Projet LO21: Rapport final

Adrien Burgun

Automne 2020

Résumé

Le projet de ce semestre pour le cours de **LO21** (Algorithmique et Programmation II) porte sur un *« système expert »*. Un système expert est constitué de 3 éléments :

Une base de connaissance, qui prend la forme suivante :

$$A \wedge B \wedge ... \wedge Z \Rightarrow \Omega$$

Où $A, B, ...$ sont les symboles (d'arité zéro, aussi appelés « propositions ») constituant la *prémisse* et Ω est la *conclusion*.

Une base de faits, qui est la liste des symboles ayant la valeur *« Vrai »* (qui correspond à l'état *« Certain »*).
Un symbole ne faisant pas partie de cette liste a par défaut la valeur *« Faux »* (qui correspond à l'état *« Incertain »*).

Un moteur d'inférence, qui, à partir de la base de connaissance et la base de faits, déduit quels autres symboles sont aussi vrais et les ajoute à la base de faits.

Nous définirons d'abords le type *« Règle »*, constituant la base de connaissance. Nous définirons ensuite le type *« BC »* (Base de Connaissance).
Nous décrirons enfin le moteur d'inférence comme implémenté dans ce projet, avec différents exemples.

1 Règles

Soit **Règle** le type représentant une règle sous la forme d'une liste de symboles :

Structure 1 : Règle		
Nom	**Type**	**Description**
symbole	Règle → Symbole	Retourne le nom du symbole correspondant au noeud en tête de liste.
suivant	Règle → Règle	Retourne une référence au prochain élément de la liste, *règle_vide* si l'élément est le dernier de la liste.
nouvelle_règle	((Symbole) × Règle) → Règle	Compose une nouvelle règle à partir du nom d'un symbole et une référence à la prochaine règle.
règle_vide	Règle	La règle vide.
mettre_suivant	(Règle × Règle) → Règle	Modifie une Règle pour y attacher une Règle comme règle suivante ; retourne également la règle modifiée.

Le type « Symbole » correspond dans l'implémentation C à une chaîne de caractères. Le dernier élément d'une telle liste correspond à la conclusion de la règle, tandis que tous les autres éléments appartiennent à la prémisse (contrainte du projet).

Les axiomes sur ces fonctions sont :

— *symbole*(*nouvelle_règle*(s, r)) = s
— *suivant*(*nouvelle_règle*(s, r)) = r
— A ⟵ *nouvelle_règle*(s, r) ; *mettre_suivant*(a, r') ⇒ *suivant*(A) = r'

Nous pouvons noter que :

Liste ≺ Règle [∅ : *règle_vide*, tête : *symbole*, reste : *suivant*, insérer_tête : *nouvelle_règle*, Élement : Symbole]

(Soit également un type concret implémentant Règle implémente également Liste(Symbole))

Nous avons implémenté ce type abstrait de donnée sous forme d'une liste chaînée, de part sa facilité d'implémentation et la difficulté d'implémenter un type abstrait de donnée grandissable par la tête sous tout autre forme qu'une liste chaînée.

Le fait que ce type abstrait soit grandissable par la tête reflète également la difficulté de décrire de manière concise et simple à comprendre toute autre forme de type abstrait représentant une liste.

1.1 Créer une règle vide

Nous représentons une règle vide par *règle_vide*. Voici l'algorithme permettant de créer une règle vide :

Algorithme 1 : RègleVide

Variables : R: La règle vide à retourner
Résultat : R: Règle
Classe : $\mathcal{O}(1)$

```
Début RègleVide()
  | R ⟵ règle_vide
Fin
```

1.2 Ajouter une proposition à la prémisse d'une règle

L'ajout des propositions (symboles) à la prémisse d'une règle se fait par l'algorithme *AjoutPrémisse* défini ci-dessous. Cet ajout se fait en queue de la liste (contrainte du projet).

La liste donnée en entrée est modifiée par l'algorithme et est ensuite retournée ; ceci est dû aux contraintes des listes chaînées dans l'implémentation C et du type abstrait de donnée Règle : si la liste est initialement vide (représenté par NULL et *règle_vide*), alors nous ne pouvons pas muter celle-ci en utilisant *mettre_suivant*. Ceci est géré par le premier **Si**.

