

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 :

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

Où A, B, \dots 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'abord 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
<i>symbole</i>	$\text{Règle} \rightarrow \text{Symbole}$	Retourne le nom du symbole correspondant au noeud en tête de liste.
<i>suivant</i>	$\text{Règle} \rightarrow \text{Règle}$	Retourne une référence au prochain élément de la liste, <i>règle_vide</i> si l'élément est le dernier de la liste.
<i>nouvelle_règle</i>	$((\text{Symbole}) \times \text{Règle}) \rightarrow \text{Règle}$	Compose une nouvelle règle à partir du nom d'un symbole et une référence à la prochaine règle.
<i>règle_vide</i>	Règle	La règle vide.
<i>mettre_suivant</i>	$(\text{Règle} \times \text{Règle}) \rightarrow \text{Règle}$	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 :

- $\text{symbole}(\text{nouvelle_règle}(s, r)) = s$
- $\text{suivant}(\text{nouvelle_règle}(s, r)) = r$
- $A \longleftarrow \text{nouvelle_règle}(s, r) ; \text{mettre_suivant}(a, r') \Rightarrow \text{suivant}(A) = r'$

Nous pouvons noter que :

Liste \prec Règle [$\emptyset : \text{règle_vide}$, tête : *symbole*, reste : *suivant*, insérer_tête : *nouvelle_règle*, Élément : 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 :	<i>R</i> : La règle vide à retourner
Résultat :	<i>R</i> : Règle
Classe :	$\mathcal{O}(1)$
1 Début	RègleVide()
2	$R \leftarrow \text{règle_vide}$
3 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 :** $\mathcal{O}(n)$

```

1 Début AjoutPrémisse( $R$ ,  $symbole$ )
2   Si  $R = règle\_vide$  alors
3     |  $R \leftarrow nouvelle\_règle(symbole, règle\_vide)$ 
4   Sinon
5     |  $R' \leftarrow R$ 
6     | // Répéter jusqu'à ce qu'on atteigne le dernier élément
7     | Tant que  $suivant(R') \neq règle\_vide$  faire
8     |   |  $R' \leftarrow suivant(R')$ 
9     | Fin
10    | //  $R'$  contient désormais le dernier élément de la liste
11    |  $mettre\_suivant(R', nouvelle\_règle(symbole, règle\_vide))$ 
12  FinSi
13 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 :** $\mathcal{O}(n)$

```

1 Début AjoutConclusion( $R$ ,  $symbole$ )
2   |  $R \leftarrow AjoutPrémisse(R, symbole)$ 
3 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 »

$R = \{\text{symbole} : "...", \text{suivant} : \text{règle_vide}\}$ (conclusion) : retourner « Faux »

$R = \{\text{symbole} : \text{symbole_recherché}, \text{suivant} : ...\}$ (symbole trouvé) : retourner « Vrai »

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

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 : $\mathcal{O}(n)$

```

1 Début TestAppartenance( $R, \text{symbole}$ )
2   Si  $R = \text{règle\_vide}$  alors
3     // Règle vide
4      $\text{résultat} \leftarrow \text{Faux}$ 
5   Sinon si  $\text{suivant}(R) = \text{règle\_vide}$  alors
6     // Conclusion
7      $\text{résultat} \leftarrow \text{Faux}$ 
8   Sinon si  $\text{symbole}(R) = \text{symbole}$  alors
9     // Symbole trouvé
10     $\text{résultat} \leftarrow \text{Vrai}$ 
11  Sinon
12     $\text{résultat} \leftarrow \text{TestAppartenance}(\text{suivant}(R), \text{symbole})$ 
13  FinSi
14 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*

$R = \{\text{symbole} : "...", \text{suivant} : \text{règle_vide}\}$ (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 : $\mathcal{O}(n)$

