De la théorie des automate au taitement de données efficace

Charles Paperman

Université de Lille

Janvier 2025 – CRIStAL

Une fonction très simple

```
def search(s: str, lettre: str):
    for i, a in enumerate(s):
        if a == letter:
            return i
    return None
```

Une fonction très simple

```
def search(s: str, lettre: str):
    for i, a in enumerate(s):
        if a == letter:
            return i
    return None
```

- **Bande passante** de 23 Mo/s

Une fonction très simple

```python
def search(s: str, lettre: str):
    for i, a in enumerate(s):
        if a == letter:
            return i
    return None
```

```python
def search(s: str, lettre: str):
    try:
        return s.index(lettre)
    except ValueError:
        return None
```

- **Bande passante** de 23 Mo/s

Une fonction très simple

```
def search(s: str, lettre: str):
    for i, a in enumerate(s):
        if a == letter:
            return i
    return None
```

- **Bande passante** de 23 Mo/s

```
def search(s: str, lettre: str):
    try:
        return s.index(lettre)
    except ValueError:
        return None
```

- **Bande passante** 40 Go/s

Version compilée en C

```
char * search(char * buffer, char value, size_t length){
        for (size_t i=0; i<length; i++)
                if (buffer[i] == value)
                    return buffer + i;
        return NULL;
}
```

Version compilée en C

```
char * search(char * buffer, char value, size_t length){
        for (size_t i=0; i<length; i++)
                if (buffer[i] == value)
                    return buffer + i;
        return NULL;
}
```

- **Bande passante sans optimisation de compilation** 1.0 Go/s
- **Bande passante avec -Ofast** 3.7 Go/s

Version compilée en C

```
char * search(char * buffer, char value, size_t length){
        for (size_t i=0; i<length; i++)
                if (buffer[i] == value)
                    return buffer + i;
        return NULL;
}
```

- **Bande passante sans optimisation de compilation** 1.0 Go/s
- **Bande passante avec -Ofast** 3.7 Go/s

Comment fait Python pour être plus rapide que ça? Quels sont les goulots d'étranglement?

Utilisons la librairie standard !

```
void *memchr(const void s[.n], char c, size_t n);
// The  memchr() function scans the initial n bytes
// of the memory area pointed to by s for the first instance of c.
```

Implémentation	naïve C	memchr
GByte/s	3.7	57
Cycles/Octet	1	0.06
Instr/Octet	5	0.1

- Le goulot d'étranglement de la version *pure C* n'est pas la RAM mais la vitesse de traitement du CPU.
- Comment est-ce possible d'avoir moins Instr/Octet < 1 ?

Utilisons la librairie standard !

```
void *memchr(const void s[.n], char c, size_t n);
// The  memchr() function scans the initial n bytes
// of the memory area pointed to by s for the first instance of c.
```

Implémentation	naïve C	memchr
GByte/s	3.7	57
Cycles/Octet	1	0.06
Instr/Octet	5	0.1

- Le goulot d'étranglement de la version *pure C* n'est pas la RAM mais la vitesse de traitement du CPU.
- Comment est-ce possible d'avoir moins Instr/Octet < 1 ?

Parallélisme ... sur un seul cœur?

Parallélisme intra-registre

On peut stocker 8 octets dans un registre 64 bits. Il est donc possible de faire un peu de parallélisme:

1. On voit un entier 64bits comme un tableau de 8 Octets
2. On construit un entier 64 bits qui contient 8 copies de la valeur qu'on recherche
3. On passe au travers le tableau par morceau de 64 bits.

Implémentation

```
void * search(char * buffer, char value, size_t length){
    // mask = {value, ..., value} stored on 8Bytes on one integer
    for (size_t i=0; i<length/8; i++){
        u64 xored = ((u64 *) buffer)[i] ^ mask;
        u64 res1 = (xored >> 1) & xored;
        u64 res2 = (res1 >> 2) & res1;
        u64 res3 = (res2 >> 4) & res2;
        if ((res3 & 0x0101010101010101) != 0)
            break;
    }
    size_t j = 8*(i - 1);
    for (; j<length; j++)
        if (buffer[i] == value) return buffer + i;
    return NULL;
}
```

Mesure

Implémentation	naïve C	64bits	`glibc` version
GByte/s	3.7	9.8	57
Cycles/Octet	1	0.4	0.06
Instr/Octet	5	2.1	0.1

Registres vectoriels

On peut améliorer la version précédente en utilisant des **registres vectoriels** qui peuvent stocker 16, 32 ou même 64 octets.

