# Calculs préalables à l'algorithme v2

Henri de Boutray

1er juin 2018

Sorry for the non-French speakers, if needed, this document will be updated to be in english as well. For now, if anyone needs explanations, I'll be happy to help on the address provided on the project Github.

## 1 Introduction

Pour la version 1 de l'algorithme, nous avons considéré l'ensemble des diagrammes (avec des angles quelconques), mais d'autres cas d'utilisations peuvent nous intéresser. En effet, imposer un angle quelconque à un nœud est physiquement irréalisable pour l'instant. On se concentre donc sur des fragments, c'est à dire des sous ensembles des diagrammes possibles. On note un $p$-fragment un fragment ne possédant que des angles multiples de p. Pour des raisons techniques, le fragment $\pi/4$ est particulièrement intéressant. La deuxième version de l'algorithme va donc se concentrer sur se fragment.

## 2 Ensemble de définition

### 2.1 $\mathbb{Z}[1/2, e^{i\pi/4}]$

Le fragment $\pi/4$ nous ramène à $\mathbb{Z}[1/2, e^{i\pi/4}]$ . En effet, les nœuds verts sont de la forme

$$\begin{pmatrix} 1 & 0 & \dots & & 0 \\ 0 & 0 & & & \\ \vdots & & \ddots & & \vdots \\ & & & 0 & 0 \\ 0 & & \dots & 0 & e^{ik\pi/4} \end{pmatrix}$$

et les nœuds rouges sont de la forme $H^{\otimes n} V H^{\otimes m}$ où $V$ est le nœud vert correspondant, et H est la matrice de la porte hadamard, de la forme

$$1/\sqrt{2} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

donc, en se rappelant qu'on peut écrire $1/\sqrt{2}$ sous la forme $\dfrac{e^{i\pi/4} + e^{-i\pi/4}}{2}$, et par stabilité des opérations usuelles dans $\mathbb{Z}[1/2, e^{i\pi/4}]$ , on a bien l'entièreté du fragment $\pi/4$ dans $\mathbb{Z}[1/2, e^{i\pi/4}]$ .

De plus, un nombre $z$ dans $\mathbb{Z}[1/2, e^{i\pi/4}]$ peut s'écrire de la manière suivante :

$$z = \frac{a_0}{2^{n_0}} e^{0i\pi/4} + \frac{a_1}{2^{n_1}} e^{1i\pi/4} + \frac{a_2}{2^{n_2}} e^{2i\pi/4} + \frac{a_3}{2^{n_3}} e^{3i\pi/4}$$

Avec $\forall i \in [\![0, 3]\!], (a_i, n_i) \in \mathbb{Z} \times \mathbb{N}$
En effet, pour $k \in [\![4, 7]\!]$, $e^{ki\pi/4} = -e^{(k-4)i\pi/4}$
Si $z \neq 0$ peut imposer que les $a_i$ soient impairs, en effet, si ce n'est pas le cas, il suffit d'augmenter $n_i$ En divisant le tout par le $\frac{1}{2^{n_i}}$ avec le $n_i$ le plus grand, que l'on notera $p$, on obtient :

$$p = \max_{i \in [\![0,3]\!]} (n_i)$$

$$z = \frac{1}{2^p} \sum_{j=0}^{3} a_i 2^{p-n_i} e^{ij\pi/4}$$

soit $b_i = a_i 2^{p-n_i}$, on a alors

$$z = \frac{1}{2^p} \sum_{j=0}^{3} b_i e^{ij\pi/4}$$

Cette écriture sera désormais appelée écriture canonique de z pour z dans le fragment $\pi/4$, elle est unique si on impose en plus que $p = 0$ si $z = 0$.
Au lieu de manipuler des nombres complexes, on manipule donc 5 entiers : $p$ et les $b_i$. De plus, on a l'unicité de cette représentation, et pour garantir cette unicité, il suffit de vérifier qu'un des nombres est impair.
Pour tout $z$ dans $\mathbb{Z}[1/2, e^{i\pi/4}]$ , il existe donc une représentation $(p, b_0, b_1, b_2 b_3)$ qui peut s'obtenir de manière automatique. soit $D : \mathbb{Z}[1/2, e^{i\pi/4}] \mapsto \mathbb{N} \times \mathbb{Z}^4$ cette fonction. Elle a une réciproque triviale $D^{-1}$. (Attention, le terme de réciproque ici est abusif, en effet $D^{-1}(D(z)) = z$ mais $D(D^{-1}((a, b, c, d, e)) \neq (a, b, c, d, e)$. En fait, pour être exact, ce que nous avons noté $D^{-1}$ ici c'est que l'inverse à droite de D)
Soit $z_0$ avec $D(z_0) = (p_0, a_{0,0}, a_{0,1}, a_{0,2}a_{0,3})$ et $z_1$ avec $D(z_1) = (p_1, a_{1,0}, a_{1,1}, a_{1,2}a_{1,3})$ Alors

$$\begin{aligned} z_0 + z_1 &= \frac{1}{2^{p_0}} \sum_{j=0}^{3} a_{0,j} e^{ij\pi/4} + \frac{1}{2^{p_1}} \sum_{j=0}^{3} a_{1,j} e^{ij\pi/4} \\ &= \sum_{j=0}^{3} \left( a_{0,i} 2^{-p_0} + a_{1,j} 2^{-p_1} \right) e^{ij\pi/4} \end{aligned}$$

