

**Universitat de Lleida**

---

## **Pràctica 2: Aquæductus Optimus**

---

Assignatura d'Algorítmica i complexitat del Grau d'Enginyeria  
Informàtica

Pablo Fraile Alonso

21 de juny de 2021

# Índex

<b>1 Greedy</b>	<b>4</b>
1.1 Cost algorisme . . . . .	5
1.1.1 Iteratiu . . . . .	5
1.1.2 Recursiu . . . . .	5
1.1.3 Empíric . . . . .	5
1.2 Pseudo-codi de l'algorisme . . . . .	6
<b>2 Backtracking</b>	<b>7</b>
2.1 Cost algorisme . . . . .	7
2.1.1 Recursiu . . . . .	7
2.1.2 Iteratiu . . . . .	8
2.1.3 Empíric . . . . .	9
2.2 Pseudo-codi de l'algorisme . . . . .	10
<b>3 Dynamic Programming</b>	<b>11</b>
3.1 Aplicació i funcionament en el nostre cas d'ús . . . . .	11
3.2 Demostració per reducció a l'absurd . . . . .	16
3.3 Especificació formal . . . . .	17
3.4 Fòrmula algorisme . . . . .	17
3.5 Cost algorisme . . . . .	18
3.5.1 Iteratiu . . . . .	18
3.5.2 Recursiu . . . . .	18
3.5.3 Empíric . . . . .	18
3.6 Pseudo-codi algorisme . . . . .	19
<b>4 Problemes al realitzar la pràctica</b>	<b>20</b>
4.1 Nombres en C++ . . . . .	20
<b>5 Consideracions</b>	<b>20</b>
<b>6 Conclusions</b>	<b>22</b>
<b>Apèndix A: Repositori github</b>	<b>23</b>

## Índex de figures

1	Exemple algorisme Greedy . . . . .	4
2	Cost empíric emprant Greedy en Python . . . . .	5
3	Exemple crides recursives Backtracking . . . . .	7
4	Nombre de nodes a les crides recursives de backtracking . . . . .	8
5	Nombre d'iteracions de la funció <i>is_valid</i> a backtracking . . . . .	9
6	Cost empíric emprant Backtracking en Python . . . . .	9
7	Entrada exemple . . . . .	11
8	Exemple representat eix de coordenades . . . . .	12
9	Exemple representat en forma de dígraf . . . . .	13
10	Resultat de $f(E)$ . . . . .	13
11	Resultat de $f(D)$ . . . . .	14
12	Resultat de $f(C)$ . . . . .	14
13	Resultat de $f(B)$ . . . . .	15
14	Resultat del aqüeducte mínim ( $f(A)$ ) . . . . .	15
15	Aqüeducte de punt A a punt J . . . . .	16
16	Aqüeducte de punt A a punt J passant per K . . . . .	16
17	Graf que representa un possible $R'_{a...k}$ . . . . .	16
18	Cost empíric emprant Dynamic Programming en Python . . . . .	18
19	Cost empíric emprant Dynamic Programming en C++ . . . . .	19

# 1 Greedy

Un algorisme greedy (o algorisme voraç), segueix l'heurística de resolució de problemes que consisteix en trobar i triar la solució local òptima en cada etapa. En aquest cas d'ús, veurem que aquesta no és una solució “real” ja que, en alguns casos, les solucions locals òptimes no compleixen el requeriment del cost mínim.

Un exemple per anar d'un punt  $a$  a un punt  $d$  seguint l'algorisme greedy, seria el de la figura 1.

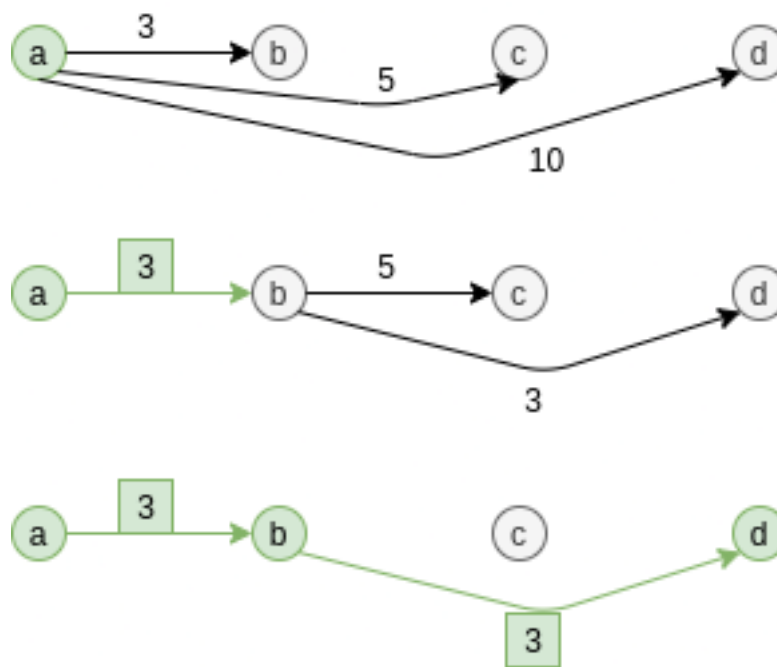


Figura 1: Exemple algorisme Greedy

## 1.1 Cost algorisme

### 1.1.1 Iteratiu

Veiem que el programa té 3 bucles anidats<sup>1</sup> on cadascun té cost de  $O(n)$ , i que, en el pitjor dels casos, haurem recorregut  $n^3$  vegades. Per tant, podem dir, que el cost de l'algorisme en forma iterativa és de  $O(n^3)$ .

