# Rapport Projet LO21 : système expert

Table des matières

[Rapport Projet LO21 : système expert 1](#_Toc60746272)

[Cahier des charges 1](#_Toc60746273)

[Contextes et définition du problème 1](#_Toc60746274)

[Objectif du projet 2](#_Toc60746275)

[Périmètre du projet 2](#_Toc60746276)

[Description fonctionnelle des besoins 2](#_Toc60746277)

[Contrainte/ Normalisation et documentation du code 3](#_Toc60746278)

[Réalisation pratique 3](#_Toc60746279)

[Structures de donnée et choix de conception 3](#_Toc60746280)

[Algorithme des sous programes : 6](#_Toc60746281)

[Choix d’implémentation 11](#_Toc60746282)

[Lecture et écriture dans les ficher 11](#_Toc60746283)

[La gestion des propositions en mémoire 11](#_Toc60746284)

[Fonctionnement du programme 12](#_Toc60746285)

Cahier des charges

Contextes et définition du problème

Le sujet de ce projet est de réaliser un système expert fonctionnel. D'après la définition donnée par la page Wikipédia sur ce sujet, un système expert est : "un logiciel capable de répondre à des questions, en effectuant un raisonnement à partir de faits et de règles connues."(source : https://fr.wikipedia.org/wiki/Syst%C3%A8me\_expert). Pour cela le système expert va se comporter comme un humain expert dans un domaine d'activité comme la médecine ou l'automobile. En effet pour établir un diagnostic ou une analyse l'humain va faire des observation et à partir de ces observation il va appliquer des règles qu'il a apprise lors de sa formation ou par son expérience pour en tirer des déduction, comme par exemple la maladie dont souffre un patient dans le cas d'un médecin ou le problème qu'a la voiture pour un garagiste. Le système expert est donc composé de trois élément principaux, le premier est la base de connaissances. La base de connaissance permet de stocker toutes les règle qui permettront au programme de faire des déductions. Le second élément est la base de fait, elle permet de stocker les propositions qui sont considérée comme vrai au lancement du programme. Le dernier élément est le moteur d'inférence, c'est lui qui est chargé à partir de la base de connaissances et de la base de fait de déduire toute les proposition qui sont vraies.

Objectif du projet

L'objectif de ce projet est de développer une programme capable de remplir les fonction d'un système expert. Le projet doit pouvoir permettre à l'utilisateur de créer ou importer une base de connaissances, puis après avoir répondu à des questions permettant d'établir la base de fait de connaitre les résultats qui peuvent en être déduit. On souhaiterait permettre à un utilisateur n'ayant pas de connaissances en informatique de pouvoir utiliser notre projet une fois qu'il a été configuré en fonction de la situation.

Périmètre du projet

Le projet devra être réaliser en langage C et s'exécuter dans la console. Il sera au maximum portable c'est à dire capable de fonctionner sous différent systèmes d'exploitation (linux et Windows au moins). La vitesse d'exécution et la place prise en mémoire sera optimiser afin d'obtenir un programme le plus efficace possible. Nous limiterons les règle contenue dans la base de connaissances à des règle du type "A et B et ... implique C" et nous utiliserons uniquement les implication directe, pas de réciprocité ou de contraposée.

Description fonctionnelle des besoins

* Définir des structure de données abstraite pour les objets suivant : Proposition, Prémisse, Règle, Base de connaissance, Base de fait.
* Implémenter les sous programmes permettant de manipuler les structure
* Permettre à l'utilisateur de créer et de stocker une base de connaissances
* Permettre à l'utilisateur de répondre au question et de créer la base de fais en fonction des réponses.
* Implémenter le sous-programme qui remplit la fonction du moteur d'inférence
* Afficher à l'utilisateur le contenue de la base de connaissance et les résultat du moteur d'inférence

