Université Piere et Marie Curie

3I003 Algorithmique

Compte rendu projet

Professeurs :
Maryse Pelletier

Auteurs :
BEROUKHIM Keyvan
FONTAINE Ulysse

Sommaire

Introduction : 2

Enumeration des solutions 2
Question 1 : 2
Question 2 : 2

Partie théorique : 2

Cas du vecteur libre : 3
Question 3 3
Question 4 3
Question 5 3
Question 6 4
Question 7 4

Cas du vecteur non libre : 5
Question 8 5
Question 9 5

Partie Expérimentale 6

Enumération 6
Question 10 6
Question 11 8
Question 13 9

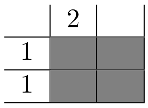
Propagation 11
Question 14 11
Question 15 14
Question 16 15

Introduction :

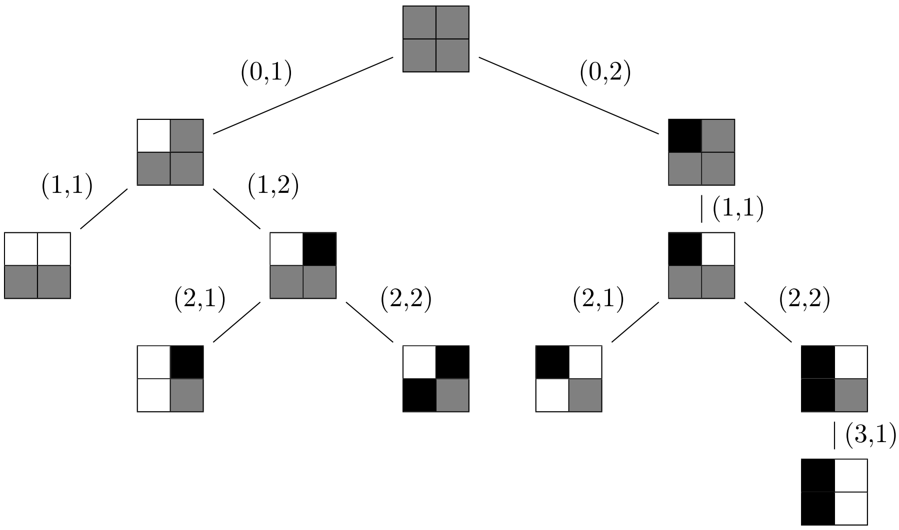
Enumeration des solutions

Question 1 :

Pour la grille libre suivante, dessiner l'arbre des appels récursifs générés par l'appel `Enumeration_Rec(0, 1)` ou `Enumeration_Rec(0, 2)` en indiquant la grille obtenue à chaque appel. Préciser (en une phrase) comment cet arbre est exploré.



Le parcours effectué est un parcours préfixe :



Question 2 :

Analyser la complexité de l'appel `[Enumeration_Rec(0; 1)` ou `Enumeration_Rec(0; 2)]` en fonction du nombre p de cases libres de M.

Nous remarquons qu'il y a au plus deux appels à la fonction `Enumeration_Rec()` par case libre. La complexité pire cas est atteinte lorsqu'il faut colorier toutes les cases en noir. On obtient donc une complexité en $\theta(2^p)$

Partie théorique :

Cas du vecteur libre :

Question 3

Si on connaît toutes les valeurs $T(j;l)$ pour $j \in \{0, \ldots, m-1\}$ et $l \in \{0, \ldots, k\}$ comment déterminer si le vecteur V peut être colorié en respectant L ?

Pour savoir si le vecteur V peut être colorié en respectant L nous devons regarder la valeur retournée par $T(m-1, k)$ car cette valeur nous indique s'il est possible de réaliser un coloriage de $V[0, \ldots, m-1] = V$ en respectant la séquence $(L_1, \ldots, L_k) = L$.

Question 4

Montrer que $T(j, 0) =$ vrai pour tout $j \in \{0, \ldots, m-1\}$.

