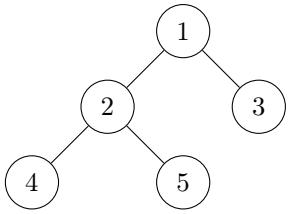
Algoritmos y Estructuras de Datos III
Resumen

amildie

-- PRUEBA DE GRAFOS --

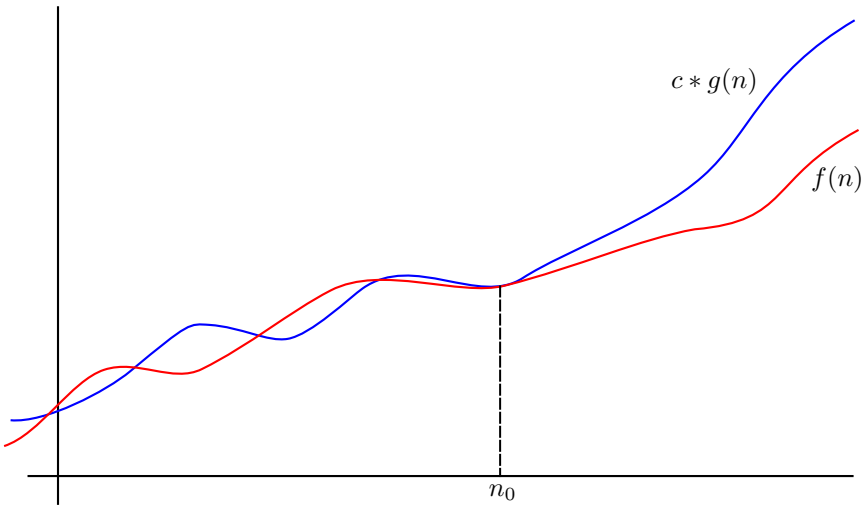


-- FIN PRUEBA DE GRAFOS --

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 Optima

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 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"*[1].

2.3. Programación Dinámica

Cuando un problema exhibe tanto subestructura óptima como soluciones sobrepuestas, 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 sobrepuestas de sus subproblemas, esta estrategia se llama *"divide & conquer"*. Es por eso que mergesort, por ejemplo, no es un problema de programación dinámica.

[1] 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 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;
}
```

2.4.2. 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.

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

```
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;
}
```

2.4.3. El problema de la mochila

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 $e_1, e_2, ..., e_k$ 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 $G = (V, X)$, 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 $G = (X, V)$ 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 $f : V \to V'$ 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"* 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.

4. Arboles

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 disconexo.

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. 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 nodo n cualquiera y fijarse si dicho nodo es el que estamos buscando. Si no lo es, marcarlo como visitado, pedir el listado de todos sus vecinos y llamarse recursivamente en cada uno de ellos, empezando por el más chico hasta llegar al más grande.

```
1: function DFS(G, v)
2:     checked ← {0, 0, ..., 0}
3:     return DFSr(G, v, 0, checked)
```

```
1: function DFSr(G, v, n, checked)
2:     if v = n then
3:         return true
4:     checked_n ← true
5:     adyacen ← getNeigh(n)
6:     for all w ∈ adyacen do
7:         if checked_w ≠ true then
8:             DFSr(G, v, w, checked)
9:     return false
```

Es decir, cuando el algoritmo llega a una hoja, hace backtracking hasta el último nodo revisado con hijos todavía sin revisar. Es por eso que la complejidad temporal del mismo es $O(n)$. También se puede implementar de forma iterativa con una pila. Al igual que BFS, DFS sólo funciona en grafos conexos.

4.3.2. BFS

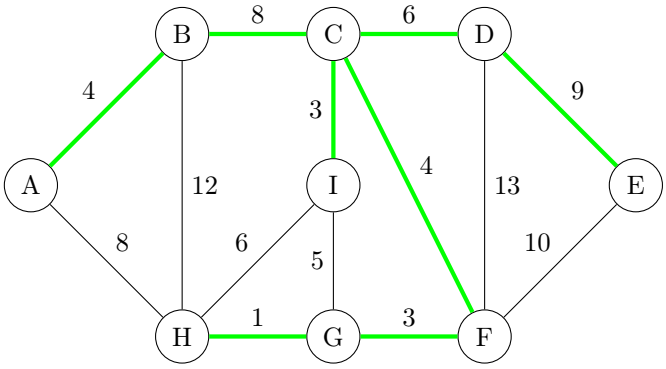
"Bradth First Search" es muy similar a DFS pero, en lugar de realizar backtracking poniendo todos los vecinos de cada vértice en un stack, pone a todos los vecinos de cada vértice en una queue.

```
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.getAdyacents(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 ← ∅
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 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 ← ∅
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 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 $l : E \to \mathbb{R}$ 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:

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

Un *"camino minimo"* 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 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. Algoritmo de Dijkstra

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, 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 u ∈ t.second.neigh() do
12:             alt ← dist[u] + length(t.first, u)
13:             if alt < dist[u] then
14:                 prev[u] ← w
15:                 dist[u] ← alt
16:                 pq.decreaseKey(< dist[u], u >)
17:     return prev
```

Explicación

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 u de $t.second$.

12: Definimos a la variable alt como la suma de la distancia mínima de u a A más la distancia de u a $t.first$ (el vértice que estamos visitando).

13: ¿Es esta nueva distancia menor a la distancia que teníamos calculada antes?. Es decir, ¿si pasamos por $t.first$ para llegar desde A a u, llegamos más rápido que si no pasamos?. Si la respuesta es **sí**, entonces mejoramos la distancia previamente calculada para u y tenemos que actualizar todo:

14: Entonces para llegar a u vamos a querer pasar por $t.first$.

15: Actualizamos la distancia mínima de A hasta u.

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.

5.2. Algoritmo de Bellman-Ford

Es un poco más lento que el algoritmo de Dijkstra, pero admite ejes de pesos negativos. Al igual que el algoritmo de Dijkstra, utiliza el principio de relaxation para ir mejorando continuamente las aproximaciones a las mejores distancias de cada vértice.

```
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 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.

5.3. Algoritmo de Floyd

5.4. Algoritmo de Dantzig

6. Grafos Eulerianos y Hamiltonianos

7. Planaridad

8. Coloreo

9. Matchings y Conjuntos Independientes

10. Flujo Máximo

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.

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

11.1. 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. La clase NP-Complete

NP-Complete es una subclase de NP, pero con la particularidad de que todos los problemas en NP pueden ser reducible en tiempo polinomial a cualquier problema de NP-Complete.

11.3. La clase NP-Hard