Algoritmos y Estructuras de Datos III
Resumen

amildie

1. Complejidad Computacional **3**

2. Programación Dinámica **4**
2.1. Subestructura Optima 4
2.2. Soluciones Sobrepuestas 4
2.3. Programación Dinámica 4
2.4. Ejemplos 5
2.4.1. Subsecuencia continua de suma máxima 5
2.4.2. Subsecuencia no-continua estrictamente creciente más larga 6
2.4.3. El problema de la mochila 7

3. Grafos **8**
3.1. Caminos y distancia 8
3.2. Subgrafos y biparticidad 9
3.3. Isomorfismo 9
3.4. Digrafos 10

4. Arboles **11**
4.1. Propiedades de los árboles 11
4.2. Arboles enraizados 11
4.3. Recorrido de árboles 12
4.3.1. DFS 12
4.3.2. BFS 13
4.4. Arbol generador 14
4.4.1. Algoritmo de Kruskal 15
4.4.2. Algoritmo de Primm 16

5. Camino Mínimo **17**
5.1. Algoritmo de Dijkstra 17

6. Grafos Eulerianos y Hamiltonianos **18**

7. Planaridad **19**

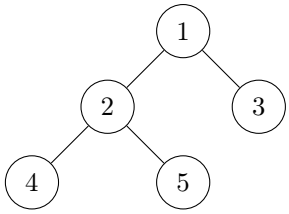
8. Coloreo **20**

9. Matchings y Conjuntos Independientes **21**

10. Flujo Máximo **22**

11. Teoría de Complejidad **23**
11.1. La clase NP 23
11.2. La clase NP-Complete 23
11.3. La clase NP-Hard 23

-- PRUEBA DE GRAFOS --

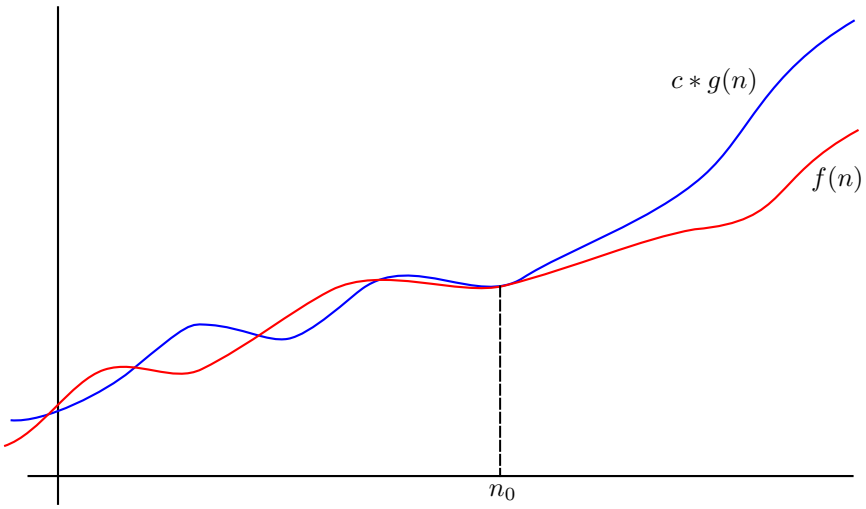


-- FIN PRUEBA DE GRAFOS --

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.

```
void DFS(Graph& g, int n, int m, bool* visited) {
  cout << "DFS␣node:␣" << m << endl;
  if(n == m) {
    cout << "found!" << endl;
    return;
  }

  visited[m] = true;
  vector<int> neigh = g.getAdyacents(m);
  for(int i = 0; i < neigh.size(); ++i) {
    if(!visited[neigh[i]]) {
      DFS(g, n, neigh[i], visited);
    }
  }
}

void DFS(Graph &g, int n) {
  bool visited[g.n];
  memset(visited, 0, sizeof(visited));
  DFS(g, n, 0, visited);
}
```

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

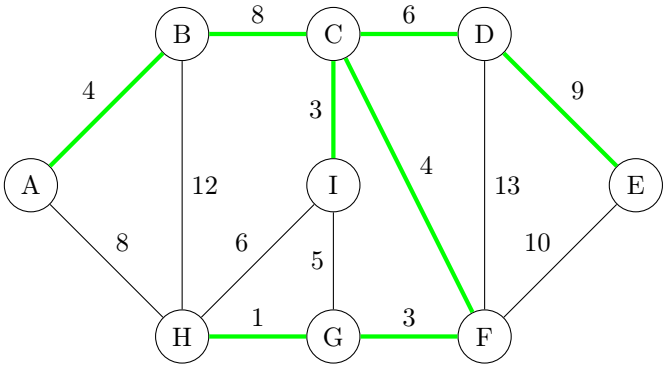
"Bradth First Search" es muy similar a DFS pero, en lugar de realizar backtracking poniendo todos los vecinos de cada vértice en un stack, pone a todos los vecinos de cada vértice en una queue.

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

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

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

    visited[t] = 1;
    vector<int> neigh = g.getAdyacents(t);
    for(int i = 0; i < neigh.size(); ++i) {
      if(!visited[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.

Para cada arista seleccionada, se fija si la misma forma un ciclo con las que ya seleccionó. Si no lo forma, la inserta en el AGM. El algoritmo termina cuando seleccionaron $n - 1$ aristas. Si el grafo no es conexo, forma un *"bosque generador mínimo"*. La complejidad del mismo es O(n log m), pero hay que cumplir un par de requisitos técnicos para alcanzarla:

1. En cada iteración del algoritmo 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).
2. También necesitamos poder determinar si el eje que estamos mirando en la iteración actual forma un ciclo con los que ya insertamos en el AGM eficientemente. Esto se puede pensar en términos de conjuntos disjuntos, donde cada conjunto es un bosque conexo dentro de mi AGM. Una estructura de datos que nos permite implementar conjuntos disjuntos es el *"union find"*.

El union find es una estructura de datos que tiene dos operaciones, find(v) y union(v_1, v_2).

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
```

4.4.2. Algoritmo de Primm

5. Camino Mínimo

5.1. Algoritmo de Dijkstra

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