### 1.1.2 Recursiu

Òbviament, si l'únic que hem fet ha sigut passar l'algorisme iteratiu a recursiu (o a l'inversa), el cost d'aquest continuarà sent el mateix. L'únic que canviarà serà que ara s'utilitzarà memòria de la pila d'execució i no memòria de heap, per tant, el programa és més propens a sofrir un *stackoverflow*. Finalment podem dir que el cost és de  $O(n^3)$ .

### 1.1.3 Empíric

Un cop executat varies vegades l'algorisme amb un conjunt de  $n$  diferents, ens crea la gràfica de la figura 2. Aquesta, concorda amb la justificació de la notació assintòtica donada anteriorment.

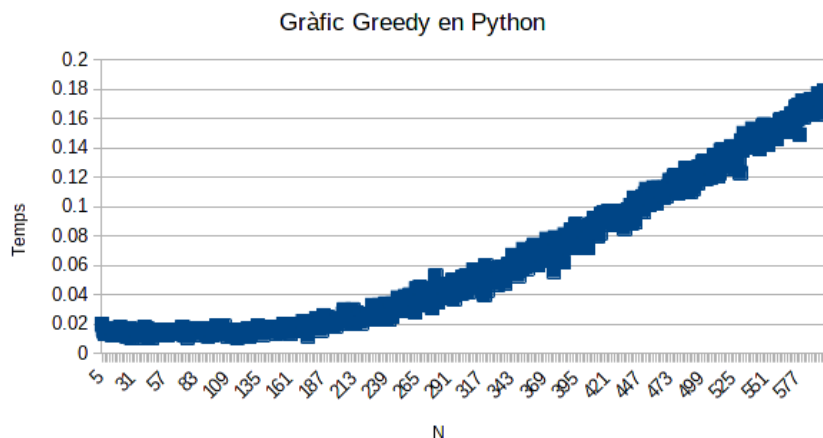


Figura 2: Cost empíric emprant Greedy en Python

<sup>1</sup>Un bucle per recórrer tots els punts, un bucle per agafar el punt de mínim cost entre un punt i el final i un bucle per comprovar si l'arc és vàlid.

## 1.2 Pseudo-codi de l'algorisme

El pseudo-codi del algorisme està inspirat en la sintaxis de python, però **NO és python**. Per tant, aquest codi únicament dona l'idea de l'algorisme. L'implementació funcional en C++ i python es mostra en la carpeta de codi del projecte.

```
def get_minimum_aqueduct():
    current_point = 0
    accumulator = cost_support(current_point)
    while current_point != final_point:
        cost, current_point = get_minimum_arch(current_point)
        if cost == infinity:
            return infinity
        accumulator += cost
    return accumulator

# returns the minimum cost (and the point who achieves it) from init_point
def get_minimum_arch(init_point):
    minimum = infinity
    minimum_point = None
    for current from init_point + 1 to final_point:
        if valid_arch(init_point, current):
            cost = total_cost(init_point, current)
            if cost < minimum:
                minimum = cost
                minimum_point = current
    return minimum - cost_support(init_point), minimum_point
```

## 2 Backtracking

La tècnica de Backtracking consisteix en anar provant totes les diferents solucions i quedar-se amb la més òptima. Una resolució mitjançant backtracking per anar del punt A al punt D on es pot crear arcs entre tots els punts seria la figura 3.

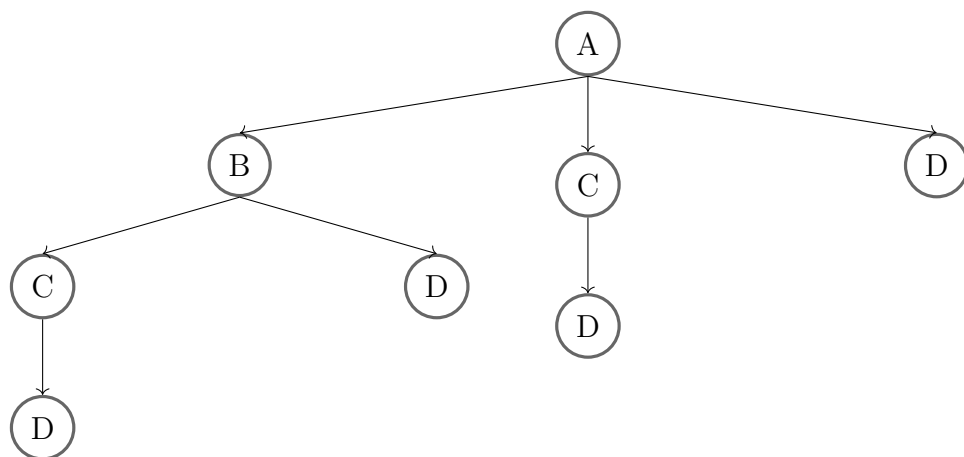


Figura 3: Exemple crides recursives Backtracking

### 2.1 Cost algorisme

#### 2.1.1 Recursiu

Primerament es va tenir en compte quin seria el cost de les crides recursives que, tal i com s'aprecia a la figura 4, veiem que creen una forma d'arbre que haurem de recórrer de forma postordre (finalitzem primer subarbre esquerra, després dret i finalment l'arrel).

