DS n° 03 — corrigé

Questions de cours

i) La ligne de compilation en C que vous devez désormais systématiquement utiliser est :

```
gcc -g -Wall -Wextra -fsanitize=address -o prog.exe prog.c
```

ii) Un automate déterministe fini est un quintuplet $\mathcal{A} = (Q, \Sigma, q_0, T, \delta)$ avec Q un ensemble fini non vide, Σ un alphabet, $q_0 \in Q$, $T \subseteq Q$ et $\delta : \mathcal{D}_\delta \subseteq Q \times \Sigma \to Q$.

iii) Pour montrer la correction de l'algorithme de Kruskal on montre que l'on maintient l'invariant « le sous-graphe en cours de construction est un sous-graphe acyclique d'un arbre couvrant de poids minimal ».

iv) On propose des arbres de preuves pour montrer que les séquents sont dérivables.

(a) En logique minimale.

$$\dfrac{\dfrac{\dfrac{}{A \land B, B \land C \vdash A \land B}\,\text{ax}}{A \land B, B \land C \vdash A}\,\land_e^g \qquad \dfrac{\dfrac{}{A \land B, B \land C \vdash B \land C}\,\text{ax}}{A \land B, B \land C \vdash C}\,\land_e^d}{A \land B, B \land C \vdash A \land C}\,\land_i$$

(b) En logique minimale. On note $\Gamma = \{A \to (B \lor C), \neg B, \neg C, A\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash A \to (B \lor C)}\,\text{ax} \quad \dfrac{}{\Gamma \vdash A}\,\text{ax}}{\Gamma \vdash B \lor C}\,\to_e \quad \dfrac{\dfrac{}{\Gamma, B \vdash B}\,\text{ax} \quad \dfrac{}{\Gamma, B \vdash \neg B}\,\text{ax}}{\Gamma, B \vdash \bot}\,\neg_e \quad \dfrac{\dfrac{}{\Gamma, C \vdash C}\,\text{ax} \quad \dfrac{}{\Gamma, C \vdash \neg C}\,\text{ax}}{\Gamma, C \vdash \bot}\,\neg_e}{\Gamma \vdash \bot}\,\lor_e}{A \to (B \lor C), \neg B, \neg C \vdash \neg A}\,\neg_i$$

(c) En logique intuitionniste. On note $\Gamma = \{(A \lor C) \to (B \lor C), \neg C, A\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash (A \lor C) \to (B \lor C)}\,\text{ax} \quad \dfrac{\dfrac{}{\Gamma \vdash A}\,\text{ax}}{\Gamma \vdash A \lor C}\,\lor_i^g}{\Gamma \vdash B \lor C}\,\to_e \quad \dfrac{}{\Gamma, B \vdash B}\,\text{ax} \quad \dfrac{\dfrac{\dfrac{}{\Gamma, C \vdash C}\,\text{ax} \quad \dfrac{}{\Gamma, C \vdash \neg C}\,\text{ax}}{\Gamma, C \vdash \bot}\,\neg_e}{\Gamma, C \vdash B}\,\bot_e}{\Gamma \vdash B}\,\lor_e}{(A \lor C) \to (B \lor C), \neg C \vdash A \to B}\,\to_i$$

(d) En logique classique. On note $\Gamma = \{C \to B, \neg C \to \neg A, A\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, C \vdash C \to B}\,\text{ax} \quad \dfrac{}{\Gamma, C \vdash C}\,\text{ax}}{\Gamma, C \vdash B}\,\to_e \quad \dfrac{\dfrac{\dfrac{}{\Gamma, \neg C \vdash A}\,\text{ax} \quad \dfrac{\dfrac{}{\Gamma, \neg C \vdash \neg C \to \neg A}\,\text{ax} \quad \dfrac{}{\Gamma, \neg C \vdash \neg C}\,\text{ax}}{\Gamma, \neg C \vdash \neg A}\,\to_e}{\Gamma, \neg C \vdash \bot}\,\neg_e}{\Gamma, \neg C \vdash B}\,\bot_e}{\Gamma \vdash B}\,\text{t.e.}}{C \to B, \neg C \to \neg A \vdash A \to B}\,\to_i$$

v) On rappelle que l'on peut utiliser le théorème de correction de la déduction naturelle : si $\Gamma \vdash A$ alors $\Gamma \vDash A$ mais pas celui de complétude qui n'est pas au programme.

(a) Ce séquent est dérivable comme le montre l'arbre de preuve suivant, dans lequel on pose $\Gamma = \{\neg(A \to B), B\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, A \vdash B}\text{ax}}{\Gamma \vdash A \to B}\to_i \qquad \dfrac{}{\Gamma \vdash \neg(A \to B)}\text{ax}}{\Gamma \vdash \bot}\neg_e}{\Gamma \vdash A}\bot_e}{\neg(A \to B) \vdash B \to A}\to_i$$

(b) Pour $A = x$ et $B = y$, avec $x, y \in \mathcal{V}$ et avec la valuation $\nu(x) = V$ et $\nu(y) = F$, on a $\tilde{\nu}(\neg(A \to B)) = V$ mais $\tilde{\nu}(A \to B) = F$. On n'a donc pas $\neg(A \to B) \models A \to B$. Le théorème de correction de la déduction naturelle implique donc que l'on n'a pas non plus $\neg(A \to B) \vdash A \to B$ et ce séquent n'est donc pas dérivable.

