

# Algoritmos y Estructuras de Datos III

## Resumen

amildie

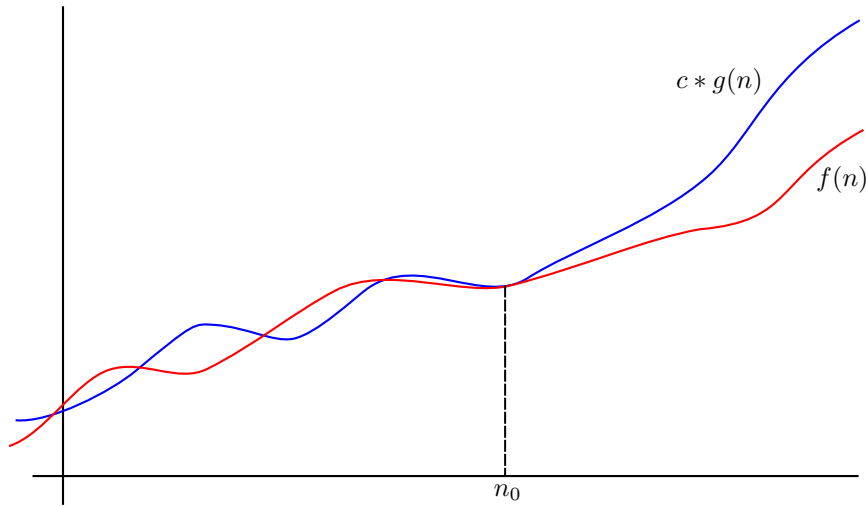
<b>1. Complejidad Computacional</b>	<b>3</b>
<b>2. Programación Dinámica</b>	<b>4</b>
2.1. Subestructura Optima . . . . .	4
2.2. Soluciones Sobrepuestas . . . . .	4
2.3. Programación Dinámica . . . . .	4
2.4. Ejemplos . . . . .	5
2.4.1. Subsecuencia continua de suma máxima . . . . .	5
2.4.2. Subsecuencia no-continua estrictamente creciente más larga . . . . .	6
2.4.3. El problema de la mochila . . . . .	7
2.4.4. Subconjunto de suma 0 . . . . .	9
<b>3. Grafos</b>	<b>10</b>
3.1. Caminos y distancia . . . . .	10
3.2. Subgrafos y bipartición . . . . .	11
3.3. Isomorfismo . . . . .	11
3.4. Digrafos . . . . .	12
3.5. Grafo de Líneas . . . . .	12
<b>4. Árboles</b>	<b>13</b>
4.1. Propiedades de los árboles . . . . .	13
4.2. Árboles enraizados . . . . .	13
4.3. Recorrido de árboles . . . . .	14
4.3.1. DFS . . . . .	14
4.3.2. BFS . . . . .	15
4.4. Árbol generador . . . . .	16
4.4.1. Algoritmo de Kruskal . . . . .	17
4.4.2. Algoritmo de Prim . . . . .	18
<b>5. Camino Mínimo</b>	<b>19</b>
5.1. BFS . . . . .	19
5.2. Algoritmo de Dijkstra (1 a n) . . . . .	20
5.3. Algoritmo de Bellman-Ford (1 a n) . . . . .	22
5.4. Algoritmo de Floyd-Warshall (n a m) . . . . .	23
5.5. Algoritmo de Dantzig (n a m) . . . . .	25
<b>6. Grafos Eulerianos y Hamiltonianos</b>	<b>26</b>
6.1. Grafos Eulerianos . . . . .	26
6.1.1. Problema del Cartero Chino . . . . .	26

6.2. Grafos Hamiltonianos . . . . .	27
6.2.1. The Travelling Salesman Problem . . . . .	27
<b>7. Planaridad</b>	<b>29</b>
7.1. Subdivisiones y Homeomorfismo . . . . .	29
7.1.1. Teorema de Kuratowski . . . . .	29
7.2. Contracciones . . . . .	29
7.2.1. Teorema de Whitney . . . . .	30
7.2.2. Teorema de Euler . . . . .	30
7.3. Algoritmo de Demoucron . . . . .	30
<b>8. Coloreo</b>	<b>31</b>
8.1. Cotas para $\chi(G)$ . . . . .	31
8.2. Algoritmos para coloreo de grafos . . . . .	32
8.3. Grafos perfectos . . . . .	32
8.4. Coloreo de ejes . . . . .	32
<b>9. Matchings y Conjuntos Independientes</b>	<b>33</b>
9.1. Matching maximal . . . . .	34
9.2. Matching máximo . . . . .	34
9.3. Caminos . . . . .	35
<b>10. Flujo en Redes</b>	<b>36</b>
10.1. Cortes . . . . .	36
10.2. Camino de Aumento . . . . .	36
10.3. Algoritmo de Ford-Fulkerson . . . . .	36
10.4. Algoritmo de Edmonds-Karp . . . . .	36
<b>11. Teoría de Complejidad</b>	<b>37</b>
11.1. La clase P . . . . .	37
11.2. La clase NP . . . . .	37
11.2.1. Reducciones Polinomiales . . . . .	37
11.3. La clase NP-Completo . . . . .	38
11.4. El problema SAT . . . . .	38
11.4.1. Teorema de Cook . . . . .	38
11.5. La clase NP-Intermedio . . . . .	38
11.6. La clase NP-Hard . . . . .	39
11.7. Algoritmos Pseudopolinomiales . . . . .	39
<b>12. Preguntas</b>	<b>40</b>

# 1. Complejidad Computacional

Dada una función  $g(n)$ , denominamos como  $O(g(n))$  al conjunto de funciones  $f(n)$  que cumplen que:

$$\exists c, n_0 \text{ tal que } 0 \leq f(n) \leq c * g(n), \forall n \geq n_0$$



Es decir  $O(g(n))$  es un conjunto de funciones  $f(n)$  que, a partir de cierto valor  $n_0$ , van a estar acotadas superiormente por  $c * g(n)$ . O, dicho de otra manera:

$$f(n) \in O(g(n)) \iff \exists c, n_0 \text{ tal que } 0 \leq f(n) \leq c * g(n), \forall n \geq n_0$$

Por ejemplo, si  $f(n) = n^2 + n + 1$  y  $g(n) = n^2$  podemos decir que  $f(n) \in O(g(n))$ , ya que si  $c = 10^{999}$  y  $n_0 = 10^{20}$  es fácil ver que se cumple que  $0 \leq n^2 + n + 1 \leq 10^{999} * n^2, \forall n \geq 10^{20}$ . Generalmente esto se describe diciendo que “ $f$  es  $n^2$ ”.

## 2. Programación Dinámica

### 2.1. Subestructura Óptima

La “*subestructura óptima*” es una propiedad que pueden exhibir algunos problemas. Se dice que un problema tiene subestructura óptima si el mismo cumple que una solución óptima del mismo puede ser construida a partir de soluciones óptimas de sus subproblemas.

Un ejemplo de esto es el problema del camino más corto. Supongamos dos puntos  $A$  y  $B$ , y un camino  $w$  que es el más corto entre ellos. Para cualquier par de puntos  $A'$  y  $B'$  dentro de  $w$ , el camino más corto  $w'$  entre ellos está necesariamente contenido adentro de  $w$ .

### 2.2. Soluciones Sobrepuestas

Al igual que la subestructura óptima, un problema tiene esta característica cuando sus subproblemas comparten soluciones entre ellos. Un ejemplo clásico de este fenómeno es el problema de calcular el  $n$ -ésimo número de Fibonacci.

La ecuación recursiva para calcularlo es  $f(n) = f(n-1) + f(n-2)$ . Supongamos entonces que queremos calcular  $f(5)$ . Voy a tener que calcular  $f(4)$  y  $f(3)$ . Pero para calcular  $f(4)$  voy a tener que calcular  $f(3)$  y  $f(2)$ . Es decir, voy a tener que calcular  $f(3)$  más de una vez.

Uno podría pensar que sería una buena idea cachear cada número de fibonacci calculado para no tener que recalcularlo más de una vez. Esta técnica se llama “*memoization*”<sup>1</sup>.

### 2.3. Programación Dinámica

Cuando un problema exhibe tanto subestructura óptima como soluciones superpuestas, es candidato a poseer un algoritmo para solucionarlo que emplee una técnica de desarrollo de algoritmos llamada “*programación dinámica*”.

Si un problema puede ser solucionado combinando soluciones no superpuestas de sus subproblemas, esta estrategia se llama “*divide & conquer*”. Es por eso que mergesort, por ejemplo, no es un problema de programación dinámica.

---

<sup>1</sup>Sí, *memoization*, sin r.

## 2.4. Ejemplos

### 2.4.1. Subsecuencia continua de suma máxima

Dado un arreglo  $A = \{-6, 2, -4, 1, 3, -1, 5, -1\}$ , dicho arreglo tiene varias subsecuencias  $s$  continuas, por ejemplo  $s_1 = \{1, 3, -1\}$ ,  $s_2 = \{-6\}$ ,  $s_3 = \{-1, 5, -1\}$ , etc. Cada una de estas subsecuencias tiene un valor  $sum(s_i)$  que representa la suma de todos los elementos de la misma. Encontrar el valor de la subsecuencia de suma máxima.

Notar que este problema sólo es interesante cuando hay tanto números negativos como positivos en el arreglo; ya que si fuesen todos positivos, la solución es simplemente devolver  $sum(A)$ , y si fuesen todos negativos, la solución es devolver el elemento más chico de  $A$ .

Un ejemplo elemental de esto es, por ejemplo teniendo el arreglo  $A = \{1, 2, 3, -100, 4, 5, 6\}$ . Como la suma de cualquier secuencia continua que no tenga al  $-100$  es bastante mayor a la suma de cualquier secuencia que lo tenga, es lógico asumir que la solución no va a tener al  $-100$ . Hay dos secuencias continuas de que no lo tienen:  $S_1 = \{1, 2, 3\}$  y  $S_2 = \{4, 5, 6\}$ . Acá es trivial ver que  $sum(S_2)$  es la respuesta al problema.

