TP3 : ELABORATION D'UN SYSTEME EXPERT D'ORDRE 0+

TABLE DES MATIERES

I. INTRODUCTION

Nous arrivons au dernier TP de l'année. Pour celui-ci, nous devons conceptualiser et implémenter un Système Expert (SE) d'ordre 0+ sous la forme de notre choix. Ce SE doit permettre de répondre à des questions afin de résoudre un problème sur un sujet que nous devons choisir.

Afin de conceptualiser un modèle ludique et intéressant pour ce projet, nous avons étudier les idées proposer les années d'avant : un Cluedo, un entraineur personnalisé pour le sport et bien d'autres encore. A la lumière de ces sujets et en mettant en commun nos centres d'intérêts, nous nous sommes décidés à élaborer un jeu. Ce jeu devait pouvoir être compatible avec la structure d'un système expert et ce dernier se devait d'être pertinent afin d'avoir une solution intéressante. Nous nous sommes donc dirigés vers un jeu de société bien connu de tous : le « Qui est-ce ? ». Néanmoins afin d'ajouter un aspect plus personnel et afin d'élargir le cercle de personnages, nous nous sommes tournés vers l'univers de Mario pour réaliser ce jeu.

Ce rapport détaillera donc les différentes étapes de conception de ce jeu ainsi que des différents algorithmes implémentés permettant son fonctionnement. Nous apporterons une attention toute particulière à la documentation de nos choix de règles ainsi qu'à l'explication de l'architecture de notre programme.

II. MISE EN SITUATION

1) Recherche des sources

Maintenant que nous avons une idée, il faut mettre en place le support que nous allons utiliser pour réaliser ce jeu. Nous avons trouvé sur le forum reddit.com une image adaptée pour ce jeu. Elle contient une liste de vingt personnages issus de l'univers de Super Mario Party. Au premier coup d'œil, tous ces personnages ont l'air assez différents et avec des caractéristiques intéressantes pour les différencier.

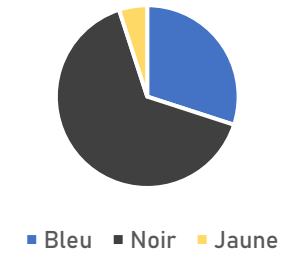
Figure 1 : Planche de Qui-est-ce? source: https://www.reddit.com/r/MARIOPARTY/comments/e9l22a/what_is_your_dream_mp_roster_you_can_choose_as/

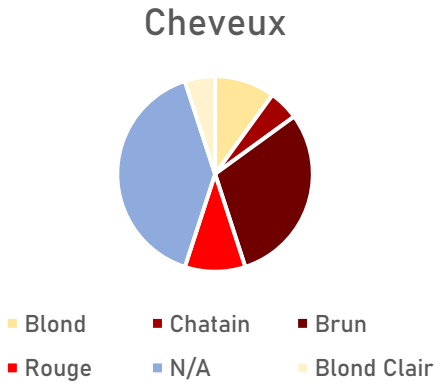
2) Traitement des données

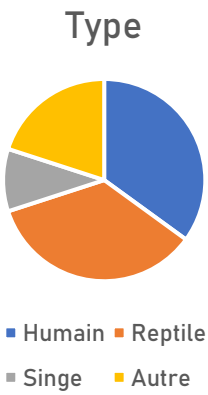
Cette planche ainsi trouvée peut s'apparenter à un jeu de données brut qu'il faut ensuite analyser et organiser. L'objectif ici était de trouver des caractères physiques ou issus de l'univers de Mario qui permette de répartir les personnages dans des classes de proportion plutôt homogène (à quelques exceptions près) afin que les questions qui pourront être posé par le joueur et l'IA soient suffisamment discriminantes pour supprimer rapidement des personnages. Les caractères choisis devront aussi permettre une fois croisés entre eux de déterminer chaque personnage individuellement et cela sans ambiguïté.

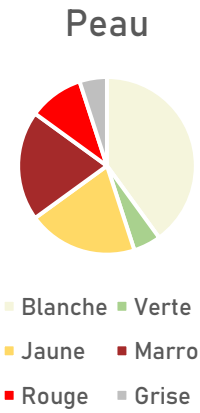
Afin de faire cette tâche, nous avons choisi arbitrairement des critères qui semblaient assez discriminant. Nous les avons ensuite regroupés dans un fichier Excel afin de voir si ceux-ci étaient pertinent. Une fois cela fait, nous avons supprimé les caractères qui dissociait les personnages de manières trop hétérogènes. Cela nous permet d'avoir une liste de critères relativement restreinte afin de ne pas surcharger notre système expert.

