TP3

La page web du cours avec quelques fichiers supplémentaires se trouve ici :

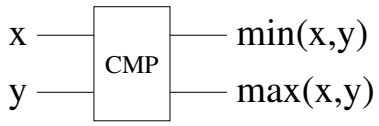
`http://www.lsv.ens-cachan.fr/~schwoon/enseignement/systemes/ws1516/`.

1 Vérification des circuits logiques

Un problème important dans la conception du matériel est d'assurer qu'un circuit satisfait sa fonction dans toutes les circonstances. Nous allons étudier une approche à ce problème en utilisant les *réseaux de tri* comme exemple.

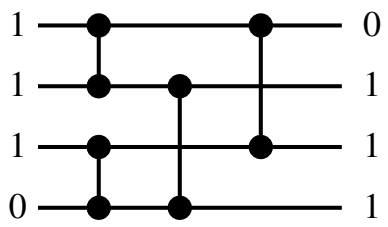
Un réseau de tri possède n nombres en entrée où n est fixe pour un réseau donné. Son objectif est de sortir ces nombres en ordre croissant. On sait qu'un réseau de tri fonctionne correctement si et seulement s'il le fait pour toute séquence de 0 et 1, du coup on va supposer que toutes les entrées sont soit 0 soit 1.

L'élément de base d'un réseau de tri est un *comparateur*. Un comparateur possède deux entrées et deux sorties. La sortie « haute » sera la valeur la plus petite parmi les entrées et la sortie « basse » la plus grande.



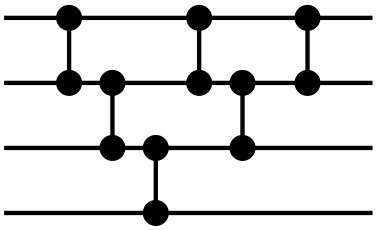
(a) Donner un circuit logique (avec des portes NON, ET, OU, NAND, ...) qui réalise les fonctions d'un comparateur.

Un réseau de tri consiste de n fils connectés par des comparateurs. On indique les comparateurs par des lignes verticales. Voici un réseau de tri pour $n = 4$ et quelques entrées et les sorties correspondantes :



(b) Le réseau ci-dessus donne le résultat correct pour l'exemple, mais pas pour toutes les entrées possibles. Trouver un jeu d'entrées pour lequel le résultat est incorrect.

Une possibilité pour concevoir un réseau correct est de s'inspirer des algorithmes classiques. Par exemple, le réseau ci-dessous simule le comportement de « Bubble Sort » : on identifie d'abord l'élément le plus grand, ensuite le deuxième plus grand etc.



La page web du cours contient un simulateur simple des réseaux de tri (`snsim.pl`). Avant de l'utiliser, il faut rendre ce programme *exécutable* : dans la ligne de commande, utilisez `cd` pour aller dans le répertoire qui contient les fichiers téléchargés, puis `chmod +x *pl`.

Le simulateur prend en entrée un fichier qui spécifie les paires de fils qui doivent être comparés. Par exemple, le réseau ci-dessus est représenté comme suit (`bubble-4.sn`) :

```
1 2
2 3
3 4
1 2
2 3
1 2
```

On utilise

```
./snsim.pl bubble-4.sn 0101
```

dans la ligne de commande pour simuler ce réseau où `0101` représente les nombres en entrée.

Le réseau inspriré par Bubble Sort n'est pourtant pas efficace, de par sa taille et sa profondeur. Pour les réseaux de tri, la taille dépend simplement du nombre de comparateurs, pour la profondeur on peut considérer les comparateurs et les tronçons des fils entre les entrées/sorties et les comparateurs :

- Un tronçon est de profondeur 0 s'il es connecté a une entrée.
- Si les deux tronçons qui entrent dans un comparateur possèdent les profondeurs d_1 et d_2, alors le comparateur ainsi que les tronçons sortants possèdent la profondeur $1 + max(d_1, d_2)$.

La profondeur d'un réseau est alors la plus grande profondeur de l'un de ses comparateurs ou tronçons. Dans l'exemple « Bubble Sort »la taille est de 6 et la profondeur de 5. (Le simulateur calcule ces données pour un réseau donné.)

(c) Créer un réseau de tri correct et plus efficace (par rapport aux deux mésures) pour $n = 4$. Utilisez le simulateur pour le tester.

(d) Trouver des réseaux corrects pour $n = 5$ et $n = 6$, en essayant de minimiser taille et profondeur.

Comment vérifier si un réseau donné est correct pour toute séquence d'entrées possibles ? Une possibilité est de le simuler pour toutes les 2^n séquences ce qui est inefficace. Une autre possibilité que nous allons explorer est de formuler le problème en logique propositionnelle,

c'est à dire on transforme un réseau N vers une formule propositionnelle ϕ_N telle que ϕ_N est satisfaisable si et seulement si N est incorrect. Des outils appelés *SAT solvers* existe pour tester la satisfaisabilité d'une telle formule. (On remarque que dans le pire des cas, les SAT solvers ne sont pas plus efficace que d'énumerer toutes les séquences d'entrées possibles. Pourtant, en pratique, ils sont souvent beaucoup plus efficaces que ça.)

Nous allons générer des formules avec des variables de form x_i^d qui représentent la valeur du tronçon du fil numéro i à profondeur d. Par exemple, les variables x_i^0, pour $i = 1, \ldots, n$, sont les valeurs en entrée.

(e) Soit C un comparateur entre les fils i et j à profondeur respectivement d_1 et d_2. En utilisant (a), donner une formule ϕ_C qui relie les variables représentant les entrées et sorties de C.