Es dedueix, a partir de l'equació de recurrència creada per l'arbre, que el nombre de nodes a recórrer en funció de  $n$  (on  $n$  és el nombre de pilars) serà  $2^{n-1}$ .

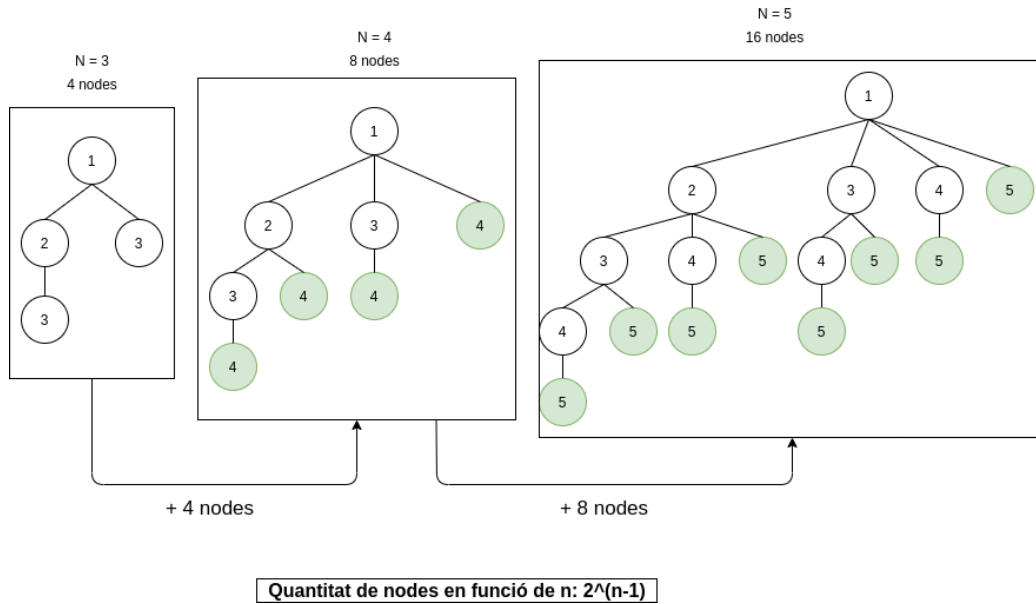


Figura 4: Nombre de nodes a les crides recursives de backtracking

A més, falta calcular el cost de la funció *is\_valid* en l'arbre anterior (figura 5). Aquest s'ha trobat calculant quantes vegades iterava el bucle en funció de  $n$ , i s'ha vist que es creava la seqüència 4, 11, 26, 57, 120, etc. S'ha deduït i comprovat que l'equació de recurrència en funció de  $n$  és  $2^{n-1} - n$ .

Finalment, un cop ja calculades les dos equacions de recurrència, podem sumar-les per obtenir el cost del algorisme, el que ens donaria:

$$2^{n-1} + 2^{n-1} - n \implies 2 * 2^{n-1} - n$$

Que, en notació assimp tòtica, podem representar com:  $O(2^n)$

### 2.1.2 Iteratiu

La forma més senzilla de passar un algorisme que utilitza backtracking a iteratiu és mitjançant l'ús d'una pila<sup>2</sup>. El cost algorímic serà igualment  $O(2^n)$ , però ara s'utilitzarà memòria de heap i no de pila d'execució, per tant és menys probable que sofrim un *stackoverflow*.

<sup>2</sup>“Imitant” en certa forma el que fa la pila d'execució



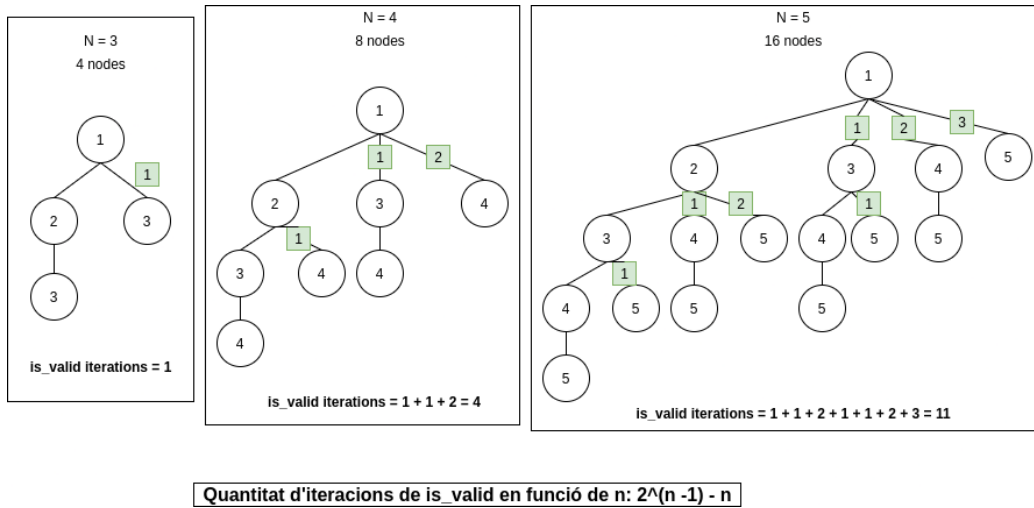


Figura 5: Nombre d'iteracions de la funció `is_valid` a backtracking

### 2.1.3 Empíric

Finalment, s'ha executat múltiples vegades el algorisme i s'ha aconseguit formar la gràfica de la figura 6, que concorda amb la nostra afirmació de que el cost és de  $O(2^n)$ .

