Grammaires de descriptions d'objets

Florent Hivert

Mél : `Florent.Hivert@lri.fr`
Adresse universelle : `http://www.lri.fr/~hivert`

Objectifs : algorithmes génériques

- **Identifier les composants de base :**

  $\Longrightarrow$ Singleton, union, produit cartésien, ensemble et multiensemble...

- Comprendre comment composer les briques de base

  $\Longrightarrow$ grammaire de description, classe combinatoire

- $\Longrightarrow$ Algorithmes génériques

Objectifs : algorithmes génériques

- **Identifier les composants de base :**

  $\Longrightarrow$ Singleton, union, produit cartésien, ensemble et multiensemble...

- Comprendre comment composer les briques de base

  $\Longrightarrow$ grammaire de description, classe combinatoire

- $\Longrightarrow$ Algorithmes génériques

Union disjointe

Definition

On écrit $C = A \sqcup B$ et on dit que C est l'**union disjointe** de A et B si $C = A \cup B$ et $A \cap B = \emptyset$.

Alors :

- $\mathrm{count}(C) = \mathrm{count}(A) + \mathrm{count}(B)$
- On peut prendre : $\mathrm{list}(C) = \mathrm{concat}(\mathrm{list}(A), \mathrm{list}(B))$

Union disjointe

Definition

On écrit $C = A \sqcup B$ et on dit que C est l'**union disjointe** de A et B si $C = A \cup B$ et $A \cap B = \emptyset$.

Alors :

- $\mathrm{count}(C) = \mathrm{count}(A) + \mathrm{count}(B)$
- On peut prendre : $\mathrm{list}(C) = \mathrm{concat}(\mathrm{list}(A), \mathrm{list}(B))$

Itération sur une union disjointe

On fixe l'ordre d'énumération tel que

$$\mathrm{list}(A \sqcup B) := \mathrm{concat}(\mathrm{list}(A), \mathrm{list}(B))$$

Itération en Python :

```
def iterunion(A, B):
    for a in A:
        yield a
    for b in B:
        yield b
```

first, next sur une union disjointe

$$\mathrm{list}(A \sqcup B) := \mathrm{concat}(\mathrm{list}(A), \mathrm{list}(B))$$

```
def first_union(A, B):
    return A.first()

def next_union(A, B, x):
    if x in A:
        try:
            return A.next(x)
        except StopIteration:
            return B.first()
    else:
        return B.next(x)
```

rank sur une union disjointe

$$\mathrm{list}(A \sqcup B) := \mathrm{concat}(\mathrm{list}(A), \mathrm{list}(B))$$

```
def rank_union(A, B, x):
    if x in A:
        return A.rank(x)
    else:
        return A.count() + B.rank(x)

def unrank_union(A, B, i):
    if i < A.count():
        return A.unrank(i)
    else:
        return B.unrank(i - A.count())
```

Le principe de l'idée récursive

**Quand on a un'bonne idée,
on l'appliqu'récursivement :
on obtient le plus souvent
une bien meilleure idée !**

- Unions disjointes récursives

Le principe de l'idée récursive

**Quand on a un'bonne idée,
on l'appliqu'récursivement :
on obtient le plus souvent
une bien meilleure idée !**

- Unions disjointes récursives

Les chaînes de n-bits ayant k-bits à 1

Une chaîne de bit non vide commence soit par un 0, soit par un 1 :

$\mathrm{BitString}(n, k) = 0 \cdot \mathrm{BitString}(n-1, k) \sqcup 1 \cdot \mathrm{BitString}(n-1, k-1)$

Idem triangle de pascal :

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

- $\mathrm{BitString}(n, k).\mathrm{count}() = \binom{n}{k}$

rank, unrank pour les chaînes de n-bits ayant k-bits à 1

```
def rank_BSnk(x):
    if not x:          # liste vide
        return 0
    if x[0] == 0:
        return rank_BSnk(x[1:])
    else:
        return binom(len(x)-1, sum(x)-1) + rank_BSnk(x[1:])

def unrank_BSnk(n, k, i):
    if n == 0:
        return []
    bn1k = binom(n-1, k)
    if i < bn1k:
        return [0]+unrank_BSnk(n-1, k, i)
    else:
        return [1]+unrank_BSnk(n-1, k-1, i-bn1k)
```

Le problème du calcul de la cardinalité

Problème

Le calcul récursif des coefficients binomiaux $\binom{n}{k}$ n'est pas efficace car on recalcule plusieurs fois la même chose.

