# Rapport du projet d’IFB : Jeu de Belote Coinchée en C dans la console

## Introduction

Dans le cadre de l’UV IFB nous avons dû réaliser un jeu de Belote Coinchée en C. Le cahier des charges nous impose de programmer uniquement en C et de faire un programme qui s’exécute dans la console.

Table des matières

[Rapport du projet d’IFB : Jeu de Belote Coinchée en C dans la console 1](#_Toc42961011)

[Introduction 1](#_Toc42961012)

[Sources utilisées 1](#_Toc42961013)

[Organisation générale du projet 1](#_Toc42961014)

[Solutions techniques et fonctionnement du programme 2](#_Toc42961015)

[Modularité 2](#_Toc42961016)

[Les types de variable personnalisée 3](#_Toc42961017)

[Les Menus 3](#_Toc42961018)

[L’affichage 4](#_Toc42961019)

[Gestion des pseudos 5](#_Toc42961020)

[Distribution des cartes et phase d’annonces des contrats 5](#_Toc42961021)

[Calcul de la force d’une carte 6](#_Toc42961022)

[Intelligence Artificielle (ia) 6](#_Toc42961023)

[Déterminer si on a le droit de jouer une carte 7](#_Toc42961024)

[Gestion des fichiers 9](#_Toc42961026)

## Sources utilisées

<https://www.belote.com/regles-et-variantes/regle-belote-coinche/>

<http://www.ffbelote.org/regles-coinche/#7>

<https://mon.gameduell.fr/gd/s03.do?gametype=bel&top>

## Organisation générale du projet

Afin de réaliser ce projet et sachant qu’à cause du confinement il serait très difficile pour les membres du projet de se retrouver en présentiel pour se coordonner sur le projet, nous avons utilisé la plateforme discord afin de communiquer sur les problèmes que nous avons rencontrés et sur les fonctions à réaliser. Pour stoker et versionner notre code, nous avons utilisé la plateforme GitHub qui nous permet de travailler simultanément sur le projet et de toujours avoir la version du code la plus à jour. Voici le lien de notre projet public sur GitHub :

<https://github.com/Flo3171/IFB_projet_Belote>

(Flo3171 est le pseudo de Florian CLOAREC et Fituning35 celui de Carlo AZANCOTH)

## Solutions techniques et fonctionnement du programme

Nous allons maintenant vous détailler les solutions techniques que nous avons mises en place et la façon dont nous avons réalisé les différents points du cahier des charges dans l’ordre chronologique ou cela a été implémenter dans le projet.

### Modularité

Dès le début du projet nous avions conscience que le projet allait être composé d’un grand nombre de fonctions et de lignes de code, c’est pourquoi afin d’avoir un projet clair et ordonné, nous avons décidé de séparer les différentes fonctions dans des fichiers séparés ; à l’intérieur de ces fichiers les fonctions sont regroupées selon un thème commun (affichage, formatage, gestion des cartes, gestion des fichier…). Afin d’inclure les prototypes des fonctions à tous les endroits où cela est nécessaire, nous avons créé un fichier main.h ; ce ficher regroupe toute les constantes, les énumérations est les structure utilisées dans ce projet, mais c’est surtout dans ce fichier que sont inclus tous les fichier .h associés à chaque fichier .c contenant nos fonctions. Ce fichier main.h est alors inclus au début de chaque fichier contenant les fonctions du projet. Ainsi cela permet de s’assurer que toutes les fonctions créées spécifiquement pour ce projet ainsi que tous les autres objets susceptibles d’être manipulés par les fonctions soient utilisables, qu’importe le fichier dans lequel elles se trouvent. C’est aussi dans ce fichier main.h que nous avons inclus les bibliothèques standards que nous utilisons dans le programme. Nous avons bien conscience que cette solution n’est pas la plus optimale. En effet dans certains fichiers, des portions de code sont incuses alors qu’elles ne seront jamais utilisées, ce qui augmente de façon non négligeable la taille de l’exécutable du programme. Mais nous avons choisi cette solution car elle permet une grande liberté lors du développement du projet. En effet avec autant de fichiers, on est amené à passer très souvent d’un fichier à l’autre. Grâce à cette solution, lorsque l’on veut rajouter une fonction ou simplement appeler une autre fonction dans une fonction, il n’est pas nécessaire de s’assurer que le prototype de la fonction est bien inclus dans le fichier dans lequel on travaille. Cela permet de se concentrer sur ce que l’on fait, de gagner du temps et surtout d’éviter de nombreuses erreurs de compilations. Néanmoins cette manière de faire entraîne un autre problème : avec autant de fichiers inclus les uns dans les autres, il est alors possible que par inadvertance on crée une boucle d’inclusion infinie, ce qui empêcherait la compilation de se dérouler normalement. C’est pourquoi chacun des fichiers .h est entouré d’un code préprocesseur qui permet de ne compiler le contenu du fichier seulement si c’est la première fois qu’il est lu par le préprocesseur.