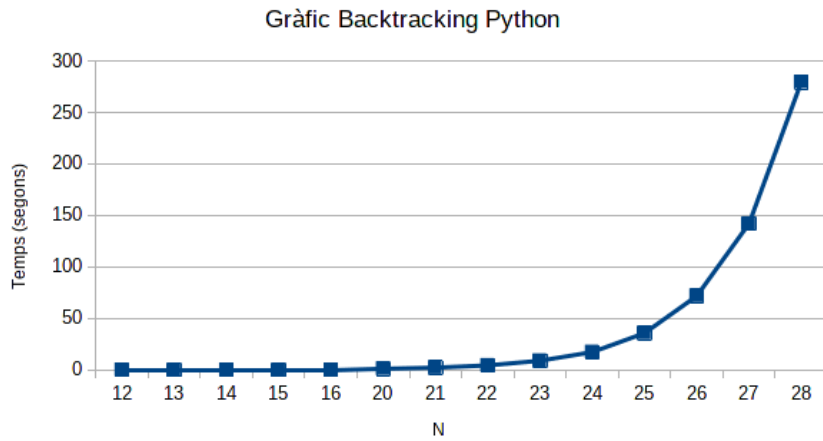


Figura 6: Cost empíric emprant Backtracking en Python

## 2.2 Pseudo-codi de l'algorisme

El pseudocodi del algorisme està inspirat en la sintaxis de python, però **NO és python**. Per tant, aquest codi únicament dona l'idea del algorisme. L'implementació funcional en C++ i python es mostra en la carpeta de codi del projecte.

```
def get_minimum_aqueduct(current_point_index=0):
    # base case
    if current_point_index == final_point_of_aqueduct:
        return cost_support(current_point_index)
    # backtrack
    min_cost = infinity
    for analized_point from current_point_index + 1 to final_point_index:
        if valid_arch(current_point_index, analized_point):
            cost_of_analized = get_minimum_aqueduct(analized_point)
            cost_of_analized += cost_support(current_point_index)
            cost_of_analized += cost_arch(current_point_index, analized_point)
            min_cost = min(min_cost, cost_of_analized)
    return min_cost
```

### 3 Dynamic Programming

Abans de poder comentar la solució, hem d'entendre que és el principi d'optimitat:

Principi d'optimitat: Una política òptima té la propietat que sigui quin sigui l'estat inicial i la decisió inicial, les decisions restants han de construir una política òptima respecte a l'estat resultat de la primera decisió.

(Richard E. Bellman)

Per tant, seguint aquesta definició, podem dir que un problema podrà ser resolt seguint el principi d'optimitat si la seva solució òptima pot ser construïda eficientment a partir de les solucions òptimes dels seus subproblemes. En altres paraules, que podem resoldre un problema gran donades les solucions dels seus problemes petits.

#### 3.1 Aplicació i funcionament en el nostre cas d'ús

En el nostre problema dels aqüeductes, veiem que podem aplicar el principi d'optimitat per trobar una solució òptima, ja que la solució és construïda eficientment a partir de les solucions òptimes dels seus subproblemes. Per explicar-ho millor he decidit resoldre un petit exemple.

Donada la següent entrada:

5	6	180	20
0	0		
2	2		
3	1		
5	3		
7	2		

Figura 7: Entrada exemple

On podem veure que tenim 5 punts, una altura d'aqüeducte de 6,  $\alpha = 180$  i  $\beta = 20$ . Si ho representem en un eix de coordenades, el perfil del sòl ens

queda com la figura 8. En canvi, si ho volem examinar en forma de dígraf (ja descartant opcions que no són vàlides) ens queda com a resultat la figura 9.

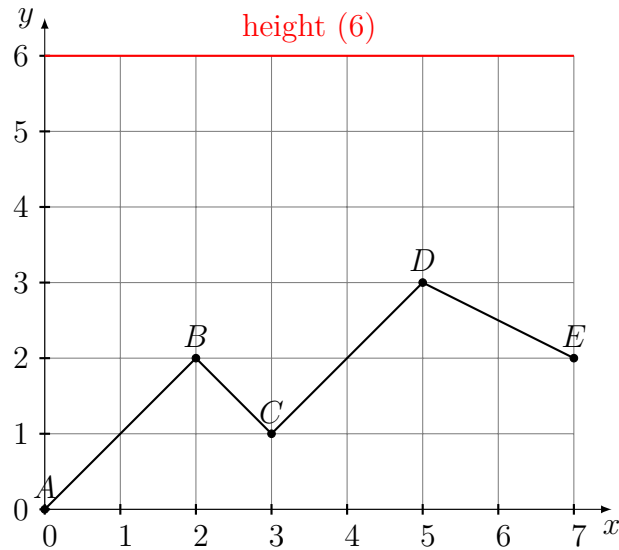


Figura 8: Exemple representat eix de coordenades

A continuació, anomenarem la funció  $f(x)$  com el mínim cost per anar al node E. En el cas d'estar al propi node E, aquesta funció retornarà 0 (figura 10).

