Pràctica 1 || Aquæductus

Pablo Fraile Alonso

17 d'abril de 2021

Índex

Índex de figures

1 Funcionament algorisme

1.1 Primera idea: Backtracking

Primerament es va provar calcular totes les diferents solucions i quedar-se amb la més òptima. Òbviament, les solucions no vàlides es podien anar descartant per estalviar temps. Aquesta tècnica s'anomena backtracking. Una resolució pera trobar tots els camins per anar del punt A al punt D mitjançant backtracking seria la figura 1.

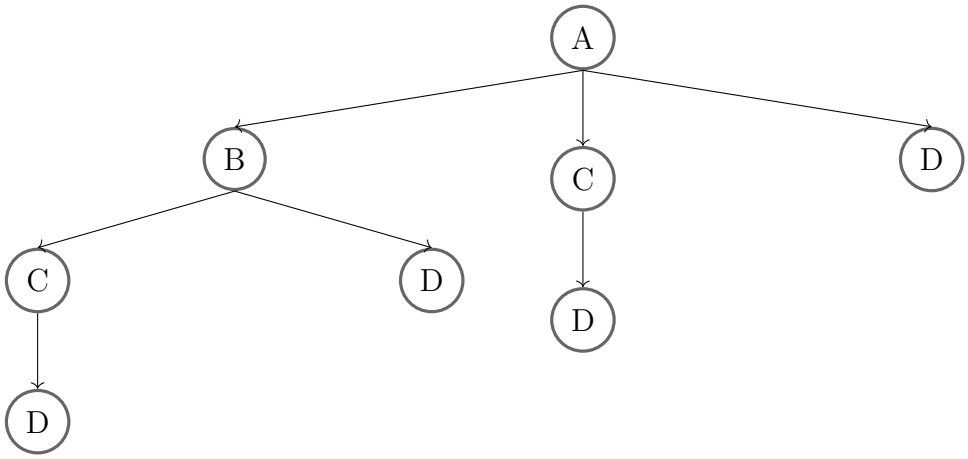

Figura 1: Exemple backtracking

1.1.1 Per què no utilitzar Backtracking en aquest cas

El cost que s'ens demanava era de $O(n^3)$, on n era el nombre de punts del sòl. En canvi, veiem que l'algorisme mitjançant backtracking té un cost de $O(n!)$, per tant, no ens serveix. Igualment, s'ha decidit deixar el codi en una branch del repositori a github anomenada *backtracking*.[1]

[1]No s'ha de confondre amb la branch *main*, que és la versió final amb un cost de $O(n^3)$.

1.2 Alternativa a backtracking: Principi d'optimitat

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)

Segons 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, podem resoldre un problema gran donades les solucions dels seus problemes petits.

1.2.1 Aplicació i funcionament en el nostre cas d'ús

En el nostre problema dels aqüeductes veiem que podem aplicar el principi d'optimitat per a trobar una solució òptima, ja que la solució és construïda eficientment a partir de les solucions òptimes dels seus subproblemes. Per a 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 2: Entrada exemple

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 3, en canvi, si ho volem examinar en forma de dígraf (ja descartant opcions que no són vàlides) ens queda com a resultat la figura 4.

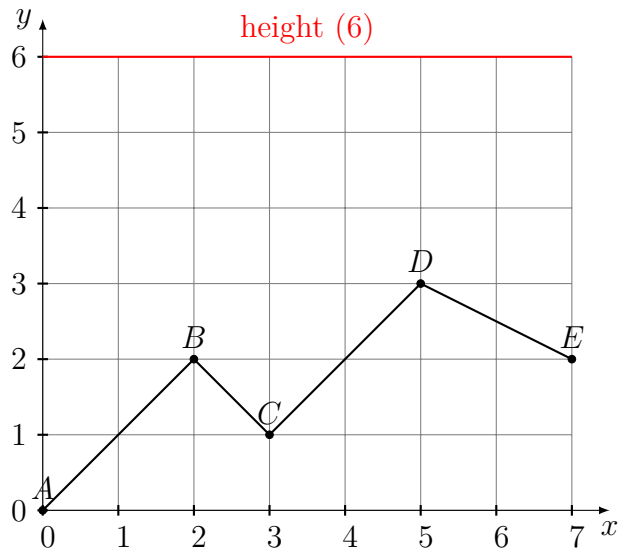

Figura 3: Exemple representat eix de coordenades

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 5).