Registres vectoriels

On peut améliorer la version précédente en utilisant des **registres vectoriels** qui peuvent stocker 16, 32 ou même 64 octets.

```
void * search (char *s, char c, size_t n){
    vector only_c = load_mask(c);
    for (size_t i = 0 ; i < n ; i += vector_size) {
        vector chunk = load(s + i);
        mask c_mask  = maskFromVector(vectorEq(chunk, only_c));
        if (cmask != 0)
            return (char *)(s + i + ctz(offsets));
    }
}
```

Registres vectoriels

On peut améliorer la version précédente en utilisant des **registres vectoriels** qui peuvent stocker 16, 32 ou même 64 octets.

```
void * search (char *s, char c, size_t n){
    vector only_c = load_mask(c);
    for (size_t i = 0 ; i < n ; i += vector_size) {
        vector chunk = load(s + i);
        mask c_mask  = maskFromVector(vectorEq(chunk, only_c));
        if (cmask != 0)
            return (char *)(s + i + ctz(offsets));
    }
}
```

Il s'agit (+/-) de l'implémentation de `memchr` de la `glibc`.

Moralité

1. Beaucoup d'algorithmes simples ont comme goulot d'étranglement le CPU.

Moralité

1. Beaucoup d'algorithmes simples ont comme goulot d'étranglement le CPU.
2. Pour accélérer ces implémentations il y a des instructions particulières qui dépendent du modèle de processeur.

Moralité

1. Beaucoup d'algorithmes simples ont comme goulot d'étranglement le CPU.
2. Pour accélérer ces implémentations il y a des instructions particulières qui dépendent du modèle de processeur.
3. Dans certains cas, le compilateur peut générer du code vectoriel ... mais pas tous le temps.

Moralité

1. Beaucoup d'algorithmes simples ont comme goulot d'étranglement le CPU.
2. Pour accélérer ces implémentations il y a des instructions particulières qui dépendent du modèle de processeur.
3. Dans certains cas, le compilateur peut générer du code vectoriel ... mais pas tous le temps.
4. C'est un vieux problème de compilation (1978), très actuel en traitement de données.

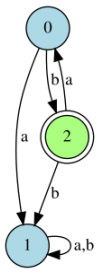
Étude de case: les automates

- `search` est l'automate qui correspond à l'expression régulières $(\Sigma \backslash \{a\})^*$
- Les automates sont les exemples les plus simples non auto-vectorisé par les compilateurs

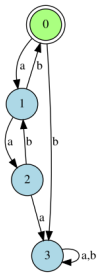
Étude de case: les automates

- `search` est l'automate qui correspond à l'expression régulières $(\Sigma \backslash \{a\})^*$
- Les automates sont les exemples les plus simples non auto-vectorisé par les compilateurs

Deux exemples filés



$(ab)^*$



$(a(ab)^*b)^*$

Intermède logique et automate

- On peut décrire certain langage calculés par automates par des formules de la logique du premier ordre.
- **Exemple.** $(ab)^*$:= "Les mots qui commencent par a, terminent par b et n'ont pas ni deux a consécutifs, ni deux b consécutifs''

$$a(\min) \ \wedge b(\max) \wedge \neg\exists x.(a(x) \wedge a(x+1)) \ \vee (b(x) \wedge b(x+1))$$

Intermède logique et automate

- On peut décrire certain langage calculés par automates par des formules de la logique du premier ordre.
- **Exemple.** $(ab)^*$:= "Les mots qui commencent par a, terminent par b et n'ont pas ni deux a consécutifs, ni deux b consécutifs''

$$a(\min) \ \wedge b(\max) \wedge \neg\exists x.(a(x) \wedge a(x+1)) \ \vee (b(x) \wedge b(x+1))$$

- Alternativement: `les mots de taille pairs avec toutes les positions pairs qui sont des $a$ et toutes les impairs qui sont des $b$`.

$$b(\max) \wedge \forall x, x \equiv 0 \bmod 2 \leftrightarrow a(x)$$

De la logique aux programmes efficaces pour (ab)*: baseline

```
void * search(char *s, size_t n){
    char state = 'b';
    for (size_t i=0; i<n; i++){
            if (s[i] == 'a' && state == 'b')
                state = 'a';
            else if (s[i] == 'b' && state == 'a')
                state = 'b';
            else
                return (char *)(s + i);
    }
    return NULL;
}
```