Algorithme 2 : AjoutPrémisse

Variables :

- R: La règle à modifier
- R': Une variable temporaire pour traverser la liste
- *symbole*: Le nom de la proposition (symbole) à insérer

Données : R: Règle, *symbole*: Symbole
Résultat : R: Règle
Assertion : R n'a pas encore de conclusion
Classe : $\mathcal{O}(n)$

Début AjoutPrémisse(R, *symbole*)
  Si R = *règle_vide* **alors**
    $R \longleftarrow$ *nouvelle_règle*(*symbole*, *règle_vide*)
  Sinon
    $R' \longleftarrow$ R
    `// Répeter jusqu'à ce qu'on atteigne le dernier élément`
    Tant que *suivant*(R') $\neq$ *règle_vide* **faire**
      $R' \longleftarrow$ *suivant*(R')
    Fin
    `//` R' `contient désormais le dernier élément de la liste`
    mettre_suivant(R', *nouvelle_règle*(*symbole*, *règle_vide*))
  FinSi
Fin

1.3 Créer la conclusion d'une règle

Créer la conclusion d'une règle revient à ajouter une proposition (symbole) à la fin de la règle. Pour ce faire, nous ré-utilisons l'algorithme AjoutPrémisse défini plus tôt.

Algorithme 3 : AjoutConclusion

Variables :

- R: La règle à modifier
- *symbole*: Le nom de la proposition (symbole) à insérer comme conclusion

Données : R: Règle, *symbole*: Symbole
Résultat : R: Règle
Assertion : R n'a pas encore de conclusion
Classe : $\mathcal{O}(n)$

Début AjoutConclusion(R, *symbole*)
  $R \longleftarrow$ AjoutPrémisse(R, *symbole*)
Fin

1.4 Tester si une proposition appartient à la prémisse d'une règle

Nous testons si une proposition appartient à la prémisse d'une règle en traversant celle-ci de manière récursive (contrainte du projet).

Les 3 cas minimaux sont :

$R = []$ (règle vide) : retourner « Faux »
$R =$ **{symbole : "...", suivant : *règle_vide*}** (conclusion) : retourner « Faux »
$R =$ **{symbole : symbole_recherché, suivant : ...}** (symbole trouvé) : retourner « Vrai »

Dans les autres cas, nous retournons de manière récursive le résultat de la même fonction, appelée sur $suivant(R)$.

Algorithme 4 : TestAppartenance

Variables :

- R: La règle à étudier
- *symbole*: La nom de la proposition à rechercher
- *résultat*: Si oui ou non la proposition à été trouvée

Données : R: Règle, *symbole*: Symbole
Résultat : *résultat*: Booléen
Assertion : R a une conclusion
Classe : $\mathcal{O}(n)$

Début TestAppartenance(R, *symbole*)
    Si R = *règle_vide* **alors**
        `// Règle vide`
        résultat ⟵ Faux
    Sinon si *suivant(R)* = *règle_vide* **alors**
        `// Conclusion`
        résultat ⟵ Faux
    Sinon si *symbole(R)* = *symbole* **alors**
        `// Symbole trouvé`
        résultat ⟵ Vrai
    Sinon
        résultat ⟵ TestAppartenance(*suivant*(R), *symbole*)
    FinSi
Fin

1.5 Supprimer une proposition de la prémisse d'une règle

Nous supprimons une proposition de la prémisse d'une règle de manière récursive. Cette décision est motivée par le format de Règle et sa simplicité d'implémentation. Les cas minimaux sont les suivants :

$R =$ **[]** (règle vide) : retourner *règle_vide*

$R =$ **{symbole : "...", suivant : *règle_vide*}** (conclusion) : retourner R

Dans le cas général, nous attribuons à $suivant(R)$ la valeur retournée par cette fonction, appelée sur $suivant(R)$, puis nous retournons soit $suivant(R)$ si le noeud correspond au symbole, soit R sinon.

