Algoritmos y Estructuras de Datos III
Resumen

amildie

1. Complejidad Computacional **3**

2. Programación Dinámica **4**
2.1. Problemas clásicos . . . 5
2.1.1. Subsecuencia continua de suma máxima . . . 5
2.1.2. Subsecuencia no-continua estrictamente creciente más larga 6
2.1.3. Longitud de la subsecuencia no-continua común más larga 7
2.1.4. El problema de la mochila . . . 9
2.1.5. Subconjunto de suma 0 . . . 11

3. Grafos **13**
3.1. Caminos y distancia . . . 13
3.2. Subgrafos y biparticidad . . . 14
3.3. Isomorfismo . . . 14
3.4. Pesos . . . 14
3.5. Digrafos . . . 15
3.6. Grafo de Líneas . . . 15

4. Arboles **16**
4.1. Propiedades de los árboles . . . 16
4.2. Arboles enraizados . . . 16
4.3. Recorrido de árboles . . . 17
4.3.1. DFS . . . 17
4.3.2. BFS . . . 18
4.4. Arbol generador . . . 19
4.4.1. Algoritmo de Kruskal . . . 20
4.4.2. Algoritmo de Prim . . . 21

5. Camino Mínimo **22**
5.1. BFS . . . 22
5.2. Algoritmo de Dijkstra (1 a n) . . . 23
5.3. Algoritmo de Bellman-Ford (1 a n) . . . 25
5.4. Algoritmo de Floyd-Warshall (n a m) . . . 26
5.5. Algoritmo de Dantzig (n a m) . . . 27

6. Grafos Eulerianos y Hamiltonianos **28**
6.1. Grafos Eulerianos . . . 28
6.1.1. Problema del Cartero Chino . . . 28
6.2. Grafos Hamiltonianos . . . 29

6.2.1. The Travelling Salesman Problem 29

7. Planaridad **31**
7.0.2. Teorema de Euler 31
7.1. Subdivisiones y Homeomorfismo 31
7.1.1. Teorema de Kuratowski 31
7.2. Contracciones 32
7.2.1. Teorema de Whitney 32
7.3. Algoritmo de Demoucron 32
7.3.1. Definiciones 32
7.3.2. Pseudocódigo 34

8. Coloreo **35**
8.1. Cotas para $\chi(G)$ 35
8.2. Algoritmos para coloreo de grafos 36
8.3. Grafos perfectos 36
8.4. Coloreo de ejes 36

9. Matchings y Conjuntos Independientes **37**
9.1. Matching maximal 38
9.2. Matching máximo 38
9.3. Caminos 39
9.4. Propiedades 39

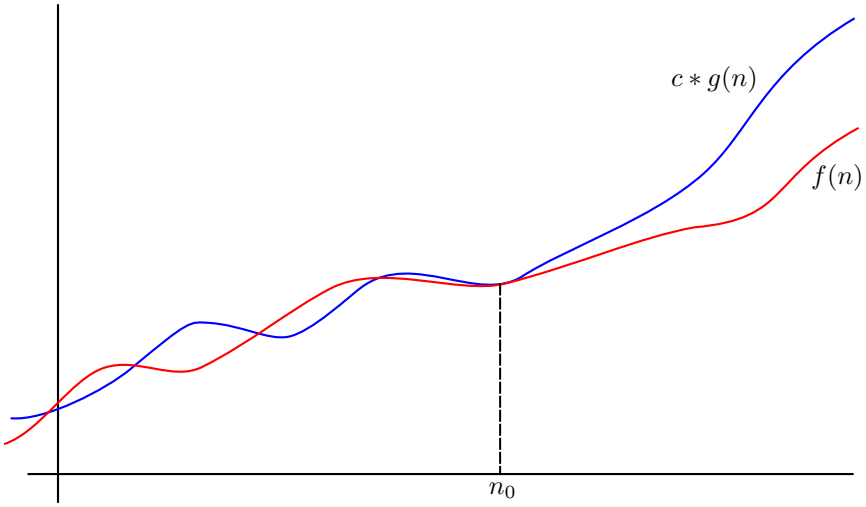
10. Flujo en Redes **41**
10.1. Cortes 42
10.1.1. Capacidad de un corte 43
10.2. Red Residual 43
10.3. Camino de Aumento 44
10.4. Algoritmo de Ford-Fulkerson 44
10.4.1. Algoritmo de Edmonds-Karp 45

11. Teoría de Complejidad **46**
11.1. La clase P 46
11.2. La clase NP 46
11.2.1. Reducciones Polinomiales 46
11.3. La clase NP-Completo 47
11.4. El problema SAT 47
11.4.1. Teorema de Cook 47
11.5. La clase NP-Intermedio 47
11.6. La clase NP-Hard 48
11.7. Algoritmos Pseudopolinomiales 48

1. Complejidad Computacional

Dada una función $g(n)$, denominamos como $\mathcal{O}(g(n))$ 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 $\mathcal{O}(g(n))$ es un conjunto de funciones $f(n)$ que, a partir de cierto valor n_0, van a estar acotadas superiormente por $c * g(n)$. Más formalmente:

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

Por ejemplo, si $f(n) = n^2+n+1$ y $g(n) = n^2$ podemos decir que $f(n) \in \mathcal{O}(g(n))$, 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

La *"programación dinámica"* es una técnica de desarrollo de algoritmos que es útil para encarar problemas que exhiben simultáneamente dos características particualres:

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 mínimo. Supongamos dos puntos A y B, y un camino c que es el más corto entre ellos. Para cualquier par de puntos A' y B' dentro de c, el camino más corto c' entre ellos está necesariamente contenido adentro de c.

2. **Soluciones Sobrepuestas:** Al igual que la subestructura óptima, un problema tiene esta característica cuando sub 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"* y la manera en la que la voy a implementar a lo largo de los ejemplos es mediante un array llamado **S**.

Es intuitivo notar que los algoritmos de programación dinámica que implementan memoization están haciendo un trade off entre complejidad temporal y complejidad espacial, disminuyendo la primera al aumentar la segunda.