De la logique aux programmes efficaces pour (ab)*: successeur

```
void * search(char *s, size_t n){
    vector va = load_mask('a');
    vector vb = load_mask('b');
    mask state = 0;
    for (size_t i = 0 ; i < n ; i += vector_size) {
        vector chunk = load(s + i); // load the chunk of the stream
        mask mask_a = maskFromVector(vectorEq(chunk, va));
        mask mask_b = maskFromVector(vectorEq(chunk, vb));
        mask shift_a = state | (mask_a << 1); //shift on the right;
        state = mask_a >> (vector_size - 1); //keep the first bit
        // ctz returns the offset of the left most 1 in mask
        mask offset = shift_a ^ mask_b;
        if (offset != 0)  return (char *)(s + i + ctz(offset));
    }
    return NULL;
}
```

De la logique aux programmes efficaces pour (ab)*: modulo 2

```
char * search(char *s, size_t n){
    vector mask_a = load_mask('a');
    vector mask_b = load_mask('b');
    mask mod2= 0b10101010101010101010101010101010;
    for (size_t i = 0 ; i < n ; i += vector_size) {
        vector chunk = load(s + i);
        vector a_in_chunk = vectorEq(chunk, mask_a);
        vector b_in_chunk = vectorEq(chunk, mask_b);
        mask offset_a = maskFromVector(a_in_chunk);
        mask offset_b = maskFromVector(b_in_chunk);
        mask result = ((~offset_a) ^ mod2) | (offset_b^mod2);
        if ( result != 0 )
            return (char *)(s + i + ctz(result));
    }
    return NULL;
}
```

Mesures

Implémentation	naïve C	Shift	Mod
GByte/s	2.1	36.7	40.2
Cycles/Octet	1.1	0.1	0.09
Instr/Octet	11	0.41	0.34

Mesures

Implémentation	naïve C	Shift	Mod
GByte/s	2.1	36.7	40.2
Cycles/Octet	1.1	0.1	0.09
Instr/Octet	11	0.41	0.34

Comment trouver ces chouettes formules logiques ?

Théorie algébrique des automates

- Un monoïde est un ensemble muni d'une loi multiplicative **associative** et d'un élément neutre (par exemple Σ^*)

Théorie algébrique des automates

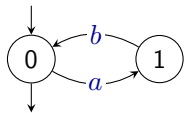
- Un monoïde est un ensemble muni d'une loi multiplicative **associative** et d'un élément neutre (par exemple Σ^*)
- Le monoïde **syntaxique** d'un langage régulier est le monoïde des *fonctions de transitions de l'automate minimal*.

Théorie algébrique des automates

- Un monoïde est un ensemble muni d'une loi multiplicative **associative** et d'un élément neutre (par exemple Σ^*)
- Le monoïde **syntaxique** d'un langage régulier est le monoïde des *fonctions de transitions de l'automate minimal*.
- **Exemple**: $(ab)^*$

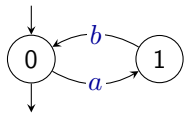
Théorie algébrique des automates

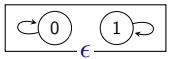
- Un monoïde est un ensemble muni d'une loi multiplicative **associative** et d'un élément neutre (par exemple Σ^*)
- Le monoïde **syntaxique** d'un langage régulier est le monoïde des *fonctions de transitions de l'automate minimal*.
- **Exemple**: $(ab)^*$



Théorie algébrique des automates

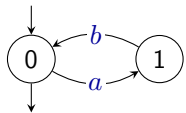
- Un monoïde est un ensemble muni d'une loi multiplicative **associative** et d'un élément neutre (par exemple Σ^*)
- Le monoïde **syntaxique** d'un langage régulier est le monoïde des *fonctions de transitions de l'automate minimal*.
- **Exemple**: $(ab)^*$

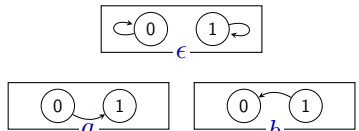




Théorie algébrique des automates

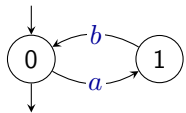
- Un monoïde est un ensemble muni d'une loi multiplicative **associative** et d'un élément neutre (par exemple Σ^*)
- Le monoïde **syntaxique** d'un langage régulier est le monoïde des *fonctions de transitions de l'automate minimal*.
- **Exemple**: $(ab)^*$