vi) On procède par induction sur $u \in \Sigma^*$ (ou par récurrence sur $n = |u| \in \mathbb{N}$ ce qui revient exactement au même). Si $u = \epsilon$ alors $\delta^\star(q, u) = q$ par définition de $\delta^\star$ et $\delta^\star(q, uv) = \delta^\star(q, v) = \delta^\star(\delta^\star(q, u), v)$. Si $u = au'$ avec $a \in \Sigma$ et $u' \in \Sigma^*$ avec $\delta^\star(q, u'v) = \delta^\star(\delta^\star(q, u'), v)$. Alors on a :

$$\begin{aligned}
\delta^\star(q, uv) &= \delta^\star(q, au'v) & \\
&= \delta^\star(\delta(q, a), u'v) & \text{par définition de } \delta^\star \\
&= \delta^\star(\delta^\star(\delta(q, a), u'), v) & \text{par hypothèse d'induction} \\
&= \delta^\star(\delta^\star(q, au'), v) & \text{par définition de } \delta^\star \\
&= \delta^\star(\delta^\star(q, u), v) &
\end{aligned}$$

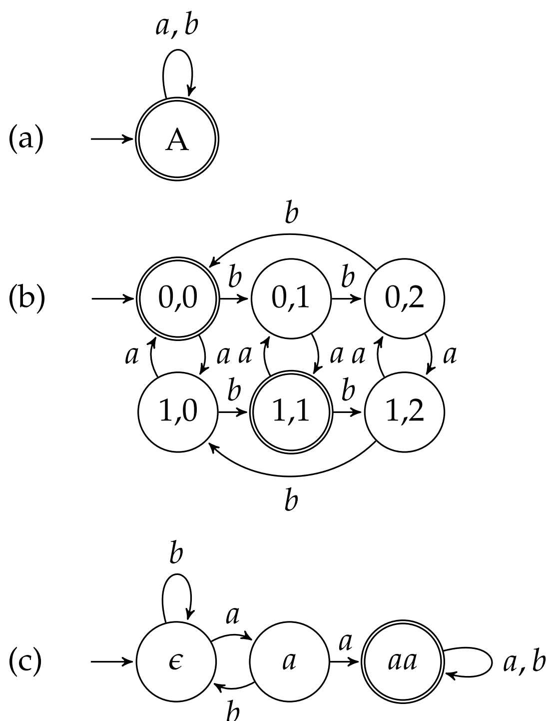
vii) Pour chacun des automates suivants, donner une expression régulière décrivant le langage qu'il reconnaît.

(a) acb^*

(b) $(a^2)^*$

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

viii) On propose les automates suivants :



ix) On parcourt les arêtes pour ajouter l'arête inverse dans un graphe initialement vide.

OCaml

```
let rec miroir g =
  let m = Array.make (nb_sommets g) [] in
  let add s t = m.(t) <- s :: m.(t) in
  for s = 0 to (nb_sommets g - 1) do
    List.iter (add s) (voisins g s)
  done;
  m
```

x) Il s'agit d'un classique parcours en profondeur où l'on ajoute dans une pile implémentée par une référence de liste les sommets lorsque l'on en a terminé avec eux.

OCaml

```
let postfixe g =
  let vu = Array.make (nb_sommets g) false in
  let pos = ref [] in
  let rec explorer s =
    if not vu.(s) then begin
      vu.(s) <- true;
      List.iter explorer (voisins g s);
      pos := s :: !pos
    end
  in
  for s = 0 to (nb_sommets g - 1) do
    explorer s
  done;
  !pos
```

xi) Il s'agit encore d'un simple parcours où l'on ajoute l'identifiant de la composante en cours lorsque l'on découvre un nouveau sommet. À chaque nouveau départ on a trouvé une nouvelle composante connexe et on incrémente l'identifiant.

OCaml

```
let composantes_connexes g ordre =
  let comp = Array.make (nb_sommets g) (-1) in
  let id = ref (-1) in
  let rec explorer s =
    if comp.(s) < 0 then begin
        comp.(s) <- !id;
        List.iter explorer (voisins g s)
    end
  in
  let depart s =
    if comp.(s) < 0 then incr id;
    explorer s
  in
  List.iter depart ordre;
  comp
```

xii) Il suffit de mettre les choses dans le bon ordre :

OCaml
```
let kosaraju g =
  composantes_connexes g (postfixe (miroir g))
```

Le corrigé pour la partie I est disponible après celui pour la partie II.

Partie II. Arbres couvrants et algorithme de Borůvka

Q 12) (a) On se contente de trouver la sentinelle -1 et de diviser par deux l'indice où elle se trouve.

C
```
int degre(graphe g, int s) {
  int i = 0;
  while (g.adj[s][i] != -1) {i++;}
  return i / 2;
}
```

(b) On compte deux fois chaque arête donc on divise par deux à la fin.

C
```
int poids_graphe(graphe g) {
  int p = 0;
  for (int s = 0; s < g.n; s++) {
    int i = 0;
    while (g.adj[s][2 * i] != -1) {
      p += g.adj[s][2 * i + 1];
      i++;
    }
  }
  return p / 2;
}
```

(c) Il faut libérer les lignes du tableau pointeurs `adj` et le tableau lui-même.