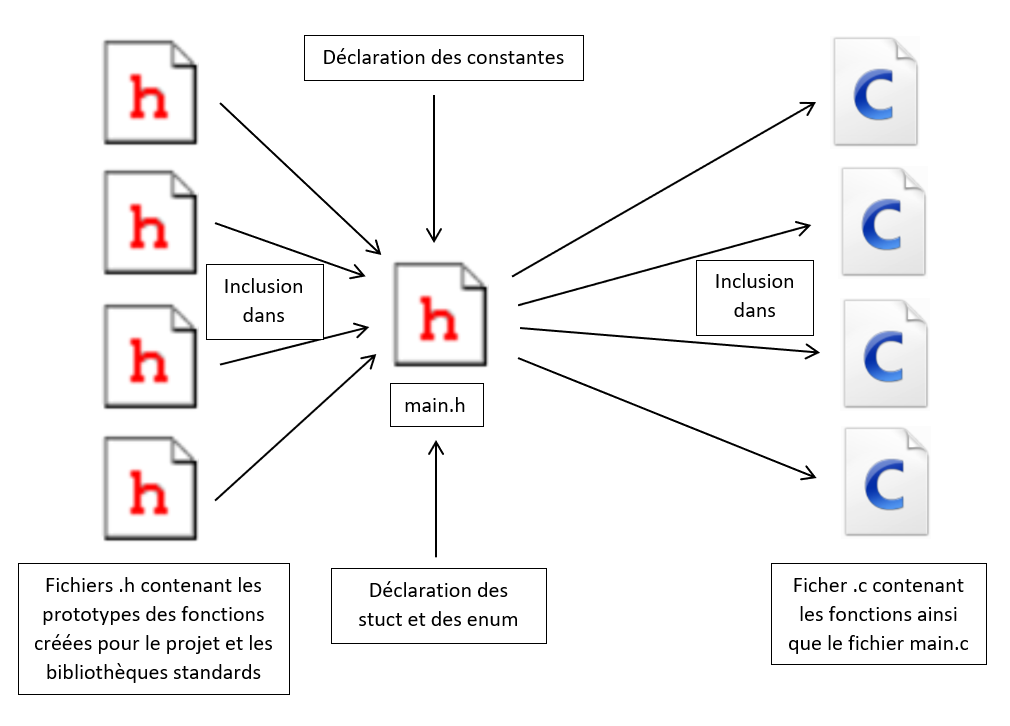
/\*Code utilisé pour protéger le programme des boucles d'inclusions infinie\*/

#ifndef \_NOM\_FICHIER\_H\_

#define \_NOM\_FICHIER\_H\_

/\*contenu du fichier\*/

#endif /\* \_NOM\_FICHIER\_H\_ \*/

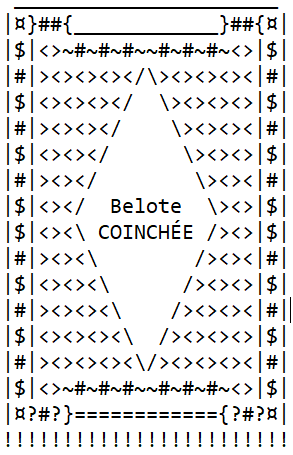
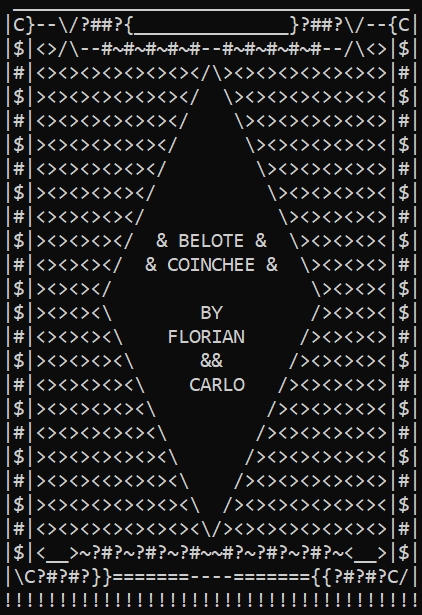