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

2.1. Problemas clásicos

2.1.1. 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 una subsecuencia nueva. Es decir:

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

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;
}
```

2.1.2. 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.

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

```
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;
}
```

2.1.3. Longitud de la subsecuencia no-continua común más larga

Dadas dos strings A y B, encontrar la longitud de la sub secuencia no necesariamente continua más larga que tengan en común.

Por ejemplo, dadas $A = ABC$ y $B = ACB$, la respuésta del problema es 2, ya que tanto AB como AC tienen longitud máxima.

Para resolver este problema vamos a definir el concepto de *"prefijo"*. Un prefijo de una secuencia s es una subsecuencia continua de s que llega hasta el caracter i. Usando a A como ejemplo, $A_1 = AB$. La idea de la solución es computar la subsecuencia no-continua común más larga entre A y B para todo par de prefijos posibles.

Para esto vamos a definir una matriz de enteros c, donde $c[i][j]$ contiene el valor de la longitud de la subsecuencia común más larga entre A_i y B_j. Si A tiene longitud n y B tiene longitud m, el valor de la matriz que resuelve nuestro problema es $c[n][m]$.

Es evidente que para computar el valor de $c[i][j]$ vamos a necesitar saber lo valores de $c[i'][j']$, para todo $i' \leq i$ y $j' \leq j$, pero lógicamente sin tener $i = i'$ ni $j = j'$ al mismo tiempo.

Primero vamos a observar el caso base: $c[i][0] = c[0][j] = 0$. Es decir, si alguna de las dos secuencias está vacía la longitud final de la LCS es 0. Formalmente:

```
1: for 0 ≤ i ≤ m do
2:     c[i][0] ← 0
3: for 0 ≤ j ≤ n do
4:     c[0][j] ← 0
```

Luego vamos a calcular el resto de los valores. Al querer calcular $c[i][j]$ pueden pasar dos cosas:

- **Que x_i y y_i sean iguales:** Por ejemplo, $X_i = ABCA$ y $Y_j = DACA$. Como ambas terminan en A, la LCS tiene que también terminar en A. Como A es parte de la LCS, podemos encontrar la LCS final sacando a A de ambas secuencias y buscando la LCS de X_{i-1} (ABC) y Y_{j-1} (DAC), la cual es AC. Luego, agregar a A al final nos deja con ACA, que es efectivamente la LCS. Luego, $c[i][j] = c[i-1][j-1] + 1$.
- **Que x_i y y_i no sean iguales:** En este caso x_i y x_j no puden ser ambos parte de la LCS (ya que deberían ser el último caracter de la misma). Esto significa que pueden pasar dos cosas:
  - x_i **no es parte de la LCS:** lo que significa que la LCS entre X_i y Y_j es la LCS entre X_{i-1} y Y_j, lo cual ya tenemos calculado en $c[i-1][j]$.
  - y_j **no es parte de la LCS:** lo que significa que la LCS entre X_i y Y_j es la LCS entre X_i y Y_{j-1}, lo cual ya tenemos calculado en $c[i][j-1]$.

  Lógicamente la LCS va a se la que tenga mayor longitud en ambos casos, por lo que $c[i][j] = max\{c[i-1][j], c[i][j-1]\}$.

Escribiendo la anterior dinámica más formalmente:

$$c[i][j] = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ c[i-1][j-1] + 1 & \text{si } i, j > 0 \text{ y } x_i = y_j \\ max\{c[i-1][j], c[i][j-1]\} & \text{si } i, j > 0 \text{ y } x_i \neq y_j \end{cases}$$

Lo que se traduce en el siguiente algoritmo:

```
1: for 0 ≤ i ≤ m do
2:     c[i][0] ← 0
3: for 0 ≤ j ≤ n do
4:     c[0][j] ← 0
5: for 1 ≤ i ≤ m do
6:     for 1 ≤ j ≤ n do
7:         if x_i = y_i then
8:             c[i][j] ← c[i − 1][j − 1] + 1
9:         else
10:            c[i][j] ← max(c[i − 1][j], c[i][j − 1])
11: return c[m][n]
```

2.1.4. El problema de la mochila

El *"problema de la mochila"* es un problema cásico de optimización combinatoria. Se trata de, dado un conjunto de items, 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 items 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:

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

La siguiente solución utilizar programación dinámica y corre en tiempo pseudo-polinomial. Asumimos que los pesos $w[1], w[2], ..., w[n]$ 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.

2.1.5. Subconjunto de suma 0

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

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

Por ejemplo, si $X = \{1, -3, 2, 4\}$ la respuesta es *"sí"* ya que $1 - 3 + 2 = 0$.

Este problema lo podemos resolver de manera similar al de la mochila mediante memoization.

Vamos a empezar definiendo una matriz booleana Q, donde $Q[i][s]$ es verdadero o falso si hay un subconjunto no vacío de x_1 hasta x_i que suma s. Entonces para obtener una solución del problema tenemos que ver cuánto vale $Q[n][0]$.

Luego definimos a A como la suma de todos los valores negativos de X y (-3)a B como la suma de todos los positivos (7). Claramente $Q[i][s]$ es falso si $s < A$ o si $s > B$. Simplemente no se pueden alcanzar esos valores, por lo que los valores de i y de s que nos interesan dentro del problema son $1 \leq i \leq n$ y $A \leq s \leq B$. Vamos a completar Q, que inicialmente está vacía:

	-3	-2	-1	0	1	2	3	4	5	6	7
0											
1											
2											
3											

Luego seteamos el caso base: indexando únicamente hasta el primer elemento. Es decir, usando sólamente el 1. Si sólo podemos agarrar el primer elemento de X, entonces la única manera en la que podemos sumar s es cuando ese elemento es igual a s:

```
1: for A ≤ s ≤ B do
2:     if x_1 == s then
3:         Q[1][s] ← true
4:     else
5:         Q[1][s] ← false
```

Lo que nos deja la tabla de la siguiente manera:

	-3	-2	-1	0	1	2	3	4	5	6	7
0	F	F	F	F	**T**	F	F	F	F	F	F
1											
2											
3											

Ahora es cuando la cosa se pone más interesante. Tenemos que completar el resto de la tabla hasta llegar a $Q[3][0]$. Al ver $Q[i][s]$ pueden pasar cuatro cosas:

- **Que x_i sea igual a s.** estamos indexando hasta x_i, pero si este x_i es s, entonces encontramos un subconjunto que cumple lo pedido: $[x_i]$. Le ponemos **T**, obviamente.
- **Que ya hayamos encontrado un subconjunto anterior que lo cumple.** Si $Q[i-1][s]$ es true, significa que ya existe un subconjunto que suma s y que usa hasta el elemento con índice $i-1$. Le ponemos **T** a la $Q[i][s]$, ya que esto significa que *sí* hay un subconjunto que suma s y que usa los números disponibles hasta x_i.
- **Que usando los elementos hasta el anterior a x_i pueda sumar $s - x_i$.** Esto significa que agregando a x_i llego a sumar exactamente s, por lo que le ponemos **T**.
- **Que no se cumplan ninguna de las anteriores.** Le ponemos F.

Dicho más formalmente:

```
1:  for 1 ≤ i < n do
2:      for A ≤ s ≤ B do
3:          if x_i == s then
4:              Q[i][s] ← true
5:          else if Q[i − 1][s] == true then
6:              Q[i][s] ← true
7:          else if Q[i − 1][s − x_i] == true then
8:              Q[i][s] ← true
9:          else
10:             Q[i][s] ← false
```

Siguiendo esta dinámica la tabla nos queda:

	-3	-2	-1	0	1	2	3	4	5	6	7
0	F	F	F	F	**T**	F	F	F	F	F	F
1	F	F	F								
2											
3											

Efectivamente mostrando la solución que estábamos buscando.

También queda en evidencia que la complejidad del algoritmo es pseudopolinomial en el orden de $\mathcal{O}(s * N)$, donde s es la suma que queremos encontrar y N la cantidad de elementos de X.

3. Grafos

Un *"grafo"* $G = (V, E)$ 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.

Dados dos vértices $v, w \in V$ se dice que v y w son *"adyacentes"* si $\exists\ e \in E$ 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:

$$\sum_{i=1}^{|V|} d(v_i) = 2 * |E|$$

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 $e_1, e_2, ..., e_k$ tal que los extremos de e_i coinciden con uno de e_{i-1} y con uno de e_{i+1}, para todo $i \in 2, ..., k-1$.

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 biparticidad

Dado un grafo $G = (V, E)$, un *"subgrafo"* del mismo es un grafo $H = (V', E')$ que cumple que $V' \subseteq V$ y que $E' \subseteq E \cap (V' \times V')$. Si se cumple que para todo $u, v \in V', (u, v) \in E \iff (u, v) \in E'$ 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 $G = (V, E)$ 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 $G = (V, E)$ y $G' = (V', E')$ se dicen *"isomorfos"* si existe una función biyectiva $f : V \to V'$ tal que para todo $v, w \in V$:

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

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. Pesos

Sea $G = (V, E)$ un grafo, decimos que los ejes de G tienen *"peso"*, si existe una función de peso $l : E \to \mathbb{R}$ para los ejes de G.

3.5. Digrafos

Un *"grafo orientado"* (o *"digrafo"*) es un grafo $G = (V, E)$ en el cual los ejes de E tienen una dirección. Para cada vértice v se define a su *"grado de entrada"* ($d_{in}(v)$) como a la cantidad de ejes en E 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.

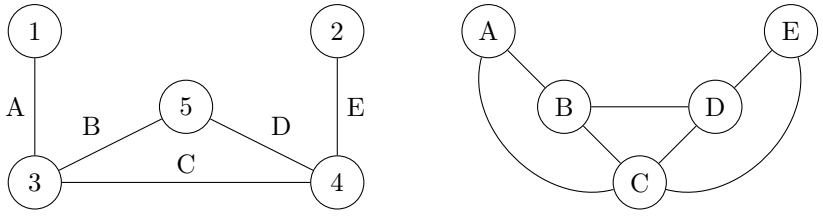
Un *"camino orientado"* es un digrafo es una sucesión de ejes $e_1, e_2, ..., e_k$ ta que el primer elemento del par e_i coincide con el segundo de e_{i-1} y el segundo elemento de e_i coincide con el primero de e_{i+1}, para todo $i = 2, ..., k-1$.

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.

3.6. 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



G $\qquad$ $L(G)$

4. Arboles

Un *árbol* es un grafo conexo sin circuitos simples. Dado $G = (V, E)$ 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 disconexo.

Sea $G = (V, E)$ un grafo conexo y $e \in E$, $G - e$ 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 $G = (V, E)$ es un árbol, valen las siguientes propiedades:

- $|E| = |V| - 1$.
- Si G tiene c componentes conexas, entonces $|E| \geq |V| - c$.
- Si G tiene c componentes conexas y no tiene circuitos simples, entonces $|E| = |V| - c$.

4.2. Arboles 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 raiz. La *altura* h de un árbol es la longitud desde la raiz hasta el vértice más lejano.

Un árbol se dice *"m-ario"* (con $m \geq 2$) si todos sus vértices salvo las hojas y la raiz tienen grado a lo sumo $m + 1$ y la raiz 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 ∈ V do
3:         visitV_v ← false
4:     for all e ∈ E do
5:         visitE_e ← false
6:     for all v ∈ V do
7:         if visitV_v = false then
8:             DFS(v)
```

```
1: function DFS(v)
2:     visited_v ← true
3:     for all e = (w, u) ∈ 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 $\mathcal{O}(|V| * |E|)$.

No obstante, no queremos recorrer ejes que ya sabemos que ya revisamos. Es verdad que, como mucho, podríamos iterar $|E|$ veces en la linea **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 temina quedando una complejidad de $\mathcal{O}(|V| + |E|)$.

4.3.2. BFS

"Bradth 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.

BFS arranca desde un vértice raíz y escanea el árbol nivel por nivel. Cuando termina un nivel avanza al siguiente, hasta que llega al último vértice del último nivel, donde termina el algoritmo:

```
1:  function BFS(v)
2:      for all v ∈ V do
3:          visited_v ← false
4:      q.push(v_0)
5:      while !q.empty() do
6:          w ← q.pop()
7:          if visited_w = false then
8:              for all e = (w, u) ∈ E do
9:                  if visited_u = false then
10:                     q.push(u)
11:             visited_w ← true
```

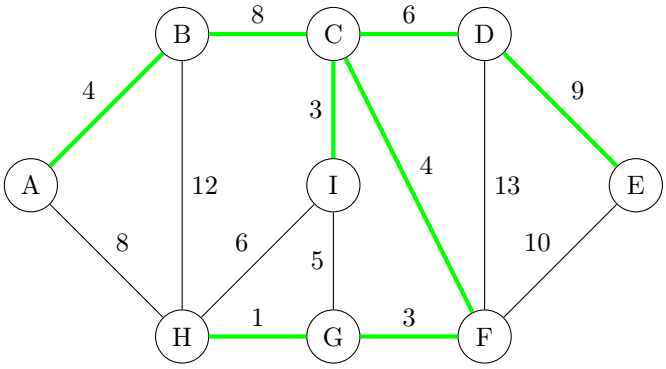
Inicialmente el algoritmo va a marcar a todos los vértices como no visitados, para luego encolar al primer vértice V_0 en q. Este proceso lleva $\mathcal{O}(|V|)$.

Luego va a ciclar hasta que q esté vacía. Va a haber una iteración del while principal por cada vértice de G. Para cada vértice que va desencolando, va a recorrer todos sus ejes incidentes, encolando a todos sus hijos.

Como no se repiten ejes, es sencillo ver que el ciclo de **8:** va iterar una vez por cada eje en E. Esto nos deja la complejidad total del algoritmo como $\mathcal{O}(|V| + |E|)$.

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 ← ∅
2: for all v ∈ V do
3:     makeDisjointSet(V)
4: sortByWeightASC(E)
5: for all (v1, v2) ∈ E do
6:     if find(v1) ≠ find(v2) then
7:         AGM ← AGM ∪ (v1, v2)
8:         union(v1, v2)
   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 $\mathcal{O}(1)$. Esto se puede lograr fácilmente ordenando todos los m ejes por peso ascendentemente usando algún sort por comparación, lo cual se puede hacer en $\mathcal{O}(|E| \log |E|)$ (4:).

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

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 ← ∅
2: V ← {v}
3: while |V| ≠ n do
4:     (v1, v2) ← buscarEje(V)
5:     AGM ← AGM ∪ {(v1, v2)}
6:     V ← V ∪ {v2}
   return AGM
```

Explicación

A diferencia del anterior, el algoritmo de Pimm hace crecer al AGM desde un vértice raiz 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 algorimto 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 greddy. La complejidad del mismo depende de las estructura utilizada para implementar la función $buscarEje$.