Pero en el arreglo  $A = \{-2, -3, 4, -1, -2, 1, 5, -3\}$  deja de ser tan evidente que el valor buscado es 7, la suma de la subsecuencia  $\{4, -1, -2, 1, 5\}$ .

Definimos el array  $S$ , donde  $S_i$  representa a la suma máxima de todas las subsecuencias continuas de  $A$  que tienen a  $A_i$  como último elemento. Es decir,  $A_i$  tiene que estar, por lo que en cada paso nos interesa saber si vamos a preservar la suma que veníamos armando desde antes o a arrancar con  $A_i$  como el inicio de una nueva subsecuencia. Es decir:

$$S(i) = \begin{cases} A[i] & \text{si } i = 0 \\ \max\{S(i-1) + A[i], A[i]\} & \text{si } i > 0 \end{cases}$$

Entonces el problema se reduce a armar a  $S$  mientras vamos buscando el máximo:

```
int sum(int* a, unsigned int n) {
    int max = numeric_limits<int>::min();
    int s[n];
    memset(s, -1, sizeof(s));
    for(int i = 0; i < n; ++i) {
        if (i == 0) {
            s[i] = a[i];
        } else {
            s[i] = std::max(a[i], a[i] + s[i-1]);
        }
        if (s[i] > max) {
            max = s[i];
        }
    }
    return max;
}
```

#### 2.4.2. Subsecuencia no-continua estrictamente creciente más larga

Dado un arreglo  $A = \{3, 2, 6, 4, 5, 1\}$ , encontrar una subsecuencia del mismo estrictamente creciente de longitud máxima.

Por ejemplo, para el array del enunciado la respuesta es  $A = \{2, 4, 5\}$ .

Similarmente al problema anterior, vamos a definir un vector de vectores  $S$ , donde  $S_i$  es la subsecuencia de  $A$  que termina en  $A_i$ .

$$S_i = \begin{cases} \{A_i\} & \text{si } i = 0 \\ \max\{S_j \text{ tal que } j < i \text{ y } A_j < A_i\} + A_i & \text{si } i > 0 \end{cases}$$

```
vector<int> lis(int* a, unsigned int n) {
    std::vector< std::vector<int> > L(n);
    L[0].push_back(a[0]);
    vector<int> res;

    for(int i = 1; i < n; ++i) {
        int maxLength = numeric_limits<int>::min();
        int maxIndex = 0;

        for(int j = i-1; j >= 0; --j) {
            if((int)L[j].size() > maxLength && L[j].back() < a[i]) {
                maxLength = L[j].size();
                maxIndex = j;
            }
        }

        std::vector<int> v;
        if(maxLength != numeric_limits<int>::min()) {
            v = L[maxIndex];
        }
        v.push_back(a[i]);
        L[i] = v;
    }

    unsigned int maxLength = numeric_limits<unsigned int>::min();
    for(int i = 0; i < L.size(); ++i) {
        if(L[i].size() > maxLength) {
            res = L[i];
            maxLength = L[i].size();
        }
    }

    return res;
}
```

### 2.4.3. El problema de la mochila

El “*problema de la mochila*” es un problema clásico de optimización combinatoria. Se trata de, dado un conjunto de ítems, cada uno con un determinado peso y valor, y una mochila que puede soportar hasta cierto peso, encontrar cuantas veces tengo que poner cada ítem para garantizar que la mochila contiene el máximo valor posible.

Este problema tiene diferentes variantes, pero cuando sólo tengo un ítem de cada peso, se lo conoce como el “*0-1 knapsack problem*”, que se define formalmente como:

Dado un conjunto de  $n$  ítems numerados de 1 hasta  $n$ , cada uno con un peso  $w_i$  y un valor  $v_i$ , y una máxima capacidad para la mochila  $W$ :

$$\text{maximizar } \sum_{i=1}^n v_i x_i \text{ cumpliendo que } \sum_{i=1}^n w_i x_i \leq W \text{ y con } x_i \in \{0, 1\}$$

La siguiente solución utilizar programación dinámica y corre en tiempo pseudo-polinomial. Asumimos que los pesos  $w[1], w[2], \dots, w[n]$  y el peso máximo de la mochila  $W$  son enteros positivos.

Primero definimos una matriz  $m[i, w]$  que representa el máximo valor que podemos alcanzar usando hasta el ítem del índice  $i$  teniendo un peso disponible de  $w$ . Claramente  $m[0, w] = 0$ , ya que el valor máximo sin poner ningún ítem en la mochila es 0.

Luego definimos la dinámica para calcular  $m[i, w]$  de la siguiente manera:

$$m[i, w] = \begin{cases} m[i-1, w] & \text{si } w_i > w \\ \max\{m[i-1, w], m[i-1, w-w_i] + v_i\} & \text{si } w_i \leq w \end{cases}$$

Por ejemplo, si estamos viendo quinto elemento pueden pasar dos cosas:

- Que  $w[5]$  sea más grande que  $w$ . Esto significa que ya no nos queda espacio para meter a  $w_5$ , por lo que no lo metemos, dejando el valor de  $m[5, w]$  igual al que teníamos en  $m[4, w]$ .
- Que  $w[5]$  sea menor o igual a  $w$ . Esto significa que todavía tenemos espacio suficiente para meter al quinto elemento en la mochila.

Pero que *podamos* meter al quinto elemento no significa que *debamos* hacerlo. Tenemos que elegir el máximo entre **no** poner al quinto elemento ( $m[i-1, w]$ ) y **sí** ponerlo ( $m[i-1, w-w_5] + v_i$ ).

Es decir, esta dinámica va reduciendo continuamente la capacidad restante en la mochila llamándose recursivamente cada vez con menos espacio disponible, hasta que eventualmente se queda sin lugar.

Para un peso máximo  $W$ , el valor de la solución está dado en  $m[n, W]$ .

La siguiente es una implementación en C++ de la dinámica anteriormente enunciada. Notar que los arrays son indexados desde 0 en lugar de 1.

```
void knapsack(int v[], int w[], int n, int W)
{
    int m[2000][2000];

    for(int j = 0; j <= W; ++j)
    {
        m[0][j] = 0;
    }

    for(int i = 0; i <= n; ++i)
    {
        for(int j = 0; j <= W; ++j)
        {
            if(w[i] > j)
            {
                m[i][j] = m[i-1][j];
            }
            else
            {
                m[i][j] = max(m[i-1][j], m[i-1][j-w[i]] + v[i]);
            }
        }
    }

    std::cout << "Rta:␣" << m[n-1][W] << std::endl;
}
```

Acá queda evidente que el tiempo de ejecución de la función depende de los dos ciclos anidados, donde uno va a iterar  $n$  veces, siendo  $n$  la cantidad de elementos y el otro va a iterar  $W$  veces, siendo  $W$  el peso máximo de la mochila.

Esto nos deja a la complejidad temporal del algoritmo como  $O(nW)$ . Notar que la matriz construida tiene que ser de  $n$  filas por  $W$  columnas, por lo que la complejidad espacial también es de  $O(nW)$ . Es decir, esta solución es pseudo-polinomial en tiempo y espacio.



#### 2.4.4. Subconjunto de suma 0

Este problema es en realidad un caso especial del problema de la mochila:

Dado un array  $A = \{a_1, a_2, \dots, a_n\}$  de enteros, determinar si existe una subsecuencia no necesariamente continua no vacía de  $A$  tal que la suma de todos sus elementos sea 0.

### 3. Grafos

Un “grafo”  $G = (V, X)$  es un par de conjuntos.  $V$  es un conjunto de “vértices” y  $X$  es un conjunto de “ejes”, que a su vez son pares no ordenados de los elementos de  $V$ . Vamos a definir a  $n = |V|$  y a  $m = |X|$ .

Dados dos vértices  $v, w \in V$  se dice que  $v$  y  $w$  son “adyacentes” si  $\exists e \in X$  tal que  $e = (v, w)$ . Un “multigrafo” es un grafo en el que pueden haber varios ejes entre el mismo par de vértices. Un “seudografo” es un multigrafo donde pueden haber ejes que unan a un vértice con si mismo.

El “grado” de un vértice  $v$  es la cantidad de ejes incidentes a él. Se nota con  $d(v)$  y da lugar a la siguiente propiedad:

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

Figura 1: La suma de los grados de todos los vértices de un grafo es igual a dos veces el número de aristas.

Un grafo se dice “completo” si todos los vértices son adyacentes entre si. Al grafo completo de  $n$  vértices se lo nota  $K_n$ .

Dado un grafo  $G$ , se denomina como el grafo “complemento” de  $G$  al grafo  $\overline{G}$  que tiene los mismos vértices que  $G$ , pero donde dichos vértices sólo son adyacentes en  $\overline{G}$  si no lo son en  $G$ .

#### 3.1. Caminos y distancia

Un “camino” en un grafo es una sucesión de ejes  $e_1, e_2, \dots, e_k$  tal que los extremos de  $e_i$  coinciden con uno de  $e_{i-1}$  y con uno de  $e_{i+1}$ , para todo  $i \in 2, \dots, k-1$ .

Cuando un camino no pasa dos veces por el mismo vértice, se lo denomina “camino simple”. Un “circuito” es un camino que empieza y termina en el mismo vértice. Cuando un circuito tiene 3 o más vértices se lo denomina “circuito simple”.

La “longitud” de un camino es la cantidad de vértices por los que pasa. La “distancia” entre dos vértices  $v$  y  $w$  de un grafo se define como la longitud del camino más corto entre ambos, y se nota con  $d(v, w)$ . Para todo vértice  $v$ ,  $d(v, v) = 0$ . Si no existe camino entre  $v$  y  $w$ , se dice que la distancia entre ambos es infinita.

Si un camino  $P$  entre  $v$  y  $w$  tiene longitud  $d(v, w)$  entonces  $P$  es un camino simple. Es decir, la distancia más corta entre dos vértices no va a dar vueltas en ningún lado.

