

LO44

RAPPORT DE PROJET

LO44

Par Dimanna Augustin DIDJEIRA & William DJOUSSE



utbm
université de technologie
Belfort-Montbéliard

Table des matières

INTRODUCTION	2
1-Description des choix de conception et d'implémentation relatifs aux structures de données utilisées	2
Les enregistrements	2
Les pointeurs	2
Les types simples.....	2
Booléen	2
2-Algorithmes des sous-programmes.....	2
• Type regle.....	2
• Création d'une règle vide	3
• Ajout d'une proposition à la prémissse d'une règle	3
• Créer la conclusion d'une règle	4
• Tester si une proposition appartient à la prémissse d'une règle de manière récursive	4
• Supprimer une proposition de la prémissse d'une règle	5
• Tester si la prémissse d'une règle est vide.....	5
• Accéder à la proposition se trouvant à la tête de la base	6
• Accéder à la conclusion d'une règle	6
• Type Base de connaissance	6
• Créer une BC vide	6
• Ajouter une règle à une base	7
• Accéder à la règle se trouvant en tête de la base	7
• Moteur d'inférence	8
CAPTURE D'ECRAN DE TEST	9
• Démarrage du programme	9
• Résultats	9
PERSPECTIVES	10
CONCLUSION.....	11

INTRODUCTION

Un système expert est basé sur l'intelligence humaine simulée par les machines ; ayant la faculté de comprendre, de saisir par la pensée ; l'aptitude à s'adapter à une situation, à choisir en fonction des circonstances ; la capacité de donner un sens à telle ou telle chose. Un système expert est essentiellement composé d'une base de connaissance contenant des règles à priori toutes vraies ; une base de faits contenant des propositions considérées comme vraies et fin un moteur d'inférence non monotone, avec variables, fonctionnant en chaînage avant, avec l'interrogation de la base des faits.

1-Description des choix de conception et d'implémentation relatifs aux structures de données utilisées

Pour mener à bien ce projet, nous avons opéré des choix parmi plusieurs pour la conception et l'implémentation des différents modules.

Les enregistrements

Comme type composé, nous avons choisi les enregistrements encore appelés structure ; car les champs sont de différent type. Par la suite on a défini un type structure pour être capable d'accéder aux différents champs d'un enregistrement.

Les pointeurs

Vu que nous sommes dans un domaine d'expertise, nous ne connaissons pas la taille exacte des données qui seront traitées plus tard ; on a préféré choisir une variable dynamique pour stocker nos différents enregistrements, à savoir des listes simplement chaînées.

Les types simples

Parmi ces types, on a le type **caractère** car les règles seront essentiellement composées de caractère.

Le type entier principalement pour les fonctions qui devront compter le nombre de proposition d'une règle et le nombre de règle d'une base de connaissance.

Booléen

Le type Booléen n'existant pas en C, il nous fallait un type qui retournerait Vrai ou Faux (la valeur 1 pour Vrai et 0 pour Faux) lors des opérations de teste, notamment pour la fonction qui teste si une proposition appartient à une prémissé et la fonction qui teste si une prémissé est vide. Grâce à des types prédéfinis nous avons créé une structure de type entier (BOOL).

2-Algorithmes des sous-programmes

- Type regle

-champ valeur : chaîne de caractères
-champ suivant : pointeur sur la proposition suivante

- Création d'une règle vide

C'est la fonction de type constructeur, elle est indispensable pour la suite du projet. Afin de créer une règle vide, qui est en réalité un pointeur, nous avons réservé de la place en mémoire avec la fonction creerR () .

Lexique	Résultat
Cette fonction crée une règle vide	Règle vide Algorithme : creerR () -> regle <u>DEBUT</u> creerR()<-indefini <u>fin</u>

- Ajout d'une proposition à la prémissse d'une règle

Ici l'ajout d'une proposition à une prémissse se fait en queue. Les éléments à manipuler étant des listes chaînées, on a juste créé une variable temporaire que l'on place à la fin de la liste tout en changeant les coordonnées de l'élément se trouvant à la fin de la liste et une nouvelle variable pour stocker la liste courante.