Plus généralement, le calcul récursif des cardinalités sera très inefficace pour la même raison.

Parenthèse : mémoization et programmation dynamique

Retenir

- ***Mémoisation :*** *on mémorise tous les calculs pendant la récursion au momment où on les fait*
- ***Programmation Dynamique :*** *résoud les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.*

En général, la programmation dynamique est plus efficace mais plus longue à mettre en oeuvre : il faut avoir planifié l'utilisation de la mémoire.

Parenthèse : mémoization et programmation dynamique

Retenir

- ***Mémoisation :*** *on mémorise tous les calculs pendant la récursion au momment où on les fait*

- ***Programmation Dynamique :*** *résoud les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.*

En général, la programmation dynamique est plus efficace mais plus longue à mettre en oeuvre : il faut avoir planifié l'utilisation de la mémoire.

Parenthèse : mémoization et programmation dynamique

Retenir

- ***Mémoisation :*** *on mémorise tous les calculs pendant la récursion au momment où on les fait*
- ***Programmation Dynamique :*** *résoud les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.*

En général, la programmation dynamique est plus efficace mais plus longue à mettre en oeuvre : il faut avoir planifié l'utilisation de la mémoire.

Autre exemple : les permutations

Les permutés d'un ensemble $X := \{x_1, x_2, \ldots, x_n\}$:

$$\mathrm{Perm}\{1,2,3\} = 1 \cdot \mathrm{Perm}\{2,3\} \ \sqcup \ 2 \cdot \mathrm{Perm}\{1,3\} \ \sqcup \ 3 \cdot \mathrm{Perm}\{1,2\}$$

Plus généralement :

Retenir

Énumération lexicographique des permutations :

$$\mathrm{Perm}(X) = \bigsqcup_{i=1}^{n} x_i \cdot \mathrm{Perm}(X/\{x_i\})$$

- $\mathrm{Perm}(X).\mathrm{count}() = |X|!$

Généralisation : permuté d'un multiensemble

$$\mathrm{Perm}\{1,1,2,3\} = 1{\cdot}\mathrm{Perm}\{1,2\} \sqcup 2{\cdot}\mathrm{Perm}\{1,1,3\} \sqcup 3{\cdot}\mathrm{Perm}\{1,1,2\}$$

Notation : $\{1,1,2,3\} = 1^2 2^1 3^1$

$$\mathrm{Perm}(1^2 2^3 3^1) = 1{\cdot}\mathrm{Perm}(1^1 2^3 3^1) \sqcup 2{\cdot}\mathrm{Perm}(1^2 2^2 3^1) \sqcup 3{\cdot}\mathrm{Perm}(1^2 2^3)$$

Retenir

Énumération lexicographique des multi-permutations :

$$\mathrm{Perm}(X) = \bigsqcup_{i=1}^{n} x_i \cdot \mathrm{Perm}(X/\{x_i\})$$

Coefficient multinomiaux :

$$\binom{|I|}{i_1, i_2, \ldots, i_k} = \binom{|I|-1}{i_1-1, i_2, \ldots, i_k} + \binom{|I|-1}{i_1, i_2-1, \ldots, i_k} + \cdots + \binom{|I|-1}{i_1, i_2, \ldots, i_k-1}$$

$$\mathrm{Perm}(x_1^{i_1} \ldots x_k^{i_k}).\mathrm{count}() = \frac{(i_1+i_2+\cdots+i_k)!}{i_1! i_2! \ldots i_k!} = \binom{|I|}{i_1, i_2, \ldots, i_k}$$