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

5. Camino Mínimo

Dado un grafo G, definimos como el *"peso"* de un camino C entre dos nodos v y w como la suma de los pesos de los pesos de los ejes del camino:

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

Un *"camino minimo"* C^0 entre v y w es un camino que cumple 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 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

Como vimos anteriormente, BFS es una estrategia para recorrer un grafo. No obstante, puede usarse para encontrar el camino mínimo en un grafo G que no tiene pesos asociados a sus ejes[1].

Si BFS arranca de un vértice v en particular la longitud del camino mínimo de v a un vértice destino w es la cantidad de vértices que los separan. Es decir, el nivel de w en el árbol enraizado en v.

Si modificamos el algoritmo para que vaya completando un array de predecesores podemos resolver el problema del camino mínimo de 1 a n en tiempo $\mathcal{O}(|V| + |E|)$.

[1] O en el que todos los pesos son idénticos

5.2. Algoritmo de Dijkstra (1 a n)

Dado $G = (V, E)$ y grafo, $l : E \to \mathbb{R}$ 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] ← −1
3:      dist[A] ← 0
4:      pq.add(< dist[A], A >)
5:      for all v ∈ V − {A} do
6:          prev[v] ← −1
7:          dist[v] ← ∞
8:          pq.add(< dist[v], v >)
9:      while !pq.empty() do
10:         t ← pq.pop()
11:         for all w ∈ t.second.neigh() do
12:             alt ← dist[t.second] + length(t.second, w)
13:             if alt < dist[w] then
14:                 prev[w] ← t.second
15:                 dist[w] ← 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 de forma greedy 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 depende del costo de las operaciones *pop*() y $decreaseKey()$.

La operación *pop*() va a extraer el vértice más cercano a A de pq hasta que pq no tenga mas vértices, por lo que van a haber $|V|$ llamados a *pop*(). A su vez, en el peor caso la operación $decreaseKey()$ se va a ejecutar una vez por cada eje de E, por lo que vamos a tener $|E|$ llamados a $decreaseKey()$.

Si llamamos T_p y T_d a los costos de ambas operaciones respectivamente, vamos a tener que la complejidad final para el algoritmo es de $\mathcal{O}(|V| * T_p + |E| * T_d)$.

Si implementamos a pq sobre un árbol del Fibonacci las operaciones *pop*() y $decreaseKey()$ van a tener costos en los órdenes de $\mathcal{O}(\log |V|)$ y $\mathcal{O}(1)$ respectivamente, lo que nos deja la complejidad concreta de la implementación en $\mathcal{O}(|V| * \log |V| + |E| * 1) = \mathcal{O}(|V| * \log |V| + |E|)$.

Pesos negativos

Este algoritmo no funciona bien cuando G tiene ejes con pesos negativos porque asume que los caminos sólo pueden volverse más pesados.

Cuando el algortimo se para en un vértice v y relaja a todos sus vecinos no va a volver a pararse en v nuevamente, ya mejoró las distancias a sus vecinos lo más posible pasando por v.

Cuando hay pesos negativos puede ser que aparezcan distancias previas más chicas que *"mejoran"* otras distancias previamente calculadas[2].

[2] stackoverflow.com/questions/6799172/negative-weights-using-dijkstras-algorithm

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 ∈ V, v ≠ A do
3:          dist[v] ← ∞
4:          prev[v] ← −1
5:      dist[A] ← 0
6:      for i ← 1 hasta n − 1 do
7:          for all (u, v) con peso w ∈ E do
8:              if dist[u] + w < dist[v] then
9:                  dist[v] ← dist[u] + w
10:                 prev[v] ← u
11:     for all (u, v) con peso w ∈ 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 ∞ 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), checkeamos si nos conviene pasar por u para llegar hasta v.

9 y 10: Si este es el caso, actualizamos los valores.

11 a 13: Checkeamos 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 $\mathcal{O}(|V| * |E|)$, ya que vamos a ciclar a lo sumo $|V|-1$ veces y, en cada uno de estos ciclos, vamos a relajar, como mucho, los $|E|$ 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 $\mathcal{O}(|V|^3)$.

Supongamos que los vértices de V están numerados de 1 a n, y definamos a la función $caminoMin(i, j, k)$ que devuelve el camino mínimo desde el vértice i hasta el j, usando los vértices que están en $\{1, 2, ..., k\}$ 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 sólamente los vértices que están en $\{1, 2, ..., k, k + 1\}$. Para cada par de vértices i, j el camino mínimo podría ser:

- Un camino que sólamente usa los vértices del conjunto $\{1, 2, ..., k\}$
- 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 $\{1, 2, ..., k\}$ está dado por la función $caminoMin(i, j, k)$, 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 $\{1, 2, ..., k\}$) y el camino mínimo de $k + 1$ a j (también usando los vértices en $\{1, 2, ..., k\}$)

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

$$caminoMin(i, j, 0) = w(i, j)$$

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

```
1:  function FloydWarshall(G = (V, E))
2:      dist ← ∞
3:      next ← −1
4:      for all v ∈ V do
5:          dist[v][v] ← 0
6:      for all e = (v, w) ∈ E do
7:          dist[v][w] ← w(e)
8:          next[v][w] ← 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 ← dist[i][k] + dist[k][j]
13:                 if alt < dist[i][j] then
14:                     dist[i][j] ← alt
15:                     next[i][j] ← 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 $\{1, 2, ..., k\}$ 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 checkear 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 $\{1, 2, ..., k-1\}$.

5.5. Algoritmo de Dantzig (n a m)

Al finalizar la iteración $k-1$, el algoritmo de Dantzig generó una matriz kxk de caminos mínimos en el subgrafo inducido por los vértices $\{v_1, ..., v_k\}$. 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} ← min_{1≤j≤k}{L_{i,j} + L_{j,k+1}}
5:              L_{k+1,i} ← min_{1≤j≤k}{L_{k+1,j} + L_{j,i}}
6:          t ← min_{1≤j≤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} ← 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 $\mathcal{O}(|E|)$:

```
1: function HIERHOLZER(G = (V, E))
2:     v ← A
3:     Z ← construirCiclo(v)
4:     while ∃ e ∈ E, e ∉ Z do
5:         w ← encontrarEje()
6:         D ← construirCicloQueNoPasePor(Z, w)
7:         Z ← Z ∪ D
8:     return Z
```

Análisis

2: Elegimos un nodo random de G. Puede ser A, o puede ser cualquier otro.

3: La función $construirCiclo(v)$ 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 $encontrarEje()$ devuelve un eje w tal que uno de los vértices de e está en Z y el otro no.
- **6:** La función $construirCicloQueNoPasePor(Z, w)$ 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 ← getRandomNode()
3:      orden[v] ← 0
4:      S ← {v}
5:      i ← 0
6:      while S ≠ V do
7:          i++
8:          w ← aristaMasBarata()
9:          orden[w] ← i
10:         S ← S ∪ {w}
11:         v ← 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 $\mathcal{O}(n^2)$. También existe la posibilidad de correr este algoritmo una vez en cada vértice, obteniendo un circuito hamiltoniano en cada ejecución, y quedándome con el de peso mínimo.

Heurística de inserción

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

```
1: C ← getCircuitoLong3()
2: S ← {vértices de C}
3: while S ≠ V do
4:     elegir un vértice v tal que v ∉ S
5:     S ← S ∪ {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_{33} el que tiene el menor número de ejes.

