Rapport final

Cyril BUI, Aymeric DUHAMEL

3 Juin 2022

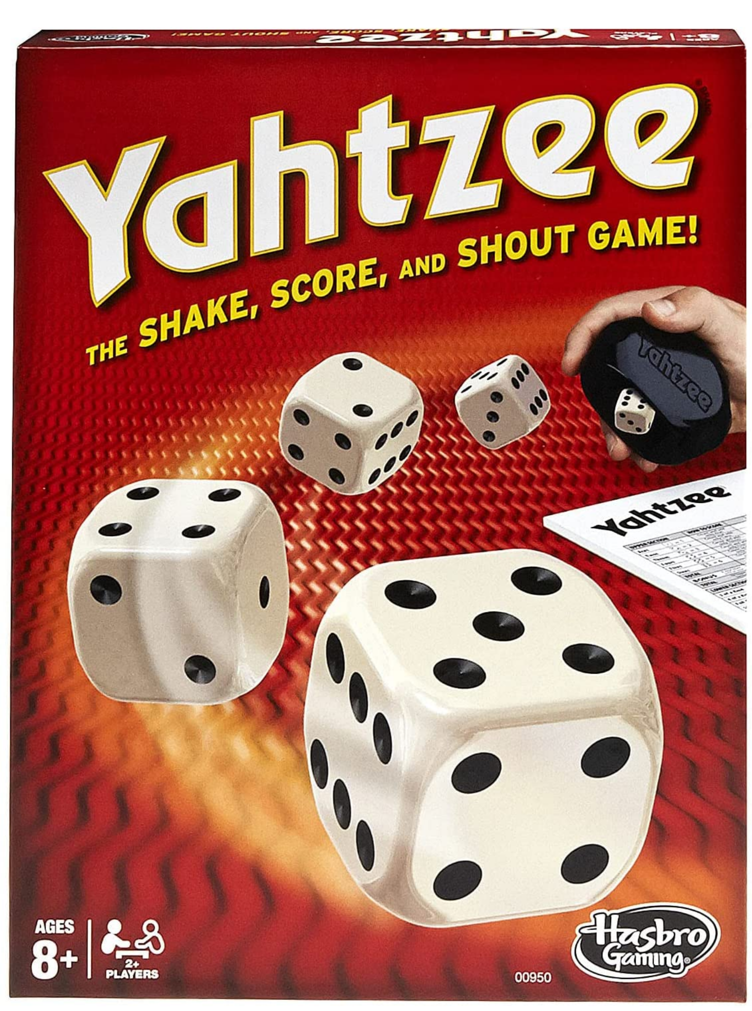

Table des matières

1 Introduction **3**

2 Description du projet **3**
2.1 Règles du Yahtzee 3
2.2 Objectif du projet 5

3 Première approche : Decision_Une **6**
3.1 Idée générale 6
3.2 La fonction de reconnaissance des combinaisons 6
3.3 Seconde partie du programme. 7
3.3.1 Poids1 8
3.3.2 Poids2 8
3.3.3 Défauts de la stratégie 9

4 Deuxième approche : Toutes_les_esperances **10**
4.1 Première étape : récupérer l'ensemble des combinaisons 10
4.2 Représentation des groupes de cases à jouer 11
4.3 Les structure de données 11
4.4 Calcul des espérances lorsqu'il reste une case à jouer 12
4.5 Calcul des autres espérances 13
4.6 Retour sur les règles annexes, le Joker et le Bonus 14
4.7 Etape finale : jouer 14

5 Efficacité des deux premiers programmes **15**
5.1 Premiers tests 15
5.2 Méthode de couplage 15
5.2.1 Première méthode de la réduction de la variance : couplage 15
5.2.2 Résultats et statistiques 17
5.3 Variable antithétique 17
5.3.1 Principe 17
5.3.2 Résultats et statistiques 19

6 Troisième approche : `Perturbation` **19**
6.1 Nouvelles évaluations des décisions 20
6.2 Optimisation des α_i 21
6.3 Résultats et analyse 22
6.3.1 Contre `Decision_Une` 22
6.3.2 Contre `Toutes_les_esperances` 22

7 Gestion du projet et conclusion **23**

1 Introduction

Ce document constitue le rapport du projet 2A IM portant sur la fabrication d'un algorithme jouant au Yahtzee. L'objectif étant de définir un algorithme qui puisse battre le plus de stratégie adverse possible. Nous pourrons confronter nos stratégies entre elles, ainsi qu'avec certaines proposés par M. Peyre. Dans ce document seront détaillées plusieurs approches afin de remplir cette objectif. Ces approches seront de plus en plus élaborées.

Dans un premier temps nous détaillerons les enjeux du projet en détaillant le jeu de Yahtzee. Nous expliquerons ensuite les 3 stratégies que nous avons conçu au cours de ce projet.

2 Description du projet

2.1 Règles du Yahtzee

Les règles sont disponibles plus en détail à l'adresse : `https://www.hasbro.com/common/instruct/Yahtzee.pdf`

Le Yahtzee, aussi appelé Yam's, est un jeu de dés et dit de hasard raisonné. Cela signifie que le joueur doit tirer partie de résultats lié au hasard afin d'augmenter ses chances de victoires. Ainsi il est possible de battre un adversaire sur le long terme, en réalisant un grand nombre de partie. En revanche le résultat d'une seule partie ne saurait être significatif, chaque joueur devant composer avec ses propres dés, et donc avec sa propre chance.

Le but du jeu est de faire un plus grand score que son adversaire, ou d'atteindre un score maximal si on joue tout seul. Chaque joueur dispose d'un grille de score comme celle ci-après, composée d'une section inférieure et d'une section supérieure.

A tour de rôle, chaque joueur va lancer 5 dés pour tenter d'obtenir une des combinaisons de la feuille et marquer des points (nombre différent selon les combinaisons). Le joueur pourra choisir deux fois de relancer ou non les dés obtenus (qu'ils aient déjà été relancés ou non). On nommera le deuxième et le troisième lancer des relancers. Une fois les relancers effectués, le joueur devra inscrire dans la case correspondant à l'une des combinaisons obtenues un certain nombre de points :

1. SECTION SUPÉRIEURE : Nombre de dés multiplié par la valeur correspondante : Cinq dés qui affichent "4" donnent $5 \times 4 = 20$ points.
2. SECTION INFÉRIEURE
   — `Brelan` et `Carré` : Somme des 5 dés.
   — `Full` : 25 points.
   — `Petite Suite` : 30 points.
   — `Grande Suite` : 40 points.
   — `Chance` : Somme des dés, cette case peut être remplie quelque soit la combinaison.

