

Algoritmos III - Apuntes para final

Gianfranco Zamboni

9 de abril de 2022

Índice

1. Introducción	2
1.1. Análisis de algoritmos	2
1.1.1. Modelo Random Access Machine (RAM)	3
1.2. Cálculo de complejidad	3
1.2.1. Tamaño de una instancia	3
1.2.2. Funciones de complejidad	4
1.2.3. Problemas bien resueltos	4
1.3. Técnicas de diseño de algoritmos	4
1.3.1. Algoritmos golosos	4
1.3.2. Divide and Conquer	5
1.3.3. Backtracking	6
1.3.4. Programación dinámica	6
1.3.5. Algoritmos probabilísticos	7
1.3.6. Heurísticas	7
2. Grafos	9
2.1. Definiciones básicas	9
2.1.1. Tipos de grafo	9
2.1.2. Propiedades de un nodo	10
A. Hoja de complejidades	10

1. Introducción

Un **algoritmo** es una secuencia **finita** de pasos **precisos** (no deben requerir tomar ninguna decisión subjetiva, ni hacer uso de la intuición o la creatividad) necesarias para llevar a cabo un cálculo de manera correcta.

Se dice que un algoritmo está **bien definido** cuando toda ejecución del mismo bajo los mismos parámetros devuelve el mismo resultado. Hay que tener en cuenta que si bien esto es lo deseable en la mayoría de los casos, muchas veces es necesario usar variables aleatorias o heurísticas para conseguir un resultado medianamente decente para problemas que no tienen solución o que son muy difíciles de calcular de manera exacta.

1.1. Análisis de algoritmos

Análisis empírico: Implementarlo en una máquina determinada utilizando un lenguaje determinado, correrlo para un conjunto de instancias y comparar sus tiempos de ejecución. Esto implicaría perder tiempo y esfuerzo programándolo, ejecutándolo y además probarlo con un conjunto de instancias acotados por lo que realmente no se podría concluir nada sobre su comportamiento con instancias que queden fuera de ese conjunto.

Análisis teórico: Determinar matemáticamente la cantidad de tiempo que llevará su ejecución como una función de la medida de la instancia considerada, independizándolo de la máquina sobre la cuál es implementado el algoritmo y el lenguaje para hacerlo. Para esto es necesario definir un modelo de cómputo, un lenguaje sobre este modelo, cuales son las instancias del problema relevantes al mismo y sus tamaños.

Dicho de otra forma, desde el punto de vista teórico, se busca determinar la velocidad de crecimiento del tiempo o el espacio requerido por el algoritmo en función del tamaño de las instancias del problema.

Dado un problema, se le asigna un entero n (llamado el **tamaño** del problema) que servirá como medida de la cantidad de datos de entrada. Y se define como **complejidad temporal** del algoritmo al tiempo ejecución de un algoritmo expresado como una función que depende de n .

Análogamente, se puede definir la **complejidad espacial** del algoritmo como el espacio de memoria necesario para su ejecución en función del tamaño del problema.

Complejidad peor caso: Se toma como complejidad del algoritmo la función que dada una instancia de tamaño de n , devuelve la máxima complejidad posible sobre todas las instancias de ese tamaño.

Complejidad caso promedio: Se toma como complejidad del algoritmo la función que dada una instancia de tamaño de n , devuelve el promedio de todas las complejidades posible sobre todas las instancias de ese tamaño.

1.1.1. Modelo Random Access Machine (RAM)

Una Random Access Machine (RAM) modela una computadora con un único registro acumulador en la cuál las instrucciones no pueden modificarse. La misma está formada por:

- Una **unidad de entrada** que contiene los datos necesarios para poder ejecutar una corrida del algoritmo. La misma es de solo lectura y, en cada una de sus celdas, puede contener un entero de tamaño arbitrario. Cuando un valor es leído, la cinta se avanza una celda.
- Una **unidad de salida** que es de solo escritura y se encuentra inicialmente vacía. Cuando una instrucción es ejecutada, se imprime un entero en la celda que se encuentra bajo la lectora y luego se la avanza una posición.
- Una **memoria** que consiste en una secuencia infinita de registros, cada uno de los cuales puede contener un entero de tamaño arbitrario.
- Un **programa** es una secuencia de instrucciones etiquetadas que no está almacenado en memoria. Asumimos que las instrucciones que contiene un programa, son las instrucciones aritméticas básicas, de entradas/salidas y direccionamiento indirecto y de branching encontradas en la mayoría de las computadoras.

Este modelo se usa cuando los tamaños de los problemas a resolver es lo suficientemente pequeño como para que entren en la memoria principal de una computadora o cuando los enteros usados en la computación son los suficientemente chicos como para que entren en una palabra de la misma.