Algorithme 5 : SupprimerSymbole

Variables :

- R: La règle à modifier
- *symbole*: La nom de la proposition à rechercher
- R': La règle privée de *symbole* dans sa prémisse

Données : R: Règle, *symbole*: Symbole
Résultat : R': Règle
Assertion : R a une conclusion
Classe : $\mathcal{O}(n)$

Début SupprimerSymbole(R, *symbole*)
    Si R = *règle_vide* **alors**
        `// Règle vide`
        $R' \longleftarrow$ *règle_vide*
    Sinon si *suivant(R)* = *règle_vide* **alors**
        `// Conclusion`
        $R' \longleftarrow R$
    Sinon
        Si *symbole(R)* = *symbole* **alors**
            `// Retourner le reste de la liste, sans ce noeud`
            $R' \longleftarrow$ SupprimerSymbole(*suivant*(R), *symbole*)
        Sinon
            $R' \longleftarrow$ *mettre_suivant*(R, SupprimerSymbole(*suivant*(R), *symbole*))
        FinSi
    FinSi
Fin

1.6 Tester si la prémisse d'une règle est vide

Voici la fonction retournant « Vrai » si la prémisse d'une règle est vide et « Faux » si la prémisse d'une règle contient au moins 1 symbole :

Algorithme 6 : PrémisseVide

Variables :

- R: La règle à étudier
- *résultat*: Si oui ou non la prémisse d'une règle est vide

Données : R: Règle
Résultat : *résultat*: Booléen
Assertion : R a une conclusion ou R est vide
Classe : $\mathcal{O}(1)$

```
Début PrémisseVide(R)
    Si R = règle_vide alors
        // La règle est vide, donc sa prémisse est vide
        résultat ⟵ Vrai
    Sinon si suivant(R) = règle_vide alors
        // La règle n'a qu'une conclusion, donc sa prémisse est vide
        résultat ⟵ Vrai
    Sinon
        résultat ⟵ Faux
    FinSi
Fin
```

1.7 Accéder à la proposition se trouvant en tête d'une prémisse

Voici la fonction retournant la valeur de la proposition se trouvant en tête d'une prémisse. Si la prémisse est vide, alors la fonction retourne *symbole_vide* ($\emptyset$).

Algorithme 7 : TêteRègle

Variables :

— R: La règle à étudier

— *résultat*: La valeur du premier symbole de la prémisse, si existant

Données : R: Règle
Résultat : *résultat*: Symbole
Assertion : R a une conclusion
Classe : $\mathcal{O}(1)$

```
Début TêteRègle(R)
    Si PrémisseVide(R) alors
        // La prémisse est vide: nous retournons symbole_vide
        résultat ⟵ symbole_vide
    Sinon
        résultat ⟵ symbole(R)
    FinSi
Fin
```

1.8 Accéder à la conclusion d'une règle

La conclusion se trouvant à la fin d'une règle, nous traversons simplement la liste jusqu'au dernier élément de celle-ci et retournons sa valeur. Si la liste est vide, alors la fonction retourne *règle_ vide*.

Algorithme 8 : ConclusionRègle

Variables :

- R: La règle à étudier
- R': Variable temporaire pour traverses la liste
- *résultat*: La valeur du premier symbole de la prémisse, si existant

Données : R: Règle
Résultat : *résultat*: Symbole
Assertion : R a une conclusion
Classe : $\mathcal{O}(n)$

```
Début ConclusionRègle(R)
    Si suivant(R) = règle_ vide alors
        résultat ⟵ règle_ vide
    Sinon
        R' ⟵ R
        // Avançer jusqu'à la fin de la liste
        Tant que suivant(R') ≠ règle_ vide faire
            R' ⟵ suivant(R')
        Fin
        résultat ⟵ symbole(R')
    FinSi
Fin
```

2 Base de Connaissance

Soit **BC** le type représentant une base de connaissance (liste de règle) ; celle-ci prend la forme d'une liste de Règles :

Structure 2 : BC

Nom	**Type**	**Description**
règle	BC → Règle	Retourne une référence à la règle correspondant à ce noeud.
suivant	BC → BC	Une référence au prochain élément de la liste, NULL si l'élément est le dernier de la liste.
nouvelle_ base	(Règle × BC) → BC	Crée une nouvelle base à partir d'une référence vers une Règle et d'une référence vers BC
base_ vide	BC	La base vide

Les axiomes sur ces fonctions sont :

— *règle*(*nouvelle_ base*(r, b)) = r
— *suivant*(*nouvelle_ base*(r, b)) = b

Pour les même raisons que celles de Règle, le type abstrait BC est grandissable par la tête et son implémentation proposée se fait sous la forme d'une liste chaînée.

