IFT2125 automne 2016 - Devoir 4

Philippe Caron

7 décembre 2016

1 Césures

Sans la programmation dynamique, il faudrait calculer toutes les configurations possible du mot w passé en entrée afin de trouver la meilleure. Sachant que le nombre d'arrangements d'espaces possibles dans le mot w correspond à $\binom{k-1}{s}$, on peut déterminer que le nombre d'appel à la fonction *realism* sera :

$$n = \sum_{s=0}^{k-1} \binom{k-1}{s}$$

Ce qui fait beaucoup d'appel. On cherche donc à trouver une méthode plus efficace avec la programmation dynamique.

Bien que le nombre de configuration d'espace soit très élevé, le nombre de mots possible à former l'est beaucoup moins, le principe est donc de remplir un tableau contenant les valeurs de probabilité de chaque mot :

longueur	w_1	w_2	...	w_{k-1}	w_k
1	$realism(w_1)$	$realism(w_2)$	...	$realism(w_{k-1})$	$realism(w_k)$
2	$realism(w_1)$	$realism(w_2)$	...	$realism(w_{k-1})$	
...	$realism(w_1)$	$realism(w_2)$	...		
k - 1	$realism(w_1)$	$realism(w_2)$			
k	$realism(w_1)$				

Ce qui ne nécessite que $\frac{k^2}{2}$ appels à *realism*. Ensuite il suffit de trouver l'arrangement optimal des valeurs de ce tableau. Pour ce faire, on commence avec un mot «complètement espacé» puis on choisi les endroits optimaux ou retirer les espaces.

```
function Cesure(w):
   array R[1..k, 1..k]
   {Remplissage du tableau}
   for s <- 1 to k do
      for u <-0 to s do
         {Conserve la partie du mot entre u et k - u}
         R[u, s] <- realism(w[u : k - u])
   {Determine l'espace optimal a supprimer}
   for s <- k to 1 do
      for u <- 0 to s do
         {comparaison}
```

*** Après réflexion, cette technique est vorace et ne fonctionne pas à tous les coups. Pour le faire avec la programmation dynamique il faudrait trouver une manière de circonscrire les résultats à un petits ensemble de valeur.

2 Tableau rectangulaire

(a) Algorithme naïf

L'algorithme naïf doit ressembler vaguement à ceci :

```
function MemeCouleur(T, l, c):
   m <- min(l, c)
   for d <- m to 1 do
      for h <- m to l do
         OuterLoop:
         for v <- m to c do
            {verifier toutes les cases dans d}
            color <- T[h - 1, v - 1]
            for i <- h - d to h do
               for j <- v - d to h do
                  if T[i, j] != color then
                     goto OuterLoop
            return d
```

En commençant par les gros carrés, l'algorithme vérifie tous les carrés existants et dès qu'il en trouve un ne présentant qu'une couleur, il retourne sa taille. Comme on peut le constater, il y a énormément de boucles ce qui rend l'exécution lente.

Dans le pire cas, la boucle h est exécutée environ l fois, et la boucle v environ c fois, et la boucle d $min(c, l)$ fois. Supposons que c et l sont relativements similaires, on peut dire que cet algorithme est $\Omega(n^3)$. On peut imaginer le pire cas où le rectangle est rayé de couleur différentes ce qui fait que la première boucle de vérification j doit s'exécuter au complet avant de trouver une couleur différente, ce qui rajoute un d opérations. C'est impossible cependant qu'un rectangle ne présentant aucun carré de la même couleur fasse rouler les boucles intérieures i et j au complet car à ce moment les boucles extérieures arrêteraient en trouvant un carré de taille $d > 1$. On peut donc être certain que cet algorithme est $O(n^5)$.