1.2. Cálculo de complejidad

1.2.1. Tamaño de una instancia

Para especificar la complejidad en peor caso (tanto espacial como temporal), debemos especificar el tiempo de ejecución requerido por cada instrucción y el espacio de cada registro.

La forma más simple para esto es usar el **criterio de costo uniforme** en el que cada instrucción requiere una unidad de tiempo y cada registro requiere una unidad de espacio. En general, usaremos este criterio para analizar problemas de ordenamientos o sobre grafos y matrices.

Una segunda opción (un poco más realista), es tomar en cuenta el tamaño real (en bits) de los operandos. Este el **criterio de costo logarítmico** que toma en cuenta que son necesarios para representar los datos.

Alfabeto: Es el conjunto de símbolos que puede interpretar en una máquina. Por ejemplo, las computadoras actuales reconocen el alfabeto binario $\{0, 1\}$.

Dada una instancia I , se define $|I|$ como el número de símbolos de un alfabeto finito necesarios para codificar I . Este tamaño depende del alfabeto elegido. En el caso de las computadoras actuales, que se manejan con el alfabeto binario, para almacenar un número natural n , se necesitan $L(n) = \lfloor \log_2 n \rfloor + 1$ bits.

Usaremos este criterio cuando analicemos problemas sobre números (por ejemplo, el calculo de un factorial).

1.2.2. Funciones de complejidad

Dadas dos funciones $f, g : \mathbb{N} \rightarrow \mathbb{R}$ decimos que:

- $f(n) = O(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que

$$f(n) \leq cg(n) \text{ para todo } n \geq n_0$$

- $f(n) = \Omega(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que

$$f(n) \geq cg(n) \text{ para todo } n \geq n_0$$

- $f(n) = \Theta(g(n))$ si

$$f(n) = O(g(n)) \text{ y } f(n) = \Omega(g(n))$$

1.2.3. Problemas bien resueltos

Decimos que un problema está **bien resuelto** si existe un algoritmo de complejidad polinomial que lo resuelva. Es decir, que no consideraremos soluciones satisfactorios a los algoritmos supra-polinomiales.

1.3. Técnicas de diseño de algoritmos

1.3.1. Algoritmos golosos

Los algoritmos golosos toman decisiones basandose en la información disponible actualmente, sin considerar los efectos que esas decisiones podrían tener en el futuro. Son fáciles de inventar, implementar y son eficientes. Sin embargo, aunque muchas veces proporcionan heurísticas sencillas para resolver problemas de optimización, no siempre funcionan.

Comunmente, los algoritmos golosos y los problemas que pueden resolver tienen las siguientes características:

- El problema se debe resolver de alguna manera óptima. Para construir su solución se mantiene **un conjunto de candidatos** entre los cuales el algoritmo puede elegir para agregar a la solución en el próximo paso. Por ejemplo, un conjunto de monedas o de nodos de un grafos.
- A medida que el algoritmo progresa, se acumulan dos conjuntos: Uno contiene candidatos que ya fueron considerados y usados; el otro contiene los candadidatos que fueron consirados y rechazados.
- Hay una **función de verificación** que dado un conjunto de candidatos verifica si puede ser una solución valida al problema propuesto aunque esta no sea la solución óptima.

- Hay una **función de factibilidad** que dado un conjunto de candidatos verifica si se puede extender con nuevos candidatos para conseguir una solución al problema.
- Hay una **función de selección** que decide cual de los candidatos disponibles (que no fueron elegidos ni rechazados) es el más prometedor para hacernos a la solución deseada.
- Hay una **función objetivo** que nos da el valor de la solución encontrada. Por ejemplo, el número de monedas utilizadas para el cambio, o la cantidad de nodos usados en un camino, o cualquier otro valor que estemos intentando optimizar. A diferencia de las otras tres funciones mencionadas anteriormente, esta no aparece explícitamente en el algoritmo goloso.

Estructura general de un algoritmo goloso:

```

function GREEDY( $C$ : Set)                                ▷  $C$  es el conjunto de candidatos
   $S \leftarrow \emptyset$                                      ▷ Conjunto en el que se construye la solución
  while  $C \neq \emptyset \wedge \neg \text{ESOLUCION}(S)$  do
     $x \leftarrow \text{SELECCIONAR}(C)$ 
     $C \leftarrow C \setminus \{x\}$ 
    if  $\text{ESVALIDO}(S \cup \{x\})$  then
       $S \leftarrow S \cup \{x\}$ 
    end if
  end while
  if  $\text{ESOLUCION}(S)$  then return  $S$ 
  else
    return Error: No hay solución
  end if
end function

```

1.3.2. Divide and Conquer

Es una técnica que consiste en dividir la instancia de un problema a resolver en varias instancias más pequeñas del mismo problema y resolverlas de manera independiente para luego combinar sus soluciones en la solución de la instancia original.