YAHTZEE

PLAYER'S NAME: ____________________

UPPER SECTION:

MINIMUM REQUIRED FOR BONUS		HOW TO SCORE	GAME #1	GAME #2	GAME #3	GAME #4	GAME #5	GAME#6
Aces	= 3	COUNT AND ADD ONLY ACES						
Twos	= 6	COUNT AND ADD ONLY TWOS						
Threes	= 9	COUNT AND ADD ONLY THREES						
Fours	=12	COUNT AND ADD ONLY FOURS						
Fives	=15	COUNT AND ADD ONLY FIVES						
Sixes	=18	COUNT AND ADD ONLY SIXES						
TOTAL	= 63	➡						
Bonus	IF 63 OR OVER	SCORE 35						
TOTAL	OF UPPER HALF	➡						

LOWER SECTION:

3 of a kind		ADD TOTAL OF ALL DICE						
4 of a kind		ADD TOTAL OF ALL DICE						
Full house		SCORE 25						
Sm. Straight	Sequence of 4	SCORE 30						
Lg. Straight	Sequence of 5	SCORE 40						
YAHTZEE	5 OF A KIND	SCORE 50						
Chance		SCORE TOTAL OF ALL 5 DICE						
TOTAL	OF LOWER HALF	➡						
TOTAL	OF UPPER HALF	➡						
GRAND TOTAL		➡						

FIGURE 1 – Feuille de score du Yahtzee

— `Yahtzee` (5 fois la même valeur) : 50 points

Les cases attribuent plus ou moins de points selon la difficulté de la combinaison réalisé. Si jamais le joueur n'a réalisé aucune combinaison, il peut inscrire 0 points dans une case de son choix. Chaque case doit être remplie une seule fois.

Il y a aussi deux règles supplémentaires qui rapportent des points sous certaines conditions :

1. Règle de la Prime : si le joueur obtient un nombre de points supérieur à 65 sur l'ensemble des cases de la SECTION SUPÉRIEURE (As à Six), le joueur obtient une prime de 35 points à la fin de la partie.
2. Règle du Joker : si le joueur obtient un Yahtzee et que la case contient déjà un ou plusieurs Yahtzee, le joueur gagne un bonus de 100 points. Il peut alors compléter la case correspondante à la valeur des dés dans la SECTION SUPÉRIEURE, et si

cette dernière est pleine, compléter une case de la SECTION INFÉRIEURE de son choix (avec les points détaillés précédemment). Si cela lui est impossible, le joueur doit inscrire un zéro dans une des cases de la SECTION SUPÉRIEURE de son choix.

2.2 Objectif du projet

L'objectif de ce projet est de réaliser des algorithme de résolution du jeu de Yahtzee. Nous nous contenterons des règles citées précédemment lors de la plupart des modélisations, avec un nombre de dés `NDES` $= 5$ et des dés à 6 faces (`NFACES` $= 6$). Nos modèles seront cependant transposables pour des valeurs de `NDES` et `NFACES` différentes, ainsi que pour d'autres manières de compter le score.

Le deuxième objectif est de rendre ces algorithmes le plus performants possibles. On considérera qu'un algorithme est d'autant plus performant qu'il bat de manière consistante d'autres algorithmes. On dira qu'une stratégie bat une autre stratégie si le taux de victoire sur un grand nombre de partie avec cette stratégie adverse est strictement supérieur à 50 %.

Commençons par introduire un peu de vocabulaire et notation :

1. `valeurDes` désigne le résultat d'un lancer de `NDES` dés à n'importe quelle moment de la partie. Ces valeurs résultent uniquement de l'aléatoire. `valeurDes` prends des valeurs parmi les 6^5 combinaisons de $[1, 6]^5$.
2. La feuille de score : elle contient les scores obtenus pour chaque case au cours de la partie. On attribue à chaque joueur une feuille de score.
3. Les cases : Ce sont les différentes combinaisons de dés à réaliser au cours de la partie. Dans les règles traditionnels, on en dénombre 13 : (As, Deux, Trois, Quatre, Cinq, Six, Brelan, Carre, Full, Petite Suite, Grande Suite, Yahtzee)
4. Un coup : On désigne ici l'ensemble des étapes entre le lancer initial de dés et le moment où l'on remplit une nouvelle case sur la feuille de score.
5. Un relancer : il s'agit des choix de dés à garder sur un 5-uplet `valeurDes`.

Nous bénéficions pour ce projet de plusieurs fonctions Python (`defiYahtzee.py`) fournie par M. Peyre permettant de simuler le déroulé d'une partie de Yahtzee. Afin de rendre une décision, nos fonctions de décisions devront donner un 5-uplet $[True, False] \times NDES$ afin de représenter les dés à relancer. S'il ne reste pas de relancer à effectuer, nos fonctions devront renvoyer la case dans laquelle on veut inscrire un score dans la feuille de score. Le reste de la partie est prise en charge dans `defiYahtzee.py`.

Nous allons dans ce projet écrire des fonctions de décision reposant sur des principes algorithmiques et mathématiques différents. Le but est de produire des fonctions de décision qui battent les autres, en faisant jouer les fonctions les unes contre les autres. On pourra ainsi déterminer quelles sont les stratégies les plus performantes.

3 Première approche : Decision_Une

3.1 Idée générale

Cette première stratégie vise à refléter notre façon de jouer et a été construite au fur et à mesure de notre expérience, toujours à partir de la question : “Face à ces cinq dés, que fait-on ?”. Nous avons identifié deux phases algorithmiques, à savoir la reconnaissance et la prise de décision. La reconnaissance, codée à travers la fonction `Possib` recense les combinaisons atteignables à partir d'une combinaison. La prise de décision, codée par la seconde partie du programme, reflète notre façon de penser avec notre modeste expérience en tant que joueur, pour déterminer quelles sont les pistes à poursuivre et quels dés relancer.