$T(j, 0)$ retourne vrai s'il est possible de réaliser un coloriage de $V[0, \ldots, j]$ en respectant la séquence $(L_1, \ldots, L_0) = \emptyset$ soit une séquence vide. Il n'y a donc aucune contrainte à respecter sur les couleurs de chacune des cases de V. Ainsi $T(j, 0) =$ vrai pour $j \in \{0, \ldots, m-1\}$.

Question 5

Montrer que :

1. $T(L_1 - 1, 1) =$ vrai
2. $T(j, l) =$ faux pour tout $l \geq 1$ et $j < L_l - 1$.
3. $T(j, l) =$ faux pour tout $l \geq 2$ et $j \leq L_l - 1$.

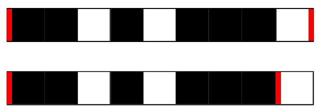
Notez que, pour ce dernier cas, $T(j, l) =$ faux même si $j \leq 0$.

1. $T(L_1 - 1, 1) =$ vrai indique qu'il est possible de réaliser un coloriage de $V[0, \ldots, L_1 - 1]$ en respectant la séquence de $(L_1, \ldots, L_1) = L_1$.
   Ceci est vrai car V est de taille L_1, et nous devions la remplir avec une séquence de taille L_1. La contrainte est donc de colorier toutes les cases $V = [V_0, \ldots, L_1 - 1]$ en noir.
2. $T(j, l) =$ faux pour tout $l \geq 1$ et $j < L_l - 1$ car nous cherchons à colorier $V[0, \ldots, j]$ avec une séquence $L = (L_1, \ldots, L_l)$ où $l \geq 1$. Puisque $j < L_l - 1$ nous avons au mieux $V[0, \ldots, L_l - 2]$ soit $L_l - 1$ cases. Ce qui n'est pas suffisant pour contenir la séquence L_l. Même si les autres séquences peuvent rentrer dans V, L_l n'en ferait pas partie. Rendant ainsi cette égalité toujours vrai.
3. $T(j, l) =$ faux pour tout $l \geq 2$ et $j \leq L_l - 1$ considère au plus L_l cases.
   La coloration des L_l cases correspondant au dernier bloc remplit au minimum toutes les cases de V, il n'est donc pas possible de colorier les cases correspondant aux séquences $(L_1, \ldots, L_{l-2})$ car $l \geq 2$.

Question 6

Si à présent $l \geq 1$ et $j \geq L_1 - 1$, donner les expressions de $T(j, l)$ si on fixe $V[j]$ à 1 ou si on fixe $V[j]$ à 2.

Au plus court, nous colorions les L_1 première cases en noir, puis la case suivante en blanc, puis les L_2 suivantes en noir. Ainsi de suite jusqu'à L_l. Nous avons finalement besoins de $L_1 + L_2 + \ldots L_l = \sum_{k=1}^{k=l} L_k$ cases que l'on coloriera en noir, avec une case blanche pour chaque bloc sauf le derniers. Soit un total de $l - 1$ cases blanches.
Finalement si nous fixons $V[j] = 1$, une case blanche de plus est nécessaire après L_l. Si nous fixons $V[j] = 2$, c'est à dire $V[0, \ldots, L_{l-2}]$ étant déjà en noire, nous n'avons pas besoin d'une case supplémentaire.
Voici un exemple :



Nous avons alors :
Pour $T[j] = 1$:

$$T(j, l) = \texttt{vrai} \iff j \geq (\sum_{k=1}^{l} L_k) + l$$

Pour $T[j] = 2$:

$$T(j, l) = \texttt{vrai} \iff j \geq (\sum_{k=1}^{l} L_k) + l - 1$$

Nous remarquons que si $T(j, l) = \texttt{vrai}$ avec $V[j] = 1$ alors $T(j, l) = \texttt{vrai}$ avec $V[j] = 2$. Par contraposée, si T n'est pas coloriable avec $V[j] = 2$, T n'est pas coloriable non plus avec $V[j] = 0$, il suffit de regarder si $T(j, l) = \texttt{vrai}$ avec $V[j] = 2$

Question 7

En déduire la formule de récurrence pour $T(j, l)$.

