Correction du TD 17

Julien Reichert

Exemple de sujet de CCINP

Question 1.1

Il y a quatre natures possibles (en comptant les changeants suivant la valeur de vérité de leur première affirmation), et chaque question ayant deux réponses possibles, il faut au moins deux questions pour trancher. En pratique, deux questions dont on connaît la réponse suffisent aussi, car les deux valeurs de vérité consécutives lèveront toute ambigüité.

Question 1.2

$$A_V = \bigwedge_{i=1}^{n} A_i, A_M = \bigwedge_{i=1}^{n} \neg A_i, A_{CV} = \bigwedge_{\substack{i=1,\\ i \text{ impair}}}^{n} A_i \wedge \bigwedge_{\substack{i=2,\\ i \text{ pair}}}^{n} \neg A_i \text{ et } A_{CM} = \bigwedge_{\substack{i=1,\\ i \text{ impair}}}^{n} \neg A_i \wedge \bigwedge_{\substack{i=2,\\ i \text{ pair}}}^{n} A_i$$

Question 1.3

$A_1 = R \wedge \neg B$,
$A_2 = R \wedge \neg V \vee \neg R \wedge V$,
$A_3 = (R \wedge V) \Rightarrow B$

Question 1.4

On a $n = 3$ dans les formules de la question 1.2 et on remplace directement à partir de la réponse à la question 1.3. Les quatre possibilités donnent :

- $A_V = (R \wedge \neg B) \wedge (R \wedge \neg V \vee \neg R \wedge V) \wedge ((R \wedge V) \Rightarrow B)$
- $A_M = \neg(R \wedge \neg B) \wedge \neg(R \wedge \neg V \vee \neg R \wedge V) \wedge \neg((R \wedge V) \Rightarrow B)$
- $A_{CV} = (R \wedge \neg B) \wedge \neg(R \wedge \neg V \vee \neg R \wedge V) \wedge ((R \wedge V) \Rightarrow B)$
- $A_{CM} = \neg(R \wedge \neg B) \wedge (R \wedge \neg V \vee \neg R \wedge V) \wedge \neg((R \wedge V) \Rightarrow B)$

Question 1.5

R	V	B	A_1	A_2	A_3	A_V	A_M	A_{CV}	A_{CM}
F	F	F	F	F	V	F	F	F	F
F	F	V	F	F	V	F	F	F	F
F	V	F	F	V	V	F	F	F	F
F	V	V	F	V	V	F	F	F	F
V	F	F	V	V	V	V	F	F	F
V	F	V	F	V	V	F	F	F	F
V	V	F	V	F	F	F	F	F	F
V	V	V	F	F	V	F	F	F	F

Donc l'orateur est un véridique qui aime le rouge uniquement.

Question 1.6

Attention à bien comprendre le sens de « que si », l'implication va de gauche à droite !

$G_1 = C \Rightarrow L$,
$H_1 = C \wedge \neg T \vee \neg C \wedge T$,
$I_1 = L$,
$I_2 = \neg T$

Question 1.7

$\neg G_1 \wedge H_1 \wedge (I_1 \wedge \neg I_2 \vee \neg I_1 \wedge I_2)$

Question 1.8

On remplace dans le résultat précédent :

$C \wedge \neg L \wedge (C \wedge \neg T \vee \neg C \wedge T) \wedge (L \wedge \neg\neg T \vee \neg L \wedge \neg T)$

Parce qu'on a C et $\neg L$ connus, on peut déduire ce qu'il en est du « ou exclusif » de H_1 (et utiliser l'absorption pour retirer la deuxième occurrence de C) :

$C \wedge \neg L \wedge \neg T \wedge (L \wedge \neg\neg T \vee \neg L \wedge \neg T)$

Dans la parenthèse de droite, la double négation s'applique, et le premier cas est de toute manière contradictoire, c'est donc le deuxième cas qui est vrai et qui disparaît aussi par la règle d'absorption :

$C \wedge \neg L \wedge \neg T$

Ainsi, seul le cercle est visible, et la première affirmation de l'orateur I étant fausse, I est un changeant commençant par mentir.

n-SAT

Exercice 2

Notons que la clause C est vraie si, et seulement si, au moins un littéral l_i est vrai.

