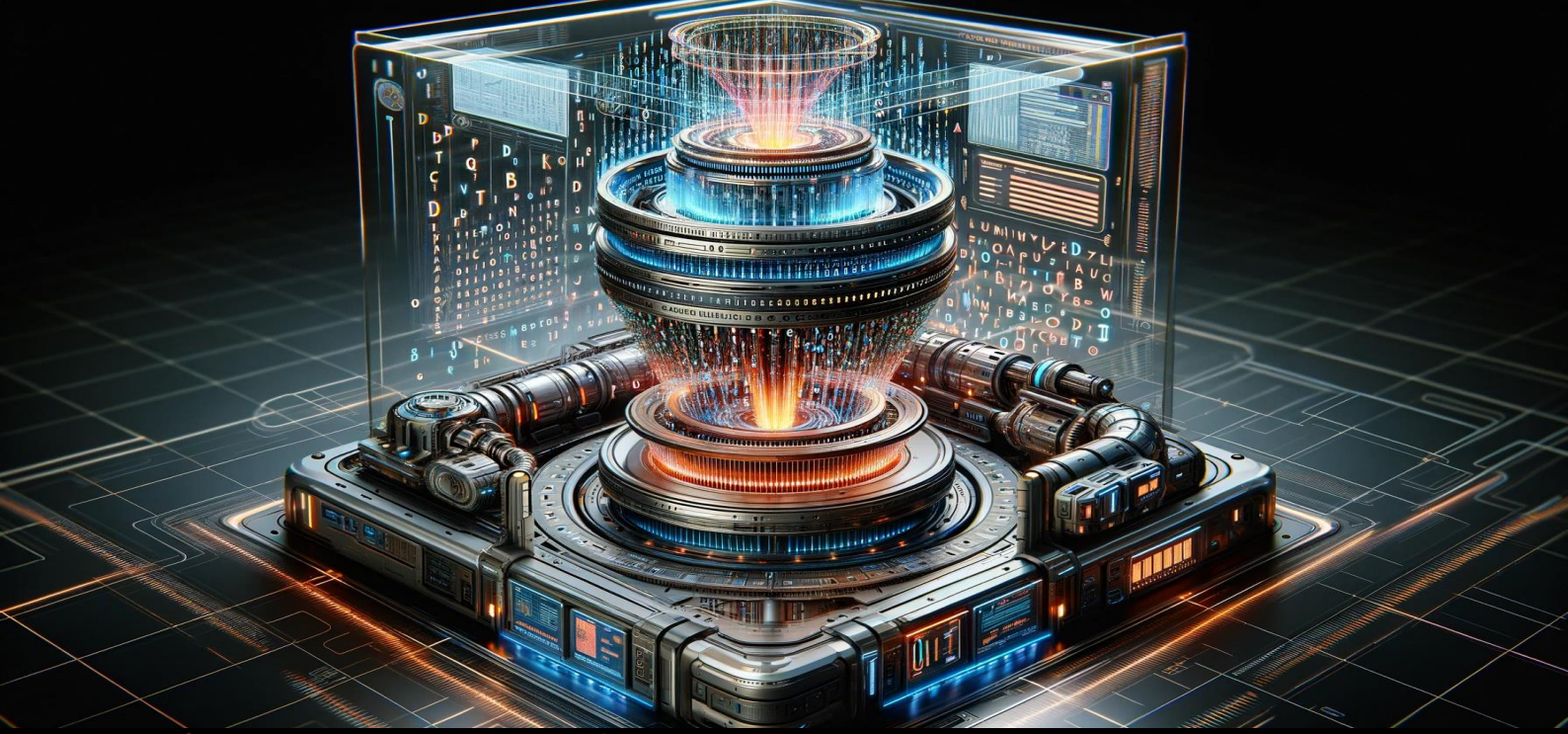

UNIVERSITÉ DE TECHNOLOGIE DE BELFORT-MONTBÉLIARD

Moteur d'inférence

Rapport Projet LO21 – Semestre A2023

BARRIOS Edgar
MANSOUR Thomas

Tronc Commun

UV LO21 Introduction to databases

Responsable UV: Abderrafiaa KOUKAM

utbm
université de technologie
Belfort-Montbéliard

Table des matières

Introduction

Présentation du projet et de son objectif.

