Algoritmos III - Apuntes para final

Gianfranco Zamboni

10 de abril de 2022

Índice

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 heurísiticas para conseguir un resultado medianamente decente para problemas que no tienen solución o que son muy dificiles de calcular de manera exacta.

Pseudocódigo: Es una descripción informal de alto nivel de un algoritmo que transmite el procedimiento a realizar de forma clara y precisa.

1.1. Analisis de algoritmos

Cuando tenemos más de un algoritmo para resolver un problema y queremos elegir el *mejor* entre ellos, hay diferentes criterios que podemos analizar para medir la eficiencia de los mismos, según el contexto de aplicación. Los recursos de mayor interés suelen ser el tiempo de cómputo y el espacio de almacenamiento requerido. Podemos considerar dos enfoques:

Análisis empírico: Implementar el algoritmo 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 programandolo, ejecutandolo y además probarlo con un conjunto de instancias acotados por lo que realmente no podríamos concluir nada sobre su comportomiento 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 medidad de la instancia considerada, independinzándolo 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.

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 escritrura 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 memoría 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

Una operación es elemental si su tiempo de ejecución puede ser acotado por una constante dependiente sólo de la implementación particular utilizada. Esta constantes no depende de la medidad de los parámetros de la instancia considerada.

En una máquina RAM, se asume que toda instrucción es una operación elemental con un **tiempo de ejecución** asociado y definimos el tiempo de ejecución de un programa como:

$t_A(I) =$ suma de los tiempos de ejecución de las instrucciones realizadas por el programa A con la instancia I

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

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.

Dadas dos funciones $f, g : \mathbb{N} \to \mathbb{R}$ 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 sastifactorios 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 actualemente, sin considerar los efectos que esas decisiones podrían tener en el fúturo. 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 carácteristicas:

- 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 consiradosy rechazados.
- Hay una **función de verificación** que dado un conjunto de candidatos verifica si puede ser una solución valida al problema propuesto auque 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 hacercanos a la solución deseada.
- Hay una f**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 cámino, o cualquier otro valor que estemos intentando optimizar. A diferencia de las otras tres funciones mencionadas anteriormente, esta no aparece explicitamente en el algoritmo goloso.

Estructura general de un algoritmo goloso:

```
function GREEDY(C: Set)                          ▷ C es el conjunto de candidatos
    S ← ∅                                        ▷ Conjunto en el que se construye la solución
    while C ≠ ∅ ∧ ¬esSolucion(S) do
        x ← SELECCIONAR(C)
        C ← C \ {x}
        if esValido(S ∪ {x}) then
            S ← S ∪ {x}
        end if
    end while
```

```
    if esSolucion(S) then return S
    else
        return Error: No hay solución
    end if
end function
```

1.3.2. Recursividad

Un algoritmo recursivo se define en términos de si mismo, esto es, en su cuerpo aparece una aplicación suya. En general, las llamadas recursivas se aplican sobre párametros más pequeños que los iniciales. Cuando el parámetro sobre el que se aplica es lo suficientemente chico se ejecuta lo que se llama el **caso base** que, para su resolución, no necesita llamar a la misma función, terminando así el proceso recursivo.

En general, para calcular la complejidad de un algoritmo recursivo, se debe plantear primero las ecuaciones de recurrencia y luego, utilizando herramientas matemáticas, encontrar la fórmula cerrada (que no dependa de la complejidad de instancias más chicas) de esta ecuación.

1.3.3. 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 algortimo 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, ..., I_k
        for i ← 1 to k do
            y_i ← DivideAndConquer(I_i)
        end for
        Combinar las soluciones y_1, y_2, ..., y_k obtenidas para obtener la solución y de I
        return y
    end if
```

end function

En este tipo de algoritmos, generalmente las instancias que son caso base toman tiempo constante c y para las casos recursivos se pueden identificar tres puntos críticos:

- Cantidad r de llamadas recursivas que haremos.
- Medida de cada subproblema: $\frac{n}{b}$ para alguna constante b.
- Tiempo requerido por el algoritmo para ejecutar, descomponer y combinar para una instancia de tamaño n, $g(n)$.

Entonces, el tiempo total $T(n)$ consumido por el algoritmo está definido por la siguiente ecuacución de recurrencia:

$$T(n) = \begin{cases} c & \text{si } n \text{ es caso base} \\ rT\left(\frac{n}{b}\right) + g(n) & \text{si } n \text{ es caso recursivo} \end{cases}$$

1.3.4. 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.

La idea básica es tratar de extender una solución parcial del problema hasta, eventualemnte, llegar a obtener una solución completa, que podría ser válida o no. Habitualmente, se utiliza un vector $a = (a_1, a_2, \ldots, a_n)$ para representar una solución candidata donde cada a_i pertenece a un dominio finito A_i. El espacio de soluciones es el producto cartesiano $A_1 \times \cdots \times A_n$.

En síntesis, en cada paso se extienden las soluciones parciales $a = (a_1, a_2, \ldots, a_k)$ con $k < n$, agregando un elemento más al final. Las nuevas soluciones parciales son sucesoras de la anterior.

Se puede pensar este espacio de búsqueda como un árbol dirigido donde cada vértice representa una solución parcial y un vértice x es hijo de y si la solución parcial x se puede extender desde la solución parcial y. La raíz del árbol se corresponde con el vector vacio.

El proceso de bactkracking recorre este árbol en profundidad. Cuando se puede deducir que una solución parcial no llevará a una solución válida, no es necesario seguir explorando esa rama del árbol y se retrocede hasta encontrar un vértice con un hijo válido por donde seguir la exploración.

El template general para algoritmos de este tipo es:

```
sol: Vector ← []
encontro: Booleano ← false
function BACKTRACKING(v[1...k], s: Instancia del problema)
    if k == 0 ∧ esSolucion(s) then
        sol ← s
```

```
        encontro ← true
    else
        for i ← 1 to k do
            if encontro then return
            end if
        end for
    end if
end function
```

1.3.5. Programación dinámica

Esta técnica es aplicada a problemas de optimizacióncombinatoria, donde puede haber muchas soluciones fáctibles, cada una con un valor (o costo asociados) y pretende obtener la solución con mejor valor (o menor costo).

Al igual que dividir y conquistar, se divide al problema en problemas que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original. Se diferencian, en que esta técnica es iterativa (no recursiva) y es adecuada cuando es necesario resolver varias subisntancias del problema que son iguales ya que se van almacenando los resultados parciales obtenidos para su posterior utilización.

Principio de Optimalidad de Bellman: Un problema de optimización satisface este principio si en una solución óptima cada subsolución es a su vez óptima del subproblema correspondiente. Es decir, dada una secuencia óptima de decisiones, todas subsecuencia de ella es, a su vez, óptima.

Este principio es condición necesaria del problema para poder usar la técnica.

1.3.6. Algoritmos probabilísticos

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

Una de las características de los algoritmos probabilisticos 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.s

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

- **Algoritmos númericos**: 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 demos, mas preciso es el intervalo que devuelven.

- **Algoritmos de Montecarlo**: Son algoritmo 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.

1.3.7. Heurísticas

Dado un problema de optimización $\sqcap$ díficl 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 instancaia del problema y $x^*(I)$ el valor óptimo de la función a optmizar 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 apróximados: Decimos que H es un algoritmo $\epsilon-$aproximado para el problema $\sqcap$ 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 dificiles 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 la 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.

2. Grafos

2.1. Definiciones básicas

Los grafos proporcionand una forma conveniente y flexible de representar problemas de la vida real que consideran una red como estructura subyacente. Esta red puede ser física (como instalaciones eléctticas) o abstractas (que modelan relaciones menos tangibles, como relaciones sociales y bases de datos).

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

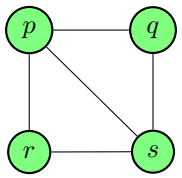

Figura 1: $G = ([p,\ q,\ r,\ s],\ [(p,q),\ (p,s),\ (q,s),\ (r,s)])$

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.

La definición de grafo no alcanza para modelar todas las situaciones posibles de una red. Por ejemplo, si se quisiese modelar la ruta de los aviones entre varias ciudades, deberiamos poder modelar varios vuelos entre dos ciudades:

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

Notación: $n = |V|$ y $m = |X|$

2.1.1. Propiedades de un nodo

Grado: El grado $d_G(v)$ de un nodo v es la cantidad de aristas incidentes a v en el grafo G. Notaremos $\Delta(G)$ al máximo grado de los vértices de G y $\delta(G)$ al mínimo.

Nota: En un seudografo, un loop aporta 2 al grado del vértice.

Teorema 1. *La suma de los grados de los nodos de un grafo es igual a dos veces el número de aristas, es decir:*

$$\sum_{i=1}^{n} v_i = 2m$$

DEMOSTRACIÓN

Sea $G = (V, X)$ un grafo de n nodos y m aristas. Haremos inducción en la cantidad de aristas:

Caso base: $m = 1$.
En este caso, el grafo G tiene sólo una arista que notamos $e = (u, w)$. Entonces $d(u) = d(w) = 1$ y $d(v) = 0$ para todo $v \in V$ tal que $v \neq u, w$. Por lo tanto,

$$\sum_{v \in V} d(v) = 2$$

y $2m = 2$, cumpliendose la propiedad.

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. Arboles: 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. Arboles 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.