Supposons la clause C vraie et notons j (arbitraire) tel que l_j soit vrai. Notons aussi c un indice (arbitraire) tel que l_j apparaisse dans C_c. Supposons que γ_i soit vrai si, et seulement si, $i < c$. Alors les C_i sont satisfaites pour $i < c$ grâce aux γ_i correspondants, pour $i = c$ grâce à l_j et pour $i > c$ grâce aux $\neg\gamma_{i-1}$, donc la conjonction des C_i est satisfaite pour au moins un choix des valeurs de vérité des γ_i.

Supposons la clause C fausse. Alors la conjonction des C_i ne peut pas être vraie quel que soit le choix des valeurs de vérité des γ_i car tous les littéraux qui apparaissent dans la conjonction sont faux sauf exactement $k-1$ (pour chaque i, on a soit γ_i soit $\neg\gamma_i$ qui est vrai) et il faut satisfaire k clauses.

Exercice 3

Première remarque : pour $n \leq 2$, la fraction n'a pas de sens (division par zéro ou dénominateur strictement négatif posant des problèmes) d'une part, et le découpage ne permet pas d'écrire de nouveaux littéraux dans la clause une fois deux occurrences d'un littéral utilisant les nouvelles variables propositionnelles écrites. On suppose donc que $n \geq 3$.

Notons que si $m \leq n$, on a déjà $\lceil \frac{m-2}{n-2} \rceil = 1$ (en ayant aussi considéré à part le cas où $m \leq 2$) et cela tombe bien car on ne veut qu'une clause. Par la suite, si on pose k le nombre de clauses, on perd en fait $2(k-1)$ des kn emplacements disponibles pour des littéraux utilisant les nouvelles variables propositionnelles, ou de manière plus pratique il reste $(n-2)k+2$ places pour écrire les m littéraux de la clause d'origine. En prenant le plus petit k convenable, et comme la dernière clause peut ne pas être remplie, on conclut par de simples transformations.

Exercice 4

```
let tous_noms clause =
  let dico = Hashtbl.create 8 in
  let f (_, nom) = if not (Hashtbl.mem dico nom) then Hashtbl.add dico nom 0
  in List.iter f clause;
  dico;;
```

On a besoin de tous les noms des variables pour assurer qu'en créant les `"gamma1"` et suivants il ne s'agisse pas accidentellement de vrais noms de variable propositionnelle.

La boucle qui augmente d'un l'entier en suffixe consulte alors la table de hachage, en ajoutant une entrée dans le dictionnaire (non nécessaire : l'indice est de toute manière incrémenté).

On note que si cette précaution était nécessaire, alors l'indice ne serait plus conforme à l'indice de la clause, mais tant pis (c'est plus important d'éviter une homonymie).

```
let decoupe clause n =
  assert (n > 2);
  let indice = ref 1 in
  let dico = tous_noms clause in
  let rec aux n_clauses n_clause taille l = match l with
  | [] ->  List.rev ((List.rev n_clause)::n_clauses)
  | [a] -> List.rev ((List.rev (a::n_clause))::n_clauses)
  | _ when taille = n-1 ->
    while Hashtbl.mem dico ("gamma" ^ (string_of_int !indice)) do incr indice done;
    let gamma = "gamma" ^ (string_of_int !indice) in
    Hashtbl.add dico gamma 0;
    incr indice;
    aux ((List.rev ((true, gamma)::n_clause))::n_clauses) [(false, gamma)] 1 l
  | a::q -> aux n_clauses (a::n_clause) (taille+1) q
  in aux [] [] 0 clause;;
```

Exercice 5

On considère tout de même les composantes fortement connexes pour la preuve, même si elles ne seront pas concernées par la résolution dans les questions suivantes.