lexique	Résultat
Variables Regle : new, I, temp; Données vide (I) ; vérifie si une règle est vide Succ () ; donne le suivant de la proposition Valeur_tete () ; renvoi la valeur de la proposition en tête	AjouterPro (R :regle, proposition :chaine)->regle <u>DEBUT</u> Valeur(new)<-proposition I<-tete(R); <u>Si</u> (vide (I)) <u>Alors</u> succ(new)<-indéfini AjouterPro(R,chaine)<-new <u>Sinon</u> <u>Tant que</u> (succ(I) /= indéfini) <u>faire</u> I <- succ (I) <u>Fait</u> Temp<-succ(I) Succ (I) <- new Succ (new) <- temp; AjouterPro(R,chaine)<-R; <u>FinSi</u> <u>FIN</u>

- Créer la conclusion d'une règle

Pour créer la conclusion d'une règle, on a opté pour l'ajout en queue dans la liste de proposition de telle sorte qu'elle corresponde au dernier élément.

lexique	Résultat
Variables Regle : new, l, temp; Données vide (l) ; Succ () ; Valeur_tete () ; On aura au préalable vérifier qu'une proposition est présente dans la règle. Une règle sans proposition ne peut donc pas avoir de conclusion	AjouterCON (R:regle, conclusion :chaine)->regle <u>DEBUT</u> (new).valeur<-proposition l<-tete(R) <u>Tant que</u> (succ(l) /= indéfini) faire l <- succ (l) Fait Temp<-succ(l) Succ (l) <- new Succ (new) <- temp AjouterCon(R,conclusion)<-R; FIN

- Tester si une proposition appartient à la prémissse d'une règle de manière récursive

lexique	Résultat
	test (R :regle, proposition :chaine)->booléen <u>DEBUT</u> <u>Si</u> (vide (R)) Alors test(R,proposition)<-FAUX <u>Sinon</u> <u>Si</u> (valeur(R)=proposition) Alors test(R,proposition)<-VRAI <u>Sinon</u> test(R,proposition)<-test(reste(R),proposition) <u>Finsi</u> FIN

- Supprimer une proposition de la prémissse d'une règle

lexique	Résultat
Variables Temp,I :regle	<p>supprimer (R :regle, proposition :chaine)->regle</p> <p><u>DEBUT</u></p> <p>I <- tete(R)</p> <p><u>Si</u> (vide(R)) <u>Alors</u></p> <p style="padding-left: 2em;">supprimer (R, proposition)<-indéfini</p> <p><u>Sinon</u></p> <p style="padding-left: 2em;"><u>Tant que</u> ((succ(I)/=indefini) <u>Faire</u></p> <p style="padding-left: 4em;">Si (valeur(R)=proposition) <u>Alors</u></p> <p style="padding-left: 6em;">R <- succ(I)</p> <p style="padding-left: 6em;">free (I)</p> <p style="padding-left: 4em;">supprimer (R, proposition)<-R</p> <p><u>Sinon</u></p> <p style="padding-left: 2em;">I<-succ(I)</p> <p><u>FinSi</u></p> <p><u>Fait</u></p> <p><u>FIN</u></p>

- Tester si la prémissse d'une règle est vide

lexique	Résultat
Tester si la prémissse d'une règle est vide revient à tester si la règle est vide car une règle sans prémissse ne peut pas avoir de conclusion	<p>test (R :regle, proposition :chaine)->booléen</p> <p><u>DEBUT</u></p> <p><u>Si</u> (valeur_tete(R)) <u>Alors</u></p> <p style="padding-left: 2em;">test(R)<-VRAI</p> <p><u>Sinon</u></p> <p style="padding-left: 2em;">test(R,proposition)<-FAUX</p> <p><u>Finsi</u></p> <p><u>FIN</u></p>

- Accéder à la proposition se trouvant à la tête de la base

Lexique	Résultat
On considère ici qu'on accède à la valeur d'une règle non vide Variables : R :rule	Valeur_tete (R :rule)->chaine <u>DEBUT</u> <u>Si</u> (non vide(R)) Alors Valeur_tete(R)<-valeur(R) <u>Finsi</u> <u>FIN</u>

- Accéder à la conclusion d'une règle

Lexique	Résultat
On considère ici que la conclusion existe Variables : R :rule	conclusion (R :rule)->chaine <u>DEBUT</u> <u>Si</u> (non vide(R)) Alors Tant que (succ(I) /= indéfini) faire I <- succ (I); Fait conclusion(R)<-valeur(I) <u>Finsi</u> <u>FIN</u>

- Type Base de connaissance

-Champ rule : rule
-Champ suivant : pointeur sur la règle suivante

- Créer une BC vide

Lexique	Résultat
BC c'est le type base de connaissance	creerB () -> BC <u>DEBUT</u> creerB()<-indefini <u>FIN</u>