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 6. 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 $f(C)$ hi ha l'opció d'anar a D o d'anar a E. En aquest cas calcularem el cost de C a E i el compararem amb el cost de C a D + $f(D)$. Agafarem el mínim. Per exemple, en el nostre cas, el cost de C a E ens dona 1940, en canvi, el cost de C a D + $f(D)$ és 2320. Per tant, el cost mínim de C serà el primer (de C a D) (figura 7).

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

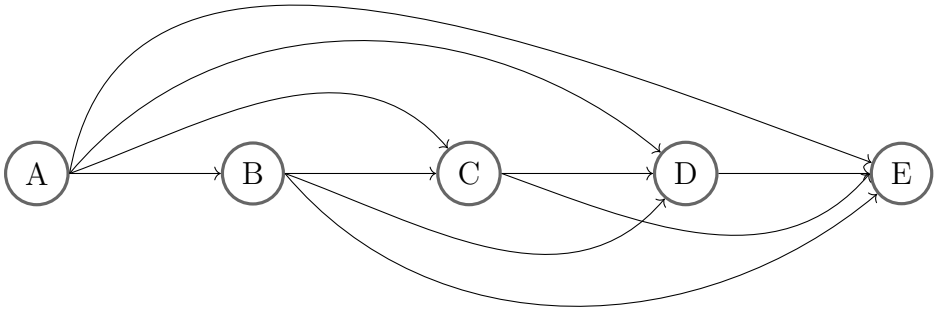

Figura 4: Exemple representat en forma de dígraf

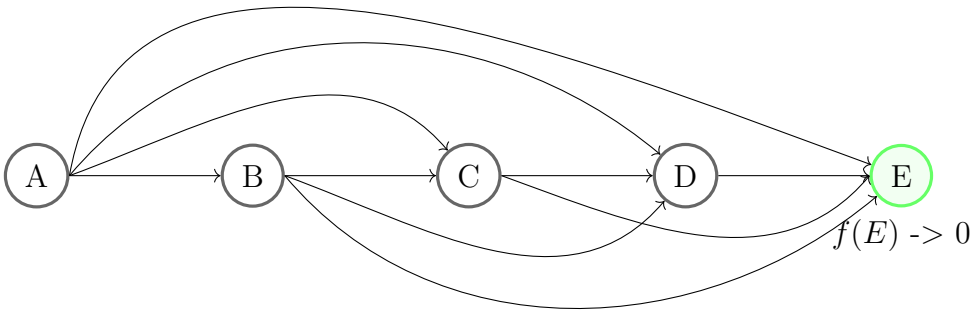

Figura 5: Resultat de $f(E)$

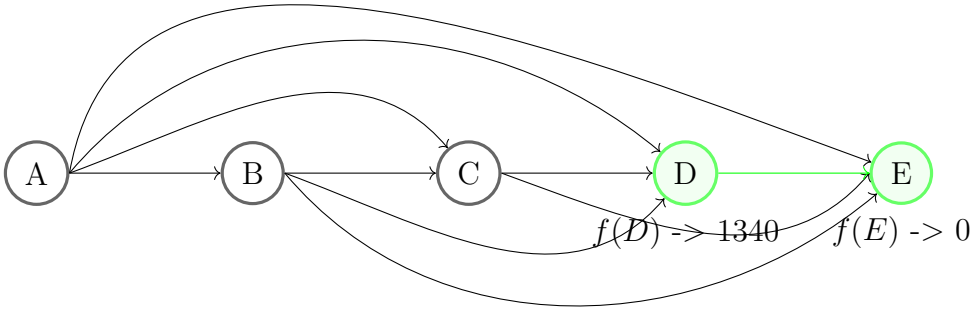

Figura 6: Resultat de $f(D)$

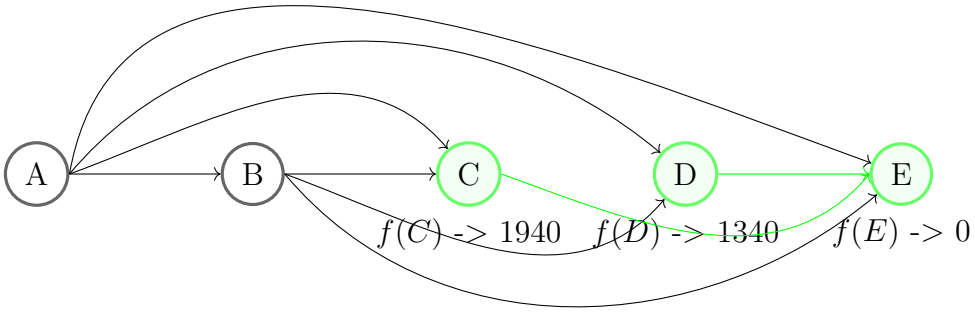

Figura 7: Resultat de $f(C)$

el cost de B a E, de B a C + $f(C)$ i de B a D + $f(D)$ i de nou, agafarem el mínim. En aquest cas, el mínim és de B a E (1940) (figura 8).