Notar que si  $P$  es un camino entre  $u$  y  $v$  de longitud  $d(u, v)$  y tenemos los puntos  $z$  y  $w$  que están adentro de  $P$ , entonces  $P_{zw}$  es un camino entre  $z$  y  $w$  de longitud  $d(z, w)$ , donde  $P_{zw}$  es el subcamino interno de  $P$  entre  $z$  y  $w$ .

### 3.2. Subgrafos y bipartición

Dado un grafo  $G = (V, X)$ , un “*subgrafo*” del mismo es un grafo  $H = (V', X')$  que cumple que  $V' \subseteq V$  y que  $X' \subseteq X \cap (V' \times V')$ . Si se cumple que para todo  $u, v \in V'$ ,  $(u, v) \in X \iff (u, v) \in X'$  entonces  $H$  es un subgrafo “*inducido*”. Es decir, para cada par de vértices de  $H$ , se preservan los mismos ejes que tenían en  $G$ .

Un grafo se dice “*conexo*” si existe un camino que conecta cada par de vértices. Una “*componente conexa*” de un grafo  $G$  es un subgrafo conexo maximal de  $G$ .

Un grafo  $G = (V, X)$  se dice “*bipartito*” si existe una partición  $V_1, V_2$  de  $V$  tal que todos los ejes de  $G$  tienen un extremo en  $V_1$  y otro en  $V_2$ . Si todo vértice de  $V_1$  es adyacente a todo vértice de  $V_2$  entonces el  $G$  es “*bipartito completo*”. Un grafo es bipartito si y sólo si no tiene circuitos simples de longitud impar.

### 3.3. Isomorfismo

Dos grafos  $G = (V, X)$  y  $G' = (V', X')$  se dicen “*isomorfos*” si existe una función biyectiva  $f : V \rightarrow V'$  tal que para todo  $v, w \in V$ :

$$(v, w) \in X \iff (f(v), f(w)) \in X'$$

Si dos grafos son isomorfos, entonces:

- tienen el mismo número de vértices
- tienen el mismo número de ejes
- tienen el mismo número de componentes conexas
- $\forall k, 0 \leq k \leq n - 1$ , tienen el mismo número de vértices de grado  $k$
- $\forall k, 1 \leq k \leq n - 1$ , tienen el mismo número de caminos simples de longitud  $k$

No se conoce un algoritmo de tiempo polinomial para detectar si dos grafos son isomorfos, ni tampoco es NP-completo, pero en la práctica hay maneras de resolverlo eficientemente.

### 3.4. Digrafos

Un “*grafo orientado*” (o “*digrafo*”) es un grafo  $G = (V, X)$  en el cual los ejes de  $X$  tienen una dirección. Para cada vértice  $v$  se define a su “*grado de entrada*” ( $d_{in}(v)$ ) como a la cantidad de ejes en  $X$  que llegan a  $v$ . Es decir, que lo tienen como segundo elemento. El “*grado de salida*” es lo mismo, pero con las aristas que salen del mismo.

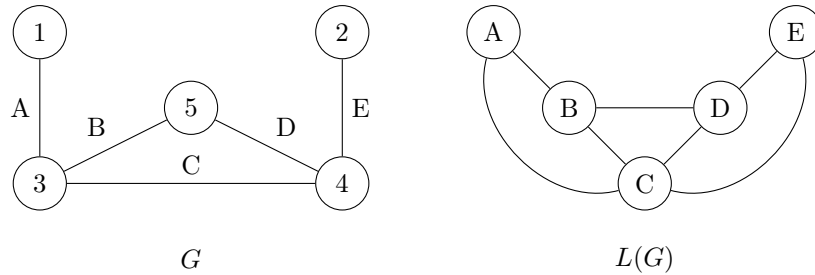
Un “*camino orientado*” en un digrafo es una sucesión de ejes  $e_1, e_2, \dots, e_k$  ta que el primer elemento del par  $e_i$  coincide con el segundo de  $e_{i-1}$  y el segundo elemento de  $e_i$  coincide con el primero de  $e_{i+1}$ , para todo  $i = 2, \dots, k - 1$ .

Un “*circuito orientado*” en un digrafo es un camino orientado que comienza y termina en el mismo vértice. Un digrafo se dice “*fuertemente conexo*” si para todo par de vértices  $v, w$  existe un camino orientado de  $v$  a  $w$  y otro de  $w$  a  $v$ .

### 3.5. Grafo de Líneas

Dado un grafo  $G$ , su “*grafo de líneas*”  $L(G)$  es un grafo que cumple que:

1. Cada vértice de  $L(G)$  representa un eje de  $G$
2. Dos vértices de  $L(G)$  son adyacentes  $\iff$  sus ejes correspondientes en  $G$  son incidentes



## 4. Árboles

Un *árbol* es un grafo conexo sin circuitos simples. Dado  $G = (V, X)$  las siguientes afirmaciones son equivalentes:

1.  $G$  es un árbol.
2.  $G$  es un grafo sin circuitos simples, pero si se le agrega una arista  $e$  a  $G$  tenemos un grafo con exactamente un circuito simple, y ese circuito contiene a  $e$ .
3. Existe exactamente un camino simple entre todo par de nodos.
4.  $G$  es conexo, pero si se le quita una cualquier arista queda desconexo.

Sea  $G = (V, X)$  un grafo conexo y  $e \in X$ ,  $G - e$  es conexo si y sólo si  $e$  pertenece a un circuito simple de  $G$ . Dentro de un árbol, definimos como *hoja* a un nodo de grado 1. Todo árbol no trivial tiene al menos 2 hojas.

### 4.1. Propiedades de los árboles

- Si  $G$  un árbol, entonces  $m = n - 1$ .
- Si  $G = (V, X)$  con  $c$  componentes conexas, entonces  $m \geq n - c$ .
- Si  $G = (V, X)$  con  $c$  componentes conexas y sin circuitos simples, entonces  $m = n - c$ .

### 4.2. Árboles enraizados

En un árbol no dirigido podemos definir un nodo cualquiera como raíz. El *nivel* de un vértice de un árbol es la distancia de ese vértice a la raíz. La *altura*  $h$  de un árbol es la longitud desde la raíz hasta el vértice más lejano.

Un árbol se dice "*m-ario*" (con  $m \geq 2$ ) si todos sus vértices salvo las hojas y la raíz tienen grado a lo sumo  $m + 1$  y la raíz a lo sumo  $m$ .

Un árbol se dice "*balanceado*" si todas sus hojas están a nivel  $h$  o  $h - 1$ . Si todas están a nivel  $h$ , se dice que es "*balanceado completo*".

Los nodos "*internos*" de un árbol son aquellos que no son hojas ni raíz.

- Un árbol  $m$ -ario de altura  $h$  tiene a lo sumo  $m^h$  hojas.
- Un árbol  $m$ -ario de altura  $h \geq 1$  y balanceado completo tiene exactamente  $m^h$  hojas.
- Un árbol  $m$ -ario con  $l$  hojas tiene  $h \geq \lceil \log_m l \rceil$ .
- Un árbol exactamente  $m$ -ario balanceado con  $l$  hojas tiene  $h = \lceil \log_m l \rceil$ .

### 4.3. Recorrido de árboles

#### 4.3.1. DFS

“*Depth First Search*” es un algoritmo de búsqueda en grafos. Lo que hace es arrancar desde un vértice  $n$  cualquiera y fijarse si dicho vértice es el que estamos buscando. Si no lo es, marcarlo como visitado, pedir el listado de todos sus vecinos y llamarse recursivamente (*DFSr*) en cada uno de ellos.

---

```
1: function DFS( $G = (V, E)$ )
2:   for all  $v \in V$  do
3:      $visitV_v \leftarrow false$ 
4:   for all  $e \in E$  do
5:      $visitE_e \leftarrow false$ 
6:   for all  $v \in V$  do
7:     if  $visitV_v = false$  then
8:       DFS( $v$ )
```

---

---

```
1: function DFS( $v$ )
2:    $visited_v \leftarrow true$ 
3:   for all  $e = (w, u) \in E$  where  $w = v$  and  $visitE_e = false$  do
4:     if  $visitV_w = false$  then
5:       DFS( $w$ )
```

---

Cuando el algoritmo llega a una hoja hace backtracking hasta el último vértice revisado con hijos todavía sin revisar.

Como todos los algoritmos de grafos, la complejidad depende de ciertos factores de la implementación. En la función  $DFS(v)$  estoy recorriendo todos los ejes de  $E$ , buscando los que salen de  $v$ . Si hago este llamado para cada vértice  $v$ , la complejidad del algoritmo me va a quedar  $O(n * m)$ .

No obstante, no queremos recorrer ejes que ya sabemos que ya revisamos. Es verdad que, como mucho, podríamos iterar  $m$  veces en la línea 3 :, pero en una buena implementación de DFS no tendríamos que nunca pasar por el mismo eje dos veces.

Es decir, por cada vértice, solo vamos a ver sus ejes no visitados. Si queremos revisar todos los vértices y todos los ejes de cada uno, nos termina quedando una complejidad de  $O(n + m)$ .

#### 4.3.2. BFS

“*Breadth First Search*” es muy similar a DFS pero, en lugar de realizar backtracking poniendo todos los vecinos de cada vértice en un stack, pone a todos los vecinos de cada vértice en una queue.

```
void BFS(Graph &g, int n) {
    queue<int> q;
    bool checked[g.n];
    memset(checked, 0, sizeof(checked));
    q.push(0);

    while(!q.empty()) {
        int t = q.front();
        q.pop();

        if(checked[t]) continue;
        if(t == n) {
            cout << "found!" << endl;
            return;
        }

        checked[t] = 1;
        vector<int> neigh = g.getAdyacentes(t);
        for(int i = 0; i < neigh.size(); ++i) {
            if(!checked[neigh[i]]) {
                q.push(neigh[i]);
            }
        }
    }
}
```

Al igual que con DFS, BFS sólo tiene sentido en grafos conexos y tiene complejidad  $O(n)$ .