Contrainte/ Normalisation et documentation du code

* Le programme doit respecter la norme C99
* Chaque fonction ou structure doit être documentée selon la formalisation imposée par doxigène. Ainsi une documentation du code sera générée par doxigène
* Les fonction seront dans la mesure du possible implémenté de manière récursive
* Les variable sont en normalisation CamelCase et les nom de structure commences par une majuscule
* Le projet est versionner avec git et héberger sur git hub
* La génération des make file et la compilation du programme ce fait avec Cmake
* Le temps d'exécution et l'espace mémoire utilisé par le programme devront rester acceptable pour que le programme soit utilisable sur le plus grand nombre de machine

## Réalisation pratique

### Structures de donnée et choix de conception

#### Règle

* La structure de donnée abstraite qui va nous permettre représenter une Règle est composée de 2 sous éléments : la prémisse et la conclusion.
  + Premisse est du type Premisse que nous définirons ci-dessous.
  + Conclusion est de type pointeur sur une Proposition que nous définirons ci-dessous.

typedef struct Regle

{

    Premisse       premisse;

    Proposition    \*conclusion;

}Regle;

#### Premisse

* La structure de donnée abstraite qui va nous permettre de représenter une Premisse est une liste chainée de Proposition. La Premisse est donc un pointeur sur le premier élément de la liste qui est de type PremisseElem que nous définirons ci-dessous.

typedef PremisseElem\* Premisse;

* La structure de donnée abstraite PremisseElem est constituée de 2 sous variables : la valeur de l’élément et le pointeur sur l'élément suivant de la liste chainée :
  + valeur est de type pointeur sur une Proposition que nous définirons ci-dessous
  + elemSuivant est un pointeur sur PremisseElem

typedef struct PremisseElem

{

    Proposition         \*valeur;

    struct PremisseElem \*elemSuivant;

}PremisseElem;

Nous avons choisi d’utiliser la structure d’une liste chainé pour stocker les prémisse car nous ne savons pas à l’avance combien d’élément il y aura dans la prémisse et on à besoin durant l’exécution du programme de pouvoir rajouter ou supprimer un ou plusieurs éléments à la liste. Ainsi avec ces contraintes utilisé un tableau n’était pas possible et la liste chainé est parfaitement adapté.

#### Proposition

* La structure de donnée abstraite qui va nous permettre de stocker les Proposition est composée de 2 sous variable : la description et la validité :
  + description est une chaine de caractère (tableau de char en C) qui correspond à ce que veut dire la proposition en lagunage naturel
  + validite est un booléen qui vaut true si la proposition est vrai et false si la proposition est fausse ou si son état est inconnue.

typedef struct Proposition

{

    char \*description;

    bool validite;

}Proposition;

Nous avons beaucoup hésité sur la façon de représenter les propositions et nous avons améliorer la façon de le faire au cour du projet, mais nous avons choisi de le faire ainsi pour plusieurs raison. D’abord afin que l’on puisse afficher à l’utilisateur la valeur de la proposition en langage naturel on la stocke dans un tableau de char, mais dans la proposition il n’y à que le pointeur vers le tableau ce qui réduit la place prise en mémoire par cette structure et qui permet d’avoir des tableau de char de taille variable. L’autre sous variable permet lors de l’exécution du moteur d’inférence de savoir quelle proposition est juste ou fausse cela permet d’éviter que le moteur d’inférence détruise la base de connaissance lors de son exécution comme cela était le cas au début, plutôt que de supprimer dans la prémisse des règles les proposition qui sont vrai on se contente de définir leur validité à vrai et on limite ainsi grandement l’effet de bord du moteur d’inférence.