Tout d'abord, on remarque que le graphe obtenu en fusionnant les composantes fortement connexes (selon la définition où elles sont maximales pour l'inclusion) d'un graphe selon une méthode intuitive est sans circuit (sinon, un circuit ferait que tous les sommets de toutes les composantes connexes du circuit seraient reliés par un chemin dans le graphe initial, contredisant la maximalité des composantes connexes).

Dans le cas du graphe d'implication, une composante fortement connexe rassemble des littéraux qui doivent tous être simultanément vrais ou simultanément faux pour que la formule puisse être satisfaite. De plus, une composante connexe ayant un arc vers une autre dans le graphe fusionné impose que si les littéraux qui lui appartiennent sont vrais, alors les littéraux qui appartiennent à l'autre le sont aussi.

En fin de compte, on est en mesure de construire un tri topologique du graphe fusionné (éventuellement via plusieurs parcours si certains sommets ne sont pas accessibles depuis le sommet où le premier parcours démarre, quel que soit le choix de celui-ci), qu'on transforme en liste des sommets en décomposant les composantes fortement connexes, et on décide que la dernière occurrence de chaque variable propositionnelle dans un littéral sera le littéral vrai parmi les deux (autrement dit, si $\neg v$ apparaît avant v alors v sera vraie, et sinon v sera fausse). Ceci constitue une interprétation qui satisfait la formule.

En pratique, cela revient à prendre dans le graphe acyclique fusionné une composante fortement connexe sans prédécesseur et à rendre faux tous ses littéraux. Les propriétés de symétrie du graphe font qu'à cette composante fortement connexe correspond une composante fortement connexe sans successeur qui contient tous les littéraux contraires, qui seront rendus vrais. Alors on peut continuer en ignorant ces deux composantes fortement connexes (autant dire en les retirant) et en continuant ainsi jusqu'à avoir laissé de côté le graphe fusionné en entier.

Exercice 6

Pour les clauses, chaque variable sera assimilée à un entier, et le booléen qui précède indique si la variable est telle quelle (cas de `true`) ou associée à un symbole de négation (cas de `false`). Quant aux graphes, ils seront donnés par liste d'adjacence, en représentant la version positive de la variable i par le sommet $2i$ et sa version négative par le sommet $2i + 1$.

```
type clause_2 = bool * int * bool * int;;
type graphe = int list array;;
```

Exercice 7

Une formule en 2-FNC sera une liste de clauses au sens de la question précédente, d'où la fonction :

```
let graphe_implication l = assert (l <> []);
  let max_var = List.fold_left (fun m (_, i, _, j) -> max m (max i j)) (-1) l in
  let indice (b, i) = 2 * i + if b then 0 else 1 in
  let graphe = Array.make (2*max_var+2) [] in
  let ajoute_arc (bs, s, bt, t) =
    let ind_s = indice (bs, s) and ind_t = indice (bt, t)
    and ind_n_s = indice (not bs, s) and ind_n_t = indice (not bt, t) in
    graphe.(ind_n_s) <- ind_t :: graphe.(ind_n_s);
    graphe.(ind_n_t) <- ind_s :: graphe.(ind_n_t)
  in List.iter ajoute_arc l; graphe;;
```

Exercice 8

On reprend une fonction de la correction du TP 12, adaptée à la structure ici.

```
exception Trouve;;

let existe_chemin graphe s t =
  let ouverts = ref [s] and fermes = ref [] in
    try
      while !ouverts <> [] do
        let sommet = List.hd (!ouverts) in
          if sommet = t then raise Trouve;
          ouverts := List.tl !ouverts;
          fermes := sommet::(!fermes);
          let rec parcours_voisins = function
          | [] -> ()
          | (a::q) -> if not (List.mem a !ouverts || List.mem a !fermes)
              then ouverts := a::(!ouverts);
            parcours_voisins q
          in parcours_voisins graphe.(sommet)
      done; false
    with Trouve -> true;;
```

Ensuite, on teste les chemins d'un sommet vers sa version négative et vice-versa.

```
let satisfaisable_chemin graphe =
  let n = Array.length graphe in
  let rec test i = i = n ||
    (not (existe_chemin graphe i (i+1)) || not (existe_chemin graphe (i+1) i))
    && test (i+2)
  in test 0;;
```

SAT et coloration

Exercice 9

La formule est $\bigwedge_{i=0}^{n-1} \bigvee_{j=0}^{k-1} c_{i,j}$.