Ce rapport présente le projet LO21, réalisé dans le cadre de notre cursus en informatique. L'objectif principal de ce projet est la conception et la réalisation d'un système expert, une composante clé dans le domaine de l'intelligence artificielle. Un système expert simule le raisonnement humain dans un domaine de connaissance spécifique, en utilisant des règles et des faits pour tirer des conclusions logiques.

Le projet LO21 vise à développer un système expert dans lequel les connaissances sont représentées sous forme de propositions logiques pouvant être vraies ou fausses. L'accent est mis sur la capacité du système à traiter ces propositions pour déduire de nouveaux faits, simulant ainsi un processus de raisonnement logique complexe. Les objectifs spécifiques comprennent la conception d'une base de connaissances, la mise en place d'une base de faits, et le développement d'un moteur d'inférence efficace.

Brève description du système expert et de ses composantes.

Base de connaissance

La base de connaissance est le cœur du système, elle stocke une série de règles. Chaque règle est formée d'une prémisse, qui est un ensemble de propositions liées par des opérateurs logiques, et d'une conclusion, qui est une proposition unique. Ces règles sont utilisées pour le processus d'inférence.

Base de faits

Une base de faits contient un ensemble de propositions considérées comme vraies. Elle représente les connaissances actuelles du système et sert de point de départ pour l'inférence.

Moteur

Il s'agit du mécanisme qui applique les règles de la base de connaissances aux faits de la base de faits pour déduire de nouveaux faits. Ce processus d'inférence est au cœur de la fonctionnalité du système expert.

En somme, ce projet LO21 est une exploration approfondie dans la conception et l'implémentation d'un système expert, mettant en avant les compétences en programmation, en logique et en conception algorithmique. Ce rapport détaille les étapes de développement du système, de sa conception à sa réalisation, et examine les résultats obtenus à travers divers tests et analyses.

Choix de structure de données

Nous avons choisi d'implémenter les composantes du système expert en utilisant des structures de données sous forme de listes chainées spécialement conçues pour représenter les règles, les propositions, et les bases de connaissances et de faits. Les choix ont été guidés par le besoin de gérer efficacement des listes de propositions et de règles, ainsi que par la nécessité d'une manipulation aisée des éléments pour les opérations d'inférence.

Les données sont donc organisées de la manière suivante :

- La base de connaissance est composée d'une liste chainée de règle
- Une règle est composée d'une liste chainée de propositions avec la conclusion comme dernière proposition
- Une base de fait est une liste chainée de faits.

Développement des types abstraits

Elem_Bc

Une règle est une structure composée des champs : pointeur vers la première proposition de la prémisse (proposition* prémisse), un pointeur vers la conclusion (proposition* conclusion) et un pointeur vers la prochaine règle (règle* prochain).