### Les types de variable personnalisée

Dans ce projet un grand nombre de données à stoker en mémoire n’était pas sous la forme d’un nombre afin d’éviter de devoir stocker ces informations sous forme de chaîne de caractères, ce qui aurait été très lourd à manipuler et aurait pris beaucoup de place en mémoire. Nous avons décidé pour éviter ce problème de créer des types de variable. Nous avons donc créé une enum nommée Couleur, une autre nommée Valeur puis une stcuct nommée Carte qui est composée d’une sous-variable de type Couleur et une sous-variable de type Valeur. Ainsi nous pouvions manipuler les variables de type Carte comme une unique variable et il devient alors très facile de manipuler des tableaux de variable de type Carte. Nous avons aussi créé des types de variable personnalisée pour les Contrat

### Les Menus

Afin que l’utilisateur puisse utiliser toutes les fonctionnalités que nous avons développées, nous avons ajouté la fonction menuPrincipal qui est l’unique fonction appelée dans le main.c. Premièrement la fonction affiche le logo (qui représente une carte), que l’on a conçu avec les 256 caractère du code ASCII étendu.



Nous avons donc réalisé 2 versions du logo, car nous nous sommes rendu compte que le premier logo était trop petit pour afficher à l’intérieur de la carte les différents sous-menus (comme sur l’image ci-dessus). Ensuite la fonction se charge d’acquérir le pseudo du joueur qui est directement enregistrée dans les fichiers de sauvegarde gestion\_scores\_joueurs.csv. Nous arrivons ensuite au menu principal qui est composé de 7 sous-menus. Tout le code est placé dans une boucle while pour répéter indéfiniment cette partie du code qui appelle le cœur du projet, le seul moyen de sortir de la boucle est le sous-menu 7 qui ferme les fichiers précédemment ouverts puis met fin à la fonction. Nous avons ensuite les sous-menus qui lance une nouvelle partie qui sera jouée par l’utilisateur ou le sous-menu pour jouer 1000 parties entre les IA, cela nous a permis de récupérer des statistiques afin de pouvoir tester la capacité de nos IA à remplir des contrats et à gagner des parties. Ensuite on a les paramètres qui nous permettent de modifier les pseudos des joueurs comme on le souhaite dans la limite des caractères autorisées et le changement d’utilisateur qui permet l’acquisition d’un nouveau pseudo et d’accéder à une autre sauvegarde. Nous avons également les statistiques, qui permettent de voir les statistiques de l’utilisateur sélectionné, comme le nombre de victoire, le score maximal et le nombre de manches minimum pour gagner une partie. Pour finir le leaderboard qui nous permet de voir les meilleurs scores enregistrés.

## Les pseudos

Pour les pseudos, nous avons décidé de crée un tableau de char à 2 dimension, il est de dimension 4x21, 4 pour les 4 joueurs et 21 qui correspond à la taille maximum des pseudos, plus ‘’\0’’. En effet en utilisant cette méthode il est plus facile d’enregistrer les noms de joueur est de les envoyer dans les fonctions. On peut donc utiliser une variable personnalisée de type Joueur pour récupérer le pointeur vers le pseudo du joueur NORD, SUD, EST, OUEST en indiquant directement l’emplacement dans le tableau, par exemple pseudo[SUD-1] correspond au pseudo du joueur SUD.

/\*\* On crée un tableau de char de dimension 4x21 \*\*/

char pseudo**[**4**][**TAILLE\_MAXI\_PESEUDO**+**1**];**

/\*\* On defini les pseudos d'IA pas défaut \*\*/

strcpy**(**pseudo**[**NORD**-**1**],**"A\_Philipe"**);**

strcpy**(**pseudo**[**EST**-**1**],**"Gilou"**);**

strcpy**(**pseudo**[**OUEST**-**1**],**"Tutu"**);**

Utiliser cette méthode à été utile pour l’acquisition des pseudos avec la fonction acquisitionPsseudoAvecMessage et surtout la fonction parametre qui l’appelle car on peut lui envoyer seulement un pointeur pour tous les pseudos. La fonction d’acquisition du pseudo elle demande à l’utilisateur de saisir un nouveau pseudo pour le joueur sélectionné précédemment, ensuite elle verifie la validité de du pseudo (longueur maximale, caractères interdits, pseudo vide) et redemande le pseudo tant que la saisie est incorrecte.