Nous en déduisons la fonction de récurrence suivante :

$$T(j, l) = \begin{cases} \texttt{vrai}, & \texttt{si } l = 0 \\ \texttt{faux}, & \texttt{si } l \geq 1 \texttt{ et } j < L_l - 1 \\ T(j - L_l - 1, l - 1), & \texttt{si } V[j] = 2 \texttt{ ou } V[j] = 0 \\ T(j - L_l, l - 1), & \texttt{sinon} \end{cases}$$

Cas du vecteur non libre :

Question 8

A quoi sert la ligne `Si` $(TT[j][l] \neq$ "`non visité`"$)$ `Retourne` $TT[j][l]$?

Cette ligne permet à la fonction de tester si elle a déjà calculé le cas testé. On appelle cela la mémoïsation. Elle permet de réduire la complexité de la fonction la rendant ainsi plus rapide au détriment d'une empreinte mémoire un peu plus grande.

Question 9

On veut prouver que la complexité de la fonction `TestVecteur_Rec` est polynomiale. Proposer pour cela une réécriture de la fonction `TestVecteur_Rec` de manière à ce que les appels récursifs au sein de la nouvelle fonction ne soient effectués que si la case $TT[j][l]$ est égale a "non visité". Cette fonction doit avoir la même complexité que la fonction initiale.

De la ligne 1 à la aligne 4, aucun appel récursif ne peut être effectué. A partir de la ligne 5 le code ne peut être exécuté que si nous ne sommes pas rentré dans le `Si` $(TT[j][l] \neq$ "`non visité`"$)$ `Retourne` $TT[j][l]$.
Donc les appels récursifs ne se font que si $TT[j][l]$ est égal à `non visité`.

Nous pouvons faire des appels récursif à `TestVecteur_Rec(`V`,`j`,`l`,`TT`)`, $\forall j, l < m$.
Grâce à la mémoïsation à partir du second appel avec les mêmes arguments, la complexité est en $\mathcal{O}(1)$. Nous pouvons donc considérer que le nombre d'appel est en $\mathcal{O}(m^2)$.

`TestSiAucun` étant en $\mathcal{O}(m)$, la complexité d'un sous-problème est aussi en $\mathcal{O}(m)$.
Au final, on a :

$$\begin{aligned}\texttt{Complexité} &= \texttt{nb\_sous\_probleme} * \texttt{complexite\_sans\_appel} \\ &= \mathcal{O}(m^2) * \mathcal{O}(m) \\ &= \mathcal{O}(m^3)\end{aligned}$$

Nous avons bien une complexité polynomiale.

Partie Expérimentale

Enumération

Question 10

Implémenter les fonctions `Compare_seq_ligne(i)` en $\mathcal{O}(n)$ et `Compare_seq_col(j)` en $\mathcal{O}(n)$ spécifiées dans la partie théorique. Implémenter la fonction `Enumeration` décrite dans la partie théorique. Testez-la sur les instances 0 à 16 (en autorisant une durée maximale d'exécution de 5min). Que constatez-vous ?

Nous constations que la durée d'exécution est supérieure à 5 minutes sauf pour les instances 0, 1 et 11 qui sont très petites.

```python
def Compare_seq_col(i):
    _C = C[i][:]
    _V = [line[i] for line in M]

    index = 0
    # elimination des cases blanches au debut du vecteur
    while (index < len(_V)) and _V[index] == 1:
        index += 1

    # pour chaque bloc
    for b in _C:
        count = 0
        # compter le nombre de cases noires
        while (index < len(_V)) and _V[index] == 2:
            count += 1
            index += 1

        # si le compte n'est pas bon la sequence n'est pas respectee
        if b != count:
            return False

        # passer les autres cases blanches
        while (index < len(_V)) and _V[index] == 1:
            index += 1
    return True
```

```python
def Compare_seq_ligne(i):
    _L = L[i][:]
    _V = M[i]

    index = 0
    # elimination des cases blanches au d but du vecteur
    while (index < len(_V)) and _V[index] == 1: # elimination
        des cases blanches
        index += 1

    # pour chaque bloc
    for b in _L:
        count = 0
        # compter le nombre de cases noires
        while (index < len(_V)) and _V[index] == 2:
            count += 1
            index += 1
        # si le compte n'est pas bon la sequence n'est pas
            respectee
        if b != count:
            return False

        # passer les autres cases blanches
        while (index < len(_V)) and _V[index] == 1:
            index += 1
    return True
```

