Rapport de projet RN40 : Système Expert

Automne 2023

Année académique: 2023-2024

Réalisé par : Pierre Charle et Michel Dieng

Professeur : Abderrafiaa Koukam

I- Choix de conception et d'implémentation relatifs aux structures de données utilisées et à la démarche adoptée.

Le projet de RN40 porte un système expert. Pour cela nous avons définis plusieurs parties :

-Une base de connaissance.

-Une base de fait.

-Un moteur d'interférence.

Pour cela nous avons définis 3 types de variable, le type Regle formé à partir du type prémisse qui est une liste chainée et le type conclusion étant une chaine de caractères, le type BaseDeConnaissance qui est une liste de règle et BaseDeFait qui est une liste chainée.

Déclaration des types :

Type Regle :

```
1)

typedef struct prop {
        char val;
        struct prop* suivant ;
} Proposition;

typedef Proposition * Premisse;
typedef char* Conclusion;

typedef struct reg{
        Premisse premisse;
        Conclusion conclusion;
} Regle;
```

Type BC :

```
typedef struct elem{
        Regle tete;
        struct elem*suivant;
}ElemBC;

typedef ElemBC* BC;
```

II- Algorithme des sous programmes

1- Algorithme des règles

a) Créer règle vide

On alloue la mémoire pour une règle et on la retourne.

```
Créer-règle
Entrée: rien
Sortie: Règle

nouvelleRegle = new(regle)
nouvelleRegle->premise=NULL
nouvelleRegle->conclusion=NULL
retourner nouvelleRegle
```

b) Ajouter une proposition à une prémisse d'une règle

```
ajouter_proposition
Entrée: Regle r et proposition p
Sortie : Regle r

On crée une nouvelle prémise P1 avec pour valeur p
Si r est vide alors P1 devient la prémise de r
Sinon on crée pémisse p2 = regle-> prémisse
        tant qu'on est pas à la fin de dernier alors:
                dernier= dernier-> suivant
        on place P1 à la fin
retourner r
```

c) Créer une conclusion à une règle

Si la règle n'est pas vide la conclusion prend pour valeur la proposition.

```
créer_conclusion
Entrée: Regle r et proposition p
Sortie: regle r

Si r est vide ce n'est pas possible de mettre une conculsion
Sinon on met p comme conclusion pour r
```

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

On utilise la récursivité afin de parcourir toute la prémisse et déduire si la proposition est dans la prémisse.

```
est_proposition
Entrée : prémisse P et proposition p
Sortir : booléen

Si prémisse est vide alors retourner False
Sinon Si P->val = p alors retourner True
      Sinon retourner est_proposition(P->suivant,p)
```

e) Supprimer une proposition de la prémisse d'une règle

Si P est vide, on retourne indéfini. Si P->suivant est indéfini et p est la proposition alors libérer(P) et retourner indéfini, sinon retourner P. Si P et P-> suivant sont définis, P->val = p alors on retourne une prémisse P2 qui vaut P->suivant . Sinon on retourne supprimer_proposition pour l'élément suivant de P.

```
Si P est indéfini alors retourner indéfini
si P->suivant est indéfini alors
                si P->val = p alors
                                libérer(p)
                                libérer(P)
                                retourner indéfini
                sinon
                        retourner P
si P->val = p alors
                Premisse P2= supprimer_proposition(P->suivant, p)
                libérer(p)
                retourner P2
sinon
        P->suivant = retirer_proposition (P->suivant, p )
        retourner P
```

f) Tester si la prémisse d'une règle est vide

On retourne vrai si la prémisse ne contient rien, sinon on retourne faux.

```
prémisse_est_vide
Entrée : prémisse P
Sortie : Booléen

Si P est indéfini alors retourner True
Sinon retourner False
```

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

On renvoie la première proposition de la prémisse

```
tete_premisse
Entrée: prémisse P
Sortie: proposition p

Retourner P->val
```

h) Accéder à la conclusion d'une règle

On renvoie simplement la conclusion.

```
conclusion_regle
Entrée : Regle r
Sortie : proposition p étant la conclusion

retourner r->conclusion
```