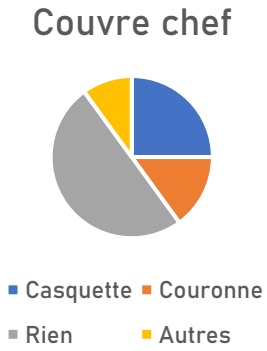
Voici la liste des critères avec leur répartition au sein de la population :

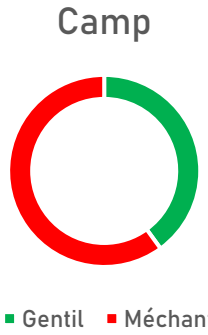

Couleur des yeux
Bleu
Noir
Jaune

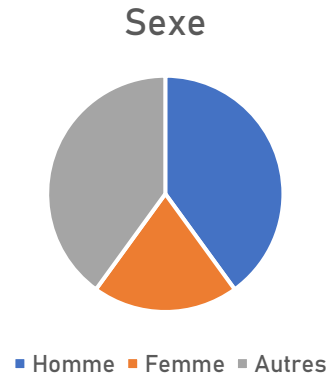

Cheveux
Blond
Chatain
Brun
Rouge
N/A
Blond Clair


Type
Humain
Reptile
Singe
Autre

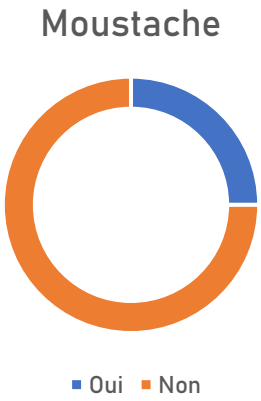

Peau
Blanche
Verte
Jaune
Marron
Rouge
Grise


Couvre chef
Casquette
Couronne
Rien
Autres


Camp
Gentil
Méchant


Sexe
Homme
Femme
Autres


Dents
Visible
Invisible


Moustache
Oui
Non


Mince
Oui
Non

Afin de savoir si ces caractères permettaient de déterminer chacun des personnages de façons unique, nous aurions pu organiser les personnages dans une table SQL en fonction de leurs critères afin de tester si l'ensemble de ces critères représentaient une clé de la table. Néanmoins, vu le nombre assez petit de personnages, nous avons décider de faire cette étape à la main en voyant si chacun des personnages n'avait pas la même liste de critère qu'un autre ce qui est relativement rapide une fois les critères attribués aux personnages.

III. STRUCTURE DU SYTEME EXPERT

1) Rappel et vérification

Pour ce TP, le système expert à élaborer devait être de l'ordre 0+. C'est-à-dire que ce dernier ne devait utiliser que des faits booléens et des relations simples de type <relation, entité, valeur>. De par les critères précisés précédemment, notre SE était destiné à être d'ordre 0+ car on retrouve aussi bien des faits booléens (Moustache ou Mince…) que des relations simple et non booléenne (Type, Cheveux…).

Selon le cours, un SE se compose en règle générale de trois éléments :

- Le moteur d'inférence
- La base de connaissances
- L'interface utilisateur

Ce SE est alimenté par un expert du domaine qui, par le biais d'un cogniticien va permettre de constituer une base de connaissances. Dans notre cas, le rôle de ces deux protagonistes fut rempli par nous lors de l'étape de traitement des données expliquée précédemment. Le reste du système sera détaillé ci-après et sera implémenter entièrement en LISP en utilisant les algorithmes vus en cours et en TD.

2) Structure du SE en détail

Pour réaliser notre jeu, il fallait que le système expert remplisse deux fonctions :

- Deviner un personnage
- Faire deviner un personnage

Afin de réaliser ces deux tâches, nous avons décider d'implémenter deux algorithmes déjà étudier pendant ce semestre.

Pour le premier cas, nous utiliserons le chaînage avant. Ainsi, l'utilisateur à la demande du programme va construire une base de fait qui permettra à l'IA de trouver le personnage voulu.