3.2 La fonction de reconnaissance des combinaisons

Avant de pouvoir rendre des décisions sur les dès à conserver ou non entre chaque relancer, il était nécessaire de reconnaître les éventuelles combinaisons des 5 valeurs des dés. A travers la fonction `possib`, nous identifions deux types de combinaisons :

1. Les combinaisons dites *atteintes*, qui sont déjà formées par les 5 dés. Ainsi si `valeurDes` $= [2, 2, 3, 3, 3]$, l'ensemble des combinaisons atteintes est (Deux,Trois,Full,Brelan)
2. Les combinaisons dites *atteignables*, qui peuvent être atteintes en relançant certains dés. Pour `valeurDes` $= [2, 2, 3, 3, 3]$, l'ensemble des combinaisons atteignables reconnues par notre algorithme est (Carré, Yahtzee).

Ici il est important de préciser que cette ensemble atteignable ne contient pas réellement toute les combinaisons atteignables depuis `valeurDes`. Cet ensemble est en vérité l'ensemble des cases, puisqu'on peut relancer tous les dés. Nous avons ici choisi arbitrairement quelles sont les combinaisons qui nous semblaient atteignables dans telle ou telle situation. Un Yahtzee ne sera atteignable que dans le cas où `valeurDes` contient au moins 3 dés identiques.

Pour la partie de décision, cette fonction de reconnaissance devait également déterminer quels sont les dès à garder pour chaque combinaison des ensembles *atteint* et *atteignable*. Ces dès gardés sont ceux qui réalise la combinaison pour *atteint*, et ceux qui maximise la probabilité d'atteindre la combinaison visée pour *atteignable*.

Algorithmiquement parlant,la fonction est divisée en plusieurs parties, qui vont chacune déterminer si telle ou telle combinaison est atteinte ou atteignable. Les combinaisons sont donc testées les unes après les autres. Les méthodes de reconnaissance varient selon les combinaisons, mais suivent à peu près le même schéma : reconnaissance de la combinaison basée sur une liste prédéfinie ou sur la taille de l'ensemble des valeurs des dés, puis construction de la liste des dés à garder. Ces méthodes de reconnaissance sont plutôt intuitives, et ne reposent sur aucun principe mathématique vraiment complexe.

Voici par exemple le code de pour la reconnaissance du `Full` :

```
1  #Full
2      if 2 in L_compt: # L_compt[i-1] = le nombre de fois que i apparait dans
       les des
3          temp=[False for i in range(5)]
4          if 3 in L_compt: # Si il y a une Paire et un Brelan, le Full est
       atteint
5              L_assur.append(('Full',[True for i in range(5)]))
6          else: #Sinon on recherche une nouvelle paire en recherchant le
       prochain deux de L_compt
7              L_compt_temp=L_compt.copy()
8              i=L_compt_temp.index(2)
9              L_compt_temp[i]=0
10             if 2 in L_compt_temp:
11                 j=L_compt_temp.index(2)
12                 for k in range(5):
13                     if valeurDes[k] in [i+1,j+1]:
14                         temp[k]=True
15                 L_possib.append(('Full',temp))
```

Et un schéma illustrant sa mise en application :

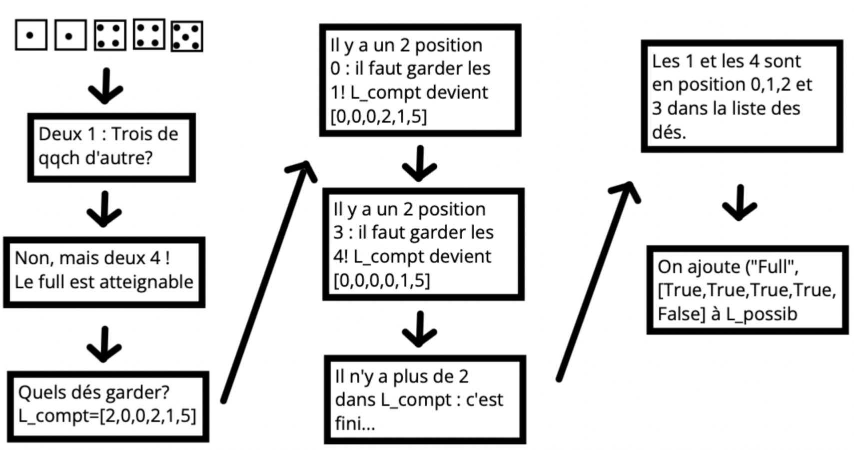

FIGURE 2 – Fonctionnement de `Possib` pour détecter les `Full` avec `L_compt`$[i]$ le nombre de fois que i apparaît dans les dés.

3.3 Seconde partie du programme.

Nous avons décidé de hiérarchiser les différentes relances possibles en les évaluant par deux fonctions `poids1` et `poids2`. L'algorithme cherche ensuite la décision qui maximise la somme de `poids1` et `poids2`, et la renvoie. La décision rendue lorsqu'il ne reste pas de relance n'est pas modifiée et se réfère toujours à la fonction `calcul_point`.

3.3.1 Poids1

Cette fonction a pour rôle de calculer les points que l'on peut obtenir si on réussit à remplir la case au terme des lancers de dés. Elle prend en argument la case que l'on veut jouer, les valeurs des dés actuels, et une liste index permettant de savoir combien de dés prennent déjà la bonne valeur pour la SECTION SUPÉRIEURE.

