# Resum Algorismica Avançada

January 16, 2020

## Contents

Resum de teoria de l'assiganatura d'informatica "Algorismica Avançada", basat en les diapositives de teoria penjades al campus.

# 1 Introducció

## 1.1 Notació Asimptòtica

- $O(n) \quad \leq n$
- $\Theta(n) \quad = n$
- $o(n) \quad < n$
- $\Omega(n) \quad \geq n$

# 2 Grafs

**Definicio 1** (**Graf no dirigit**)**.** Un graf no dirigit ens una parella ordenada **G=(V,E)**, on:

- **V** conjunt de vertex
- **E** $\subseteq \{\{x, y\} \mid x, y \in V \wedge x \neq y\}$ conjunt d'arestes

**Definicio 2** (**Graf dirigit**)**.** És igual que un graf no dirigit, però els elements de $E$ són parelles ordenades, *i.e.* $E \subseteq \{(x, y) \mid x, y \in V \wedge x \neq y\}$.

**Definicio 3** (**Graf amb pes**)**.** Un graf amb pes es un graf on a cada element de $E$ li assignem un valor, que pot representar una distacia o alguna altra cosa.

**Observacio 1.** També podem assignar altres qualitats, com color o text, a les arestes, que no cal que tinguin valor numeric.

## 2.1 Estructura de Graf

Un graf el podem representar de diverses maneres:

1. Donant una **llsita linkada** dels seus elements, es a dir donant $V$ i $E$
2. Donant una matriu amb les "relacions" entre els vertex del graf, a aqesta matriu li diem **Matriu d'adjacencia**
3. Donant una **llista d'adjacencia**, donar una llista de $V$ on a cada vertex es diu amb quins altres està conectat i com ho esta amb cada un d'ells.

**Observacio 2.** Si el graf te pocs vertex, hi ha poques "relacions" entre els diferents vertex, aleshores es poc convenient usar la representació matricial.

**Observacio 3.** Si tots els vertex del graf estan continguts en almenys una aresta, aleshores, a 1., per donar tots ens seus elements es suficient donar només $E$, les arestes.

## 2.2 Definicions

**Definicio 4** (**Grau d'un vertex**). El grau d'un vertex és el nombre de vertex al que esta conectat aquest.

**Definicio 5** (**Component conexa**). Una component conexa $V_1$ és un subconjunt de verterx que estan conectats entre si per arestes.

Això és que $V_1 \subseteq V$ i $\forall x, y \in V_1 \quad \exists a_0, \ldots, a_n \in E$, amb $a_i = \{z_i, z_{i+1}\}$, on $\forall z_i \in V$ i $z_0 = x, z_n + 1 = y$.

**Definicio 6** (**Cami**). A el conjunt ordenat $\{a_0, \ldots, a_n\} \subseteq E$de la definició anterior l'anomenem cami entre $x$ i $y$.

**Definicio 7** (**Cicle**). Un cicle es un cami on $a_0 = a_{n+1}$. Es diu que un graf és acíclic si no conte cicles.

**Definicio 8** (**Graf conex**). Diem que un graf és conex quan esta foramt per una única component conexa.

**Definicio 9** (**Cami d'Euler**). És una camí que pasa per totes les arestes del graf només una vegada.

**Definicio 10** (**Cicle d'Euler**). És un camí d'Euler que comença i acaba en el mateix vertex. Si un graf conte un cicle d'Euler s'anomena Eulerian graph.

**Definicio 11** (**Cami Hamiltonia**). És un camí que passa per tots els vertex del graf una vegada.

**Definicio 12** (**Cicle Hamiltonia**). És un camí Hamiltonia que comen'a i acaba en el mateix vertex. Un graf que conte un cicle Hamiltonia s'anomena graf Hamiltonia.

## 2.3 Algoritmes sobre Grafs

### 2.3.1 DFS

DFS, Depth-Fisrt Search, és un algoritme per buscar els vertex accessibles

Pseudo-codi per DFS recursiu:

```
function DFS(G,v):
    label v as discovered
    for all edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G,w)
```

Pseudo-codi per DFS no-recursiu:

```
function DFS-iterative(G,v):
    let S be a stack
    S.push(v)
    while S is not empty
        v = S.pop()
        if v is not labeled as discovered:
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

**Complexitat (en funció de l'estructura de graf usada):**

- Llista linkada = $\Theta(|E|^2)$
- Matriu adjacencia = $\Theta(|V|^2)$
- Lista d'adjacencia = $\Theta(|E|)$

### 2.3.2 BFS

BFS, Breath-First Search, és un algoritme per buscar vertex accessibles. Al contrari que DFS, aquest es basa en amplada, es a dir, explora tots els vertex d'una "profunditat" abans de pasar a la següent. Gracies a això ens permet trobar el camí minim entre dos vertex.

Pseudo-codi:

