Algorithmique
– version 0.1

Dr M. GUEDJ

Table des Matières

1 Algorithmes sur tableaux

1.1 Un tableau est-il vide ?

Algorithm 1 EST_VIDE (t : tableau, n : taille du tableau)

```
if n = 0 then
    return True
else
    return False
end if
```

Complexité : $O(1)$.

1.2 Afficher les éléments d'un tableau

Algorithm 2 AFFICHER_TABLEAU (t : tableau, n : taille du tableau)

```
for i ← 0, ..., n − 1 do
    print(t[i])
end for
```

Complexité : $O(n)$.

1.3 Afficher les éléments positifs d'un tableau

Algorithm 3 AFFICHER_POSITIFS (t : tableau, n : taille du tableau)

```
for i ← 0, ..., n − 1 do
    if t[i] ≥ 0 then
        print(t[i])
    end if
end for
```

Complexité : $O(n)$.

1.4 Retourner l'éléments maximum d'un tableau

Algorithm 4 MAXIMUM (t : tableau, n : taille du tableau)

```
                                        ▷ On suppose n > 0
max ← t[0]
for i ← 1, ..., n − 1 do
    if t[i] ≥ max then
        max ← t[i]
    end if
end for
return max
```

Complexité : $O(n)$.

1.5 Retourner l'indice de l'éléments maximum d'un tableau

Algorithm 5 INIDICE_MAXIMUM (t : tableau, n : taille du tableau)

```
                                        ▷ On suppose n > 0
max ← t[0]
iMax ← 0
for i ← 1, ..., n − 1 do
    if t[i] ≥ max then
        max ← t[i]
        iMax ← i
    end if
end for
return iMax
```

Complexité : $O(n)$.

1.6 Retourner la somme des éléments d'un tableau

Algorithm 6 SOMME (t : tableau, n : taille du tableau)

```
res ← 0
for i ← 0, ..., n − 1 do
    res ← res + t[i]
end for
return res
```

Complexité : $O(n)$.

1.7 Rechercher un élément dans un tableau

Algorithm 7 RECHERCHE (t : tableau, n : taille du tableau, x : élément)

for $i \leftarrow 0, ..., n-1$ **do**
    if $t[i] = x$ **then**
        return True
    end if
end for
return False

Complexité : $O(n)$.

2 Algorithmes sur matrices

2.1 Afficher les éléments d'une matrice

Algorithm 8 AFFICHER_MATRICE (A : matrice $n \times m$)

```
for i ← 0, ..., n − 1 do                ▷ parcours sur les lignes
    for j ← 0, ..., m − 1 do            ▷ parcours sur les colonnes
        print(A_{i,j})
    end for
    print(saut de ligne)
end for
```

Complexité : $O(n \times m)$.
Cas d'une matrice carré $n \times n$: $O(n^2)$ (complexité linéaire !).

2.2 Additionner deux matrices

Algorithm 9 ADDITIONNER (A, B : matrice $n \times m$)

```
C ← matrice n × m
for i ← 0, ..., n − 1 do
    for j ← 0, ..., m − 1 do
        C_{i,j} ← A_{i,j} + B_{i,j}
    end for
end for
return C
```

Complexité : $O(n \times m)$.

2.3 La matrice est-telle diagonale ?

Rappel : la matrice carré $n \times n$, soit M, est diagonale si :

$$\forall i, j \in \{0, ..., n-1\}, i \neq j \Rightarrow M_{i,j} = 0$$

Algorithm 10 EST_DIAGONALE (M : matrice $n \times n$)

```
for i ← 0, ..., n − 1 do
    for j ← 0, ..., n − 1 do
        if i ≠ j et M_{i,j} ≠ 0 then
            return False
        end if
    end for
end for
return True
```

Complexité : $O(n^2)$.

3 Complexité en temps (cas le pire)

3.1 Approximation asymptotique

Définition 1 (Notation grand O). *Soient f et g deux fonctions de $\mathbb{N} \to \mathbb{R}^+$.*
$f \in O(g)$ si :

- $\exists K \in \mathbb{R}^{*+}$;
- $\exists n_0 \in \mathbb{N}$;

et

$$\forall n \geq n_0, \; f(n) \leq K.g(n)$$

($f(n) \leq K.g(n)$ à partir d'un certain rang).

Exemples

- $7n - 3 \in O(n)$
- $7n - 3 \in O(n^2)$
- $987654321 + 10n^2 \in O(n^2)$

Remarque Le but est de trouver l'approximation la plus petite possible.

3.2 Complexités en temps classiques

Complexité	Notation asymptotique	Exemple
Logarithmique	$O(\log n)$	Recherche dichotomique dans un tableau trié.
Linéaire	$O(n)$	Recherche séquentielle dans un tableau.
Quasi-linéaire	$O(n \log n)$	Tri fusion.
Quadratique	$O(n^2)$	Tri sélection; tri à bulles.
Polynomiale	$O(n^k), k \geq 0$	...
Exponentielle	$O(k^n), k > 1$	Algorithme récursif pour Fibonacci.
Factorielle	$O(n!)$	Résolution des n-reines par *backtracking*.

