Algorithmique avancée

Frédéric Wagner

Transcrit par Julien Henry

Ce cours n'est pas un polycopié officiel. Aussi, il n'est pas certifié sans erreurs. Pour toute remarque ou correction, vous pouvez me contacter à l'adresse

julien.henry@ensimag.imag.fr

2010

École Nationale Supérieure d'Informatique et de Mathématiques Appliquées, Grenoble

Table des matières

Chapitre I

Théorie des jeux

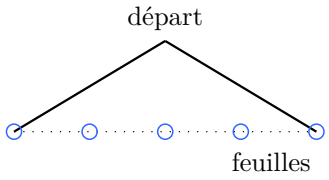
Nous allonc aborder dans ce chapitre la théorie des jeux de stratégie au tour par tour et à deux joueurs.

Pour toute intelligence infinie, tout jeux de stratégie n'a aucun intérêt : celui-ci se termine toujours de la même façon :

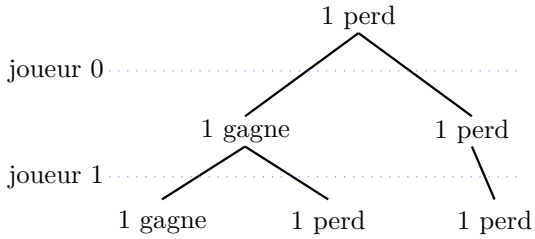
- Toujours le même joueue gagne.
- Il y a toujours match nul.
- le jeu ne se finit pas.

1. Algorithme Min-Max

Pour créer une intelligence artificielle pour un jeu, l'idée serait de tester toutes les combinaisons de coups possibles. Cela revient à envisager l'arbre des parties possibles.



Les feuilles contiennent alors l'information de qui gagne la partie. L'algorithme consiste donc à partir des feuilles et à remonter l'information à la racine en passant par les noeuds intermédiaires de l'arbre.




```
int MinMax(noeud n) {
  si n est une feuille {
    renvoyer evaluation(n);
  }
  si n est un noeud du joueur 0 {
    renvoyer min_{f∈fils} (MinMax(f));
  } sinon {
    renvoyer max_{f∈fils} (MinMax(f));
  }
}
```

Le joueur 0 essaie de minimiser l'évaluation, tandis que le joueur 1 tente de la maximiser. Dans `blobwar`, l'évaluation peut être `nbrouges - nbbleus`.

Il existe plusieurs possibilités d'implémentation de ce code :

- on peut allouer de nouveaux noeuds à chaque fois.
- on peut faire la modification dans le noeud courant :
  1. on modifie n pour obtenir son fils f.
  2. on evalue `MinMax(f)`
  3. on annule les modifications pour retrouver n.

On se confronte à un problème : la taille de l'arbre est exponentielle. Dans un certain nombre de jeux, comme dans `blobwar`, cette taille est même infinie. On doit donc restreindre cet algorithme à un sous-arbre (par exemple, les 8 premiers coups). Celà provoque donc une perte de force considérable de l'algorithme, mais elle est nécessaire pour minimiser le temps de calcul.

2. Optimisation : algorithme alpha-beta

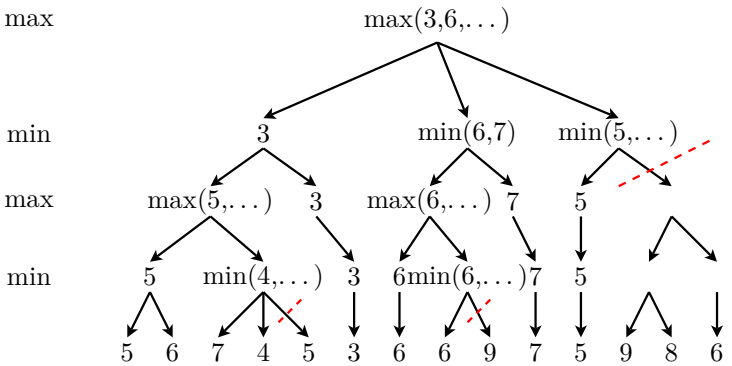
On peut optimiser l'algorithme MinMax en faisant des coupes dans l'arbre lorsque l'on est sûr que les noeuds en question ne seront jamais joués.

Les coupes nécessitent des informations supplémentaires.

```
Alpha_Beta(n, alpha, beta) {
  si n est au niveau limite {
    retourner evaluation(n);
  }
  Meilleur = -∞;
  pour tout fils f de n faire {
    Val = -Alpha_Beta(f,-beta,-alpha);
    si (Val > Meilleur) {
      Meilleur = Val;
      si (Meilleur > alpha) {
        alpha = Meilleur;
        si (alpha ≥ beta) {
          retourner Meilleur;
        }
      }
    }
  }
}
```

Cet algorithme permet d'obtenir la meilleure valeur que le joueur 0 (α) ou le joueur 1 (β) est garanti d'obtenir sur le noeuds du noeud courant à la racine.

Exemple On appelle $Alpha\_Beta(racine, -\infty, +\infty)$, et le résultat obtenue est 6 :



Chapitre II

Complexités des problèmes

1. Introduction

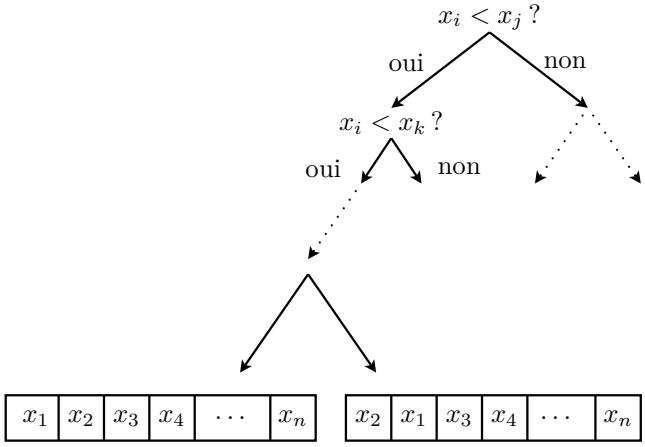
On se pose dans ce chapitre la question suivante :
Est-ce qu'on peut prouver qu'un problème nécessite pour sa résolution un nombre minimal d'opérations ?

***Définition* 1.** *Soit P un problème. Soit $\mathcal{A}$ l'ensemble des algorithmes résolvant P. On note $c(A, n)$ le coût au pire cas de l'algorithme A avec une entrée de taille n.*

On définit la complexité f de P comme étant la fonction :

$$n \longmapsto \min_{A \in \mathcal{A}} \{c(A, n)\}$$

Exemple (Complexité du tri par comparaison) Soit A un algorithme pour le tri par comparaison. À A correspond un arbre de décision. Supposons que A a en entrée les éléments $x_1, \ldots, x_n$. L'arbre de décision est donc de la forme :



Les feuilles de l'arbre donnent les solutions possibles. On sait que ces solutions sont

$$\{\text{permutations des } x_i, i \in 1..n\}.$$

Cet ensemble est de cardinal $n!$. L'arbre de décision de A pour une taille n est donc un arbre binaire et contient au moins $n!$ feuilles. ∗

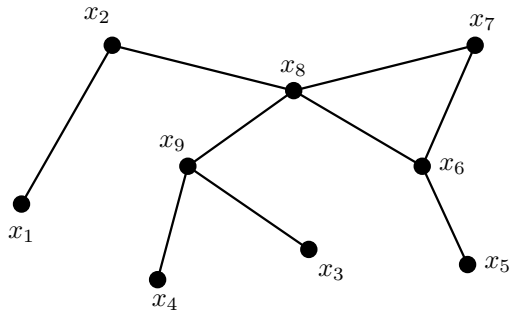
Propriété 1:
Un arbre binaire est de hauteur minimal si il est parfait.

Ainsi, en utilisant la formule de Stirling :

$$\begin{aligned} h &\geq \log(n!) \\ &\sim \log\left(\sqrt{(}2\pi n)\left(\frac{n}{2}\right)^n\right) \\ &\sim O(n\log n) \end{aligned}$$

Exemple (Maximum d'un tableau) On peut d'ores et déjà dire qu'une borne inférieure pour la recherche du maximum d'un tableau de taille n est $n-1$ comparaisons.

DÉMONSTRATION On peut tracer le graphe de comparaisons. Les sommets sont les entrées et les arêtes sont les comparaisons effectuées.

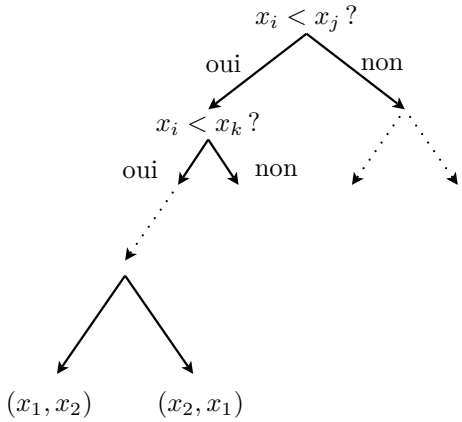


Pour trouver le maximum des éléments, on doit nécessairement avoir un graphe connexe. Or tout graphe connexe de n sommets contient au moins $n-1$ arêtes. □

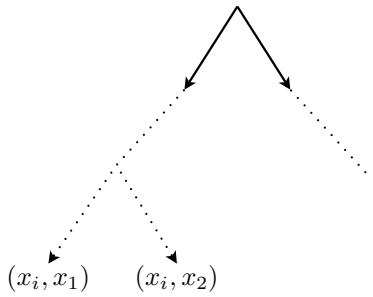
Exemple (2^e maximum d'un tableau) En utilisant ce que l'on a vu sur le maximum d'un tableau, on peut dire que la borne inférieure I vérifie :

$$n-1 \leq I \leq n-1+n-2$$

L'idée est similaire à celle du tri par comparaison : Soit A un algorithme qui résout le problème. On considère l'arbre de décision pour A



Supposons que x_i est le maximum. Soit T_i le sous arbre de T de la racine aux feuilles où x_i est le max.



Il y a donc deux types de sommets intermédiaires :
- degré sortant 1 : ceux qui servent à trouver le maximum.
- degré sortant 2 : ceux qui servent à trouver le 2^e maximum. *

On élimine de T_i les sommets de degré sortant 1 en les fusionnant, et on regarde l'arbre T_i' obtenu. On suppose que T_i est un arbre parfait (on peut toujours faire en sorte de s'y rapporter). On trouve donc le 2^e max parmi les $n-1$ éléments : celà nécessite $n-2$ comparaisons.

Le nombre total de feuilles vaut :

$$\sum_{i \leq n} feuilles(T_i') \geq n2^{n-2}$$

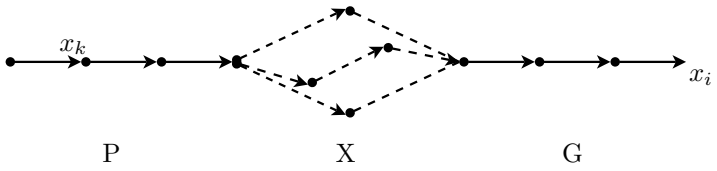
Ainsi, on en déduit que :

$$hauteur \geq \log(n2^{n-2}) = n - 2 + \log(n)$$

Exemple (Trouver le $k^{ème}$ élément) On utilise dans ce cas la une technique par adversaire.

Soit A un algorithme qui résout le $k^{ème}$ élément. On décrit un adversaire pour A tel que l'exécution de A prenne un temps grand. Ici, l'adversaire s'arrange pour que chaque comparaison ramène peu d'information à A. On encode les informations sur les x_i avec un graphe.

On distingue 3 types de sommets :
- P : petits sommets
- G : grands sommets
- X : sommets indéterminés



Démonstration Initialement, tous les sommets sont dans X.

A s'exécute : $x_i < x_j$?. Si oui Alors on crée la paire $xi \longrightarrow x_j$ dans X.

Adversaire : $x_a < x_b$?. Il existe plusieurs cas :

1. $x_a, x_b \in X$ et ne forment pas de paire :
   - oui et fabrique la paire
2. x_a, x_b forment une paire :
   - on redonne la réponse correspondante
3. x_a ou $x_b \in P$ ou G et l'autre sommet n'appartient pas à une paire :
   - on donne la réponse juste sans rien changer
4. x_a ou $x_b \in P$ ou G et l'autre sommet appartient à une paire :
   - Si $x_a \in P$ et (x_b, x_c) est une paire, on déplace x_b dans P.
   - autres cas similaires...

On note c le nombre de comparaisons et p le nombre de paires. On sait que mettre un élément dans P ou G coûte 2 comparaisons. On a donc l'invariant suivant :

$$c > 2(|P| + |G|) + p$$

On a $|P| + |G| = n - |X|$. Donc

$$c \geq 2n - 2|X| + p$$

On s'arrête dès que $|G| = k - 1$ ou $|P| = n - k$. En effet, à cet instant il suffit de rechercher le max ou le min dans X. Au moment de l'arrêt, on a

$$c_0 \geq 2n - 2x_0 + p_0$$

La dernière étape nécessite $p_0 - 1$ comparaisons :

$$2n - 2x_0 + p_0 + x_0 - p_0 - 1 = 2n - x_0 - 1$$

Or $x_0 \leq n - \min(k-1, n-k)$, d'où le résultat :

$$c \geq n + \min(k-1, n-k) - 1$$

On a donc trouvé une borne inférieure pour la complexité de A. □

2. Bornes de problèmes : réductions

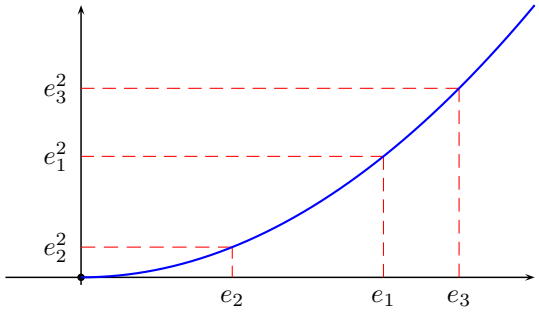
Il est difficile de calculer une borne inférieure de la complexité d'un problème. La solution est de faire des réductions : propagation des résultats.

Pour trouver une réduction d'un problème A vers un problème B, on écrit un algorithme de résolution de A à l'aide d'un oracle sur B. On dit que A se réduit à B.

Exemple Le tri se réduit à l'enveloppe convexe.

fonction de réduction :

- Entrée : un ensemble $(e_i)_{1 \leq i \leq n}$ d'entiers.
- Sortie : le tri des $(e_i)_{1 \leq i \leq n}$ par ordre croissant. *




```
P = ∅;
// construction de l'entree de B
pour chaque a_i {
  construire p_i = (a_i, a_i^2);
  ajouter p_i a P;
}
L = oracle_enveloppe(P) // L = (l_i, m_i)_{1≤i≤n}
// L est la sortie de B
// on calcule ensuite la sortie de A
pour i de 1 a n faire {
  si l_i > l_{i+1} break;
}
renvoyer (l_{i+1}, l_{i+2}, ..., l_1, ..., l_i);
```

Propriété 2:
Soit $f(n)$ le coût d'exécution de l'oracle B pour une taille n. Le coût de la fonction de réduction est $O(n)+f(n)$. Une borne supérieure pour la complexité de A est donc :

$$O(n) + f(n)$$

On reprend l'exemple du tri. Le meilleur algorithme connu de calcul de l'enveloppe convexe est de complexité $n\log(n)$. On a donc une borne sur la complexité du tri :

$$O(n) + O(n\log(n)) = O(n\log(n))$$

Soit $g(n)$ une borne inférieure sur la complexité de A. Alors tout algorithme résolvant A à un coût supérieur à $g(n)$.

Ceci est vrai pour la fonction de réduction $f(n)$. D'où :

$$\begin{array}{rcl} & O(n) + f(n) & \geq g(n) \\ \Rightarrow & f(n) & \geq f(n) - O(n) \end{array}$$

On en déduit une borne inférieure sur la complexité de B : $g(n) - O(n)$.

Exemple Produits et inversions de matrices

On va montrer ici que ces deux problèmes sont équivalents.

1. Montrons que le produit se réduit à l'inversion.

   La fonction de réduction est la suivante :
   - Entrée : 2 matrices A et B
   - Sortie : $A.B$

   ```
   // calcul de E a partir de A et B
   E = ( I_n  A   O   )
       ( O    I_n B   )
       ( O    O   I_n ) ;
   R = oracle_inversion(E);
   // calcul de M a partir de R
   E^-1 = R = ( I_n  -A   AB  )
              ( O    I_n  -B  )
              ( O    O    I_n ) ;
   M = sous-matrice de R encadree;
   renvoyer M;
   ```

   On calcule le coût :

   $$O(3n^2) + f(n) + O(n^2) = f(n) + O(n^2)$$

   Donc $f(n) + O(n^2)$ est une borne supérieure sur la complexité du produit de matrices.

   Soit $g(n)$ la complexité de la multiplication.

   $$\begin{array}{rcl} & g(n) & \leq f(n) + O(n^2) \\ \Rightarrow & f(n) & \geq g(n) - O(n^2) \end{array}$$

   En général, si $n^{2,5}$ est une borne inférieure sur la multiplication, alors c'est une borne inférieure sur l'inversion.

2. On doit montrer que l'inversion de matrice se réduit à la multiplication.

   Remarque Indication : Soit $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$ une matrice par blocs. On a

   $$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{pmatrix}$$

   - Entrée : X, Y
   - Sortie $X.Y$

   Lors de la réduction, on peut appeler plusieurs fois l'oracle.

   coût : $O(n)^2 + f(2n) + O(n^2) + f(n)$ *

3. Problèmes $\mathcal{P}$ et $\mathcal{NP}$

3-a. Problèmes $\mathcal{P}$

L'objectif est de faire une classification des problèmes, en se basant sur les ressources nécessaires pour les résoudre.

***Définition* 2.** *Soit $\mathcal{P}$ l'ensemble des problèmes de décision $\{\pi_i\}$ tel que :*

$\forall\ \pi \in \mathcal{P}, \exists\ \mathcal{M}$ une machine de Turing déterministe , $\forall\ d$ entrée de π, $\mathcal{M}$ résout π en temps polynômial sur $|d|$

Remarque Intuitivement, $\mathcal{P}$ est l'ensemble des problèmes abordables. ∗

***Définition* 3 (Problème de décision).** *Un problème de décision est un problème pour lequel la réponse est un booléen.*

Exemple (sac à dos) En entrée :
- n objets $e_1, \ldots, e_n$ de poids $p_1, \ldots, p_n$,
- $k \in \mathbb{N}$,
- le poids limite du sac P,
- les valeurs des objets $v_1, \ldots, v_n \in \mathbb{N}$.

En sortie :

est-il possible de charger le sac en respectant la contrainte pour une valeur $\geq k$? ∗

3-b. Problèmes $\mathcal{NP}$

***Définition* 4.** *Il existe 3 définitions :*

1. *Un problème $\pi \in \mathcal{NP}$ si et seulement si :*
   $\exists \mathcal{M}$ une machine de Turing NON déterministe, $\forall d$ entrée pour π :
   - *si $\pi(d) = true$, alors $\mathcal{M}$ donne $true$ en temps polynômial.*
   - *si $\pi(d) = false$, alors $\mathcal{M}$ donne $false$ en temps polynômial ou boucle.*
2. *version parallèle : au lieu d'une machine non déterministe, on a une machine avec un nombre infini de processeurs.*
3. *version certificat : $\pi \in \mathcal{NP}$ si et seulement si $\forall d \exists C$ un certificat (une preuve) de taille polynômiale que $\pi(d) = true$.*

Exemple (sac à dos) certificat : la liste des objets choisis.
La taille du certificat est O(nombre d'objets choisis) $= O(n)$.

Remarque Avec les définitions, on remarque que $\mathcal{P} \subset \mathcal{NP}$ La question est de savoir si $\mathcal{P} = \mathcal{NP}$. Ceci pourrait revenir à dire que trouver une preuve est aussi difficile que la vérifier. La conjecture actuelle est que $\mathcal{P} \neq \mathcal{NP}$. Il existe sans doute des problèmes dans $\mathcal{NP}$ et qui ne sont pas dans $\mathcal{P}$.
Une information très importante sur un problème est donc de savoir à quelle classe il appartient. ∗

3-c. Problèmes $\mathcal{NP}-$complets

***Définition* 5.** *Un problème π est $\mathcal{NP}-$dur ($\mathcal{NP}-$difficile) si et seulement si :*
$\forall \pi_2 \in \mathcal{NP}, \pi_2$ se réduit à π par une réduction polynômiale.

Réductions polynômiales algorithmes de réduction de coût polynômial :
- Karp : un seul appel à l'oracle
- Coor/Turing : plusieurs appels à l'oracle

Intuitivement, tous les problèmes de $\mathcal{NP}$ sont plus simples que π.

***Définition* 6.** *π est $\mathcal{NP}-$complet si :*
- *π est $\mathcal{NP}-$dur,*
- *$\pi \in \mathcal{NP}$*

En résumé : Soit un problème π.
Est-il possible de trouver un algorithme résolvant π ?
Est-ce que π est $\mathcal{NP}-$complet ?
Comment montrer qu'un problème π est $\mathcal{NP}-$complet ?
- $\pi \in \mathcal{NP}$
- π est plus dur que tous les problèmes de $\mathcal{NP}$. Le problème est qu'il est difficile de faire des preuves incluant un nombre infini de problèmes.

Pour cela on part d'un problème A, $\mathcal{NP}-$complet : Tout problème de $\mathcal{NP}$ se réduit à A, qui se réduit à π.

Théorème 1 (Théorème de Cook):
Le problème SAT est $\mathcal{NP}-$complet.
Entrée : une expression booléenne.
Sortie : est-il possible que l'expression soit vraie ?

Cela nous amène vers les 21 problèmes de Karp. Ce sont des problèmes classiques pour lesquels les réductions sont faciles.

Exemple (independent set) Entrée : un graphe $G = (V, E)$, un entier k
Sortie : Existe-t-il un ensemble indépendant de sommets de cardinalité $\geq k$?
On va montrer que *independent set* (IS) est $\mathcal{NP}-$complet :

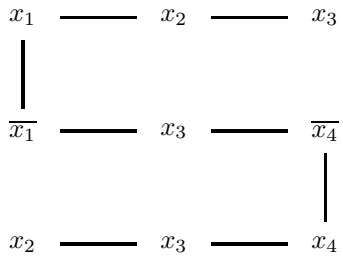
Démonstration On a deux choses à montrer.

1. $IS \in \mathcal{NP}$
   certificat : la liste des sommets de l'ensemble.
2. SAT se réduit à IS
   (a) écrire la réduction.
   (b) prouver que la réduction est correcte.

   la réduction : algorithme polynômial résolvant SAT à l'aide de l'oracle sur IS.
   réduction :
   Entrée : ensemble de clauses C. On construit le graphe G tel que :
   - chaque littéral de C donne un sommet de G.
   - il y a une arête entre 2 sommets de littéraux opposés.
   - il y a une arête entre 2 sommets de littéraux dans la même clause.

   $k = \#clauses$
   return $oracle_{IS}(G, k)$

Exemple $(x_1 + x_2 + x_3) \times (\neg x_1 + x_3 + \neq x_4) \times (x_2 + x_3 + x_4)$



On a envie de faire une preuve plus formelle. On va donc montrer que $oracle_{IS}(G, k)$ répond *true* si et seulement si la réponse à SAT est *true*.

(a) $oracle_{IS}$ *true* $\Rightarrow$ SAT *true* Si la réponse à $oracle_{IS}$ est *true*, alors il existe $S \subset V$ un ensemble de sommets indépendants de taille égale au nombre de clauses. On affecte les littéraux correspondants à chacuns des sommets de S à *true*.

On n'a pas de contradiction car on n'a pas d'arêtes de type 2.

On a un sommet par clause, donc on a un littéral vrai dans chaque clause. L'expression est donc vraie et la réponse à SAT est *true*.

(b) SAT *true* $\Rightarrow$ $oracle_{IS}$ *true* Montrons que si la réponse à SAT est *true*, alors l'oracle est *true*. La réponse à SAT est *true* donc $\exists$ une assignation des variables tel que l'expression soit *true*. Donc toutes les clauses sont vraies.

Dans chaque clause, il existe au moins un littéral vrai.

Soit S l'ensemble des sommets de G correspondant aux littéraux vrais. On construit $S' \subset S$ tel que S' contient un sommet par clause.

$$\forall (a, b) \in S' \times S', (a, b) \in E$$

car les littéraux correspondant ne peuvent pas être opposés (pas d'arêtes de type 2) et il n'y a qu'un seul sommet par clause (pas d'arêtes de type 1), donc S' est indépendant.

Ainsi, (IS) est $\mathcal{NP}-$complet. □

Chapitre III

Algorithmes d'approximation

1. Définition

L'objectif de ces algorithmes est de résoudre des problèmes d'optimisation difficiles. Pour cela, on accepte une solution dégradée (non optimale) à condition qu'elle ne soit pas trop mauvaise.

Définition 7. *Soit A un algorithme d'approximation. Soit y une entrée de A, et $opt(y)$ la solution optimale correspondant à y.*

On note $A(y)$ la solution trouvée par A. Si e est une fonction d'évaluation, on note

$$r(y) = \max\left(\frac{e(A(y))}{e(opt(y))}, \frac{e(opt(y))}{e(A(y))}\right)$$

On dit que A est un algorithme d'approximation de ratio ρ si et seulement si

$$\forall y, r(y) \leq \rho$$

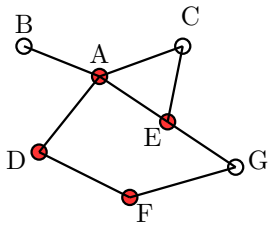
---

2. Exemples

Exemple (Vertex Cover) Entrée : $G = (V, E)$

Sortie : $C \subset V, \forall (a, b) \in E, (a \in C \vee b \in C)$ et $|C|$ est minimale.

Solution : ensemble de sommets :



$$\begin{aligned} A(y) &= \{A, E, F, D\} \\ e(A(y)) &= 4 \\ r &= \max(4/3, 3/4) \end{aligned}$$

$e : |C|$: ici, on cherche à minimiser e. *

On se pose maintenant les questions suivantes :

1. Comment trouver un alorithme d'approximation ?
   Souvent on a des algos simples → preuve ratio : exemple qui atteint le plus mauvais ratio → modification algorithme → preuve ratio...

2. Comment prouver un ratio d'approximation ?
   - pire exemple possible : Qu'est ce que l'algo fait mal ?
   - on utilise une borne sur l'optimal.
   - preuve mathématique

Exemple (Vertex cover) On reprend l'exemple précédent : A :
Entrée : $G = (V, E)$
Sortie : C

```
Tant que E est non vide faire

  choisir (a, b) ∈ E
  C = C ∪ {a, b}
  enlever de E toutes les aretes adjacentes a a et b
```

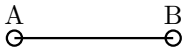
Dans l'exemple, on a $C = \{A, B, C, D, E, F, G, H\}$ En exécutant l'algorithme, on obtient :

$$opt? = \{B, D, E, G, H\}$$

Cela donne un ratio de $8/5$. Si A est une ρ-approximation, alors :

$$\rho \leq \frac{8}{5}$$

De plus dans le cas où on a le graphe suivant :



On a :

$$\begin{aligned} C &= \{A, B\} \\ opt &= \{A\} \end{aligned}$$

Montrons que A est une 2-approximation :

DÉMONSTRATION Soit C l'ensemble de sommets séléctionnés par A et S l'ensemble des arêtes séléctionnées par A. Soit C^* une solution optimale.

On sait que C^* est la couverture optimale. Donc C^* couvre toutes les arêtes, donc C^* couvre toutes les arêtes de S.

Or

$$\forall (v_1, v_2) \in S, (v_3, v_4) \in S, v_1 \neq v_2 \neq v_3 \neq v_4$$

Pour chaque arête de S, C^* contient au moins une des 2 arêtes. On en conclut que $|C^*| \geq |S|$

$$|C| = 2|S|$$

On en déduit le ratio :

$$\frac{|C|}{|C^*|} \leq \frac{|C|}{|S|} = \frac{2|S|}{|S|} = 2$$

On a donc bien un ratio de 2. □

Exemple (Sac à dos) On a :
Entrée : n objets de valeur v_i et de poids p_i. P la capacité du sac.
Sortie : $(x_i)_{1 \leq i \leq n}$: le nombre de fois que l'objet i est pris, tel que

$$\sum_i x_i \; p_i \leq P$$

et on maximise

$$\sum_i x_i v_i$$

trier les objets par ratio $\frac{v_i}{p_i}$ decroissant.
$R = P$
pour chaque objet i (dans l'ordre) **faire**
$x_i = \left\lfloor \frac{R}{p_i} \right\rfloor$
$R = R - \left\lfloor \frac{R}{p_i} \right\rfloor p_i$
fin pour

Essayer de trouver le pire exemple :
On choisit $P = 100$ et les objets suivants :

P	51	50
v	102	99

$$r_1 = \frac{102}{51} = 2$$
$$r_2 = \frac{99}{50} < 2$$

L'algorithme va alors choisir $x_1 = 1$ et $x_0 = 0$. $R = 49$ et la valeur totale est 102.
D'autre part, l'optimal est $x_1 = 0, x_2 = 2$ de valeur 198.

$$r = \frac{198}{102}$$

DÉMONSTRATION Montrons que l'algorithme est une 2-approximation.

Soit s la valeur de la solution retournée par A et s^* la valeur optimale. On cherche à calculer $\frac{s^*}{s} \leq \frac{borne}{s} = 2$. Il nous faut une borne supérieure sur s^*.

Soit E le poids libre restant à la fin de l'exécution de A.

- Montrons que $E \leq \frac{P}{2}$:
  par l'absurde, si $E \geq \frac{P}{2}$, alors on peut remettre les éléments déjà placés : contradiction.
- borne supérieure sur s
  Comme $E < \frac{P}{2}$, prendre 2 fois les éléments choisis par A remplit tout le poids possible et comme les objets sont triés, alors $2ss > s^*$.
  Le ratio est donc :

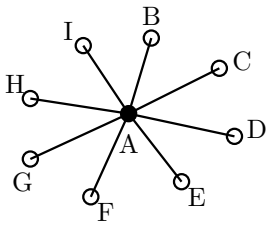
$$\frac{s^*}{s} < \frac{2s}{s} = 2$$

On a donc bien une 2-approximation. □

Exemple (Independant set) On rappelle le problème :
Entrée : $G = (V, E)$
Sortie : $C \subset V$ tel que $|C|$ soit maximal et $\forall (v_1, v_2) \in C^2, (v_1, v_2) \notin E$.
On propose l'algorithme suivant :

$C = \varnothing$
tant que V n'est pas vide **faire**
choisir $v \in V$
$C = C \cup \{v\}$
enlever v et tous ses voisins de V
fint **tant** que

Est-ce que A est un algorithme d'approximation ? Quel est le pire exemple possible ?



Ici, il se peut que l'algorithme choisisse $\{A\}$, alors que la solution optimale est $\{B, C, \ldots, I\}$

Dans un cas plus général, pour un graphe en étoile avec n sommets, le ratio tend vers l'infini lorsque $n \longrightarrow +\infty$. *

Exemple (Ordonnancement par liste) On considère le problème de l'ordonnancement :

Entrée :

– m machines homogènes.

– n tâches de calcul de durées $(p_i)_{1 \leq i \leq n}$. *

Sortie : Comment placer les tâches sur la machine pour minimiser le temps de complétion (C_{max}) ?

On utilise l'algorithme d'ordonnancement par liste :

```
trier les taches suivant une liste
des qu'une machine devient idle, lui donner la premiere tache de la liste
```

On arrive a trouver que pour m machines, on a un ratio de $2 - \frac{1}{m}$. On peut donc essayer de montrer que l'ordonnancement par liste est une 2-approximation.

C'est un problème de minimisation : borne inférieure sur la solution optimale.

$$C^*_{max} \geq \max\left(\frac{\sum p_i}{m}, \max_i(p_i)\right)$$

$$\frac{C_{max}}{C^*_{max}} = \frac{C_1 + C_2}{C^*_{max}} \leq \frac{C_1 + C_2}{\max\left(\frac{\sum p_i}{m}, \max_i(p_i)\right)} \leq \frac{\frac{\sum p_i}{m} + \max_i(p_i)}{\max\left(\frac{\sum p_i}{m}, \max_i(p_i)\right)} \leq \frac{2\max\left(\frac{\sum p_i}{m} + \max_i(p_i)\right)}{\max\left(\frac{\sum p_i}{m}, \max_i(p_i)\right)} = 2$$

Jusqu'où peut-on améliorer le ratio ?

I y a deux cas :

– $\longrightarrow 1 + \varepsilon$: *PTAS : polynomial time approximation scheme* : temps d'exécution exponentiel en $\frac{1}{\varepsilon}$

– limite au ratio d'approximation

Exemple (cycle hamiltonien, voyageur de commerce) Considérons un algorithme d'approximation pour le voyageur de commerce.

$$\forall G, v(A(G)) \leq \rho v(opt(G))$$

$\exists$ un cycle hamiltonien $\Rightarrow \exists$ *opt* de longueur 0. $\rho.0 = 0$.

$$\forall \rho, v(A(G)) \leq 0 = 0$$

A trouve le cycle hamiltonien.

Or $\mathcal{P} \neq \mathcal{NP}$ donc A n'existe pas.

Donc $\nexists$ d'algorithmes d'approximation pour le voyageur de commerce. *

***Définition* 8 (Gap-réduction).** *C'est une technique permettant de prouver qu'un problème d'optimisation π n'est pas approximable à moins d'un ratio ρ si $\mathcal{P} \neq \mathcal{NP}$.*

Chapitre IV

Algorithmes probabilistes

1. Introduction

Ce sont des algorithmes utilisant des tirages aléatoires. Cela équivaut à une machine de Turing à un ruban supplémentaire contenant des nombres aléatoires. Ces algorithmes ont essentiellement deux intérêts :

- algorithmes *online*
- clarté

On distingue trois types d'algorithmes :

- *Las Vegas*
- *Atlantic City* : 2-sided error
- *Monte Carlo* : 1-sided error

***Définition* 9.** *Soit E l'ensemble des entrées et S l'ensemble des sorties. Si $x \in E$ une entrée, on cherche $y \in S$ tel que $x\mathcal{R}y$.*

On a trois types de sorties :

- *(O) : $y \in S,\ x\mathcal{R}y$*
- *(N) : $\nexists\, y \in S,\ x\mathcal{R}y$*
- *erreur*

1. ***Atlantic City** :*
   $\forall x \in E$, l'algorithme retourne soit une réponse (O) soit une réponse (N) mais avec une probabilité d'erreur inférieure à une constante P.
2. ***Monte Carlo** :*
   $\forall x \in E$, l'algorithme retourne soit une réponse (O) correcte, soit une réponse (N) avec une probabilité d'erreur inférieure à une constante P.

Exemple (Recherche de chaîne) Soit $X = [X_0, \ldots, X_{m_1}]$ une chaîne de caractères (1 caractère = 1 octet) dont on cherche les occurences dans un fichier $Y = [Y_0, \ldots, Y_{n_1}]$.

L'algorithme de conparaison octet par octet est :

```
pour i de 0 a n_1 - m_1 faire
    trouve = faux
    pour j de 0 a m_1 faire
        si X_j = Y_{i+j} alors
            trouve = vrai
        fin si
    fin pour

    si trouve alors
        afficher(i)
    fin si

fin pour
```

Cet algorithme est de coût $O(n.m)$.
Pour construire un algorithme de coût $O(n)$, on hache les chaînes dans les entiers par :

$$\Phi([A_i, \dots, A_{i+k}]) = \sum_{j=0}^{k} A_{i+k-j}\ 256^j$$

Pour limiter le coût des opérations arithmétiques, on calcule modulo K où K est un entier pas trop grand, de façon à ce que le coût des opérations arithmétiques modulo K soit $O(1)$.
On note $N_i = \Phi([Y_i, \dots, Y_{i+m-1}]) \mod K$ pour $i \in \{0, \dots, n-m\}$. On calcule N_i en fonction de N_{i-1} :

$$N_i = 256(N_{i-1} - 256^{m-1} \,.\, Y_{i-1} \mod K) + Y_{i+m-1} \mod K$$

Ainsi, toutes les valeurs N_i peuvent se calculer en un temps $O(n)$.
On propose maintenant un algorithme de Monte-Carlo :

```
calcul des N_i
calcul de Φ(X) mod K = N_X
pour i de 0 a n-m faire // cout O(n)
   si N_i = N_X alors
      afficher i
   fin si
fin pour
```

On définit :

$$M_{X,Y} = \prod_{i=0;[Y_i,\dots Y_{i+m-1}]\neq X}^{n-m-1} (\Phi(Y_i, \dots, Y_{i+m-1}) - \Phi(X))$$

Montrons que si K ne divise pas $M_{X,Y}$, alors l'algorithme ne trouve pas d'occurences incorrectes.
Problème : il est possible que deux chaînes différentes soit détectées comme identiques, lorsque :

$$\begin{array}{rl} & \Phi(Y_i, \dots) \mod K = \Phi(X) \mod K \\ \Longrightarrow & (\Phi(Y_i, \dots) - \Phi(X)) \mod K = 0 \end{array}$$

Donc $M_{X,Y}$ est divisible par K.
Montrons que $N_{X,Y} < 256^{n.m}$.

$$\begin{array}{rcl} \Phi(Y) - \Phi(X) & < & 256^m \\ \prod_{i=0}^{n-m-1} \Phi(Y) - \Phi(X) & < & 256^{m.(n-m)} < 256^{n.m} \end{array}$$

Au pire cas tous les facteurs premiers sont égaux à 2. Donc le résutat est inférieur à $n.m.\log(256)$.
On choisit K aléatoirement parmi les $\varphi(\alpha)$ nombres premiers $\leq \alpha$.
Sachant que $\varphi(\alpha) \geq \dfrac{\alpha}{\log \alpha}$:

$$\begin{array}{rcl} p_{erreur} & \leq & \dfrac{n.m.\log(256)}{\varphi(\alpha)} \\ & \leq & \dfrac{n.m.\log(256)}{\alpha}.\log 2 \end{array}$$

*

2. Las Vegas

Las Vegas est en fait la combinaison entre Monte Carlo et un algo de vérification.

```
tant que le resultat est faux faire
    res = Algo_Monte_Carlo();
    si le temps ecoule depasse une limite L
       renvoyer E
    fin si
fin tant que
```

Exemple (Skip List (Liste à enjambements)) Ce type de liste en utilisé en particulier dans les dictionnaires. L'idée consiste à faire plusieurs listes.

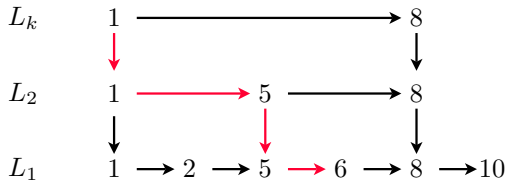


FIGURE IV.1 – Pour rechercher l'élément 6, on parcours la liste rouge

Algorithme probabiliste : Lors de l'insertion d'un élément x :

```
On insere x_i dans L_1.
On tire a pile ou face son insertion dans L_2
    pile -> insertion dans L_2
    face -> arret
On tire a pile ou face son insertion dans L_3
    pile -> insertion dans L_3
    face -> arret
...
jusqu'a l'obtention d'un face
```

Avec n éléments, en moyenne $k = lg(n)$. On note H_i la hauteur maximale de l'élément x_i. On va calculer, pour une constante l donnée, la probabilité $P(H_i > l)$:

$$P(H_i > 2) = \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \cdots = \frac{1}{4}$$

Donc :

$$P(H_i > l) = \sum_{j>l} \frac{1}{2^j} = \frac{1}{2^l}$$

On en déduit donc une borne : $P(k > l) \leq \frac{n}{2^l}$. Si $l = c \log_2 n$, on a :

$$P(k > c \log_2 n) \leq \frac{n}{2^{c \log_2 n}} = \frac{1}{n^{c-1}}$$

analyse du coût d'une recherche :

On part de la fin (élément cherché) et on recule en remontant. À chaque élément, on a une chance sur deux de remonter vers la liste supérieure.

- En $d \log n$ éléments, avec d assez grand, on a une forte probabilité d'être sur L_k.
- De plus, avec une forte probabilité, le monbre d'éléments de L_k est constant.
- Le coût total est donc, avec une forte probabilité, de $d \log n + cste$

Question : Au lieu de tirer à pile ou face, on considère une probabilité p d'être présent dans la liste supérieure. QUe se passe-t-il en faisant varier p ?

Lorsque p diminue, la consommation mémoire baisse, et on améliore le temps de calcul jusqu'à un certain point. *

Chapitre V

Algorithmes parallèles

1. Introduction

Le calcul parallèle est très difficile. Il faut avoir un fort niveau en programmation pour produire des programmes parallèles performants. Il faut un bon niveau en ordonnancement. Ici, nous n'écrirons les algorithmes qu'en pseudo-code.

L'objectif du chapitre est le suivant :

- Étant donné un algorithme :
  1. Quel est son temps d'exécution ?
  2. Quel est son coût ?
- Comment écrire un algorithme parallèle ?
- Mon problème admet-il un algorithme parallèle efficace ?
- Est-ce que tout problème admet un algorithme parallèle efficace ?
- Quelle est la puissance et quelles sont les limites du calcul parallèle ?

2. Pré-requis : Vol de travail

2-a. Présentation

Le vol de travail est un algorithme pour le calcul parallèle. Prenons l'exemple du problème d'ordonnancement. En pratique, les tâches que l'on doit réaliser présentent des dépendances.

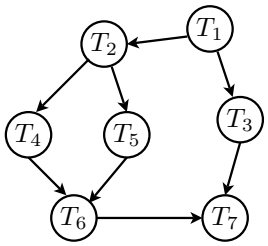


Figure V.1 – Exemple de graphe de dépendances

Le code va alors ressembler à ceci :

```
p {                              T2 {
    T1();                            calcul_T2();
    fork(T2);                        fork(T4);
    fork(T3);                        fork(T5);                    etc...
    sync;                            sync;
    T7();                            T6();
}                                }
```

On a un ordonnancement avec dépendances. L'ordonnancement par liste fonctionne, mais il nécessite une liste de tâches prêtes, tenue à jour en continu. De plus il y a un point de centralisation (très mauvais).

La solution est d'utiliser l'algorithme de vol de travail : Chaque machine dispose d'une pile de tâches.

```
tant que l'execution n'est pas finie faire
   tant que du travail est disponible sur la pile faire
      depiler la tache
      l'executer
   fint tant que
   choisir au hasard une machine distante
   prendre une ou plusieurs taches de sa pile
fin tant que
```

2-b. Analyse

Algorithme de la collection complète

On a n images à collectionner. Pour cela on fait des achats aléatoires. Combien d'achats doit-on faire pour obtenir toutes la collection ?

En moyenne, on doit effectuer $n \log n$ achats.

fonction SIPS (Strictly Increasing Per Stool)

On essaie de borner le nombre d'opérations de vol. On associe une fonction f à chaque machine tel que f augmente à chaque vol.

$$f(\mathcal{M}_i) = \min_{j \in \mathcal{M}_i} (profondeur(j))$$

Propriété 3:
Soit $\mathcal{M}_i$ la machine sur laquelle f est minimale. Alors tout vol sur $\mathcal{M}_i$ augmente f de 1.

Démonstration (indications) Il faut distinguer deux cas :
- $\mathcal{M}_i$ a du travail en trop.
- $\mathcal{M}_i$ n'a pas de travail en trop.

On constate que dans les deux cas, on augmente bien la valeur de la fonction de 1. □

Sur toutes les machines, f est bornée par D, où D est la profondeur du graphe. En un vol par machine, le min des f augmente de 1. Cela nécessite $p \log p$ vols. On a donc au plus $D \cdot p \log p$ vols.

3. Complexité parallèle : classe $\mathcal{NC}$

***Définition* 10 (Nick's Class).** *L'ensemble $\mathcal{NC}$ est l'ensemble des fonctions évaluables en temps polylogarithmique sur un nombre polynômial de processeurs.*

Théorème 2:
On a le résultat suivant :

$$\mathcal{NC} \subset \mathcal{P}$$

Lemme 1 (Lemme de Brent):
Toute exécution parallèle peut être repliée efficacement sur un nombre parallèle de processeurs.

Exemple Soit 3 processeurs et $I_1 \dots I_9$ des tâches processeurs à effectuer.

Processeur 1	Processeur 2	Processeur 3
I_1	I_2	I_3
I_6	I_5	I_4
I_7	I_8	I_9

Si on replie ces tâches sur 2 processeurs, on obtient (sachant que I_4 ne peut être exécuté qu'après I_1, I_2, I_3, etc.) :

Processeur 1	Processeur 2
I_1	I_2
I_3	$\varnothing$
I_4	I_5
I_6	$\varnothing$
...	

*

3-a. $\mathcal{NC} = \mathcal{P}$?

Soit $\Pi \in \mathcal{NC}$. Alors il existe un algorithme parallèle en $O(\log^k(n))$ sur $O(n^{k'})$ processeurs. En repliant l'exécution sur un seul processeur on obtient un algorithme en $O(n^{k'} \log^k(n))$. Ainsi $\Pi \in \mathcal{P}$.

On a donc $\mathcal{NC} \in \mathcal{P}$. Toutefois on ne sait pas si $\mathcal{NC} = \mathcal{P}$. La conjecture est que l'égalité est fausse.

Ainsi, il existe des problèmes de $\mathcal{P}$ qui sont très difficiles à paralléliser.

***Définition* 11.** *On appelle $\mathcal{P}-$complet la classe des problèmes qui sont très difficiles à paralléliser.*

---

4. Algorithmes

La différence majeure entre les algorithmes parallèles et séquentiels est que pour les algorithmes parallèles, l'ordre sur les opérations n'est pas total.

L'objectif est d'estimer les différents coûts. Cela nécessite un modèle de la réalité. Cependant, il existe de nombreux modèles pour les algorithmes parallèles :

- hétérogénéité matérielle
- réseau (aléas)
- hiérarchie réseau, voire plusieurs réseaux
- hétérogénéité des machines
- pannes
- ...

4-a. Modèles simples

Vol de travail

On suppose les hypothèses suivantes :

- Les machines sont homogènes.
- On a p processeurs.
- La puissance des machines est fixe.
- Les coûts de communication sont en $O(1)$.
- En entrée, on a un DAG de degré sortant ≤ 2, avec des tâches unitaires, une profondeur D et un travail W.
- On a un sommet initial.

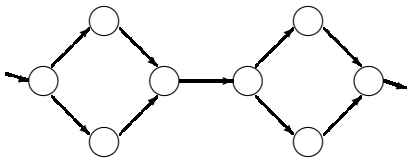


FIGURE V.2 – Ici, $D = 6$ et $W = 8$

Avec ces hypothèses, on a :

$$T_p = \frac{W}{P} + O(D)$$

Le nombre de vols est en $O(p.D)$. Ceci est optimal si $D << W$.

PRAM

- On a plusieurs processeurs.
- On a des mémoires locales.
- On a de la mémoire partagée.
- Les processeurs sont synchrones : à chaque top, chaque CPU avance d'une instruction.

Ce modèle a différentes variantes :

1. CREW : Concurrent Read Exclusive Write.
2. EREW : Exclusive Read Exclusive Write
3. CRCW : Concurrent Read Concurrent Write : il en existe plusieurs variantes.

4-b. Parallel for

Si on a :

```
for// (i = 0; i<20; i++) {
  A[i] = 0;
}
```

D et W ne sont pas les même suivant le modèle.

- Avec PRAM, $D = 1$ et $W = 20$.
- Avec le vol de travail, $D = \log(10)$ et $W \leq 20$.

On peut écrire ce *parallel for* avec des forks :

```
init (int debut, int fin) {
  if (debut == fin) {
    A[debut] = 0;
  } else {
    fork init(debut + ⌊(fin − debut)/2⌋, fin)
    fork init(debut, debut + ⌊(fin − debut)/2⌋)
  }
}
```

4-c. Calcul de W et D

Étant donné un programme, comment calculer D et W ?