Pour l'ensemble des cases de la SECTION INFÉRIEURE, la fonction `poids1` renvoie le score attribué au succès de cette case par les règles du Yahtzee. Pour les cases de la SECTION SUPÉRIEURE, on choisit de prendre en compte le nombre de dés que l'on a déjà pour jouer la case, et la prime dans les cas où celle-ci semble encore atteignable dans la partie en cours. Ici, les points associés aux situations ne suivent pas réellement une logique par rapport au jeu de Yahtzee, mais une volonté de pondérer plus fortement cette section. En effet, l'ensemble de ces cases ne peuvent s'obtenir qu'avec une seule valeur pour tous les dés, alors que le Yahtzee et même le Brelan, le Carré ou le Full peuvent s'obtenir aussi bien en utilisant les 3 que les 5. Ainsi obtenir trois « 3 » permet de remplir la case Brelan et Trois, alors qu'obtenir un Brelan n'assure en pratique que rarement d'obtenir un bon score à Trois. Pour cette raison nous augmentons artificiellement le poids1 de la section supérieure, en favorisant toujours les grandes valeurs par rapport aux plus petites

3.3.2 Poids2

Cette fonction a pour rôle de donner une valeur à une décision selon le risque qu'elle engendre : plus sa probabilité de succès sera forte, plus la valeur attribuée à la décision sera grande. La fonction prend en arguments la case que l'on veut jouer, la même liste index que `poids1`, et le nombre de relances restantes. La valeur renvoyée est comprise entre 0 et 50. Nous avons globalement essayé de retranscrire les probabilités de succès pour une case, mais dans des cas simplifiés ou modifiés pour mieux pondérer les décisions.

Pour la SECTION SUPÉRIEURE, on attribue un nombre de points multiplié par le nombre de dés qui ont déjà la bonne valeur, le score par dés étant plus important lorsqu'il reste une relance que lorsqu'il en reste deux.

Pour la SECTION INFÉRIEURE, on distingue le Full, la Grande suite et la Petite suite du Carré, Brelan et Yahtzee. En effet pour les premières cases, seules deux valeurs de dés peuvent permettre de compléter la combinaison (les reconnaissances de Petite Suite et Grande Suite dans `Possib` ont été programmées de cette manière : $[1, 2, 3, 5, 5]$ ne sera pas reconnu comme une possibilité de Grande Suite). Pour la seconde famille de case, une seule valeur de dés est vraiment recherchée. On soustrait ensuite au score la probabilité d'échec $\times 10$ pour chaque dé à relancer lorsqu'il reste une relance, et la probabilité d'échec, et on divise le score obtenu lorsqu'il reste encore un relancer par 3 pour obtenir celui lorsqu'il reste deux relances (on considère que beaucoup de relances vont beaucoup améliorer notre situation avant d'arriver à une relance, pour éventuellement pouvoir jouer d'autres cases, ce qui explique que l'on divise le score par 3 et non par 2).

Voici ainsi une synthèse des poids de la SECTION INFÉRIEURE :

1. `Full`, `Petite suite`, `Grande suite` :
   — 2 relancers restants :

$$Poids2 = 50 - \frac{N_{DesManquants} \times \frac{2}{3} \times 10}{3} \tag{1}$$

   — 1 relancer restants :

$$Poids2 = 50 - N_{DesManquants} \times \frac{2}{3} \times 10 \tag{2}$$

2. `Carré`, `Brelan`, `Yahtzee` :
   — 2 relancers restants :

$$Poids2 = 50 - \frac{N_{DesManquants} \times \frac{5}{6} \times 10}{3} \tag{3}$$

   — 1 relancer restants :

$$Poids2 = 50 - N_{DesManquants} \times \frac{5}{6} \times 10 \tag{4}$$

3.3.3 Défauts de la stratégie

Cette première stratégie possède des lacunes assez évidentes au regard de notre connaissance du jeu aujourd'hui. En effet nous avons commis certaines erreurs lors de l'élaboration de cette stratégie. La partie allouée à la reconnaissance de combinaison était probablement trop stricte pour l'ensemble *atteignable*. En effet, toutes les cases de la SECTION SUPÉRIEURE ne sont pas *atteignables* si aucun dé de `valeurDes` n'a pour valeur le chiffre en question. Il est pourtant dommage de se priver de cette possibilité au moment du relancer des dès. Ce critère de reconnaissance pose surtout problème à la fin de partie, lorsque la case à cibler est plus importante que la valeur des dés. Il aurait également était intéressant de prendre en compte qu'une combinaison aurait pu être *atteignable* dans une situation donné lorsque il reste 2 relancers, mais pas s'il reste 1 relancer.

Un autre problème majeur est probablement le choix d'avoir multiplié `poids1` et `poids2` pour comparer les décisions entre elles. Cela a immédiatement tendance à éliminer les décisions qui rapporte peu de points même si celle-ci sont très peu risquées et vice-versa. Cela rend le poids attribué à chaque décision moins contrôlable, ce que des poids additifs n'auraient pas engendrés.

Pour finir, cette approche dite "à la main" ne rend compte que d'un nombre limité d'idées et n'est efficace que dans certaines situations parmi l'ensemble des situations possibles. Il est difficile de rendre compte en une dizaines d'idées et stratégies une manière de jouer au Yahtzee qui soit efficace dans toutes les situations de jeu, que ce soit par rapport à la feuille des scores ou à la valeur des dés.

4 Deuxième approche : Toutes_les_esperances

Passons maintenant à une approche un peu plus généraliste du jeu de Yahtzee. On va désormais chercher à maximiser le nombre de points que l'on obtient à l'issue de la partie. Ce nombre de points devra être vu comme la moyenne de points obtenus par la stratégie sur un grand nombre de parti (plus de 10000).On estime en effet que nous avons plus de chance de battre un adversaire si nous obtenons un score moyen plus important que notre adversaire. C'est pourquoi cette idée paraît intéressante.

L'idée de cette stratégie est de calculer les espérances de tous les cas de figure possibles. On considérera ainsi l'ensemble des premiers lancers de dés possibles, l'ensemble des groupes de cases qu'il peut rester à remplir pendant la partie, le nombre de relancers restants. Le but est de calculer toutes ces espérances et de les stocker dans une structure de données (`ESPERANCE`). Il restera ensuite à écrire une fonction (`decision_esperance`) qui pourra chercher dans cette structure de données à partir des informations sur la partie en cours afin de pouvoir déterminer la décision à prendre qui maximise l'espérance de point.