C
```
void liberer_graphe(graphe g) {
  for (int s = 0; s < g.n; s++) {free(g.adj[s]);}
  free(g.adj);
}
```

Q 13) (a) L'algorithme de Kruskal parcourt les arêtes par poids croissants et les ajoute dans le sous-graphe acyclique en cours de construction lorsque que celles-ci ne créent pas de cycle. On maintient une structure *unir et trouver* pour vérifier efficacement qu'une arête ne connecte pas deux composantes connexes du graphe en cours de construction.

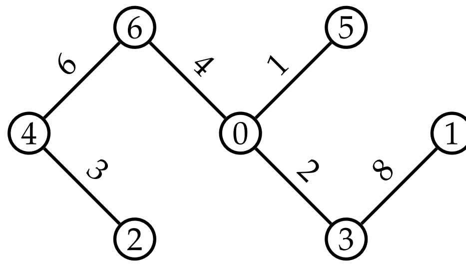


FIGURE 1 – L'arbre couvrant de poids minimal du graphe G.

(b) On considère les arêtes de poids : 1 (ajout), 2 (ajout), 3 (ajout), 4 (ajout), 5 (ignoré), 6 (ajout), 7 (ignoré), 8 (ajout). On peut s'arrêter à ce stade car on a ajouté 6 arêtes pour 7 sommets donc le sous-graphe acyclique est connexe et on a terminé. On obtient l'arbre représenté à la figure 1.

(c) On peut commencer par trier les arêtes pour une complexité en $\mathcal{O}(|A| \log |A|)$ qui est un $\mathcal{O}(|A| \log |S|)$. On effectue ensuite dans le pire cas une boucle sur toutes les arêtes en $\mathcal{O}(|A|)$ en effectuant deux appels à *trouver* et un éventuel appel à *unir* qui sont de complexité en $\mathcal{O}(\log |S|)$ dans le pire cas avec l'heuristique d'union par rang. Au final la complexité est en $\mathcal{O}(|A| log |S|)$.

Q 14) L'arête sûre pour la composantes connexe $\{1\}$ est l'arête de poids 8.

Q 15) L'algorithme de Kruskal ignore les arêtes qui relient deux sommets dans une même composante connexe, qui sont précisément les arêtes inutiles. À chaque étape, l'algorithme de Kruskal ajoute la plus petite arête parmi toutes les arêtes non inutiles, qui est donc en particulier une arête sûre et a fortiori la plus petite parmi les arêtes sûres.

Q 16) (a) Une arête inutile créerait un cycle dans H et donc dans B^* qui est acyclique. Donc B^* ne contient aucune d'arête inutile.

(b) Soit C une composante connexe de H, $a = \{s, t\}$ une arête sûre pour C avec $s \in C$ et supposons que l'arbre couvrant de poids minimal $T^* = (S, B^*)$ ne contient pas a. T^* étant connexe, il contient un chemin de s à t. Comme $s \in C$ et $t \notin C$, ce chemin contient une arête b entre un sommet de C et un sommet qui n'est pas dans C. On a $p(b) > p(a)$ car a est une arête sûre et que les arêtes ont toutes un poids différent. On considère alors le sous-graphe $\hat{T} = (S, B^* \cup \{a\} \setminus \{b\})$ qui est encore un arbre couvrant de G et dont le poids est strictement inférieur à celui de T^*, ce qui contredit la minimalité de T^*.

Q 17) Il s'agit à nouveau d'une simple et classique adaptation d'un parcours (ici en profondeur). On ne peut pas définir de fonctions locales en C, mais on peut quand même définir des fonctions auxiliaires.

C

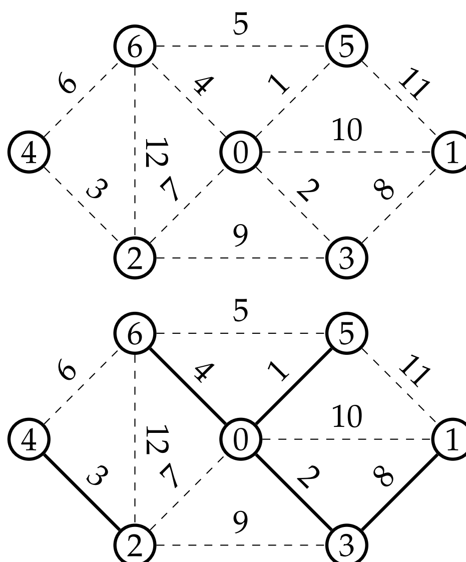
```c
void explorer(graphe h, int s, int* comp, int id) {
  if (comp[s] < 0) {
    comp[s] = id;
    int i = 0;
    while (h.adj[s][2 * i] != -1) {
      explorer(h, h.adj[s][2 * i], comp, id);
      i++;
    }
  }
}
```

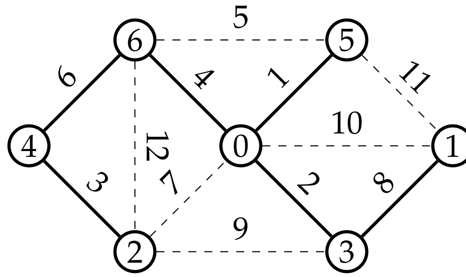
C

```c
int* composantes(graphe h, int* m) {
  *m = -1;
  int* comp = malloc(sizeof(int) * h.n);
  for (int s = 0; s < h.n; s++) {comp[s] = -1;}
  for (int s = 0; s < h.n; s++) {
    if (comp[s] < 0) {
      *m += 1;
      explorer(h, s, comp, *m);
    }
  }
  return comp;
}
```