Para que valga la pena realizar un algoritmo con esta técnica se deben cumplir tres condiciones:

1. Debe ser posible descomponer las instancias en subinstancias y combinar las subsoluciones de manera eficiente.
2. Las subinstancias deben ser todas más o menos del mismo tamaño. La mayoría de los algoritmos de divide and conquer son tales que el tamaño de una subinstancia I , es aproximadamente $\frac{n}{b}$ donde n es el tamaño de la instancia original y b alguna constante.

La estructura general de un algoritmo de divide and conquer es la siguiente:

```

function DIVIDEANDCONQUER( $I$ : Instancia del problema)
  if  $I$  es suficientemente pequeño o simple then return resolver( $I$ )
  else
    Descomponer  $I$  en subinstancias  $I_1, I_2, \dots, I_k$ 
    for  $i \leftarrow 1$  to  $k$  do
       $y_i \leftarrow$  DIVIDEANDCONQUER( $I_i$ )
    end for
    Combinar las soluciones  $y_1, y_2, \dots, y_k$  obtenidas para obtener la solución  $y$  de  $I$ 
    return  $y$ 
  end if
end function

```

1.3.3. Backtracking

Es una técnica que permite recorrer sistemáticamente todas las posibles configuraciones del espacio de soluciones de un problema computacional. Cuando el algoritmo comienza, no se sabe nada sobre la solución del problema. A medida que va avanzando, se agrega un elemento a la solución para ir consiguiendo soluciones parciales. Si en algún momento, la solución parcial deja de poder ser extendida entonces esa solución se descarta.

Se puede pensar este espacio como un árbol dirigido, donde cada vértice representa una solución parcial s y un vértice x es hijo de s si la solución parcial x se puede extender desde s .

El template general para algoritmos de este tipo es:

```

sol: Vector  $\leftarrow []$ 
encontro: Booleano  $\leftarrow false$ 
function BACKTRACKING( $v[1 \dots k], s$ : Instancia del problema)
  if  $k == 0 \wedge esSolucion(s)$  then
    sol  $\leftarrow s$ 
    encontro  $\leftarrow true$ 
  else
    for  $i \leftarrow 1$  to  $k$  do
      if encontro then return
    end if
    end for
  end if
end function

```

1.3.4. Programación dinámica

Es una técnica que consiste en tomar la subinstancia más pequeña del problema, resolverla e ir extendiendo la solución para instancias cada vez mas grandes. Esta técnica es un método **bottom-up**, es decir tomamos las instancias más simples del problema y la vamos extendiendo y combinando hasta conseguir la solución correspondiente a la instancia original.

La programación dinámica se utiliza para resolver problemas de optimización que cumplen con el **principio de optimalidad de bellman** que nos asegura que si tomamos una serie de decisiones óptimas, entonces todas las subsecuencias de decisiones de esa serie son óptimas.

1.3.5. Algoritmos probabilísticos

Cuando un algoritmo tiene que hacer una elección, a veces es preferible elegir al azar en vez de gastar mucho tiempo tratando de ver cual es la mejor opción.

Una de las características de los algoritmos probabilísticos es que puede comportarse de maneras distintas si se lo usa para computar una misma instancia dos o más veces. Su tiempo de ejecución y resultados pueden variar considerablemente de un uso a otro pudiendo incluso nunca llegar a terminar en ciertas corridas para una instancia específica. Sin embargo, dada la posibilidad de reiniciar el algoritmo y obtener un resultado válido éste es un comportamiento que no molesta.

Por otro lado, una consecuencia de este comportamiento es que, si hay más de una solución al problema, las mismas se pueden obtener corriendo el algoritmo más de una vez.

Los algoritmos probabilísticos se pueden clasificar dependiendo de la probabilidad de que devuelvan una respuesta correcta:

- **Algoritmos numéricos:** Estos algoritmos devuelven un intervalo de confianza (del estilo “La solución es $x \pm y$ con probabilidad z ”). Por lo general, mientras más tiempo de proceso les damos, más preciso es el intervalo que devuelven.
- **Algoritmos de Montecarlo:** Son algoritmos que dan la respuesta exacta con alta probabilidad. Por lo general, no es posible verificar que la respuesta dada sea correcta sin embargo la probabilidad de error disminuye mientras más tiempo se este ejecutando el algoritmo.
- **Algoritmos Las Vegas:** Son algoritmos que siempre dan una respuesta correcta pero puede llegar a no dar ninguna respuesta.
- **Algoritmos Sherwood:** Son algoritmos que agregan una componente aleatoria a algoritmos determinísticos para tratar de evitar tiempos de ejecución del peor caso. Un ejemplo de esto es el Quicksort con pivote seleccionado aleatoriamente.