#### 4.4. Arbol generador

Dado un grafo conexo  $G$ , un “*árbol generador*” de  $G$  es un subgrafo de  $G$  que es un árbol y tiene el mismo conjunto de vértices. Si los ejes del grafo tienen peso, entonces cada árbol generador va a tener una determinada “*longitud*”, que se define como la suma de todos los pesos de sus ejes. Cuando dicha longitud es mínima, decimos que tenemos un “*árbol generador mínimo*”.





#### 4.4.1. Algoritmo de Kruskal

El algoritmo de Kruskal es un algoritmo greedy que lo que hace es ir seleccionando las aristas que van a conformar el AGM una por una, arrancando de la de menor peso y subiendo progresivamente. Si el grafo no es conexo, forma un “bosque generador mínimo”.

---

**Algorithm 1** Algoritmo de Kruskal

---

```
1:  $AGM \leftarrow \emptyset$ 
2: for all  $v \in V$  do
3:    $\text{makeDisjointSet}(V)$ 
4:  $\text{sortByWeightASC}(E)$ 
5: for all  $(v_1, v_2) \in E$  do
6:   if  $\text{find}(v_1) \neq \text{find}(v_2)$  then
7:      $AGM \leftarrow AGM \cup (v_1, v_2)$ 
8:      $\text{union}(v_1, v_2)$ 
return  $AGM$ 
```

---

#### Explicación

- 1: Empieza declarando un  $AGM$  vacío. Este va a ser un conjunto de ejes, que es lo que va a devolver el algoritmo.
- 2: y 3: Crea un conjunto disjunto para cada vértice de  $V$ .
- 4: Ordena ascendentemente a todos los ejes por peso.
- 5: Empieza a iterar por todos los ejes  $(v_1, v_2)$  de  $E$ 
  - 6: Para cada eje, se fija que  $v_1$  y  $v_2$  estén en diferentes conjuntos del disjoint set previamente creado.
  - 7: Si no lo están, agrega a  $(v_1, v_2)$  al  $AGM$ .
  - 8: Y los une adentro del disjoint set.

#### Complejidad

En cada iteración del algoritmo (5:) necesitamos poder obtener el eje no todavía insertado en el AGM en  $O(1)$ . Esto se puede lograr fácilmente ordenando todos los  $m$  ejes por peso ascendentemente usando algún sort por comparación, lo cual se puede hacer en  $O(m \log m)$  (4:).

Los pasos 6: y 8: (find y union) se pueden hacer en  $O(\log m)$ . Como los estoy haciendo en cada iteración del algoritmo, y como voy a iterar  $n-1$  veces, la complejidad del mismo termina siendo  $O(n (2 \log m)) = O(n \log m)$ .

#### 4.4.2. Algoritmo de Prim

Al igual que el algoritmo de Kruskal, el algoritmo de Prim nos permite encontrar un árbol generador mínimo para un grafo. Este es un pseudocódigo del mismo:

---

**Algorithm 2** Algoritmo de Prim

---

```
1:  $AGM \leftarrow \emptyset$ 
2:  $V \leftarrow \{v\}$ 
3: while  $|V| \neq n$  do
4:    $(v_1, v_2) \leftarrow \text{buscarEje}(V)$ 
5:    $AGM \leftarrow AGM \cup \{(v_1, v_2)\}$ 
6:    $V \leftarrow V \cup \{v_2\}$ 
return  $AGM$ 
```

---

#### Explicación

A diferencia del anterior, el algoritmo de Prim hace crecer al AGM desde un vértice raíz arbitrario, y agrega un eje nuevo en cada iteración. El algoritmo termina cuando todos los vértices están en el AGM. El pseudocódigo funciona de la siguiente manera:

- 1: Declaramos el AGM que vamos a devolver como vacío. Al igual que en el algoritmo de Kruskal, el AGM devuelto es un conjunto de ejes.
- 2: Creamos un conjunto  $V$  de vértices que sólo tiene a un vértice arbitrario  $v$ , que puede ser cualquiera.
- 3: Vamos a iterar hasta que  $V$  tenga a todos los vértices del grafo.
- 4: Obtenemos un eje  $(v_1, v_2)$  usando una función llamada `buscarEje`. El eje devuelto tiene que cumplir las siguientes propiedades:
  - $v_1 \in V$
  - $v_2 \notin V$
  - El peso de  $(v_1, v_2)$  tiene que ser mínimo.
- 5: Agregamos a  $(v_1, v_2)$  a nuestro AGM.
- 6: Agregamos a  $v_2$  a  $V$ .

#### Complejidad

Lógicamente toda la complejidad del algoritmo está en la función `buscarEje`, la cual hace que el algoritmo sea greedy. La complejidad del mismo depende de la estructura utilizada para implementar la función `buscarEje`.

Si utilizo una cola de prioridad implementada sobre un min-heap binario esta función cuesta  $O(\log m)$ . Como este paso se va a tener que realizar  $n$  veces ya que está adentro del while, el costo total del algoritmo es de  $O(n \log m)$ .

## 5. Camino Mínimo

Sea  $G = (V, E)$  un grafo y  $l : E \rightarrow \mathbb{R}$  una función de peso para los ejes de  $G$ , definimos como el “*peso*” de un camino  $C$  entre dos nodos  $v$  y  $w$  como la suma de los pesos de los ejes del camino:

$$l(C) = \sum_{e \in C} l(e) \quad (1)$$

Un “*camino mínimo*”  $C^0$  entre  $v$  y  $w$  es un camino tal que  $l(C^0) = \min\{l(C), \forall C \text{ camino entre } v \text{ y } w\}$ . Esto significa que no necesariamente tiene que ser único. Dado un grafo  $G$ , se pueden definir 3 variantes de problemas sobre caminos mínimos:

1. **Unico origen - único destino:** determinar un camino mínimo entre dos vértices  $v$  y  $w$ .
2. **Unico origen - múltiples destinos:** determinar un camino mínimo desde un vértice  $v$  al resto de los vértices de  $G$
3. **Múltiples orígenes - múltiples destinos:** Determinar un camino mínimo entre todo par de vértices de  $G$ .

Si el grafo  $G$  no contiene ciclos con peso negativo (o contiene alguno pero no es alcanzable desde  $v$ ) entonces el problema sigue estando bien definido, aunque algunos caminos pueden tener longitud negativa. Sin embargo, si  $G$  tiene algún ciclo con peso negativo alcanzable desde  $v$ , el concepto de camino mínimo deja de estar bien definido.

Un camino mínimo no puede contener circuitos. También es importante notar que un camino mínimo exhibe la propiedad de subestructura óptima, ya que dados dos puntos  $v'$  y  $w'$  que están adentro del camino mínimo entre  $v$  y  $w$ , el subcamino entre estos dos puntos también es un camino mínimo entre ambos.

### 5.1. BFS

Cuando el grafo no tiene pesos.

## 5.2. Algoritmo de Dijkstra (1 a n)

Dado  $G = (V, E)$  y grafo,  $l : E \rightarrow \mathbb{R}$  una función que asigna a cada eje un cierto peso y  $v$  un vértice de  $G$ , calcular los caminos mínimos desde  $v$  al resto de los vértices. El algoritmo de Dijkstra asume que los pesos de los ejes son positivos.

---

```
1: function DIJKSTRA( $G = (V, E)$ )
2:    $prev[A] \leftarrow -1$ 
3:    $dist[A] \leftarrow 0$ 
4:    $pq.add(< dist[A], A >)$ 
5:   for all  $v \in V - \{A\}$  do
6:      $prev[v] \leftarrow -1$ 
7:      $dist[v] \leftarrow \infty$ 
8:      $pq.add(< dist[v], v >)$ 
9:   while  $!pq.empty()$  do
10:     $t \leftarrow pq.pop()$ 
11:    for all  $w \in t.second.neigh()$  do
12:       $alt \leftarrow dist[t.second] + length(t.second, w)$ 
13:      if  $alt < dist[w]$  then
14:         $prev[w] \leftarrow t.second$ 
15:         $dist[w] \leftarrow alt$ 
16:         $pq.decreaseKey(< dist[w], u >)$ 
17:   return  $prev$ 
```

---

### Analisis

Este algoritmo toma un grafo  $G$  y devuelve un array de vértices llamado  $prev$ , que consiste en los caminos mínimos en el grafo desde el vértice  $A$ .

- 2: Seteamos al vértice previo de  $A$  como  $-1$ , ya que es el vértice por el que comienzan todos los caminos mínimos.
- 3: Lógicamente la distancia mínima de  $A$  hacia  $A$  es  $0$ . Esto se asume porque  $G$  no tiene ciclos negativos.
- 4: Declaramos como  $pq$  a una cola de prioridad. Acá vamos a guardar tuplas  $< d, w >$ , donde  $d$  es la distancia mínima desde el vértice  $A$  hasta el vértice  $w$ . Hacemos esto para poder obtener el vértice de menor distancia a  $A$  en  $O(\log n)$ . Acá hay información redundante (las distancias mínimas ya se están guardando en  $dist$ ) pero es beneficiosa para la complejidad del algoritmo.
- 5 a 8: Luego vamos a empezar a iterar todos los vértices de  $G$  seteando la información necesaria para cada uno. Esto significa setear su vértice previo en  $-1$  y su distancia mínima hacia  $A$  en infinito. Este proceso es  $O(n)$ .
- 9: Ahora comienza el ciclo principal del algoritmo. En cada iteración del **while** vamos a visitar el vértice de menor distancia hacia  $A$  que encontremos en  $pq$ . El algoritmo termina cuando  $pq$  está vacía, es decir,

visitamos a todos los vértices. Vale destacar que el primer vértice que visitamos es  $A$ .