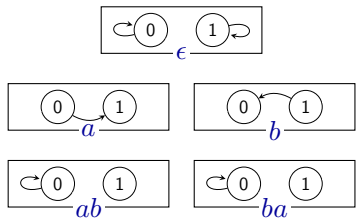




Théorie algébrique des automates

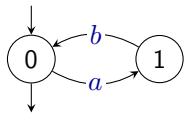
- Un monoïde est un ensemble muni d'une loi multiplicative **associative** et d'un élément neutre (par exemple Σ^*)
- Le monoïde **syntaxique** d'un langage régulier est le monoïde des *fonctions de transitions de l'automate minimal*.
- **Exemple**: $(ab)^*$

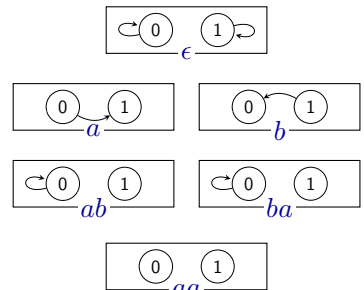




Théorie algébrique des automates

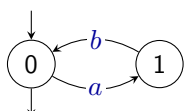
- Un monoïde est un ensemble muni d'une loi multiplicative **associative** et d'un élément neutre (par exemple Σ^*)
- Le monoïde **syntaxique** d'un langage régulier est le monoïde des *fonctions de transitions de l'automate minimal*.
- **Exemple**: $(ab)^*$



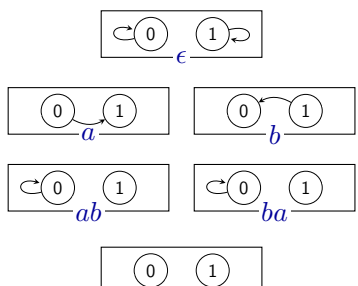


Théorie algébrique des automates

- Un monoïde est un ensemble muni d'une loi multiplicative **associative** et d'un élément neutre (par exemple Σ^*)
- Le monoïde **syntaxique** d'un langage régulier est le monoïde des *fonctions de transitions de l'automate minimal*.
- **Exemple**: $(ab)^*$

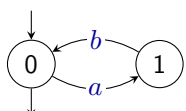


- $aba = a$
- $bab = b$
- $aa = bb$

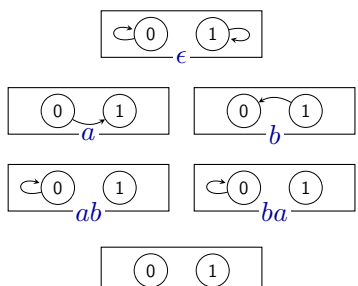


Théorie algébrique des automates

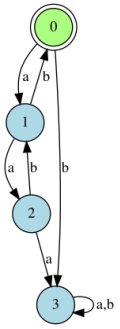
- Un monoïde est un ensemble muni d'une loi multiplicative **associative** et d'un élément neutre (par exemple Σ^*)
- Le monoïde **syntaxique** d'un langage régulier est le monoïde des *fonctions de transitions de l'automate minimal*.
- **Exemple**: $(ab)^*$



- $aba = a$
- $bab = b$
- $aa = bb$



Intermède algèbre et automate



Automate pour L_2

1	

ab	a
b	ba

bbaa	bba	bb
baa	baab	abb
aa	aab	aabb

bbb

Monoïde syntaxique de L_2

Quelques résultats du domaine

Théorème de Schützenberger, McNaughton et Papert

Un langage régulier est définissable dans la logique du premier ordre si son **monoïde syntaxique** vérifie que $x^n = x^{n+1}$ à partir d'un certain rang (**apériodique**).

Quelques résultats du domaine

Théorème de Schützenberger, McNaughton et Papert

Un langage régulier est définissable dans la logique du premier ordre si son **monoïde syntaxique** vérifie que $x^n = x^{n+1}$ à partir d'un certain rang (**apériodique**).

Preuve

Introspection de la structure interne des semigroupes, décomposition en produit semi-direct, généralisation des théorèmes de Jordan-Holder de la théorie des groupes.