*donc*

$$z_0 + z_1 = D^{-1}\left(1, \left(a_{0,j} 2^{-p_0} + a_{1,j} 2^{-p_1}\right)_{j \in [\![0,3]\!]}\right)$$

Et de même

$$\begin{aligned} z_0 \times z_1 &= \left( \frac{1}{2^{p_0}} \sum_{j=0}^{3} a_{0,i} e^{ij\pi/4} \right) \times \left( \frac{1}{2^{p_1}} \sum_{j=0}^{3} a_{1,i} e^{ij\pi/4} \right) \\ &= \frac{1}{2^{p_0+p_1}} \sum_{j=0}^{3} \left( \sum_{k+l=j}^{(k,l) \in [\![0,3]\!]^2} a_{0,k} \times a_{1,l} \right) + e^{ij\pi/4} \end{aligned}$$

*donc*

$$z_0 \times z_1 = D^{-1}\left(p_0 + p_1, \left( \sum_{k+l=j}^{(k,l) \in [\![0,3]\!]^2} a_{0,k} \times a_{1,l} \right)_{j \in [\![0,3]\!]}\right)$$

Notons que $Im(D) \subsetneq \mathbb{N} \times \mathbb{Z}^4$. De plus, passer de $D \circ D^{-1}(a, b, c, d, e)$ n'est pas trivial à calculer : pour repasser en écriture canonique, il faut calculer la plus grande puissance de 2 divisant $(b, c, d, e)$. Si ce n'est pas une opération très lourde, opérée sur des milliers de nombres (dans le cas de certaines matrices représentatives de diagrammes de ZX), cela peut ralentir l'algorithme ne manière non négligeable. De plus amples tests seront fait quant à la faisabilité de cette opération au cour du calcul.

## 2.2 $M_{2^m,2^n}(\mathbb{Z}[1/2, e^{i\pi/4}])$

En appliquant les calculs précédents aux matrices, on peut écrire une matrice M avec ses coefficients en écriture canonique. Mais on peut aussi aller plus loin, en définissant $p' = max_{(k,l) \in [\![0,m-1]\!] \times [\![0,n-1]\!]} p_{k,l}$ avec $M = (z_{k,l})_{(k,l) \in [\![0,m-1]\!] \times [\![0,n-1]\!]}$ et $D(z_{k,l}) = \left(p_{k,l}, (a_{k,l,j})_{j \in [\![0,3]\!]}\right)$.
On peut alors écrire :

$$M = \frac{1}{2^{p'}} \sum_{j=0}^{3} M_j e^{ij\pi/4}$$

Où $M_i \in M_{2^m,2^n}(\mathbb{Z})$ et soit $M = 0$, soit il existe un coefficient impair dans une des composante $M_j$ de M.
On peut alors définir $D : M_{2^m,2^n}(\mathbb{Z}[1/2, e^{i\pi/4}]) \mapsto M_{2^m,2^n}(\mathbb{N}) \times M_{2^m,2^n}(\mathbb{Z})^4$ de manière analogue à la partie précédente et $D^{-1}$ son inverse à droite.
On a donc, comme avant :

$$M_0 + M_1 = D^{-1}\left(1, \left(M_{0,j}2^{-p_0} + M_{1,j}2^{-p_1}\right)_{j\in[\![0,3]\!]}\right)$$

Et :

$$M_0 \times M_1 = D^{-1}\left(p_0 + p_1, \left(\sum_{k+l=j}^{(k,l)\in[\![0,3]\!]^2} M_{0,k} \times M_{1,l}\right)_{j\in[\![0,3]\!]}\right)$$

On saisi mieux alors la difficulté d'obtention de $D(M)$ dans ce contexte. Indépendamment des opérations + et ×, trouver la représentation canonique de M demande des opérations sur $4(2^m \times 2^n)$ éléments ce qui est inconcevable dans le cas de grandes matrices, ou dans le cas d'opération effectuées souvent (l'algorithme se trouve dans ces deux cas).

Cependant, le rôle de $D$ n'est pas que de garantir l'unicité. La représentation canonique, en plus de garantir l'unicité, permet de prévenir l'underflow (un nombre trop petit dont des chiffres significatif sont perdus). L'overflow en revanche doit toujours être surveillée et géré correctement (A vérifier si il a des chances d'arriver, si les chances sont faibles, autant juste lever une erreur).