Q 18) Pour obtenir l'ensemble des arêtes sûres pour un sous-graphe acyclique H, on commence par calculer ses composantes connexes avec l'algorithme précédent en en $\mathcal{O}(|S| + |A|)$. On peut ensuite créer un tableau d'arêtes initialement vide destiné à contenir, pour chaque composante connexe, l'arête candidate pour être une arête sûre, c'est-à-dire l'arête ayant exactement une extrémité dans cette composante connexe et minimale parmi celles ayant cette propriété. On parcourt ensuite toutes les arêtes, en $\mathcal{O}(|A|)$, en ignorant les arêtes inutiles (il suffit de vérifer en $\mathcal{O}(1)$ si les identifiants de composante de ses extrémités sont les mêmes) et en mettant à jour le tableau des arêtes sûres, à nouveau en $\mathcal{O}(1)$. La complexité obtenue est bien en $\mathcal{O}(|S| + |A|))$ qui est un $\mathcal{O}(|A|)$ si le graphe est connexe.

Q 19) (a) On obtient sucessivement les étapes ci-dessous. Les arêtes sûres sont celles ajoutées entre deux étapes. Pour la toute première étape, chaque sommet étant seul dans sa composante connexe, il n'y a pas d'arêtes inutiles et on ajoute l'arête incidente de poids minimal à chaque sommet. L'arête de poids 1 et l'arête de poits 3 sont toutes les deux sûres pour deux composantes. On se retrouve avec deux composantes comportant toutes les deux l'arête de poids 6 comme arête sûre et c'est terminé, il n'y a ensuite plus d'arêtes sûres. On retrouve bien l'arbre couvrant de poids minimal.





(b) À chaque étape on ajoute à B au moins une arête sûre pour H ce qui diminue strictement le nombre de composantes connexes de H. Ce nombre est donc un variant de boucle, ce qui montre bien la terminaison.

(c) La propriété « H est acyclique » est un invariant de boucle. En effet, à chaque étape, ajouter les arêtes de l'ensemble des arêtes sûres calculées ne peut pas créer de cycle. En effet, supposons qu'ajouter ces arêtes sûres créé un cycle. Comme le graphe était acyclique, ses composantes connexes l'étaient également. Une arête sûre relie deux composantes connexes donc il ne peut pas y avoir de cycle au sein des composantes connexes. Ce cycle passe donc par au moins deux composantes connexes. Notons $C_1, ..., C_k, C_1$ les composantes connexes parcourues par ce cycle, qui passe donc par des arêtes sûres $a_1, ..., a_k$ deux à deux distinctes ajoutées, chaque a_i étant une arête entre C_i et C_{i+1} (où l'on assimile C_{k+1} avec C_1). Sans perte de généralité, on peut supposer que a_1 est l'arête minimale parmi ces arêtes. Mais alors a_1 est aussi une arête ayant exactement une extrémité dans C_k de poids strictement inférieur à a_k, ce qui est absurde puisque a_k est une arête sûre pour C_k. La propriété « H est un sous-graphe de l'arbre couvrant de poids minimal T^* » est également un invariant puisqu'à chaque étape les arêtes ajoutés à H sont des arêtes de T^* par la question 16) b). Lorsque l'algorithme termine, le graphe H est connexe (sinon il possède encore des arêtes sûres). C'est donc un arbre couvrant de G qui est un sous-graphe de T^* : c'est donc T^*.

(d) On remarque que le nombre de composantes connexes est au moins divisé par deux à chaque passage dans la boucle **Tant que**. En effet, à chaque composante connexe est associée une arête sûre. Une arête sûre, elle, peut être associée à au plus deux composantes connexes. Il y a donc au moins $\frac{m}{2}$ arêtes sûres à chaque itération, m étant le nombre de composantes connexes. On en déduit qu'il y a au plus $\log |S|$ passages dans la boucle. Chaque passage dans la boucle fait le calcul des composantes connexes, puis des arêtes sûres, puis de l'ajout des arêtes au graphe H. Toutes ces opérations se font en temps $\mathcal{O}(|S| + |A|)$. La complexité totale est donc en $\mathcal{O}((|S| + |A|) \log |S|) = \mathcal{O}(|A| \log |S|)$, tout comme l'algorithme de Kruskal. En pratique, cette complexité n'est généralement pas atteinte. Il s'agit d'un pire cas où toutes les arêtes sûres à chaque étape sont celles d'exactement deux composantes connexes. En pratique, comme on a d'ailleurs pu le voir dans l'exemple ci-dessus, la situation est plus favorable. En pratique, l'algorithme de Borůvka est plus rapide que l'algorithme de Kruskal, même si la complexité dans le pire cas est identique.