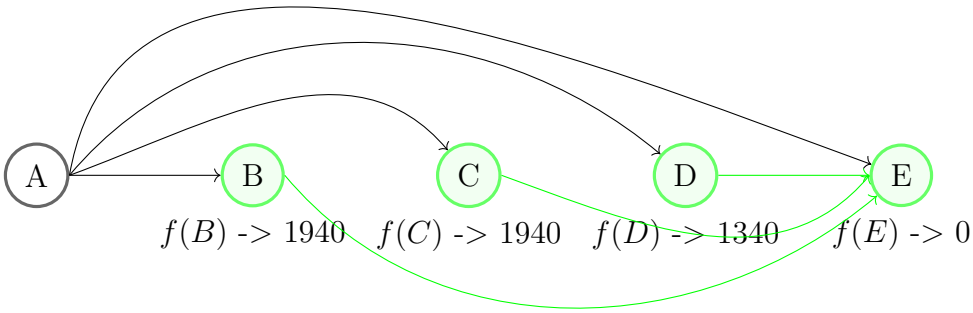

Figura 8: Resultat de $f(B)$

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 9).

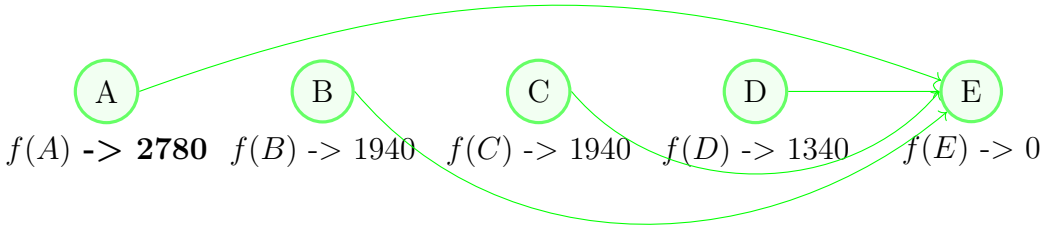

Figura 9: Resultat del aqüeducte mínim ($f(A)$)

1.2.2 Demostració per reducció al 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 10).

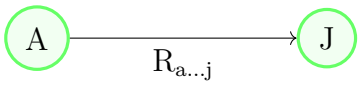

Figura 10: Aqüeducte de punt A a punt J

Assumint també que aquest recorregut passa per el punt K, per la qual cosa podem separar el recorregut com $R_{a...k}$ & $R_{k...j}$ (figura 11).

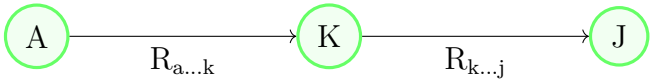

Figura 11: Aqüeducte de punt A a punt J passant per K

Donarem com a hipòtesis que del punt A al punt K pot haver-hi un recorregut mes òptim que anomenarem $R'_{a...k}$ (figura 12).

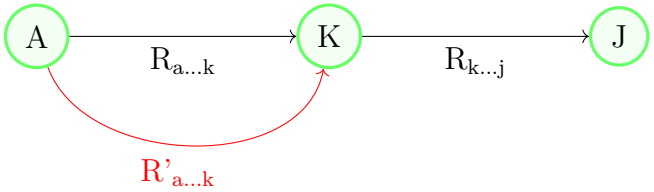

Figura 12: Dígraf que representa un possible $R'_{a...k}$

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

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

Per tant:

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

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, com a conseqüència, no pot haver-hi cap més petita que aquesta.

1.2.3 Especificació formal

Precondició :

- Una altura màxima del aqueducte (h), on $(1 \leq h \leq 10^5)$
- Els factors de cost α i β, on $(1 \leq \alpha, \beta \leq 10^4)$
- Un conjunt de punts (x, y):
  $P = \{p1, p2, ...p_n \mid p_n = (x, y)\} \wedge \forall_{x,\, y} \in \mathbb{N} \wedge n \ni \mathbb{N} \wedge n >= 2 \wedge x_i(0 \leq x_1 < x_2 < ... < x_n \leq 10^5) \wedge y(0 \leq y_i < h)$

Postcondició: cost mínim de crear l'aqueducte mitjançant els valors α i β o ∞ en cas de que no sigui possible [2]:

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

1.2.4 Fórmula algorisme

Un cop ja sabem el funcionament del algorisme i hem demostrat que el seu comportament és correcte, podem especificar-ho amb una formula molt similar a la de l'equació de Bellman (ja que tal i com s'ha dit a la subsecció 1.2.1 aquest problema es resolt seguint el principi d'optimitat).

$$v(x_0) = min(f(x_0) + v(x_1)) \tag{1}$$

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

[2] Ja que, tal i com es comentava a classe, havia de retornar un tipus (en aquest cas, un enter o infinit). Tot i que en aquest programa s'ha de treure un enter si és possible o una cadena de caràcters dient que és impossible.