Question 11

En suivant le principe de programmation dynamique décrit dans la fonction `TestVecteur_Rec` de la partie théorique, implémenter les fonctions `TestVecteurLigne_Rec` et `TestVecteurColonne_Rec` pour tester s'il existe un coloriage possible pour une ligne ou une colonne de la matrice (même si elle est partiellement coloriée)

```
def TestVecteur_Rec(V, _L, j, l, TT):
    if l == 0:
        return TestSiAucun(V, 0, j, 2)
    if l == 1 and j == _L[l-1] - 1:
        return TestSiAucun(V, 0, j, 1)
    if j <= _L[l-1] - 1:
        return False
    if TT[j][l-1] != None:
        return TT[j][l-1]
    if V[j] == 2:
        c1 = False
    else:
        c1 = TestVecteur_Rec(V, _L, j-1, l, TT)
    if not TestSiAucun(V,j-(_L[l-1]-1),j,1):
        c2 = False
    else:
        if V[j-_L[l-1]] == 2:
            c2 = False
        else:
            c2 = TestVecteur_Rec(V, _L, j-_L[l-1]-1, l-1, TT)
    TT[j][l-1] = c1 or c2
    return TT[j][l-1]
```

```
def TestVecteurLigne_Rec(i):
    _V = M[i] # recuperation de la ligne
    _L = L[i] # recuperation de la sequence
    j = len(_V) - 1
    l = len(_L)

    # initialisation du tableau TT
    TT = [[None for _ in range(l+1)] for __ in range(m)]

    return TestVecteur_Rec(_V, _L, j, l, TT)
```

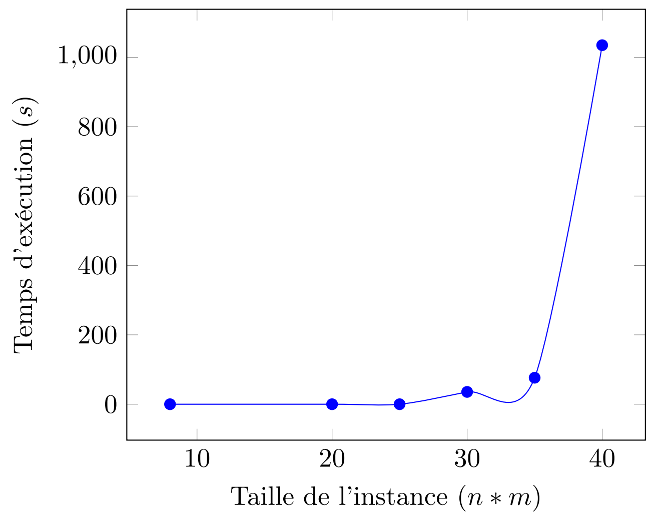
```
def TestVecteurColonne_Rec(i):
    _V = [line[i] for line in M] # recuperation de la colonne
    _C = C[i]   # recuperation de la sequence
    j = len(_V) - 1
    l = len(_C)

    # initialisation du tableau TT
    TT = [[None for _ in range(l)] for __ in range(m)]

    return TestVecteur_Rec(_V, _C, j, l, TT)
```

Question 13

Tester vos fonctions `Enumeration` et `TestVecteurLigne_Rec` sur le lot d'instances de vecteurs en arrêtant l'exécution après 30 minutes si l'exécution est trop longue. Pour chacune des fonctions (sur deux graphiques distincts), tracer une courbe permettant d'évaluer le temps d'exécution CPU en fonction de la taille du vecteur (attention le temps CPU peut être différent du temps d'exécution). Générer ces courbes à l'aide d'un tableur ou d'un outil graphique tel que `xgraphic` ou `gnuplot` (une documentation est en ligne sur le site du module). Vous pourrez ainsi analyser de façon critique la valeur expérimentale obtenue en fonction des complexités théoriques et aussi comparer les performances des algorithmes.

