser muy grande. Cada bloque incluye el hash de la cadena anterior como parte de su información y es esta característica la que lo hace muy seguro.

Imaginemos que queremos adulterar información, por ejemplo introduciendo un cambio pequeño en el primer bloque, reemplazando “Cat” por “Cat1”

Inmediatamente veremos que el Hash calculado no representa esa nueva información, asi que lo primero que se nos ocurriría es recalcular ese hash.

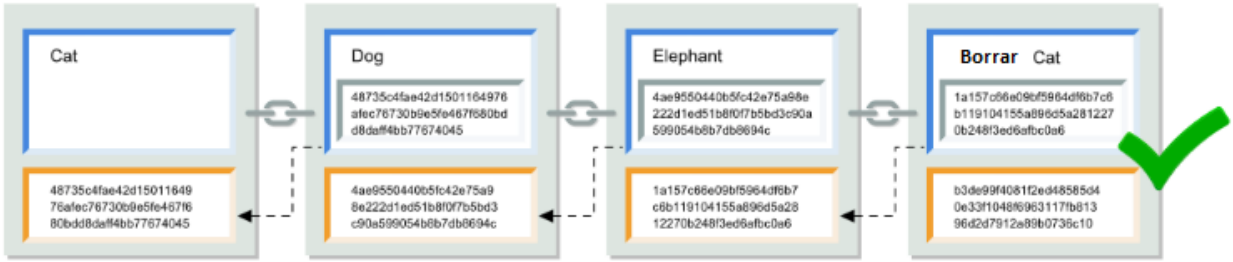


Pero una vez recalculado, existe otro bloque en algún lugar (es una base de datos distribuida) que contiene como “dato” la cadena hash del bloque que acabamos de modificar, por lo cual se detectaría el fraude. Pero aún asi, si tuviésemos suerte y encontrásemos ese bloque, al modificar la referencia de hash al bloque anterior, deberíamos modificar su propio hash… y asi quedaría la inconsistencia en ese otro nuevo bloque.

Los bloques en una cadena de bloques nunca se eliminan o modifican. Los bloques existentes se quedan para siempre en el blockchain.

.

Si los datos deben actualizarse, se incluye una actualización de estos datos en un bloque que se agrega al final de la cadena de bloques



Debido a que ningún bloque se actualiza o se elimina de una cadena de bloques, las cadenas de bloques a menudo se denominan libros mayores.
Este sistema de libro mayor se puede usar como un sistema de base de datos. Por ejemplo, una solución bancaria podría usar una cadena de bloques para registrar depósitos y retiros de una cuenta bancaria.

Cuando se hace referencia a un bloque, las soluciones blockchain generalmente usan el hash del bloque. Una alternativa es usar la altura del bloque, a veces llamado el número de bloque.

Clasificación:

Según el acceso a los datos. Las cadenas de bloques se pueden clasificar basándose en el acceso a los datos almacenados en la misma:
*** Cadena de bloques pública**: Es aquella en la que no hay restricciones ni para leer los datos de la cadena de bloques (los cuales pueden haber sido cifrados) ni para enviar transacciones para que sean incluidas en la cadena de bloques. En ellas es fácil entrar y salir, son transparentes, están construidas con precaución para la operación en un entorno no confiable. Son ideales para uso en aplicaciones totalmente descentralizadas como por ejemplo para el Internet
*** Cadena de bloques privada**: Es aquella en la que tanto los accesos a los datos de la cadena de bloque como el envío de transacciones para ser incluidas, están limitadas a una lista predefinida de entidades
Ambos tipos de cadenas deben ser considerados como casos extremos pudiendo haber casos intermedios.

Según los permisos. Las cadenas de bloques se pueden clasificar basándose en los permisos para generar bloques en la misma:
Cadena de bloques sin permisos: Es aquella en la que no hay restricciones para que las entidades puedan procesar transacciones y crear bloques. Este tipo de cadenas de bloques necesitan tokens nativos para proveer incentivos que los usuarios mantengan el sistema. Ejemplos de tokens nativos son los nuevos bitcoins que se obtienen al construir un bloque y las comisiones de las transacciones. La cantidad recompensada por crear nuevos bloques es una buena medida de la seguridad de una cadena de bloques sin permisos.
Cadena de bloques con permisos: Es aquella en la que el procesamiento de transacciones está desarrollado por una predefinida lista de sujetos con identidades conocidas. Por ello

generalmente no necesitan tokens nativos. Los tokens nativos son necesarios para proveer incentivos para los procesadores de transacciones. Por ello es típico que usen como protocolo de consenso prueba de participación