Afin de rendre le calcul réalisable en un temps raisonnable, on fera abstraction de la règle du bonus sur la SECTION SUPÉRIEURE qui augmenterait trop le nombre de calculs. On prendra cependant en compte la règle du JOKER. Estimons le nombre de situation différente qu'il faudra calculer pour cette stratégie :
- Le nombre de valeur de dés possibles : 252 (voir partie suivante)
- Le nombre de relancers restants (1 ou 2 relancers) : 2
- Application de la règle du JOKER (JOKER réalisable ou non) : 2
- Le nombre de relancers possibles (2NDES) : 32
- Le nombre de possibilités pour la feuille de score (la case est elle déjà remplie ou non parmi les 13 cases de feuilleScore : $2^{13} = 8192$

Cela donne donc un total de : $252 \times 2 \times 2 \times 32 \times 8192 = 2,6 \times 10^8$
Chacune de ses situations représentant un certain nombre de calculs, nous avons donc du négliger la PRIME pour ne pas faire exploser ce nombre.

4.1 Première étape : récupérer l'ensemble des combinaisons

Pour mettre en oeuvre cette approche, il nous a fallu dans un premier temps récupérer l'ensemble des combinaisons de dés. Ce travail nous a d'ailleurs permis de répondre à la question posé par M. Peyre : "Montrer qu'il y a 252 combinaisons de dés".L'ensemble de ces combinaison de dés est donc représenté par la combinaison triée dans l'ordre décroissant.

Nous avons dans un premier temps calculé l'ensemble des $NFACES^{NDES}$ NDES-uplets à valeurs dans $[\![1, NFACES]\!]$, sous forme d'une liste de listes nommée par la suite `TCOMBI`, que nous avons triées selon la méthode "tri-bulle" et transformées en chaînes de caractères afin de s'en servir plus tard comme dictionnaire. Nous avons ensuite filtré la grande liste

en comparant les valeurs (sous forme de caractères) pour éliminer les combinaisons de dés redondantes et obtenir au final 252 listes dans le cas $NDES = 5$ et $NFACES = 6$.

On notera par la suite `NCOMBI` = 252 le nombre de combinaisons.

4.2 Représentation des groupes de cases à jouer

Il est nécessaire de pouvoir indicer l'ensemble des groupes de cases qu'il reste à jouer dans une partie, afin de pouvoir stocker de l'information dans des tableaux indicés par des entiers. Nous avons choisi de représenter les cases par des entiers, en suivant l'ordre dans lesquelles ils apparaissent dans la feuille de score. Ainsi, $[As, Deux, \ldots, Yahtzee, Chance]$ seront indicés par $[0, 1, \ldots., 11, 12]$. S'il reste à compléter DEUX, QUATRE, YAHTZEE, on représentera cette situation par le 3-uplet $(1, 3, 11)$. Les n-uplets seront toujours triés dans l'ordre décroissant, afin d'avoir un unique représentant par situation dans le jeu.

Cette représentation ne permet toujours pas de retrouver efficacement dans des tableaux de l'information. Nous avons donc utilisé un dictionnaire qui associe à chaque n-uplet un entier, permettant de retrouver la bonne information parmi toutes les autres informations déjà calculées. Deux n-uplets de même longueur ne pourront être associés au même entier. Si on note k_n le nombre de n-uplet de longueur n, pour $n \in [1, 13]$, les entiers associés à ces n-uplet seront compris dans $[0, k_{n-1}]$.

Afin de pouvoir rendre transposable nos algorithmes à des règles du jeux différentes, nous avons créé ce dictionnaire, nommé `INDICE_COUPLE`, de la manière suivante :

1. On crée un dictionnaire initial tel que $DICO[(i,)] = i, \forall i \in [0, N_CASE - 1]$
2. On procède par récurrence en construisant le dictionnaire indiçant les k-uplets à partir du dictionnaire des $(k-1)$-uplets, en ajoutant aux $(k-1)$-uplets les entiers qui n'appartiennent pas encore à ce $(k-1)$-uplet et ceux pour toutes les clés du dictionnaires des $(k-1)$-uplets. Nous avons pris garde à ne pas ajouter deux fois le même k-uplet. (fonction `indicage_suivant`)
3. Une fois le dictionnaire des k-uplets construit, on l'ajoute à l'ensemble des n-uplets déjà présent dans le dictionnaire.

Cette méthode permet de pouvoir créer la liste des situations sans pour autant connaître explicitement l'entier représentant le nombre de case à jouer.

4.3 Les structure de données

La structures de données que nous avons choisi pour stocker les espérances est une liste de 12 tableaux numpy, nommé `ESPERANCE` (tout n'est pas sous un même tableau numpy pour des problèmes de dimensions). Le n^{eme} tableau numpy est construit de la façon suivante :

1. Le tableau est de dimension (`NCOMBI`, k_n ,2,2,2).

2. Le premier indice du tableau correspond à la combinaison de dés obtenue à l'issus du premier lancer d'un joueur. Ces combinaisons sont indicées par `TCOMBI` et `NCOMBI` = 252 dans les règles classiques du Yahtzee. Chaque combinaison (au nombre de $NFACE^{NDES}$) est représentée par la combinaison triée dans l'ordre décroissant ce qui réduit le nombre de combinaison différente à `NCOMBI`.
3. Le second indice du tableau correspond au groupe de cases restantes à jouer. Chaque entier compris entre 0 et k_{n-1} représente un n-uplet. Cette association est stockée par `INDICE_COUPLE`
4. Le troisième indice indique si le joker est jouable ou non. Dans le cas ou cet indice vaut 1, cela signifie que le Yahtzee a été réussi précédemment dans la partie. Si l'indice vaut 0, le Yahtzee n'a pas encore été réussi ou le jouer a choisi de remplir cette case avec 0.
5. Le quatrième indice correspond aux nombres de relancers restants, 0 s'il en reste un et 1 s'il en reste deux.
6. Enfin la dernière dimension contient l'information que l'on souhaite stocker. Il s'agit d'une deux-liste. Le premier élément de cette deux-liste est l'espérance maximale en termes de score parmi les 32 relancers qui s'offrent aux joueurs. Le second élément est le relancer à effectuer pour atteindre cette espérance. Ce second élément est un entier entre 0 et 31. Cette entier, s'il est écrit en base binaire, représente le relancer à effectuer. Ainsi 14 est associé à $[0, 1, 1, 1, 0] \rightsquigarrow [\text{False}, \text{True}, \text{True}, \text{True}, \text{False}]$.