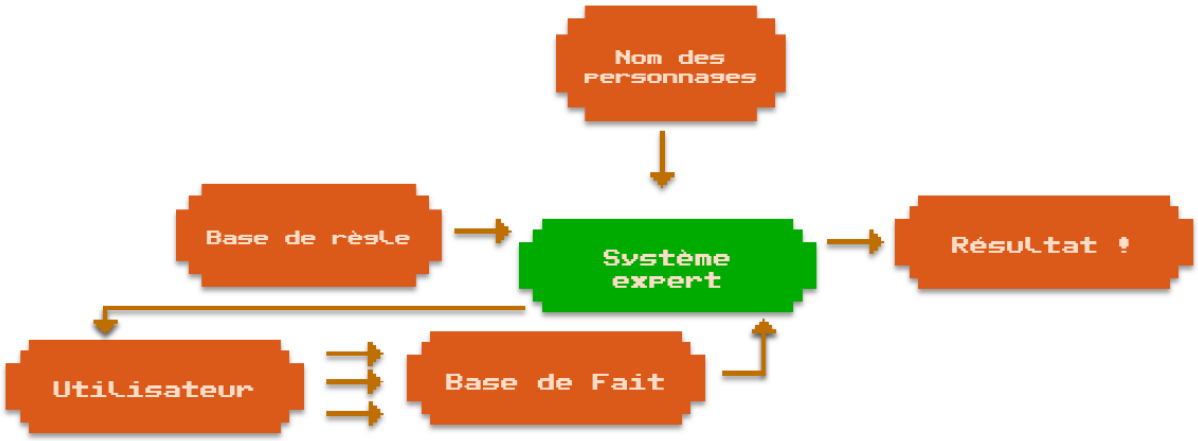


Figure 2 : Schéma de l'architecture en chaînage avant

Dans le deuxième cas, nous avions commencer à réfléchir à un chainage arrière. En effet, cela représentait à première vue une solution efficace pour répondre à ce problème, néanmoins, nous nous sommes vite rendu compte que ce cas pouvait être implémenter beaucoup plus rapidement et efficacement en utilisant une méthode hybride.

Notre programme réagit ainsi pour ce cas : l'utilisateur pourra poser des questions au SE qui se chargera de vérifier si la réponse est positive ou négative. Dans ce cas-ci, le personnage que l'utilisateur devra deviner sera choisi par le programme de manière aléatoire.