Ce type dispose de fonctions clés pour pouvoir les manipuler comme :

```
Fonction qui cree une regle vide
Fonction creer_regle_vide()->elem_BC
Debut_fonction
    nouvelle_regle<-creer()
    si nonvide(nouvelle_regle) alors
        premisse(nouvelle_regle)<-null
        conclusion(nouvelle_regle)<-null
        prochain(nouvelle_regle)<-null
    fin si
    creer_regle_vide<-nouvelle_regle
Fin_fonction
```

```
Fonction qui ajoute une proposition à une regle
Fonction ajouter_proposition(regle:elem_BC, valeur:sequence de caracteres)->elem_BC
Debut_fonction
    nouvelle_prop<-creer()
    value(nouvelle_prop)<-valeur
    next(nouvelle_prop)<-conclusion(regle)

    si vide(premisse(regle)) alors
        premisse(regle)<-nouvelle_prop
        prev(nouvelle_prop)<-null
    sinon
        si nonvide(conclusion(regle)) et nonvide(prev(conclusion(regle))) alors
            next(prev(conclusion(regle)))<-nouvelle_prop;
            prev(nouvelle_prop)<-prev(conclusion(regle));
            prev(conclusion(regle))<-nouvelle_prop;
        sinon
            derniere: proposition<-premisse(regle)
            tant que nonvide(next(derniere)) faire
                derniere<-next(derniere)
            fin tant que
            next(derniere)<-nouvelle_prop
            prev(nouvelle_prop)<-derniere
        fin si
    fin si
    ajouter_proposition<-regle
Fin_fonction
```

```
Fonction qui cree la conclusion d'une regle
Fonction creer_conclusion(regle:elem_BC, valeur:sequence de caracteres)->elem_BC
Debut_fonction
    si nonvide(conclusion(regle)) alors
        value(conclusion(regle))<-valeur
    sinon
        concl:proposition<-creer()
        value(concl)<-valeur
        next(concl)<-null
        prev(concl)<-null

        si nonvide(premisse(regle)) alors
            tmp:proposition<-premisse(regle)
            tant que nonvide(next(tmp)) faire
                tmp<-next(tmp)
            fin tant que
            next(tmp)<-concl
            prev(concl)<-tmp
        fin si

        conclusion(regle)<-concl
    fin si

    creer_conclusion<-regle
Fin_fonction
```

```
Les 2 prochaines fonctions permettent de verifier si une proposition appartient à la premisse d'une regle de maniere recursive
Fonction appartient_premisse_recursif(courante:proposition, conclusion:proposition, valeur:sequence de caracteres)->booleen
Debut_fonction
    si vide(courante) ou courante = conclusion alors
        appartient_premisse_recursif<-0
    fin si

    si value(courante) = valeur alors
        appartient_premisse_recursif<-1
    fin si

    appartient_premisse_recursif<-appartient_premisse_recursif(next(courante), conclusion, valeur)
Fin_fonction

Fonction appartient_premisse(regle:elem_BC, valeur:sequence de caracteres)->booleen
Debut_fonction
    si vide(regle) ou vide(valeur)
        appartient_premisse<-0
    fin si

    appartient_premisse<-appartient_premisse_recursif(premisse(regle), conclusion(regle), valeur)
Fin_fonction
```

```
Fonction qui supprime une proposition d'une regle
Fonction supprimer_proposition(regle:elem_BC, valeur:sequence de caracteres)->elem_BC
Debut_fonction
    si vide(regle) ou vide(valeur) ou vide(premisse(regle)) alors
        supprimer_proposition<-regle
    fin si

    courante:proposition<-premisse(regle)
    precedente:proposition<-null

    tant que nonvide(courante) et courante different de conclusion(regle) faire
        si value(courante) = valeur alors
            si vide(precedente) alors
                premisse(regle)<-next(courante)
            sinon
                next(precedente)<-next(courante)
                si nonvide(next(courante)) alors
                    prev(next(courante))<-precedente
                fin si
            fin si
        fin si
        precedente<-courante
        courante<-next(courante)
    fin tant que

    supprimer_proposition<-regle
Fin_fonction
```

```
Fonction qui verifie si la premisse d'une regle est vide
Fonction premisse_est_vide(regle:elem_BC)->booleen
Debut_fonction
    si vide(regle) alors
        premisse_est_vide<-1
    fin si

    si premisse(regle) = conclusion(regle) alors
        premisse_est_vide<-1
    sinon
        premisse_est_vide<-0
    fin si
Fin_fonction
```

```
Fonction qui accede à la premiere proposition d'une regle
Fonction premiere_proposition(regle:elem_BC)->proposition
Debut_fonction
    si vide(regle) ou vide(premisse(regle)) alors
        premiere_proposition<-null
    fin si

    premiere_proposition<-premisse(regle)
Fin_fonction
```

```
Fonction qui accede à la conclusion d'une regle
    Fonction acceder_conclusion(regle:elem_BC)->proposition
    Debut_fonction
        si vide(regle) ou vide(conclusion(regle)) alors
            acceder_conclusion<-null
        fin si

        acceder_conclusion<-conclusion(regle)
    Fin_fonction
```

Proposition

Une proposition est une structure dont les champs sont : une chaine de caractères (char* value), un pointeur vers la prochaine proposition (prop* next) et un pointeur vers la précédente proposition (prop* prev).