CCP 2016 - OPTION INFORMATIQUE - UN CORRIGÉ

Corrigé proposé par Alain Schauber : alain.schauber@prepas.org

Partie I. Logique et calcul des propositions

I.1. Soit $A \in \mathcal{V}$ et f une tri-valuation définie sur $\mathcal{V}$ telle que $f(A) = ?$, donc $\hat{f}(\neg A) = ?$ et par suite $\hat{f}(A \vee \neg A) = ? \neq \top$.

$$\boxed{\not\models_3 (A \vee \neg A)}$$

I.2. Telle que définie dans l'énoncé, la logique tri-valuée ne reconnaît pas les formules constantes $\bot$ et $\top$, sinon cette dernière aurait pu fournir une réponse à la question.
On peut cependant proposer : $A \Rightarrow A$. En effet, si A est une variable propositionnelle alors quelle que soit la tri-valuation f, on a $\hat{f}(A \Rightarrow A) = \top$, car les lignes de la table de vérité de $A \Rightarrow B$ correspondant à deux tri-valuations identiques pour A et B fournissent une tri-valuation égale à $\top$ pour $A \Rightarrow B$. Par induction structurelle (ou à l'aide du théorème de substitution), ce résultat reste vrai si on prend pour A une formule logique quelconque.

$$\boxed{\models_3 (A \Rightarrow A)}$$

I.3. On peut proposer pour tout couple $(A, B) \in \mathcal{V}^2$ et toute tri-valuation f :

$$\boxed{\hat{f}(A \wedge B) = \min(f(A), f(B)) \text{ et } \hat{f}(A \vee B) = \max(f(A), f(B))}$$

I.4. Il est clair en observant la table de vérité de $A \vee B$ que les propositions $A \vee B$ et $B \vee A$ sont équivalentes. Par suite, si on avait $\neg A \vee B$ équivalente à $A \Rightarrow B$ on aurait (par transitivité et spécialisation) équivalence entre les propositions $A \vee \neg A$ et $A \Rightarrow A$. Mais on a vu en question **I.1.** que $\not\models_3 (A \vee \neg A)$ alors que dans la question **I.2.** on a établi que $\models_3 (A \Rightarrow A)$. En conclusion :

$$\boxed{\text{Les propositions } \neg A \vee B \text{ et } A \Rightarrow B \text{ ne sont pas équivalentes en logique tri-valuée}}$$

I.5. Comme demandé, on construit la table de vérité conjointe des deux propositions pour les comparer :

A	B	$A \Rightarrow B$	$\neg B$	$\neg A$	$\neg B \Rightarrow \neg A$
$\top$	$\top$	$\top$	$\bot$	$\bot$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\top$	?	?	?	$\bot$	?
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
$\bot$	?	$\top$	?	$\top$	$\top$
?	$\top$	$\top$	$\bot$	?	$\top$
?	$\bot$	?	$\top$	?	?
?	?	$\top$	?	?	$\top$

On en déduit que : $\boxed{\text{Les propositions } \neg B \Rightarrow \neg A \text{ et } A \Rightarrow B \text{ sont équivalentes}}$

I.6. Soit $\psi = ((A \Rightarrow B) \wedge ((\neg A) \Rightarrow B)) \Rightarrow B$. On en construit la table de vérité :

A	B	$A \Rightarrow B$	$\neg A \Rightarrow B$	$((A \Rightarrow B) \wedge ((\neg A) \Rightarrow B)$	B	ψ
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$
$\top$	?	?	$\top$	?	?	$\top$
$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\top$
$\bot$	?	$\top$	?	?	?	$\top$
?	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
?	$\bot$	?	?	?	$\bot$	?
?	?	$\top$	$\top$	$\top$	?	?

On constate en observant les deux dernières lignes que : $\boxed{\not\models_3 \psi}$

I.7. La dernière ligne de la table de vérité montre que $\hat{q}(A \to A) = ? \neq \top$, d'où : $\boxed{A \to A \text{ n'est pas une tautologie}}$

I.8. Dans cette question semble se confirmer la remarque de la question **I.2.** concernant l'absence de la constante $\top$ dans l'ensemble des formules de la logique tri-valuée, car sinon elle représenterait une tautologie.
Notons q la tri-valuation de $\mathcal{V}$ dans $\{\top, \bot, ?\}$ définie par : $\forall A \in \mathcal{V}, q(A) = ?$. Pour montrer qu'il n'existe pas de tautologie dans la logique tri-valuée associée aux opérateurs $\{\neg, \wedge, \vee, \to\}$, il suffit d'établir que : $\forall \phi \in \mathcal{F}, \hat{q}(\phi) = ?$.
Montrons cela par induction structurelle. Soit $\phi \in \mathcal{F}$.

- si $\phi \in \mathcal{V}$ alors $\hat{q}(\phi) = q(\phi) = ?$
- supposons $\phi = \neg\phi_0$ et $\hat{q}(\phi_0) = ?$. La table de vérité de l'opérateur $\neg$ montre que dans ce cas $\hat{q}(\phi) = \hat{q}(\neg\phi_0) = ?$
- supposons $\phi = \phi_1 \diamondsuit \phi_2$, où ϕ_1, ϕ_2 vérifient l'hypothèse d'induction et $\diamondsuit$ représente l'un quelconque des opérateurs binaires $\wedge, \vee$ ou $\to$. En remarquant qu'une tri-valuation indéterminée à la fois de ϕ_1 et ϕ_2 correspond à la dernière ligne du tableau de vérité de tous ces opérateurs binaires, et que le résultat est dans les trois cas égal à ?, on en déduit que $\hat{q}(\phi) = \hat{q}\,(\phi_1 \diamondsuit \phi_2) = ?$

$$\boxed{\text{Il n'existe pas de tautologie dans la logique tri-valuée associée aux opérateurs } \{\neg, \wedge, \vee, \to\}}$$

I.9. Cette question est inintelligible ! Et cela pour plusieurs raisons, dont :

- la proposition à étudier n'est pas une formule logique tri-valuée, donc le mot « tautologie » est inadapté.
- l'expression « est équivalent à » n'est pas définie dans le sujet : cela signifie-t-il $\Leftrightarrow$ au sens de la logique bi-valuée ? en un sens analogue non défini de la logique tri-valuée ? ou au sens de « même valeur de vérité » ?
- la présence de $(\models_3 A \to B)$ est pour le moins étrange au regard de ce qu'on a montré dans la question **I.8.**...