**switch** **(**controle**)**

**{**

**case** 1 **:** afficheSousMenus**(**"Votre pseudo possede des carracteres interdis. Choisisez votre pseudo:"**,**"pseudo"**);**

**break;**

**case** 2 **:** afficheSousMenus**(**"Votre pseudo n'as pas le nombre de carracteres recquis. Choisisez votre pseudo:"**,**"pseudo"**);**

**break;**

**case** 3 **:** afficheSousMenus**(**"Votre pseudo ne respecte pas les criteres. Choisisez votre pseudo:"**,**"pseudo"**);**

**}**

### L’affichage

Nous avons décidé de développer les fonctions d’affichage dès le début du projet afin de pouvoir voir ce qui se passe lors des tests et du debug des autres fonctions. Etant donné que nous utilisons des types de variable personnalisées pour les cartes et non des chaînes de caractères, nous ne pouvions pas afficher directement le contenu de la variable tel quel. C’est pourquoi nous avons mis en place les fonctions de formatages, qui permettent de faire la convention entre une variable de type Carte (ou Contrat) et une ou plusieurs chaînes de caractère. Puis ces chaînes de caractère une fois formatées sont affichées dans la console à l’aide de la fonction printf par fonctions chargées de l’affichage.

Sachant que nous devions réaliser ce projet et qu’il n’était pas envisageable de mettre en place une interface graphique, nous avons tenté de rendre l’interface avec l’utilisateur la plus belle et la plus ergonomique. Pour ce faire nous avons créé des fonctions de formatage qui permettent de centrer les chaînes de caractères pour que le pseudo des joueurs quel qu’il soit s’affiche toujours au centre de l’espace où il est sensé s’afficher. Nous avons de plus prévu le cas où la chaîne de caractères à afficher dépasserait la taille disponible dans le cadre où elle est affichée ; dans ce cas, la chaîne est coupée afin de ne pas décaler les autres affichages qui sont parfois sur la même ligne. Afin de formater les messages de tailles plus longues, une fonction (decoupeChaine) se charge de couper la chaîne au niveau d’un espace et d’afficher le reste de la chaîne sur la ligne suivante.

Pour afficher les Menus nous avions premièrement fait un affichage uniquement avec un printf pour le menu principal ||||

printf**(**"\t \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \n"**);**

printf**(**"\t|¤}##{\_\_\_\_\_\_\_\_\_\_\_\_}##{¤|\n"**);**

printf**(**"\t|$|<>Menu Principal<>|$|\n"**);**

printf**(**"\t|#| |#|\n"**);**

printf**(**"\t|$| 1-nouvelle partie|$|\n"**);**

printf**(**"\t|#| |#|\n"**);**

printf**(**"\t|$| 2-leaderboard |$|\n"**);**

printf**(**"\t|#| |#|\n"**);**

printf**(**"\t|$| 3-statistiques |$|\n"**);**

printf**(**"\t|$| |$|\n"**);**

printf**(**"\t|#| 4-changement |#|\n"**);**

printf**(**"\t|$| d'utilisateur |$|\n"**);**

printf**(**"\t|#| 5-paramètres |#|\n"**);**

printf**(**"\t|$| |$|\n"**);**

printf**(**"\t|#| 6-quitter |#|\n"**);**

printf**(**"\t|$|<>~#~#~#~~#~#~#~<>|$|\n"**);**

printf**(**"\t|¤?#?}============{?#?¤|\n"**);**

printf**(**"\t!!!!!!!!!!!!!!!!!!!!!!!!\n"**);**

La fonction afficheInterfacePli est la fonction qui ce charge d’afficher le plateau de jeu durant un pli. Elle est appelée un grand nombre de fois et c’est le résultat de cette fonction que l’utilisateur voit durant la majorité du temps d’exécution du programme. Afin de déterminer la forme et l’emplacement des différente information qui sont affichée à l’écran, nous avons commencé à travailler sur un fichier texte ouvert dans le bloc note ceci nous a permis de voir directement le résultat. La fonction prend comme paramètre toure les information qui vont devoir être affichée comme la main du joueur, les scores ou bien les carte posée sur la tables… La fonction envoie toute ces informations au fonction de formatage associée puis les chaine de caractères ainsi formatée sont affichée a l’endroit voulu sur l’écran grâce à plusieurs printf(). Nous avons choisi d’utiliser un printf par ligne à afficher pour des raison de lisibilité du code, ainsi en lisant le code il est possible de distinguer la forme de l’interface ce qui permet de faire des modification beaucoup plus vite.