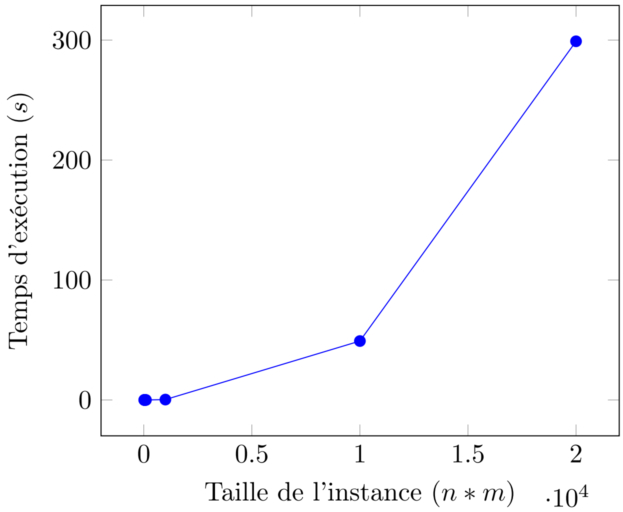
Graphique du temps d'exécution de la fonction `Enumeration` en fonction de la taille de l'instance.

Taille	temps (s)
8	0.00006
20	0.00608
25	0.00141
30	35.48397
35	76.34503
40	1,034.82797



Graphique du temps d'exécution de la fonction `TestVecteurLigne_Rec` en fonction de la taille de l'instance.

taille	temps
20	0.00019
30	0.00029
35	0.0003
40	0.00047
45	0.00045
50	0.00077
55	0.00101
60	0.0011
100	0.00311
1,000	0.32907
10,000	49.12354
20,000	299.00011



Le temps d'exécution de la fonction `Enumération` avec une taille de 45 cases est d'environ 1034 secondes (CPU). À partir d'une taille de 45 cases, le temps d'exécution dépasse les 30 minutes (réelles). Cela correspond à la complexité théorique qui est exponentielle. La fonction `TestVecteurLigne_Rec` est beaucoup plus rapide. Le temps d'exécution pour une instance de 40 cases est à peu près nul. La complexité théorique indique que cette fonction est moins coûteuse asymptotiquement, c'est déjà vrai pour une taille de 40 cases. Nous remarquons aussi que son temps d'exécution dépasse les 30 minutes pour les instances supérieur à une taille de 20000 cases.

Propagation

Question 14

Les annexes 1 et 2 donnent le pseudo-code des fonctions `PropagLigne` et `Propagation` qui décrivent comment utiliser les fonctions `TestVecteurLigne` et `TestVecteurColonne` pour déterminer des cases qui doivent nécessairement être coloriées en blanc ou en noir. Implémenter les fonctions `PropagLigne`, `PropagCol` et `Propagation` correspondantes.

```python
def PropagLigne(i, marque):
    nb = 0
    for j in range(m):
        if M[i][j] == 0:
            M[i][j] = 1
            c1 = TestVecteurLigne_Rec(i)
            M[i][j] = 2
            c2 = TestVecteurLigne_Rec(i)
            M[i][j] = 0
            if (not c1) and (not c2):
                return (False,nb)
            if c1 and (not c2):
                M[i][j] = 1
                if not marque[j] :
                    marque[j] = True
                    nb += 1
            if (not c1) and c2:
                M[i][j] = 2
                if not marque[j] :
                    marque[j] = True
                    nb += 1
    return (True,nb)
```

```python
def PropagCol(j, marque):
    nb = 0
    for i in range(n):
        if M[i][j] == 0:
            M[i][j] = 1
            c1 = TestVecteurColonne_Rec(j)
            M[i][j] = 2
            c2 = TestVecteurColonne_Rec(j)
            M[i][j] = 0
            if (not c1) and (not c2):
                return (False, nb)
            if c1 and (not c2):
                M[i][j] = 1
                if not marque[i] :
                    marque[i] = True
                    nb += 1
            if (not c1) and c2:
                M[i][j] = 2
                if not marque[i] :
                    marque[i] = True
                    nb += 1
    return (True,nb)
```