#### Base de connaissances

La base de connaissances est une liste chainée de Règle, nous avons déjà définit la structure de donnée règle ainsi que toutes les fonction qui lui sont associer, il faut maintenant rajouter une structure permettant de stocker un élément de cette liste chainée et la structure qui fera office de tête de liste

* La structure de donnée abstraite BDConnaissancesElem permet de représenter un élément de la liste chainée de règles. Elle est composée de deux sous-variables, un pointeur sur Règle ainsi qu’un pointeur sur l'élément suivant dans la liste chainée

typedef struct BDConnaissancesElem

{

    Regle\* valeur;

    struct BDConnaissancesElem\* suivant;

}BDConnaissancesElem;

* La structure de donnée abstraite BDConnaissances permet de représenter la base de connaissance, c'est un pointeur sur le premier élément de la liste chainé de règle.

typedef BDConnaissancesElem\* BDConnaissances;

Nous avons choisi de représenter la Base de connaissance par une liste chainé de règle car nous ne connaissons pas à l’avance le nombre de règle que va contenir la cette variable et nous avons besion de pourvoir ajouter ou supprimer un ou plusieurs élément au cour de l’éxécution du programme. De plus cette structure de donné permet d’implémenter très facilement des fonction récursive sur la structure ce qui nous à été demandé par le surjet.

### Algorithme des sous programes :

* newRegle : créer une nouvele règle vide
  + donnée : rien
  + résultat : un pointeur sur la règle qui vient d'être créée

fonction : newRegle(): \*Regle

    Soit nouvelRegle une Regle

    nouvelRegle.premise <-- NULL

    nouvelRegle.conclusion <-- NULL

    newRegle <-- nouvelRegle

fin fonction

* deleteRegle : supprimer une règle et toutes ces composantes
  + donnée : regleToDelete, un pointeur sur règle, la règle que l'on veut supprimer
  + résultat : supprime de la mémoire la règle

procédure : deletRegle(Regle \*regleToDelete)

    deletePremisse(regleToDelete.premisse)

    libère(regleToDelete)

fin procédure

* deletePremisse : supprimer de la mémoire une prémisse
  + donné : un pointeur sur la prémisse à supprimer
  + résultat : supprime de la mémoire la prémisse

procédure : deletePremisse(Premisse prem)

    si (premisseNonVide(prem))

        deletePremisse(suivant(prem))

        libère(prem)

    fin si

fin procédure

* Ajouter une Proposition à la prémisse d'une règle en queue
  + donnée : la proposition à ajouter et la Règle à laquelle l'ajouter
  + résutat : ajoute la proposion à la liste chainé en queue en faisant le lien entre les élément

fonction :  insertTailPremisseRegle(Regle\* regle, Proposition \*prop)

    premisse(regle) <-- inseretTailPremisse(premisse(regle), prop)

    insertTailPremisseRegle <-- regle

fin fonction

* insertTailPremisse : ajouter une proposition à une premisse en queue
  + donnée : la premisse à laquel on veut ajouter la proposition et la proposition à ajouter
  + résultat : renvoie la valeur de la premmise et ajoute la propositon à la prémisse

fonction : insertTailPremisse(Premisse prem, Proposition\* prop)

    si premisseVide(prem)

        soit newElem un PremisseElem

        valeur(newElem) <-- prop

        elmSuivant(newElem) <-- NULL

        insertTailPremisse <-- newElem

    sinon si premisseVide(elemSuivant(newElem))

        soit newElem un PremisseElem

        valeur(newElem) <-- prop

        elmSuivant(newElem) <-- NULL

        elemSuivant(prem) <-- newElem

        instertTailPremisse <-- prem

    sinon

        insertTailPremisse(elemSuivant(prem), prop)

        insertTailPremisse <-- prem

    fin si

fin fonction

* addConclusion : Créer la conclusion d'une règle
  + donnée : la règle que l'on veut modifier et la proposition à donner à la conclusion
  + résultat : modifie ou initialise la conclution de la règle

procédure : addConclusion(Regle\* regle, Proposition\* prop)

    conclusion(regle) <-- prop

fin procédure

* propositionDansPremisse: Tester si une Proposition appartient à la prémisse d'une règle récursivement
  + donnée : un pointeur sur le premier élément de la prémisse dans laquel on veut rechercher, la propositon à rechercher
  + résulat : renvoie vrai si la proposion à été trouvée dans la prémisse et faux sinon

fonction : propositionDandPremisse(Premisse prem, Proposition\* prop)