En el cas de  $f(D)$ , únicament té una opció possible, anar del node D a F. Per tant, el cost mínim serà el recorregut mostrat a la figura 11 i la funció retornarà el valor del cost de crear un pilar a D, més el cost de crear un pilar a F i el cost de crear el arc de D a F.

En el cas de  $f(C)$ , té l'opció d'anar a D o d'anar a E. En aquest cas calcularem el cost de C a E i el cost de C a D +  $f(D)$  i agafarem el mínim. Calculem cost de C a E i ens dona 1940, en canvi, el cost de C a D +  $f(D)$  ens dona 2320. Per tant, el cost mínim des de C serà anant de C a E (figura 12).

En el cas de  $f(B)$ , farem el mateix que amb  $f(C)$ . Calcularem quant val

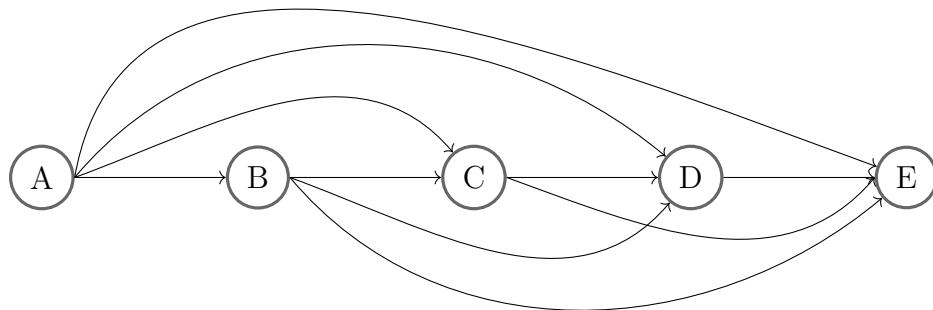


Figura 9: Exemple representat en forma de dígraf

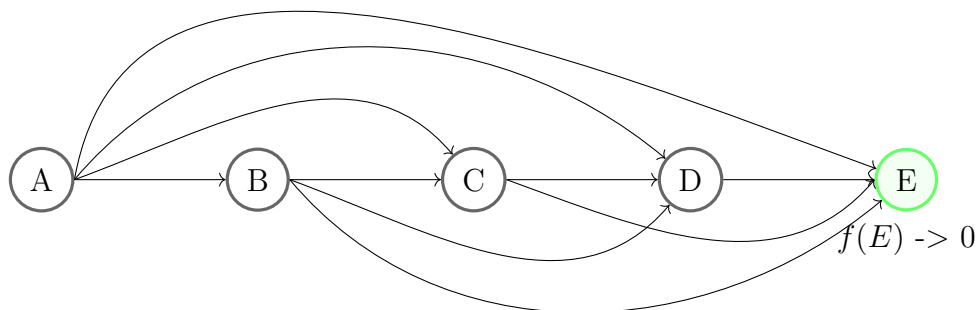


Figura 10: Resultat de  $f(E)$

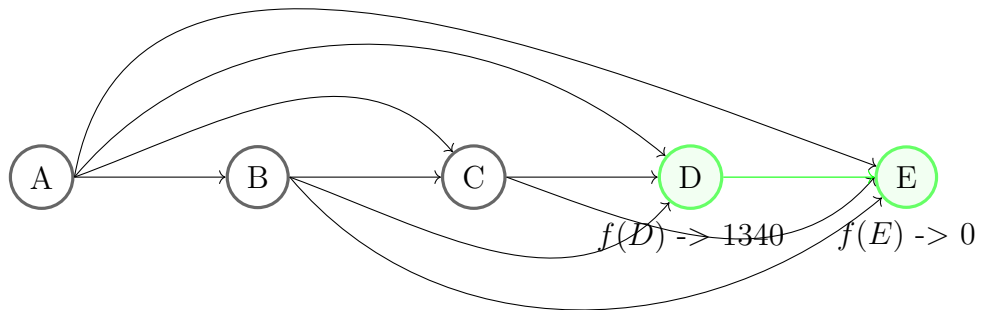


Figura 11: Resultat de  $f(D)$

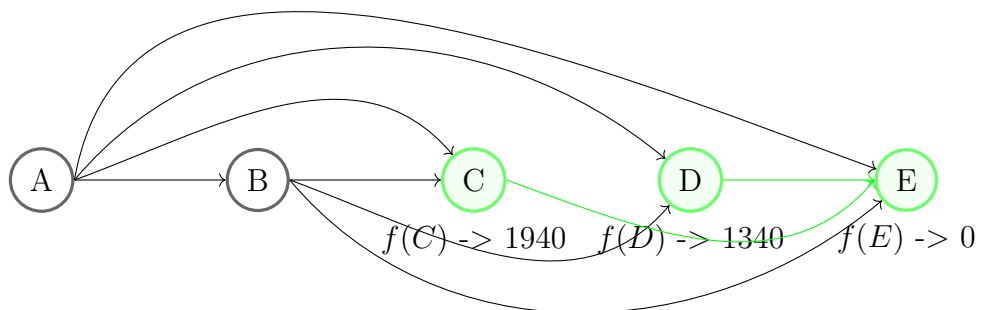


Figura 12: Resultat de  $f(C)$

el cost de B a E, de B a C +  $f(C)$  i de B a D +  $f(D)$  i agafarem el mínim. En aquest cas, el mínim és de B a E (1940) (figura 13).