2.1 Créer une base vide

Nous représentons une base vide par *base_ vide*. Voici l'algorithme permettant de créer une base vide :

Algorithme 9 : BaseVide

Variables : B: La base vide à retourner
Résultat : B: BC
Classe : $\mathcal{O}(1)$
Début BaseVide()
  $B \longleftarrow$ *base_ vide*
Fin

2.2 Ajouter une règle à une base de connaissance

L'ajout de règle à la base de connaissance se fait en tête (pour sa simplicité d'implémentation). Voici son algorithme :

Algorithme 10 : AjoutRègle

Variables :
— B: La base de connaissance à modifier
— *règle*: La valeur de *règle* à mettre dans le noeud
— B': La base de connaissance contenant la nouvelle règle

Données : B: BC ; *règle*: Règle
Résultat : B': BC
Classe : $\mathcal{O}(1)$
Début AjoutRègle(B, *règle*)
  $B' \longleftarrow$ *nouvelle_ base*(*règle*, B)
Fin

2.3 Accéder à la règle se trouvant en tête de la base

Voici l'algorithme permettant d'accéder à la règle se trouvant en tête ; s'il n'y a pas de règle en tête, nous retournons *règle_vide*.

Algorithme 11 : TêteBase

Variables :

— B: La base de connaissance à modifier

— R: La règle se trouvant en tête de la base, *règle_vide* si la base est vide

Données : B: BC
Résultat : R: Règle
Classe : $\mathcal{O}(1)$

```
Début TêteBase(B)
    Si B = base_vide alors
        R ⟵ règle_vide
    Sinon
        R ⟵ règle(B)
    FinSi
Fin
```

3 Moteur d'inférence

Le moteur d'inférence est un algorithme permettant de déduire à partir de la liste initiale des symboles (propositions) ayant la valeur « Vrai » et de la base de connaissance (BC) la liste de tous les symboles étant vrais.

L'algorithme éxecute un maximum de n (la longueur de la base de connaissance) passages sur les règles, celles-cis prenant la forme suivante ($p \geq 0$ symboles S_x dans la prémisse et le symbole Ω_k dans la conclusion) :

$$BC \vdash (\bigwedge_{x=0}^{x<p} S_x) \Rightarrow \Omega_k$$

$$\text{Avec :} \quad (\bigwedge_{x=0}^{x<p} S_x) = \begin{cases} S_0 \wedge S_1 \wedge ... \wedge S_{p-1} & \text{Si } p > 0 \\ Vrai & \text{Si } p = 0 \end{cases}$$

Si Ω_k n'appartient pas encore à la liste des symboles vrais et que $\forall x \in [0, p[, S_x = Vrai$ ou $p = 0$, alors on ajoute Ω_k à la liste des symboles vrais.

Nous notons par la suite n pour la longueur de la base de connaissance, p pour la longueur maximale des règles de la base de connaissance et q pour le nombre de symboles différents faisant partie de la base de connaissance ou étant initiallement présents dans

S. Nous assumons que les symboles ont une longueur maximale fixée au préalable, faisant de leur comparaison une fonction de classe $\mathcal{O}(1)$.