Según modelo de cambio de estado. Las cadenas de bloques también se pueden clasificar según el modelo de cambio de estado en la base de datos en:
Basado en el gasto de salidas de transacciones, también llamado modelo UTXO (en referencia a los UTXO de Bitcoin): En ellas cada transacción gasta salidas de transacciones anteriores y produce nuevas salidas que serán consumidas en transacciones posteriores. A este tipo de cadenas de bloques pertenecen por ejemplo las de Bitcoin, R3, Blockstream, BOSCoin y Qtum. Este enfoque tiene ventajas como:
En la propia estructura de la cadena existe una prueba de que nunca se puede gastar dos veces ya que cada transacción prueba que la suma de sus entradas es más grande que la suma de sus salidas.
Cada transacción puede ser procesada en paralelo porque son totalmente independientes y no hay conflictos en las salidas.
Sin embargo el problema de este tipo de cadenas es que solo son utilizables para aplicaciones donde cada salida es propiedad de uno y solo un individuo como por ejemplo es el caso de las monedas digitales. Una salida multipropietario sería muy lenta y no sería eficiente para aplicaciones de propósito general. Por ejemplo, supongamos un contrato inteligente que implementa un contador que puede ser incrementado. Imagina que hay algún incentivo económico para que cada nodo incremente en uno el contador, y que hay 1000 nodos activamente intentado incrementarlo. Usando este modelo de cadena de bloques tendríamos una salida con el valor del contador que sería solicitada por muchos nodos. Finalmente un nodo tendría éxito y produciría una transacción con una nueva salida con el contador incrementado en una unidad más. El resto de nodos estarían forzados a reintentar hasta que su transacción sea aceptada. Este sistema es muy lento e ineficiente. Esto es debido a que un cuando se realiza la transacción se bloquea la salida, se realiza una transformación y finalmente se produce la nueva salida. Esta claro que sería mucho más óptimo si se realizara todo de una sola vez y se produjera directamente el estado resultante. Además el problema puede estar no solo en el tiempo de las transacción si no también en el de proceso. Supongamos que el contador tiene adjunto un buffer de 1MB cuyo valor cambia de forma determinista cada vez que el contador cambia. Se tendría que procesar 1MB cada vez que realizara una transacción.

Basado en mensajes. En este caso, la cadena de bloques representa un consenso sobre el orden de los mensajes y el estado es derivado de forma determinísta a partir de estos mensajes. Este enfoque es utilizado por las cadenas de bloques de Steem y Bitshares. Por ejemplo para implementar un contador cada usuario debería simplemente firmar un mensaje pidiendo el incremento en 1. No se necesita saber el estado actual del contador para que el mensaje sea válido. En este modelo si 1000 nodos envían la petición al mismo tiempo, el productor del bloque podría agregar todas la peticiones en un bloque y en un solo paso el contador pasaría de valer de 0 a valer 1000. Una aplicación del mundo real que aprovecharía las cualidades de este modelo sería el siguiente:
Se emite una orden de compra de productos financieros indicando un precio máximo y un volumen concreto. A partir de ahí hay una competición sobre

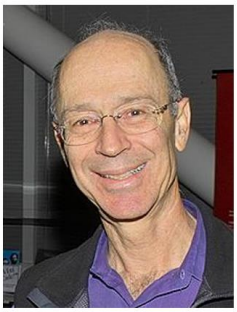
esa salida entre los participante que quieren la solicitud al mismo tiempo. Supongamos que se desea realizar la transacción de forma que sea lo más beneficiosa posible realizando una subasta a la baja para que la solicitud compre activos por el menor precio.

Sidechain:

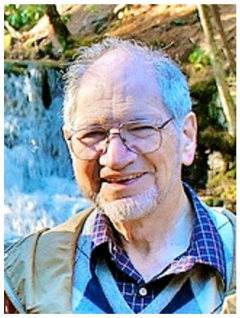
Una **cadena lateral**, en inglés **sidechain**, es una cadena de bloques que valida datos desde otra cadena de bloques a la que se llama principal. Su utilidad principal es poder aportar funcionalidades nuevas, las cuales pueden estar en periodo de pruebas, apoyándose en la confianza ofrecida por la cadena de bloques principal. Las cadenas laterales funcionan de forma similar a como hacían las monedas tradicionales con el patrón oro.
Un ejemplo de cadena de bloques que usa cadenas laterales es Lisk. Debido a la popularidad de Bitcoin y la enorme fuerza de su red para dar confianza mediante su algoritmo de consenso por prueba de trabajo, se quiere aprovechar como cadena de bloques principal y construir cadenas laterales vinculadas que se apoyen en ella. Una cadena lateral vinculada es una cadena lateral cuyos activos pueden ser importados desde y hacia la otra cadena. Este tipo de cadenas se puede conseguir de las siguiente formas:
Vinculación federada, en inglés federated peg. Una cadena lateral federada es una cadena lateral en la que el consenso es alcanzado cuando cierto número de partes están de acuerdo (confianza semicentralizada). Por tanto tenemos que tener confianza en ciertas entidades. Este es el tipo de cadena lateral Liquid, de código cerrado, propuesta por Blockstream.
Vinculación SPV, en inglés SPV peg donde SPV viene de Simplified Payment Verification. Usa pruebas SPV. Esencialmente una prueba SPV está compuesta de una lista de cabeceras de bloque que demuestran prueba de trabajo y una prueba criptográfica de que una salida fue creada en uno de los bloques de la lista. Esto permite a los verificadores chequear que cierta cantidad de trabajo ha sido realizada para la existencia de la salida. Tal prueba puede ser invalidada por otra prueba demostrando la existencia de una cadena con más trabajo la cual no ha incluido el bloque que creó la salida. Por tanto no se requiere confianza en terceras partes. Es la forma ideal. Para conseguirla sobre Bitcoin el algoritmo tiene que ser modificado y es difícil alcanzar el consenso para tal modificación. Por ello se usa con bitcoin vinculación federada como medida temporal

Algoritmo A - Path Finding*

El algoritmo de búsqueda **A*** *(pronunciado "A asterisco" o "A estrella")* se clasifica dentro de los algoritmos de búsqueda en grafos. Presentado por primera vez en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael, el algoritmo A* encuentra, siempre y cuando se cumplan unas determinadas condiciones, el camino de menor coste entre un nodo origen y uno objetivo.
El problema de algunos algoritmos de búsqueda en grafos informados, como puede ser un algoritmo voraz, es que


Peter E. Hart


Nils J. Nilsson


Bertram Raphael

se guían en exclusiva por la función heurística, la cual puede no indicar el camino de coste más bajo, o por el coste real de desplazarse de un nodo a otro *(como los algoritmos de escalada)*, pudiéndose dar el caso de que sea necesario realizar un movimiento de coste mayor para alcanzar la solución. Es por ello bastante intuitivo el hecho de que un buen algoritmo de búsqueda informada debería tener en cuenta ambos factores, el valor heurístico de los nodos y el coste real del recorrido.

Así, el algoritmo **A*** utiliza una función de evaluación **f(n)=g(n)+h'(n)** , donde **h'(n)** representa el valor heurístico del nodo a evaluar desde el actual, **n**, hasta el final, y **g(n)**, el coste real del camino recorrido para llegar a dicho nodo, **n**, desde el nodo inicial. **A*** mantiene dos estructuras de datos auxiliares, que podemos denominar abiertos, implementado como una cola de prioridad *(ordenada por el valor f(n) de cada nodo)*, y cerrados, donde se guarda la información de los nodos que ya han sido visitados. En cada paso del algoritmo, se expande el nodo que esté primero en abiertos, y en caso de que no sea un nodo objetivo, calcula la **f(n)** de todos sus hijos, los inserta en abiertos, y pasa el nodo evaluado a cerrados.

El algoritmo es una combinación entre búsquedas del tipo primero en anchura con primero en profundidad: mientras que **h'(n) tiende a primero en profundidad**, **g(n) tiende a primero en anchura**. De este modo, se cambia de camino de búsqueda cada vez que existen nodos más prometedores.

Como todo algoritmo de búsqueda en amplitud, **A*** es un algoritmo completo: **en caso de existir una solución, siempre dará con ella**.
Si para todo nodo **n** del grafo se cumple **g(n)=0**, nos encontramos ante una búsqueda voraz. Si para todo nodo **n** del grafo se cumple **h(n)=0**, **A*** pasa a ser una búsqueda de coste uniforme no informada.
Para garantizar la optimización del algoritmo, la función **h(n)** debe ser heurística admisible, esto es, que no sobrestime el coste real de alcanzar el nodo objetivo.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

De no cumplirse dicha condición, el algoritmo pasa a denominarse simplemente **A**, y a pesar de seguir siendo completo, no se asegura que el resultado obtenido sea el camino de coste mínimo. Asimismo, si garantizamos que **h(n)** es consistente *(o monótona)*, es decir, que para cualquier nodo **n** y cualquiera de sus sucesores, el coste estimado de alcanzar el objetivo desde **n** no es mayor que el de alcanzar el sucesor más el coste de alcanzar el objetivo desde el sucesor.