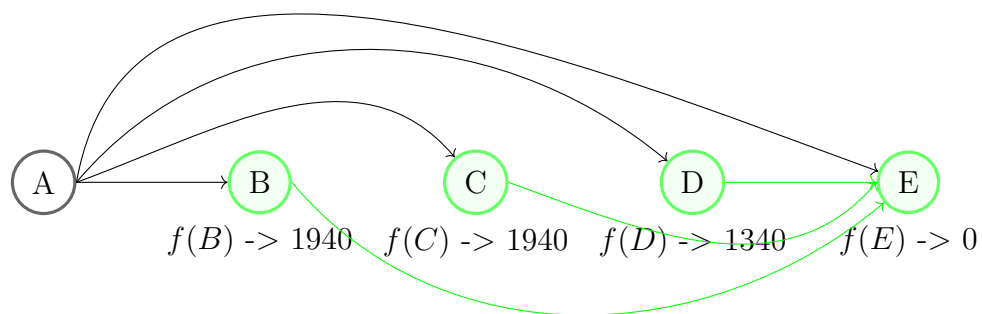


Figura 13: Resultat de  $f(B)$

Finalment, en el cas de  $f(A)$ , haurem de calcular el cost de A a E, de A a B +  $f(B)$ , de A a C +  $f(C)$  i de A a D +  $f(D)$  i agafar el mínim cost. En aquest cas, el mínim és de A a E (2780) (figura 14).

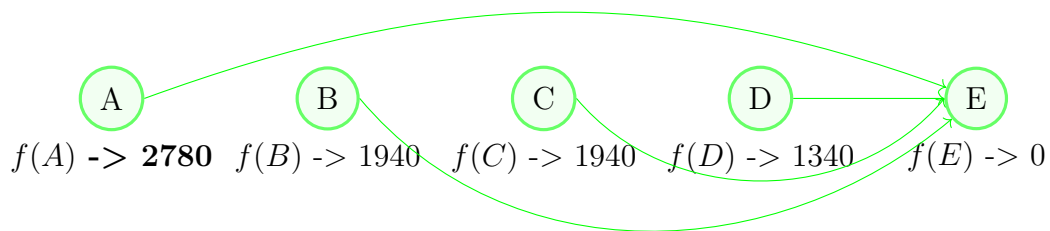


Figura 14: Resultat del aqüeducte mínim ( $f(A)$ )

### 3.2 Demostració per reducció a l'absurd

Donat un aqüeducte que va d'un punt A a un punt J i del qual sabem que el recorregut  $R_{a...j}$  és l'òptim (figura 15).

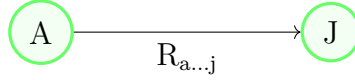


Figura 15: Aqüeducte de punt A a punt J

Assumirem també que aquest recorregut passa per el punt K. Per tant, ara podem separar el recorregut com  $R_{a...k}$  &  $R_{k...j}$  (figura 16).

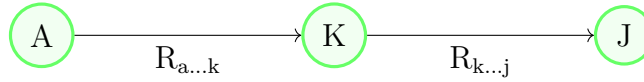


Figura 16: Aqüeducte de punt A a punt J passant per K

Ara donarem com a hipòtesis que del punt A al punt K pot haver-hi un recorregut més òptim, que anomenarem  $R'_{a...k}$  (figura 17).

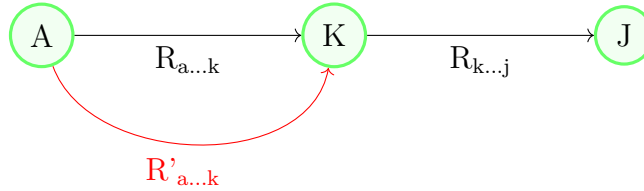


Figura 17: Graf que representa un possible  $R'_{a...k}$

Si  $R'_{a...k}$  és més òptim que  $R_{a...k}$ , llavors vol dir que:

$$R'_{a...k} < R_{a...k}.$$

Llavors:



$$R'_{a...k} + R_{k...j} < R_{a...k} + R_{k...j}$$

Però aquesta afirmació **NO** pot ser certa! Ja que en un principi hem assegurat que  $R_{a...k} + R_{k...j}$  era la solució òptima i, per tant, no hi pot haver-hi cap més petita que aquesta.

### 3.3 Especificació formal

Precondició :

- Una altura màxima del aqueducte ( $h$ ), on  $(1 \leq h \leq 10^5)$
- Els factors de cost  $\alpha$  i  $\beta$ , on  $(1 \leq \alpha, \beta \leq 10^4)$
- Un conjunt de punts  $(x, y)$ :  
 $P = \{p_1, p_2, \dots, p_n \mid p_n = (x, y)\} \wedge \forall_{x, y} \in \mathbb{N} \wedge n \geq 2 \wedge x_i (0 \leq x_1 < x_2 < \dots < x_n \leq 10^5) \wedge y (0 \leq y_i < h)$

Postcondició: cost mínim de crear l'aqueducte mitjançant els valors  $\alpha$  i  $\beta$  o  $\infty$  en cas de que no sigui possible <sup>3</sup>:

$resultat \in \mathbb{N} \wedge resultat \leq \min(R)$ , on  $R$  és el conjunt de possibles resultats.