Quelques résultats du domaine

Théorème de Schützenberger, McNaughton et Papert

Un langage régulier est définissable dans la logique du premier ordre si son **monoïde syntaxique** vérifie que $x^n = x^{n+1}$ à partir d'un certain rang (**apériodique**).

Preuve

Introspection de la structure interne des semigroupes, décomposition en produit semi-direct, généralisation des théorèmes de Jordan-Holder de la théorie des groupes.

Complexité

Ces résultats sont **effectifs** ...

Quelques résultats du domaine

Théorème de Schützenberger, McNaughton et Papert

Un langage régulier est définissable dans la logique du premier ordre si son **monoïde syntaxique** vérifie que $x^n = x^{n+1}$ à partir d'un certain rang (**apériodique**).

Preuve

Introspection de la structure interne des semigroupes, décomposition en produit semi-direct, généralisation des théorèmes de Jordan-Holder de la théorie des groupes.

Complexité

Ces résultats sont **effectifs** ... mais très couteux: **PSPACE-complet**

Retour sur le second exemple: $L_2 = (a(ab)^*b)^*$

- Pour $\sigma \in \Sigma$, on pose $\mathrm{double}_\sigma(x) := \sigma(x) \wedge \sigma(x+1)$ (*en x il y a le facteur $\sigma\sigma$*)

Retour sur le second exemple: $L_2 = (a(ab)^*b)^*$

- Pour $\sigma \in \Sigma$, on pose $\mathrm{double}_\sigma(x) := \sigma(x) \wedge \sigma(x+1)$ (*en x il y a le facteur $\sigma\sigma$*)

$$
\begin{aligned}
& a(\min) \wedge b(\max) \\
\wedge & \forall x < y, \mathrm{double}_a(x) \wedge \mathrm{double}_a(y) \wedge \exists z, x < z < y \wedge \mathrm{double}_b(z) \\
\wedge & \forall x < y, \mathrm{double}_b(x) \wedge \mathrm{double}_b(y) \wedge \exists z, x < z < y \wedge \mathrm{double}_a(z)
\end{aligned}
$$

Retour sur le second exemple: $L_2 = (a(ab)^*b)^*$

- Pour $\sigma \in \Sigma$, on pose $\mathrm{double}_\sigma(x) := \sigma(x) \wedge \sigma(x+1)$ (*en x il y a le facteur $\sigma\sigma$*)

$$
\begin{aligned}
& a(\min) \wedge b(\max) \\
\wedge & \forall x < y, \mathrm{double}_a(x) \wedge \mathrm{double}_a(y) \wedge \exists z, x < z < y \wedge \mathrm{double}_b(z) \\
\wedge & \forall x < y, \mathrm{double}_b(x) \wedge \mathrm{double}_b(y) \wedge \exists z, x < z < y \wedge \mathrm{double}_a(z)
\end{aligned}
$$

- Comment transformer ça en programme vectoriel ?

Représentation vectoriel du calcul fait sur $(a(ab)^*b)^*$

- On pose $\vec{a}$ et $\vec{b}$ des tableaux de booléen qui représentent les positions de a et de b dans l'entrée (*bitmap*)

Pour $aaba$, on a $\vec{a} = 1101$ et $\vec{b} = 0010$.

Représentation vectoriel du calcul fait sur $(a(ab)^*b)^*$

- On pose $\vec{a}$ et $\vec{b}$ des tableaux de booléen qui représentent les positions de a et de b dans l'entrée (*bitmap*)

Pour $aaba$, on a $\vec{a} = 1101$ et $\vec{b} = 0010$.

- $\mathbf{aa} := \vec{a} \gg 1 \wedge \vec{a}$ (a *qui suit un* a) $\quad$ $\mathbf{bb} := \vec{b} \gg 1 \wedge \vec{b}$ (b *qui suit* b)
- $\mathbf{ab} := \vec{a} \gg 1 \wedge \vec{b}$ (b *qui suit un* a) $\quad$ $\mathbf{ba} := \vec{b} \gg 1 \wedge \vec{a}$ (a *qui suit* b)

Pour $aaba$ on a $\mathbf{aa} := 0100$ et $\mathbf{ab} := 0010$

Représentation vectoriel du calcul fait sur $(a(ab)^*b)^*$

- On pose $\vec{a}$ et $\vec{b}$ des tableaux de booléen qui représentent les positions de a et de b dans l'entrée (*bitmap*)

Pour $aaba$, on a $\vec{a} = 1101$ et $\vec{b} = 0010$.