Exercice 10

La formule est $\bigwedge_{i=0}^{n-1} \bigwedge_{0 \leq a < b < k} \neg c_{i,a} \vee \neg c_{i,b}$.

Exercice 11

On note A l'ensemble des arêtes du graphe, en tant qu'ensemble de couples de sommets dont le premier est conventionnellement strictement inférieur au second.

La formule est $\bigwedge_{(s,t) \in A} \bigwedge_{j=0}^{k-1} \neg c_{s,j} \vee \neg c_{t,j}$.

Exercice 12

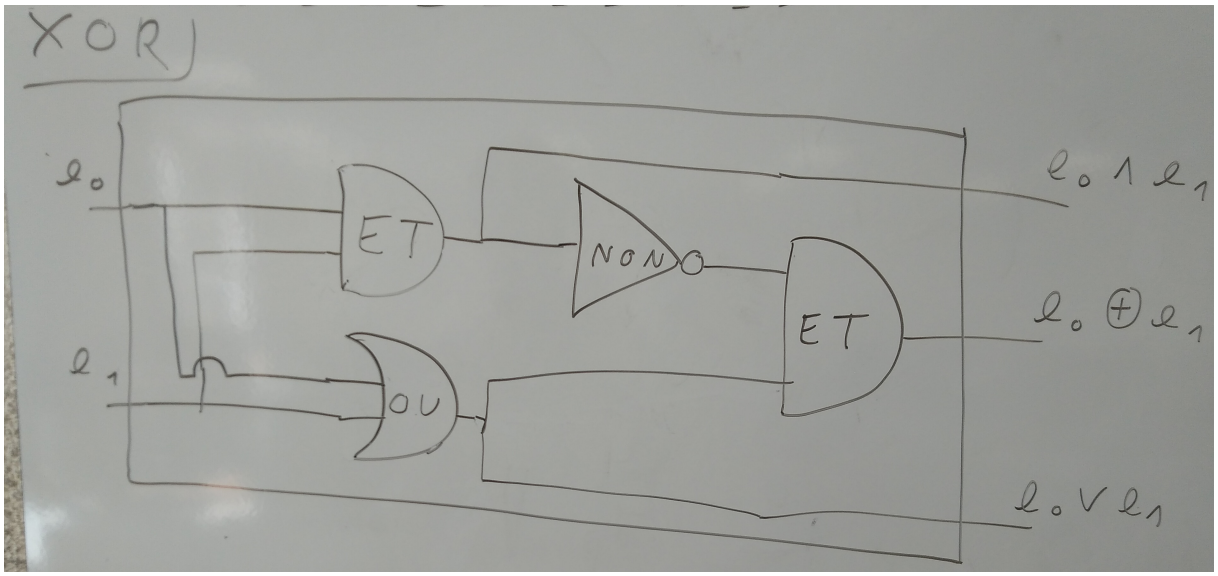
```
type f_b = Vrai | Faux | Var of string | Non of f_b | Et of f_b list | Ou of f_b list;;
type graphe1 = { nb_s : int; aretes : (int * int) list };;

let couleur sommet coul = Var(Printf.sprintf "c_%d_%d" sommet coul);;

let formule_1 g k =
  let rec aux i =
    if i = g.nb_s then []
    else Ou(List.init k (couleur i)) :: aux (i+1)
  in aux 0;;

let formule_2 g k =
  let rec aux i a b =
    if i = g.nb_s then []
    else if a = k then aux (i+1) 0 1
    else if b = k then aux i (a+1) (a+2)
    else Ou([Non(couleur i a); Non(couleur i b)]) :: aux i a (b+1)
  in aux 0 0 1;;
```