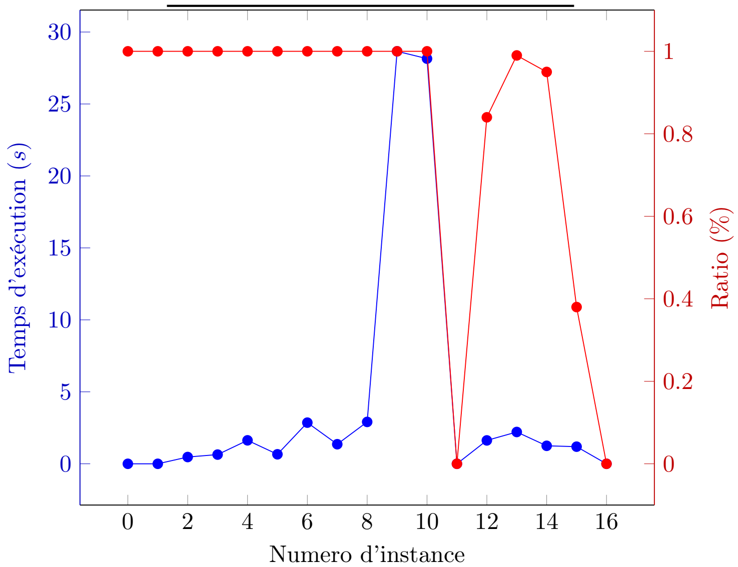
```python
def Propagation():
    nb = 0
    nbmL = n
    nbmC = m
    marqueL = [True for _ in range(n)]
    marqueC = [True for _ in range(m)]
    ok = True

    while ok and ((nbmL != 0) or (nbmC !=0)):
        i = 0
        while ok and (i < n):
            if marqueL[i]:
                (ok, nb) = PropagLigne(i, marqueC)
                nbmC += nb
                marqueL[i] = False
                nbmL -= 1
            i += 1
        j = 0
        while ok and (j < m):
            if marqueC[j] :
                (ok,nb) = PropagCol(j,marqueL)
                nbmL += nb
                marqueC[j] = False
                nbmC -= 1
            j += 1
    return ok
```

Question 15

Tester la fonction `Propagation` en indiquant son temps d'exécution ainsi que le pourcentage de cases coloriées pour les différentes instances.

D'après les instances fournies, on remarque que la fonction remplit généralement tout le tableau (de l'instance 0 à 10 puis la 13ième). Dans certains cas la fonction remplit à moitié ou pas du tout le tableau (instance 11 non comprit et les instances 12, 14 et 15 à moitié). Finalement, dans le cas de l'instance 16, bien qu'elle ne soit pas particulièrement grande (1750 cases) le temps d'exécution est largement supérieur aux autres.

instance	taille	ratio	temps
0	20	1	0
1	25	1	0
2	400	1	0.47
3	481	1	0.64
4	625	1	1.63
5	675	1	0.66
6	900	1	2.86
7	1,054	1	1.36
8	1,400	1	2.91
9	2,500	1	28.66
10	9,801	1	28.15
11	8	0	0
12	924	0.84	1.625
13	2,025	0.99	2.21
14	1,140	0.95	1.25
15	900	0.38	1.19
16	1,750	0	0



Question 16

Enchaîner les fonctions `Propagation` et `Enumeration` pour les instances fournis. Que constatez-vous ?

En enchaînant les deux fonctions, le temps d'exécution est fortement réduit comparé à l'exécution de `Enumeration` seule. Cela s'explique par le fait que le plus gros du travail est traité en temps polynomial et que seule la fin du problème a une complexité exponentielle. Par ailleurs, nous remarquons que le temps prit par la fonction n'est pas proportionnel à la taille du tableau. Par exemple pour une instance de 9801 cases nous avons mesuré un temps de 345 secondes CPU alors que pour une instance de 2500 cases nous avons mesuré un temps de 371 secondes CPU.