Soit $\mathcal{C}$ l'ensemble des comparateurs et d_i la profondeur finale de l'entrée numéro i, alors ϕ_N prend la forme suivante :

$$\phi_N := \left(\bigwedge_{C \in \mathcal{C}} \phi_C \right) \wedge \neg \phi^{sort}(x_1^{d_1}, \ldots, x_n^{d_n}),$$

où ϕ^{sort} exprime que les sorties doivent être bien triées.

(f) Spécifier ϕ^{sort}.

La page web du cours contient un programme (encore incomplet !) qui génère ϕ_N étant donné un réseau N (avec le syntaxe donné ci-dessus) génère ϕ_N. La formule est par la suite fourni à l'outil Z3 (un SAT solver) qui teste la satisfaisabilité de ϕ_N. Notre programme (`sn_verif.pl`) va analyser le résultat de Z3 pour en extraire une séquence pas correctement triée, le cas échéant.

(g) Compléter `snverif.pl` en spécifiant ϕ_C et ϕ^{sort} dans les bons endroits (lire attentivement les commentaires). Essayer le programme pour vérifier vos réseaux. Dans l'exemple, le syntaxe pour invoquer le programme est `./snverif.pl bubble.sn 4`, où 4 est la valeur de n.

2 Synthèse de circuits

On peut aller encore plus loin et utiliser des techniques similaires pour automatiquement créer un circuit avec des propriétes souhaitées. Au lieu de demander si, pour un circuit donné, il existe des valeurs en entrée satisfaisant une propriété (p.ex., les valeurs en sortie ne sont pas triées), on demande s'il existe un circuit (avec certaines bornes) qui possède les bonnes propriétés. On discutera cette idée, inspiré par le papier *Synthesis of Parallel Sorting Networks using SAT Solvers* par Morgenstern et Schneider, 2011.

L'idée de base est qu'un réseau de tri de profondeur d peut être exprimé par une séquence des *permutations* $P_1, \ldots, P_d$, où P_i spécifie le comportement des comparateurs de profondeur i. Plus précisement, on suppose que $P_{k,i} = j$ and $P_{k,j} = i$ si un comparateur de profondeur k relie les fils i et j, et que $P_{k,i} = i$ si le fils numéro i n'est relié à aucun comparateur de profondeur k. Pour l'exemple de Bubble Sort, les permutations résultants sont comme suit :

$P_1 = (2\ 1\ 3\ 4)$, $P_2 = (1\ 3\ 2\ 4)$, $P_3 = (2\ 1\ 4\ 3)$, $P_4 = (1\ 3\ 2\ 4)$, $P_5 = (2\ 1\ 3\ 4)$

Pour n et d donné, nous allons construire une formule $\phi^{synth}_{n,d}$ qui demande l'existence d'un réseau de tri pour n nombres et de profondeur d. Ses variables seront les permutations P_k (vecteurs d'entiers) et les variables booléennes x^k_i ($i = 1, \ldots, n$ and $k = 0, \ldots, d$). La formule possède trois composants qui assurent :

(i) que P_k est une permutation pour tout k (ϕ^{perm}) ;

(ii) que les valeurs x^k_i sont consistentes avec P_k, pour tous les tronçons (ϕ^{cons}) ;

(iii) et que le résultat final est bien trié, pour toutes les entrées (ϕ^{sort}).

$$\phi^{synth}_{n,d} := \left(\bigwedge_{k=1}^{d} \phi^{perm}(P_k) \right) \wedge \left(\bigwedge_{x^0_1,\ldots,x^0_n \in \{0,1\}} \left(\bigwedge_{k=1}^{d} \phi^{cons}(P_k, x_{k-1}, x_k) \right) \wedge \phi^{sort}(x_d) \right)$$

Pour (i), ϕ^{perm} exploite le fait que les permutations sont symmétriques :

$$\phi^{perm}(P_k) := \bigwedge_{i=1}^{n} P_{k,P_{k,i}} = i$$

Pour (ii), ϕ^{cons} est une adaptation de notre formule ϕ_C de l'exercice 1(e) :

$$\phi^{cons}(P_k, x_{k-1}, x_k) := \bigwedge_{i=1}^{n} x^k_i = \begin{cases} x^{k-1}_i \vee x^{k-1}_{P_{k,i}} & \text{si } i \leq P_{k,i} \\ x^{k-1}_i \wedge x^{k-1}_{P_{k,i}} & \text{sinon} \end{cases}$$

ϕ^{sort} est comme dans l'exercice 1(f).

(a) Le programme `snfind.pl` génère $\phi^{synth}_{n,d}$. Utilisez-le pour trouver des réseaux de profondeur optimale pour $n = 5, 6, 7, \ldots$

Le même principe peut être appliqué à d'autres fonctionnalités. Supposons par exemple que nous souhaitons construire des circuits à partir d'autres éléments de base comme les portes NON-ET. Supposons que le principe des permutations reste inchangé, mais la formule ϕ^{cons} demandera que le fils avec l'indice le plus petit obtienne le résultat (l'autre fils deviendra inutile).

(b) Adapter ϕ^{cons} pour traiter des portes NON-ET.

(c) En supposant que $n = 2$ et que la fonctionnalité souhaité est l'ou exclusive, trouver un replacement pour ϕ^{sort}.

(d) Cette idée, est-ce qu'elle va être utile de trouver le circuit de profondeur minimale ? Sinon, pourquoi, et quels changements seront nécessaires ?

(e) (exercice avancée) Adapter `snfind.pl` pour trouver le circuit le plus petit pour XOR.