- $\mathbf{aa} := \vec{a} \gg 1 \wedge \vec{a}$ (a *qui suit un* a) $\quad$ $\mathbf{bb} := \vec{b} \gg 1 \wedge \vec{b}$ (b *qui suit* b)
- $\mathbf{ab} := \vec{a} \gg 1 \wedge \vec{b}$ (b *qui suit un* a) $\quad$ $\mathbf{ba} := \vec{b} \gg 1 \wedge \vec{a}$ (a *qui suit* b)

Pour $aaba$ on a $\mathbf{aa} := 0100$ et $\mathbf{ab} := 0010$

$$((\mathbf{bb}|\mathbf{ab}) +_2 \mathbf{bb}) \wedge \mathbf{bb} == 0$$
$$((\mathbf{aa}|\mathbf{ba}) +_2 \mathbf{aa}) \wedge \mathbf{aa} == 0$$

Transformation en code du circuits

- $\mathbf{aa} := \vec{a}\text{»}1 \wedge \vec{a}$ (a *qui suit un* a) $\quad \mathbf{bb} := \vec{b}\text{»}1 \wedge \vec{b}$ (b *qui suit* b)
- $\mathbf{ab} := \vec{a}\text{»}1 \wedge \vec{b}$ (b *qui suit un* a) $\quad \mathbf{ba} := \vec{b}\text{»}1 \wedge \vec{a}$ (a *qui suit* b)

$$((\mathbf{bb}|\mathbf{ab}) +_2 \mathbf{bb}) \wedge \mathbf{bb} == 0$$
$$((\mathbf{aa}|\mathbf{ba}) +_2 \mathbf{aa}) \wedge \mathbf{aa} == 0$$

Transformation en code du circuits

- $\mathbf{aa} := \vec{a}\text{»}1 \wedge \vec{a}$ (a *qui suit un* a) $\quad \mathbf{bb} := \vec{b}\text{»}1 \wedge \vec{b}$ (b *qui suit* b)
- $\mathbf{ab} := \vec{a}\text{»}1 \wedge \vec{b}$ (b *qui suit un* a) $\quad \mathbf{ba} := \vec{b}\text{»}1 \wedge \vec{a}$ (a *qui suit* b)

$$((\mathbf{bb}|\mathbf{ab}) +_2 \mathbf{bb}) \wedge \mathbf{bb} == 0$$
$$((\mathbf{aa}|\mathbf{ba}) +_2 \mathbf{aa}) \wedge \mathbf{aa} == 0$$

- Chaque shift peut être évaluer en flot par bloc avec une continuation de 1 bit
- L'addition binaire peut être évaluer en flot par bloc en propageant la retenu de bloc en bloc

Transformation en code du circuits

- $\mathbf{aa} := \vec{a}\text{»}1 \wedge \vec{a}$ (a *qui suit un* a) $\quad \mathbf{bb} := \vec{b}\text{»}1 \wedge \vec{b}$ (b *qui suit* b)
- $\mathbf{ab} := \vec{a}\text{»}1 \wedge \vec{b}$ (b *qui suit un* a) $\quad \mathbf{ba} := \vec{b}\text{»}1 \wedge \vec{a}$ (a *qui suit* b)

$$((\mathbf{bb}|\mathbf{ab}) +_2 \mathbf{bb}) \wedge \mathbf{bb} == 0$$
$$((\mathbf{aa}|\mathbf{ba}) +_2 \mathbf{aa}) \wedge \mathbf{aa} == 0$$

- Chaque shift peut être évaluer en flot par bloc avec une continuation de 1 bit
- L'addition binaire peut être évaluer en flot par bloc en propageant la retenu de bloc en bloc

Remarque

C'est très effiace car les CPU optimise la propagation de retenu pour permettre de l'arithmétique sur des entiers non bornées (instruction `addcarry` et utilisation du `carry flag`).