2) Algorithme des Base de connaissance

a) Créer une base vide

On alloue la mémoire pour une variable BC et on la renvoie.

```
créerBCvide
Entrée: rien
Sortie: BC

nouvelleBC = new(ElemBC)
nouvelleBC->suivant=NULL
retourner nouvelleBC
```

b) Ajouter une règle à une base

Si la base est vide, on ajoute simplement la règle dans la base de connaissance. Sinon on parcourt la base et l'on place la règle à la fin de celle-ci.

```
ajouter_regle
Entrée: Base de connaissance B et Regle r
Sortie : Base de connaissance B

Si B est vide alors B->tete = r
Sinon on crée B1 = base->tete
        tant que B1->suivant!=NULL alors
                B1= B1->suivant
        B1->suivant = r
retourner B1
```

c) Accéder à la règle se trouvant en tête de la base

On retourne simplement la tête.

```
TeteBase
Entrée: Base de Connaissance B
Sortie : Regle r

retourner B->tete
```

3) Algorithmes du moteur d'inférence

a) Recherche de maladie

On crée des variables tampons afin de ne pas abimé nos données, puis tant que la base de fait tampon et la base de connaissance ne sont pas vide, on ajoute à une base de fait connu la règle comprenant le fait donné. On retourne alors la base de fait connu.

```
rechercheMaladie
Entrée: BC knowledge_basis et BF fact_basis
Sortie:  BC known_fact

BC knowledge_buffer= créerBaseVide()
BC know_fact = créer BaseVide()
BF fact_buffer= créerListe()

knowledge buffer= knowledge_basis
fact_buffer= fact_basis

tant que estVide(fact_butter)== Faux faire
  tant que estVide(knowledge_buffer)== Faux faire
    Premisse premisse_buffer = premisse (tete(knowledge_buffer->suivant))
    tant que premisse_buffer != indéfini faire
        si premisse_buffer->val = fact_buffer->val alors
          known_fact = ajouter_regle(know_fact, tete(knowledge_buffer->suivant)
        fin si
        premisse_buffer = premisse_buffer->suivant
    fin tant que
  knowledge_buffer= knowledge_buffer->suivant
fin tant que
retourner known_fact
```