- 10: Guardamos en una tupla  $t$  al vértice que sacamos de  $pq$ . Hay que tener en cuenta que en  $t.first$  vamos a tener la distancia mínima del vértice que está en  $t.second$  hasta  $A$ .
- 11: Vamos a iterar a todos los vecinos  $w$  de  $t.second$ .
  - 12: Definimos a la variable  $alt$  como la longitud que tendría el camino hacia  $w$  si pasáramos antes por  $t.second$ .
  - 13: ¿Es esta nueva distancia menor a la distancia que teníamos calculada antes?. Es decir, ¿si pasamos por  $t.second$  para llegar desde  $A$  a  $w$ , llegamos más rápido que si no pasamos?. Si la respuesta es **sí**, entonces mejoramos la distancia previamente calculada para  $w$  y tenemos que actualizar todo:
  - 14: Entonces para llegar a  $w$  vamos a querer pasar por  $t.second$ .
  - 15: Actualizamos la distancia mínima de  $A$  hasta  $w$ .
  - 16: Hacemos lo mismo, pero en  $pq$ .

## Complejidad

La complejidad del algoritmo está dada por el ciclo principal del mismo que empieza en la línea 9. Vamos a tener que iterar tantas veces como elementos haya en  $pq$ , donde inicialmente van a haber  $n$  elementos.

En cada iteración vamos a tener que obtener el vértice más cercano a  $A$  (10:), lo cual puede hacerse en  $O(\log n)$  usando una cola de prioridad basada en un min-heap.

### 5.3. Algoritmo de Bellman-Ford (1 a n)

Es un poco más lento que el algoritmo de Dijkstra, pero admite ejes de pesos negativos. La idea general es similar a la del algoritmo de Dijkstra, pero un poco más simple.

Mientras Dijkstra obtiene el vértice no visitado de menor peso en cada iteración, Bellman-Ford simplemente relaja todos los ejes  $n - 1$  veces. En cada iteración el número de vértices con caminos mínimos bien calculados se va incrementando, hasta que ya no se pueden mejorar.

---

```
1: function BELLMANFORD( $G = (V, E)$ )
2:   for all  $v \in V, v \neq A$  do
3:      $dist[v] \leftarrow \infty$ 
4:      $prev[v] \leftarrow -1$ 
5:    $dist[A] \leftarrow 0$ 
6:   for  $i \leftarrow 1$  hasta  $n - 1$  do
7:     for all  $(u, v)$  con peso  $w \in E$  do
8:       if  $dist[u] + w < dist[v]$  then
9:          $dist[v] \leftarrow dist[u] + w$ 
10:         $prev[v] \leftarrow u$ 
11:   for all  $(u, v)$  con peso  $w \in E$  do           ▷ ¿Hay ciclos negativos?  $O(E)$ 
12:     if  $dist[u] + w < dist[v]$  then
13:       return null
14:   return prev
```

---

#### Analisis

- 2 a 4:** Seteamos, para todos los vértices que no son  $A$ , la distancia mínima en  $\infty$  y a su vértice previo como  $-1$ .
- 5:** Seteamos la distancia mínima hacia  $A$  (el vértice inicial) en  $0$ .
- 6:** Comenzamos la iteración principal del algoritmo. Vamos a ciclar, como mucho  $n - 1$  veces.
- 7:** Vamos a iterar por todos los ejes del grafo.
- 8:** Para cada eje  $(u, v)$ , chequeamos si nos conviene pasar por  $u$  para llegar hasta  $v$ .
- 9 y 10:** Si este es el caso, actualizamos los valores.
- 11 a 13:** Chequeamos si tenemos un ciclo negativo. Si este es el caso, el problema deja de estar bien definido y devolvemos *null*.

#### Complejidad

En este caso es bastante evidente que la complejidad es  $O(n * m)$ , ya que vamos a ciclar a lo sumo  $n - 1$  veces y, en cada uno de estos ciclos, vamos a relajar, como mucho, los  $m$  ejes de  $G$ .

## 5.4. Algoritmo de Floyd-Warshall (n a m)

Dado un grafo dirigido  $G = (V, E)$  sin pesos negativos, el algoritmo de Floyd sirve para calcular el camino mínimo entre cada par de vértices de  $G$  en tiempo  $O(n^3)$ .

Supongamos que los vértices de  $V$  están numerados de 1 a  $n$ , y definamos a la función  $\text{caminoMin}(i, j, k)$  que devuelve el camino mínimo desde el vértice  $i$  hasta el  $j$ , usando los vértices que están en  $\{1, 2, \dots, k\}$  como posibles puntos intermedios del camino.

Ahora bien, a partir de esa función, nuestro objetivo es el de encontrar el camino mínimo de  $i$  hasta  $j$  usando sólo los vértices que están en  $\{1, 2, \dots, k, k+1\}$ . Para cada par de vértices  $i, j$  el camino mínimo podría ser:

- Un camino que sólo usa los vértices del conjunto  $\{1, 2, \dots, k\}$
- Un camino que va de  $i$  a  $k+1$ , y de  $k+1$  hasta  $j$

Pero nosotros sabemos que el mejor camino de  $i$  a  $j$  que sólo usa los vértices del conjunto  $\{1, 2, \dots, k\}$  está dado por la función  $\text{caminoMin}(i, j, k)$ , y es claro que si hubiera un mejor camino de  $i$  a  $j$  que pasara por  $k+1$ , entonces la longitud de dicho camino debería ser la concatenación del camino mínimo de  $i$  a  $k+1$  (usando los vértices en  $\{1, 2, \dots, k\}$ ) y el camino mínimo de  $k+1$  a  $j$  (también usando los vértices en  $\{1, 2, \dots, k\}$ )

Si  $w(i, j)$  es el peso del eje entre los vértices  $i$  y  $j$ , podemos definir a la función  $\text{caminoMin}(i, j, k+1)$  con la siguiente recursividad:

$$\begin{aligned} \text{caminoMin}(i, j, 0) &= w(i, j) \\ \text{caminoMin}(i, j, k+1) &= \min \left\{ \begin{array}{l} \text{caminoMin}(i, j, k), \\ \text{caminoMin}(i, k+1, k) + \text{caminoMin}(k+1, j, k) \end{array} \right\} \end{aligned}$$

---

```

1: function FLOYDWARSHALL( $G = (V, E)$ )
2:    $dist \leftarrow \infty$ 
3:    $next \leftarrow -1$ 
4:   for all  $v \in V$  do
5:      $dist[v][v] \leftarrow 0$ 
6:   for all  $e = (v, w) \in E$  do
7:      $dist[v][w] \leftarrow w(e)$ 
8:      $next[v][w] \leftarrow v$ 
9:   for all  $k$  from 1 to  $n$  do
10:    for all  $i$  from 1 to  $n$  do
11:      for all  $j$  from 1 to  $n$  do
12:         $alt \leftarrow dist[i][k] + dist[k][j]$ 
13:        if  $alt < dist[i][j]$  then
14:           $dist[i][j] \leftarrow alt$ 
15:           $next[i][j] \leftarrow next[i][k]$ 
16:   return  $next$ 

```

---

Nos construimos una matriz  $D$  que contiene las longitudes de los caminos mínimos entre cada par de vértices, y el algoritmo la inicializa con los valores de  $L$ .

El algoritmo empieza a iterar. Después de la iteración  $k$ ,  $D$  contiene las longitudes mínimas que sólo usan los vértices en el conjunto  $\{1, 2, \dots, k\}$  como vértices intermedios. Después de  $n$  iteraciones  $D$  contiene las longitudes mínimas que usan cualquiera de los  $n$  vértices, lo cual es el resultado que queríamos.

En la iteración  $k$ , el algoritmo debe chequear para cada par de vértices  $i$  y  $j$  si existe un camino desde  $i$  hasta  $j$  que pasa por el vértice  $k$  que tiene longitud menor que la que ya teníamos calculada antes, la cual correspondía al camino de  $i$  a  $j$  que sólo usaba los vértices en  $\{1, 2, \dots, k - 1\}$ .



### 5.5. Algoritmo de Dantzig (n a m)

Al finalizar la iteración  $k - 1$ , el algoritmo de Dantzig generó una matriz  $k \times k$  de caminos mínimos en el subgrafo inducido por los vértices  $\{v_1, \dots, v_k\}$ . Asumimos que el grafo es orientado. El algoritmo detecta si hay circuitos de longitud negativa.

---

```
1: function DANTZIG( $G = (V, E)$ )
2:   for all  $k$  from 1 to  $n - 1$  do
3:     for all  $k$  from 1 to  $n - 1$  do
4:        $L_{i,k+1} \leftarrow \min_{1 \leq j \leq k} \{L_{i,j} + L_{j,k+1}\}$ 
5:        $L_{k+1,i} \leftarrow \min_{1 \leq j \leq k} \{L_{k+1,j} + L_{j,i}\}$ 
6:        $t \leftarrow \min_{1 \leq j \leq k} \{L_{k+1,i} + L_{i,k+1}\}$ 
7:       if  $t < 0$  then
8:         return null
9:       for all  $i$  from 1 to  $k$  do
10:        for all  $j$  from 1 to  $k$  do
11:           $L_{i,j} \leftarrow \min\{L_{i,j}, L_{i,k+1} + L_{k+1,j}\}$ 
12:   return  $L$ 
```

---

## 6. Grafos Eulerianos y Hamiltonianos

### 6.1. Grafos Eulerianos

Un circuito  $C$  en un grafo  $G$  es un “*circuito euleriano*” si  $C$  pasa por todos los ejes de  $G$  una y sólo una vez. Un grafo es euleriano si tiene un circuito euleriano. Un circuito euleriano puede pasar varias veces por el mismo vértice. Un grafo conexo es euleriano  $\Leftrightarrow$  todos sus nodos son de grado par. Podemos encontrar un circuito euleriano utilizando el algoritmo de Hierholzer en  $O(m)$ :

---

```
1: function HIERHOLZER( $G = (V, E)$ )
2:    $v \leftarrow A$ 
3:    $Z \leftarrow \text{construirCiclo}(v)$ 
4:   while  $\exists e \in E, e \notin Z$  do
5:      $w \leftarrow \text{encontrarEje}()$ 
6:      $D \leftarrow \text{construirCicloQueNoPasePor}(Z, w)$ 
7:      $Z \leftarrow Z \cup D$ 
8:   return  $Z$ 
```

---

#### Análisis

- 2: Elegimos un nodo random de  $G$ . Puede ser  $A$ , o puede ser cualquier otro.
- 3: La función  $\text{construirCiclo}(v)$  nos devuelve un ciclo euleriano que pasa por  $v$ .
- 4: Quizás el ciclo construido en 3: es un ciclo euleriano (por ejemplo, si el grafo es un polígono). Si no lo es, entonces vamos a tener que iterar hasta que dejen de haber ejes que no están en  $Z$ .
  - 5: La función  $\text{encontrarEje}()$  devuelve un eje  $w$  tal que uno de los vértices de  $e$  está en  $Z$  y el otro no.
  - 6: La función  $\text{construirCicloQueNoPasePor}(Z, w)$  devuelve un ciclo  $D$ , cuyos ejes no están en  $Z$ .
  - 7: Ahora tenemos dos ciclos eulerianos que comparten como único vértice a  $w$ . Al unirlos, es trivial ver que todavía voy a seguir teniendo un ciclo euleriano.