En parallèle, on exécute le programme qui est simplement la suite d'instruction $f1(); f2();$. $f1$ est exécuté en parallèle puis $f2$ est exécuté en parallèle.

En connaissant D_{f1}, D_{f2} et W_{f1}, W_{f2}, alors :

$$D = D_{f1} + D_{f2}$$

$$W = W_{f1} + W_{f2}$$

De la même façon, si on a le programme :

```
for (i=0; i<n; i++) {
  f();
}
```

Si D_f et W_f sont constants, alors :

$$D = nD_f$$

$$W = nW_f$$

4-d. Notion d'efficacité

***Définition* 12.** *On définit les termes suivants :*

1. *Un algorithme est dit* ***efficace*** *si :*
$$T_p \cdot p = W_{seq} \cdot O(\log^k(n))$$
2. *Un algorithme est dit* ***optimal*** *si :*
$$T_p \cdot p = W_{seq}$$

5. *Exemples*

Exemple (Produit itéré) On veut calculer :

$$s = \sum_{i=1}^{n} a_i$$

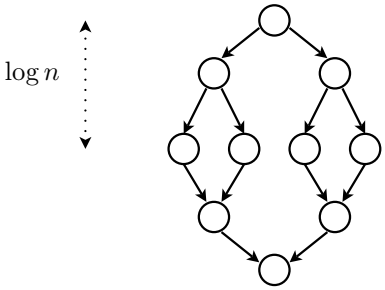
L'algorithme séquentiel est donc :

```
s = 0;
for (i=0; i<=n; i++) {
  s += A[i];
}
```

On a $W = n$.

La version parallèle est la suivante :

```
somme (int debut, int fin, int result) {
  if (debut == fin) {
    result = A[debut];
    return;
  }
  shared int r1,r2;
  fork somme(debut + ⌊(fin − debut)/2⌋, fin, r1);
  fork somme(debut, debut + ⌊(fin − debut)/2⌋, r2);
  // synchronisation
  fork addition(r1,r2,result);
}
```



On a $D = O(\log n)$ et $W = O(n)$. Le temps d'exécution est :

$$t_{exec} = \frac{W}{p} + O(D) = \frac{O(n)}{p} + O(\log_2 n)$$

Pour obtenir un temps d'exécution logarithmique, on doit avoir :

$$p = \frac{n}{\log_2 n}$$

Ainsi, on aura bien $\frac{n}{p} = \log_2 n$. Finalement, on aura $t_{exec} = O(\log_2 n)$.

Remarque Sur un modèle plutôt PRAM, on aurait $n/2$ processeurs à l'étage 0,$n/4$ à l'étage 1, ..., 1 seul processeur à l'étage $\log_2 n$. *

Équilibre des travaux On fixe le temps de calcul à $O(\log n)$. On cherche maintenant à minimiser le nombre de processeurs.

Pour cela, on s'appuie sur l'algorithme séquentiel. On modifie notre ajgorithme en rajoutant au début :

```
if (fin - debut == log2 n) {
  return algo_sequentiel(debut,fin);
}
```

La hauteur de l'arbre reste en $O(\log n)$, car la partie séquentielle est en $O(\log n)$.

6. Algorithme du préfixe

En Entrée : $A = (a_1, \ldots, a_n)$.
En Sortie : $R = (r_1, \ldots, r_n)$ tel que

$$\forall i \in 1..n, r_i = \sum_{j=1}^{i} a_j$$

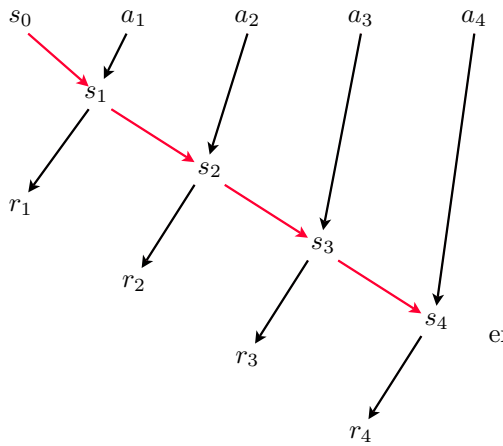
On veut écrire un algorithme parallèle qui réalise le calcul.

6-a. écriture d'un algorithme séquentiel

```
s = 0;
for (int i = 1; i <= n; i++) {
  s += a[i];
  r[i] = s;
}
```

Cet algorithme est clairement en $O(n)$. La borne inférieure est $O(n)$.

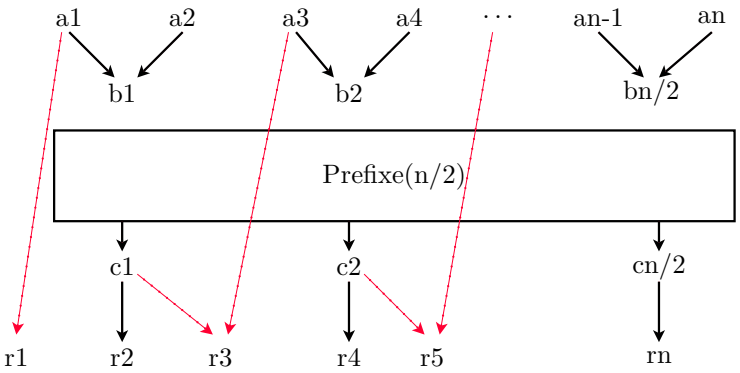
6-b. Graphe de dépendances



extrêmement séquentiel

Le calcul est très séquentiel. On va donc essayer une nouvelle approche.

Seconde approche



On a alors l'algorithme suivant :

```
prefixe(A,n) {
  for//(i = 1; i <= n/2; i++)
      b_i = a_{2i-1} + a_{2i};
  C = prefixe(B, n/2);
  for//(i = 1; i <= n; i++) {
    si i est pair
        r_i = c_{i/2};
    sinon
        r_i = c_{(i-1)/2} + a_i;
    fin si
  }
}
```

On a :

$$D(n) = O(1) + D(n/2) \Rightarrow O(\log(n))$$

$$W(n) = O(n) + W(n/2) \Rightarrow O(n)$$

– Avec PRAM : $n/2$ processeurs. $T_p = \log n$. La consommation est

$$p.T_p = n/2 \ . \ \log n = O(n \log n)$$

L'algorithme est efficace mais pas optimal.

– Avec le Vol de travail :

$$T_p = \frac{W}{p} + O(D) = \frac{O(n)}{p} + O(\log n)$$

On veut avoir $T_p = O(\log n)$, donc $p = O\left(\frac{n}{\log n}\right)$. La consommation est $\frac{n}{\log n}.\log n = n$.

$$D = O(\log n) \ . \ D_1\left(\frac{n}{\log n}\right) = O(\log n)$$

$$D_1\left(\frac{n}{\log n}\right) = D_1(n) - D_1(\log n) \leq D_1(n) O(\log n)$$

$$p = \frac{n}{\log n}$$