### 3.4 Fòrmula algorisme

Un cop ja sabem el funcionament del algorisme i hem demostrat que el seu comportament és correcte, podem especificar-lo amb una fórmula molt similar a la de l'equació de Bellman (ja que tal i com s'ha dit a la subsecció 3.1 aquest problema es resol seguint el principi d'optimitat).

$$v(x_0) = \min(f(x_0) + v(x_1)) \quad (1)$$

On  $v(x)$  és la fórmula per calcular el cost mínim del aqueducte,  $x_0$  és el primer pilar del aqueducte i  $x_1$  és el resultat d'aplicar  $v(x)$  al pilar que va després de  $x_0$ .

---

<sup>3</sup>Ja que tal i com es comentava a classe, havia de retornar un tipus, en aquest cas un enter o infinit. Realment en cas del programa s'ha de treure un enter si és possible o una cadena de caràcters dient que és impossible.

## 3.5 Cost algorisme

### 3.5.1 Iteratiu

Veiem que el programa té tres bucles anidats, i que en el pitjor dels casos (quan estem analitzant el primer punt,  $p_0$ ) haurem recorregut els tres  $n^3$  cops. Per tant, podem dir que el cost de l'algorisme en forma iterativa és de  $O(n^3)$ .

### 3.5.2 Recursiu

Òbviament, si l'únic que hem fet ha sigut passar l'algorisme iteratiu a recursiu, el cost d'aquest continuarà sent el mateix. L'únic que canviarà serà que ara s'utilitzarà memòria de la pila d'execució i no memòria de heap, per tant el programa es més propens a sofrir un *stackoverflow* (o *recursion error* en python). Finalment podem dir que el cost és de  $O(n^3)$ .

### 3.5.3 Empíric

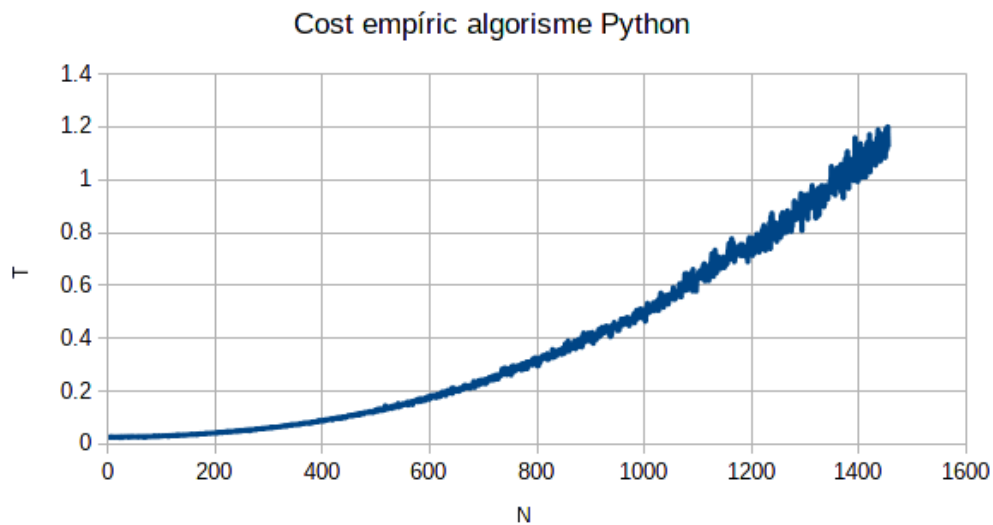


Figura 18: Cost empíric emprant Dynamic Programming en Python

Recalcar que, en python, es van fer els testos fins  $N=1450$  (figura 18), ja que si no el temps d'execució era sumament lent, mentres que amb C++

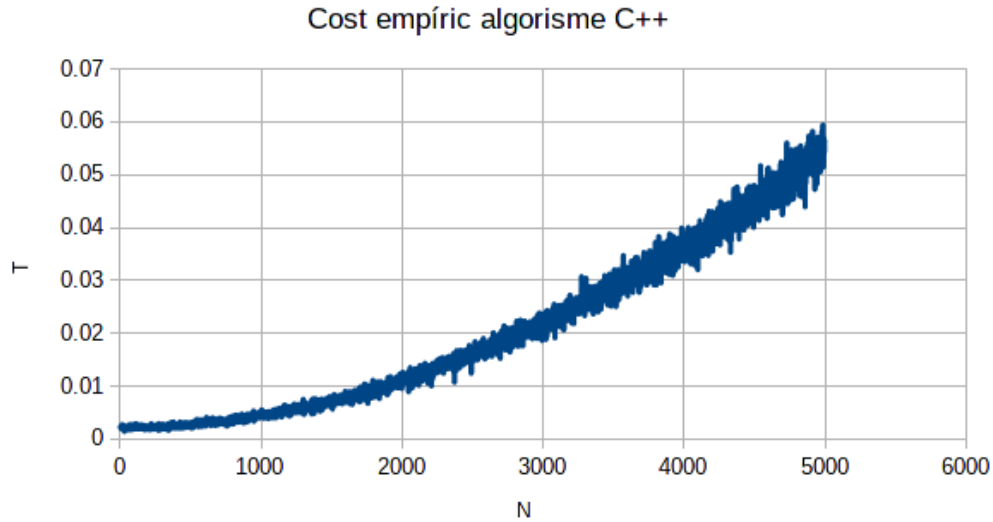


Figura 19: Cost empíric emprant Dynamic Programming en C++

(figura 19) si que es van poder fer execucions fins  $N=5000$ .