Elles sont essentielles pour la construction des prémisses et conclusions dans les règles.

Liste_BC

La base de connaissance est une structure dont les champs sont : un pointeur vers la première règle (elem_BC* BC) et le nombre de règles (int nb_elem).

Ce type dispose de fonctions clés pour pouvoir les manipuler comme :

```
Fronction qui cree une base de connaissances vide
Fonction creer_base_vide()->liste_BC
Debut_fonction
    base:liste_BC<-creer()
    si nonvide(base) alors
        BC(base)<-null
        nb_elem(base)<-0
    fin si
    creer_base_vide<-base
Fin_fonction
```

```
Fonction qui ajoute une regle à la base de connaissances
Fonction ajouter_regle(base:liste_BC, nouvelle_regle:elem_BC)->liste_BC
Debut_fonction
    si vide(base) alors
        ajouter_regle<-null
    fin si

    si vide(nouvelle_regle) alors
        ajouter_regle<-base
    fin si

    si vide(BC(base)) alors
        BC(base)<-nouvelle_regle
    sinon
        tmp:elem_BC<-BC(base)
        tant que nonvide(prochain(tmp)) faire
            tmp<-prochain(tmp)
        fin tant que
        prochain(tmp)<-nouvelle_regle
    fin si

    nb_elem(base)<-nb_elem(base)+1
    ajouter_regle<-base
Fin_fonction
```

```
Fonction qui accede à la premiere regle de la base de connaissances
Fonction acceder_regle_tete(base:liste_BC)->elem_BC
Debut_fonction
    si vide(base) ou vide(BC(base)) alors
        acceder_regle_tete<-null
    fin si

    acceder_regle_tete<-BC(base)
Fin_fonction
```

```
 Fonction qui supprime les regles dont la premisse est vide
Fonction supprimer_regle_vide(base:liste_BC)->liste_BC
   Debut_fonction
      si vide(base) ou vide(BC(base)) alors
         supprimer_regle_vide<-base
      fin si

      regle_actuelle:elem_BC<-BC(base)
      regle_precedente:elem_BC<-null

      tant que nonvide(regle_actuelle) faire
         si premisse_est_vide(regle_actuelle)=1 alors
            si vide(regle_precedente) alors
               BC(base)<-prochain(regle_actuelle)
            sinon
               prochain(regle_precedente)<-prochain(regle_actuelle)
            fin si

            regle_a_supprimer:elem_BC<-regle_actuelle
            regle_actuelle<-prochain(regle_actuelle)

            nb_elem(base)<-nb_elem(base)-1
         sinon
            regle_precedente<-regle_actuelle
            regle_actuelle<-prochain(regle_actuelle)
         fin si
      fin tant que

      supprimer_regle_vide<-base
   Fin_fonction
```

Liste_BF

La base de fait est une structure dont les champs sont : un pointeur vers le premier fait (proposition* BF) et le nombre de faits (int nb_elem).

Ce type dispose également de fonctions clés pour pouvoir les manipuler comme :

```
Fonction qui cree une base de faits vide
Fonction creer_base_vide_BF()->liste_BF
Debut_fonction
    base:liste_BF<-creer()
    si nonvide(base) alors
        BF(base)<-null
        nb_elem(base)<-0
    fin si
    creer_base_vide_BF<-base
Fin_fonction
```

```
Fonction qui ajoute une proposition à la base de faits
Fonction ajouter_proposition_BF(base:liste_BF, valeur:sequence de caracteres)->void
Debut_fonction

    nouvelle_prop:proposition<-creer()
    value(nouvelle_prop)<-valeur
    next(nouvelle_prop)<-null

    si nonvide(base)
        si vide(BF(base))
            BF(base)<-nouvelle_prop
            prev(nouvelle_prop)<-null
        sinon
            tmp:proposition<-BF(base)
            tant que nonvide(next(tmp)) faire
                tmp<-next(tmp)
            fin tant que
            prev(nouvelle_prop)<-tmp
            next(tmp)<-nouvelle_prop
        fin si

        nb_elem(base)<-nb_elem(base)+1
    fin si
Fin_fonction
```