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

7.0.2. 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 $|V| \geq 3$
$\Rightarrow |E| \leq 3 * |V| - 6$.

Colorario 2: Si G es conexo, planar y bipartito con $|V| \geq 3$
$\Rightarrow |E| \leq 2 * |V| - 4$.

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 invariate 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 homemomorfo a K_5 o a K_{33}.

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 contraible 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_{33}.

7.3. Algoritmo de Demoucron

El siguiente algoritmo encuentra una represetación 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.

La idea del algoritmo es partir de una representación planar R de un subgrafo S de G, y empezar a expandirala iterativamente hasta obtener una representación planar de todo G o hasta concluir que dicha representación no existe.

7.3.1. Definiciones

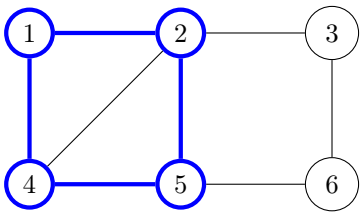
A partir de ahora van a empezar a aparecer unas operaciones que pierden completamente el hilo de coherencia de las diapositivas. Especialmente la operación $G - R$, que vendría a ser algo como *"restarle una representación planar a un grafo"*, lo cual adolece completamente de cualquier significado formal.

Quien haya llegado hasta acá va a tener que valerse más de la intuición para poder comprender las intenciones de quien hizo su mejor esfuerzo por armar las diapositivas.

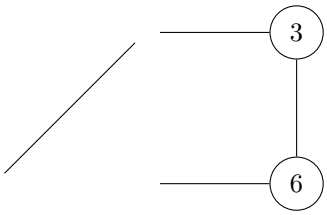
Tener especial cuidado con el concepto de *"vértice de contacto"*, el cual pertenece a una representación planar R a veces y a una parte p otras veces.

- Llamamos ***"parte p de G relativa a R"*** a:
  - Una componente conexa de $G - R$ junto con los ejes que la conectan a vértices de R (ejes colgantes). Notar que R es una representación de un subgrafo S de G.
  - Un eje $e = (v, w)$ de $G - R$, con $v, w \in R$. Es decir, un eje suelto sin vértices.

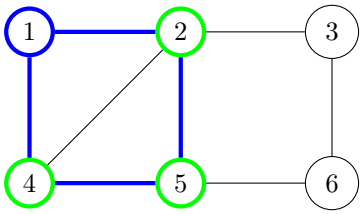
Es decir, una parte es *lo que queda* de un grafo al que le sacamos la representación planar de un subgrafo. Esta parte puede tener ejes colgantes que no están conectados con ningún vértice, e incluso una parte puede ser un eje suelto volando por ahí.



Por ejemplo, el grafico anterior usa como R al ciclo $1, 2, 5, 4$, por lo que las dos partes que nos quedan son:



- Dada una parte p de G relativa a R, llamamos ***"vértice de contacto"*** a un vértice de R incidente a un eje colgante de p:



- Decimos que la representación R es ***"extensible"*** a una representación planar de G si se puede obtener una representación planar de G a partir de R. *Extender* acá es el proceso de ir agregandole cosas a una representación planar.
- 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 vértice de contacto de p pertenece a la frontera de f.
- Llamamos $F(p)$ al conjunto de regiones de R donde p es potencialmente dibujable.

7.3.2. Pseudocódigo

```
1:  function DemoucRon(G = (V, E))
2:      R ← obtenerRepresentacionPlanarCiclo(G)
3:      while R no sea una representación planar de G do
4:          calculamos F(p) para cada parte p de G relativa a R
5:          if existe un p tal que F(p) = ∅ then
6:              return false
7:          else if existe un p tal que F(p) = {f} then
8:              elegir ese p y ese f
9:          else
10:             elegir cualquier p y un f ∈ F(p)
11:         Buscar camino o circuito q en p que empieza y termina en dos
12:         ejes colgantes diferentes si es posible. Caso contrario, q es
13:         el camino formado por el único eje colgante.
14:         R ← R ∪ q
15:     return R
```

Explicación

2: El algoritmo empieza definiendo una representación base R, que inicialmente puede ser de cualquier ciclo de G.

3: Luego va a ciclar hasta que R sea una representación de todo G o hasta determinar que G no es planar.

4: Empezamos calculanedo $F(p)$ para todas las partes p de G relativas a R. Esto significa que para cada parte $p_1, p_2, ..., p_k$, vamos a tener los conjuntos $F(p_1), F(p_2), ..., F(p_k)$; donde $F(p_i)$ es el conjunto de regiones en las que p_i es potencialmente dibujable.

5 y 6: Si existe un p_i tal que $F(p_i) = \emptyset$ significa que no existen regiones de R donde p_i sea potencialmente dibujable, por lo que G no es planar.

7 y 8: Si existe un p_i tal que $F(p_i)$ es igual a una única region f, solo puedo seleccionar esa región f para dibujar a p_i.

9 y 10: Si no se cumplió ninguna de las dos condiciones, elijo cualquier p_i y cualquier region $f_k \in F(p_i)$ en la que p_i sea potencialmente dibujable.

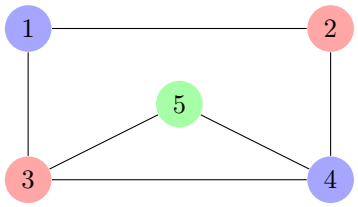
11 a 13: Voy a buscar un camino o circuito q en p_i que empieza y termina en dos ejes colgantes diferentes si es posible. Si no es posible, q es el camino formado por el único eje colgante.

Notar que no necesariamente tienen que estar todos los vértices de p_i en q. Pueden faltar algunos, lo que importa es que estamos agregando un circuito o un eje, por lo que se preserva la planaridad al extender R. Si faltan vértices, se van a resolver en las próximas iteraciones al recalcular las partes con el nuevo R.

14: Le agrego q a R.

8. Coloreo

Un *"coloreo"* de los vértices de un grafo $G = (V, E)$ es una asignación $f : V \to C$, 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 $\Rightarrow \chi(G) = 4$[3]
- 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 $\chi(G) \leq \Delta(G)$[4].
- Si H es un subgrafo de G, entonces $\chi(H) \leq \chi(G)$.
- $\omega(G) \leq \chi(G)$[5].

[3] Teorema de los Cuatro Colores
[4] Teorema de Brooks
[5] 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 $|V|$ 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 $\mathcal{O}(2^{|V|} * |V|)$.

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 $v_1, v_2, ..., v_{|V|}$ 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 asignado 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 [6].

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.

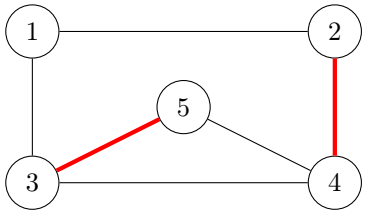
El *"índice cromático"* $\chi'(G)$ de un grafo G es el menor número de colores con el que se pueden colorear los ejes de G. Para todo grafo G se cumple que $\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1$ [7].