Je pense que la question posée porte sur le problème suivant : on sait qu'en logique bi-valuée, pour démontrer un théorème de la forme $H \Rightarrow C$ il suffit de supposer que H est satisfaite et d'en déduire que C est satisfaite. Cette méthode correspond au *lemme de déduction*, qu'on peut énoncer ainsi (on note $\mathcal{T}_2$ l'ensemble des bi-valuations) :

$$\forall A, B \in \mathcal{F}, \forall f \in \mathcal{T}_2,\ \texttt{si}\ \hat{f} \vdash_2 (A)\ \texttt{entraîne}\ \hat{f} \vdash_2 (B)\ \texttt{alors}\ \hat{f} \vdash_2 (A \Rightarrow B)$$

A contrario, la *règle du modus ponens* exprime le fait que si un théorème du type $H \Rightarrow C$ est démontré, il suffit de vérifier H (l'hypothèse) pour pouvoir affirmer que C (la conclusion) est vérifiée, ce qui s'exprime par :

$$\forall A, B \in \mathcal{F}, \forall f \in \mathcal{T}_2,\ \texttt{si}\ \hat{f} \vdash_2 (A \Rightarrow B)\ \texttt{alors}\ \hat{f} \vdash_2 (A)\ \texttt{entraîne}\ \hat{f} \vdash_2 (B)$$

On peut donc exprimer une équivalence (au sens où chacune entraîne l'autre) entre les propositions :

$$\left(\hat{f} \vdash_2 (A)\ \texttt{entraîne}\ \hat{f} \vdash_2 (B)\right) \text{ et } \left(\hat{f} \vdash_2 (A \Rightarrow B)\right)$$

Le but de cette question est donc d'étudier si cette équivalence reste valable en tri-valuation et avec le nouvel opérateur d'implication. Autrement dit, le problème est d'examiner l'énoncé suivant :

$$\forall A, B \in \mathcal{F}, \forall f \in \mathcal{T}_3, \left(\hat{f} \vdash_3 (A)\ \texttt{entraîne}\ \hat{f} \vdash_3 (B)\right)\ \texttt{si et seulement si}\ \left(\hat{f} \vdash_3 (A \to B)\right)$$

qui peut être reformulée à l'aide des opérateurs classiques $\Rightarrow$ et $\Leftrightarrow$ de la logique bi-valuée « ordinaire » :

$$\forall A, B \in \mathcal{F}, \forall f \in \mathcal{T}_3, \left(\hat{f} \vdash_3 (A) \Rightarrow \hat{f} \vdash_3 (B)\right) \Leftrightarrow \left(\hat{f} \vdash_3 (A \to B)\right)$$

Or il est clair en considérant $A, B \in \mathcal{V}$ et $f \in \mathcal{T}_3$ telle que $f(A) = ?$ et $f(B) = \bot$ que la proposition $\left(\hat{f} \vdash_3 (A)\ \texttt{entraîne}\ \hat{f} \vdash_3 (B)\right)$ est vraie (car $\hat{f} \vdash_3 (A)$ est fausse), tandis que $\left(\hat{f} \vdash_3 (A \to B)\right)$ est fausse (car $\hat{f}(A \to B) = ? \neq \top$: voir avant-dernière ligne de la table de vérité de l'opérateur $\to$).

$$\boxed{\text{La proposition } \left(\hat{f} \vdash_3 (A) \text{ entraîne } \hat{f} \vdash_3 (B)\right) \text{ n'est pas équivalente à } \left(\hat{f} \vdash_3 (A \to B)\right)}$$

Remarque : c'est donc le lemme de déduction qui est mis en défaut en logique tri-valuée (et ceci avec les deux définitions de l'implication proposées dans le sujet). On peut toutefois vérifier que le modus ponens reste valable.

I.10. Apparition tardive des constantes propositionnelles... On définit la formule ψ de la question **I.6.**

```
let psi = Implique (Et (Implique (Var "A", Var "B"), Implique (Non (Var "A"), Var "B")), Var "B");;
```