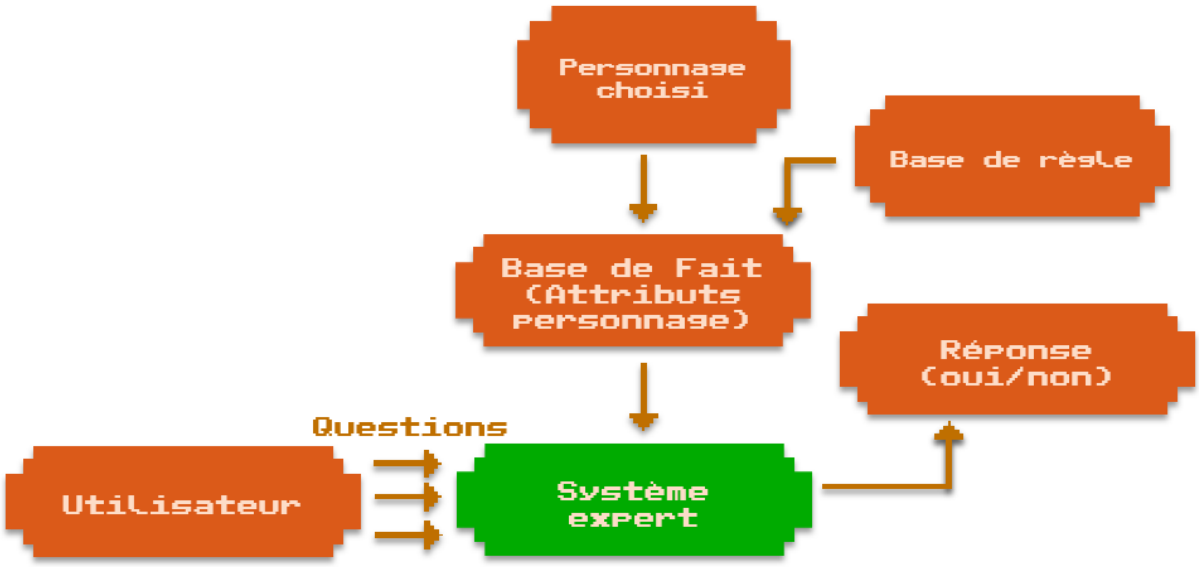


Figure 3 : Schéma de l'architecture du SE pour deviner le personnage de l'IA

IV. PROGRAMMATION DE L'APPLICATION

1) Le chaînage avant

Afin de faire deviner un personnage à l'IA, le raisonnement le plus intuitif est d'utiliser un moteur d'inférence en chaînage avant. Pour cela, nous avons besoins de plusieurs fonctions de services telles que :

- *getPremisses* : renvoie les prémisses d'une règle.
- *getCcl* : renvoie la conclusion d'une règle.
- *liste-membre-p* : renvoie *T* si une sous liste appartient à une liste NIL sinon *ex :*
  - *(liste-membre-p '(couvre_chef casque) '((cheveux chatains) (couvre_chef casquette)))* renvoie *NIL*
  - *(liste-membre-p '(peau (not jaune)) '((peau verte) (espece reptile)))* renvoie *T*
- *satisfaites-p* : renvoie *T* si les prémisses d'une règle sont satisfaites

Ainsi, voici la fonction de chaînage avant :

```
(defun chainage-avant (bdf bdr attributes &optional multiplayer)
  (if (assoc 'nom bdf)
      (progn
        (format t "~%Le personnage est : ~a" (cadr (assoc 'nom bdf)))
        (return-from chainage-avant))
    (let ((bdf_initiale bdf))
      (dolist (regle bdr)
        (let ((premisses (getPremisses regle)) (conclusion (getCcl regle)))
          (if (satisfaites-p premisses bdf)
              (if (not (liste-membre-p conclusion bdf))
                  (push conclusion bdf)))))
      ;(print bdf)
      (when (and (null attributes) (not (assoc 'nom bdf)))
        (error "Impossible de déterminer le personnage"))
      (if (not multiplayer)
          (progn
            (when (equal bdf bdf_initiale)
              (setq bdf (deviner-personnage-ia bdf attributes)))
            (chainage-avant bdf bdr attributes))
        (if (not (assoc 'nom bdf))
            (if (not (equal bdf bdf_initiale))
                (chainage-avant bdf bdr attributes multiplayer)
              (setq bdf (deviner-personnage-ia bdf attributes)))
          bdf))
        )))
```

Figure 4 : Fonction du chaînage avant

Cette fonction prend en paramètre la base de faits, la base de règle, une liste contenant tous les attributs possible ainsi qu'un argument optionnel « multiplayer » (utile pour la partie 3). La fonction commence par vérifier si « nom » est présent dans la base de fait (si un personnage a été trouvé). Si ce n'est pas le cas, on fait une copie de la base de fait puis on parcourt les règles de la base de règle une à une. La fonction *satisfaites-p* permet ainsi de vérifier si toutes les prémisses d'une règle sont présentes dans la base de fait. Si c'est le cas, alors on peut ajouter la conclusion de la règle à la base de fait (si celle-ci n'est pas déjà présente). Une fois que toute les règles ont été parcourues, on vérifie que la liste d'attributs n'est pas vide. Si c'est le cas et que le personnage n'a pas été trouvé, alors la fonction est dans l'incapacité de déterminer le personnage. Dans le cas où la recherche peut continuer, on distingue deux cas. Si la partie est à deux joueurs (voire partie 3), alors l'IA commence par vérifier si elle peut essayer de trouver de nouvelles conclusions dans la base de règles avant de finir son tour (en vérifiant l'égalité entre la base de fait et sa copie faite au début de la fonction), sinon, l'IA pose une question à l'utilisateur afin d'ajouter un attribut à la base de fait et de ne pas rester bloqué dans une boucle infinie. Dans l'autre cas, la fonction *chainage-avant* est systématiquement appelée puisque l'on s'arrête seulement lorsque le personnage a été trouvé (ou n'est pas trouvable).