#### 6.1.1. Problema del Cartero Chino

El “*Problema del Cartero Chino*” es el problema de encontrar, dado un grafo  $G$  con pesos en sus ejes, un circuito euleriano de longitud mínima que pase por cada eje al menos una vez. Si  $G$  es euleriano, la solución del problema es su circuito euleriano. Hay algoritmos polinomiales para resolverlo.

## 6.2. Grafos Hamiltonianos

Un circuito en un grafo  $G$  es un “*circuito hamiltoniano*” si pasa por cada vértice de  $G$  sólo una vez. Un grafo es hamiltoniano cuando tiene dicho circuito. No se conocen caracterizaciones para grafos hamiltonianos ni tampoco un algoritmo polinomial para decidir si un grafo es hamiltoniano o no.

**Teorema (condición necesaria):** Sea  $G$  un grafo conexo, si existe  $W \subset V$  tal que  $G - W$  tiene  $c$  componentes conexas con  $c > |W|$ , entonces  $G$  no es hamiltoniano.

**Teorema de Dirac (condición suficiente):** Sea  $G$  un grafo con  $n \geq 3$  tal que para todo  $v \in V$  se verifica que  $d(v) \geq n/2$ , entonces  $G$  es hamiltoniano.

### 6.2.1. The Travelling Salesman Problem

Dado un grafo completo  $G = (V, E)$  con longitudes asignadas a sus ejes, queremos determinar un circuito hamiltoniano de longitud mínima. No se conocen algoritmos polinomiales para resolver este problema, pero existen varias heurísticas para atacarlo:

#### Heurística del vecino más cercano

Esta heurística greedy es bastante rápida, pero no siempre obtiene el circuito hamiltoniano de menor peso. Básicamente lo que hace es partir de un vértice cualquiera e ir moviéndose por el camino de menor peso que encuentre, y así sucesivamente hasta haber recorrido todos los vértices.

---

```
1: function NNA( $G = (V, E)$ )
2:    $v \leftarrow getRandomNode()$ 
3:    $orden[v] \leftarrow 0$ 
4:    $S \leftarrow \{v\}$ 
5:    $i \leftarrow 0$ 
6:   while  $S \neq V$  do
7:      $i++$ 
8:      $w \leftarrow aristaMasBarata()$ 
9:      $orden[w] \leftarrow i$ 
10:     $S \leftarrow S \cup \{w\}$ 
11:     $v \leftarrow w$ 
12:  return  $orden$ 
```

---

La función *aristaMasBarata()* devuelve siempre una arista  $(v, w)$  de peso mínimo, con  $w \notin S$ . Este algoritmo puede no siempre devolver un ciclo, y si lo devuelve, puede no ser el de peso mínimo. Su complejidad es  $O(n^2)$ . También existe la posibilidad de correr este algoritmo una vez en cada vértice, obteniendo un circuito hamiltoniano en cada ejecución, y quedándose con el de peso mínimo.

### Heurística de inserción

Las heurísticas de inserción son las que ejecutan el siguiente procedimiento:

---

```
1:  $C \leftarrow \text{getCircuitoLong3}()$ 
2:  $S \leftarrow \{\text{vértices de } C\}$ 
3: while  $S \neq V$  do
4:   elegir un vértice  $v$  tal que  $v \notin S$ 
5:    $S \leftarrow S \cup \{v\}$ 
6:   insertar  $v$  en  $C$ 
7: return  $C$ 
```

---

Lógicamente la particularidad de cada heurística de inserción depende de cómo se está eligiendo el vértice y cómo se lo está insertando en  $C$ . Hay diferentes maneras de elegir el vértice:

- Elegir  $v$  al azar.
- Elegir el  $v$  más lejano a un vértice que ya está en  $C$ .
- Elegir el  $v$  más cercano a un vértice que ya está en  $C$ .
- Elegir el  $v$  más barato, es decir, que hace crecer menos la longitud de  $C$ .

Para insertar el nodo  $v$  elegido, lo que hacemos es:

1. Sea  $c(v_i, v_j)$  el costo de la longitud del eje  $(v_i, v_j)$ .
2. Elegimos dos vértices  $v_i, v_{i+1}$  consecutivos en  $C$ , tales que  $C(v_i, v_{i+1})$  sea mínimo.
3. Insertamos a  $v$  entre  $v_i$  y  $v_{i+1}$ .

### Heurística del árbol generador

Se basa en el siguiente procedimiento:

1. Encontrar un AGM  $T$  de  $G$ .
2. Duplicar los ejes de  $T$ .
3. Armar un circuito euleriano  $E$  con los ejes de  $T$  y sus duplicados.
4. Recorrer  $E$  usando DFS y armar un circuito hamiltoniano de  $G$ .

## 7. Planaridad

Una “representación planar” de un grafo  $G$  es un conjunto de puntos en el plano que se corresponden con los vértices de  $G$ , unidos por curvas que se corresponden con los ejes de  $G$ , sin que estas se crucen entre sí. Si un grafo admite una representación planar, decimos que el mismo es planar.

Dada una representación planar de un grafo, llamamos “región” al conjunto de todos los puntos alcanzables desde un punto que no sea un vértice o un eje, sin atravesar ningún vértice o eje. Es por eso que toda representación planar de un grafo tiene exactamente una región de área infinita: la región exterior.

La “frontera” de una región es el circuito que rodea a la región, y el “grado” de la región es el número de ejes que tiene su frontera.

**Propiedad:**  $K_5$  es el grafo no-planar con el menor número de vértices, y  $K_{3,3}$  el que tiene el menor número de ejes.

**Propiedad:** Si un grafo  $G$  contiene un subgrafo  $G'$  no planar, entonces  $G$  tampoco es planar.

### 7.1. Subdivisiones y Homeomorfismo

Subdividir un eje  $e = (v, w)$  de un grafo  $G$  consiste en agregar un vértice nuevo  $u \notin V$  a  $G$  y reemplazar al eje  $e$  por dos ejes  $e' = (v, u)$  y  $e'' = (u, w)$ . Un grafo  $G'$  es una subdivisión de otro grafo  $G$  si  $G'$  se puede obtener mediante sucesivas operaciones de subdivisión sobre  $G$ . Dos grafos  $G$  y  $H$  se dicen “homeomorfos” si hay un isomorfismo entre una subdivisión de  $G$  y una de  $H$ .

**Propiedad 1:** Si  $G'$  es una subdivisión de  $G$ , entonces  $G$  es planar si y sólo si  $G'$  es planar.

**Propiedad 2:** La planaridad es invariante bajo homeomorfismo.

**Propiedad 3:** Si un grafo  $G$  tiene un subgrafo que es homeomorfo a un grafo no-planar, entonces  $G$  es no-planar.

#### 7.1.1. Teorema de Kuratowski

Un grafo es planar si y sólo si no contiene ningún subgrafo homeomorfo a  $K_5$  o a  $K_{3,3}$ .

### 7.2. Contracciones

La operación de “contracción” de un eje  $e = (v, w)$  consiste en eliminar el eje del grafo y considerar sus extremos como un sólo vértice  $u \notin V$ , quedando como ejes incidentes a  $u$  todos los ejes que eran incidentes a  $v$  y a  $w$ .

Un grafo  $G'$  es una contracción de otro grafo  $G$  si se puede obtener a partir de  $G$  por sucesivas operaciones de contracción. Se dice que “ $G$  es contraíble a  $G'$ ”.

### 7.2.1. Teorema de Whitney

$G$  es planar si y sólo si no contiene ningún subgrafo contraible a  $K_5$  o a  $K_{3,3}$ .

### 7.2.2. Teorema de Euler

Si  $G$  es un grafo conexo planar entonces cualquier representación planar de  $G$  determina  $r = m - n + 2$  regiones en el plano.

**Colorario 1:** Si  $G$  es conexo y planar con  $n \geq 3$ , entonces  $m \leq 3n - 6$ .

**Colorario 2:** Si  $G$  es conexo, planar y bipartito con  $n \geq 3$ , entonces  $m \leq 2n - 4$ .

## 7.3. Algoritmo de Demoucron

El siguiente algoritmo encuentra una representación planar si existe, y si  $G$  es no planar lo reconoce correctamente. El tiempo de ejecución es de  $O(n^2)$ , pero existen algoritmos que pueden determinar planaridad en tiempo lineal. Esto significa que el problema de decidir si un grafo es o no planar está en  $P$ . Pero antes de exponer el algoritmo hay que hacer las siguientes definiciones:

- Una **“parte  $p$  de  $G$  relativa a  $R$ ”** puede ser alguna de las siguientes cosas:
  - una componente conexa de  $G - R$  junto con los ejes que la conectan a todos los nodos de  $R$  (ejes colgantes).
  - un eje  $e = (v, w)$  de  $G - R$  con  $v, w \in R$ .
- Dada una parte  $p$  de  $G$  relativa a  $R$ , un **“nodo de contacto”** es un nodo  $R$  incidente a un eje colgante de  $p$ .
- Decimos que  $R$  **“es extensible a una representación planar de  $G$ ”** si se puede obtener una representación planar de  $G$  a partir de  $R$ .
- Una parte  $p$  es **“dibujable en una región  $f$  de  $R$ ”** si existe una extensión planar de  $R$  en la que  $p$  queda en  $f$ .
- Una parte  $p$  es **“potencialmente dibujable en una región  $f$  de  $R$ ”** si todo nodo de contacto de  $p$  pertenece a la frontera de  $f$ .
- Llamamos  $F(p)$  al conjunto de regiones de  $R$  donde  $p$  es potencialmente dibujable.