Importation des données

Base de données

Le moteur d'inférence du projet LO21 joue un rôle crucial en traitant les données importées pour réaliser des déductions logiques. Les données sont importées à partir d'un fichier "sauvegarde.txt", où elles sont organisées dans un format spécifique. Chaque ligne du fichier représente soit une règle sous la forme "A, B => C", où A et B sont les prémisses menant à la conclusion C, soit un fait isolé comme "A" ou "B". Cette organisation facilite la séparation claire entre les règles et les faits, éléments fondamentaux pour le processus d'inférence.

Fonction de chargement

```
// Fonction qui charge la base de connaissances et la base de faits depuis un fichier texte
   Fonction charger_base_de_connaissances(nom_fichier:chaine)->booleen
   Debut_fonction
       fichier:FILE<- ouverture du fichier nom_fichier en mode lecture seule
       afficher "Chargement du fichier"
       afficher nom_fichier
       ligne:séquence de caracteres
       token:caractere

       si vide(fichier) alors
           afficher "Erreur lors de l'ouverture du fichier"
           charger_base_de_connaissances<-0
       fin si

       tant que nonvide(ligne du fichier) faire
           Enlever le saut de ligne à la fin de ligne si présent

           si "=>" présent dans ligne alors
               nouvelle_regle:elem_BC<-creer_regle_vide();

               token<-chaine de caracteres de ligne jusqu'a la prochaine virgule
               tant que nonvide(token) et token different de "=>" faire
                   nouvelle_regle<-ajouter_proposition(nouvelle_regle, token);
                   token<-chaine de caracteres de ligne jusqu'a la prochaine virgule
               fin tant que

               si nonvide(token) et nonvide(???) alors
                   token<-chaine de caracteres de ligne jusqu'a la prochaine virgule
                   si nonvide(token) alors
                       nouvelle_regle<-creer_conclusion(nouvelle_regle, token);
                       base_de_connaissances<-ajouter_regle(base_de_connaissances, nouvelle_regle);
                   fin si
               fin si


           sinon
               ajouter_proposition_BF(base_faits, ligne)
           fin si
       fin tant que

       fermeture du fichier ouvert
       afficher "Chargement termine."
       charger_base_de_connaissances<-1
   Fin_fonction
```

La fonction « charger_base_de_connaissances_et_faits » joue un rôle essentiel dans ce processus. Elle lit le fichier ligne par ligne, distinguant les règles des faits. Pour chaque règle, elle extrait les prémisses et la conclusion, construisant une structure « elem_BC » qui est ensuite ajoutée à la base de connaissances. De même, chaque fait est identifié et ajouté à la base de faits. Cette fonction assure ainsi que toutes les données nécessaires sont correctement chargées et structurées pour être utilisées par le moteur d'inférence.

Le main

```
VARIABLES
    fichier_charge_avec_succes:booleen<-0

DEBUT_ALGORITHME

    base_connaissances:liste_BC<-creer_base_vide()
    base_faits:liste_BF<-creer_base_vide_BF()
    fichier_charge_avec_succes<-charger_base_de_connaissances("base_de_connaissances.txt")

    si fichier_charge_avec_succes=1 alors
        fait:proposition<-BF(base_faits)
        regle:elem_BC

        tant que nonvide(fait) faire
            afficher "On regarde pour le fait" value(fait)
            regle<-acceder_regle_tete(base_connaissances)

            tant que nonvide(regle)
                si appartient_premisse(regle, value(fait))=1 alors
                    regle<-supprimer_proposition(regle, value(fait))
                    si premisse_est_vide(regle)=1 alors
                        si nonvide(acceder_conclusion(regle)) alors
                            ajouter_proposition_BF(base_faits, value(acceder_conclusion(regle)))
                            base_connaissances<-supprimer_regle_vide(base_connaissances)
                        fin si
                    fin si
                fin si

                regle<-prochain(regle)
            fin tant que

            fait<-next(fait)
            regle<-acceder_regle_tete(base_connaissances)
        fin tant que
    fin si

    afficherBCBF(base_faits, base_connaissances)

FIN_ALGORITHMES
```