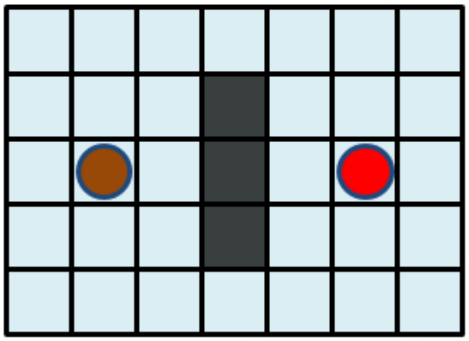
La **complejidad computacional** del algoritmo está íntimamente relacionada con la calidad de la heurística que se utilice en el problema. En el caso peor, con una heurística de pésima calidad, la complejidad será exponencial, mientras que en el caso mejor, con una buena **h'(n)**, el algoritmo se ejecutará en tiempo lineal. Para que esto último suceda, se debe cumplir que

$$h'(x) \leq g(y) + h'(y)$$

donde **h'** es una heurística óptima para el problema, como por ejemplo, el coste real de alcanzar el objetivo.

Complejidad en memoria: El espacio requerido por **A*** para ser ejecutado es su mayor problema. Dado que tiene que almacenar todos los posibles siguientes nodos de cada estado, la cantidad de memoria que requerirá será exponencial con respecto al tamaño del problema. Para solucionar este problema, se han propuesto diversas variaciones de este algoritmo, como pueden ser **RTA***, **IDA*** o **SMA***.

El problema inicial, como hemos visto, trata de trazar el camino para ir del punto A (círculo marrón) al punto B (círculo rojo).



La primera operación a realizar, es dividir el espacio en casillas. En nuestro ejemplo las casillas son **cuadradas**, pero realmente podremos adaptar el algoritmo a dividir el espacio de la forma que deseemos *(triángulos, hexágonos, etc)*. A partir de ahora cada casilla será una celda espacial *(crearemos una estructura de datos que la represente)* relacionada con sus adyacentes y con su **"padre";** el padre será la celda por la cual hemos llegado a tener en cuenta a la actual como posible candidata a formar parte del camino final.

Aspectos:
1) **Tener dos listas de celdas**; una abierta y una cerrada. En la lista abierta iremos introduciendo celdas que tenemos que evaluar, para ver si son buenas candidatas a formar parte del camino final. En la lista cerrada introduciremos las celdas ya evaluadas.
2) La evaluación de las celdas se hace en base a **dos factores**: la longitud del camino más corto para llegar desde el punto de origen a la celda actual, y la estimación del camino más corto que podría quedar para llegar al destino. La estimación para el destino se hace usando una función heurística, en concreto se suele usar la **Distancia Manhattan**, que esencialmente consiste en contar cuantas celdas nos separan del destino vertical y horizontalmente, sin movimientos diagonales; como si estuviéramos recorriendo manzanas en una gran ciudad y no pudiéramos atravesarlas, sólo bordearlas; de ahí viene el nombre.

Algoritmo:
Paso 0 Añadimos la celda origen a la lista abierta.