Pour poser une question à l'utilisateur, on utilise la fonction *deviner-personnage-ia* dont voici une partie du code Lisp :

```
(defun deviner-personnage-ia (bdf attributes)
  (setq attributes (remove-useless-attributes attributes bdf))
  (let ((attribute (choose-random-element attributes)) val)
    (progn
      ;(print attributes)
      (cond
       ((equal attribute 'espece)
        (format t "~%De quelle espèce est le personnage ?")
        (format t "~%~t1. Humain")
        (format t "~%~t2. Singe")
        (format t "~%~t3. Reptile")
        (format t "~%~t4. Autre")
        (format t "~%Choisissez une option : ")
        (let ((choix2 (get-value 1 4)))
          (cond
           ((= choix2 1) (setq val 'humain))
           ((= choix2 2) (setq val 'singe))
           ((= choix2 3) (setq val 'reptile))
           ((= choix2 4) (setq val 'autre)))))
       ((equal attribute 'peau)
        (format t "~%Quelle est la couleur de peau du personnage ?")
        (format t "~%~t1. Blanche")
        (format t "~%~t2. Grise")
        (format t "~%~t3. Jaune")
        (format t "~%~t4. Verte")
        (format t "~%~t5. Rouge")
        (format t "~%~t6. Marron")
        (format t "~%Choisissez une option : ")
        (let ((choix2 (get-value 1 6)))
          (cond
           ((= choix2 1) (setq val 'blanche))
           ((= choix2 2) (setq val 'grise))
           ((= choix2 3) (setq val 'jaune))
           ((= choix2 4) (setq val 'verte))
           ((= choix2 5) (setq val 'rouge))
           ((= choix2 6) (setq val 'marron)))))
```

Figure 5 : Fonction permettant de poser des questions à l'utilisateur

Cette fonction commence par retirer tous les attributs de la liste *attributes* dont on à déjà connaissance dans la base de fait afin d'éviter de poser des questions redondantes à l'utilisateur. On choisit ensuite aléatoirement un attribut dans la liste restante, puis on pose alors la question correspondante à cet attribut à l'utilisateur. Un menu est alors affiché et l'utilisateur peut faire son choix. A la fin de la fonction, on ajoute le couple *(attribut valeur)* à la base de fait avant de retourner cette dernière.

2) Le « chaînage arrière »

Après avoir fini la fonction de chaînage avant, nous avons essayé de réfléchir à une manière d'implanter une fonction de chaînage arrière afin de pouvoir également deviner un personnage choisi aléatoirement par l'IA. Nous n'avons malheureusement pas trouvé de solution viable pour implémenter cette fonction correctement. Néanmoins, nous avons décidé de récupérer tous les attributs du personnage choisi en une seule fois (sans inférence). Pour cela, nous avons créé une seconde base de

règles comportant tous les personnages ainsi que tous les attributs qui leurs sont reliés. Ainsi, voici le fonctionnement de la fonction de « chaînage arrière » :

```
(defun chainage-arriere (bdf bdr characters)
  (let ((perso (choose-random-element characters)))
    (setq bdf (car (get-attributes bdr perso)))
    (if (null bdf)
        (format t "Impossible de déterminer le personnage.")
      (loop
        (let ((result (deviner-personnage-aleatoire bdf perso)))
          (when result  ; quand result n'est pas nil, on sort de la boucle
            (return result))))
      )))
```

Figure 6 : Fonction du chaînage arrière

Ici, l'IA choisi un personnage aléatoirement et récupère tous ses attributs. Ensuite, la fonction boucle en attendant un résultat de la fonction *deviner-personnage-aléatoire*. Cette dernière permet à l'utilisateur de poser des questions à l'IA à travers un menu et de sélectionner un personnage lorsqu'il pense avoir trouvé la réponse finale. Cette fonction renvoie «*T*» lorsque le joueur à correctement trouver le personnage.