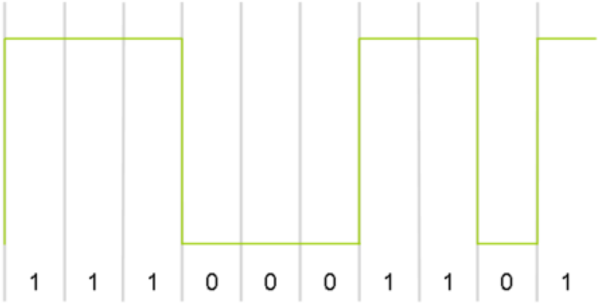
Coefficient multinomiaux :

$$\binom{|I|}{i_1, i_2, \dots, i_k} = \binom{|I|-1}{i_1 - 1, i_2, \dots, i_k} + \binom{|I|-1}{i_1, i_2 - 1, \dots, i_k} + \dots + \binom{|I|-1}{i_1, i_2, \dots, i_k - 1}$$

$$\mathrm{Perm}(x_1^{i_1} \dots x_k^{i_k}).\mathrm{count}() = \frac{(i_1 + i_2 + \dots + i_k)!}{i_1! i_2! \dots i_k!} = \binom{|I|}{i_1, i_2, \dots, i_k}$$

Autre application : transmission en codage NRZ

Non Return to Zero, Manière très élémentaire pour transmettre de l'information sur un ligne : 0 : -V, 1 : +V



Source : `https://fr.wikipedia.org/wiki/Non_Return_to_Zero`

Perte de synchronisation en codage NRZ

S'il on envoie une suite trop longue de bits identiques, on perd la synchronisation.

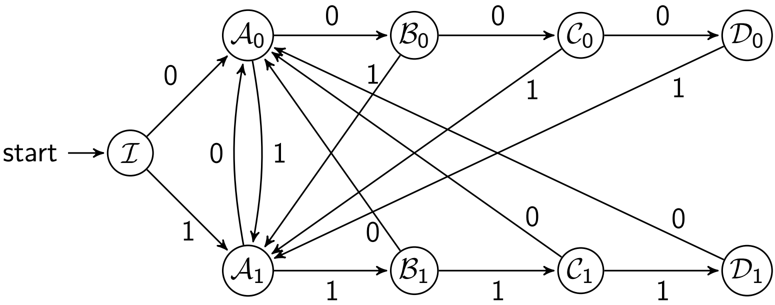
Definition

Une séquence de longueur n est dite non-repétitive d'ordre k (abréviation $NR(k, n)$) si elle ne contient pas se séquence de plus de k bits identiques consécutifs.

Notation : $NR(k, n, c, i)$ les suites qui commencent par au plus c i :

- $NR(k, n, 0, 0) = 1 \cdot NR(k, n-1, k-1, 1)$
- $NR(k, n, c, 0) = 0 \cdot NR(k, n-1, c-1, 0) \sqcup 1 \cdot NR(k, n-1, k-1, 1)$
- Idem en échangeant les rôles de 0 et 1.

C'est un automate fini !



Tous les états sont acceptants.

Union récursive et automates finis

Retenir

La méthode précédente fonctionne pour toute les automates finis déterministes. Si l'on note $\mathrm{Lang}_n(E)$ *l'ensemble des mots de longueur n acceptés à partir de l'état E, alors on a la définition récursive :*

- *Cas de base, pour un état E :*

$$\mathrm{Lang}_0(E) = \begin{cases} \{\varepsilon\} & \textit{si } E \textit{ est terminal} \\ \emptyset & \textit{sinon} \end{cases}$$

- *Étape de la récursion :*

$$\mathrm{Lang}_n(E) = \bigsqcup_{E \to_a E'} a \cdot \mathrm{Lang}_{n-1}(E')$$

Le produit cartésien

Definition

On appelle **produit cartesien** de A et B l'ensemble C noté $C := A \times B$ défini par

$$C := \{(a, b) \mid a \in A, b \in B)\}.$$

Alors :

- $\mathrm{count}(C) = \mathrm{count}(A) \cdot \mathrm{count}(B)$
- On peut prendre la liste dans l'ordre lexicographique :

$$\begin{aligned}\mathrm{list}(C) = [&(a_0, b_0), (a_0, b_1), (a_0, b_2), \dots, (a_0, b_l),\\ &(a_1, b_0), (a_1, b_1), (a_1, b_2), \dots, (a_1, b_l),\\ &(a_2, b_0), (a_2, b_1), (a_2, b_2), \dots, (a_2, b_l),\\ &\qquad\qquad\qquad \dots \qquad\qquad\qquad].\end{aligned}$$

Le produit cartésien

Definition

On appelle **produit cartesien** de A et B l'ensemble C noté $C := A \times B$ défini par

$$C := \{(a, b) \mid a \in A, b \in B)\}.$$

Alors :