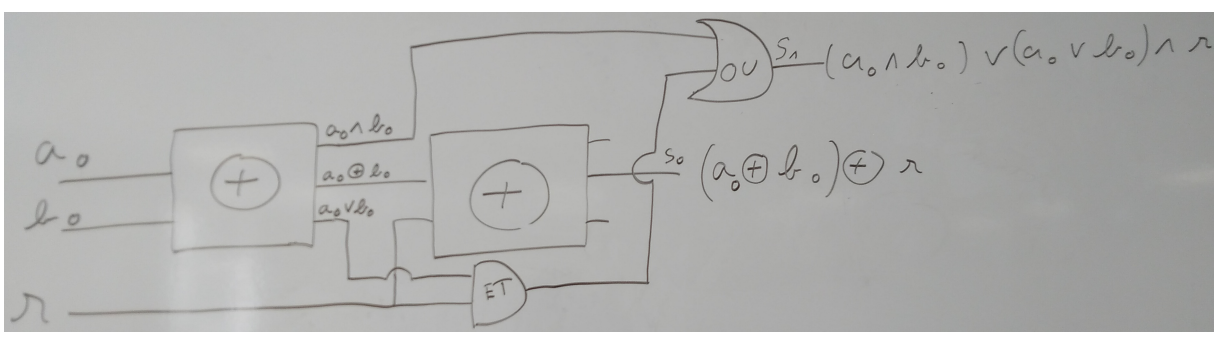
```
let formule_3 g k =
  let rec aux s t j =
    if j = k then []
    else Ou([Non(couleur s j); Non(couleur t j)]) :: aux s t (j+1)
  in List.fold_left (fun accu (s, t) -> aux s t 0 @ accu) [] g.aretes;;

let formule g k =
  Et((formule_1 g k) @ (formule_2 g k) @ (formule_3 g k));;
```