1.3.6. Heurísticas

Dado un problema de optimización Π difícil de resolver (y para el cual probablemente no exista un algoritmo eficiente), los **algoritmos heurísticos** definen un procedimiento que intenta conseguir soluciones para el mismo pero puede devolver resultados erróneos o no devolver nada directamente.

En otras palabras: Sea I es una instancia del problema y $x^*(I)$ el valor óptimo de la función a optimizar en dicha instancia. Buscamos crear una heurística H tal que la solución $x^H(I)$ devuelta por la misma sea lo más cercano a $x^*(I)$ posible.

Algoritmos aproximados: Decimos que H es un algoritmo ϵ -aproximado para el problema \square si para algún $\epsilon > 0$ vale que $|x^H(I) - x^*(I)| \leq \epsilon|x^*(I)|$

Algoritmos con certificados: Por lo general, si bien los problemas difíciles no pueden ser resueltos con algoritmos polinomiales si se puede verificar que las soluciones provistas por las heurísticas sean correctas con algoritmos polinomiales.

Se dice que los algoritmos diseñados para realizar esta verificación proveen un certificado que afirma la validez de las soluciones propuestas por una heurística.

Bibliografía

- (1) Algoritmos y Estructuras de Datos III - Clases Teóricas, 2019.
- (2) Brassard, G. y Bratley, P., *Fundamentals of Algorithmics*; Prentice-Hall, Inc.: USA, 1996.
- (4) Aho, A. V. y Hopcroft, J. E. en *The Design and Analysis of Computer Algorithms*, 1st; Addison-Wesley Longman Publishing Co., Inc.: USA, 1974; cap. 1. Models Of Computation, págs. 1-43.

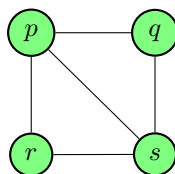


Figura 1: $G = ([p, q, r, s], [(p, q), (p, s), (q, s), (r, s)])$

2. Grafos

2.1. Definiciones básicas

2.1.1. Tipos de grafo

Un grafo $G = (V, X)$ es un par de conjuntos, donde V es un conjunto de **puntos / nodos / vértices** y X es un subconjunto del conjunto de pares no ordenados de elementos distintos de V . Los elementos de X se llaman **aristas, ejes o arcos**.

Dados v y $w \in V$, si $e = (v, w) \in X$ se dice que v y w son **adyacentes** y que e es **incidente** a v y a w .

Multigrafo: Es un grafo en el que puede haber varias aristas entre el mismo par de nodos distintos.



Figura 2: Multigrafo

Seudografo: Es un grafo en el que puede haber varias aristas entre cada par de nodos y también puede haber aristas (*loops*) que unan a un nodo con sí mismo.



Figura 3: Multigrafo

N

2.1.2. Propiedades de un nodo

A. Hoja de complejidades

Algoritmo	Complejidad
De Búsqueda	
Secuencial	$O(n)$
Binaria	$O(\log n)$
De Ordenamiento	
Bubblesort	$O(n^2)$
Quicksort	$O(n^2)$
Heapsort	$O(n \log n)^*$

* $O(n \log n)$ es la complejidad óptima para algoritmos de ordenamiento basados en comparaciones.

Grafos Definiciones básicas: adyacencia, grado de un nodo, isomorfismos, caminos, conexión, etc. Grafos bipartitos. Árboles: caracterización, árboles orientados, árbol generador. Enumeración. Grafos eulerianos y hamiltonianos. Planaridad. Coloreo. Número cromático. Matching, conjunto independiente, recubrimiento. Recubrimiento de aristas y vértices.

Algoritmos en grafos y aplicaciones Representación de un grafo en la computadora: matrices de incidencia y adyacencia, listas. Algoritmos de búsqueda en grafos: BFS, DFS, A*. Mínimo árbol generador, algoritmos de Prim y Kruskal. Árboles ordenados: códigos unívocamente descifrables. Algoritmos para detección de circuitos. Algoritmos para encontrar el camino mínimo en un grafo: Dijkstra, Ford, Dantzig. Planificación de procesos: PERT/CPM. Algoritmos heurísticos: ejemplos. Nociones de evaluación de heurísticas y de técnicas metaheurísticas. Algoritmos aproximados. Heurísticas para el problema del viajante de comercio. Algoritmos para detectar planaridad. Algoritmos para coloreo de grafos. Algoritmos para encontrar el flujo máximo en una red: Ford y Fulkerson. Matching: algoritmos para correspondencias máximas en grafos bipartitos. Otras aplicaciones.

Problemas NP-completos Problemas tratables e intratables. Problemas de decisión. P y NP. Maquinas de Turing no determinísticas. Problemas NP-completos. Relación entre P y NP. Problemas de grafos NP-completos: coloreo de grafos, grafos hamiltonianos, recubrimiento mínimo de las aristas, corte máximo, etc.