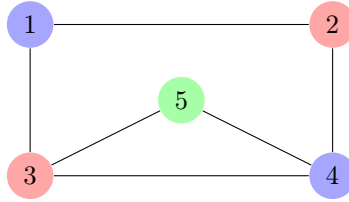
---

```
1: function DEMOUCRON( $G = (V, E)$ )
2:   return orden
```

---

## 8. Coloreo

Un “coloreo” de los vértices de un grafo  $G = (V, E)$  es una asignación  $f : V \rightarrow C$ , tal que  $f(v) \neq f(w)$  para todo  $(v, w) \in E$ . Para todo entero positivo  $k$ , un “ $k$ -coloreo” de  $G$  es un coloreo que usa exactamente  $k$  colores.



Un grafo se dice “ $k$ -coloreable” si existe un  $k$ -coloreo de  $G$ . De esta manera definimos al número cromático de  $G$ ,  $\chi(G)$ , como el menor número de colores necesarios para colorear los vértices de  $G$ . Un grafo se dice “ $k$ -cromático” si  $\chi(G) = k$ . Notar que el grafo de la figura es 3-cromático.

Algunos ejemplos:

- $\chi(K_n) = n$
- Si  $G$  es planar  $\Rightarrow \chi(G) = 4^2$
- Si  $G$  es bipartito  $\Rightarrow \chi(G) = 2$
- Si  $C_{2k}$  es un circuito simple par  $\Rightarrow \chi(C_{2k}) = 2$
- Si  $C_{2k+1}$  es un circuito simple impar  $\Rightarrow \chi(C_{2k+1}) = 3$

### 8.1. Cotas para $\chi(G)$

- Si  $\Delta(G)$  es el grado máximo de  $G$ , entonces  $\chi(G) \leq \Delta(G) + 1$ .
- Si  $G$  es un grafo conexo que no es un circuito impar ni un grafo completo, entonces  $\chi(G) \leq \Delta(G)^3$ .
- Si  $H$  es un subgrafo de  $G$ , entonces  $\chi(H) \leq \chi(G)$ .
- $\omega(G) \leq \chi(G)^4$ .

---

<sup>2</sup>Teorema de los Cuatro Colores

<sup>3</sup>Teorema de Brooks

<sup>4</sup>Una “clique” es un subgrafo completo maximal de un grafo. El “número clique” de un grafo es el máximo número de vértices de una clique de  $G$ , y se denota por  $\omega(G)$ .

## 8.2. Algoritmos para coloreo de grafos

Dado un grafo  $G$  de  $n$  vértices y un entero  $k$ , el problema de decisión que consiste en determinar si existe un  $k$ -coloreo para  $G$  es NP-completo, y su complejidad es  $O(2^n * n)$ .

Dado un grafo  $G$  de  $n$  vértices y un entero  $k$ , el problema de decisión que consiste en determinar si  $\chi(G) = k$  es NP-Hard.

- **Heurística secuencial (S):** Dado un orden  $v_1, v_2, \dots, v_n$  de  $V$ , asignar en el paso  $i$  el menor color posible a  $v_i$ . Es decir, el color más chico que no está siendo usado por los vecinos de  $v_i$  (greedy). Puede devolver resultados bastante malos en algunos grafos.
- **Algoritmo secuencial con backtracking (exacto):** La idea es ir asignando colores uno por uno a los diferentes vértices, arrancando desde el vértice 0. Antes de asignar un color, miramos los colores de los vecinos de dicho vértice. Si encontramos un color apropiado, se lo ponemos al vértice en cuestión. Si no es posible encontrar un color apropiado, volvemos hacia atrás y probamos con otro color en algún vértice anterior.

## 8.3. Grafos perfectos

Un grafo  $G$  es “perfecto” si  $\chi(H) = \omega(H)$  para todo subgrafo inducido  $H$  de  $G$ . Existe un algoritmo polinomial para determinar  $\chi(G)$  si  $G$  es perfecto <sup>5</sup>.

Un grafo es perfecto si y sólo si su complemento es perfecto. Un grafo es perfecto si y sólo si no tiene ciclos impares ni complementos de ciclos impares como subgrafos inducidos.

## 8.4. Coloreo de ejes

Un “coloreo de ejes” de un grafo  $G$  es una asignación de colores a los mismos en el cual dos ejes que tienen un vértice en común no pueden tener el mismo color.

El “índice cromático”  $\chi'(G)$  de un grafo  $G$  es el menor número de colores con el que se pueden colorear los ejes de  $G$ . Para todo grafo  $G$  se cumple que  $\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1$  <sup>6</sup>.

---

<sup>5</sup>Teorema de Grottschel, Lovász y Schrijver

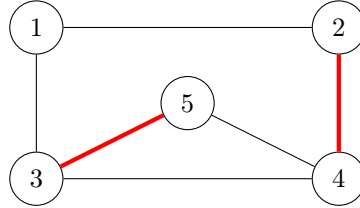
<sup>6</sup>Teorema de Vizing



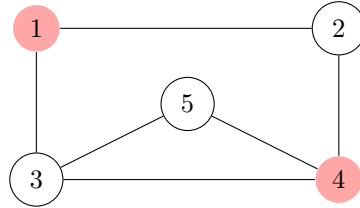
## 9. Matchings y Conjuntos Independientes

Dado un grafo  $G = (V, E)$  tenemos las siguientes definiciones:

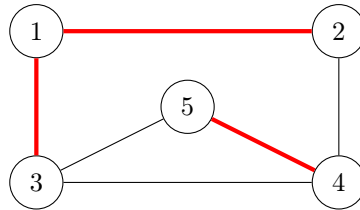
- Un “*matching*” es un conjunto  $M \subseteq E$  de ejes de  $G$  tal que para todo vértice  $v \in V$ ,  $v$  es incidente **a lo sumo** de un eje  $e \in M$ . Se dice entonces que  $v$  está “*saturado*” por  $M$ .



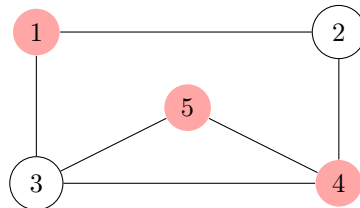
- Un “*conjunto independiente*” es un conjunto de  $I \subseteq V$  tal que para todo eje  $e \in E$ ,  $e$  es incidente **a lo sumo** a un vértice  $v$  de  $I$ .



- Un “*recubrimiento de los vértices*” de  $G$  es un conjunto  $R_e$  de ejes tales que para todo  $v \in V$ ,  $v$  es incidente a **al menos** un eje  $e \in R_e$ :



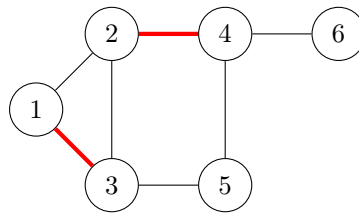
- Un “*recubrimiento de los ejes*” de  $G$  es un conjunto  $R_v$  de vértices tal que para todo  $e \in E$ ,  $e$  es incidente a **al menos** un vértice  $v \in R_v$ .



$S \subseteq V$  es un conjunto independiente si y sólo si  $V - S$  es un recubrimiento de ejes.

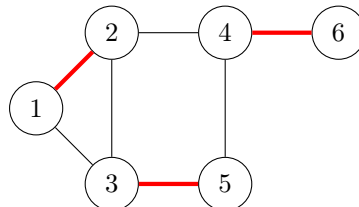
### 9.1. Matching maximal

Un “*matching maximal*”  $M$  en un grafo  $G$  es un matching que cumple la propiedad de que si se le agrega un eje cualquiera  $M$  deja de ser un matching. Es decir, pasa a existir un vértice  $v$  que es incidente a dos ejes de  $M$ , el que le agregué y uno que ya estaba.



### 9.2. Matching máximo

Un “*matching máximo*” es un matching que contiene la mayor cantidad posible de ejes. El “*número de matching*”  $\nu(G)$  de un grafo  $G$  es el tamaño de su matching máximo. Todo matching máximo es maximal, pero no todo matching maximal es máximo, como se puede ver al comparar los dibujos de cada uno.



El problema de encontrar un matching máximo en un grafo arbitrario está bien resuelto en  $P$ , con el algoritmo Blossom que tiene un costo de  $O(V^2E)$ . Como un matching máximo también es maximal, es posible encontrar en el matching maximal más grande en tiempo polinomial.

No obstante, todavía no hay ningún algoritmo polinomial para encontrar el matching maximal mínimo, esto es, un matching maximal que contenga la menor cantidad de ejes.

### 9.3. Caminos

Dado un matching  $M$  en  $G$ , un “camino alternado” en  $G$  con respecto a  $M$ , es un camino simple donde se alternan ejes que están en  $M$  con ejes que no lo están. Para el graáfico anterior, un ejemplo de camino alternado es 6, 4, 2, 1, 3, 5.

Dado un matching  $M$  en  $G$ , un “camino de aumento” en  $G$  con respecto a  $M$  es un camino alternado entre vértices no saturados por  $M$ .