Programme pour $(a(ab)^*b)^*$

```
char * L2_naif(char *s, size_t n){
    char state = 0;
    for (size_t i=0; i<n; i++){
        if (s[i] == 'a')
            state += 1;
        if (s[i] == 'b')
            state -= 1;
        if ((state < 0) || (state > 2))
            return (char *) (s + i);
    }
    return NULL;
}
```

Implémentation	naïve	vect
GByte/s	0.66	15.4
Cycles/Octet	5.4	0.2
Instr/Octet	12.0	1.0

```
char * L2_vect(char *s, size_t n){
    mask last_a = 0;
    mask last_b = 0;
    mask result_aa = 0;
    mask result_bb = 0;
    char carry1 = 0;
    char carry2 = 0;
    for (size_t i = 0 ; i < n ; i += vector_size) {
        vector chunk = load(s + i);
        mask offset_a = maskFromVector(vectorEq(chunk, mask_a));
        mask aa = (offset_a << 1 | last_a) & offset_a;
        ...
        mask ba = (offset_b << 1 | last_b) & offset_a;
        last_b = offset_b >> (vector_size - 1);
        last_a = offset_a >> (vector_size - 1);
        carry1 = addcarry(carry1, (bb | ab), bb, &result_bb);
        carry2 = addcarry(carry2, (aa | ba), aa, &result_aa);
        mask result = (result_aa & aa) | (result_bb & bb);
        if (result != 0)
            return (char*)(s + i + ctz(result));
    }
    return NULL;
}
```

Résultats

Theorem (Serre 2006)

Pour un langage régulier L, les propositions suivantes sont équivalentes

- L *est définissable dans la logique du premier ordre*
- *Son monoïde syntaxique est apériodique*
- L *admet* ***un circuit vectoriel***

Résultats

Theorem (Serre 2006)

Pour un langage régulier L, les propositions suivantes sont équivalentes

- L *est définissable dans la logique du premier ordre*
- *Son monoïde syntaxique est apériodique*
- L *admet* ***un circuit vectoriel***

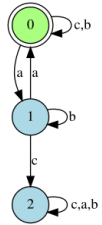
Remarque

- On peut toujours transformer la forme de circuit en programme concret
- Construire le circuit depuis le monoïde est PTIME (Paperman et al. 2023)
- Construire le monoid depuis un automate est exponentiel

Mais ...

- Certains langages importants ne sont pas capturés par cette construction

Exemple

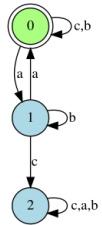


$(ab^*a + (b + c)^*)^*$

Mais ...

- Certains langages importants ne sont pas capturés par cette construction

Exemple



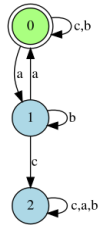
$$(ab^*a + (b + c)^*)^*$$

- Langage non apériodique ...

Mais ...

- Certains langages importants ne sont pas capturés par cette construction

Exemple



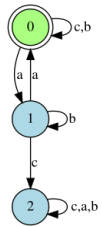
$$(ab^*a + (b + c)^*)^*$$

- Langage non apériodique ...
- mais admet un chouette programme vectoriel ...

Mais ...

- Certains langages importants ne sont pas capturés par cette construction

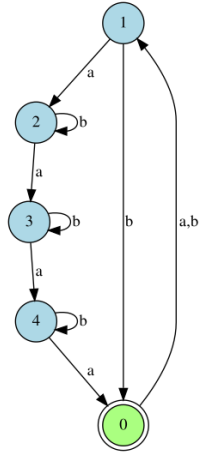
Exemple



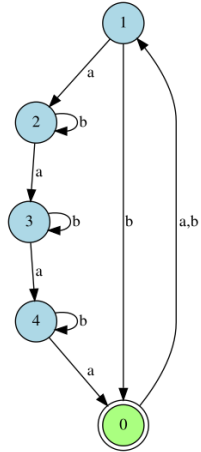
$(ab^*a + (b + c)^*)^*$

- Langage non apériodique ...
- mais admet un chouette programme vectoriel ...
- grâce a une instruction cryptographique (CLMUL, produit dans les polynômes sur $\mathrm{GF}(2)$).

Un cas ouvert important



Un cas ouvert important



- Son monoïde est le groupe de permutation sur 5 éléments (S_5)
- Aucune instruction ne semble pouvoir le capturer
- Langage qui a une complexité booléenne elevée (Théorème de Barrington)
- Difficulté lié à des problèmes de théorie des groupes finis (résolubilité, théorie de Galois)

Conclusion

- Écrire du code efficace c'est dure
- Compiler des boucles avec des conditionnelles encore plus
- La théorie algébrique des automates aident à y voir plus claire ...
- ... mais il reste encore beaucoup de travail pour comprendre tout ça.

Objectif du projet ANR Shannon meet cray[1]

[1] sxc.inria.fr