Igualment, es pot apreciar, que les dos gràfiques són similars a una funció  $f(x^3)$ .

### 3.6 Pseudo-codi algorisme

El codi iteratiu segueix l'algorisme explicat a la secció 3. Tal i com s'ha anat comentant, el pseudocodi està inspirat en la sintaxis de python, però **NO és python**. Per tant, aquest codi únicament dona l'idea del algorisme, l'implementació en C++ i python es mostra en la carpeta de codi del projecte.

```
def get_minimum_aqueduct():
    buffer = Array of N elements containing Nones (null)
    for current_point from penultimate_point to first_point:
        buffer[current_point] = get_minimum_cost(current_point, buffer)
    return buffer[0]
```

```

def get_minimum_cost(index, buffer):
    minimum = infinity
    for current_point from index + 1 to final_point:
        if valid_arch(index, current_point):
            cost = total_cost(index, i, buffer)
            minimum = min(minimum, cost_points)
    return minimum

def total_cost(first_point, second_point, buffer):
    # returns cost support first point + cost arch between first point
    # and second point + cost_second_point (if exists in buffer,
    # if not cost support second point)

```

## 4 Problemes al realitzar la pràctica

### 4.1 Nombres en C++

Primerament es va pensar i elaborar l'algorisme en el llenguatge de programació python, i a continuació es va migrar el codi a C++. Això ens va originar un problema, ja que no es va pensar que python emprava tipus dinàmics (el mateix intèrpret assigna un tipus a cada variable de forma intel·ligent), mentre que a C++ el propi programador assigna els tipus. Llavors, com els tests eren summament grans i podien arribar a fer operacions com per exemple  $10000^2$ , el tipus integer era insuficient, i es va haver d'emprar tipus com, per exemple, “long long int” o “unsigned long long int”.

## 5 Consideracions

1. Al arxiu Makefile del projecte de python, s'ha afegit una opció per fer test del codi mitjançant l'eina pylint, però habilitant l'opció d'ignorar els “warnings” provocats per no tenir comentaris per cadascun dels mètodes i classes.
2. Al programari de python s'ha emprat el patró de disseny *template*, per tal de que el codi de les classes *Point*, *Circumference* i el *parser* del fitxer no estigués repetit en 6 arxius *.py* diferents. El fet de que python

s'hagi de situar a un altre directori (anomenat *common*) per importar els fitxers ha ocasionat afegir linees de codi (que a ulls de l'eina pylint eren poc correctes). Aquestes s'han evitat afegint els comentaris:

```
# pylint: disable=wrong-import-position
# pylint: disable=import-error
```

per tal d'obtenir bona puntuació a l'evaluació.

3. S'han mogut tots els tests a un directori anomenat *test*, ja que com s'ha fet la pràctica en python y C++, si no es separava, es repetien els mateixos tests en dues carpetes diferents.
4. S'ha decidit que el binari resultant del codi escrit en C++ porti habilitades les opcions d'optimització O3, ja que si no el rendiment baixava considerablement.
5. El codi no inclou anàlisi d'errors sobre el fitxer que se li proporciona. En cas de que el format/fitxer sigui erroni, es veurà l'excepció corresponent de python o C++.
6. Tot i que s'ha augmentat la pila d'execució en el codi recursiu de python, recordem que la pila no és infinita i, per tant, en cas d'una N molt gran el programa llançarà un error de recursivitat.
7. L'algorisme iteratiu de backtracking s'ha implementat amb la tècnica adquirida a la tercera pràctica de l'assignatura d'Estructura de Dades del quadrimestre passat. Per tant, és probable que alguns companys que també hagin cursat l'assignatura tinguin la solució similar ja que s'ha emprat la mateixa tècnica.
8. S'ha decidit mantenir els fitxers de python en format *.py*, ja que així es podien emprar altres intérprets de python més fàcilment modificant el Makefile del projecte. En cas de que l'avaluador volgués l'execució com es mostra en l'enunciat (*\$/aqueducte <fitxer entrada>*), s'hauria d'afegir la *shebang* al fitxer de python i, a més, canviar el nom de l'arxiu:

```
Al principi de l'arxiu .py
#!/usr/bin/env python3
```

```
Modificar el nom de l'arxiu:
mv aqueducte.py aqueducte
```

```
En cas del recursiu:
mv aqueducte_recursive.py aqueducte_recursive
```

## 6 Conclusions

Aquesta pràctica m'ha servit per adquirir els següents coneixements:

- Comprendre i implementar un algorisme mitjançant greedy.
- Comprendre i implementar un algorisme mitjançant backtracking.
- Aprendre les bases de la programació dinàmica i veure quina millora afegien en aquest algorisme.
- Iniciar-me en la programació en C++.
- Analitzar els diferents costs (empírics i teòrics) d'un algorisme.
- Convertir un algorisme iteratiu a recursiu (i a l'inversa).

## Apèndix A: Repositori github

El repositori de github es pot trobar [aquí](#). Recalcar que llegir el README adjuntat amb el repositori és indispensable per saber com executar els tests de forma correcta i donar-se una idea de com es troba estructurat el projecte.