: booléen

    si premisseVide(prem)

        propositionDansPremisse <-- faux

    sinon si valeur(prem) = prop

        propositionDansPremisse <-- vrai

    sinon

        propostionDansPremisse <--

        propostionDansPremisse(elemSuivant(prem))

    fin si

fin fonction

* rechercheSupprimePremisse :Supprimer une Proposition de la prémisse d'une règle
  + donnée : la prémisse dans laquelle on veut supprimer, la proposition à supprimer
  + résultat : supprime la proposions si elle a été trouvée et renvoie la prémisse sur laquelle on travail

fonction : rechercheSupprimePremisse(Premisse prem,

Propostion\* prop) : Premisse

    si premisseVide(prem)

        rechercherSupprimePremisse <-- NULL

    sinon si premisseVide(elemSuivant(prem)) et description(prop) =

description(valeur(prem))

        libère prem

        rechercherSypprimePremisse <-- NULL

    sinon si description(prop) = description(valeur(elemSuivant(prem)))

        Soit toDelete une Premisse

        toDelete <-- elemSuivant(prem)

        elemSuivant(prem) <-- elemSuivant(toDelete)

        libérer toDelete

        rechercherSupprimePremisse <--  prem

    sinon

        rechercheSupprimePremisse(elemSuivant(prem), prop)

        rechercheSupprimePremisse <-- prem

    fin si

fin fonction

* reglePremisseIsEmpty :Tester si la prémisse d'une règle est vide
  + Donées : un pointeur sur la règle à tester
  + Resultat : renvoie vrai si la prémisse est vide et faux sinon

fonction : reglePremisseIsEmpty(Regle \*regle) : booléen

    reglePremisseIsEmpty <-- premisseIsEmpty(premisse(regle))

fin fonction

* returnHeadPremisse : Accéder à la proposition ce trouvant en tête d'une prémisse
  + donnée : la premisse sur laquelle on veut travailler
  + renvoie : pointeur sur la proposition en tête de la premisse

fonction : returnHeadPremisse(Premisse prem) : Proposition \*

    returnHeadPremisse <-- valeur(prem)

fin fonction

* conclutionRegle : Accéder à la conclusion d'une règle
  + donnée : un pointeur sur la règle dont on veut connaitre la conclusion
  + résultat : renvoie un pointeur sur la proposition qui est la conclusion de la règle

fonction : conclutionRegle(Regle\* regle) : Proposition\*

    conclutionRegle <-- conclution(regle)

fin fonction

* isEmptyBDC : tester si la base de connaissances est vide
  + donnée : la base de connaissances que l'on veut tester
  + résultat : renvoie vrai si la basse de connaissance est vide et faux sinon

fonction : isEmptyBDC(BDConnaisances bdc)

    si bdc = NULL

        isEmptyBDC <-- vrai

    sinon

        isemptyBDC <-- faux

fin fonction

* addHeadBDC : insérer en tête une nouvelle règle
  + donnée : la base de connaissance avec laquelle on travail et un pointeur sur la règle à ajouter
  + résultat : ajoute la règle à la base de connaissance et renvoie la base de connaissance

fonction : addHeadBDC(BDConnaissance bdc, Regle regle): BDConnaissances

    soit newElem un pointeur sur BDConnaissanceElem

    valeur(newElem) <-- regle

    suivant(newElem) <-- bdc

    addHeadBDC <-- newElem

fin fonction

* deleteHeadBDC : supprimer en tête une règle
  + donnée : la base de connaisance dont on veut supprimer le premier élément
  + supprime l'élément en tête de la liste chainée et renvoie la base de connaissance

fonction : deleteHeadBDC(BDConnaissances bdc) : BDConnaissances

    si isEmptyBDC(bdc) = faux

        soit toDeleteNext un pointeur sur un BDConnaissancesElem

        toDeleteNext <-- suivant(bdc)

        deleteRegle(valeur(bdc))