2 Cost de l'algorisme

2.1 Iteratiu

Veiem que el programa té 3 bucles anidats i, en el pitjor dels casos, recorrerem n^3 cops. Per tant podem dir que el cost de l'algorisme en forma iterativa és $O(n^3)$.

2.2 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, la qual cosa el programa serà més propens a sofrir un *stackoverflow*. Finalment, podem dir que el cost és $O(n^3)$.

2.3 Empíric

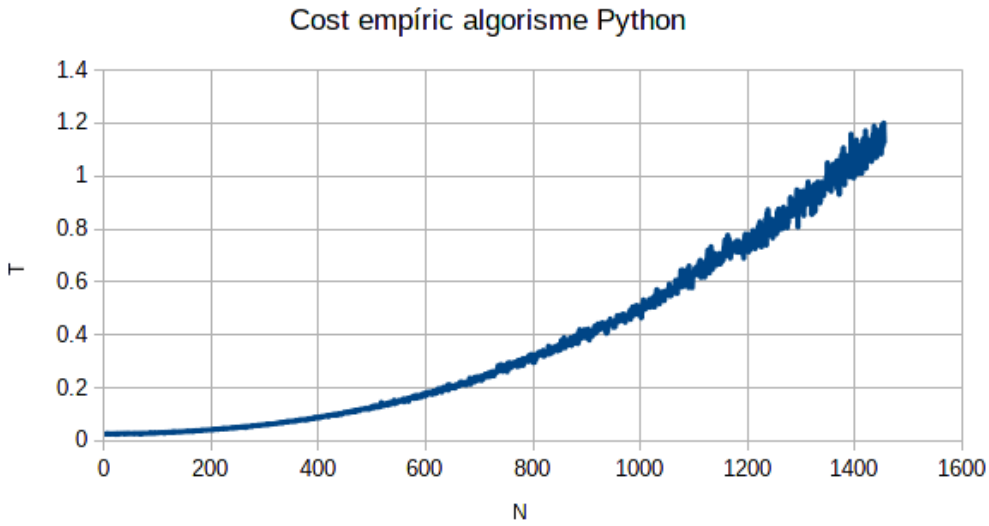

Figura 13: Cost empíric en Python

Recalcar que a Python es van fer els tests fins N=1450 (figura 13) ja que, a partir de N=900, el temps d'execució era summament lent. En canvi, amb

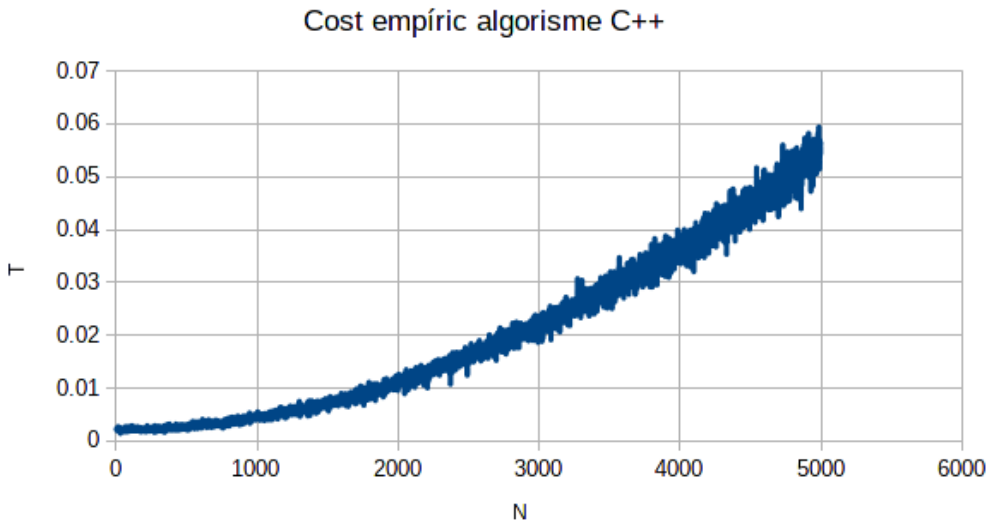

Figura 14: Cost empíric en C++

C++(figura 14) si que es van poder fer execucions fins N=5000.
Igualment es pot apreciar que les dos gràfiques surten similars a una funció $O(n^3)$.

