

# Projet Info LDD2 – TD 9

Renaud Vilmar -- vilmar@lsv.fr

**Objectifs du TD :** Synthèse de circuit via une table de vérité; tables de Karnaugh; codes de Gray.

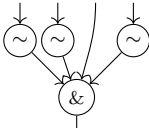
On a vu la dernière fois comment synthétiser un circuit booléen à partir d'une formule propositionnelle. On va maintenant voir des méthodes pour faire de même lorsque c'est la table de vérité de l'opérateur qui est fournie. Prenons un exemple qui va nous suivre pendant ce TD :

$x_0$	$x_1$	$x_2$	$x_3$	Op
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

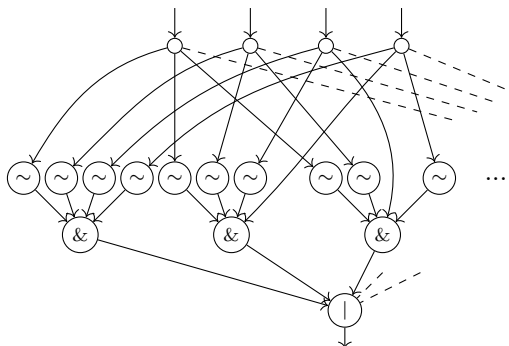
On veut synthétiser un circuit correspondant, donc avec 4 entrées (chacune correspondant à une variable), et une sortie qui va contenir le bon résultat pour toutes les valuations possibles des variables.

On n'a pas besoin de donner toute la table comme entrée du problème, on peut se contenter de donner la dernière colonne, soit une chaîne de bits '111000100011111' soit comme une liste de booléens. Pour être valable, une telle chaîne doit avoir une longueur d'une puissance de 2 (et l'exposant nous donne le nombre de variables).

Pour créer le circuit, on peut pour commencer considérer une ligne de la table dont le résultat est 1, par exemple 0010  $\mapsto$  1 (la 3ème ligne). On peut facilement créer un morceau de circuit qui va donner 1 uniquement pour la

valuation 0010 des 4 variables :  . On peut faire ça pour chaque

ligne dont le résultat est 1, il suffira ensuite de regrouper toutes ces sorties en une porte OU :



▮ **Exercice 1 :** Implémenter une méthode de classe pour les circuits booléens qui réalise cette construction. Astuce : on peut obtenir l'écriture binaire d'un `int x` via `bin(x)`.

Donner une estimation de la taille du circuit (en nombre de portes), et de sa profondeur (on pourra marquer ça dans la doc). On peut prendre en compte le nombre de variables  $n$  et le poids de Hamming  $p$  de la chaîne de bits (i.e. le nombre de 1 dans cette chaîne). ▮

On peut difficilement faire mieux en terme de profondeur de circuit. Le nombre de portes peut par contre être amélioré, pour une profondeur équivalente. Une technique pour ce faire utilise ce qu'on appelle les *tables de Karnaugh*.

Pour les utiliser il faut d'abord comprendre le code de Gray. L'idée derrière ce code est de pouvoir compter, dans un système binaire, d'une façon qui ne change qu'un bit à la fois, ce qui n'est pas le cas habituellement (en binaire passer de 3 à 4 donne 011 → 100, ce qui change trois bits d'un coup dans la représentation). Voici à quoi ressemblent les premiers nombre du code de Gray :

(000, 001, 011, 010, 110, 111, 101, 100)

On remarque bien qu'entre deux nombre consécutifs, un seul bit est modifié (ça reste même vrai entre le premier et le dernier nombre, ce qui sera important par la suite).

On peut facilement énumérer les codes de Gray :

- les deux éléments ordonnés du code de Gray à 1 bit sont (0, 1)
- si  $(a_1, \dots, a_{2^n})$  sont les  $2^n$  éléments ordonnés du code à  $n$  bits,  $(0 \cdot a_1, \dots, 0 \cdot a_{2^n}, 1 \cdot a_{2^n}, \dots, 1 \cdot a_1)$  sont les éléments ordonnés du code à  $n + 1$  bits (avec  $\cdot$  qui dénote la concaténation).

On voit bien que l'exemple ci-dessus est en fait le code de Gray sur 3 bits.

▮ **Exercice 2 :** Créer une fonction qui étant donné  $n$  renvoie le code de Gray à  $n$  bits (sous la forme d'une liste de chaînes de caractères par exemple). ▮

L'étape d'après consiste à partitionner nos variables en 2 listes, qui vont servir pour indexer un tableau selon l'ordre donné par le code de Gray. On verra

dans l'étape d'après pourquoi ce code est important. Avec l'exemple précédent, si les deux ensembles de variables sont  $[x_0, x_1]$  et  $[x_2, x_3]$ , la table de Karnaugh associée est la suivante :

		$x_2, x_3$			
		00	01	11	10
$x_0, x_1$	00	1	1	0	1
	01	0	0	0	1
	11	1	1	1	1
	10	0	0	1	1

▮ **Exercice 3 :** Créer la fonction `K_map` qui prend une chaîne de bits en entrée et retourne la table de Karnaugh associée (le choix du partitionnement des variables est laissé à votre appréciation, à la contrainte près suivante : les deux ensembles doivent être équilibrés (i.e. de même taille  $\pm 1$  si le nombre de variables est impair). ▮

Comment maintenant exploiter cette table ? On va devoir considérer des blocs de 1 dans celle-ci, par exemple les blocs rouge et bleu dans :