I.11. L'exemple fourni dans le sujet étant assez court, je choisis d'écrire une fonction a minima, c'est-à-dire sans parenthésage des constantes, variables et négations de constantes ou de variables mais en conservant tous les autres parenthésages. En particulier, les règles de priorité usuelles entre opérateurs ne sont pas implémentées.

```
let rec lectureFormule f =
  (* identification des formules logiques qui ne sont pas parenthésées *)
  let est_simple = function
    |Vrai |Faux |Indetermine |Var _
    |Non Vrai |Non Faux |Non Indetermine |Non (Var _) -> true
    |_ -> false
  in match f with
     Vrai -> "vrai"
    |Faux -> "faux"
    |Indetermine -> "Indéterminé"
    |Var s -> s
    |Non g when est_simple g -> "non "^(lectureFormule g)
    |Non g -> "non ("^(lectureFormule g)^")"
    |Et (g,h) when est_simple g && est_simple h -> (lectureFormule g)^" et "^(lectureFormule h)
    |Et (g,h) when est_simple g -> (lectureFormule g)^" et ("^(lectureFormule h)^")"
    |Et (g,h) when est_simple h -> "("^(lectureFormule g)^") et "^(lectureFormule h)
    |Et (g,h) -> "("^(lectureFormule g)^") et ("^(lectureFormule h)^")"
    |Ou (g,h) when est_simple g && est_simple h -> (lectureFormule g)^" ou "^(lectureFormule h)
    |Ou (g,h) when est_simple g -> (lectureFormule g)^" ou ("^(lectureFormule h)^")"
    |Ou (g,h) when est_simple h -> "("^(lectureFormule g)^") ou "^(lectureFormule h)
    |Ou (g,h) -> "("^(lectureFormule g)^") ou ("^(lectureFormule h)^")"
    |Implique (g,h) when est_simple g && est_simple h ->
                   (lectureFormule g)^" implique "^(lectureFormule h)
    |Implique (g,h) when est_simple g -> (lectureFormule g)^" implique ("^(lectureFormule h)^")"
    |Implique (g,h) when est_simple h -> "("^(lectureFormule g)^") implique "^(lectureFormule h)
    |Implique (g,h) -> "("^(lectureFormule g)^") implique ("^(lectureFormule h)^")"
;;
```

Exemples :

```
#lectureFormule (Et(Var "A",Non (Var "B")));;
- : string = "A et non B"

#lectureFormule psi;;
- : string = "((A implique B) et (non A implique B)) implique B"
```

Partie III. Logique propositionnelle

Ce problème utilise le langage OCaml.

Dans ce problème, on considère des formules de logique propositionnelle sur n variables notées X_i avec $0 \leq i < n$. Une valeur de vérité est un élément de $\mathbb{B} = \{V, F\}$. Une valuation est une fonction v de $\{0, 1, \ldots, n-1\}$ dans $\mathbb{B}$, qui assigne une valeur de vérité à chacune des variables. On note $\mathcal{V}_v(f)$ la valeur de vérité de la formule f pour la valuation v.

On choisit de représenter nos formules en OCaml avec le type suivant.

```
type formula =
  | True
  | False
  | If of int * formula * formula
```

La valeur de vérité d'une telle formule est définie de la manière suivante :

$$\begin{array}{rcl}\mathcal{V}_v(\texttt{True}) & = & V \\ \mathcal{V}_v(\texttt{False}) & = & F \\ \mathcal{V}_v(\texttt{If}(i, f, g)) & = & \text{si } v(i) = V \text{ alors } \mathcal{V}_v(f) \text{ sinon } \mathcal{V}_v(g)\end{array}$$

On ajoute par ailleurs la contrainte que toute formule est *ordonnée*, au sens où si elle est de la forme $\texttt{If}(i, f, g)$, alors les formules f et g ne font intervenir que des variables *strictement plus grandes que* i. Ainsi,

$$\texttt{If}(0, \texttt{If}(1, \texttt{False}, \texttt{True}), \texttt{If}(2, \texttt{True}, \texttt{False}))$$

est une formule ordonnée, mais

$$\texttt{If}(0, \texttt{If}(1, \texttt{False}, \texttt{True}), \texttt{If}(0, \texttt{True}, \texttt{False}))$$

n'en est pas une.

Question 20. Écrire une fonction `check: int -> formula -> bool` qui prend en arguments un entier n et une formule f et qui détermine si d'une part f est bien une formule ordonnée et si d'autre part f est limitée aux variables X_i avec $0 \leq i < n$. La complexité doit être linéaire en la taille de la formule. On ne demande pas de justifier la complexité.

Correction : On commence par une fonction qui vérifie le caractère ordonné et que les variables sont dans $[n, m[$:

```
let rec check n m = function
  | True | False -> true
  | If (i, l, r) -> n <= i && i < m && check (i+1) m l && check (i+1) m r
```

La fonction demandée s'en déduit trivialement :

```
let check n f = check 0 n f
```

Question 21. Donner une formule ordonnée pour $n = 3$ variables qui est vraie si et seulement si les trois variables ont la même valeur de vérité.

Correction :
```
If (0,
    If (1, If (2, True, False), False),
    If (1, False, If (2, False, True)))
```

Tautologies. Afin de décider si une formule est une tautologie, on se donne le type OCaml `result` suivant :

```
type assignment = bool array
type result = Tautology | Refutation of assignment
```

Le type `assignment` correspond à une valuation. Une valeur de type `assignment` est un tableau `a` de taille n, où `a.(`i`)` donne la valeur de la variable X_i.

Question 22. Écrire une fonction `decide: int -> formula -> result` qui prend en arguments un entier n et une formule f, supposée ordonnée et sur n variables, et qui renvoie

- `Tautology` si f est une tautologie ;
- `Refutation` v sinon, avec $\mathcal{V}_v(f) = F$.

On s'efforcera de proposer quelque chose de plus efficace que le test systématique de toutes les valuations possibles, en faisant intervenir le caractère ordonné de la formule.