[6] Teorema de Grotschel, Lovász y Schrijver
[7] Teorema de Vizing

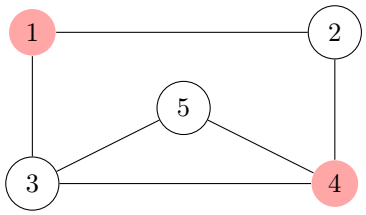
9. Matchings y Conjuntos Independientes

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

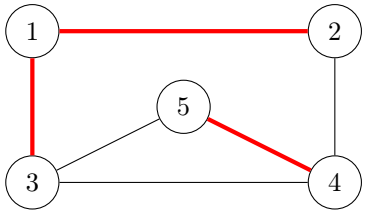
- Un *"matching"* es un conjunto $M \subseteq E$ de ejes de G tal que para todo vértice $v \in V$, v es incidente **a lo sumo** de un eje $e \in M$. Se dice entonces que v está *"saturado"* por M.



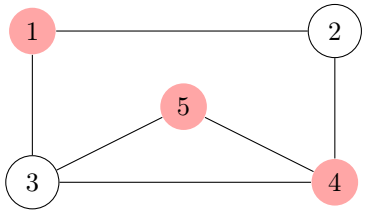
- Un *"conjunto independiente"* es un conjunto de $I \subseteq V$ tal que para todo eje $e \in E$, e es incidente **a lo sumo** a un vértice v de I.



- 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 $e \in R_e$:

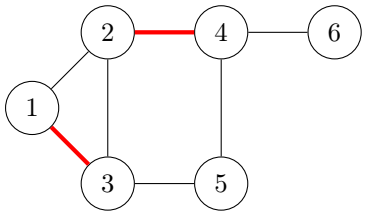


- Un *"recubrimiento de los ejes"* de G es un conjunto R_v de vértices tal que para todo $e \in E$, e es incidente a **al menos** un vértice $v \in R_v$.



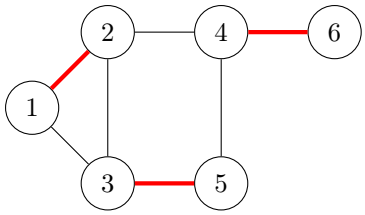
9.1. Matching maximal

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.



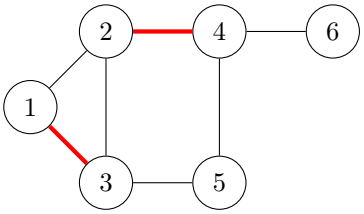
El problema de encontrar un matching máximo en un grafo arbitrario está bien resuelto en P, con el algoritmo Blossom que tiene un costo de $\mathcal{O}(|V|^2 * |E|)$. Como un matching máximo también es maximal, es posible encontrar en el matching maximal más grande en tiempo polinomial.

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, definimos:

- Un ***"camino alternado"*** en G con respecto a M, como 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 $6, 4, 2, 1, 3, 5$.
- Un ***"camino de aumento"*** en G con respecto a M, como un camino alternado entre vértices no saturados por M. Es decir, es un camino entre vértices que si y que no tocan ejes de M. Por ejemplo, el camino $3, 5, 4, 6$ en el siguiente matching:



9.4. Propiedades

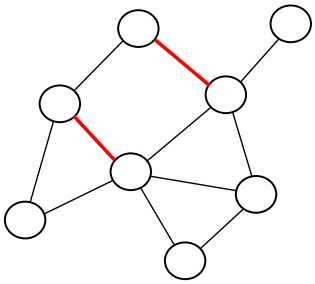
- M es un matching máximo de $G \Leftrightarrow$ no existe un camino de aumento en G con respecto a M.
- $S \subseteq V$ es un conjunto independiente $\Leftrightarrow V - S$ es un recubrimiento de ejes.

  Es decir, si S es un conjunto de vértices para los cuales todo eje es incidente con como mucho uno de esos vértices entonces, al sacarle ese conjunto a G, los vértices que nos quedan cumplen la propiedad de que saturan todos los ejes

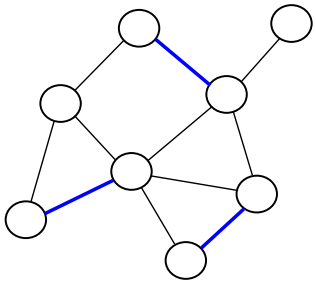
  Notar que un vértice solo cumple la propiedad de ser un conjunto independiente y al sacárselo a G, lógicamente va a pasar que todos los otros vértices saturan el resto de los ejes de G.

  También es trivial ver como vale que si S **no** es un conjunto independiente entonces $V - S$ no es un recubrimiento de ejes. Por ejemplo, si S tiene dos vértices v, w tales que son adyacentes, al sacárselos a G el eje (v, w) no va a estar saturado, por lo que $V - S$ no es un recubrimiento de ejes.
- 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 $G \Rightarrow |M| + |R_e| = n$.
- Dado un grafo G, si I es un conjunto independiente máximo de G y R_n un recubrimiento mínimo de los ejes de $G \Rightarrow |I| + |R_n| = n$.

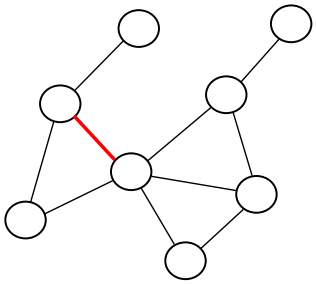
- Sean M_0 y M_1 dos matchings en G, y sea $G' = (V, E')$ con $E' = (M_0 - M_1) \cup (M_1 - M_0) \Rightarrow$ las componentes conexas de G' son de alguno de los siguientes tipos:
  - Vértice aislado
  - Circuito simple con ejes alternadamente entre M_0 y M_1.
  - Camino simple con ejes alternadamente entre M_0 y M_1.



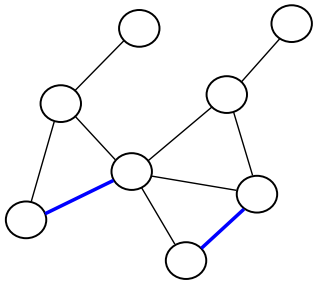
M_0



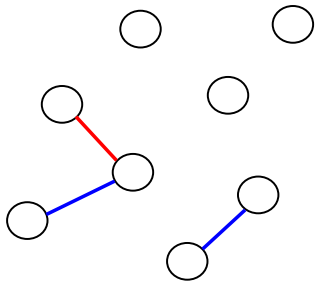
M_1



$M_0 - M_1$



$M_1 - M_0$



$(M_0 - M_1) \cup (M_1 - M_0)$

10. Flujo en Redes

Una *"red"* $G = (V, E)$ es un grafo orientado conexo que tiene dos vértices 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 $c : E \to \mathbb{R}^{\geq 0}$. La función de capacidad determina, para cada eje del grafo, la capacidad de transportar flujo que posee.