		$x_2, x_3$			
		00	01	11	10
$x_0, x_1$	00	1	1	0	1
	01	0	0	0	1
	11	1	1	1	1
	10	0	0	1	1

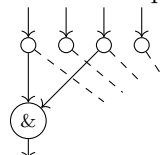
Certaines contraintes doivent être respectées pour qu'un bloc soit valide :

- un bloc est rectangulaire (pas de diagonale autorisée par exemple)
- il ne recouvre que des 1
- ses côtés sont de longueur une puissance de 2

et certaines choses sont autorisées :

- un bloc peut "boucler" entre la dernière et la première colonne/ligne (comme le bloc rouge ci-dessus)
- deux blocs peuvent se chevaucher

À quoi correspondent ces blocs ? Prenons l'exemple du bloc bleu : les cellules de la table correspondent à la valuation  $x_0 \leftarrow 1 \wedge x_2 \leftarrow 1$ , sans avoir besoin de spécifier des valeurs pour  $x_1$  et  $x_3$ . Ça va ainsi correspondre à ce morceau

de circuit :  . Il faudra in fine récupérer toutes les sorties de ces

morceaux de circuit dans une porte OU (comme c'est fait dans la première méthode).

L'intérêt du code de Gray se trouve ici : quand on passe d'une cellule à une autre adjacente, on est assuré de n'avoir qu'une variable qui change de valeur. Ainsi, si un bloc recouvre 2 colonnes, seulement 1 variable de colonne change

de valeur, ce qui correspond au fait que cette variable peut être ignorée. Plus généralement, si le bloc recouvre  $2^n$  colonnes, seules  $n$  variables changent de valeur, et sont donc ignorées, d'où l'importance de la puissance de 2 dans la longueur des côtés du bloc.

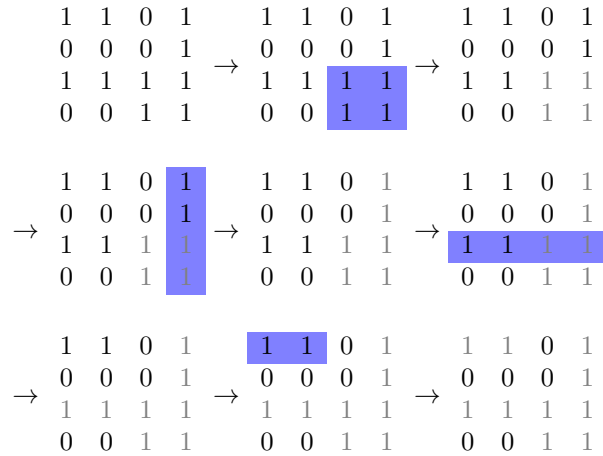
Le but avec la table de Karnaugh est donc de :

- recouvrir tous les 1 avec des blocs
- utiliser le moins de blocs possibles

On va donc essayer de former des blocs aussi gros que possible. On remarque par ailleurs que si on ne fait que des blocs de taille 1, on se retrouve à construire le même circuit que dans l'exo 1.

Il existe plusieurs façons de trouver ces blocs. Pour raison de simplicité, on va implémenter un algorithme glouton (. La table va être modifiée pendant l'exécution, plus spécifiquement on va distinguer les 1 qui sont déjà dans un bloc (notés 1) des autres. Notons  $n$  le nombre de variables (la table contient donc  $2^n$  cellules). L'algorithme s'arrête dès que chaque 1 est recouvert par au moins un bloc (i.e. tous les 1 sont transformés en 1). Sinon, pour chaque  $i$  allant de  $n$  à 0, on regarde tous les blocs de surface  $2^i$  valides, et on choisit celui (un) qui maximise le nombre de 1 recouverts s'il existe. Ce bloc est sélectionné, tous les 1 recouverts par lui deviennent de 1. On continue avec ce même  $i$  tant qu'on trouve des blocs (on ne prend donc pas en compte les blocs qui ne recouvrent que des 1).

Sur notre exemple :



L'algo s'arrête car tous les 1 ont été transformés en 1 : ils sont tous recouverts par un bloc. On finit donc avec 4 blocs, le premier correspondant à  $x_0 \& x_2$ , le deuxième à  $x_2 \& \sim x_3$ , le troisième à  $x_0 \& x_1$  et le dernier à  $\sim x_0 \& \sim x_1 \& \sim x_2$ . On peut ainsi obtenir une formule propositionnelle :

$$(x_0 \& x_2) | (x_2 \& \sim x_3) | (x_0 \& x_1) | (\sim x_0 \& \sim x_1 \& \sim x_2)$$

▮ **Exercice 4 :** Implémenter cet algorithme dans une fonction, qui donne en sortie une formule propositionnelle. ▮

▮ **Exercice 5 :** Implémenter une méthode de classe pour les circuits booléens qui implémente cette approche (comme pour l'exo 1, elle doit simplement prendre en entrée une chaîne de bits qui correspond à la table de vérité de la fonction qu'on veut construire). ▮

▮ **Exercice 6 (Bonus) :** L'algo glouton ci-dessus peut produire des blocs qu'on peut aisément ignorer, par exemple, dans :

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1

le bloc rouge est créé par l'algorithme glouton, mais est en fait inutile car entièrement recouvert par d'autres blocs.

Modifier l'algo de l'exo 4 pour qu'en post-processing il cherche et supprime les blocs inutiles (en commençant par les plus petits). ▮