(b) Programmation dynamique

3 Monochrome

Le principe de la fonction `findWay()` est de trouver un chemin entre le sommet considéré et d'autres sommets en se déplaçant toujours par en avant. Quand la fonction trouve un nouveau sommet, elle doit également s'assurer que le nouveau sommet est connecté à tous les autres. Si celui-ci ne l'est pas, elle retourne le nombre de sommet actuel, sinon elle continue d'explorer à partir du nouveau sommet.

```
1  # Fonction utilisee pour determiner le k maximal
2  def maxK(T):
3     max = 0;
4     for x in range(1, 3):
5        v = findWay(T, x, [0], 0)
6        if v > max:
7           max = v
8     return max
9
10 # Backtracking, explore les solutions
11 def findWay(G, C, V, lvl):
12    max = lvl
13    for x in range(V[-1] + 1, len(G)):
14       if G[0][x] == C:
15          for y in range(len(V)):
16             if G[V[y]][x] == C:
17                t = findWay(G, C, V + [x], lvl + 1)
18                if t > max:
19                   max = t
20             else: return lvl
21    return max
```

4 Monochrome2

(a) Biais

Cet algorithme est vrai-biaisé, car s'il répond vrai il ne se trompe jamais. En effet, il teste la question précisément et celle-ci ne cherche qu'à démontrer l'existence d'un sous-ensemble quelconque.

(b) $p(m)$-correct

Le calcul de la fiabilité de l'algorithme reviens à calculer la probabiliter de tomber sur des bons sommets. Cela inclu la probabilité de piger la bonne couleur.

Supposons que la probabilité qu'un sous-ensemble existe au sein d'une couleur soit uniforme. La probabilité que la solution existe dans le vert, le rouge, le bleu, ou aucune couleur (pas de solution) est la même. De plus on considère que les 4 situations possible (aucune solution, solution dans 1 couleur, solution dans 2 couleurs, solutions dans 3 couleurs) sont equiprobables. Alors la probabilité de piger la bonne couleur est de 0% dans le premier cas, 33%

dans le second, 66% à deux couleurs, puis 100%. Donc un probabilité générale de une chance sur deux. Maintenant, viens la probabilité de piger des sommets qui font partie de la solution.

On peut voir le choix des sommets comme une série de pige sans remise successives. À chaque pige, la probabilité de prendre un sommet qui fait parti de la solution est de k. Plus on pige, plus la probabilité de piger des sommets qui font exclusivement partie de la solution diminue (multipliée par k à chaque fois). La probabilité que les m sommets qui soient pigés fassent partie de la solution est donc de k^m.

L'algorithme est donc $0.5 \cdot k^m$-correct, ou k représente la densité de sommet appartenant à une solution. Si cette probabilité est de 50%, il est donc $(0.5 \cdot 0.5^m)$-correct.

(c) Probabilité d'erreur

On peut définir une fonction dont le but est d'identifier le nombre d'essai nécessaire requis pour faire descendre la probabilité d'erreur à une probabilité arbitraire.

Selon (b), disons que la probabilité de succès est de $0.5 \cdot k$, obtenir la probabilité d'erreur revient à calculer la probabilité de manquer un succès après n essais. Celle-ci est obtenue par :

$$\left(1 - 0.5 \cdot k^m\right)^n$$

et puisque que $1 - 0.5 \cdot k^m < 1$ on sait que le tout tend vers 0 au fur et à mesure que n tend vers ∞. Donc oui, il est possible de faire descendre la probabilité d'erreur en dessous de 1% étant donné un certain nombre d'essais n, et la formule pour le faire est la suivante :

$$\left\lceil \frac{\log 1\%}{\log\left(1 - 0.5 \cdot k^m\right)} \right\rceil = n$$

Cependant c'est une fonction qui croit très rapidement, il est donc possible que pour un certain m relativement faible, n soit impraticable.

(d) Cas précis

Selon la formule en (b) :

$$\left\lceil \frac{\log 0.5}{\log\left(1 - 0.5 \cdot k^m\right)} \right\rceil = n$$

Ce qui donne les résultats suivants pour $k = 0.5$

$$\begin{aligned} m = 2 &\Rightarrow 6 \\ m = 10 &\Rightarrow 1420 \\ m = 100 &\Rightarrow 1.75734... \times 10^{30} \end{aligned}$$