Correction : On exploite ici le fait que la formule est ordonnée : la réfutation trouvée pour `l` ou `r` n'inclut pas de valeur pour la variable i, que l'on peut donc fixer ensuite à loisir.
```
let rec decide n = function
  | False -> Refutation (Array.make n false)
  | True -> Tautology
  | If (i, l, r) ->
      (match decide n l with
       | Tautology -> (match decide n r with
                       | Tautology    -> Tautology
                       | Refutation a -> a.(i) <- false; Refutation a)
       | Refutation a -> a.(i) <- true; Refutation a)
```

Construire des formules arbitraires. Pour montrer que toute formule booléenne classique admet une formule ordonnée équivalente, il suffit de se donner des fonctions sur les formules ordonnées qui correspondent aux connecteurs logiques usuels, telles que la négation, la conjonction, la disjonction, etc.

Question 23. Écrire une fonction `mk_not: formula -> formula` qui reçoit en argument une formule ordonnée f et qui renvoie une formule ordonnée correspondant à la négation $\neg f$, c'est-à-dire une formule ordonnée g telle que, pour toute valuation v, on a $\mathcal{V}_v(g) = \neg\mathcal{V}_v(f)$. Donner, sans la justifier, la complexité de votre code en fonction de la taille de f.

Correction : Pas de difficulté. On fait un simple parcours de la formule :

```
let rec mk_not = function
  | True -> False
  | False -> True
  | If (i, l, r) -> If (i, mk_not l, mk_not r)
```

Il est clair que la formule renvoyée est bien ordonnée et que la complexité est linéaire en la taille de f.

Question 24. Écrire une fonction `mk_or: formula -> formula -> formula` qui reçoit en arguments deux formules ordonnées f et g et qui renvoie une formule ordonnée correspondant à la disjonction $f \vee g$, c'est-à-dire une formule ordonnée h telle que, pour toute valuation v, on a $\mathcal{V}_v(h) = \mathcal{V}_v(f) \vee \mathcal{V}_v(g)$. Donner, sans la justifier, la complexité de votre code en fonction de la taille des formules f et g.

Correction : On fait les simplifications lorsque l'une ou l'autre des formules est `False` ou `True`. Sinon, on compare les deux indices, pour décider en premier lieu sur la variable de plus petit indice, garantissant ainsi le caractère ordonné.

```
let rec mk_or a b = match a, b with
  | True, _ | _, True -> True
  | False, c | c, False -> c
  | If (ia, la, ra), If (ib, lb, rb) ->
      if ia = ib then If (ia, mk_or la lb, mk_or ra rb)
      else if ia < ib then If (ia, mk_or la b, mk_or ra b)
      else If (ib, mk_or a lb, mk_or a rb)
```

La complexité est proportionnelle au produit des tailles de f et de g.

Poisson d'avril. Le poisson d'avril est une espèce extrêmement rare qui n'a jamais été permis d'observer dans la nature. Néanmoins, les ichtyologues ont permis de déterminer les faits suivants :

(F_1) tout poisson d'avril qui ne nage pas en mer chaude a des rayures rouges ;

(F_2) tout poisson d'avril a des nageoires bleues ou n'a pas de rayures rouges ;

(F_3) les poissons d'avril qui vivent dans le corail ne mangent pas de crevettes ;

(F_4) un poisson d'avril mange des crevettes si et seulement s'il nage en mer chaude ;

(F_5) tout poisson d'avril qui a des nageoires bleues nage en mer chaude et vit dans le corail ;

(F_6) tout poisson d'avril qui nage en mer chaude a des nageoires bleues.

On souhaite mettre en application le code OCaml développé plus haut pour démontrer qu'il n'existe pas de poisson d'avril.

Question 25. Expliquer comment construire une formule ordonnée de type `formula` qui est une tautologie si et seulement si les poissons d'avril n'existent pas.

Correction : L'idée est de construire la formule

$$F_1 \Rightarrow (F_2 \Rightarrow (F_3 \Rightarrow (F_4 \Rightarrow (F_5 \Rightarrow (F_6 \Rightarrow \texttt{False})))))$$

Il y a six variables propositionnelles, correspondant aux six critères. On commence par les introduire :

```
let var i = If (i, True, False)
let mc = var 0 (* nage en mer chaude *)
let rr = var 1 (* a des rayures rouges *)
let nb = var 2 (* a des nageoires bleues *)
let vc = var 3 (* vit dans le corail *)
let  c = var 4 (* mange des crevettes *)
```

On se donne ensuite deux fonctions pour construire une implication et une conjonction :

```
let mk_imp a b = mk_or (mk_not a) b
let mk_and a b = mk_not (mk_or (mk_not a) (mk_not b))
```

On peut alors enfin construire la formule :

```
let f1 = mk_imp (mk_not mc) rr
and f2 = mk_or nb (mk_not rr)
and f3 = mk_imp vc (mk_not c)
and f4 = mk_and (mk_imp c mc) (mk_imp mc c)
and f5 = mk_imp nb (mk_and mc vc)
and f6 = mk_imp mc nb
let pas_de_poisson =
  mk_imp f1 (mk_imp f2 (mk_imp f3 (mk_imp f4 (mk_imp f5 (mk_imp f6 False)))))
```

et vérifier sa tautologie avec

```
let () = assert (decide 6 pas_de_poisson = Tautology)
```

Remarque. On aurait pu ici travailler avec d'autres formules équivalentes comme $(F_1 \wedge F2 \wedge F3 \wedge F4 \wedge F5) \Rightarrow \texttt{False}$ ou encore $\neg F_1 \vee \neg F2 \vee \neg F3 \vee \neg F4 \vee \neg F_5$

---

* *
*