Algorithme 12 : MoteurInférence

Variables :

- B: La base de connaissance, de longueur n et dont les règles ont pour longueur maximale p.
- B': Variable temporaire pour traverser la base de connaissance
- S: La liste des symboles initialements vrais
- S': La liste des symboles initialement vrais ainsi que ceux déduits des règles d'induction
- *ajouté*: Si oui ou non un symbole à été rajouté à S' dans le dernier passage

Données : B: BC ; S: Liste(Symbole)
Résultat : S': Liste(Symbole)
Assertion : $\forall R \in B, R \neq règle\_vide$
Classe : $\mathcal{O}(n^2 \cdot q \cdot p)$

```
Début MoteurInférence(B, S)
    S' ⟵ S
    // Boucle principale: Répétée jusqu'à ce qu'il n'y aie plus de
       modification à S' possibles
    Faire
        ajouté ⟵ Faux
        B' ⟵ B
        // Pour toutes les règles de la base de connaissance...
        Tant que B' ≠ base_vide faire
            // Si la conclusion n'appartient pas à S'...
            Si non(ListeContient(S', ConclusionRègle(TêteBase(B')))) alors
                // Si la prémisse est vraie, alors on ajoute la
                   conclusion à S'
                Si PrémisseVraie(TêteBase(B'), S') alors
                    S' ⟵ insérer_tête(S', ConclusionRègle(TêteBase(B')))
                    ajouté ⟵ Vrai
                FinSi
            FinSi
            B' ⟵ suivant(B')
        Fin
    Tant que ajouté
Fin
```

Algorithme 13 : ListeContient

Variables :

— T: Un type d'objets comparables

— L: Une liste d'objets de type T

— E: La valeur à trouver dans L

— *résultat*: Si oui ou non E est trouvé dans L

Données : L: Liste(T) ; E: T
Résultat : *résultat*: Booléen
Classe : $\mathcal{O}(longueur(L))$

```
Début ListeContient(L, E)
    Si est_vide(L) alors
        résultat ⟵ Faux
    Sinon si tête(L) = E alors
        résultat ⟵ Vrai
    Sinon
        résultat ⟵ ListeContient(reste(L), E)
    FinSi
Fin
```

Algorithme 14 : PrémisseVraie

Variables :

— S: La liste de symboles vrais, de longueur maximale q

— R: La règle à vérifier, de longueur maximale p

— R': Variable utilisée pour traverser R

— *résultat*: Si oui ou non la prémisse de la règle est vraie

Données : R: Règle ; S: Liste(Symbole)
Résultat : *résultat*: Booléen
Assertion : $R \neq règle\_vide$
Classe : $\mathcal{O}(p \cdot q)$

```
Début PrémisseVraie(R, S)
    résultat ⟵ Vrai
    Si non(PrémisseVide(R)) alors
        R' ⟵ R
        Tant que suivant(R') ≠ règle_vide et résultat faire
            Si non(ListeContient(S, TêteRègle(R'))) alors
                résultat ⟵ Faux
            FinSi
            R' ⟵ suivant(R')
        Fin
    FinSi
Fin
```

4 Jeux d'essais

Voici quelques jeux d'essais pour tester les capacités du moteur d'inférence. Ceux-cis peuvent être retrouvés dans le fichier **src/test.c** (utilisant les fonctions décrites jusqu'ici).

Pour simplifier la lecture de ceux-cis, nous omettrons les algorithmes permettant de générer la base de connaissance et fournirons à la place une représentation visuelle de celle-ci.

4.1 Test basique

Ce premier essai est un essai très simple, visant à vérifier le bon fonctionnement des fonctions PrémisseVraie et MoteurInférence. Le symbole C ne doit être ajouté que si A et B sont vrais ; dans les autres cas, la liste d'entrée reste intouchée.

Début BC Test 1
  $A \wedge B \Rightarrow C$
Fin

Test 1 :	
Symboles d'entrée	**Symboles de sortie**
$\emptyset$	$\emptyset$
A	A
B	B
A, B	$A, B,$ **C**

4.2 Somme égale à 2

Ce test ajoute le symbole D à la liste de sortie si 2 symboles parmis A, B et C sont présents (A, B ou A, C ou B, C). Ce test vise à vérifier le bon fonctionnement du moteur d'inférence : plusieures règles peuvent avoir la même conclusion, permettant de simuler l'opérateur $\vee$.

Début BC Test 2
  $A \wedge B \Rightarrow D$
  $A \wedge C \Rightarrow D$
  $B \wedge C \Rightarrow D$
Fin

Ces trois règles peuvent être réduites à $(A \wedge B) \vee (A \wedge C) \vee (B \wedge C) \Rightarrow D$ ($\vee$ correspond à « ou »).