### Distribution des cartes et phase d’annonces des contrats

Afin de distribuer les cartes au différent joueur de manière rapide et simple nous avons décidé de ne pas utiliser un méthode proche de ce qui ce fait lors d’une partie de carte réel ou on distribue une carte par joueur en faisant les tour. Nous avons préféré le faire de manière aléatoire. Pour ce faire on déclare un tableau de 32 variables de typer Carte et on le remplit avec toutes les carte d’un jeu de 32 cartes. Puis on déclare un second tableau de Carte à deux dimensions (Carte tableauCarte**[**4**][**8**];**), la première dimension vaut 4 et correspond au nombre de joueur et la seconde vaut 8 et correspond au nombre de carte que chaque joueur à en main. C’est ce tableau à deux dimension qui va contenir toutes les carte que tous les joueur on en main durant la partie. Afin de s’assurer que le mélange des carte sera bien aléatoire on se sert de le fonction rand() on obtient alors un nombre aléatoire qui sera l’indice de la carte que l’on va distribuer en premier, cette carte est alors ajoutée au premier emplacement du tableau contenant les cartes de tous les joueur et elle est supprimé du tableau qui contient les cartes à distribuer afin qu’aucune carte ne puisse être distribuée 2 fois. On continue cette opération en incrémentant l’indice ou la carte va être distribuée jusqu’à ce que le tableau contenant les cartes des joueur soit remplie de toutes les cartes du jeu de manière unique. Ainsi pour accéder à la carte n du joueur SUD il faudra entrer : tableauCarte**[**SUD**-**1**][**n**]** ce qui en therme de pointeur est équivalent à **\*(**pTableauCarte **+** **(**SUD**-**1**)\***8 **+** n**).**

Une fois les cartes distribuées, on entre la phase d’annonce des contrats. La fonction qui ce charge de cela appelle selon si le joueur qui doit proposer un contrat est un utilisateur ou un ordinateur la fonction associé respectivement proposeContratUtilisateur et choixCarteIA. Ces fonctions renvoient chacune une variable de type Contrat qui représente le contrat que le joueur a choisi de prendre. Si ce contrat à un nombre de point supérieur a 0, ce qui signifie que le joueur n’a pas passé alors ce nouveau contrat vient remplacer celui proposer par le dernier joueur à avoir proposé un contrat. C’est lors de l’acquisition que l’on ce charge de vérifier que le nouveau contrat proposé à bien un nombre de point supérieur au dernier contrat. Les joueur continue de proposer des contrat chacun lors tour jusqu’à qu’après le premier tour, 3 joueur d’affilé passent. Une fois que plus personne ne souhaite surenchérir, la fonction qui gère les contrat de termine et revoit le contrat définitif qui sera appliqué pour le reste de la manche.

La possibilité de coincher et de sur coincher à aussi été mise en place, ainsi un joueur à la possibilité de coincher si un joueur de l’équipe adverse à proposé un contrat. En coinchant, le joueur propose un nouveau contrat identique au précèdent à la différence que la sous variable .coinche ne vaut plus NORMAL, mais vaut alors COINCHE. Un joueur à la possibilité de sur coincher si un joueur de l’équiper adverse à coinché un contrat proposé par un membre de sont équipe. La sur coinche se passe comme pour la coinche à la différence que la sous variable .coinche vaut alors SURCOICHE.

### Calcul de la force d’une carte

Afin de déterminer quelle carte est la plus forte et va remporter le plis, la fonction forceCarte permet d’associer à chaque carte une valeur numérique comprise entre 0 et 1 qui reflète ca force, plus ce nombre sera grand plus la carte sera forte. Après avoir cherché qu’elle serait la méthode la plus efficace et la plus pratique afin de donner une force à chaque carte, nous avons décidé de passer par une approche probabiliste. Ainsi la couleur de l’atout (qui peut être sans-atout et tout-atout) est et la couleur de l’entame (la première carte jouée dans le plis) sont des paramètres de la fonction forceCarte. La force d’une carte se calcul avec la formule suivante : . Ce qui signifie par exemple que le valet d’atout va battre les 31 autres cartes du paquet et ainsi il aura une force de 1, à l’inverse le 7 de non-atout s’il n’est pas dans la couleur de l’entame ne bâtera aucune carte et aura donc une force de 0. Avoir fait des enum et des struct pour définir la variable personnalisée Carte nous permet de manipuler les sous variable .couleur et .valeur comme des entier et ainsi nous pouvons utiliser des switch qui dans le cas de cette fonction améliorent grandement la lisibilité du code. Grâce à cette fonction, il devient très facile de trouver le vainqueur d’un pli, il faut alors chercher quelle carte à la force maximum parmi les cartes du plis. Mais cette fonction à aussi d’autres usages, elle est utilisée dans les fonctions ia, afin d’évaluer et de pouvoir faire des calcul dessus afin de prendre une décision.