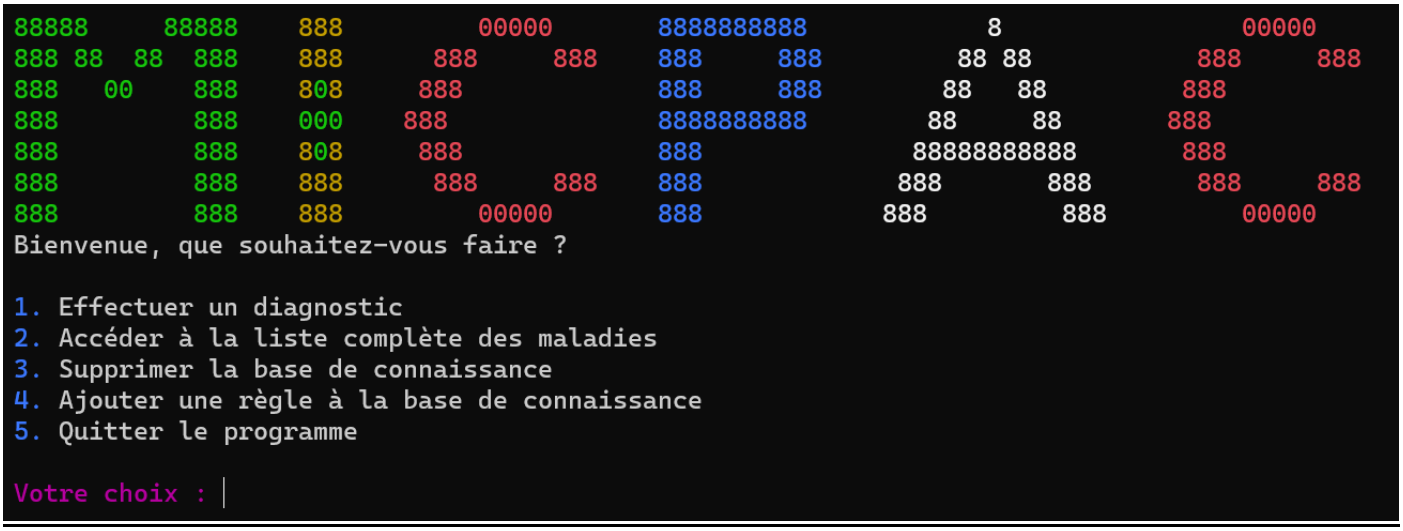
b) Moteur d'inférence

Si la base de connaissance est vide, alors on indique qu'on ne peut rien déduire, sinon tant que la base de fait tampon est définie, on applique rechercheMaladie à la base de connaissance tampon puis on avance sur la base de fait, une fois la boucle finis on indique ce que l'on a trouvé.

```
moteurInférence
Entrée: BC knowledge_basis et BF fact_basis
Sortie: void

Si estVide(knowledge_basis) alors
  afficher("On ne peut rien déduire")
Sinon BC knowledge_buffer=créerBaseVide()
      BF fact_buffer= créerBaseVide()

      knowledge_buffer= knowledge_basis
      fact_buffer= fact basis

      tant que fact_buffer != indéfini faire
        knowledge_buffer = rechercheMaladie(knowledge_buffer, fact_buffer)
        fact_buffer= fact_buffer->suivant
      fin tant que

      Si estVide(knowledge_buffer) alors
        afficher("Aucune maladie ne correspond")
      Sinon
        afficher(conclusion(tete(knowledge_buffer->suivant)))
      fin si
fin si
```

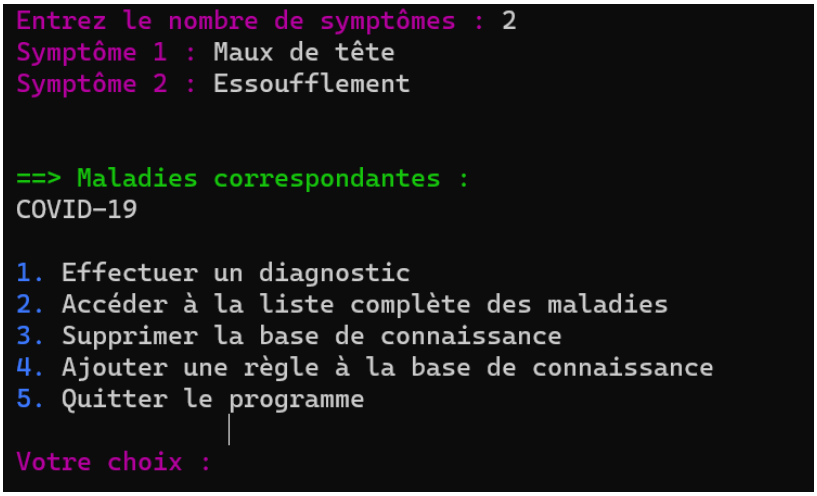
III- Jeux d’essais

Menu principal :