Le cœur du moteur d'inférence, situé dans la fonction principale (main), commence par charger les données à l'aide de la fonction mentionnée. Une fois les données chargées, le moteur parcourt la base de faits pour identifier les faits actuels. Ensuite, il applique récursivement les règles de la base de connaissances sur ces faits. À chaque étape, il vérifie si les prémisses d'une règle sont satisfaites par les faits actuels. Si c'est le cas, la conclusion de cette règle est ajoutée à la base de faits. Ce processus se poursuit jusqu'à ce qu'aucune nouvelle information ne puisse être déduite, signifiant que toutes les déductions possibles ont été réalisées.

Le moteur d'inférence, grâce à cette méthodologie, parvient à simuler un raisonnement logique, partant d'un ensemble de faits et de règles pour arriver à de nouvelles conclusions. Cette approche reflète le fonctionnement des systèmes experts, où la capacité à tirer des conclusions à partir de connaissances préétablies est essentielle. L'organisation efficace des données et la fonctionnalité robuste de chargement et de traitement garantissent la fiabilité et l'efficacité du système dans la réalisation de son objectif principal : l'inférence de nouvelles connaissances.

Test de lecture de fichier

Objectif : Vérifier que le système lit correctement les données du fichier et les charge dans les bases de connaissances et de faits.

Procédure : Nous avons utilisé la fonction « afficherBCBF » que nous avions créé initialement dans le but de déboguer le bon fonctionnement de l'algorithme, comprenant ainsi le bon chargement des données. Cette fonction a été placé au début du main avant que le moteur d'inférence ne commence son travail afin de vérifier que les données étaient bien importées.

```
fonction qui affiche chaque valeur de la base de faits et chaque regle de la base de connaissances
    Fonction afficherBCBF(base_faits:liste_BF, base_connaissances:liste_BC)->void
    Debut_fonction
        afficher "on a" nb_elem(base_faits) "faits et" nb_elem(base_connaissances) "regles"
        liste_fait:proposition<-BF(base_faits)
        tant que nonvide(liste_fait) faire
            afficher value(liste_fait)
            liste_fait<-next(liste_fait)
        fin tant que

        afficher "Liste des regles:"
        liste_regle:elem_BC<-BC(base_connaissances)
        tant que nonvide(liste_regle)
            courante:proposition<-premisse(liste_regle)

            tant que nonvide(courante) et courante différent de conclusion(liste_regle) faire
                afficher value(courante)
                courante<-next(courante)
            fin tant que

            si nonvide(conclusion(liste_regle)) alors
                afficher "=>" value(conclusion(liste_regle))
            sinon
                afficher "=> NULL"
            fin si

            liste_regle<-prochain(liste_regle)
        fin tant que
    Fin_fonction
```

Résultat : Les données sont correctement chargées dans les variables nous permettant ainsi de travailler avec notre base de connaissance et de faits.

Gestion erreurs de format

Objectif : Tester la robustesse du système face à des erreurs de format dans le fichier de données.

Procédure : Nous avons créé plusieurs fichiers de test avec des erreurs de format différentes, comme des règles mal formées, des faits incorrects, ou des caractères inattendus.

Nous avons tenté de charger ces fichiers et avons observez comment le système réagit. Le système devait être capable d'identifier ces erreurs et de les gérer sans crasher.