# 3 Sous parties de l'algorithme

L'idée de cet algorithme est de travailler par dichotomie. Une des fonctions principales est donc celle de la réunion de deux diagrammes dont les matrices ont été calculées. On va donc définir les points suivants :

- Point de connexion : une arrête est connectée à une entrée/sortie donnée de la matrice.

```
conection_point = {
    "matrix": bool,
    "is_out": matrix,
    "index": int
}
```

- Entrées :

```
inputs = [connection_point]
```

- Sorties :

```
outputs = [connection_point]
```

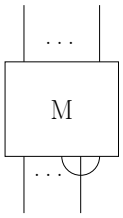
- Connexions entre matrices :

```
connections = [(connection_point, connection_point)]
```

ouf

## 3.1 Cup

Nous utiliserons l'appellation cup pour parler d'une arrête qui va d'une sortie d'une matrice vers une autre sortie de cette même matrice (le concept est plus général que pour juste un nœud, mais cette notion nous suffira ici) :




Ajouter une cup revient à forcer l'égalité entre deux vecteurs de base des lignes de la matrice. Prenons l'exemple d'une matrice $8 \times 4$ :

$$\begin{matrix} |000\rangle \\ |001\rangle \\ |010\rangle \\ |011\rangle \\ |100\rangle \\ |101\rangle \\ |110\rangle \\ |111\rangle \end{matrix} \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ & & & \\ & & & \\ & & \vdots & \\ & & & \\ & & & \\ & & & \\ a_{7,0} & a_{7,1} & a_{7,2} & a_{7,3} \end{pmatrix} \quad \text{Soit } L_i = (a_{i,0} \quad a_{i,1} \quad a_{i,2} \quad a_{i,3})$$

Alors, si on connecte la première sortie à la dernière, on cherche les vecteurs $|e_0 e_1 e_2\rangle$ où $e_0 = e_2$, on a donc $|000\rangle$ et $|010\rangle$ ainsi que $|101\rangle$ et $|111\rangle$. Donc :

$$M \to \begin{pmatrix} L_0 \\ L_2 \end{pmatrix} + \begin{pmatrix} L_5 \\ L_7 \end{pmatrix}$$

Pour expliquer cela autrement, nous avons perdu deux sorties, donc la hauteur de la matrice est divisée par $4 = 2^2$ et nous additionnons les différents cas possibles, en l'occurrence, ces sorties peuvent être à $|0\rangle$ ou à $|1\rangle$.

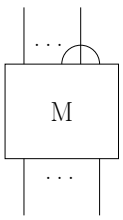
On peut généraliser cette idée à $n$ cups, mais ce n'est pas forcément intéressant au niveau temps de calculs, nous ne développerons donc pas cette partie.

Formule générale de connexion de la sortie $i$ à la sortie $j$ :
Soit $L$ l'ensemble des lignes de $M$
Soit $(e_k)_{k \in [\![0, 2^n - 1]\!]}$ la base des vecteurs avec $e_k = |e_{k,0} \dots e_{k,n-1}\rangle$
Soit $L^b = \{L_k \in L / e_{k,i} = e_{k,j} = b\}$ avec $b \in \{0, 1\}$
Soit $(L^b_k)_{k \in [\![0, 2^{n-2} - 1]\!]}$ l'ensemble $L^b$ ordonné selon l'indice initial des lignes dans $M$
Alors :

$$M \to \begin{pmatrix} L^0_0 \\ \vdots \\ L^0_{2^{n-2}-1} \end{pmatrix} + \begin{pmatrix} L^1_0 \\ \vdots \\ L^1_{2^{n-2}-1} \end{pmatrix}$$

## 3.2 Cap

De manière analogue à la partie précédente, on parlera de cap quand on connecte deux entrées. Les phénomènes sur les lignes se déroulerons alors sur les colonnes. Les calculs étant très proches



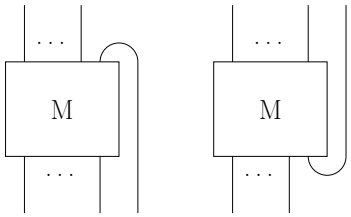

de ceux de la section précédente, nous ne donnerons que le résultat :

Formule générale de connexion de l'entrée $i$ à l'entrée $j$ :
Soit $C$ l'ensemble des colonnes de $M$
Soit $(e_k)_{k\in[\![0,2^m-1]\!]}$ la base des vecteurs avec $e_k = |e_{k,0}\dots e_{k,m-1}\rangle$
Soit $C^b = \{C_k \in C / e_{k,i} = ek, j = b\}$ avec $b \in \{0,1\}$
Soit $(C^b_k)_{k\in[\![0,2^{m-2}-1]\!]}$ l'ensemble $C^b$ ordonné selon l'indice initial des colonnes dans $M$
Alors :

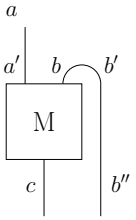
$$M \to \begin{pmatrix} C^0_0 & \dots & C^0_{2^{n-2}-1}\end{pmatrix} + \begin{pmatrix} C^1_0 & \dots & C^1_{2^{n-2}-1}\end{pmatrix}$$

## 3.3 Échange entrée-sortie

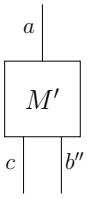
Les cup et cap précéda ment définis peuvent aussi servir à transformer une entrée en sortie ou inversement, comme suit.