`ESPERANCE` est une variable globale que l'on remplit au fur et à mesure à l'aide de différentes fonctions.

4.4 Calcul des espérances lorsqu'il reste une case à jouer

Le calcul de toutes les espérances du jeu va se faire par dénombrement des situations, la seule chose que nous sachions calculer pour l'instant étant le score obtenu pour une case et pour une valeur de dés.

Le cas où il reste une seule case à jouer doit être distingué du reste des situations. En effet, on peut calculer l'espérance pour chaque case de manière indépendante, en dénombrant l'ensemble des combinaisons possibles à partir d'un relancer et en calculant l'espérance en faisant une moyenne des scores sur toutes ces combinaisons possibles. De plus chaque espérance dépend de ce qui à été calculé précédemment. Ainsi on devra calculer l'espérance pour un relancer

Nous avons créé une fonction `combipossible` qui prend en entrée une combinaison de dés et un choix des dés à garder sous forme d'une *NDES*-liste de 0 (dés à relancer) ou de 1 (dé à garder), et qui renvoie toutes les combinaisons qu'il est possible d'obtenir en relançant les dés indiqués. on notera $N_{combipossible}$ le nombre de ces combinaisons.

Lorsqu'il reste un relancer, pour une valeur de dés, pour chaque relancer, l'espérance pour une case s'écrit :

$$\mathbb{E}_{valeurDes,relancer,1}(CASE) = \frac{\sum_{i \in combipossible(valeurDes,relancer)} Score(i, CASE)}{N_{combipossible}} \tag{5}$$

Où le score est calculé à l'aide de `defiyahtzee.Calcul_score` fourni par M Peyre, et les combinaisons de dès sont obtenues par `combi_possible`. On retiendra ensuite la meilleure des espérances parmi les 32 calculées :

$$\mathbb{E}_{valeurDes,1}(CASE) = max(\mathbb{E}_{valeurDes,relancer,1}(CASE))_{relancer \in [0,31]} \tag{6}$$

Pour la case du Joker, le calcul ne change pas, si ce n'est que le score peut changer lorsque la combinaison contient 5 dés identiques. On ajoute alors 100 au score dans ces cas spécifiques.

Lorsqu'il reste deux relancers, on peut utiliser les espérances calculées dans la phase précédentes. C'est pourquoi nous calculerons d'abord `esperance_rel1` avant d'utiliser `esperance_rel2` qui calcule les espérances lorsqu'il reste deux relancers.

L'espérance dans ce cas peut s'écrire pour une valeur de dés et pour un relancer comme :

$$\mathbb{E}_{valeurDes,relancer,2}(CASE) = \frac{\sum_{i \in combipossible(valeurDes,relancer)} \mathbb{E}_{i,1}(CASE)}{N_{combipossible}} \tag{7}$$

avec `combipossible` l'ensemble des combinaisons que l'on peut obtenir avec la fonctions `combi_possible(valeurDes,relancer)`.
L'espérance maximale est encore une fois la meilleure parmi les 32 calculées :

$$\mathbb{E}_{valeurDes,2}(CASE) = max(\mathbb{E}_{valeurDes,relancer,2}(CASE))_{relancer \in [0,31]} \tag{8}$$

Pour le joker, le calcul ne diffère pas mais il faut considérer les espérances calculées avec la règle du joker lorsque l'on fait la moyenne (valeur 1 pour le troisième indice dans (ESPERANCE)).

C'est en suivant ces méthodes de calcul que l'on remplit le premier tableau dans ESPERANCE..

4.5 Calcul des autres espérances

Passons maintenant au calculs des autres espérances. Les espérances sont calculés par taille croissante de groupe de case à jouer (N-UPLET). On notera $\mathbb{E}_{global}(N_UPLET)$ les espérances globales d'un coup lorsqu'il reste (N_UPLET) case à jouer, avant même d'avoir lancer le moindre dés. C'est l'espérance avant de jouer, sachant les cases qu'il reste à remplir. Pour calculer $\mathbb{E}_{global}(N_UPLET)$, il est nécessaire de faire la moyenne des espérances pour toutes les valeur de dés possible (En prenant en compte que $[1, 2, 3, 4, 5]$ est une valeur de

dés qui apparaît beaucoup plus souvent que $[1, 1, 1, 1, 1]$. On peut maintenant définir les espérances lorsqu'il reste plusieurs case à jouer :

$$\mathbb{E}_{valeurDes,i}(N_UPLET) = max(\mathbb{E}_{valeurDes,i}(CASE) + \mathbb{E}_{global}(N_UPLET \backslash CASE) pour CASE \in N_UPLET \quad (9)$$

L'idée est ici qu'à chaque coup, on regarde quel est le meilleur compromis entre l'espérance du coup à jouer et l'espérance de l'ensemble des coups restants dans la partie.

4.6 Retour sur les règles annexes, le Joker et le Bonus

Afin de diminuer le temps de calcul, nous n'avons pas calculé les espérances avec le Joker lorsque celui-ci n'était pas jouable. Pour chaque n-uplet, il est possible de le calculer en cherchant les informations au bon endroit (1 pour le troisième indice). Lorsque le Yahtzee fait partie des cases du n-uplet, le joker est par définition non jouable. Il est alors nécessaire de stocker les mêmes informations dans la partie du tableau consacré au joker et celle où cette règle n'est pas applicable.

En ce qui concerne le bonus, cette règle n'est pas prise en compte dans cette méthode de jeu du Yahtzee. Il s'agit probablement de la plus grande faiblesse de cette approche. En effet le bonus n'est que très rarement réalisé lorsque l'on observe de quelle manière joue `decision_esperance`. Le calcul du tableau `ESPERANCE` et de `ESP_CASE`, `ESP_CASEJ` prend environ 10 minutes. Il serait nécessaire d'ajouter un indice à `ESPERANCE`, avec des indices allant de 0 à 65 (seuil de validation du BONUS) ce qui multipliera au moins le temps de calculs par 65. En réalité, comme il est nécessaire de savoir de combien on a augmenté le score de la SECTION SUPÉRIEURE pour passer de l'étape n-1 à n, il faudrait probablement revenir au dénombrement de toutes les combinaisons lorsqu'il reste 1 relancer, ce qui augmenterait encore fortement le temps de calcul.