        libère bdc

        deleteHeadBDC <-- toDeleteNext

    sinon

        deleteHeadBDC <-- NULL

fin fonction

* returnHeadBDC : Accéder à la règle se trouvant en tête de la base
  + donné : la base de connaissances dont on veut connaitre la prémière règle
  + résultal : renvoie un pointeur sur la règle en tête

fonction : returnHeadBDC(BDConnaissances bdc) : Regle\*

    returnHeadBDC <-- valeur(bdc)

fin fonction

* moteurDInference : recherche à partir de la base de vérité et de la base de connaissances les propositions qui sont vraie
  + donnée : base de connaissance (une liste chainée règle), base de vérité (liste chainée de proposition)
  + résultat : renvoie une liste chainée de proposition qui ont été déduite grâce à l'exécution de cette fonction

fonction : moteurDInference(Premisse baseVerite, BDConnaissances bdc)

 : Premisse

    Soit conclusion une Premisse

    Pour chaque proposition dans baseVerite

        Pour chaque regle dans bdc

            si (propositionDansPremisse(premisse(regle), proposition))

                setValidite(proposition, vraie)

                si (isPremisseTrue(premisse(regle))

                    addTailPremisse(conclusion, conclusion(regle))

                fin si

            fin si

        fin pour

    fin pour

    moteurDInference <-- conclusion

fin fonction

### Choix d’implémentation

Lecture et écriture dans les ficher

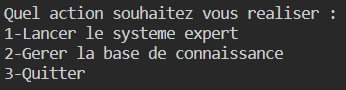
* ReadBDC : lis un fichier CSV contenant plusieurs règles et les ajoute à la base de connaissance du système expert.
  + donnée : base de connaissance (une liste chainée de règle), le chemin vers le fichiers et la liste de toutes les propositions.
  + résultat : renvoie un pointeur sur la base de connaissance remplie
* WriteBDC : gère l’acquisition de nouvelles règles qui vont être ajoutée à la liste chainée de la base de connaissance et l’écris dans le fichier CSV permettant la création de la base de connaissance.
  + donnée : base de connaissance (une liste chainée de règle), le chemin vers le fichiers et la liste de toutes les propositions.
  + résultat : renvoie un pointeur sur la base de connaissance auquel a été ajouté une nouvelle règle.

### La gestion des propositions en mémoire

Beaucoup de fonction du programme ont besoin de créer des propositions que ce soit pour créer une règle ou ajouter une proposition dans la base de vérité et nous avons remarqué que bien souvent beaucoup de proposition avaient la même description (donc étaient égale) ce qui prend de la place inutilement en mémoire en rend les opération sur les proposition plus complexe car pour savoir si deux proposition sont les même il faut comparer les chaine de caractère de leur description. Nous avons donc choisi de changer la manière donc les proposions était utilisé de façon à que chaque proposition n’apparaisse qu’une unique fois dans la mémoire de l’ordinateur. Pour ce faire on crée la variable listProp qui est une liste chainée de proposition qui va contenir toutes les proposition qui sont créée au cour de l’exécution du programme. A chaque fois qu’un sous-programme désire créer une proposition la méthode est la suivante, d’abord on vérifie si une proposition aillant la même description n’existe pas déjà listProp, si c’est le cas on renvoie la à la fonction appelante le pointeur vers cette proposition déjà existante et le tour est joué. A l’inverse si la proposions n’existe pas alors la on crée une nouvelle proposition et on l’ajoute immédiatement à la liste chainée listProp. Ainsi le nombre de proposition est très limité (cela réduit par 5 l’espace mémoire occupé par les proposition en moyenne) mais surtout il est alors beaucoup plus simple de savoir si deux pointeur sur proposition sont égaux car il suffit de tester si l’adresse sur lequel ils pointent est la même se qui réduira énormément la complexité du test par rapport à une comparaison des chaine de caractère. En conclusion cette nouvelle implémentation que nous avons mis en place nécessite un peu plus de temps pour créer des proportion mais une fois cela fait il devient alors beaucoup plus rapide et moins gourmand ne mémoire vide de travailler avec comme ça pourrait arriver si on à une grande base de connaissances qui contient beaucoup de règles.