Pour cette patrie, de nouvelles notations peuvent aider à avoir une vision plus claire du problème, Je vais donc les introduire avec un exemple :




La notation est la suivante : on annote chaque composante avec ses entrées/sorties ainsi : $M^{a'b}_c$, $\delta_{bb'}$, $\delta^a_{a'}$ et $\delta^{b'}_{b''}$. Avec les entrées symboliques notées en haut, et les sorties en bas. de cette manière, si on prend $M'$ de la manière suivante :




On peut faire la décomposition suivante : $M'^a_{cb''} = (\delta^a_{a'} \otimes \delta_{bb'}) \circ (M^{a'b}_c \otimes \delta^{b'}_{b''})$. Ici, les $\delta$ ont le sens usuel : c'est le symbole de Kronecker, ils imposent l'égalité entre les coefficients qui leurs sont passés, on peut alors écrire : $M'^a_{cb''} = M^{ab''}_c$. Et voilà le résultat qui nous intéresse : on peut remplacer chaque $a$, $b$ ou $c$ par $|0\rangle$ ou $|1\rangle$, on obtient alors les égalités suivantes :

$M'^0_{00} = M^{00}_0 \quad M'^1_{00} = M^{10}_0$
$M'^0_{01} = M^{01}_0 \quad M'^1_{01} = M^{11}_0$
$M'^0_{10} = M^{00}_1 \quad M'^1_{10} = M^{10}_1$
$M'^0_{11} = M^{01}_1 \quad M'^1_{11} = M^{11}_1$

On a donc pu calculer $M'$ sans autre opération que des lectures !
Il faut vérifier certains détails d'implémentation encore sur python, mais cela devrait être exponentiellement plus rapide que l'alternative qui aurait été de faire les calculs matriciels.
En fait, ce qui nous épargne les calculs matriciels ici est le fait que nous connaissons la forme des matrices et donc que nous pouvons prédire le résultat de certains calculs.

Formule d'échange entrées-sorties :
Soient $e = |e_i\rangle_{i\in[\![0,2^m-1]\!]}$ les entrées de $M$ et $s = |s_i\rangle_{i\in[\![0,2^n-1]\!]}$ les sorties.

Soit $\sigma$ les échanges ayant lieu entre les entrées et les sorties. ($\sigma$ prend en entrée deux ensembles et renvoie deux autre ensembles. En fait, $\sigma$ peut aussi encoder des permutations), alors

$$M_s^e \to M'^{\sigma(e\times s)[0]}_{\sigma(e\times s)[1]}$$

## 3.4 Permutations des entrées et sorties

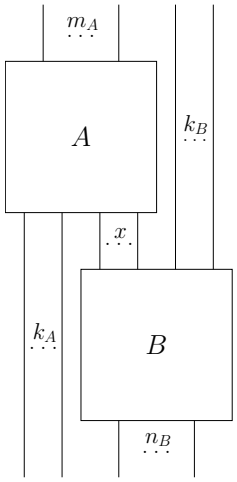
Cette partie concerne les permutations au sein d'un même ensemble (une entrée reste une entrée, une sortie reste une sortie).
On peu alors utiliser le résultat le la partie précédente, avec l'extension évoquée.
Dans ce cas $\sigma$ est de la forme suivante : $\sigma(a \times b) = \sigma_e(a) \times \sigma_s(b)$ où $\sigma_e$ est la permutation sur les entrées et $\sigma_s$ est celle sur les sorties.
Alors :

$$M_s^e \to M'^{\sigma_e(e)}_{\sigma_s(s)}$$

Pour interpréter ça : Une permutation sur les entrées du diagramme $M$ fait une permutation sur les colonnes de la matrice $M$ (les deux permutations évoquées sont liées, mais pas égales), et de même, une permutation sur les sorties fait une permutation sur les lignes.

## 3.5 Connexions incomplètes

Nous cherchons à unifier deux matrices. Notons les $A$ et $B$. Toutes les entrées du diagramme ne sont pas forcément connectées à l'une de ces deux matrices, on ne peut donc pas juste multiplier l'une par l'autre. Avec les méthodes vu précédemment, on peut considérer que les matrices que l'on cherche à réunir sont agencées comme suit :

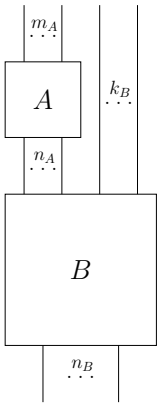



On a dans cette représentation $n_A = k_A + x$ et $m_B = k_B + x$ où $m_A$ et $m_B$ sont les nombres d'entrées respectifs des diagrammes $A$ et $B$. Notons que pour ne pas alourdir les écritures, on parle à la fois de $A$ et $B$ pour les diagrammes et pour les matrices correspondantes, en fonction du contexte).

### 3.5.1 $k_A = 0$

Pour résoudre ce problème, commençons par considérer que $k_A = 0$, alors :
La matrice représentant de diagramme vaut donc $(A \otimes I_{2^{k_b}}) \circ B$.
À ce point, on serait tentés de réécrire B afin d'avoir $I_{2^{k_b}} \otimes A$ plutôt, mais étant donné qu'il faudra étudier le cas où $k_B = 0$ après, et qu'il faudra réunir ces deux cas ensuite, ça ne serait que retarder le problème.

Coupons donc $B$ en matrices de dimensions $2^{n_B}, 2^{k_B}$, comme $m_B = k_B + x$, il y a donc $k_B$ ainsi construits.
Maintenant, définissons $B_i$ comme étant la concaténation de chaque $i^{ème}$ colonne des blocs précédemment construits.



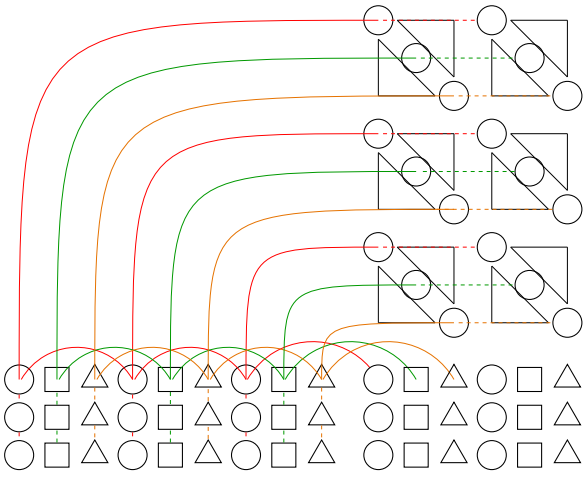

Soient $(a_{i,j})_{(i,j)\in[\![1,2^{n_A}]\!]\times[\![1,2^{m_A}]\!]}$ les coefficients de $A$, on a alors :

$$\begin{pmatrix} a_{1,1} & & 0 & \cdots & a_{1,2^{m_A}} & & 0 \\ & \ddots & & & & \ddots & \\ 0 & & a_{1,1} & & 0 & & a_{1,2^{m_A}} \\ \vdots & & \ddots & & \vdots & & \\ a_{2^{n_A},1} & & 0 & & a_{2^{n_A},2^{m_A}} & & 0 \\ & \ddots & & & & \ddots & \\ 0 & & a_{2^{n_A},1} & & 0 & & a_{2^{n_A},2^{m_A}} \end{pmatrix}$$

Soient $\left(b^i_{l,m}\right)_{(l,m)\in[\![1,2^{n_B}]\!]\times[\![1,2^{m_A}]\!]}$ les coefficients de $B_i$. Alors, en appelant $C$ le système global, et en découpant $C$ comme $B$ (en $C_i$ de taille $2^{n_B}\times 2^{k_B}$), on obtient $n_A$ blocs, avec :

$$C_i = B_i \times A$$

On peut visualiser cela avec l'exemple suivant montrant le début de la multiplication (à extrapoler pour avoir un exemple complet) :

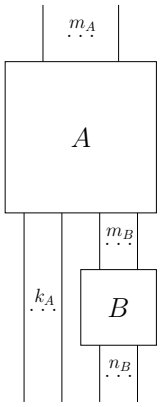


Pour faire cette opération, il faut donc :

- diviser $B$ en sous-matrices $B_i$
- obtenir les $C_i$ avec une multiplication matricielle
- reconstruire $C$ à partir des $C_i$

### 3.5.2 $k_B = 0$

Si on retombera sur un résultat conceptuel similaire, ce cas est en revanche plus simple sur l'aspect calculatoire.




En effet, dans ce cas, $I_{2^{k_A}} \otimes B$ est une matrice diagonale par blocs avec chaque bloc étant égal à $B$. C'est alors plus simple de voire que si on découpe A en lignes de hauteur $m_A$, on a $2^{k_A}$ telles lignes, notées $A_i$ et le résultat sera la concaténation de ces $B \times A_i$.

$$\begin{pmatrix} B & & 0 \\ & \ddots & \\ 0 & & B \end{pmatrix} \times \begin{pmatrix} A_0 \\ \vdots \\ A_{2^{k_i}} \end{pmatrix} = \begin{pmatrix} BA_0 \\ \vdots \\ BA_{2^{k_i}} \end{pmatrix}$$

Donc là encore, pour faire cette opération, il faut :

- diviser $A$ en sous-matrices $A_i$
- obtenir les $C_i$ avec une multiplication matricielle
- reconstruire $C$ à partir des $C_i$

Mais l'obtention des $A_i$ et la reconstruction de C sont bien plus simple !

### 3.5.3 Généralisation

Cette solution simple et la solution précédente plus complexe doivent se généraliser, on devrait même retrouver le cas du produit tensoriel quand $x = 0$.

Bilan des variables :

- matrice $A$, de dimensions $2^{n_A}$ (hauteur), $2^{m_A}$ (largeur)
- $(a_{i,j})_{(i,j)\in[\![0,2^{n_A}-1]\!]\times[\![0,2^{m_A}-1]\!]}$ les coefficients de $A$
- matrice $B$, de dimensions $2^{n_B}$ (hauteur), $2^{m_B}$ (largeur)
- $(b_{i,j})_{(i,j)\in[\![0,2^{n_B}-1]\!]\times[\![0,2^{m_B}-1]\!]}$ les coefficients de $B$
- $x$, le nombre de fils allant de $A$ à $B$
- $k_A$, le nombre de fils sortant de $A$ n'allant pas à $B$
- $k_B$, le nombre de fils allant à $B$ ne sortant pas de $A$
- matrice $C$, de dimensions $2^{n_B+k_A}$ (hauteur), $2^{m_A+k_B}$ (largeur)
- $(C_{i,j})_{(i,j)\in[\![0,2^{n_B+k_A}-1]\!]\times[\![0,2^{m_A+k_B}-1]\!]}$ les coefficients de $C$

De plus, on conserve les notations et opérations introduites dans les deux sections précédentes :

- $A$ est découpé en $(A_i)_{i\in[\![0,2^{k_A}-1]\!]}$, avec $A_i = (a_{k,l})_{(k,l)\in[\![2^x i,2^x(i+1)-1]\!]\times[\![0,2^{m_A}-1]\!]}$
- $B$ est découpé en $(B_i)_{i\in[\![0,2^{k_B}-1]\!]}$, avec $B_i = (b_{k,i+l\times 2^{k_b}})_{(k,l)\in[\![0,2^{n_B}-1]\!]\times[\![0,2^x-1]\!]}$

En revanche, $C$ doit avoir un nouveau découpage, en effet, on sent intuitivement que si $A$ et $B$ sont découpées, le découpage de $C$ sera de dimension 2. On peut même prévoir que ce découpage se fera comme $A$ pour les lignes et comme $B$ pour les colonnes.

On peut montrer que cela fonctionne, et on a alors

$$C_{i,j} = B_j A_i$$

Avec les coefficients de $C_{i,j}$ de la forme suivante :

$$C_{i,j} = (c_{k,l\times 2^{k_B}})_{(k,l)\in[\![2^{n_B}i,(2^{n_B}+1)i-1]\!]\times[\![j,2^{m_A}+j-1]\!]}$$

$$C_{i,j} = (c_{i\times 2^{n_B}+k,j+l\times 2^{k_B}})_{(k,l)\in[\![0,2^{n_B}-1]\!]\times[\![0,2^{m_A}-1]\!]}$$

De plus, on retrouve bien le produit tensoriel si $x = 0$, en effet :

$$A_i = \begin{pmatrix} a_{i,0} & \dots & a_{i,2^{m_A}} \end{pmatrix}, B_i = \begin{pmatrix} b_{0,i} \\ \vdots \\ b_{2^{n_B},i} \end{pmatrix}$$

Donc :

$$C_{i,j} = \begin{pmatrix} b_{0,j}a_{i,0} & \dots & b_{0,j}a_{i,2^{m_A}} \\ \vdots & & \vdots \\ b_{2^{n_B},j}a_{i,0} & \dots & b_{2^{n_B},j}a_{i,2^{m_A}} \end{pmatrix}$$

ou, autrement écrit :

$$C_{i,j} = \begin{pmatrix} B_0 a_{i,0} & \dots & B_j a_{i,2^{m_A}} \end{pmatrix}$$

Donc :

$$C = \begin{pmatrix} B_0a_{0,0} & \dots & B_{2^{m_B}}a_{0,0} & \dots & B_0a_{0,0} & \dots & B_{2^{m_B}}a_{0,2^{m_A}} \\ & \vdots & & & & \vdots & \\ B_0a_{2^{n_A},0} & \dots & B_{2^{m_B}}a_{2^{n_A},0} & \dots & B_0a_{2^{n_A},0} & \dots & B_{2^{m_B}}a_{2^{n_A},2^{m_A}} \end{pmatrix}$$

On retrouve bien :

$$C = \begin{pmatrix} a_{0,0}B & \dots & a_{0,2^{m_A}}B \\ \vdots & & \vdots \\ a_{2^{n_A},0}B & \dots & a_{2^{n_A},2^{m_A}}B \end{pmatrix}$$

## 3.6 Évolution de la complexité

### 3.6.1 Cup et cap

Dans l'algorithme précédent, ce cas n'existait pas grâce à la méthode de parcourt du diagramme, cependant, sa philosophie aurait été de créer une porte cap et une porte cup, de calculer le produit de ces portes et de l'identité et enfin de multiplier $M$ à la matrice ainsi obtenue.

Soient $2^m \times 2^n$ les dimensions de $M$, la nouvelle matrice aurait alors été de dimension $2^n \times 2^{n-2}$, pour l'obtenir, il aurait fallu faire $n-2$ produits tensoriels, avec pour chaque produit tensoriel un maximum de $2n'^2$ lectures et écritures où $n'$ est la dimension de la plus grosse matrice du produit.