- $\mathrm{count}(C) = \mathrm{count}(A) \cdot \mathrm{count}(B)$
- On peut prendre la liste dans l'ordre lexicographique :

$$\begin{aligned}\mathrm{list}(C) = [&(a_0, b_0), (a_0, b_1), (a_0, b_2), \dots, (a_0, b_l),\\ &(a_1, b_0), (a_1, b_1), (a_1, b_2), \dots, (a_1, b_l),\\ &(a_2, b_0), (a_2, b_1), (a_2, b_2), \dots, (a_2, b_l),\\ &\qquad\qquad\dots\qquad\qquad].\end{aligned}$$

Itération sur un produit cartésien

Ordre lexicographique :

Itération en Python :

```
def iter_cartprod(A, B):
    for a in A:
        for b in B:
            yield (a, b)
```

first, next sur un produit cartésien

Ordre lexicographique :

```
def first_cartprod(A, B):
    return (A.first(), B.first())

def next_cartprod(A, B, x):
    (a , b) = x      # pattern matching
    try:
       return (a, B.next(b))
    except StopIteration:
       return (A.next(a), B.first())
```

rank sur un produit cartésien

Ordre lexicographique :

```
def rank_cartprod(A, B, x):
    (a , b) = x       # pattern matching
    A.rank(a)*B.count() + B.rank(b)

def unrank_cartprod(A, B, i):
    c = B.count()
    return (A.unrank(i // c), B.unrank(i % c))
```

Notion de classe combinatoire

Définition (Classe combinatoire)

On appelle ***classe combinatoire*** *un ensemble $\mathcal{C}$ dont les éléments e ont une taille (nommée aussi degrée) noté $|e|$ et tels que l'ensemble $\mathcal{C}_n$ des éléments de taille n est fini :*

$$\mathrm{count}(\{e \in \mathcal{C} \mid |e| = n\}) < \infty$$

Exemple :

- Les mots sur un alphabet où la taille est la longueur
- Les permutations de $\{1, \dots n\}$ (taille $= n$)
- Les arbres binaires où la taille est le nombre de noeuds

L'union disjointe graduée

Si $\mathcal{C} = \mathcal{A} \sqcup \mathcal{B}$, les élements de $\mathcal{A}$ et $\mathcal{B}$ gardent leur taille dans l'union disjointe graduée :

$$\mathcal{C}_n := \mathcal{A}_n \sqcup \mathcal{B}_n$$

Alors :

- $\mathcal{C}.\,\mathrm{count}(n) = \mathcal{A}.\,\mathrm{count}(n) + \mathcal{B}.\,\mathrm{count}(n)$
- On peut prendre : $\mathcal{C}.\,\mathrm{list}(n) = \mathrm{concat}(\mathcal{A}.\,\mathrm{list}(b), \mathcal{B}.\,\mathrm{list}(n))$

$\Longrightarrow$ On peut réutiliser tout ce que l'on a vu sur les unions disjointes.

L'union disjointe graduée

Si $\mathcal{C} = \mathcal{A} \sqcup \mathcal{B}$, les élements de $\mathcal{A}$ et $\mathcal{B}$ gardent leur taille dans l'union disjointe graduée :

$$\mathcal{C}_n := \mathcal{A}_n \sqcup \mathcal{B}_n$$

Alors :

- $\mathcal{C}.\operatorname{count}(n) = \mathcal{A}.\operatorname{count}(n) + \mathcal{B}.\operatorname{count}(n)$
- On peut prendre : $\mathcal{C}.\operatorname{list}(n) = \operatorname{concat}(\mathcal{A}.\operatorname{list}(b), \mathcal{B}.\operatorname{list}(n))$

$\Longrightarrow$ On peut réutiliser tout ce que l'on a vu sur les unions disjointes.

Le produit cartesien gradué

Idée : les tailles (complexité, coût, nbr d'emplacements mémoires) s'ajoutent.