Un *"flujo factible"* en una red $G = (V, E)$ con función de capacidad c es una función $f : E \to \mathbb{R}^{\geq 0}$ que verifica:

- $0 \leq f(e) \leq c(e)$ para todo eje $e \in E$. Es decir, el flujo factible de un eje no puede ser superior a su capacidad.
- La *"Ley de conservación de Flujo"*, que dice que dado un vértice v, la suma de los flujos de los ejes que llegan a él es la misma que la suma de los flujos de los vértices que salen de él. Formalmente:

$$\forall v \in V - \{s, t\} \text{ se cumple que } \sum_{e \in in(v)} f(e) = \sum_{e \in out(v)} f(e)$$

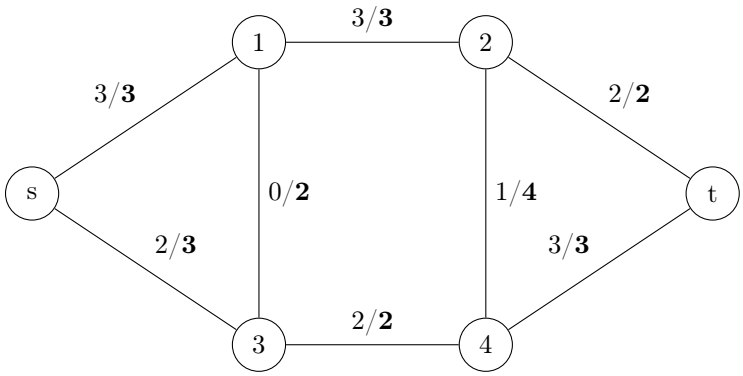
donde:

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

El *"valor del flujo"* es:

$$F = \sum_{e \in in(t)} f(e) - \sum_{e \in out(s)} f(e)$$

El problema más común cuando hablamos de flujo es el de encontrar un flujo máximo. Esto es, encontrar un F máximo en una red con una única fuente y un único sumidero.



Este problema está en P. Notar que un flujo máximo no requiere que todos los ejes estén transportando su máxima capacidad (ver $(s, 3)$). No obstante, la suma de los flujos de t es máxima.

10.1. Cortes

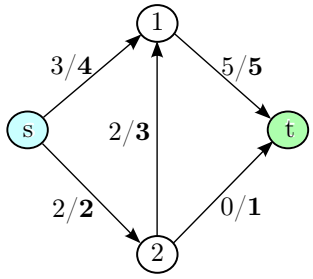
Un *"corte"* en una red $G = (V, E)$ es un subconjunto $S \subseteq V - t$ tal que $s \in S$. Es decir, un subconjunto de vértices de la red en los que está la fuente pero no el sumidero.

Dados dos cortes S y T, $ST = \{(v \to w) \in E$ tales que $v \in S$ y $w \in T\}$. Es decir, ST es el conjunto de ejes para los cuales el vértice de salida está en S y el de llegada está en T. Notar que $t \in T$.

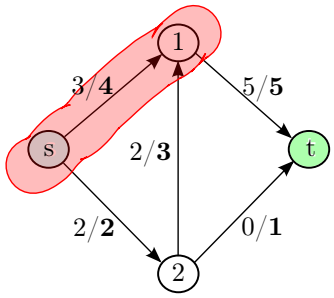
Sea f un flujo definido en una red $G = (V, E)$, sea S un corte y sea $\bar{S} = V - S$; entonces:

$$F = \sum_{e \in S\bar{S}} f(e) - \sum_{e \in \bar{S}S} f(e)$$

Es decir, el valor del flujo de la red va a ser igual al fujo saliente de un corte menos el flujo entrante al corte. Veamos esto con un ejemplo sencillo, supongamos que tenemos la siguiente red:



Es trivial ver que $F = 5$. Pero ahora miremos que pasa cuando seleccionamos los vértices s y 1 para nuestro corte:



Vemos que la suma de los flujos que van de S a $\bar{S}$ es de $5 + 2$, mientras que los flujos que van de $\bar{S}$ a S suman 2. Luego $F = 5 + 2 - 2 = 5$, que era lo que queríamos ver.

Notar también que esto pasa cuando $S = s, 1, 2$, ya que en ese caso $\sum_{e \in S\bar{S}} f(e) = 5$ y $\sum_{e \in \bar{S}S} f(e) = 0$, teniendo como resultado el mismo $F = 5$ de antes.

También se puede ver que, en el caso de que S fuera simplemente el vértice s, $F = 3 + 2$.

10.1.1. Capacidad de un corte

Cuando hablamos de un corte S, llamamos *"capacidad"* del mismo a la capacidad de todos los ejes que salen de un vértice de S y terminan en un vértice de $\bar{S}$. Dicho formalmente:

$$c(S) = \sum_{e \in S\bar{S}} c(e)$$

El *"problema del corte mínimo"* consiste en encontrar un corte S que cumple que $c(s)$ es mínima.

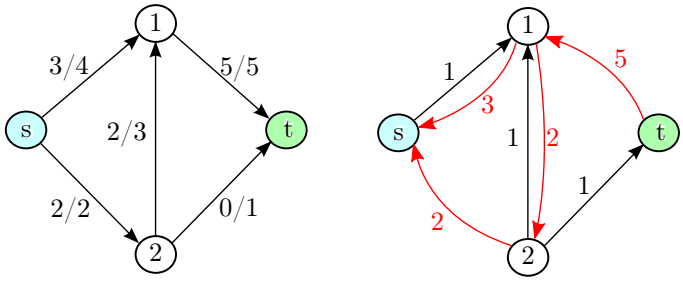
10.2. Red Residual

La *"capacidad residual"* de un eje e respecto de un flujo factible f es la diferencia entre la capacidad de e y su flujo. Es decir:
$c_f(e) = c(e) - f(e)$.

Dada una red $G = (V, E)$ con una función de capacidad c y un flujo factible f, definimos a la red que modela la capacidad disponible de G como la *"red residual"* $G_f = (V, E_R)$, donde para todo $(v \to w) \in E$:

1. $(v \to w) \in E_R \iff f((v \to w)) < c((v \to w))$. Es decir, el eje e va a estar en la red residual si su flujo es menor a su capacidad. El peso de e es $c(e) - f(e)$.
2. $(w \to v) \in E_R \iff f((v \to w)) > 0$. Es decir, todos los ejes con flujo mayor a 0 van a tener un eje contrario, y el peso de e va a ser igual a $f(e)$.

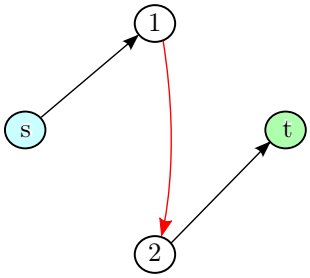
De esta manera es trivial ver que en G_f pueden haber ejes que no están en G. Por ejemplo, miremos el siguiente grafo y su red residual:



Los ejes que tenían su máxima capacidad saturada desaparecieron por no cumplir con 1., mientras que el resto de los ejes se preservaron, y ahora tienen como peso el valor de su capacidad previa. Notar también que todos los ejes con un flujo > 0 generaron ejes inversos (en rojo).