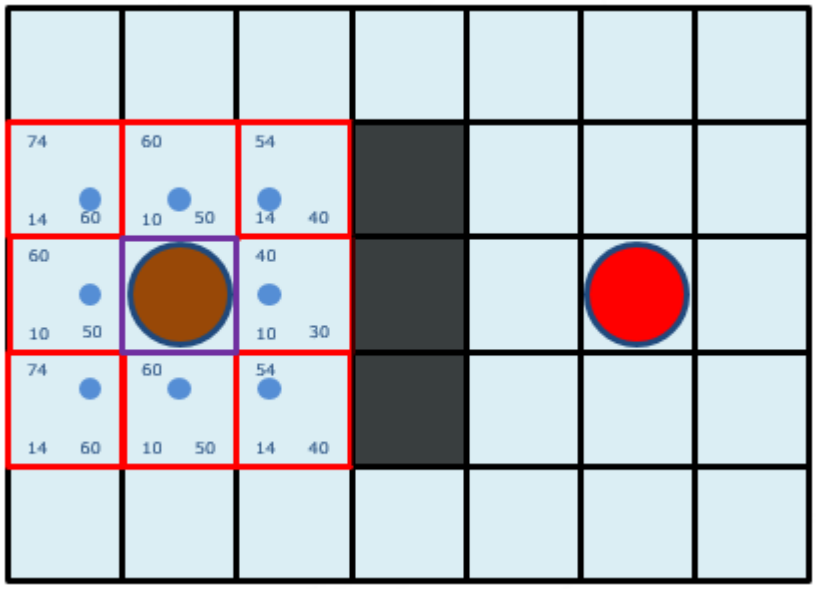
Paso 1 Tomamos el primer elemento de la lista abierta y lo sacamos y lo insertamos en la lista cerrada.
Paso 2 Tomamos las celdas adyacentes a la celda extraída.
Paso 3 Para cada celda adyacente:

A) Si la celda es la celda destino, hemos terminado. Recorremos inversamente la cadena de padres hasta llegar al origen para obtener el camino.
B) Si la celda representa un muro o terreno infranqueable; la ignoramos.
C) Si la celda ya está en la lista cerrada, la ignoramos.
D) Si la celda ya está en la lista abierta, comprobamos si su nueva **G** *(lo veremos más adelante)* es mejor que la actual, en cuyo caso recalculamos factores *(lo veremos más adelante)* y ponemos como padre de la celda a la celda extraída. En caso de que no sea mejor, la ignoramos.
E) Para el resto de celdas adyacentes, les establecemos como padre la celda extraída y recalculamos factores. Después las añadimos a la lista abierta.

Paso 4 Ordenamos la lista abierta. La lista abierta es una lista ordenada de forma ascendente en función del factor **F** de las celdas.
Paso 5 Volver al **Paso 1**.

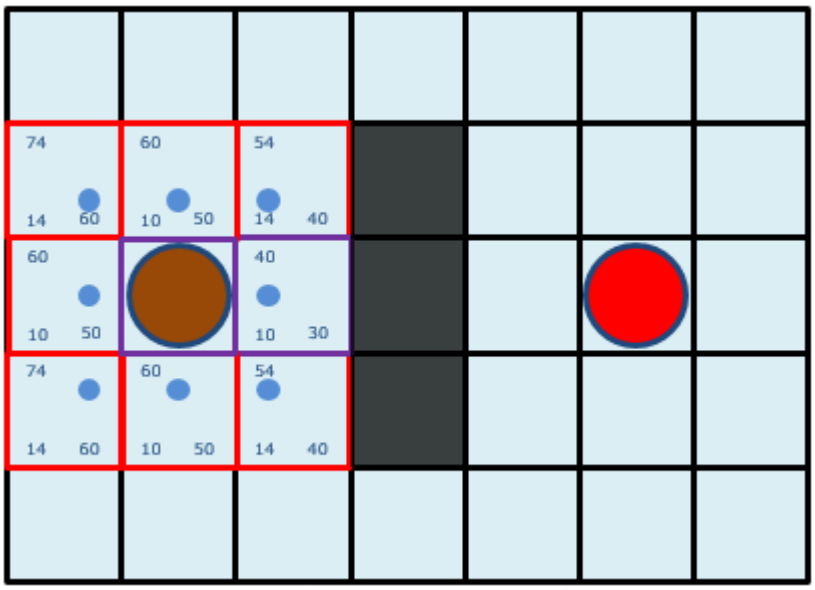
Cada celda va a tener 3 factores. **G**, **H** y **F**.
– **G**: Es el coste de ir desde la celda origen a la celda actual. Cada vez que nos movamos un paso en horizontal o vertical, añadiremos 10 puntos de coste. Cada vez que nos movamos en diagonal, añadiremos 14. ¿Por qué 14? Porque aunque geométricamente la proporción exacta debería ser 14.14213 ($\sqrt{10^2 + 10^2}$), 14 es una buena aproximación entera que nos hará ganar velocidad al evitar el uso de coma flotante *(nota: hay lenguajes como javascript en los que esta mejora no aplica, ya que todos los números se tratan como floats internamente)*.
– **H**: Es la distancia mínima y optimista, sin usar diagonales, que queda hasta el destino. La heurística basada en **Distancia Manhattan** de la que hemos hablado antes.
– **F**: Es la suma de **G** y **H**.