### Intelligence Artificielle (ia)

Afin de réaliser ce programme, nous avons dû créer deux ia. La première permet de déterminer quelle sera le contrat que l’ordinateur va prendre ou s’il va passer et la seconde permet de choisir quelle carte l’ordinateur va jouer lors de chaque pli. Ces deux ia prennent la forme de deux fonctions (proposeContratIA et choixCarteIA) qui prennent comme paramètre de nombreuses informations sur la partie comme les cartes que le joueur à en main et retournent respectivement une variable de type contrat et un entier qui correspond à la position de la carte à jouer dans la main du joueur.

Afin de déterminer quel contrat l’ordinateur va prendre, le programme fait la somme de la force de chacune des cartes dans toutes les couleur possible (y compris tout-atout et sans-atout) et regarde dans quelle couleur la valeur est la plus grande, une grande valeur correspond à un main forte dans la couleur. En faisant des test sur cette méthode nous avons découvert que la somme des forces des carte de la main était plus grande en tout atout que dans les autres couleur, afin de régler ce problème, nous avons remis cette valeur à l’échelle, ainsi la plus forte main en tout-atout vaut la même valeur que la plus forte main dans une autre couleur et le nombre que l’on obtient est compris entre 0 et 1. Puis la fonction regarde si ce nombre dans la meilleur couleur est supérieur au seuil minimum de prise de contrat. Si tel est le cas, l’ordinateur va chercher à prendre un contrat dans cette couleur. La valeur du contrat que l’ordinateur va prendre est proportionnelle au nombre trouver précédemment plus ce nombre sera supérieur à la valeur du seuil minimum de prise de contrat, plus l’ordinateur prendra un contrat avec un nombre important de points. Afin de respecter les règle du jeux, la fonction va alors tester si le contrat quelle veut proposer est valide et pour se faire on teste, si le nombre de point proposé par ce nouveau contrat est supérieur au dernier contrat proposé (qui est celui avec le plus grand nombre de point sachant que les autres joueur doivent aussi surenchérir), alors la fonction renvoie une variable de type contrat avec le nombre de points et la couleur qu’elle à déterminer. Sinon ou si sa main ne lui permet pas de dépasser le seuil minimal de prise alors elle passe, ce qui ce matérialise par le renvoi pas la fonction d’une variable de type Contrat dont le nombre de point vaut 0.

-->fonction choixCartIa

### Déterminer si on a le droit de jouer une carte

A de nombreuse reprise lors de l’exécution du programme il est nécessaire de déterminer si la carte qu’un joueur ou un ordinateur est valide ou non. Pour ce faire nous avons créé la fonction carteValide qui renvoie 1 si la carte que l’on veut jouer et valide en fonction des cartes déjà présente sur la table, des carte dans la main du joueur et de la couleur de l’atout. Afin de transposer les règle du jeux qui ne sont pas évidente à traduire en langage algorithmique, nous avons créé l’organigramme si dessous afin de déterminer si une carte peut être jouer, ainsi une fois se travaille préliminaire effectué, il a été beaucoup plus facile de rédiger la fonction carteValide en faisant une succession de if et de else if imbriqué les un dans les autres.

Organigramme décisionnel de la fonction qui détermine si une carte peut être posée dans un pli

Oui

Non

1er carte du pli ?

Non

Oui

Le joueur a-t-il un atout de valeur supérieure au meilleur atout posé ?

Oui

Non

Y a-t-il déjà un atout de posé ?

Oui

Non

Oui

L’entame est-elle en atout ?

Non

Le joueur a-t-il un atout ?

Non ou il n’a pas encore joué

Oui

Le partenaire est-il maitre ?

Non

Le joueur possède il la couleur demandée ?

Oui

### Gestion des fichiers

Si le joueur n’existe pas déjà et enregistre la ligne à laquelle est stockée sa sauvegarde.

Nous avons utilisé des fichier .csv car ils nous permettent d’enregistrer corectement les informations à stocker et de pouvoir lire ces information avec un tableur (excel).

### Gestion des score

### Jouer des partie ente 4 ordinateurs