Test 2 :	
Symboles d'entrée	**Symboles de sortie**
A, B	$A, B, \mathbf{D}$
A, C	$A, C, \mathbf{D}$
B, C	$B, C, \mathbf{D}$
A, B, C	$A, B, C, \mathbf{D}$

4.3 Distributivité du « et »

Ce test tente de prouver que $(A \vee B) \wedge C \Rightarrow (A \wedge C) \vee B$. Pour ce faire, les deux moitiés de l'implication sont séparés en deux symboles « cibles », D et E :

Début BC
    $(A \vee B) \wedge C \Rightarrow D$
    $(A \wedge C) \vee B \Rightarrow E$
Fin

Nous devons ensuite réecrire ces deux règles sous le format accepté par le moteur d'inférence ; nous faisons la substitution $tmp \equiv (A \vee B)$ et distribuons les $\vee$ sur les $\Rightarrow$:

Début BC Test 3
    `// Partie gauche`
    $A \Rightarrow tmp$
    $B \Rightarrow tmp$
    $tmp \wedge C \Rightarrow D$
    `// Partie droite`
    $A \wedge C \Rightarrow E$
    $B \Rightarrow E$
Fin

Il nous suffit ensuite de vérifier que $D \rightarrow E$ pour les 8 différentes combinaisons de A, B et C :

Test 3 :	
Symboles d'entrée	**Symboles de sortie**
$\emptyset$	$\emptyset$
A	$A, \textbf{tmp}$
B	$B, \textbf{tmp}, \textbf{E}$
A, B	$A, B, \textbf{tmp}, \textbf{E}$
C	C
A, C	$A, C, \textbf{tmp}, \textbf{D}, \textbf{E}$
B, C	$B, C, \textbf{tmp}, \textbf{D}, \textbf{E}$
A, B, C	$A, B, C, \textbf{tmp}, \textbf{D}, \textbf{E}$

Nous pouvons observer dans les trois derniers cas que si D est présent, alors E est présent. D n'étant pas présents dans les 5 autres cas, notre théorème est vérifié.

5 Extension du moteur d'inférence

Comme nous l'avons vu précédemment, nous pouvons facilement introduire le symbole $\vee$ (« ou ») et éxecuter différentes transformations pour arriver à un ensemble de règles ne contenant que les opérateurs $\wedge$ et $\Rightarrow$.

La version de ce projet publiée en ligne contient une telle extension, introduisant un langage dédié à celle-ci.

Cette extension introduit aussi l'opérateur $\neg$ (« non ») et l'opérateur $\Leftrightarrow$ ('og ssi »). Les opérations de simplification sont les suivantes :

— $\neg(A \wedge B) \ \mapsto\ \neg A \vee \neg B\ \ [A, B]$
— $\neg(A \vee B) \ \mapsto\ \neg A \wedge \neg B\ \ [A, B]$
— $A \vee B \Rightarrow C \ \mapsto\ (A \Rightarrow C), (B \Rightarrow C)\ \ [A, B, C]$
— $A \Rightarrow C \wedge D \ \mapsto\ (A \Rightarrow C), (A \Rightarrow D)\ \ [A, C, D]$
— $\neg\neg A \ \mapsto\ A$
— $A \Leftrightarrow B \ \mapsto\ (A \Rightarrow B), (\neg A \Rightarrow \neg B)$

Une fois ces simplifications appliquées, nous obtenons un ensemble de règle que le moteur d'inférence peut comprendre.