Iteración 1:



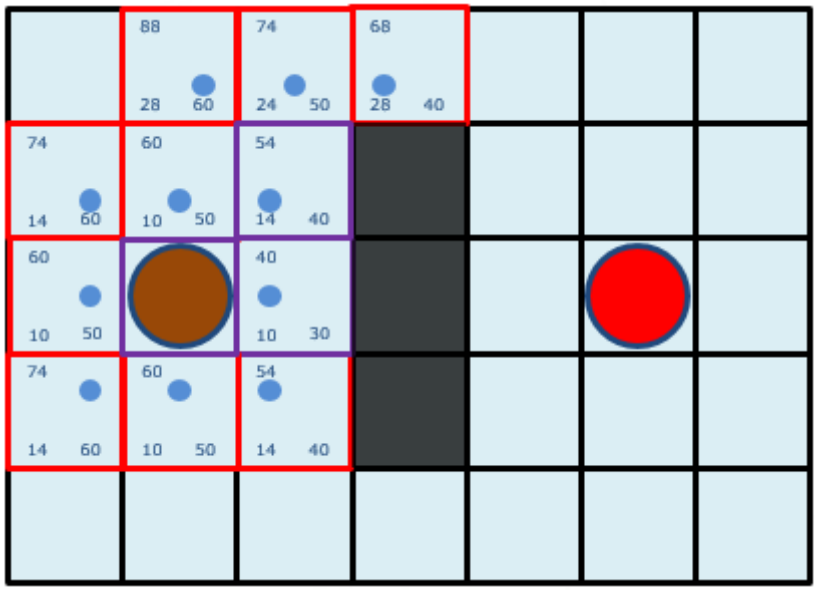
Tras la primera iteración, la celda origen queda en la lista cerrada *(borde violeta).* Las adyacentes quedan en la lista abierta *(borde rojo)*. Las celdas tienen sus tres valores **G**, **H** y **F** calculados. La **F** aparece en la esquina superior izquierda de cada celda. La **G** en la inferior izquierda. La **F** en la inferior derecha.
Además cada celda tiene un círculo azul que indica qué celda es su padre, tras esta iteración, todas las celdas de la lista abierta tienen como padre a la celda origen.

Iteración 2:



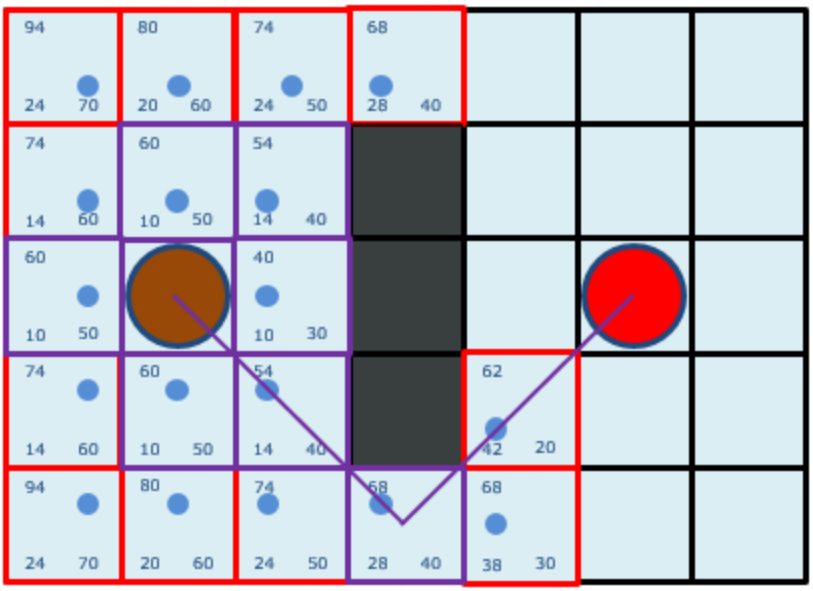
En esta iteración hemos añadido la celda a la derecha *(la de menor F de la lista abierta)* de la celda origen a la lista cerrada *(borde violeta).* No se han producido más cambios.

Iteración 3:



Se añade la celda superior derecha desde el origen a la lista cerrada *(es la de menor **F** de la lista abierta)* y se añaden las adyacentes que no estaban ya.

Iteración N:



Y algoritmo sigue. De forma sencilla, hasta alcanzar el objetivo.

Ejemplo mas grande: