LO21

Marius Bozane
Louis Esteban

Automne 2020

Résumé

Pour notre projet de LO21 nous devions réaliser un système expert composé de bases de connaissances, de règles et d'un moteur d'inférence.

Une règle est composé d'une prémisse et d'une conclusion

Les règles sont sous la forme $A \wedge B \wedge C \Rightarrow D$
Avec $A \wedge B \wedge C$ comme prémisse et D comme conclusion.

Une base de connaissances est un ensemble de règles.

Le moteur d'inférence permet de déduire des faits certains en partant de la base de faits et en appliquant les règles de la base de connaissances.

Table des matières

Première partie

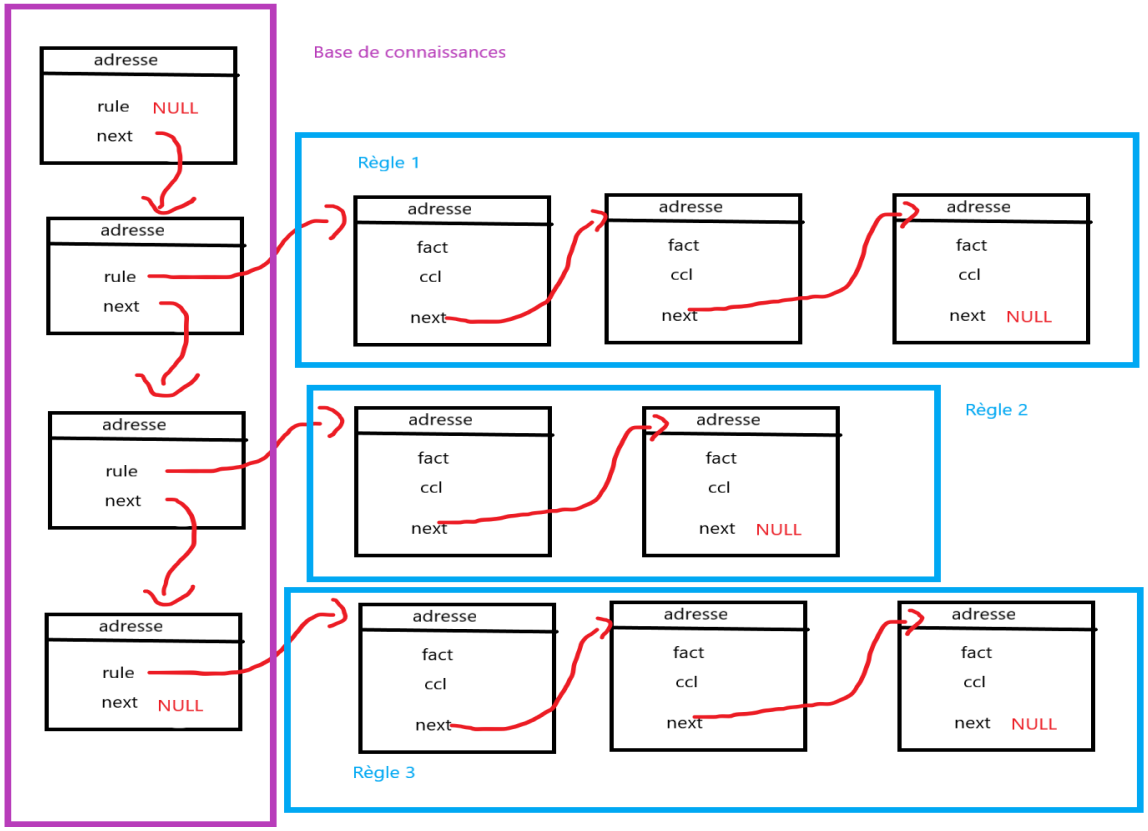
Choix de conception et d'implémentation

1 Choix de conception

Pour réaliser notre projet nous avons décidé de créer une structure pour les règles, une structure pour les bases de connaissances ainsi qu'une structure pour le moteur d'inférence.

Nous avons séparé le projet en 3 fichiers source et 3 fichiers header, chacun correspondant au programme et à la librairie associé de respectivement les règles, les bases de connaissances et le moteur d'inférence. Un fichier main est utilisé pour tester les différentes fonctions.

Les structures des base de faits et des règles se présentent sous la forme ci-dessous :



2 Choix d'implémentation

Pour le type booléen, on a utilisé la valeur 0 pour Faux et 1 pour Vrai.

2.1 structure des règles

```
9   struct rule {
10   char fact[FACT_LEN];
11   int ccl;
12   struct rule* next;
13  };
14
15  typedef struct rule rule_t;
```

2.2 structure des bases de connaissances

```
9    struct kb {
10    rule_t* rule;
11    struct kb* next;
12   };
13   typedef struct kb kb_t;
```

2.3 structure des listes de faits

```
9   #define FACT_LEN 256
10
11   struct facts {
12    char fact[FACT_LEN];
13    struct facts* next;
14  };
15  typedef struct facts facts_t;
```

Deuxième partie

Algorithmes

3 Règles

3.1 Nouvelle Règle

L'algorithme ci-dessous permet de créer une règle vide et de retourner son pointeur.

Algorithme 1 : RègleVide

Variables : R: La nouvelle Règle
Résultat : R: Règle
Début RègleVide()
  $R \longleftarrow$ *règle_vide*
Fin

3.2 Ajout de Prémisses

L'algorithme ci-dessous permet d'ajouter une prémisse à une règle.

Algorithme 2 : AjoutPrémisse

Variables :
- P: Prémisse à rajouter
- R: Règle dont on veut rajouter une prémisse
- R': Stockage du dernier objet d'une règle
- T: Stockage de l'avant-dernier objet de la règle
- NP: L'adresse de la nouvelle prémisse

Données : R: Règle
Résultat : R: Règle
Début AjoutPrémisse (C,R)
  Tant que *Suivant(*R*)* $\neq$ *indéfini* **faire**
    $T \longleftarrow$ Suivant(R)
    $R \longleftarrow$ Suivant(R)
  Fin
  Conclusion(R') $\longleftarrow$ R
  Fait(NP) $\longleftarrow$ R
  Si *Conclusion(*R'*)=1* **alors**
    Suivant(T) $\longleftarrow$ NP
    Suivant(NP) $\longleftarrow$ R'
  FinSi
  Sinon
    Suivant(R) $\longleftarrow$ NP
  FinSi
Fin

3.3 Créer une conclusion

L'algorithme ci-dessous permet d'ajouter une conclusion à une règle.

Algorithme 3 : CréerConclusion

Variables :
- C : Conclusion à rajouter
- R : Règle dont on veut rajouter une conclusion
- R' : Règle de transit

Données : R : Règle
Résultat : R : Règle

```
1  Début CréerConclusion (C,R)
2      Tant que Suivant(R) ≠ indéfini faire
3          R ⟵ Suivant(R)
4      Fin
5      Si Conclusion(R) = 0 alors
6          R' ⟵ RègleVide()
7          Fait(R') ⟵ C
8          Conclusion(R') ⟵ 1
9          Suivant(R) ⟵ R'
10         Retourner Vrai
11     FinSi
12     Sinon
13         Retourner Faux
14     FinSi
15 Fin
```

3.4 Test 1 : Une prémisse appartient à une règle

L'algorithme ci-dessous permet de vérifier si une règle contient une prémisse ou non, il retourne vrai si la prémisse a été trouvée et il retourne faux si cette prémisse n'est pas contenue dans la règle.

Algorithme 4 : TestPrémisse

```
   Variables :
   - R : Règle
   - P : Prémisse que l'on veut tester
   Données : R : Règle
   Résultat : Resultat : Boléen
 1 Début TestPrémisse (P,R)
 2 |   Tant que Suivant(R) ≠ indéfini faire
 3 |   |   Si ComparerCaractère(Fact(R),P)=0 alors
 4 |   |   |   Retourner Vrai
 5 |   |   FinSi
 6 |   |   R ⟵ Suivant(R)
 7 |   Fin
 8 |   Si (Conlusion(R)=0) ( et ComparerCaractère(Fact(R),P)=0) alors
 9 |   |   Retourner Vrai
10 |   FinSi
11 |   Sinon
12 |   |   Retourner Faux
13 |   FinSi
14 Fin
```

3.5 Supprimer une prémisse d'une règle

L'algorithme ci-dessous permet de supprimer une prémisse d'une règle, il retourne Vrai si l'opération a pu être réalisé et retourne faux s'il n'a pas pu trouver la prémisse en question.

Algorithme 5 : SupprimerPrémisse

Variables :
- P: Prémisse à rajouter
- R: Règle dont on veut rajouter une prémisse
- R': Stockage du dernier objet d'une règle
- T: Stockage de l'avant-dernier objet de la règle

Données : R: Règle
Résultat : $Resultat$: Boléen

```
1  Début SupprimerPrémisse (P,R)
2      Si PrémisseVide(R=1 alors
3          Retourner Faux
4      FinSi
5      Sinon
6          Tant que R ≠ indéfini faire
7              T ⟵ R
8              Si (ComparerCaractère(Fait(R),P)=1) ( et Conclusion(R)=1) alors
9                  Suivant(T) ⟵ Suivant(R)
10                 Libérer(R)
11                 Retourner Vrai
12             FinSi
13             R ⟵ Suivant(R)
14         Fin
15         Retourner Faux
16     FinSi
17 Fin
```

3.6 Test 2 : Prémisse vide d'une règle

L'algorithme ci-dessous test si une règle contient des prémisses ou non. Il retourne Vrai si la règle ne contient pas de prémisses (elle peut tout de même contenir une conclusion) et retourne faux si elle contient des prémisses.

Algorithme 6 : PrémisseVide

```
   Variables : R : Règle que l'on veut tester
   Données : R : Règle
   Résultat : Resultat : Boléen
1  Début PrémisseVide (R)
2      Si Suivant(R) ≠ indéfini alors
3          Retourner Faux
4      FinSi
5      Sinon si Conclusion(Suivant(R))=1 alors
6          Retourner Vrai
7      FinSi
8      Sinon
9          Retourner Faux
10     FinSi
11 Fin
```

3.7 Accéder à la première prémisse d'une règle

L'algorithme ci-dessous permet de connaître la première prémisse d'une règle (si celle ci en contient bien sûr). Ce dernier retourne un pointeur lié à la première prémisse de la règle.

Algorithme 7 : PremièrePrémisse

```
   Variables : R : Règle dont on veut voir la prémisse
   Données : R : Règle
   Résultat : P : Prémisse
1  Début PremièrePrémisse (R)
2      Tant que Suivant(R) ≠ indéfini faire
3          R ⟵ Suivant(R)
4      Fin
5      Si Conclusion(R)=1 alors
6          Retourner (Fait(R))
7      FinSi
8      Retourner indéfini
9  Fin
```

3.8 Accéder à la conclusion d'une règle

L'algorithme ci-dessous permet de connaître la conclusion d'une règle (si celle ci existe bien sûr). Ce dernier retourne un pointeur lié à la conclusion de la règle.

Algorithme 8 : VoirConclusion

```
   Variables : R: Règle dont on veut la conclusion
   Données : R: Règle
   Résultat : P: Prémisse
 1 Début VoirConclusion (C,R)
 2 |   Tant que Suivant(R) ≠ indéfini faire
 3 |   |   R ⟵ Suivant(R)
 4 |   Fin
 5 |   Si Conclusion(Rule)=1 alors
 6 |   |   P ⟵ Fait(R)
 7 |   FinSi
 8 |   Sinon
 9 |   |   P ⟵ indéfini
10 |   FinSi
11 Fin
```

4 Base de Connaissances

4.1 Nouvelle Base de Connaissances

L'algorithme ci-dessous permet de créer une base de connaissances vide. Il retourne le pointeur lié à cette base de connaissances.

Algorithme 9 : BC_Vide

Variables : BC: La nouvelle Base de Connaissances
Résultat : BC: Base de Connaissances
1 **Début** BC_Vide()
2 | $BC \longleftarrow$ *BC_vide*
3 **Fin**

4.2 Ajouter une nouvelle règle

L'algorithme ci-dessous permet d'ajouter à une base de connaissances existante une nouvelle règle. Il retourne vrai si l'opération s'est déroulé avec succès.

Algorithme 10 : AjoutRègle

Variables :
- R: Règle à rajouter
- BC: Base dont on veut rajouter une règle
- NR: Nouvelle règle

Données : BC: Base de Connaissances
Résultat : $Resultat$: Boléen
1 **Début** AjoutRègle (R,BC)
2 | $NR \longleftarrow$ *BC_Vide()*
3 | **Tant que** *Suivant*(BC) $\neq$ *indéfini* **faire**
4 | | $BC \longleftarrow$ Suivant(BC)
5 | **Fin**
6 | Suivant(BC) $\longleftarrow NR$
7 | Règle(NR) $\longleftarrow R$
8 | Retourner Vrai
9 **Fin**

4.3 Afficher la première règle

L'algorithme ci-dessous permet d'afficher la première règle d'une base de connaissance. Il retourne le pointeur lié à la première règle de la base (si celle-ci existe).

Algorithme 11 : PremièreRègle

```
   Variables : BC : La base de connaissances en question
   Résultat : R : Règle
1  Début PremièreRègle(BC)
2  |  Si Suivant(BC) = indéfini alors
3  |  |  Retourner indéfini
4  |  FinSi
5  |  Sinon
6  |  |  Retourner Règle(Suivant(BC))
7  |  FinSi
8  Fin
```

5 Moteur d'Inférence

5.1 Fait Vrai

L'algorithme ci-dessous permet de tester si un fait est dans une liste de fait. Il retourne vrai si le fait est contenu dans la liste de fait et faux si il n'est pas contenu dedans.

Algorithme 12 : FaitVrai

```
   Variables :
   - LF : La liste de faits
   - F : Un fait
   Données : LF : Liste de faits
   Résultat : Resultat : Boléen
 1 Début FaitVrai(F,LF)
 2     Tant que Suivant(LF) ≠ indéfini faire
 3         Si ComparerCaractère(Fait(LF),F)=0 alors
 4             Retourner Vrai
 5         FinSi
 6         LF ⟵ Suivant(LF)
 7     Fin
 8     Si ComparerCaractère(Fait(LF),F)=0 alors
 9         Retourner Vrai
10     FinSi
11     Sinon
12         Retourner Faux
13     FinSi
14 Fin
```

5.2 Ajouter un fait à une liste

L'algorithme ci-dessous ajoute un fait à une liste de fait. Il retourne vrai si l'opération s'est déroulée avec succès et faux si l'opération n'a pas pu se faire

Algorithme 13 : AjouterFaitsListe

```
   Variables :
   - LF : La liste de faits
   - F : Un fait
   - NF : Un fait transitoire
   Données : LF : Liste de faits
   Résultat : Resultat : Boléen
 1 Début AjouterFaitsListe(BC)
 2 |   Si FaitVrai(F,LF)=Vrai alors
 3 |   |   Retourner Vrai
 4 |   FinSi
 5 |   Sinon
 6 |   |   CopierCaractère(Fait(NF,F))
 7 |   |   Suivant(NF)=Suivant(LF)
 8 |   |   Suivant(LF)=NF
 9 |   |   Retourner Faux
10 |   FinSi
11 Fin
```

5.3 Règle Vraie

L'algorithme ci-dessous permet de savoir si une règle est vraie. Il retourne indéfini si la règle est fausse et la conclusion si la règle est vraie

Algorithme 14 : RègleVraie

Variables :
- LF : Une liste de faits
- R : Une règle

Données : LF : Une liste de faits R : Une règle
Résultat : $Resultat$: Un fait

```
1  Début RègleVraie(BC)
2      Si (PrémisseVide(R) = Faux) ( et VoirConclusion(R) ≠ indéfini) alors
3          Tant que Suivant(R) ≠ indéfini faire
4              Si FaitVrai(Fait(R),LF)=Faux alors
5                  Retourner indéfini
6              FinSi
7              R ⟵ Suivant(R)
8          Fin
9          Retourner Fait(R)
10     FinSi
11     Sinon
12         Retourner indéfini
13     FinSi
14 Fin
```

5.4 Moteur d'Inférence

L'algorithme ci-dessous retourne une liste chaînée des faits vrais.

Algorithme 15 : MoteurInférence

Variables :
- LF: Liste de Faits
- BC: Base connaissances
- KB: Base de connaissances transitoire
- PF: Liste de faits transitoires
- RF: Liste de faits transitoire

Données : LF: Liste de faits
Résultat : R: Règle

```
1  Début MoteurInférence(BC,LF)
2      Tant que PF ≠ indéfini faire
3          AjouterFaitsListe(PF,F)
4          PF ⟵ Suivant(PF)
5      Fin
6      Tant que KB ≠ indéfini faire
7          Si RègleVraie(Règle(KB),RF)=Vrai alors
8              AjouterFaitsListe(VoirConclusion(Rule(KB),RF)
9          FinSi
10         KB ⟵ Suivant(KB)
11     Fin
12     Retourner RF
13 Fin
```

Troisième partie

Jeux d’essais

6 Jeu d'essais 1

6.1 Postulat

A ∧ B ⇒ C
D ∧ B ∧ E ⇒ G
Base de fait : A, B, D, E
Résultat : A, B, D, E, C, G

6.2 Test

```c
int main(){
  char v[3] = "";
  char a[3] = "A";
  char b[3] = "B";
  char c[3] = "C";
  char d[3] = "D";
  char e[3] = "E";
  char f[3] = "F";
  char g[3] = "G";

  //first rule : (A and B -> C)
  rule_t *regle1;
  regle1 = new_rule();
  add_fact(&a, regle1);
  add_fact(&b, regle1);
  create_ccl(&c, regle1);

  //second rule : (D and B and E -> G)
  rule_t *regle2;
  regle2 = new_rule();
  add_fact(&d, regle2);
  add_fact(&b, regle2);
  add_fact(&e, regle2);
  create_ccl(&g, regle2);

//knowledge base with rules 1 and 2
  kb_t* pkb = new_kb();
  addrule(regle1, pkb);
  addrule(regle2, pkb);

//fact list
  facts_t* factList = malloc(sizeof(facts_t));
  strcpy(factList->fact, &v);
  factList->next = NULL;
  add_fact_to_list(&a, factList);
  add_fact_to_list(&b, factList);
  add_fact_to_list(&d, factList);
  add_fact_to_list(&e, factList);

//computing
  facts_t* result = inference_engine(pkb, factList);

//prompt result
  facts_t *res = result;
  printf("list : ");
  while (res != 0){
    printf("%s \n", res->fact);
    res = res->next;
  }

  return 0;
}
```

```
D:\dev\git\LOP21_Project\LO21_Project\Lo21\bin\Debug\Lo21.exe
list :
G
C
A
B
D
E

Process returned 0 (0x0)   execution time : 0.035 s
Press any key to continue.
```

7 Jeu d'essais 2

7.1 Postulat

A ∧ B ⇒ C
D ∧ C ∧ E ⇒ G
Base de fait : A, B, D, E
Résultat : A, B, D, E, C, G

7.2 Test

```
//first rule : (A and B -> C)
rule_t *regle1;
regle1 = new_rule();
add_fact(&a, regle1);
add_fact(&b, regle1);
create_ccl(&c, regle1);

//second rule : (D and B and E -> G)
rule_t *regle2;
regle2 = new_rule();
add_fact(&d, regle2);
add_fact(&c, regle2);
add_fact(&e, regle2);
create_ccl(&g, regle2);

//knowledge base with rules 1 and 2
kb_t* pkb = new_kb();
addrule(regle1, pkb);
addrule(regle2, pkb);

//fact list
facts_t* factList = malloc(sizeof(facts_t));
strcpy(factList->fact, &v);
factList->next = NULL;
add_fact_to_list(&a, factList);
add_fact_to_list(&b, factList);
add_fact_to_list(&d, factList);
add_fact_to_list(&e, factList);
```

```
D:\dev\git\LOP21_Project\LO21_Project\Lo21\bin\Debug\Lo21.exe
list :
G
C
A
B
D
E

Process returned 0 (0x0)   execution time : 0.017 s
Press any key to continue.
```

8 Jeu d'essais 3

8.1 Postulat

A ∧ B ⇒ C
D ∧ C ⇒ G
A ∧ B ∧ C ∧ D ∧ E ∧ G ⇒ F
Base de fait : A, B, D
Résultat : A, B, D, C, G

8.2 Test

```
//first rule : (A and B -> C)
rule_t *regle1;
regle1 = new_rule();
add_fact(&a, regle1);
add_fact(&b, regle1);
create_ccl(&c, regle1);

//second rule : (D and B and E -> G)
rule_t *regle2;
regle2 = new_rule();
add_fact(&d, regle2);
add_fact(&c, regle2);
create_ccl(&g, regle2);

//first rule : (A and B -> C)
rule_t *regle3;
regle3 = new_rule();
add_fact(&a, regle3);
add_fact(&b, regle3);
add_fact(&e, regle3);
add_fact(&d, regle3);
add_fact(&c, regle3);
add_fact(&g, regle3);
create_ccl(&f, regle3);

//knowledge base with rules 1 and 2
kb_t* pkb = new_kb();
addrule(regle1, pkb);
addrule(regle2, pkb);
addrule(regle3, pkb);

//fact list
facts_t* factList = malloc(sizeof(facts_t));
strcpy(factList->fact, &v);
factList->next = NULL;
add_fact_to_list(&a, factList);
add_fact_to_list(&b, factList);
add_fact_to_list(&d, factList);
//add_fact_to_list(&e, factList);
```

```
D:\dev\git\LOP21_Project\LO21_Project\Lo21\bin\Debug\Lo21.exe
list :
G
C
A
B
D

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

9 Jeu d'essais 4

9.1 Postulat

A ∧ B ⇒ C
D ∧ C ⇒ G
A ∧ B ∧ C ∧ D ∧ E ∧ G ⇒ F
Base de fait : A, B, D, E
Résultat : A, B, D, C, G, E, F

9.2 Test

```
//first rule : (A and B -> C)
rule_t *regle1;
regle1 = new_rule();
add_fact(&a, regle1);
add_fact(&b, regle1);
create_ccl(&c, regle1);

//second rule : (D and B and E -> G)
rule_t *regle2;
regle2 = new_rule();
add_fact(&d, regle2);
add_fact(&c, regle2);
create_ccl(&g, regle2);

//first rule : (A and B -> C)
rule_t *regle3;
regle3 = new_rule();
add_fact(&a, regle3);
add_fact(&b, regle3);
add_fact(&e, regle3);
add_fact(&d, regle3);
add_fact(&c, regle3);
add_fact(&g, regle3);
create_ccl(&f, regle3);

//knowledge base with rules 1 and 2
kb_t* pkb = new_kb();
addrule(regle1, pkb);
addrule(regle2, pkb);
addrule(regle3, pkb);

//fact list
facts_t* factList = malloc(sizeof(facts_t));
strcpy(factList->fact, &v);
factList->next = NULL;
add_fact_to_list(&a, factList);
add_fact_to_list(&b, factList);
add_fact_to_list(&d, factList);
add_fact_to_list(&e, factList);
```

```
D:\dev\git\LOP21_Project\LO21_Project\Lo21\bin\Debug\Lo21.exe
list :
F
G
C
A
B
D
E

Process returned 0 (0x0)   execution time : 0.033 s
Press any key to continue.
```

Quatrième partie

Commentaires sur les résultats

10 Réussite

Le système expert fonctionne, l'ensemble des sous-programmes demandés est opérationnel. Nous avons réussis à réaliser l'ensemble des algorithmes correspondant à ce dernier. Pour faciliter la programmation du moteur d'inférence nous avons fait 3 sous-programmes.

11 Amélioration

Nous pourrions rajouter des fonctions permettant le stockage des différentes règles/bases de connaissances dans des fichiers afin de les garder en mémoire. Nous pourrions également implémenter les opérations logiques $\vee$(Ou) et $\neg$(Non) en plus du $\wedge$(Et) que l'on utilise actuellement.

12 Conclusion

Ce projet nous a permis de comprendre ce qu'est un système expert et comment l'utiliser pour nos futurs projets. De plus nous avons améliorer nos connaissances en algorithmie, dans le langage C ainsi qu'en LaTeX.