Fonctionnement du programme

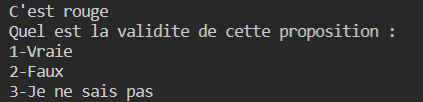
Lorsque l’utilisateur lance le programme il arrive sur le menu principal :



A ce moment le programme demande à l’utilisateur de choisir l’action qu’il veut faire. L’utilisateur doit entrer un chiffre entre 1 et 3, l’acquisition est contrôlée sans message d’erreur. Puis le programme lance le sous-programme désiré par l’utilisateur, et nous allons détailler par la suite comment ils fonctionnent.

#### Lancer le système expert

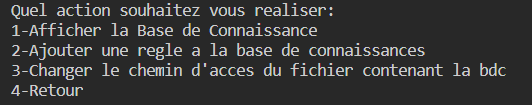
La fonction systemExpert est le cœur du projet c’est elle qui réalise tous les attendu du cahier des charge. Pour commencer la variable de la Base de connaissance ainsi que celle qui stockera toute les propositions (listProp) sont créées. Puis la fonction ReadBDC se charge de construire la base de connaissances à partir de toute les règles qui sont stocker dans le fichier que l’on passe en paramètre. Ensuite la fonction genereBDVerite se charge de définir la validité des proposition, pour ce faire on pose la question à l’utilisateur :



Afin que l’utilisateur n’ait pas trop de question au quelle répondre et car il ne sert à rien de lui demander la validité de la proposition que notre programme est censé déterminer on exclu de cette procédure de question toute les proposition qui appartienne à la conclusion d’une règle de la base de connaissance. Pour finir on lance le moteur d’inférence, et on affiche le résultat de ce dernier pour que l’utilisateur sache quelles proposition ont été déterminées comme vraies. Avant que la fonction ne se termine il faut bien pensé à libérer l’espace mémoire utiliser par nos structure de donné pour éviter les fuites de mémoires. Pour ce faire on utilise deleteAllBDC qui déconstruit la base de connaissance et libère la mémoire utilisée par les règles. Puis deletePremisse qui se charge de déconstruire la conclusion mais sans détruire les propositions et enfin la fonction deletePremisseProposition se charge de libérer toutes les propositions qui sont dans listProp. Comme nous l’avons expliqué précédemment toutes les propositions sont enregistré dans cette variable ainsi on est sûr de bien tout avoir libérer.

#### Gérer la base de connaissance

Lorsque l’utilisateur choisi l’option 2 sur le menu principal il est envoyé sur ce sous menu qui va lui permettre de faire des opérations sur la base de connaissances.



Si l’utilisateur choisi d’affiche la Base de connaissance, alors la base de connaissance est construite à l’aide de la fonction readBDC à partir des informations contenues dans le fichier associé. Voici comment fonctionne cette fonction, tout d’abord le fichier doit être normalisé, c’est pourquoi pour nos test nous utilisons des fichier CSV, ils peuvent être facilement remplis en utilisant un tableur et utilise un séparateur bien défini, ce qui nous à aidé à séparer les différentes propositions. Ainsi on obtient une structure lisible par la fonction qui va utiliser les points-virgules pour séparer le texte. Dans ce fichier chaque ligne est une règle de la base de connaissance, et chaque règle est écrite comme ceci : nombre de proposition ; conclusion de la règle ; proposition numéro 1 ; proposition numéro 2 ; … ; proposition numéro n; . Nous avons fait le choix d’écrire en premier le nombre de propositions pour nous faciliter le travail lors de l’allocation dynamique de nos variables, c’est-à-dire que le nombre de proposition va déterminer l’espace en mémoire nécessaire pour stocker la lecture du fichier avant de l’ajouter dans la liste chainée qui constitue notre base de connaissance. Sachant que le nombre de propositions est variable, écrire la conclusion à la suite des propositions n’était pas viable, cela aurait été plus compliqué de savoir quand a lieux la lecture de la conclusion. C’est pourquoi la conclusion vient juste après le nombre de propositions. Ce que la fonction va faire c’est qu’elle lire ligne par ligne jusqu’à la fin du fichier, après avoir récupéré le nombre de propositions elle va ligne la conclusion et toutes les propositions qui vont être stockée dans une variable de type char. Ainsi cette variable va être découpée en fonction des points-virgules trouvés et du nombre de propositions. Dans un premier temps la conclusion va être extraite et stockée dans une autre variable propre à la conclusion et ensuite toutes les propositions vont être extraites est stockée dans des variables qui seront au nombre indiqué au début de la ligne. Après le traitement du texte lu par la fonction, on envoie directement toutes les variables à la fonction qui ajoute des règles dans la base de connaissance addRegleBDC(), puis on libère l’espace en mémoire des variables utilisées. Ces opérations vont être répétée tant que la lecture du fichier ne sera pas terminée. Une fois la base de connaissance crée on l’affiche avec la fonction afficheBDC. Puis on n’oublie pas de libérer l’espace en mémoire utilisé par la base de connaissance et listProp comme nous l’avons vu précédemment.