```

1 Début SupprimerSymbole( $R$ , symbole)
2   Si  $R = \text{règle\_vide}$  alors
3     // Règle vide
4      $R' \leftarrow \text{règle\_vide}$ 
5   Sinon si suivant( $R$ ) = règle_vide alors
6     // Conclusion
7      $R' \leftarrow R$ 
8   Sinon
9     Si symbole( $R$ ) = symbole alors
10      // Retourner le reste de la liste, sans ce noeud
11       $R' \leftarrow \text{SupprimerSymbole}(\text{suivant}(R), \text{symbole})$ 
12    Sinon
13       $R' \leftarrow \text{mettre\_suivant}(R, \text{SupprimerSymbole}(\text{suivant}(R), \text{symbole}))$ 
14  FinSi
15 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 : $\mathcal{O}(1)$

```
1 Début PrémisseVide( $R$ )
2   Si  $R = règle\_vide$  alors
3     // La règle est vide, donc sa prémisse est vide
4      $résultat \leftarrow Vrai$ 
5   Sinon si  $suivant(R) = règle\_vide$  alors
6     // La règle n'a qu'une conclusion, donc sa prémisse est vide
7      $résultat \leftarrow Vrai$ 
8   Sinon
9      $résultat \leftarrow Faux$ 
10  FinSi
11 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 : $\mathcal{O}(1)$

```
1 Début TêteRègle( $R$ )
2   Si PrémisseVide( $R$ ) alors
3     // La prémisse est vide: nous retournons symbole_vide
4     résultat  $\leftarrow$  symbole_vide
5   Sinon
6     résultat  $\leftarrow$  symbole( $R$ )
7   FinSi
7 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 traverser 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 : $\mathcal{O}(n)$

```

1 Début ConclusionRègle( $R$ )
2   Si suivant( $R$ ) = règle_vide alors
3     | résultat  $\leftarrow$  règle_vide
4   Sinon
5     |  $R' \leftarrow R$ 
6       // Avancer jusqu'à la fin de la liste
7     Tant que suivant( $R'$ )  $\neq$  règle_vide faire
8       |  $R' \leftarrow$  suivant( $R'$ )
9     Fin
10    | résultat  $\leftarrow$  symbole( $R'$ )
11  FinSi
12 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
<i>règle</i>	$BC \rightarrow \text{Règle}$	Retourne une référence à la règle correspondant à ce noeud.
<i>suivant</i>	$BC \rightarrow BC$	Une référence au prochain élément de la liste, NULL si l'élément est le dernier de la liste.
<i>nouvelle_base</i>	$(\text{Règle} \times BC) \rightarrow BC$	Crée une nouvelle base à partir d'une référence vers une Règle et d'une référence vers BC
<i>base_vide</i>	BC	La base vide

Les axiomes sur ces fonctions sont :

- $r\grave{e}gle(nouvelle_base(r, b)) = r$
- $suivant(nouvelle_base(r, b)) = b$

Pour les m\^eme raisons que celles de R\^egle, le type abstrait BC est grandissable par la t\^ete et son impl\^ementation propos\^ee se fait sous la forme d'une liste cha\^een\^ee.

2.1 Cr\^eer une base vide

Nous repr\^esentons une base vide par *base_vide*. Voici l'algorithme permettant de cr\^eer une base vide :

Algorithme 9 : BaseVide

Variables : B : La base vide \`a retourner
R\^esultat : B : BC
Classe : $\mathcal{O}(1)$
1 D\^ebut BaseVide()
2 | $B \leftarrow base_vide$
3 Fin

2.2 Ajouter une r\^egle \`a une base de connaissance

L'ajout de r\^egle \`a la base de connaissance se fait en t\^ete (pour sa simplicit\^e d'impl\^ementation). Voici son algorithme :

Algorithme 10 : AjoutR\^egle

Variables :
 — B : La base de connaissance \`a modifier
 — $r\grave{e}gle$: La valeur de $r\grave{e}gle$ \`a mettre dans le noeud
 — B' : La base de connaissance contenant la nouvelle r\^egle
Donn\^ees : B : BC ; $r\grave{e}gle$: R\^egle
R\^esultat : B' : BC
Classe : $\mathcal{O}(1)$
1 D\^ebut AjoutR\^egle($B, r\grave{e}gle$)
2 | $B' \leftarrow nouvelle_base(r\grave{e}gle, B)$
3 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 : $\mathcal{O}(1)$