```
function BFS(G,start_v):
    let Q be a queue
    label start_v as discovered
    Q.enqueue(start_v)
    while Q is not empty
        v = Q.dequeue()
        if v is the goal:
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as discovered:
                label w as discovered
                w.parent = v
                Q.enqueue(w)
```

**Complexitat** $O(|V| + |E|)$, si es coneix el nombre de vertex del graf i s'usa una estructura adecuada la complexitat es pot reduir a $O(|V|)$.

### 2.3.3 Dijktra

Dijktra és un algoritme per a trobar camins minims entre vertex. La diferencia entre Dijktra i BFS és que Dijktra té en conte el pes de cada aresta, mentre que pel contrari BFS no.

Pseudo-codi:

```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:
6          dist[v] = INFINITY
7          prev[v] = UNDEFINED
8          add v to Q
9      dist[source] = 0
10
11     while Q is not empty:
12         u = vertex in Q with min dist[u]
13
14         remove u from Q
15
16         for each neighbor v of u:   // only v that are still in Q
17             alt = dist[u] + length(u, v)
18             if alt < dist[v]:
19                 dist[v] = alt
20                 prev[v] = u
21
22     return dist[], prev[]
```

**Observacio 4.** Si només busqeum el camí minim fins a un cert vertex , aleshores despres de la linia 14 si $u$ és l'objectiu ja podem parar l'algorisme i fer el "backtraking".

**Observacio 5.** Si en ves de tenir un conjunt $Q$ tenim una cua de preferencia, aleshores a la linia 12 quan haguem de trobar l'element minim no haurem de fer res perque ja estara al principi de la cua. Pero llvors haurem de modificar la prioritat de la cua en cada iteració.

**Complexitat**

$$O(|E| \cdot T_{dk} + |V| \cdot T_{em}),$$

on $T_{dk}$ i $T_{em}$ són les complexitats de $decrease - key$ i de $extract - minimum$ respectivament.

- Usant una llista linkada $= O(|V|^2)$.
- Usant una cua de prioritats, Binary Heap, $= O((|V| + |E|)log(|V|))$.

**Observacio 6.** Dijtra troba el camí minim si tots els pesos són no negatius.

### 2.3.4 Bellman-Frod

Bellman-Ford és un algorsme equivialent a Dijktra que tot i ser menys eficient és més versatil ja que també funciona amb pesos negatius.

Pseudo-codi

```
function BellmanFord(list vertices, list edges, vertex source)
   ::distance[],predecessor[]

   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays
   // (distance and predecessor) about the shortest path
   // from the source to each vertex

   // Step 1: initialize graph
   for each vertex v in vertices:
       distance[v] := inf       // Initialize the distance to all
   vertices to infinity
       predecessor[v] := null   // And having a null predecessor

   distance[source] := 0        // The distance from the source to
   itself is, of course, zero

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (u, v) with weight w in edges:
           if distance[u] + w < distance[v]:
               distance[v] := distance[u] + w
               predecessor[v] := u

   // Step 3: check for negative-weight cycles
   for each edge (u, v) with weight w in edges:
       if distance[u] + w < distance[v]:
           error "Graph contains a negative-weight cycle"