3 Problemes al realitzar la pràctica

3.1 Nombres en C++

Primerament es va desenvolupar l'algorisme en el llenguatge de programació python i, a continuació, es va migrar el codi a C++. Però, no es va pensar que python funciona amb tipus dinàmics i és el mateix intèrpret qui assigna un tipus a cada variable de forma intel · ligent, mentre que a C++, el propi programador és qui assigna els tipus. Això va generar un problema, ja que, com els tests eren summament grans i podien arribar a fer operacions de 10000^2, el rang del tipus integer no era suficient, i es va haver d'utilitzar altres tipus amb un rang superior com, per exemple, "long long int" o "unsigned long long int".

4 Consideracions

1. Al arxiu Makefile del projecte de python s'ha afegit una opció per executar el test de llegibilitat de codi mitjançant l'eina pylint. S'ha habilitat l'opció d'ignorar els "warnings" provocats per no generar comentaris per cadascun dels mètodes i classes.

2. S'han mogut tots els tests a un directori anomenat *test* ja que, com s'ha fet la pràctica en python i C++, si no es separaven, els mateixos tests es duplicaven en dues carpetes diferents.

3. S'ha decidit que el binari resultant del codi escrit en C++ porti habilitades les opcions d'optimització O3 ja que, en cas contrari, el rendiment baixava considerablement.

4. El codi no inclou anàlisi d'errors sobre el fitxer proporcionat. En cas de que el format/fitxer sigui erroni es veurà l'excepció corresponent de python o C++.

5. Tot i que s'ha augmentat la pila d'execució en el codi recursiu de python, recordem que la seva mida no és infinita i, per tant, en cas d'una N molt gran, el programa llançarà un error de recursivitat.

5 Conclusions

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

- Comprendre i implementar un algorisme mitjançant backtracking.
- Aprendre les bases de la programació dinàmica i veure quina millora proporcionaven en aquest cas.
- Iniciar-me en la programació en C++.
- Analitzar els diferents costs (empírics i teòrics) d'un algorisme.
- Convertir un algorisme iteratiu a recursiu.
- Crear un informe mitjançant LaTeX, que facilita l'ús d'expressions lògiques, matemàtiques i grafs.

Apèndix A: Pseudocodi algorisme iteratiu

El codi iteratiu segueix l'algorisme explicat a la secció 1.2.

Recalcar que la llista que guarda els resultats és utilitzada dintre de la funció *get_minimum_cost_for_index* per no tornar a calcular valors mínims de punts que ja havíem resolt amb anterioritat.

```
def get_minimum_aqueduct(self):
    for i in range(self.num_points - 2, -1, -1):
        minimum_of_this_point = self.get_minimum_cost_for_index(
                                                i)
        self.point_values_buffer[i] = minimum_of_this_point
    return self.point_values_buffer[0]
```

Apèndix B: Pseudocodi algorisme recursiu

El codi recursiu és molt similar al iteratiu. De fet, l'única diferència és pensar el cas base (quan l'índex és 0) i a partir d'allí anar movent-se per tots els punts augmentant el sufix de l'array mentres disminueix el prefix fins arribar al cas base (índex 0, on el sufix és tota l'array mentre que el prefix és inexistent).

Igualment, per mantindre la concordança de codi amb la versió iterativa, s'ha decidit fer una funció que serveix de *wrapper* per amagar que realment estem cridant a una solució recursiva.

```
def get_minimum_aqueduct_recursive(self, index: int):
    if index == 0:
        minimum_of_this_point = self.get_minimum_cost_for_index(
                                            index)
        return minimum_of_this_point
    self.point_values_buffer[index] = self.
                                    get_minimum_cost_for_index(
                                    index)
    return self.get_minimum_aqueduct_recursive(index - 1)

# wrapper for recursive function
def get_minimum_aqueduct(self):
    return self.get_minimum_aqueduct_recursive(self.num_points -
                                     2)
```

Apèndix C: Repositori github

El repositori de github es pot trobar aquí. Tal i com s'ha comentat anteriorment hi ha dues branques: la branca que inclou el codi final (branch: main) i la que conté el codi de backtracking (branch: backtracking). Recalcar que la versió de backtracking no té el codi tant polit com la versió final, ja que no és l'objectiu d'aquesta pràctica.