```

1 Début TêteBase( $B$ )
2   Si  $B = \text{base\_vide}$  alors
3      $R \leftarrow \text{règle\_vide}$ 
4   Sinon
5      $R \leftarrow \text{règle}(B)$ 
6   FinSi
7 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 exécute un maximum de n (la longueur de la base de connaissance) passages sur les règles, celles-ci prenant la forme suivante ($p \geq 0$ symboles S_x dans la prémisse et le symbole Ω_k dans la conclusion) :

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

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

Si Ω_k n'appartient pas encore à la liste des symboles vrais et que $\forall x \in [0, p[, S_x = \text{Vrai}$ 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 initialement 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 initialement 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 :** $\forall R \in B, R \neq \text{règle_vide}$ **Classe :** $\mathcal{O}(n^2 \cdot q \cdot p)$

```

1  Début MoteurInférence( $B, S$ )
2  |  $S' \leftarrow S$ 
   | // Boucle principale: Répétée jusqu'à ce qu'il n'y aie plus de
   | // modification à  $S'$  possibles
3  | Faire
4  | |  $\text{ajouté} \leftarrow \text{Faux}$ 
5  | |  $B' \leftarrow B$ 
   | | // Pour toutes les règles de la base de connaissance...
6  | | Tant que  $B' \neq \text{base\_vide}$  faire
   | | | // Si la conclusion n'appartient pas à  $S'$ ...
7  | | | Si non( $\text{ListeContient}(S', \text{ConclusionRègle}(\text{TêteBase}(B'))))$  alors
   | | | | // Si la prémisse est vraie, alors on ajoute la
   | | | | conclusion à  $S'$ 
8  | | | | Si  $\text{PrémisseVraie}(\text{TêteBase}(B'), S')$  alors
9  | | | | |  $S' \leftarrow \text{insérer\_tête}(S', \text{ConclusionRègle}(\text{TêteBase}(B')))$ 
10 | | | | |  $\text{ajouté} \leftarrow \text{Vrai}$ 
11 | | | FinSi
12 | | FinSi
13 | |  $B' \leftarrow \text{suivant}(B')$ 
14 | Fin
15 | Tant que  $\text{ajouté}$ 
16 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 :** $\mathcal{O}(\text{longueur}(L))$

```

1 Début ListeContient( $L, E$ )
2   Si  $est\_vide(L)$  alors
3     |  $résultat \leftarrow$  Faux
4   Sinon si  $tête(L) = E$  alors
5     |  $résultat \leftarrow$  Vrai
6   Sinon
7     |  $résultat \leftarrow$  ListeContient( $reste(L), E$ )
8   FinSi
9 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 :** $R \neq \text{règle_vide}$ **Classe :** $\mathcal{O}(p \cdot q)$

```

1 Début PrémisseVraie( $R, S$ )
2    $résultat \leftarrow$  Vrai
3   Si non( $PrémisseVide(R)$ ) alors
4     |  $R' \leftarrow R$ 
5     | Tant que  $suivant(R') \neq \text{règle\_vide}$  et  $résultat$  faire
6     |   Si non( $ListeContient(S, TêteRègle(R'))$ ) alors
7     |     |  $résultat \leftarrow$  Faux
8     |   FinSi
9     |  $R' \leftarrow suivant(R')$ 
10  Fin
11 FinSi
12 Fin

```

4 Jeux d'essais

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

Pour simplifier la lecture de ceux-ci, 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.

```
1 Début BC Test 1
2 |    $A \wedge B \Rightarrow C$ 
3 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 parmi 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](#) : plusieurs règles peuvent avoir la même conclusion, permettant de simuler l'opérateur \vee .