- Ajouter une règle à une base

lexique	Résultat
Variables BC : new, l, temp;	<p>Algorithme : AjouterPro (base :BC, R :regle)->BC <u>DEBUT</u></p> <p>Valeur(new)<-R</p> <p>l<-tete(base);</p> <p><u>Si</u> (vide (l)) <u>Alors</u></p> <p style="padding-left: 2em;">succ(new)<-indéfini</p> <p style="padding-left: 2em;">AjouterPro(base,R)<-new</p> <p><u>Sinon</u></p> <p style="padding-left: 2em;"><u>Tant que</u> (succ(l) /= indéfini) <u>faire</u> l <- succ (l)</p> <p style="padding-left: 2em;"><u>Fait</u></p> <p style="padding-left: 2em;">Temp<-succ(l)</p> <p style="padding-left: 2em;">Succ (l) <- new</p> <p style="padding-left: 2em;">Succ (new) <- temp</p> <p style="padding-left: 2em;">AjouterPro(B,R)<-base</p> <p><u>FinSi</u></p> <p><u>FIN</u></p>

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

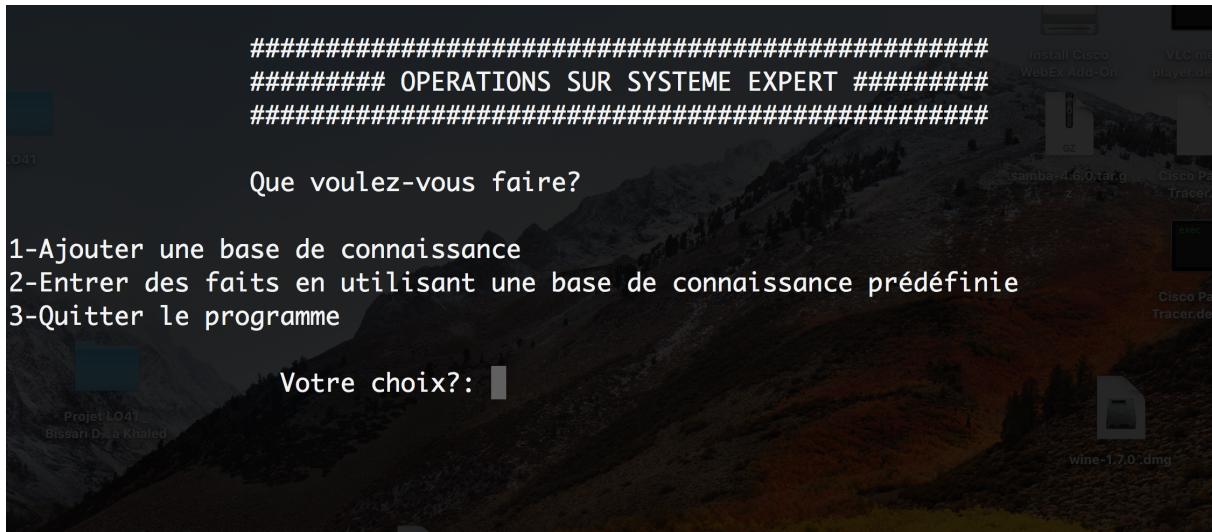
Lexique	Résultat
Valeur(base) indique le champ valeur de base comme base->valeur en C	<p>Valeur_tete (base : BC)->regle <u>DEBUT</u></p> <p><u>Si</u> (base/=indéfini) <u>Alors</u></p> <p style="padding-left: 2em;">Valeur_tete(B)<-valeur(base)</p> <p><u>FinSi</u></p> <p><u>FIN</u></p>