3) La combinaison des deux types de chaînage

Afin de rendre l'application plus vivante, nous avons décidé de combiner les deux fonctions présentées précédemment afin de pouvoir faire une vraie partie tour par tour contre l'IA. Ainsi, voici la fonction qui combine les deux précédentes :

```
(defun multiplayer (bdr1 bdr2 attributes characters)
  (let ((perso (choose-random-element characters)) bdf_ia bdf_joueur result_ia)
    (setq bdf_ia (car (get-attributes bdr2 perso)))
    (loop while (not (or result_ia (assoc 'nom bdf_joueur)))
        do
          (progn
            ;Joueur:
            (format t "~%C'est à vous de jouer :")
            (setq result_ia (deviner-personnage-aleatoire bdf_ia perso))
            (when result_ia
              (return result_ia))
          ;IA:
            (format t "~%C'est au tour de l'IA :")
            (setq attributes (remove-useless-attributes attributes bdf_joueur))
            (setq bdf_joueur (chainage-avant bdf_joueur bdr1 attributes T))
          ))
    (if result_ia
        (format t "~%Vous avez gagné !")
      (format t "~%L'IA a gagné ! Votre personnage était ~a" (cadr (assoc 'nom bdf_joueur))))))
```

Figure 7 : Fonction combinant les deux types de chaînage

Dans cette fonction, on reprend le même principe de la fonction de « chaînage arrière » pour la sélection d'un personnage aléatoirement et de ses attributs. L'IA et le joueur jouent alors chacun leur tour tant que la fonction *deviner-personnage-*

aléatoire ne renvoie pas « *T* » (le joueur a trouvé le personnage) ou tant que la base de fait utilisée dans la fonction de *chainage-avant* ne contient aucun nom de personnage.

4) L'interface

Afin de rendre le SE opérationnel et agréable à utiliser, nous avons implémenter une interface textuelle complète afin de répondre et poser des questions facilement au système. Cette interface se charge aussi d'éviter des erreurs lors de la formulation des requêtes d'utilisateur.

```
(defun get-value (min max)
  (let ((choix (read)))
    (loop while (or (not (numberp choix)) (< choix min) (> choix max))
        do
          (format t "~%Choix incorrect, veuillez réessayer : ")
          (setq choix (read)))
    choix))
```

Figure 8 : Fonction get-value évitant les erreurs utilisateur

```
(defun menu_jeu ()
  (let (choix (all_attributes '(espece peau cheveux couvre_chef yeux sexe dents camp moustache mince)) (all_characters '(Mari
    (loop while (or (not (numberp choix)) (not (= choix 4)))
          do
          (format t "~%Menu :~%")
          (format t "1. Faire deviner un personnage à l'IA (chaînage avant)~%")
          (format t "2. Deviner un personnage choisi aléatoirement par l'IA (chaînage arrière)~%")
          (format t "3. Jouer contre l'IA~%")
          (format t "4. Quitter~%")
          (format t "Choisissez une option : ")
          (setq choix (get-value 1 4))
            (cond
             ((= choix 1) (chainage-avant '() *BR* all_attributes))
             ((= choix 2) (chainage-arriere '() *BR2* all_characters))
             ((= choix 3) (chainage_mixte *BR* *BR2* all_attributes all_characters))
             ((= choix 4) (format t "Au revoir !")))
            )))
```

Figure 9 : Exemple d'interface : le menu principal

V. FONCTIONNEMENT

Le fonctionnement de l'application est assez intuitif, il suffit de se laisser guider par la console. Trois modes de jeu sont disponibles :

1) Faire deviner un personnage

Ce mode va poser des questions dont l'ordre est aléatoire à l'utilisateur afin de deviner le personnage, une fois qu'il a assez d'éléments, il renvoie le nom du personnage avant de revenir au menu.

```
 2. Blonds clairs
 3. Bruns
 4. Châtains
 5. Rouges
 6. Oranges
 7. Aucuns (chauve)
Choisissez une option : 3

Les dents du personnage sont-elles visibles ?
 1. Oui
 2. Non
Choisissez une option : 1

Dans quel 'camp' se trouve le personnage ?
 1. Gentil
 2. Méchant
Choisissez une option : 1

Le personnage est : MARIO
Menu :
1. Faire deviner un personnage à l'IA (chaînage avant)
2. Deviner un personnage choisi aléatoirement par l'IA (chaînage arrière)
3. Jouer contre l'IA
4. Quitter
Choisissez une option :
```