4.7 Etape finale : jouer

Une fois tous les tableaux créés, on appelle une fonction nommée`decision_esperance` qui prend en entrée la 5-liste des valeurs des dés, le nombre de relancers restant au tour, ainsi que les feuilles de score des deux joueurs, et qui renvoie la case à jouer. Cette fonction sert à traduire les différentes informations contenues dans les variables d'entrées afin de retrouver l'espérance mais surtout le relancer à effectuer. Celui-ci est calculé pour une liste `valeurDes` triée, il est donc nécessaire de modifier la décision pour la faire correspondre à la `valeurDes` d'entrée (à l'aide de la fonction `reordonner`)

Dans le cas où il ne reste pas de lancer, on utilise la fonction de calcul de score classique et les valeurs contenues dans `ESP_CASE` ou `ESP_CASEJ`. La méthode de calcul est la même que dans `esperanceN`, mais cette fois-ci, il n'y a plus d'aléa sur la valeur des dés. On se contente de regarder quelle case maximise $Score(CASE) + \mathbb{E}(N_UPLET/CASE), \forall\ CASE \in N_UPLET$.

5 Efficacité des deux premiers programmes

5.1 Premiers tests

Decision_Une se basant sur un raisonnement humain et limité, on s'attend à ce que Toutes_les_esperances soit plus performant. Nous avons donc dans un premier temps fait jouer nos deux algorithmes l'un contre l'autre (100000 parties). En appliquant la méthode de Monte-Carlo, on peut alors déterminer l'écart moyen en points entre les deux algorithmes. Toutes_les_esperances l'emportait 53,2±0,3% du temps avec un écart de 58,71±0,36 points lors des victoires, ce qui nous a conforté dans l'idée que cet algorithme devait être meilleur.

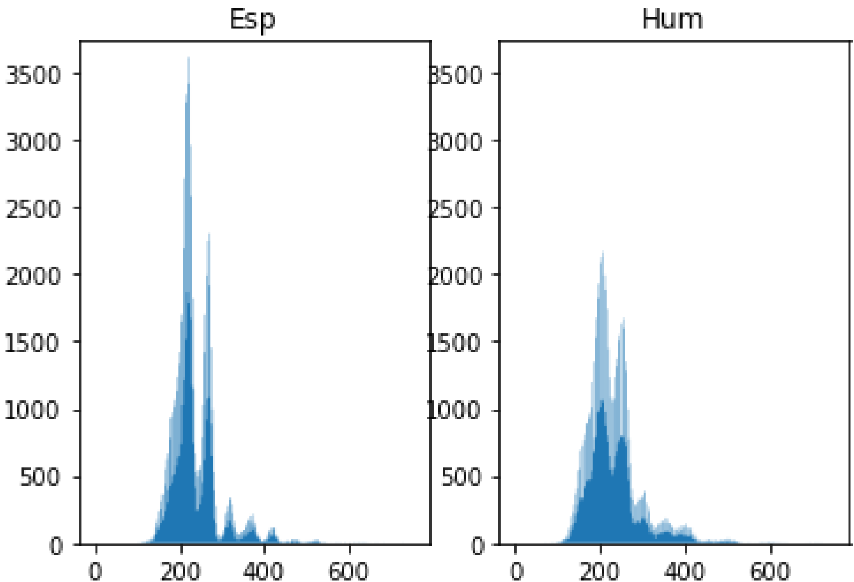

Figure 3 – Histogramme des scores de Decision_Une (hum) et Toutes_les_esperances (esp) pour 100000 sets

Algo	Score moyen	1er quartile	Médiane	3eme quartile
Esp	236.84 ± 0.33	208.0	224,0	264,0
Hum	230.95 ± 0.38	193,0	219,0	256,0

Ecart-type	Ecart moyen des scores quand victoire
52.96	58.71
60.74	60.22

5.2 Méthode de couplage

5.2.1 Première méthode de la réduction de la variance : couplage

Nous avons ensuite confronté les deux algorithmes en les faisant s'affronter dans 100000 parties "couplées", c'est-à-dire dans des parties ou les deux algorithmes reçoivent les mêmes

dés. Par exemple, considérons que les deux algorithmes obtiennent la "liste-destin" $[1, 3, 4, 2, 1, 4, 5, 5, 2, ...]$. Les premiers lancers donneront $[1, 3, 4, 2, 1]$. Les relancers suivants seront pris dans la liste dans l'ordre. Si un algorithme souhaite relancer 2 dés, les numéros 4 et 5 sortiront.

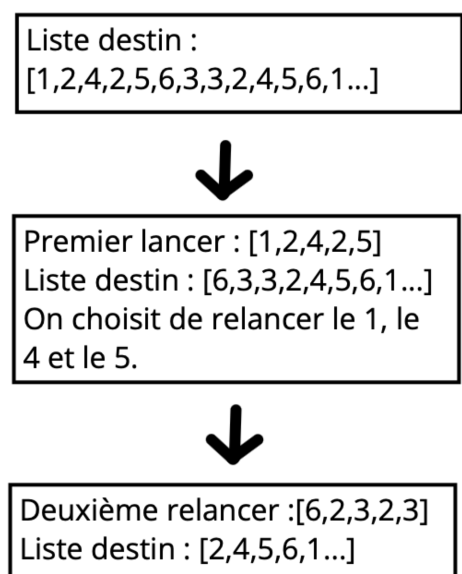

FIGURE 4 – Illustration du fonctionnement des "listse-destin"