Definition

La taille de la paire $(a, b) \in \mathcal{A} \times \mathcal{B}$ est la somme des tailles :

$$|(a, b)|_{\mathcal{A}\times\mathcal{B}} := |a|_{\mathcal{A}} + |b|_{\mathcal{B}}$$

Le produit cartesien gradué

Retenir

Si $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ *alors*

$$\mathcal{C}_n \quad = \quad \bigsqcup_{i+j=n} \mathcal{A}_i \times \mathcal{B}_j$$

Calcul de la cardinalité :

$$|\mathcal{C}_n| = \sum_{i+j=n} |\mathcal{A}_i| \times |\mathcal{B}_j| = \sum_{i=0}^{n} |\mathcal{A}_i| \times |\mathcal{B}_{n-i}|$$

On peut alors prendre l'ordre union/lexicographique suivant :

$\mathcal{A}_0 \times \mathcal{B}_n$	$\mathcal{A}_1 \times \mathcal{B}_{n-1}$	$\mathcal{A}_2 \times \mathcal{B}_{n-2}$	$\dots$	$\mathcal{A}_n \times \mathcal{B}_0$

Le produit cartesien gradué

Retenir

Si $\mathcal{C} = \mathcal{A} \times \mathcal{B}$ *alors*

$$\mathcal{C}_n = \bigsqcup_{i+j=n} \mathcal{A}_i \times \mathcal{B}_j$$

Calcul de la cardinalité :

$$|\mathcal{C}_n| = \sum_{i+j=n} |\mathcal{A}_i| \times |\mathcal{B}_j| = \sum_{i=0}^{n} |\mathcal{A}_i| \times |\mathcal{B}_{n-i}|$$

On peut alors prendre l'ordre union/lexicographique suivant :

$\mathcal{A}_0 \times \mathcal{B}_n$	$\mathcal{A}_1 \times \mathcal{B}_{n-1}$	$\mathcal{A}_2 \times \mathcal{B}_{n-2}$	$\dots$	$\mathcal{A}_n \times \mathcal{B}_0$

Application les arbres binaires

Spécification récursive :

$$\mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} \quad = \quad \mathrm{Leaf} \quad \sqcup \quad \mathrm{Node}(\mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} \times \mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree})$$

Deux manières de compter les tailles :

1. Nombre de feuille :

$$\mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} \quad = \quad \mathrm{Leaf}_1 \quad \sqcup \quad \mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} \times \mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree}$$

2. Nombre de Noeuds :

$$\mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} \quad = \quad \mathrm{Leaf}_0 \quad \sqcup \quad \mathrm{Node}_1 \times \mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} \times \mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree}$$

Application les arbres binaires

Spécification récursive :

$$\mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} \quad = \quad \mathrm{Leaf} \quad \sqcup \quad \mathrm{Node}(\mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} \times \mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree})$$

Deux manières de compter les tailles :

1. Nombre de feuille :

$$\mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} \quad = \quad \mathrm{Leaf}_1 \quad \sqcup \quad \mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} \times \mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree}$$

2. Nombre de Noeuds :

$$\mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} \quad = \quad \mathrm{Leaf}_0 \quad \sqcup \quad \mathrm{Node}_1 \times \mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} \times \mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree}$$

Liste de tous les arbres à n Nœuds

Algorithme

- ***Entrée :*** *un entier positif ou nul* n
- ***Sortie :*** *une liste d'arbres*

```
if n == 0:
    yield arbreVide()
for i in range(n):
    for g in BinTree(i):
        for d in BinTree(n-1-i):
            yield Noeud(g,d)
```

Nombre de Catalan

Proposition

Le nombre d'arbres binaires à n nœuds est appelé n-ième nombre de Catalan noté C_n. Les nombre de Catalan vérifient la récurrence :

$$C_0 = 1 \qquad C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}\,.$$

On peut trouver une formule close :

$$C_n = \frac{(2n)!}{n!(n+1)!}\,.$$

Voici les premières valeurs :

$$C_0 = 1,\ C_1 = 1,\ C_2 = 2,\ C_3 = 5,\ C_4 = 14,\ C_5 = 42,\ c_6 = 132\,.$$

Specification d'une classe combinatoire

But : on veut décrire une **classe combinatoire** de manière à pouvoir appliquer automatiquement les algorithmes de comptage, itération, génération,. . ..

Pour ceci, on va utiliser une **grammaire** qui code comment appliquer récursivement les constructions précédentes.