Si l’utilisateur choisi d’ajouter une règle à la base de connaissances alors la fonction writeBDC se lance. Cette fonction marche en 4 étapes, l’acquisition par l’utilisateur, le traitement du texte entré, l’écriture dans le fichier et l’ajout à la base de connaissance. La première étape consiste à demander à l’utilisateur la règle qu’il veut entrer, pour cela on va en amont lui demander le nombre de proposition de sa règle, cela va déterminer le nombre de fois ou l’algo va demander l’acquisition d’une proposition. Lorsque ce nombre est déterminée on va demander à l’utilisateur de rentrer ces propositions qui font être vérifiées par la fonction verificationPropositionAvecMessage(), celle-ci va traiter le texte envoyé par l’utilisateur et afficher soit une message d’erreur soit rien quand la saisie est correcte. La vérification de saisie autorise donc tous les caractères de l’alphabet en minuscule ou majuscule, les nombres mais également les espaces, les apostrophes et les tirets, mais cependant il interdit les accents et tout autres caractères. De plus cette fonction vérifie la longueur du texte saisi, si le texte est trop long la fonction renvoie en sortie un code d’erreur qui va permettre de finir la lecture des éléments qui n’ont donc pas été lus pour ainsi éviter des bugs dans les prochaines saisies. Cette fonction a donc 4 états de réponse, le texte possède des caractères in

Pour que les fonction readBDC et writeBDC puisse fonctionner correctement elles besoin d’un ficher dans lequel lire et écire les informations avec la bonne structure, par défaut le chemin de ce fichier est ../../file/bdc.csv. Mais selon les plateformes et les manière de compiler notre projet il arrive que le ficher n’ai pas tout à fait ce même chemin par rapport à l’exécutable ou alors que l’utilisateur souhaite utiliser un tout autre fichier enregistré à un endroit quelconque de son ordinateur. C’est pourquoi nous avons rajouter une fonctionnalité permettant de modifier le chemin d’accès du fichier que l’on souhaite utiliser afin de rendre notre programme plus flexible. Ainsi lorsque l’utilisateur change le chemin du fichier le nouveau chemin est passer en paramètre au fonction readBDC et writeBDC travaillent alors avec cet autre ficher. Néanmoins pour cette acquisition on ne contrôle pas si l’utilisateur entre le chemin d’un fichier ou notre programme aura les droits d’accès ou même si ce chemin est cohérent. Il est donc de la responsabilité de l’utilisateur de ne pas rentrer quelque chose de faux à ce moment-là. De plus dans le cas de l’écriture dans le fichier, il est ouvert dans un mode de lecture qui s’il ne trouve pas le fichier dans l’ordinateur, un nouveau fichier sera créé dans le dossier ou se trouve l’exécutable et l’utilisateur pourra tout de même crée sa base de connaissance.

#### Quitter

Cmake