- Moteur d'inférence

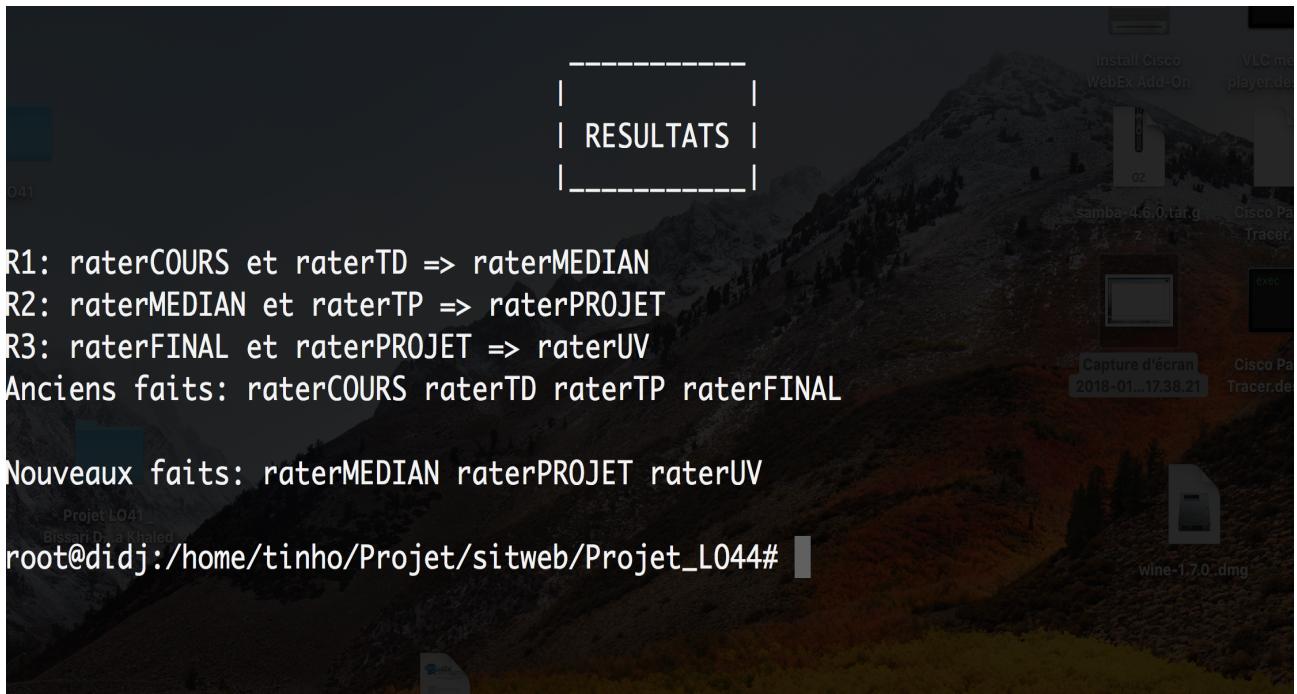
Lexique	Résultat
Variables newfaits et temp : BF//BF base de faits BC tempb e : entier add : booléen compterP compte le nombre de proposition dans la règle y compris la conclusion	<p>MI(base : BC, faits : BF)->BF DEBUT</p> <p>Newfaits<-indéfini</p> <p>add<-VRAI</p> <p>tant que add=VRAI Faire</p> <p>tempb<-base</p> <p>Tant que (temp=/indefini) faire</p> <p>Si (test (rule(tempb), valeur(temp)=VRAI) Alors</p> <p>A<-a+1</p> <p>FinSi</p> <p>temp<-succ(temp)</p> <p>Fait</p> <p>Si (a=compterP(rule(tempb)) -1) Alors</p> <p>Si(non existe(faits, conclusion(rule(tempb)))) Alors</p> <p>newfaits<-ajouterF(newfaits, conclusion(rule(tempb)))</p> <p>add<-VRAI</p> <p>Sinon</p> <p>add=FAUX</p> <p>FinSi</p> <p>tempb=succ(tempb)</p> <p>Si (newfaits =indéfini) Alors</p> <p>add=FAUX</p> <p>Finsi</p> <p>Fait</p> <p>MI(base,faits)<-newfaits</p> <p>FIN</p>

CAPTURE D'ECRAN DE TEST

- Démarrage du programme



- Résultats



PERSPECTIVES

Faute de temps, nous n'avons pu finir toutes les fonctionnalités que nous avions envisagées pour ce projet. Mais voici une liste non exhaustive de celles que nous trouvons pertinentes :

- Ajouter une proposition de type chaîne de caractères avec des espaces
- Pouvoir identifier des propositions quelque soit la casse (que ce soit en majuscules ou en minuscules ne fera aucune différence)
- Pouvoir utiliser les fichiers pour stocker la base de connaissance pour une future utilisation ou un futur lancement du programme sans avoir à la ressaisir entièrement.

CONCLUSION

Ce projet a été pour nous un exemple de réalisation possible grâce aux connaissances acquises au cours de l'UV LO41, un approfondissement. La modélisation, la conception et la réalisation ont été pour nous un défi à relever. Et notre démarche à surtout porté sur la modularisation du code et l'optimisation des ressources.