Specification d'une classe combinatoire

Retenir (Grammaire de description d'objets)

Constructeurs terminaux :

- $\mathcal{E}(o)$ *: la classe qui contient un seul objet o de taille* 0
- $\mathcal{Z}(o)$ *: la classe qui contient un seul objet o de taille* 1

Constructeurs binaires :

- $\mathcal{C} = \mathrm{Union}(\mathcal{A}, \mathcal{B})$ *: union disjointe graduée :*

$$|a|_{\mathcal{C}} = |a|_{\mathcal{A}} \text{ si } a \in \mathcal{A} \qquad |a|_{\mathcal{C}} = |b|_{\mathcal{B}} \text{ si } b \in \mathcal{B}$$

- $\mathcal{C} = \mathrm{Prod}(\mathcal{A}, \mathcal{B})$ *: produit cartésien graduée :*

$$|(a, b)|_{\mathcal{C}} = |a|_{\mathcal{A}} + |b|_{\mathcal{B}} \text{ si } a \in \mathcal{A} \text{ et } b \in \mathcal{B}$$

Exemples :

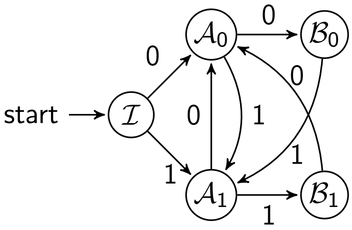
- Les arbres binaires, la taille est le nombres de **Noeuds**

$$\mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} = \mathrm{Union}(\mathcal{E}(\perp), \mathrm{Prod}(\mathcal{Z}(\circ), \mathrm{Prod}(\mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree}, \mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree}))$$

- Les arbres binaires, la taille est le nombres de **Feuilles**

$$\mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree} = \mathrm{Union}(\mathcal{Z}(\perp), \mathrm{Prod}(\mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree}, \mathcal{B}\mathrm{in}\mathcal{T}\mathrm{ree}))$$

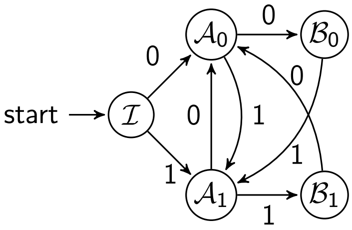
Exemples : Codage d'un automate fini



Les mots binaires qui n'ont pas plus de deux lettres identiques consécutives :

$$
\begin{aligned}
\mathcal{I} &= \mathcal{E}(\varepsilon) \sqcup (\mathcal{Z}("0") \times \mathcal{A}_0) \sqcup (\mathcal{Z}("1") \times \mathcal{A}_1) \\
\mathcal{A}_0 &= \mathcal{E}(\varepsilon) \sqcup (\mathcal{Z}("0") \times \mathcal{B}_0) \sqcup (\mathcal{Z}("1") \times \mathcal{A}_1) \\
\mathcal{B}_0 &= \mathcal{E}(\varepsilon) \sqcup (\mathcal{Z}("1") \times \mathcal{A}_1) \\
\mathcal{A}_1 &= \mathcal{E}(\varepsilon) \sqcup (\mathcal{Z}("0") \times \mathcal{A}_0) \sqcup (\mathcal{Z}("1") \times \mathcal{B}_1) \\
\mathcal{B}_1 &= \mathcal{E}(\varepsilon) \sqcup (\mathcal{Z}("0") \times \mathcal{A}_0)
\end{aligned}
$$

Exemples : Codage d'un automate fini



Dans cet exemple : plus simple, on inclut la lettre dans l'état :

$$\mathcal{I} = \mathcal{E}(\varepsilon) \sqcup \mathcal{A}_0 \sqcup \mathcal{A}_1$$

$$\mathcal{A}_0 = \mathcal{Z}("0") \times (\mathcal{E}(\varepsilon) \sqcup \mathcal{B}_0 \sqcup \mathcal{A}_1)$$

$$\mathcal{B}_0 = \mathcal{Z}("0") \times (\mathcal{E}(\varepsilon) \sqcup \mathcal{A}_1)$$

$$\mathcal{A}_1 = \mathcal{Z}("1") \times (\mathcal{E}(\varepsilon) \sqcup \mathcal{B}_1 \sqcup \mathcal{A}_0)$$

$$\mathcal{B}_1 = \mathcal{Z}("1") \times (\mathcal{E}(\varepsilon) \sqcup \mathcal{A}_0)$$