L'implémentation de l'extension lit et analyse un fichier écrit dans le langage dédié et le transforme par distributivité du $\wedge$ en un ensemble de règles du format :

$$BC* \vdash (\bigvee_{x=0}^{x<p} \bigwedge_{y=0}^{y<p'_x} S_{x,y}) \Rightarrow \bigwedge_{x=0}^{y<q} T_x$$

$$\text{Avec (1) :} \quad \forall x (\bigwedge_{y=0}^{y<p'_x} S_{x,y}) = S_{x,0} \wedge S_{x,1} \wedge ... \wedge S_{x,p1}$$

$$\text{Avec (2) :} \quad (\bigvee_{x=0}^{x<p} S'_x) = \begin{cases} S'_0 \vee S'_1 \vee ... \vee S'_{p-1} & \text{Si } p > 0 \\ Vrai & \text{Si } p = 0 \end{cases}$$

$$\text{Avec (3) :} \quad \forall (x,y), S_{x,y} = \begin{cases} S \in \text{Symbole} & \text{Si le symbole n'est pas inversé} \\ \neg S \in \neg\text{Symbole} & \text{Si le symbole est inversé} \end{cases}$$

$$\text{Avec (4) :} \quad \forall x, p'_x > 0$$

$$\text{Avec (5) :} \quad \bigwedge_{x=0}^{y<q} T_x = T_0 \wedge T_1 \wedge ... \wedge T_q 1$$

$$\text{Avec (6) :} \quad \forall x, T_x = \begin{cases} S \in \text{Symbole} & \text{Si le symbole n'est pas inversé} \\ \neg S \in \neg\text{Symbole} & \text{Si le symbole est inversé} \end{cases}$$

$$\text{Avec (7) :} \quad q > 0$$

Les simplifications sont ensuite appliquées pour arriver à un ensemble de règles du format accepté par le moteur d'inférence, avec pour différence que l'ensemble des symboles est étendu à $Symbole* = Symbole \cup \neg Symbole$. Lors de cette simplification, le programme est également capable de transformer la majorité des règles en leurs règles opposées.

Le programme attend enfin que l'utilisateur entre une ou plusieures commandes, celles-ci éxecutant le moteur d'inférence avec différents symboles d'entrée.

5.1 Exemple : distributivité du « et »

Similairement au Test 3, nous pouvons exprimer la relation $(A \vee B) \wedge C \Rightarrow (A \wedge C) \vee B$ dans cette extension :

Début BC
$(A \vee B) \wedge C \Leftrightarrow D$
$(A \wedge C) \vee B \Leftrightarrow E$
$(D \wedge E) \vee \neg D \Leftrightarrow \textit{résultat}$
Fin

Nous éxécutons après simplification le moteur d'inférence avec pour entrée les permutations de $A/\neg A$, $B/\neg B$ et $C/\neg C$:

Test 3 :	
Symboles d'entrée	**Symboles de sortie**
$\neg A, \neg B, \neg C$	$\neg A, \neg B, \neg C, \neg \mathbf{D}, \neg \mathbf{E}, \textbf{résultat}$
$A, \neg B, \neg C$	$A, \neg B, \neg C, \neg \mathbf{D}, \neg \mathbf{E}, \textbf{résultat}$
$\neg A, B, \neg C$	$\neg A, B, \neg C, \neg \mathbf{D}, \mathbf{E}, \textbf{résultat}$
$A, B, \neg C$	$A, B, \neg C, \neg \mathbf{D}, \mathbf{E}, \textbf{résultat}$
$\neg A, \neg B, C$	$\neg A, \neg B, C, \neg \mathbf{D}, \neg \mathbf{E}, \textbf{résultat}$
$A, \neg B, C$	$A, \neg B, C, \mathbf{D}, \mathbf{E}, \textbf{résultat}$
$\neg A, B, C$	$\neg A, B, C, \mathbf{D}, \mathbf{E}, \textbf{résultat}$
A, B, C	$A, B, C, \mathbf{D}, \mathbf{E}, \textbf{résultat}$

résultat étant vrai dans tous les cas, le théorème $(A \vee B) \wedge C \Rightarrow (A \wedge C) \vee B$ est validé.

6 Conclusion

Nous avons, pour ce projet, décrit le fonctionnement d'un système expert simple sous forme d'algorithme et implémenté celui-ci en langage C. Nous avons aussi pu étendre ce système expert à l'ensemble de l'algèbre booléen et le munir d'un langage dédié et d'une interface interactive.

L'implémentation de base ainsi que son extension ont étés capables de prouver des théorèmes simples dans l'algèbre booléen.