   return distance[], predecessor[]
```

**Observacio 7.** Hi ha un problema i és que podem trobar cicles amb cost negatiu.

**Complexitat** $O(|V| \cdot |E|)$.

## 2.4 Flux maxim (s'ha de millorar xd

A teoria de grafs, una xarxa de flux és un graf dirigit que conte un origen, source $S$, i una desmbocadura, sink $T$, entre altres nodes conectats amb edges. Cada edge te una capacitat maxima de flux que pot assumir. El flux dins la xara ha de complir les següents condicions:

- $\forall e_i \in E$ amb $e_i \neq S$, $e_i \neq T$, es té que el flux que entra i el que surt són iguals, *i.e.* $\sum f_{input} = \sum f_{output}$, equivalentment $\sum_{\{u:(u,v)\in E\}} f_{uv} = \sum_{\{w:(v,w)\in E\}} f_{vw}$.
- $\forall e_i \in E$ es té que $0 \leq flow(e_i) = |f_{e_i}| \leq Capacity(e_i)$.
- El flux total $|f|$ que surt de $S$ és igual al flux total que arriba a $T$, *i.e.* $|f| = \sum_{\{v:(S,v)\in E\}} f_{Sv} = \sum_{\{v:(v,T)\in E\}} f_{vT}$.
- El flux entre dos vertex $u$ i $v$, $f(u,v)$ compleix la simetria $f(u,v) = -f(v,u)$.

**Definicio 13** (**Flux maxim**)**.** El flux maxim és la maxima cuantitat de flux que la xarxa pot suportar que pasi desde $S$ fins a $T$, *i.e.* el maxim $|f|$.

**Definicio 14** (**Tall $s-t$**)**.** Un tall $C = (A, B)$, amb $A, B \subset V$ i $A \cup B = V$, és una partició dels nodes, tal que $S \in A$ i $T \in B$, i amés tenim que $X_C = \{(u,v) \in E \mid u \in A, v \in B\}$. I la capacitat del tall és $Capacity(A,B) = \sum_{(u,v)\in X_c} c_{uv}$

**Teorema 1** (**max-flow min-cut**)**.** El maxim flux entre $S$ i $T$ és igual a la minima capacitat d'entre tots els els talls que divideixen la xarxa.

**Definicio 15** (**Xarxa Residual**)**.** La xarxa residual consisteix en edges que admeten més flux. Sigui $G = V, E)$ una xaraxa de flux amb inici $S$ i destinacio $T$. Sigui $f$ el seu flux, aleshores doants $u, v \in V$la quantitat de flux addicional que accepte aquesta aresta 'es la seva capacitat residual $c_f(u,v)$, *i.e.* $c_f(u,v) = c(u,v) - f(u,v)$.

### 2.4.1 Ford-Fulkerson

És un algoritme gready per a trobar el flux maxim d'una xarxa de flux.

Pseudo-codi[1]:

```
function: FordFulkerson(Graph G,Node S,Node T):
    Initialise flow in all edges to 0
    while (there exists an augmenting path(P) between S and T in
   residual network graph):
        Augment flow between S to T along the path P
        Update residual network graph
    return
```

[1] https://www.hackerearth.com/practice/algorithms/graphs/maximum-flow/tutorial/

# 3 Gready

Un algoritme greedy és un algoritme que segueix la heuristica solucionar problemes a partir de dividir aquests en sub-problemes i trobar les solucions optimes locals a aquests per tal de donar una solució global. En molts problemes l'heuristica greedy no troba el resultat optim del problema.

En general els algoritmes greedy consta de les següents parts

- Un conjunt de candidats d'on es treura la solució.
- Una funció de selcció que triara el millor candidat i l'afegira a la solució.
- Una funció que determina la validesa del candidat per estar a la solució.
- Una funció que assigna un valor (heuristic) a les solucions parcials.
- Una funció que ens determini si hem arribat a la solució.

Els algoritmes greedy tenen la propietat que mai reavaluen les sub-solucions que ja han trobat.

**Definicio 16.** Diem que un problema te una subestructura optima si la seva solució optima conté les solucions optimes dels seus sub-problemes.

**Observacio 8.** Els algoritmes greedy són utils per a solucionar problemes que tenen una subestructura optima.

**Definicio 17.** Un arbre és un graf sense cicles.

**Definicio 18.** Donat un graf $G$ connex qualsevol, el seu Minimum-spanning-tree, $MST$, és l'arbre que té el menor pes possible contenint tots els nodes de $G$.

**Observacio 9.** Notem que el problema de trobar el $MTS$ d'un graf admet una subestructura optima.

## 3.1 Kruskal

És un algoritme que troba el minimum-spanning-tree d'un graf.

**Observacio 10.** Si el graf no és connex aleshores troba el $MST$ per a cada una de les seves compenents connexes.

Pseudo-codi

```
function KRUSKAL(G):
    A = void
    foreach v in G.V:
        MAKE-SET(v)
    foreach (u, v) in G.E ordered by weight(u, v), increasing:
        if FIND-SET(u) different of FIND-SET(v):
            A = A  {(u, v)}
            UNION(FIND-SET(u), FIND-SET(v))
    return A
```

on

MAKE-SET(v): Crea un singleton amb v.
FIND-SET(v): Troba el conjunt que conté l'element v.
UNION(B,C): Fa la unió dels conujunts B i C.

**Complexitat** $O(E \log E)$

## 3.2 Prim

És una alternativa a Kruskal. Però només funciona amb grafs connexos.

Pseudo-codi:

```
function Prim(G,s):
    for each vertex v in graph G:
        cost(v) = infinity
        prev(v) = nil
        PQinsert(v, Q)
    cost(s) = 0

    Q = makequeue() #priority queue uith cost as key
    while Q not empty:
        v = pop_min(Q)
        for each edge (v,u) | u is in Q:
            if cost(u) > ueight((v,u))
                cost(u)=ueight((v,u,))
                prev(u) = v
                decreasekey(Q,u)
```

**Complexitat** $O(E \log E)$

# 4 Programació dinamica

Hi ha dos factors importants en un problema per a poder aplicar programació dinàmica, el problema ha de tenir una subestrucura optima i subproblemes superposats (Overlapping subproblems).

**Definicio 19** (**Overlapping subproblems**)**.** Es diu que un problema té "subproblemes superposats" si el problema es pot dividir en subproblemes que són reutilitzats varies vegades o un algoritme recursiu resol el problema resolent sempre el mateix subproblema enves de sempre creantne un de nou.

Donades les dues caracteristiques dels problemes que es poden resoldre amb programació dinamica, la idea d'aquests algortmes és trobar una solució optima de cada subproblema i guradarla per a no haver de tornar a calcular-la en un subproblema futur en que la requereixi. Això té un cost computacional ja que augmenta considerablement la memoria necesaria per a solucionar el problema (la complexitat espacial), però redueix la complexitat temporal.

## 4.1 Exemples

Alguns problemes que es podene solucionar usant programació dinamica:

- Fibonacci
- Hanoi Towers
- knapsack problem

En particular l'algoritme de Floyd-Warshall és un algoritme per a trobar el cami minim entre dos nodes d'un graf amb pesos positius o negatius (sense cilces negatius), el qual és util per a grafs densos.

```
let dist be a |V||V| array of minimum distances initialized to (infinity)
for each edge (u, v) do
    dist[u][v] = w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] = 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] = dist[i][k] + dist[k][j]
            end if
```

Té complexitat $O(|V|^3)$.

# 5 Enumeratius

Els algoritmes enumeratius són aquells que s'apliquen a problemes tals que donat un input l'algoritme retorna una llista amb totes les solucions, no duplicades.

ELs algoritmes enumeratius es separent en 3 tipus principals:

- Recorregut vs. Cerca
- Backtracking
- Ramificació i poda

## 5.1 Backtracking

Són algoritmes que busquen totes, o algunes, solucions d'un problema afegint constantment candidats a la solució, i descartant cada possible candidat ("backtrack") quan determini que aquest no és una possible solució del problema.

### 5.1.1 Exemples

- Map coloring
- 8 queens
- Sudoku

## 5.2 Ramificació i poda

Donat un problema considerem l'espai de les possibles solucions, aleshores en cada pas generem dos o més casos per cada element de l'espai (branch/ramificació), i calculem una cota/bound de cada un d'aquests, i si aquesta cota no és satisfactoria es considera que no pot ser solució i s'elimina (poda), de manera que no es segueix ramificant aquest candidad.

Les cotes normalment es calculen usant funcions heuristiques, que donen una aproximació de si un candidat pot o no ser solució sense solucionar el problema, amb un cost molt menor que comprovant-ho ralment com es fa en el cas de backtracking.

Els algoritmes de ramificació i poda són utils per a "resoldre" problemes NP-hard, com el Traveling salesman problem o el 0/1 knapsack problem.

# 6 Complexitat

**Teorema 2** (**Master Theorem**)**.** Analisis de la complejidad de algoritmos recursivos.

Siendo $T(n)$ la complejidad del algoritmo, $f(n)$ la complejidad del caso base, $a$ el número de subproblemas a cada nivel de recursión y $b$ el factor por el que dividimos la entrada, denotamos la complejidad de un algoritmo recursivo como:

$$T(n) = aT\big(\frac{n}{b}\big) + f(n)$$

Si notamos la función $f(x)$ en base a su complejidad de la forma $O(n^d)$ obtenemos:

$$T(n) = aT\big(\frac{n}{b}\big) + O(n)$$

Y según el teorema masters podemos afirmar que:

$$\begin{aligned} T(n) &= O(n^d) && \text{if } d > log_b a \\ T(n) &= O(n^d \log n) && \text{if } d = log_b a \\ T(n) &= O(n^{\log_b a}) && \text{if } d < log_b a \end{aligned}$$