```
1 Début BC Test 2
2 |    $A \wedge B \Rightarrow D$ 
3 |    $A \wedge C \Rightarrow D$ 
4 |    $B \wedge C \Rightarrow D$ 
5 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 :

```

1 Début BC
2   |  $(A \vee B) \wedge C \Rightarrow D$ 
3   |  $(A \wedge C) \vee B \Rightarrow E$ 
4 Fin

```

Nous devons ensuite réécrire 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 :

```

1 Début BC Test 3
2   | // Partie gauche
3   |  $A \Rightarrow tmp$ 
4   |  $B \Rightarrow tmp$ 
5   |  $tmp \wedge C \Rightarrow D$ 
6   | // Partie droite
7   |  $A \wedge C \Rightarrow E$ 
8   |  $B \Rightarrow E$ 
9 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, \mathbf{tmp}
B	$B, \mathbf{tmp}, \mathbf{E}$
A, B	$A, B, \mathbf{tmp}, \mathbf{E}$
C	C
A, C	$A, C, \mathbf{tmp}, \mathbf{D}, \mathbf{E}$
B, C	$B, C, \mathbf{tmp}, \mathbf{D}, \mathbf{E}$
A, B, C	$A, B, C, \mathbf{tmp}, \mathbf{D}, \mathbf{E}$

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 exécuter 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 (« ssi »).

Nous pouvons appliquer les opérations de simplification suivantes pour passer du langage utilisant les opérateurs $(\vee, \wedge, \neg, \Rightarrow, \Leftrightarrow)$ au langage utilisant uniquement les opérateurs $(\wedge, \neg, \Rightarrow)$:

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

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

5.1 Implémentation

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 :

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

Tel que :

$$\forall x \left(\bigwedge_{y=0}^{y < p'_x} S_{x,y} \right) = S_{x,0} \wedge S_{x,1} \wedge \dots \wedge S_{x,p1} \quad (1)$$

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

$$\forall (x, y), S_{x,y} = \begin{cases} S \in Symbole & \text{Symbole non-inversé} \\ \neg S \in \neg Symbole & \text{Symbole inversé} \end{cases} \quad (3)$$

$$\bigwedge_{x=0}^{y < q} T_x = T_0 \wedge T_1 \wedge \dots \wedge T_q1 \quad (4)$$

$$\forall x, T_x = \begin{cases} S \in Symbole & \text{Symbole non-inversé} \\ \neg S \in \neg Symbole & \text{Symbole inversé} \end{cases} \quad (5)$$

$$\forall x, p'_x > 0 \quad (6)$$

$$q > 0 \quad (7)$$

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 plusieurs commandes, celles-ci exécutant le moteur d'inférence avec différents symboles d'entrée.

5.2 Exemple : distributivité du « et »

Similairement au [Test 3](#), nous pouvons exprimer la relation $(A \vee B) \wedge C \Rightarrow (A \wedge C) \vee B$ de cette manière, en utilisant cette fois-ci les symboles que l'extension nous propose :

```

1 Début BC Test 4
2   | (A ∨ B) ∧ C ⇔ D
3   | (A ∧ C) ∨ B ⇔ E
4   | (D ∧ E) ∨ ¬D ⇔ résultat
5 Fin

```

Cette base de donnée est encodée dans le langage de l'extension et lui est donnée en entrée. Nous exécutons ensuite 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
$\neg A, \neg B, \neg C$	$\neg A, \neg B, \neg C, \neg D, \neg E, \text{résultat}$
$A, \neg B, \neg C$	$A, \neg B, \neg C, \neg D, \neg E, \text{résultat}$
$\neg A, B, \neg C$	$\neg A, B, \neg C, \neg D, E, \text{résultat}$
$A, B, \neg C$	$A, B, \neg C, \neg D, E, \text{résultat}$
$\neg A, \neg B, C$	$\neg A, \neg B, C, \neg D, \neg E, \text{résultat}$
$A, \neg B, C$	$A, \neg B, C, D, E, \text{résultat}$
$\neg A, B, C$	$\neg A, B, C, D, E, \text{résultat}$
A, B, C	$A, B, C, D, E, \text{résultat}$

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 sont capables de prouver des théorèmes simples dans l'algèbre booléen, comme celui présenté en exemple.

Ce modèle de système expert reste pour autant limité, car il ne supporte que des symboles de type booléen.