4 Tris quadratiques

4.1 Algorithme d'échange

Algorithm 11 ECHANGER(t : tableau, i, j : entiers)

$tmp \leftarrow t[i]$
$t[i] \leftarrow t[j]$
$t[j] \leftarrow tmp$

4.2 Tri par sélection

Algorithm 12 TRI_SELECTION(t : tableau, n : taille du tableau)

for $i \leftarrow 0, ..., n-2$ **do**
    $i_{min} \leftarrow$ INDICE_MIN_SOUS_TAB$(t, i, n-1)$
    ECHANGER(t, i, i_{min})
end for

Algorithm 13 INDICE_MIN_SOUS_TAB(t : tableau, a, b : entiers)

1: $i_{min} \leftarrow a$
2: **for** $i \leftarrow a+1, ..., b$ **do**
3:     **if** $t[i] < t[i_{min}]$ **then**
4:         $i_{min} \leftarrow i$
5:     **end if**
6: **end for**
7: **return** i_{min}

Théorème 2. *La complexité de* TRI_SELECTION *est en* $O(n^2)$.

Théorème 3. *Le nombre de comparaisons de* TRI_SELECTION *est en* $O(n^2)$.

Preuve. Calcul du nombre de comparaisons $\mathcal{C}(n)$ (ligne 3 de l'algorithme 13).

$$\mathcal{C}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$\mathcal{C}(n) = \sum_{i=0}^{n-2} \Big((n-1) - (i+1) + 1\Big)$$

$$\mathcal{C}(n) = \sum_{i=0}^{n-2} \big(n - 1 - i - 1 + 1\big)$$

$$\mathcal{C}(n) = \sum_{i=0}^{n-2} \big(n - 1 - i\big)$$

$$\mathcal{C}(n) = \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

$$\mathcal{C}(n) = (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

$$\mathcal{C}(n) = (n-1)(n-1) - \sum_{i=0}^{n-2} i$$

$$\mathcal{C}(n) = (n-1)^2 - \sum_{i=0}^{n-2} i$$

$$\mathcal{C}(n) = (n-1)^2 - \frac{(n-2)(n-1)}{2}$$

$$\mathcal{C}(n) = (n-1)\Big((n-1) - \frac{(n-2)}{2}\Big)$$

$$\mathcal{C}(n) = (n-1)\Big(\frac{2.(n-1) - (n-2)}{2}\Big)$$

$$\mathcal{C}(n) = (n-1)\Big(\frac{2n - 2 - n + 2}{2}\Big)$$

$$\mathcal{C}(n) = (n-1)\frac{n}{2}$$

$$\mathcal{C}(n) = \frac{n^2}{2} - \frac{n}{2} \in O(n^2)$$

$\square$

4.3 Tri à bulles

Algorithm 14 TRI_BULLES(t : tableau, n : taille du tableau)

```
1: for i ← n − 1, ..., 1 do
2:     for j ← 0, ..., i − 1 do
3:         if t[j + 1] < t[j] then
4:             ECHANGER(t, j + 1, j)
5:         end if
6:     end for
7: end for
```

Théorème 4. *La complexité de* TRI_BULLES *est en* $O(n^2)$.

Théorème 5. *Le nombre de comparaisons de* TRI_BULLES *est en* $O(n^2)$.

Preuve. Calcul du nombre de comparaisons $\mathcal{C}(n)$ (ligne 4 de Algorithme 14).

$$\mathcal{C}(n) = \sum_{i=n-1}^{1} \sum_{j=0}^{i-1} 1 = \sum_{i=n-1}^{1} (i - 1 - 0 + 1) = \sum_{i=n-1}^{1} i$$

$$\mathcal{C}(n) = \frac{(n-1+1)(n-1)}{2} = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \in O(n^2)$$

□

5 Récursivité

5.1 Considérations sur la récursivité

La version itérative d'un traitement est souvent à préférer.

En effet :

- Un dépassement de pile (*stack overflow*) peut se produire ;
- L'exécution d'une version récursive d'un algorithme est généralement un peu moins rapide que celle de la version itérative correspondante ; et ce même si le nombre d'instructions est le même (à cause de la gestion des appels de fonction) ;
- Un algorithme récursif (naïf) peut conduire à exécuter bien plus d'instructions que la version itérative correspondante (cas du calcul de la suite de Fibonacci).

En revanche, la récursivité peut être adaptée dans certains cas.

En effet :

- Sur des structures de données naturellement récursives, il est plus facile d'écrire des algorithmes récursifs qu'itératifs ;
- Certains algorithmes sont, en outre, difficiles à écrire en itératif.