Résultat : Nous avons ajusté la fonction « charger_base_de_connaissances_et_faits » en créant une syntaxe précise. Les règles et les faits doivent être écris en respectant la syntaxe suivante : « A, B =>

C ». Il doit y avoir une virgule suivie d'un espace entre chaque fait, un espace entre la flèche et le dernier fait et au moins un espace entre la flèche et la conclusion.

Test de doublons

Objectif : Assurer que le système gère correctement les doublons dans les fichiers de données.

Procédure : Nous avons créé un fichier de données contenant des règles et des faits en double.

Nous avons chargé ces données et vérifié que les doublons ne sont pas ajoutés aux bases de connaissances et de faits.

Résultat : Nous avons ajusté la fonction « charger_base_de_connaissances_et_faits » de telle sorte que les doublons ne soient pas importés en plusieurs exemplaires.

Conclusion

Récapitulatif des réalisations et des apprentissages.

Le projet LO21 a été un enrichissant et éducatif, marqué par la conception et la mise en œuvre d'un système expert. Au cours de ce projet, nous avons développé une base de connaissances et une base de faits, ainsi qu'un moteur d'inférence robuste. Ces composantes nous ont permis de simuler le raisonnement logique, un aspect fondamental de l'intelligence artificielle. Le processus d'apprentissage était intensif mais gratifiant, offrant des perspectives profondes sur les structures de données complexes, l'algorithmique, et le traitement logique. Cela nous a également permis de manière concrète d'observer à quel point il est efficace de penser son algorithme avant de l'écrire directement dans un langage machine.

Perspectives d'amélioration

Pour l'avenir, plusieurs améliorations peuvent être envisagées pour renforcer la fonctionnalité et l'efficacité du système :

- Mise à jour dynamique de la base de faits : Améliorer le stockage des résultats du moteur d'inférence pour une mise à jour plus efficace et précise de la base de faits.
- Interface Interactive : Développer une interface utilisateur interactive pour faciliter l'ajout de nouvelles données et la manipulation des structures existantes.
- Gestion d'Hypothèses : Implémenter une gestion des hypothèses, qu'elles soient monotones ou non monotones, pour une flexibilité accrue dans le raisonnement.
- Chaînage Avant, Arrière ou Mixte : Explorer différentes méthodes de chaînage pour diversifier les approches d'inférence.
- Complétude Déductive : Évaluer et améliorer la complétude déductive du système, garantissant des résultats d'inférence exhaustifs et précis.

Extensions futures du projet.

En regardant vers l'avenir, le potentiel d'extension de ce projet est vaste :

- Création d'un Logiciel IA : Transformer le système en un logiciel complet d'IA, applicable dans divers contextes et industries.
- Système de Gestion de Données : Adapter le système pour la gestion de données dans différents secteurs professionnels, apportant une aide précieuse dans la prise de décision et l'analyse.
- Création de Jeux de Déduction : Utiliser le système pour développer des jeux basés sur la déduction, offrant un divertissement éducatif et stimulant l'esprit.

En conclusion, le projet LO21 a jeté les bases d'un système expert prometteur, avec un potentiel d'expansion et d'adaptation dans de nombreux domaines. Les compétences et les connaissances acquises au cours de ce projet serviront de tremplin pour des innovations futures dans le domaine de l'intelligence artificielle.

Mots clef

Projet, Système expert, inférence, faits, règles

BARRIOS Edgar

MANSOUR Thomas

Rapport Projet LO21 – Semestre A2023

Résumé

Dans le cadre de l'unité de valeur LO21 à l'Université de Technologie, ce rapport détaille notre projet de conception d'un système expert. Le projet, une partie intégrante de notre cursus en informatique, vise à développer un système capable de réaliser des inférences logiques à partir d'une base de connaissances et d'une base de faits. Nous avons également élaboré un moteur d'inférence pour traiter ces données. Des séries de tests ont été effectuées pour valider la fonctionnalité et l'efficacité du système. Ce rapport résume les processus de développement, les défis rencontrés, et les leçons apprises, tout en proposant des perspectives pour l'amélioration et l'expansion future du projet.

Moteur d'inférence