Or le produit tensoriel demande une opération par coefficients de la matrice résultante, soit $2^{2n-2}$ coefficients dans notre cas. Pour le produit classique qui utilise numpy, la complexité dépend de nombreux paramètres (elle peut être creuse [1] ou pas [2], et le type des coefficients joue aussi sur la complexité à cause de BLAS [3]) on admet donc que la complexité de la multiplication utilisée est de l'ordre de $n^{2.5}$ pour une matrice carré de taille $n$ (on peut trouver mieux par exemple dans [4]). Cependant, on ne peut pas juste considérer la complexité, en effet, le produit tensoriel est fait en python alors que la multiplication est faite par l'intermédiaire de numpy, on a donc une constante multiplicative très importante : en dessous de matrices de taille 1024, le produit tensoriel

est plus lent que la multiplication (ceci dit, une matrice de taille 1024 n'est pas absurde puisque cela ne correspond qu'à 10 entrées et 10 sorties).

Étant donné que nous travaillerons potentiellement souvent en grandes dimensions, prenons $2^{2.5n}$ comme notre ordre de grandeur de complexité globale.

Avec l'amélioration proposée, nous ne faisons plus que $2^{n-2}$ additions, donc, sans compter que nous n'avons plus de multiplications (donc des opérations plus rapides à effectuer), nous sommes passé de :

$$\mathcal{O}(2^{2.5n}) \text{ à } \mathcal{O}(2^n)$$

opérations.

### 3.6.2 Échange entrée-sortie

Pour l'échange d'une entrée en une sortie ou l'inverse, là encore, l'algorithme précédent ne pouvait pas tomber dans ce genre de situations, mais en suivant les principes de conceptions, on aurait eu : $(M \otimes I_2) \times (I_{2^{m-1}} \otimes Cup)$ soit un nombre de calculs de l'ordre de $2^{n-2}$. Or là, nous sommes passé à aucun calcul, que des réécritures (instancier une matrice peut tout de même être assez long). Nous sommes donc passé de :

$$\mathcal{O}(2^{2.5n}) \text{ à } 0$$

opérations. Pour un temps de calcul proportionnel à $2^n$.

### 3.6.3 Permutations

On est dans un cas similaire à la partie précédente, cette fois ci, c'était bien un calcul régulièrement exécuté, durant lequel on construisait la matrice de permutation $P$ en un temps proportionnel à $2^n$ puis on calculait $M \times P$, obtenant donc ainsi un temps de calcul en $\mathcal{O}(2^{2.5n})$ et là encore, nous sommes passés à 0 opérations arithmétiques mais un temps de calcul total de l'ordre de $2^n$ à cause de l'initialisation de la nouvelle matrice.

### 3.6.4 Connexions incomplètes

Une fois de plus, ce cas n'était pas présent dans la version 1, mais l'idée aurait été : de calculer une première matrice de permutation $P_1$, de calculer $A \otimes I_{2^{k_B}}$, de calculer une deuxième matrice de permutation $P_2$, de calculer $I_{2^{k_B}} \otimes B$, de calculer une troisième matrice de permutation $P_3$, de calculer $P_3 \times (I_{2^{k_B}} \otimes B) \times P_2 \times (A \otimes I_{2^{k_B}}) \times P_1$. La complexité de chaque étape de calcul serait alors de :

(1) $2^{2(m_A+k_B)}$

(2) $2^{2(m_A+k_B)+n_A}$

(3) $2^{2(m_B+k_A)}$

(4) $2^{2(n_A+k_B)+(m_A+k_B)}$

(5) $2^{2(m_B+k_A)+(n_B+k_A)}$

(6) $2^{2(n_B+k_A)}$

(7) $2^{2(n_B+k_A)+m_B}$

soit un ordre de $2^{2(m_A+k_B)+n_A} + 2^{2(m_B+k_A)+m_A} + 2^{2(m_B+k_A)+n_B} + 2^{2(n_B+k_A)+m_B}$. Or avec l'alternative, on fait $2^{k_A+k_B}$ multiplication de matrices de taille $x \times m_A$ et $n_B \times x$. La multiplication d'une de ces matrices a une complexité en $2^{x+m_A+n_B}$ donc la complexité globale est de $2^{x+m_A+n_B+k_A+k_B}$. Comparons ce résultat aux quatre composantes du résultat précédent :

$$\begin{cases} x + m_A + n_B + k_A + k_B - (2(m_A + k_B) + n_A) = n_B - (m_A + k_B) \\ x + m_A + n_B + k_A + k_B - (2(n_A + k_B) + (m_A + k_B)) = n_B - (m_B + k_A + k_B) \\ x + m_A + n_B + k_A + k_B - (2(m_B + k_A) + (n_B + k_A)) = m_A - (m_B + k_A + k_B) \\ x + m_A + n_B + k_A + k_B - (2(n_B + k_A) + m_B) = m_A - (n_B + k_A) \end{cases}$$

Ces quatre termes ne peuvent pas être simultanément positifs, en effet, si le premier l'est, $n_B \geq m_A + b_B$ donc $m_A \leq (n_B + k_A)$ donc le quatrième ne l'est pas, et réciproquement.