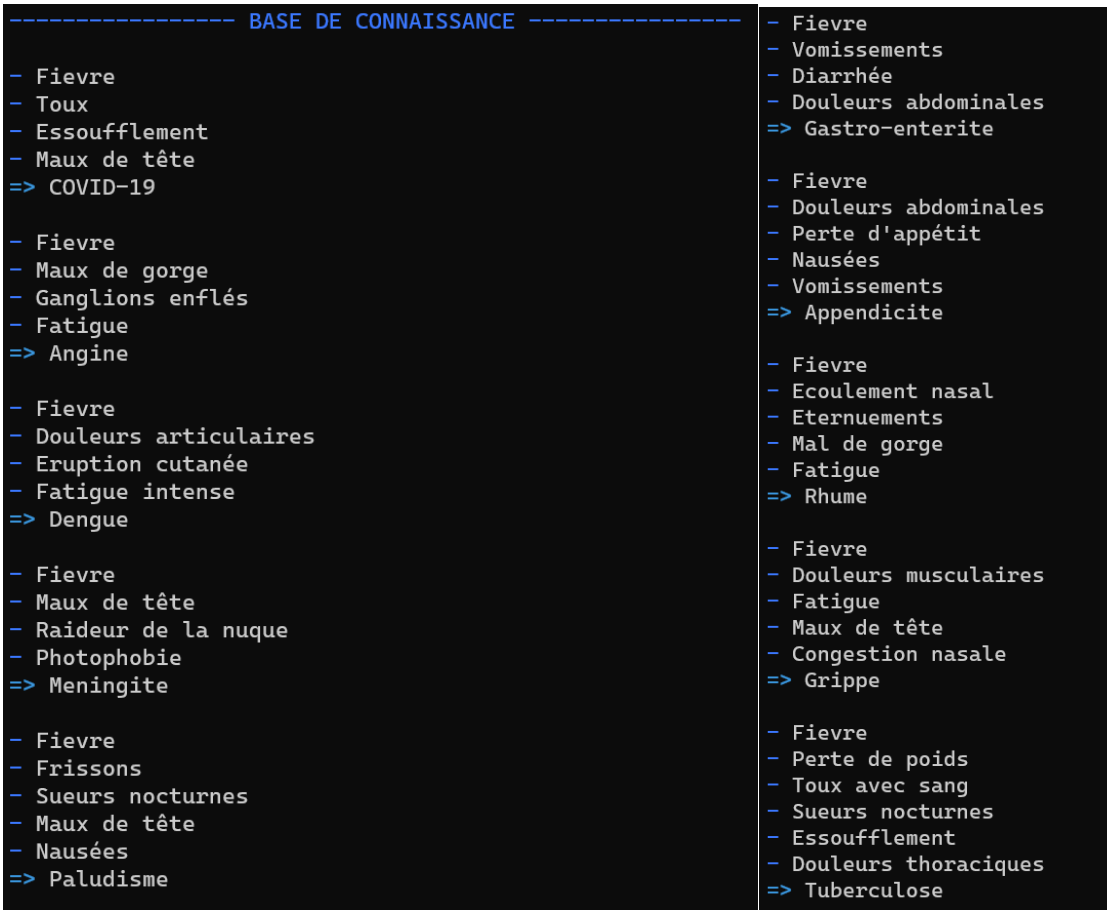
Le programme nous propose 5 choix d’actions différents, nous allons toute les essayer.

Lorsque que je tape 1, j’effectue un diagnostic :



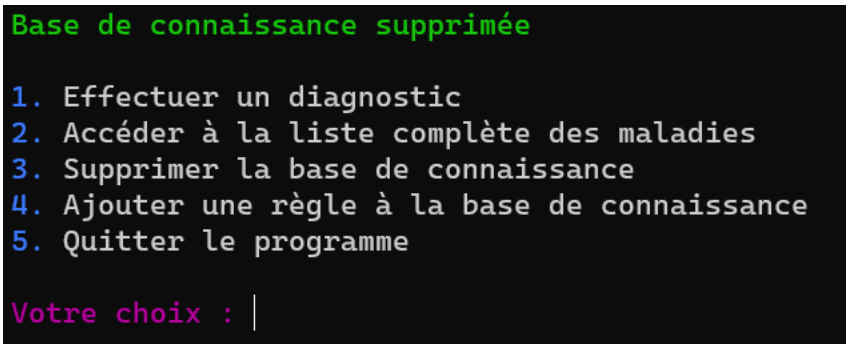
Pour diagnostiquer une maladie, il faut préciser le nombre de symptômes puis les énoncer. Le programme nous renvoie ensuite quelle maladie correspond aux symptômes.

Lorsque je tape 2, j’accède à la liste des maladies :

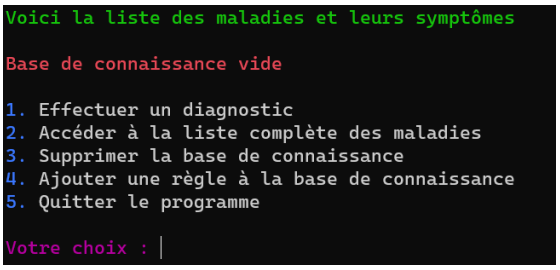


Nous avons décider de remplir notre base de connaissance de 10 différentes maladies afin d'illustrer notre travail.

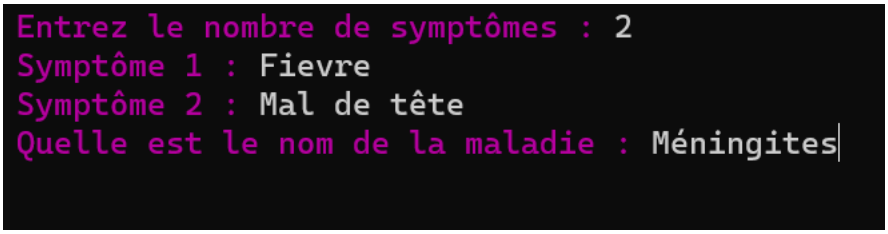
Lorsque je tape 3, je supprime la base de connaissance :