10.3. Camino de Aumento

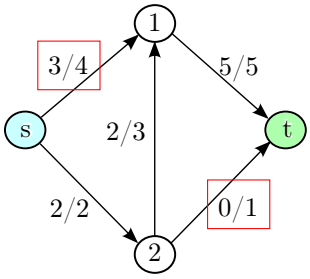
Un *"camino de aumento"* es un camino orientado P de s a t en G_f. Dada una red G, esta se encuentra en su flujo máximo si y sólo si no hay camino de aumento en G_f. Para el ejemplo anterior tenemos el siguiente camino de aumento:



Por lo que es evidente que el flujo f no es máximo. Para encontrar el flujo máximo podemos utilizar el...

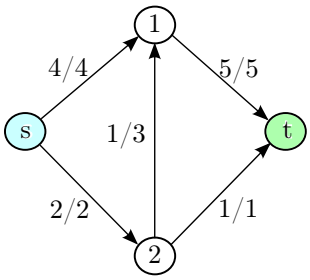
10.4. Algoritmo de Ford-Fulkerson

La idea del algoritmo es sencilla: mientras exista un camino de aumento, mejorarlo. Viendo el grafo anterior, podemos ver que el camino de aumento $s, 1, 2, t$ tiene un par de cuellos de botella:



Notar que interpretamos como *"camino"* un conjunto de ejes dirigidos que pueden no tener las direcciones necesarias para llegar de s a t.

Luego buscamos la magnitud mínima (1), se la sumamos a todos los ejes que van en nuestra dirección y se la restamos a los que van en contra:



Pseudocódigo

```
1: function FordFulkerson(G = (V, E), c : E → ℝ^{≥0})
2:     for all e ∈ E do
3:         f(e) ← 0
4:     G_f ← calcularRedResidual(G, c, f)
5:     while p ← obtenerCaminoDeAumento(G_f) do
6:         c_min ← min{c_f(e) tal que e ∈ p}
7:         for all (v, w) ∈ p do
8:             f(v, w) ← f(v, w) + c_min
9:             f(w, v) ← f(w, v) − c_min
```

Explicación

El algoritmo utiliza fuertemente la propiedad de que, mientras exista un camino de aumento en G_f, f no va a ser un flujo máximo.

2: y 3: Inicialmente el algoritmo genera un flujo factible f, donde todos los flujos para todos los ejes son 0.

4: Luego calcula la red residual G_f, utilizando la red original G, el flujo factible f y la función de capacidad c.

5: Acá comienza la iteración principal del algoritmo. Mientras pueda encontrar un camino de aumento para asignárselo a p, el flujo f no es máximo y va a poder ser mejorado haciendo lo siguiente:

6: Dado el camino p que encontré, busco el eje de menor capacidad residual del mismo. Esta capacidad residual se llama c_{min}.

7: Empiezo a recorrer todos los ejes del camino p, y:

8: A los ejes que van para mi lado, le sumo c_{min}.

9: A los que van para el otro lado, les resto c_{min}.

Si los valores del flujo inicial y las capacidades de los ejes son enteros este algoritmo realiza a lo sumo $|V| * U$ iteraciones, siendo entonces su complejidad del orden de $\mathcal{O}(|E| * |V| * U)$, con U siendo una cota superior finita para el valor de las capacidades.

10.4.1. Algoritmo de Edmonds-Karp

Este algoritmo es en realidad una implementación de Ford Fulkerson que usa específicamente BFS para encontrar el camino de aumento p en la función $obtenerCaminoDeAumento(G_f)$. Esto mejora la complejidad, dejándola en $\mathcal{O}(|V|^2 * |E|)$.

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 $\mathcal{O}(n^k)$, 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 Π de determinar si el vértice v se encuentra en un grafo G puede resolverse en tiempo $\mathcal{O}(|V| + |E|)$ usando DFS, por lo que $\Pi \in$ P.

11.2. La clase NP

Un problema de decisión Π pertenece a la clase NP (no-detérministico 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 Π que consiste en: *"dado un array A de n enteros, existe una subsecuencia del mismo que sume 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 Π_1 a uno Π_2 es una función polinomial que transforma una instancia i_1 de Π_1 en una instancia i_2 de Π_2, tal que i_1 tiene respuesta *"sí"* en Π_1 si y sólo si i_2 tiene respuesta *"sí"* en Π_2.

El problema de decisión Π_1 se reduce polinomialmente a otro problema de decisión Π_2 ($\Pi_1 \leq_p \Pi_2$) si existe una transformación polinomial de Π_1 a Π_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 Π 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 Π es un problema de decisión, podemos probar que $\Pi \in$ NP-Completo encontrando otro problema Π' 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.

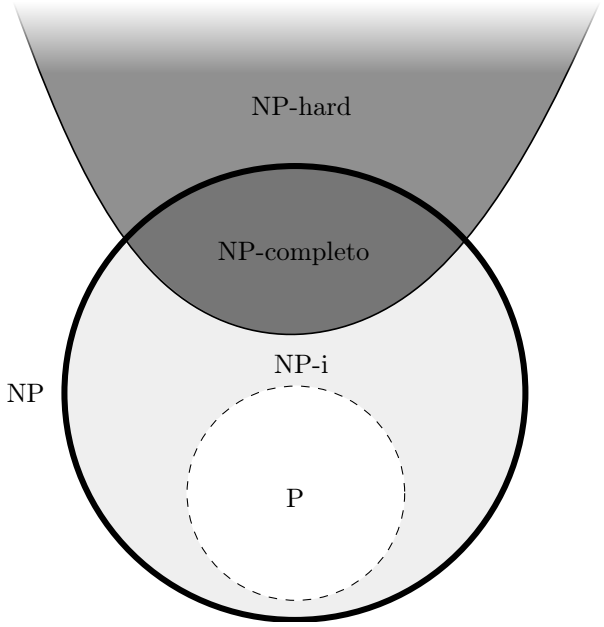
Hasta ahora no se conoce ningún problema de NP-i. De existir uno, implicaría que P $\neq$ NP, lo cual no se sabe, pero se sospecha ampliamente. Un ejemplo de un problema que se sospecha que podría estar en NP-i es el de la factorización de números naturales.

11.6. La clase NP-Hard

Similarmente a la clase NP-Completo, un problema de decisión Π es *"NP-hard"* si todo problema de NP se puede reducir polinomialmente a Π, **pero Π 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-Completos son NP-Hard, pero no todos los problemas NP-Hard son NP-Completos, por ejemplo, el Halting Problem.

Hasta el día de la fecha no se pudo demostrar que P $\neq$ NP, pero se sospecha fuertemente que este es el caso. De serlo así, el siguiente diagrama ilustra las relaciones entre las clases de problemas descriptas anteriormente:



11.7. Algoritmos Pseudopolinomiales

Decimos que un algoritmo corre en tiempo *"pseudopolinomial"* 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 $[2, .., n]$.

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