Donc on a gagné en complexité (sans compter qu'il y a bien moins d'étapes dans cette nouvelle version et qu'on a donc moins de ralentissement dus à l'utilisation de Python).

Calculs additionnels :

Si on veut évaluer plus précisément l'amélioration, on peut faire le raisonnement suivant. Comme les multiplications sont les opérations les plus coûteuses asymptotiquement dans l'algorithme v1, ne considérons qu'elles. On a donc 4 multiplications possibles, leur complexité sont données plus haut. Appelons $(N)$ le nouvel algorithme et $(1), (2), (3), (4)$ les différentes multiplications, alors :

Si $(N)$ est plus lent que (1), $(N)$ est plus rapide que (4) d'un facteur au moins $2^{k_A+k_B}$.

De même, si $(N)$ est plus lent que (4), $(N)$ est plus rapide que (1) d'un facteur au moins $2^{k_A+k_B}$.

Enfin, dans les autres cas, (2) ou (3) prédominent, dans ces cas :

Si (2) prédomine, $(N)$ est une amélioration sur (2) d'un facteur au moins $2^{k_A}$.

Si (3) prédomine, $(N)$ est une amélioration sur (3) d'un facteur au moins $2^{k_B}$.

Démonstration :

si

$$\begin{aligned}(N) \geq (4) : \; x + k_A + k_B + m_A + n_B &\geq m_B + 2n_B + 2k_A \\ x + k_B + m_A &\geq n_A + n_B + k_B \\ m_A &\geq n_A + n_B - x \\ m_A &\geq k_A + n_B \\ -k_A &\geq n_B - m_A\end{aligned}$$

or

$$\begin{aligned}(N) - (1) : \; x + k_A + k_B + m_A + n_B \; &-(n_A + 2m_A + 2k_B) \\ &= x + k_A + n_B - (n_A + m_A + k_B) \\ &= x + k_A + n_B - m_B + m_A + k_A \\ &= n_B - m_A - k_B \leq -k_A - k_B\end{aligned}$$

On obtient le résultat de la même manière si $(N) \geq (1)$.

Si (2) ou (3) prédominent il faut examiner ce qu'il se passe :

Si (2) prédomine :

①

$$\begin{cases} m_A + 2n_A + 3k_B \geq n_A + 2m_A + 2k_B \\ n_A + k_B \geq m_A - m_A \geq -n_A - k_B \end{cases}$$

②

$$\begin{cases} m_A + 2n_A + 3k_B \geq n_B + 2m_B + 3k_A \\ m_A + 2n_A + 3k_B \geq n_B + k_A + 2n_A + 2k_B \\ m_A + k_B \geq n_B + k_A \end{cases}$$

③

$$\begin{cases} m_A + 2n_A + 3k_B \geq m_B + 2n_B + 2k_A \\ m_A + 2n_A + 3k_B \geq n_A + k_B + 2n_B + k_A \\ m_A + n_A + 2k_B \geq 2n_B + k_A \end{cases}$$

(rappelons que $x$ représente le recouvrement entre $A$ et $B$, donc on a $k_A + x = n_A$ et $k_B + x = m_B$ en particulier)

Soit $Y = m_A + 2n_A + 3k_B$ et $X = x + k_A + k_B + m_A + n_B$, alors ② nous donne $Y \geq n_B + 2n_A + 2k_B + k_A$ et

$$\begin{aligned}n_B + 2n_A + 2k_B + k_A &= n_B + m_A + k_B + k_A + 2n_A + k_B - m_A \\ &= X - x + n_A + m_B + k_A - m_A \\ &= X + x + 2k_A + k_B - m_A \\ &= X + n_A + k_A + k_B + m_A\end{aligned}$$

Donc $Y \geq X + n_A + k_A + k_B + m_A$ et avec ①, on trouve finalement que $Y \geq X + k_B$. Avec l'étape (2) de l'ancien algorithme ayant une complexité en $2^Y$ et le nouvel algorithme ayant une complexité en $2^X$, d'où l'amélioration d'un facteur $2^{k_B}$.

Pour avoir le résultat si (3) prédomine, la méthode est très proche

# Références


[1] R. Yuster and U. Zwick, "Fast sparse matrix multiplication." [Online]. Available : http://www.cs.tau.ac.il/~zwick/papers/sparse.pdf

[2] F. Le Gall, "Faster algorithms for rectangular matrix multiplication," 2012. [Online]. Available : https://arxiv.org/pdf/1204.1111.pdf

[3] U. of Tennessee, "Basic linear algebra subprograms technical forum standard," 2001. [Online]. Available : http://www.netlib.org/blas/blast-forum/blas-report.pdf

[4] V. Strassen, "Relative bilinear complexity and matrix multiplication." *Journal für die reine und angewandte Mathematik*, vol. 0375_0376, pp. 406–443, 1987, document from : http ://eudml.org/doc/152921. [Online]. Available : https://gdz.sub.uni-goettingen.de/id/PPN243919689_0375_0376?tify={"pages":[410],"view":"info"}