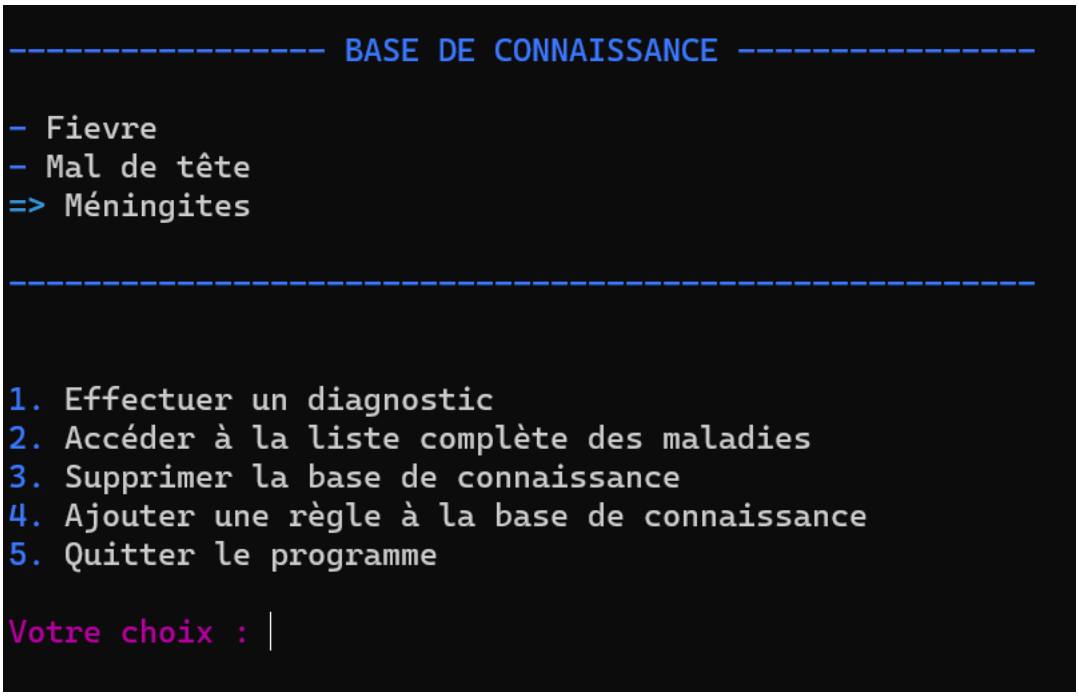
Lorsque nous supprimons la base de connaissance, un message nous indique que celle-ci a bien été supprimé. Ainsi lorsque nous souhaitons afficher la base de connaissance en tapant 2, un message nous indiquant que celle-ci est vide apparait :



Lorsque j'appuie sur 4, on ajoute une règle à la base de connaissance :



Un menu apparait pour spécifier le nom et les symptômes de la maladie, ainsi lorsque que l'on affiche la base de connaissance notre règle apparait :



Enfin, lorsque l'on tape 5 on quitte le programme.

IV- Commentaires

Nous sommes satisfaits des menus et l'interaction que l'on peut avoir avec ceux-ci. Le moteur d'inférence fonctionne comme demandé dans le sujet, on lui transmet une base de fait et en ressort une conclusion. Le seul défaut que nous avons pu constater dans ce que l'on a fait se trouve dans la fonction qui permet d'ajouter une règle à la base de connaissance : en effet, nous nous sommes rendu compte au dernier moment que l'on ne vérifiait pas si la règle que l'on voulait intégrer faisait déjà partie de la base de connaissance ou pas. La conséquence est que l'on peut mettre plusieurs fois la même règle dans la base de connaissance.

D'un point de vue plus global, ce projet a été pour nous une bonne manière de tester nos compétences en programmation et en algorithmie, et plus précisément notre capacité à concevoir des algorithmes abstraits. Malgré la situation actuelle plutôt atypique et discordante avec le travail, nous avons fait tout notre possible pour aller le plus loin dans ce projet, et cela a été pour nous un défi imposant. Nous sommes plutôt fiers de ce que nous avons pu faire, c'est-à-dire tout ce à quoi nous avions pensé lorsque nous avons commencé le projet.