Circuits logiques

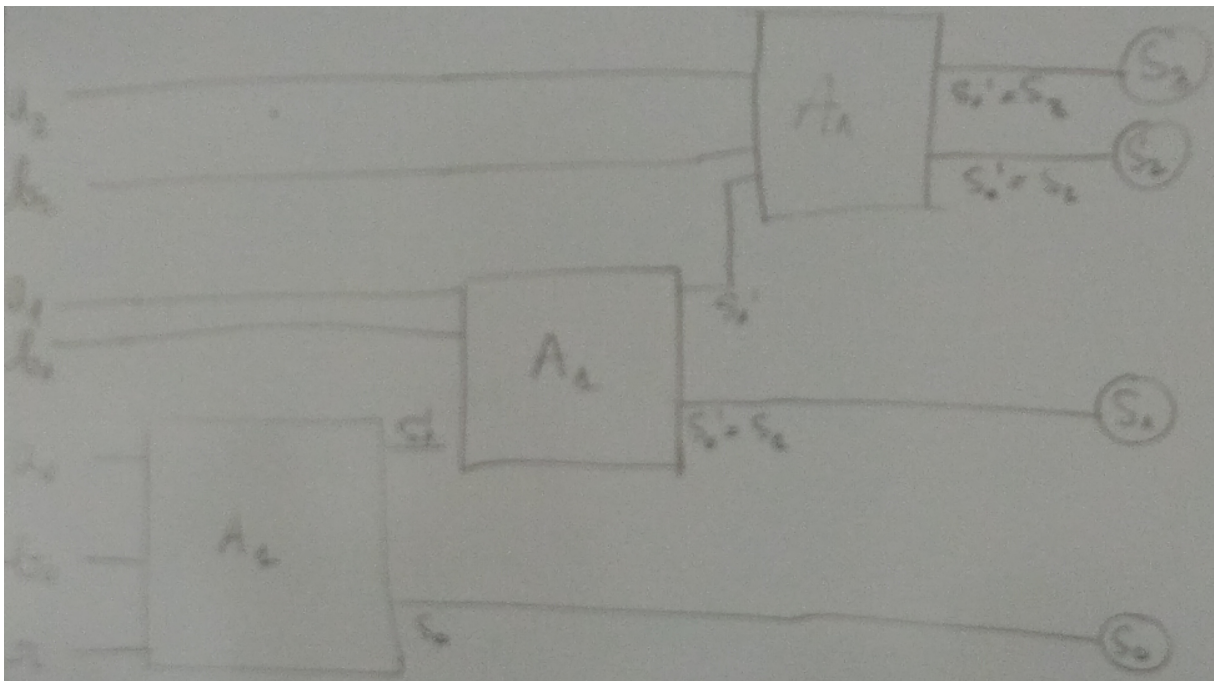
Exercice 13

Question 13.1





Question 13.2



Question 13.3

Une fois qu'on a compris le principe pour 3 bits, la généralisation à n bits est évidente. Avec notre additionneur de base utilisant deux pseudo-portes réalisant un ou exclusif à l'aide de quatre portes chacune, plus deux portes, qui font dix portes, on a besoin de n additionneurs, soit $10n$ portes pour une addition de nombres à n chiffres binaires chacun.

Question 13.4

Ici, c'est la profondeur du circuit qui compte. Les sorties d'une pseudo-porte réalisant un ou exclusif nécessitent une porte pour la sortie c, idem pour la sortie d et trois pour la sortie s.

Un additionneur utilise la sortie s deux fois de suite pour produire s_0, d'où 6 portes, et c'est le maximum parmi les chemins possibles. La sortie s_1 ne nécessite que le passage par trois portes.

Ainsi, l'additionneur complet réutilisant les sorties s_1 de n additionneurs à un bit consécutifs, il faut passer deux portes pour chaque additionneur sauf le premier où les bits hors retenue doivent en passer trois pour arriver à la retenue et le dernier où il faut en passer trois pour la sortie hors retenue. Le plus long chemin passe donc par $3+2(n-2)+3$ portes dès que $n \geq 2$ (et 6 dans le cas particulier où $n = 1$), et (après une rapide factorisation) le temps mis est donc $2(n+1)t$ secondes pour $n \geq 2$ et $6t$ pour $n = 1$.

Bien entendu, un additionneur optimisé donnerait d'autres résultats, mais la partie laissée de côté après les exercices donnés ici propose une optimisation bien plus efficace.

Question 13.5

(pas de photo cette fois)

Avec la représentation d'un additionneur DPR proposée, on observe que le plus long chemin se réalise en passant par les deux additionneurs via la sortie « retenue » du premier.

En fait, il faut disposer de quatre valeurs :

- nn_k : nombre maximal de portes rencontrées dans un additionneur DPR de 2^k bits en partant d'un bit hors retenue et en finissant sur une sortie hors retenue.
- nr_k : idem mais en finissant sur la sortie de la retenue.
- rn_k : partant du bit de retenue et finissant sur une sortie hors retenue.
- rr_k : partant du bit de retenue et finissant sur la sortie de la retenue.

Alors on a les initialisations suivantes pour $n = 1 = 2^0$ sur l'additionneur à un bit déjà rencontré : $nn_0 = 6$, $nr_0 = 3$, $rn_0 = 3$ et $rr_0 = 2$.

Par suite, on calcule nn_{k+1} en tant que maximum entre nn_k et $nr_k + rn_k$, ainsi que nr_{k+1} en tant que $nr_k + rr_k$ (forcément supérieur à nr_k), de même rn_{k+1} en tant que $rr_k + rn_k$ (même remarque) et $rr_{k+1} = 2rr_k$.

Finalement, on a vite que $rr_k = 2^{k+1}$, ce qui permet de déduire que $rn_k = 3 + (2^{k+1} - 2) = 2^{k+1} + 1$, $nr_k = 2^{k+1} + 1$ et $nn_k = 2^{k+1} + 2$ dès que $k \geq 2$ (le maximum était évidemment réalisé par la deuxième formule, avec $nn_1 = 6$ mais les valeurs coïncident en pratique).

Le nombre de portes maximal est alors $2^{k+1} + 2$ pour $k \geq 1$, d'où un temps de $(2^{k+1} + 2)t = 2(n+1)t$ secondes.