5.2 Exemple 1 : la fonction factorielle

Définition 6 (fonction factorielle)**.** *La fonction factorielle est définie, sur* $\mathbb{N}$*, par :*

$$\begin{cases} 0! = 1 \\ n! = \Pi_{i=1}^{n} i = n \times (n-1) \times ... \times 2 \times 1 & si\ n \geq 1 \end{cases}$$

Définition 7 (définition récursive de la fonction factorielle)**.**

$$n! = \begin{cases} 0 & si\ n = 0 \\ n \times (n-1)! & si\ n \geq 1 \end{cases}$$

Algorithm 15 FACT($n \in \mathbb{N}$)

if $n = 0$ **then**
    return 1
else
    return $n \times$ FACT$(n-1)$
end if

Complexité : $O(n)$

Algorithm 16 FACT_IT($n \in \mathbb{N}$)

$res \leftarrow 1$
for $i \leftarrow 1, ..., n$ **do**
    $res \leftarrow res \times i$
end for
return res

Complexité : $O(n)$

5.3 Exemple 2 : la suite de Fibonacci

Définition 8 (suite de Fibonacci)**.**

$$F_n = \begin{cases} 0 & si\ n = 0 \\ 1 & si\ n = 1 \\ F_{n-1} + F_{n-2} & si\ n \geq 2 \end{cases}$$

Algorithm 17 FIBO($n \in \mathbb{N}$)

if $n = 0$ **then**
    return 0
else if $n = 1$ **then**
    return 1
else
    return FIBO$(n-1)$ + FIBO$(n-2)$
end if

6 Recherche Dichotomique

Algorithm 18 DICHO_INIT(t : tableau, n : taille du tableau, x : élément)

$d \leftarrow 0$
$f \leftarrow n - 1$
return DICHO(t, d, f, x)

Algorithm 19 DICHO(t : tableau, d, f : indices, x : élément)

if $d > f$ **then return** -1 **end if** ▷ Non trouvé.
if $f = d$ **then**
    if $t[d] = x$ **then return** d **else return** -1 **end if**
end if
$m \leftarrow E(\frac{d+f}{2})$ ▷ Partie entière.
if $t[m] = x$ **then**
    return m
end if
if $t[m] < x$ **then**
    return DICHO$(t, m+1, f, x)$
else
    return DICHO$(t, d, m-1, x)$
end if

Théorème 9. *La complexité de* DICHO *est en* $O(\log n)$.

Preuve (d'approximation). Soit $\mathcal{C}(n)$ le nombre de comparaisons pour une instance de taille n. On a,

$$\mathcal{C}(n) = \gamma + \mathcal{C}(\frac{n}{2}) \quad (\gamma \text{ constante})$$

La deuxième instance appelée vérifie :

$$\mathcal{C}(\frac{n}{2}) = \gamma + \mathcal{C}(\frac{n}{4})$$

D'où,

$$\mathcal{C}(n) = \gamma + \mathcal{C}(\frac{n}{2}) = \gamma + \left(\gamma + \mathcal{C}(\frac{n}{4})\right) = 2\gamma + \mathcal{C}(\frac{n}{4})$$

Soit,

$$\underline{\mathcal{C}(n) = 2\gamma + \mathcal{C}(\frac{n}{2^2})}$$

La troisième instance appelée vérifie :

$$\mathcal{C}(\frac{n}{2^2}) = \gamma + \mathcal{C}(\frac{n}{2^3})$$

D'où,

$$\underline{\mathcal{C}(n) = 3\gamma + \mathcal{C}(\frac{n}{2^3})}$$

En itérant, $\mathcal{C}(n)$ s'écrit :

$$\underline{\mathcal{C}(n) = k\gamma + \mathcal{C}(\frac{n}{2^k})}$$

Où k correspond au $(k+1)$-ième appel récursif.

On a* :

$$\frac{n}{2^k} = 1 \Rightarrow \underline{\mathcal{C}(\frac{n}{2^k}) = \alpha \in \mathbb{N}}$$

Et :

$$\frac{n}{2^k} = 1 \iff n = 2^k \iff \underline{\log_2 n = k}$$

$C(n)$ s'écrit alors :

$$\mathcal{C}(n) = k\gamma + \mathcal{C}(\frac{n}{2^k}) = \underline{\log_2(n)\gamma + \alpha \in O(\log n)}$$

$\square$

(*) Le cas $\frac{n}{2^k} < 0$ (i.e. le cas d'une taille négative d'instance), prévu par l'algorithme (ligne 1 de l'algorithme 19), est laissé en exercice. Vous devriez trouver (si $n > 0$) :

$$\frac{n}{2^k} < 0 \Rightarrow \frac{n}{2^{k-1}} = 2 \iff n = 2.2^{k-1} = 2^k$$

(Considérer l'appel parent).

7 Tri Fusion

Algorithm 20 TRI_FUSION(lst : liste de taille n)

if $n = 1$ **return** lst **end if**
$m = E(n/2)$ ▷ Partie entière.
$lst_1 \leftarrow$ TRI_FUSION($lst[0 \rightarrow m-1]$)
$lst_2 \leftarrow$ TRI_FUSION($lst[m \rightarrow n-1]$)
return FUSION(lst_1, lst_2)

Algorithm 21 FUSION(lst_1 : liste de taille n_1, lst_2 : liste de taille n_2)

1: $res \leftarrow$ Liste vide
2: **while** non (lst_1 vide et lst_2 vide) **do**
3:     **if** lst_1 est vide **then**
4:         **return** $res + lst_2$
5:     **end if**
6:     **if** lst_2 est vide **then**
7:         **return** $res + lst_1$
8:     **end if**
9:     **if head**(lst_1) $\leq$ **head**(lst_2) **then**
10:         $res \leftarrow res +$ [**head**(lst_1)]
11:         $lst_1 \leftarrow$ **tail**(lst_1)
12:     **else**
13:         $res \leftarrow res +$ [**head**(lst_2)]
14:         $lst_2 \leftarrow$ **tail**(lst_2)
15:     **end if**
16: **end while**
17: **return** res

Exercice. La ligne 17 de l'algorithme 21 n'est jamais atteinte. Pourquoi ?

Théorème 10. *La complexité de* TRI_FUSION *est en* $O(n \log n)$.

Preuve (réduite aux cas des puissances de deux). Soit p une puissance de 2.

$$\mathcal{C}(p) = 1 + 2.\,\mathcal{C}(\frac{p}{2}) + \gamma.p$$

Où γ est une constante. On a de même :

$$\mathcal{C}(\frac{p}{2}) = 1 + 2.\,\mathcal{C}(\frac{p}{4}) + \gamma.\frac{p}{2}$$

Soit :

$$\mathcal{C}(p) = 1 + 2(1 + 2.\,\mathcal{C}(\frac{p}{4}) + \gamma.\frac{p}{2}) + \gamma.p$$

$$\mathcal{C}(p) = 1 + 2 + 4.\,\mathcal{C}(\frac{p}{4}) + \gamma.p + \gamma.p$$

$$\mathcal{C}(p) = 4.\,\mathcal{C}(\frac{p}{4}) + 2\gamma.p + 3$$

$$\underline{\mathcal{C}(p) = 2^2.\,\mathcal{C}(\frac{p}{2^2}) + 2\gamma.p + (2+1)}$$

$$\mathcal{C}(\frac{p}{2^2}) = 1 + 2.\,\mathcal{C}(\frac{p}{2^3}) + \gamma.\frac{p}{2^2}$$

$$\mathcal{C}(p) = 2^2.(1 + 2.\,\mathcal{C}(\frac{p}{2^3}) + \gamma.\frac{p}{2^2}) + 2\gamma.p + (2+1)$$

$$\mathcal{C}(p) = 2^2 + 2^2.2.\,\mathcal{C}(\frac{p}{2^3}) + \gamma.p + 2\gamma.p + (2+1)$$

$$\underline{\mathcal{C}(p) = 2^3.\,\mathcal{C}(\frac{p}{2^3}) + 3\gamma.p + (2^2+2+1)}$$

$$\mathcal{C}(\frac{p}{2^3}) = 1 + 2.\,\mathcal{C}(\frac{p}{2^4}) + \gamma.\frac{p}{2^3}$$

$$\mathcal{C}(p) = 2^3.(1 + 2.\,\mathcal{C}(\frac{p}{2^4}) + \gamma.\frac{p}{2^3}) + 3\gamma.p + (2^2+2+1)$$

$$\mathcal{C}(p) = 2^3 + 2^3.2.\,\mathcal{C}(\frac{p}{2^4}) + 2^3.\gamma.\frac{p}{2^3} + 3\gamma.p + (2^2+2+1)$$

$$\mathcal{C}(p) = 2^3 + 2^4.\,\mathcal{C}(\frac{p}{2^4}) + \gamma.p + 3\gamma.p + (2^2+2+1)$$

$$\underline{\mathcal{C}(p) = 2^4.\,\mathcal{C}(\frac{p}{2^4}) + 4\gamma.p + (2^3+2^2+2+1)}$$

En itérant,

$$\underline{\mathcal{C}(p) = 2^t.\,\mathcal{C}(\frac{p}{2^t}) + t\gamma.p + (2^t + ... + 2^2+2+1)}$$

On a :

$$2^t + ... + 2^2 + 2 + 1 = 2^{t+1} - 1$$

D'où,

$$\underline{\mathcal{C}(p) = 2^t.\,\mathcal{C}(\frac{p}{2^t}) + t\gamma.p + 2^{t+1} - 1}$$

On a :

$$\frac{p}{2^t} = 1 \iff p = 2^t \iff t = \log_2 p$$

Et $\mathcal{C}(1) = 1$; d'où :

$$\mathcal{C}(p) = 2^{\log_2 p}.1 + \log_2(p)\gamma.p + 2^{\log_2(p)+1} - 1$$

$$\mathcal{C}(p) = p + p.\log_2(p).\gamma + 2^{\log_2(p)}.2 + -1$$

$$\mathcal{C}(p) = p + p.\log_2(p).\gamma + 2.p - 1$$

$$\underline{\mathcal{C}(p) = p.\log_2(p).\gamma + 3.p - 1} \in O(p \log p)$$

$\square$

8 Arbres Binaires et ABR

8.1 Définitions préliminaires

Définition 11 (Type `Noeud_Binaire`). *Le type* `Noeud_Binaire` *est un triplet* (id, val, f_g, f_d) *tel que :*

(i) id est un identifiant à valeur dans $\mathbb{N} \cup \{-1\}$ *;*

(ii) $val \in V$ *(valeur du noeud) ; (par exemple* $V = \mathbb{R}$*) ;*

(iii) f_g *(fils gauche) est de type* `Noeud_Binaire` *;*

(iv) De même pour f_d *(fils droit).*

On considère le prédicat $Null$, définit sur `Noeud_Binaire` par :

$$Null(n) \iff n.id = -1$$

Soint un ensemble d'éléments de type `Noeud_Binaire`, on pose :

$$parent(s) := \{s' : s'.f_g = s \text{ ou } s'.f_d = s\}$$

Définition 12 (Arbre binaire). *Un arbre binaire* $\mathcal{A}$ *est un ensemble d'éléments de type Noeud_Binaire vérifiant :*

(i) (Existence d'une racine unique)

$$\exists ! s \in \mathcal{A}, parent(s) = \emptyset$$

(ii) (Fils gauche et droit distincts)

$$\forall s \in \mathcal{A}, s.f_g.id \neq s.f_d.id \text{ ou } s.f_g.id = s.f_d.id = -1$$

(iii) (Parent unique)

$$\forall s \in \mathcal{A}, |parent(s)| \leq 1$$

On pose :

$$A_{\mathcal{A}} := \{(x, y) : x, y \in \mathcal{A}, x \in parent(y) \text{ ou } y \in parent(x)\}$$

On pose $G_{\mathcal{A}}$ le graphe non orienté associé à l'arbre binaire $\mathcal{A}$, défini par : $G_{\mathcal{A}} := (S, A)$, tel qu'il existe une bijections $\sigma : S \to \mathcal{A}$ assurant : $\forall x, y \in S$,

$$(x, y) \in A \Rightarrow (\sigma(x), \sigma(y)) \in A_{\mathcal{A}}$$

Théorème 13. *La fonction $\alpha : A \to A_{\mathcal{A}}$ définie par : $(x,y) \mapsto (\sigma(x), \sigma(y))$ est une bijection.*

Preuve. 1. α est injective.

$$\begin{aligned} & \alpha(x,y) = \alpha(x',y') \\ \iff & (\sigma(x), \sigma(y)) = (\sigma(x'), \sigma(y')) \\ \iff & \big(\sigma(x) = \sigma(x') \text{ et } \sigma(y) = \sigma(y')\big) \\ \iff & \big(x = x' \text{ et } y = y'\big) \end{aligned}$$

(en utilisant le fait que σ est bijective donc en particulier injective).

2. α est surjective. Soit $(x,y) \in A_{\mathcal{A}}$. $\big(\sigma^{-1}(x), \sigma^{-1}(y)\big)$ est un antécédant de (x,y).

□

Définition 14 (Hauteur d'un noeud)**.** *Soit $\mathcal{A}$ un arbre binaire, la hauteur d'un noeud $s \in \mathcal{A}$, notée $height(s)$, est définie par : $\forall s \in \mathcal{A}$,*

(i) $height(s) = 0$ si s est racine de $\mathcal{A}$;

(ii) $height(s) = 0$ si $Null(s)$;

(iii) $height(s) = height(s') + 1$, où $parent(s) = \{s'\}$, sinon.

Théorème 15. *Le graphe non orienté associé à un arbre binaire est connexe et sans cycle.*

Définition 16 (ABR)**.** *Un arbre binaire de recherche (ABR) est soit un arbre vide ; soit un arbre binaire vérifiant, pour tout noeud s :*

- $\forall s' \in G(s), s'.val \leq s.val$;
- $\forall s' \in D(s), s.val < s'.val$;

où $G(s)$ (resp. $D(s)$) est le sous-arbre gauche (resp. droit) du noeud s.

8.2 Recherche dans un ABR

Algorithm 22 RECHERCHE_ABR ($s \in \mathcal{A}$, $x \in V$: valeur recherchée)

if $Null(s)$ **then**
    return False
else if $s.val = x$ **then**
    return True
else if $s.val > x$ **then**
    return RECHERCHE_ABR($s.f_g, x$)
else
    return RECHERCHE_ABR($s.f_d, x$)
end if

Complexité : $O(\log n)$ en moyenne.

8.3 Parcours infixe dans un arbre binaire

Algorithm 23 INFIXE ($s \in \mathcal{A}$)

if $Null(s)$ **then** **return** `None` **end if**
if not $Null(s)$ **then**
    INFIXE(s_g)
end if
`print`(s)
if not $Null(s_d)$ **then**
    INFIXE(s_d)
end if

Complexité : $O(n)$.

Théorème 17. *Le parcours infixe d'un ABR donne une séquence des noeuds triés selon l'ordre croissant des valeurs.*

Preuve. (Par récurrence sur la taille de l'ABR). Si $|\mathcal{A}| = 1$, alors la proposition est vraie.

Supposons que, pour tout ABR de taille $m \leq k$, la proposition soit vraie. Considérons un ABR de taille $k + 1$; alors la séquence affichée est de la forme :

séquence affichée par INFIXE($s.f_g$) ; s ; séquence affichée par INFIXE($s.f_d$).

Par définition d'un ABR,

- $\forall s' \in G(s), s'.val \leq s.val$;
- $\forall s' \in D(s), s.val < s'.val$.

D'autre part, l'hypothèse de récurrence nous assure que la séquence affichée par INFIXE($s.f_g$) (resp. INFIXE($s.f_d$)) est conforme à la proposition. $\square$

9 Représentation des graphes

9.1 Considérations préliminaires

Soit un graphe $G = (S, A)$ tel que : $|S| = n$ et $|A| = m$ (avec $n, m \in \mathbb{N}$).
Les sommets de G sont numérotés de 0 à $n - 1$.

9.2 Représentation par matrice d'adjacence

Définition 18. *La matrice d'adjacence du graphe G, soit M, est une matrice booléenne de type $n \times n$ vérifiant :*

$$M_{i,j} = \begin{cases} 1 & \text{si } i \text{ et } j \text{ sont adjacents} \\ 0 & \text{sinon} \end{cases}$$

Pour $i, j \in \{0, ..., n-1\}$, si s_i est le i-ième sommet, et si s_j est le j-ième sommet, alors :

$$M_{i,j} = 1 \iff (s_i, s_j) \in A$$

9.3 Représentation par liste d'adjacence

Définition 19. *La liste d'adjacence du graphe G, soit* **succ**, *est une liste indexée par les sommets de G, et telle que :*

$$\forall s \in S, \ \mathbf{succ}(s) = \{s' : (s, s') \in A\}$$

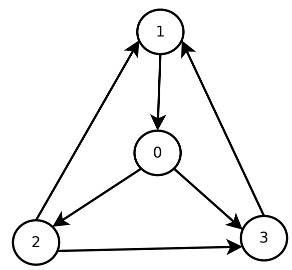
Autrement dit, $\forall s \in S$, $\mathbf{succ}(s)$ est l'ensemble des sommets adjacents à s.

9.4 Exemple

Soit le graphe $G = (S, A)$, défini par :

$$\begin{cases} S = \{0, 1, 2, 3\} \\ A = \{(0,2), (0,3), (1,0), (2,1), (2,3), (3,1)\} \end{cases}$$

Un tel graphe peut être représenté comme suit :



Les représentatons par matrice et liste d'adjacence sont données ci-après.

Matrice d'adjacence	**Liste d'adjacence**
$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$	0: $\{2,3\}$; 1: $\{0\}$; 2: $\{1,3\}$; 3: $\{1\}$

9.5 Espace mémoire

Matrice d'adjacence	**Liste d'adjacence**
$O(n^2)$	$O(n+m)$

9.6 Complexité de quelques opérations

Opérations	**Matrice d'adjacence**	**Liste d'adjacence**
Tester l'existence d'un arc $s \to s'$	$O(1)$	$O(\lvert \mathbf{succ}(s) \rvert)$
Retourner les sommets adjacents à un sommet	$O(n)$	$O(1)$
Parcourir l'ensemble des arcs	$O(n^2)$	$O(m)$

Lemme 20.

$$\sum_{s \in S} |\mathbf{succ}(s)| = |A|$$

9.7 Choix d'utilisation

- D'une manière générale, on considère que si le graphe a "peu" d'arêtes, il est plus intéressant d'utiliser une représentation par liste d'adjacence, plutôt que par matrice d'adjacence (qui contiendrait alors beaucoup de 0).
- Mais si le graphe a "beaucoup" d'arêtes, il est plus intéressant d'utiliser une matrice d'adjacence.

10 Parcours en largeur de graphes (***Breadth First Search***)

Algorithm 24 BFS($G = (S, \mathbf{succ}), s_0$)

done $\leftarrow$ Liste vide
todo $\leftarrow$ File vide
todo.**enfiler**(s_0)
done.**append**(s_0)
while *todo* non vide **do**
    $s \leftarrow$ *todo*.**defiler**()
    for $s' \in \mathbf{succ}(s)$ **do**
        if $s' \notin$ *done* **then**
            todo.**enfiler**(s')
            done.**append**(s')
        end if
    end for
end while
return *done*

11 Problème de l'arrêt

Définition 21 (ARRÊT).
Entrées :

1. <Prog> *: le code source d'un programme* Prog *;*
2. x *: une entrée pour* Prog*.*

Sortie : Prog(x) *s'arrête-t-il ?*

Théorème 22 (Turing). ARRÊT *est indécidable.*

Preuve. (Par l'absurde). On suppose qu'ARRÊT est décidable ; i.e. il existe un programme, soit Halt, qui décide le problème de l'arrêt ; i.e., pour tout programme Prog de code source <Prog>, pour toute entrée x de Prog :

- Prog(x) s'arrête $\Longleftrightarrow$ Halt(<Prog>, x) répond Vrai ;
- Prog(x) ne s'arrête pas $\Longleftrightarrow$ Halt(<Prog>, x) répond Faux.

On considère le programme Diagonale ci-après :

```
Diagonale(y):
    Si Halt(y, y) = Vrai :
        opérer une boucle infinie
    Sinon
        retourner "toto"
    Fin Si
```

Nous considérons l'exécution : Diagonale(<Diagonale>).

(a) Cas 1 : Halt(<Diagonale>, <Diagonale>) = Vrai

D'après le code de Diagonale, il suit que Diagonale(<Diagonale>) ne s'arrête pas, donc que Halt(<Diagonale>, <Diagonale>) répond Faux.

(b) Cas 2 : Halt(<Diagonale>, <Diagonale>) = Faux

D'après le code de Diagonale, il suit que Diagonale(<Diagonale>) s'arrête, donc que Halt(<Diagonale>, <Diagonale>) répond Vrai.

$\square$

12 Approche des problèmes d'optimisation

Nous considérons une veille radio que l'on trouve dans une brocante ; elle comporte une molette : A ; et 3 boutons : B, C et D.

- A permet de capter des fréquences (100 fréquences disponibles) ;
- B, C et D sont trois boutons qui peuvent prendre 10 valeurs chacune – mais dont on ne connait la signification.

Une solution est la donnée d'un quadruplet (a, b, c, d) où a, b, c et d indiquent des positions, de la molette et des différents boutons.

Au total : $100 \times 10 \times 10 \times 10 = 100\ 000$ solutions possibles.

Le but est de trouver une station diffusant une chanson que l'on aime bien, et avec une bonne qualité de diffusion.

Pour chaque solution, i.e. chaque possibilité de quadruplet, on donne une note.

Exemple de notation.

On entend :

- Seulement des bruits de grésillement $\rightarrow 0/20$;
- Des voix, avec beaucoup de grésillement $\rightarrow 5/20$;
- Une chanson avec grésillement $\rightarrow 12/20$;
- Des voix ; avec bonne qualité d'écoute $\rightarrow 14/20$;
- Une chanson « sympathique » ; avec bonne qualité d'écoute $\rightarrow 17/20$;
- Une chanson que l'on aime davantage ; avec bonne qualité d'écoute $\rightarrow 19/20$.

Nous nous faisons face à un problème d'**optimisation** ; il s'agit, en effet, de trouver une solution qui **maximise** la note.

Etant donné le nombre de solutions possibles ; on utilise alors une heuristique de résolution. (Tester 100 000 configurations possibles n'est pas acceptable pour un humain).

13 Algorithmes de type glouton

Définition 23 (Heuristique)**.** *Une heuristique (du grec ancien "eurisko" « je trouve ») est une méthode de calcul qui fournit "rapidement" une solution "réalisable" (non nécessairement optimale ou exacte) pour un problème d'optimisation "difficile".*

Note : Une heuristique s'impose quand les algorithmes de résolution exacte sont de complexité exponentielle, et dans beaucoup de problèmes "difficiles". L'usage d'une heuristique est également pertinent pour calculer une solution approchée d'un problème, ou encore pour "accélérer" un processus de résolution exacte. Généralement, une heuristique est conçue pour un problème particulier, en s'appuyant sur sa structure propre, mais peut contenir des principes plus généraux.