Sean  $M_0$  y  $M_1$  dos matchings en  $G$ , y sea  $G' = (V, E')$  con  $E' = (M_0 - M_1) \cup (M_1 - M_0)$ , entonces las componentes conexas de  $G'$  son de alguno de los siguientes tipos:

- Vértice aislado
- Circuito simple con ejes alternadamente entre  $M_0$  y  $M_1$ .
- Camino simple con ejes alternadamente entre  $M_0$  y  $M_1$ .

**Teorema:**  $M$  es un matching máximo de  $G$  si y sólo si no existe un camino de aumento en  $G$  con respecto a  $M$ .

**Teorema:** Dado un grafo  $G$  sin vértices aislados, si  $M$  es un matching máximo de  $G$  y  $R_e$  un recubrimiento mínimo de los vértices de  $G$ , entonces  $|M| + |R_e| = n$ .

**Teorema:** Dado un grafo  $G$ , si  $I$  es un conjunto independiente máximo de  $G$  y  $R_n$  un recubrimiento mínimo de los ejes de  $G$ , entonces  $|I| + |R_n| = n$ .

## 10. Flujo en Redes

Una “red”  $N = (V, E)$  es un grafo orientado conexo que tiene dos nodos distinguidos: una “fuente”  $s$ , con un grado de salida positivo y un “sumidero”  $t$  con un grado de entrada positivo.

Una “función de capacidades en la red” es una función  $c : E \rightarrow \mathbb{R}^{\geq 0}$

Un “flujo factible” en una red  $N = (V, E)$  con función de capacidad  $c$  es una función  $f : E \rightarrow \mathbb{R}^{\geq 0}$  que verifica:

- $0 \leq f(e) \leq c(e)$  para todo eje  $e \in E$ .
- La “Ley de conservación de Flujo”:

$$\sum_{e \in \text{in}(v)} f(e) = \sum_{e \in \text{out}(v)} f(e)$$

para todo vértice  $v \in V - \{s, t\}$ , donde

$$\text{in}(v) = \{e \in E, e = (w \rightarrow v), w \in V\}$$

$$\text{out}(v) = \{e \in E, e = (v \rightarrow w), w \in V\}$$

Es decir, la cantidad de flujo que entra a un vértice es la misma que la que sale.

El “valor del flujo” es  $F = 3$

### 10.1. Cortes

### 10.2. Camino de Aumento

### 10.3. Algoritmo de Ford-Fulkerson

### 10.4. Algoritmo de Edmonds-Karp

## 11. Teoría de Complejidad

Denominamos como “*problema de decisión*” a los problemas cuya respuesta es “*sí*” o “*no*”. El objetivo de esta teoría es el de clasificar a este tipo de problemas según su complejidad.

### 11.1. La clase P

La clase P es donde viven los problemas cuyas soluciones tienen complejidad “*polinomial*”. Decimos que un algoritmo corre en tiempo polinomial si su complejidad es  $O(n^k)$ , donde  $n$  denota el número de bits de entrada en el algoritmo.

Un algoritmo “*eficiente*” es un algoritmo de complejidad polinomial, y decimos que un problema está “*bien resuelto*” si se conocen algoritmos eficientes para resolverlo. Por ejemplo, el problema  $\Pi$  encontrar un vértice  $v$  en un grafo de  $n$  vértices puede resolverse en tiempo  $O(n)$  usando DFS, por lo que  $\Pi \in P$ .

### 11.2. La clase NP

Un problema de decisión  $\Pi$  pertenece a la clase  $NP$  (no-determinístico polinomial) si dada cualquier instancia del mismo para las cuales la respuesta es *sí* y un “*certificado*”, podemos chequear que dicho certificado es correcto usando un algoritmo de tiempo polinomial.

Veamos por ejemplo el problema  $\Pi$  que consiste en: “*dado un array  $A$  de  $n$  enteros, existe una subsecuencia del mismo que suma 0?*”. Una instancia del problema puede ser el array  $A = \{-7, -3, -2, 8, 5\}$  y un certificado puede ser  $c = \{-3, -2, 5\}$ . Como es trivial ver que la suma de todos los elementos de  $c$  es 0 en tiempo polinomial, entonces tenemos que  $\Pi \in NP$ .

Es trivial darse cuenta que  $P$  está contenido en  $NP$ , pero la gran incógnita en la teoría de la complejidad es determinar si  $P = NP$ . Es decir, si para cada problema en  $NP$  existe una solución polinomial.

#### 11.2.1. Reducciones Polinomiales

Una “*reducción polinomial*” de un problema de decisión  $\Pi_1$  a uno  $\Pi_2$  es una función polinomial que transforma una instancia  $i_1$  de  $\Pi_1$  en una instancia  $i_2$  de  $\Pi_2$ , tal que  $i_1$  tiene respuesta “*sí*” en  $\Pi_1$  si y sólo si  $i_2$  tiene respuesta “*sí*” en  $\Pi_2$ .

El problema de decisión  $\Pi_1$  se reduce polinomialmente a otro problema de decisión  $\Pi_2$  ( $\Pi_1 \leq_p \Pi_2$ ) si existe una transformación polinomial de  $\Pi_1$  a  $\Pi_2$ .

### 11.3. La clase NP-Completo

“NP-Completo” es una subclase de NP, pero con la particularidad de que todos los problemas en NP pueden ser reducibles en tiempo polinomial a cualquier problema de NP-Completo. Formalmente, un problema de decisión  $\Pi$  es NP-Completo si:

- $\Pi \in \text{NP}$
- $\forall \Pi' \in \text{NP}, \Pi' \leq_p \Pi$ .

Es importante notar que la operación de reducción es transitiva. Si  $\Pi$  es un problema de decisión, podemos probar que  $\Pi \in \text{NP-Completo}$  encontrando otro problema  $\Pi'$  que ya sabemos que es NP-Completo y demostrando que:

- $\Pi \in \text{NP}$
- $\Pi' \leq_p \Pi$

### 11.4. El problema SAT

El problema SAT (“*problema de satisfacibilidad booleana*”) consiste en, dada una fórmula booleana, decidir si existe una manera de reemplazarla con valores de verdadero o falso de manera que dicha fórmula devuelva verdadero.

#### 11.4.1. Teorema de Cook

El Teorema de Cook dice que SAT es NP-Completo. Es decir, que todo problema en NP puede ser reducido en tiempo polinomial al problema de decidir si una fórmula booleana es satisfacible.

Una consecuencia importante de este problema es que si existe un algoritmo de tiempo polinomial para resolver SAT, entonces existe un algoritmo polinomial para todos los problemas de NP, lo que demostraría que  $P = NP$ .

### 11.5. La clase NP-Intermedio

Esta es la clase de problemas de NP que no están ni en P ni en NP-Completo. El Teorema de Ladner dice que, si  $P \neq NP$ , entonces NP-I no es vacío. Es decir,  $P = NP$  si y sólo si NP-I es vacío.

Hasta ahora no se conoce ningún problema de NP-I. De existir uno, implicaría que  $P \neq NP$ , lo cual no se sabe, pero se sospecha ampliamente. Un ejemplo de un problema que se sospecha que podría estar en NP-I es el de la factorización de números naturales.

## 11.6. La clase NP-Hard

Similarmente a la clase NP-Completo, un problema de decisión  $\Pi$  es “*NP-hard*” si todo problema de NP se puede reducir polinomialmente a  $\Pi$ , **pero  $\Pi$  no tiene que necesariamente estar en NP**. Es decir, la clase NP-Completo está contenida dentro de la clase NP-Hard.

Todos los problemas NP-Completo son NP-Hard, pero no todos los problemas NP-Hard son NP-Completo, por ejemplo, el Halting Problem.

## 11.7. Algoritmos Pseudopolinomiales

Decimos que un algoritmo corre en tiempo “*pseudopolynomial*” si la complejidad del mismo es del orden polinomial en el valor numérico de la entrada, en lugar de la cantidad de bits necesaria para representarla.

Un ejemplo de esto es el algoritmo que determina si un número cualquiera  $n$  es o no primo, y para esto se fija si dicho número tiene algún divisor en el intervalo  $[2, \dots, n]$ .

Este algoritmo va a iterar  $n$  veces, y en cada iteración va a calcular  $n \bmod i$ . Si hacer la operación de módulo costara  $O(n^3)$ , el algoritmo corre en  $O(n^4)$ .

## 12. Preguntas

- Camino mínimo Cómo justificar la complejidad del algoritmo de Dijkstra?
- Camino mínimo Cómo se aplica el principio del optimalidad en el algoritmo de Floyd?
- Camino mínimo Si usando un Fibonacci heap puedo hacer que el algoritmo de Dijkstra resuelva el problema de camino mínimo en  $O(m + n \log n)$ , que es mejor que  $O(n^2)$ , no es más rápido aplicar  $n$  veces el algoritmo de Dijkstra en lugar de Floyd? Me terminaría quedando una complejidad de  $O(n(m + n \log n)) = O(n * m + n^2 \log n)$ , que es mejor que  $O(n^3)$ .
- Camino mínimo Cómo funciona el algoritmo de Dantzig?
- Planaridad Definimos a la **“parte  $p$  de  $G$  relativa a  $R$ ”** como un eje  $e = (u, v)$  de  $G - R$  con  $u, v \in R$ , pero entonces ni  $u$  ni  $v$  van a estar en  $G - R$ , por lo que  $e = (u, v)$  no estaría bien definido.
- Planaridad Si  $R$  es una región, qué significa hacer  $G - R$ ?
- Complejidad Si un problema  $\Pi$  es NP-Hard entonces eso significa que todo problema de  $NP$  puede ser reducible polinomialmente a  $\Pi$ . Como  $P$  está en  $NP$ , cualquier problema de  $P$  es reducible polinomialmente al halting problem. Eso significa que, dado un árbol en un grafo conexo, decidir si dicho árbol es un AGM es reducible polinomialmente al halting problem.
- Complejidad Encontrar un problema de NP que no esté en P ni tampoco en NP-Completo.