Figure 10 : Exemple de partie dans ce mode

2) Deviner le personnage de l'IA

Cette fois-ci, la console va proposer à l'utilisateur de poser la question qu'il veut sur l'attribut de son choix. Une fois que l'utilisateur à assez d'éléments, il peut tenter de trouver le personnage afin de finir la partie.

```
La réponse à la question 'Est-ce que la peau du personnage est VERTE ?' est : non
Quelle question voulez-vous poser ?
 1. Espece
 2. Peau
 3. Cheveux
 4. Couvre-chef
 5. Yeux
 6. Sexe
 7. Dents
 8. Camp
 9. Moustache
 10. Corpulence
 11. Take a guess !
Choisissez une option : 2

Votre question : Est-ce que la peau du personnage est :
 1. Blanche
 2. Grise
 3. Jaune
 4. Verte
 5. Rouge
 6. Marron
Choisissez une option : 2

La réponse à la question 'Est-ce que la peau du personnage est GRISE ?' est : non
```

Figure 11 : Une partie en cours

```
Quel est votre guess ? :
 1. Mario
 2. Luigi
 3. Peach
 4. Daisy
 5. Wario
 6. Waluigi
 7. Yoshi
 8. Harmonie
 9. Donkey Kong
 10. Diddy Kong
 11. Bowser
 12. Goomba
 13. Maskass
 14. Koopa
 15. Topi Taupe
 16. Bowser Jr
 17. Boo
 18. Frere Marto
 19. Skelerex
 20. Pom Pom
Choisissez une option : 20

C'est correct! Bravo!
```

Figure 12 : Fin de partie

3) Jouer contre L'IA

Ce mode combine simplement les deux modes de jeu. Ainsi, l'utilisateur et l'IA posent des questions chacun leurs tours afin de trouver le plus vite le personnage de l'autre. La partie s'arrête lorsque l'un des deux a trouvé.

VI. COMPLEMENTS : PISTES D'AMELIORATION

Après plusieurs ajustements au niveau de la base de règle, le programme réussit à deviner tous les personnages. Néanmoins, il est facile de voir après quelques parties que le système expert manque d'optimisation pour deviner le personnage de l'utilisateur. Ici le problème est facilement repérable et il tient en deux point.

Tout d'abord nous pourrions développer et optimiser la base de règle. En effet celle-ci est constitué de beaucoup de règles et il est difficile à première vue de savoir lesquelles sont pertinentes. Malgré une phase d'optimisation assez longue pour cette conception, nous pourrions utiliser des outils algorithmiques plus poussés à l'instar de la mise en forme de la grille de personnage en table SQL afin d'en percer tous les secrets. Ainsi, nous pourrions trouver éventuellement de nouvelles règles plus discriminante pour notre programme.

Ensuite, un autre point faible de l'IA développé se trouve au niveau de la partie chaînage avant. En effet pour que l'IA arrive à deviner le personnage choisi par l'utilisateur, elle pose des questions aléatoirement afin d'enrichir sa base de fait jusqu'à trouver une réponse valable. Afin de rendre l'IA plus performante nous pourrions utiliser une méthode proche de celle étudiée dans le TP2 afin que le programme apprenne au fur et à mesure quelles sont les questions les plus intéressantes et lesquelles sont les moins utiles. Ainsi, il aurait été possible de pondérer les questions suivant un algorithme simple tel que :

DEBUT :
Si IA a gagné :
    Donner 1 pt à la dernière question posée
Sinon :
    Ne rien faire
FIN

Ensuite lors de prendre la question au hasard, l'algorithme donnera plus de poids aux questions ayant le plus de points. Nous pouvons par exemple imaginer qu'une question ayant 2 points aurait deux fois plus de chance d'être prise qu'une fonction avec 1 seul point. Néanmoins, cet algorithme n'améliorera le programme qu'après un grand nombre de parties jouées.

Avec un tel système, il serait aussi possible de pondérer les réponses en fonction du nombre de personnages éliminés. Cela permettrait d'optimiser visiblement le programme dès les premières parties. Un tel algorithme pourrait fonctionner ainsi :

DEBUT : (après chaque réponse de l'utilisateur)
Nb_perso_elimines = calcul_perso_elimines()
Donner [Nb_perso_elimines] points à la dernière question posée
FIN

Dès lors, l'IA aura tendance rapidement à poser des questions qui vont éliminer le plus de personnage.

Ces optimisations permettraient de calquer automatiquement le comportement de l'IA sur celui d'un humain qui au bout de quelques parties voudra poser les questions les plus efficaces possibles.