Définition 24 (Algorithme glouton)**.** *Un algorithme glouton (greedy algorithm en anglais) est un algorithme qui suit le principe de faire, étape par étape, un choix optimum (local). Dans certains cas, cette approche permet d'arriver à un optimum global ; mais dans le cas général, c'est une heuristique.*

14 Problème de la somme du sous-ensemble (*Subset Sum Problem*)

14.1 Définition du problème

Définition 25 (Problème de la somme du sous-ensemble)**.**

- Entrée.
  - *Un tableau t, de taille n, à valeurs entières.*
  - *Une capacité $c \in \mathbb{N}$.*
- Problème.

  Trouver k indices de t, distincts, soient $i_1, ..., i_k$, tels que la somme :

$$t[i_1] + ... + t[i_k]$$

  Approche la capacité c, sans la dépasser.

Théorème 26. *Le problème de la somme du sous-ensemble est NP-hard.*

Proof. Admis. □

14.2 Heuristique gloutonne

Algorithm 25 GREEDY(t, n, c)

trier t selon l'ordre décroissant
$res \leftarrow$ Liste vide
$val \leftarrow 0$
for $i \leftarrow 0, ..., n-1$ **do**
    if $val + t[i] \leq c$ **then**
        $res.$**append**(i)
        $val \leftarrow val + t[i]$
    end if
end for
return res

Complexité : $O(n. \log n)$.

15 Problème de la coloration de graphe (*Graph Coloring Problem*)

15.1 Définition du problème

Définition 27 (Problème de la coloration de graphe)**.**

- Entrée.

  Un graphe G, non orienté sans boucle.

- Problème.

  Trouver le plus petit entier k, tel que G soit k-colorable.

Théorème 28. *Le problème de la coloration de graphe est NP-hard.*

Proof. Admis. □

15.2 Coloration gloutonne

Algorithm 26 GREEDY_COLOURING$(G = (S, A))$ tel que $S = \{s_i : i \in \{0, ..., n-1\}\}$

$color \leftarrow$ un dictionnaire vérifiant : $\forall s \in S, color[s] = -1$
for $i \leftarrow 0, .., n-1$ **do**
  $C_i \leftarrow \{color[s_j] : (s_i, s_j) \in A\}$
  $color[s_i] \leftarrow \min\{c \in \mathbb{N} : c \notin C_i\}$ ▷ Le plus petit entier positif n'appartenant pas à C_i
end for
return $color$

16 Problème du voyageur de commerce (*Travelling Salesman Problem*)

16.1 Définition du problème

Définition 29 (Cycle hamiltonien)**.** *Un cycle hamiltonien (d'un graphe) est un cycle passant par tous les sommets du graphe, une fois et une seule.*

Définition 30 (Problème du voyageur de commerce)**.**

- Entrée.

  Un graphe $G = (S, A, cost)$*, tel que :*
  - G *est non orienté ;*
  - G *est complet ;*
  - *Les arcs de* G *sont valués par la fonction cost :* $A \rightarrow \mathbb{R}$.

- Problème.

  Trouver le cycle hamiltonien ayant le coût le plus faible.

Théorème 31. *Le problème du voyageur de commerce est NP-hard.*

Proof. Admis. □

16.2 Heuristique : algorithme du plus proche voisin

Algorithm 27 NEAREST_NEIGHBOUR($G = (S, A, cost)$)

$todo \leftarrow \{\}$
$res \leftarrow$ Liste vide
$s_0 \leftarrow$ choisir un sommet arbitraire de G
$todo \leftarrow G - \{s_0\}$
$res.$**append**(s_0)
$s \leftarrow s_0$
while $todo \neq \{\}$ **do**
    $s' \leftarrow$ le sommet le plus proche de s
    $todo \leftarrow todo - \{s'\}$
    $res.$**append**(s')
    $s \leftarrow s'$
end while
return $res.$**append**(s_0)

17 Algorithmes génétiques

On cherche à maximiser $f : E \to \mathbb{R}$; où E est fini.

Exemple (pour un entier n) :

- $E = \texttt{float}^n$;
- $E = \mathbb{B}^n$;
- $E = \{x_1, ..., x_k\}^n$ (pour un entier k) ;
- $E = S_n$ (l'ensemble des permutations de {1, ..., n}).

Algorithm 28 ALGORITHME_GENETIQUE(n : taille d'un individu, p : nombre d'individus, ...)

```
P ← Population initiale
res ← Null                                          ▷ En considérant f(Null) = −∞
while not(Critère Fin) do
    Calculer la valeur de fitness de chaque individu    ▷ i.e. f(p), ∀p ∈ P
    best ← le meilleur individu de P    ▷ Au sens de la maximisation de f
    if f(best) > f(res) then
        res ← best
    end if
    P ← selection(P)
    P_new ← reproduction(P)
    P_new ← mutation(P_new)
    P ← P_new
end while
return res
```

(Les phases de *selection* et de *reproduction* peuvent être regroupées en une procédure).