5.2.2 Résultats et statistiques

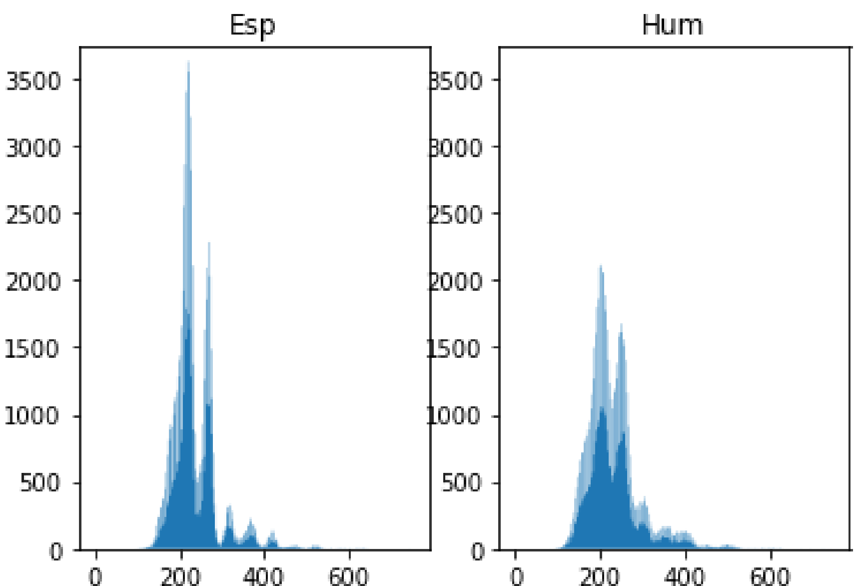

FIGURE 5 – Histogramme des scores de `Decision_Une` (hum) et `toutes_les_esperances` (esp) pour 100000 parties

Algo	Score moyen	1er quartile	Médiane	3eme quartile
Esp	236,94±0,33	208,0	224,0	264,0
Hum	231,11±0,38	193,0	219,0	256,0

Ecart-type	Ecart moyen des scores quand victoire
53,14	37,58
60,56	41,57

Ces statistiques nous assurent donc bien que `toutes_les_esp` est plus performant que `Decision_Une`, avec notamment une probabilité de victoire de 0,594±0,003 % (dans ces conditions) et un écart moyen de 37,58±0,23 en score. L'algorithme `toutes_les_esperances` possède une meilleure moyenne, et ses résultats sont aussi plus condensés, avec un écart-type plus faible : l'algorithme est globalement plus stable que `Decision_Une`.

On remarquera que l'écart moyen des scores lors des victoires a diminué, et de même que l'intervalle de confiance (réduction de la variance).

5.3 Variable antithétique

5.3.1 Principe

Nous avons enfin appliqué dans la continuité de la méthode de Monte-Carlo une autre technique de réduction de la variance suggérée par M. Peyre, à savoir par variable antithétique.

Nous avons fait affronté nos deux algorithmes dans un match composé de plusieurs sets de deux parties. Pendant la première partie, les deux algorithmes jouent normalement, avec des combinaisons de dés différentes. Lors de la deuxième partie, nous avons fait jouer les algorithmes avec les combinaisons de dés qu'a reçu l'autre lors de la première partie. Cette méthode, inspirée du "Top Chess Engine Championship", nous permet aussi de nous rapprocher de la réalité et de mieux coller à l'idée de "performance" introduite plus tôt.

Pour compter les scores en termes de victoire, on dira qu'une victoire est totale si un algorithme bat l'autre sur les deux parties d'un set, et qu'une victoire est partielle quand il ne gagne qu'une seule fois, avec un score moyen sur le set supérieur à l'autre. Une victoire totale rapporte un point, et une victoire partielle pour 0,5. Sinon, aucun point n'est attribué.

Une fois le principe des variables antithétiques posé et celui d'un match expliqué, il nous reste à savoir ce que l'on doit mesurer pour que l'apport des variables antithétiques soit pertinent. Ce qui est en fait assez compliqué. Si l'on veut montrer que `Toutes_les_esperances` est meilleur que `Decision_Une`, il nous faut une donnée mesurée sur le premier algorithme et par rapport aux performances de `Decision_Une`. Nous n'avons malheureusement pas réussi à en trouver une autre que la probabilité de victoire, à cause du caractère des algorithmes. Pour les statistiques déjà étudié, à part pour la probabilité de victoire, la méthode de la variance antithétique revient à augmenter le nombre de parties. Nous avions envisager d'étudier l'écart des scores lors des victoires (qui dépend par définition du score des performances de chaque algorithme), mais cela aurait nécessité qu'il y ait autant de matchs-aller gagnés que de matchs-retour remportés.

Cependant, cette méthode a permis de mettre en lumière un autre phénomène.

100000 sets de deux parties, soit 200000 parties, ont été simulés dans ce match.

5.3.2 Résultats et statistiques

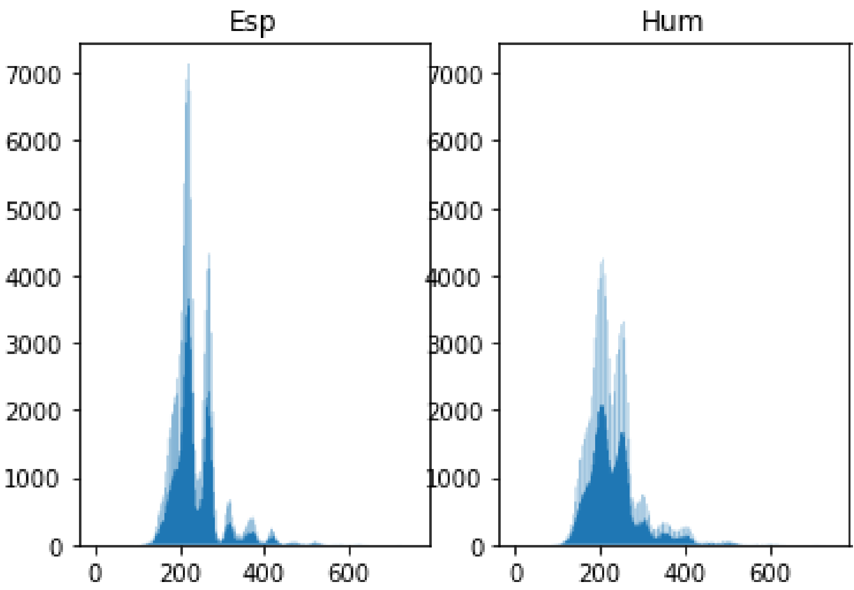

Figure 6 – Histogramme des scores de `Decision_Une` (hum) et `Toutes_les_esperances` (esp) pour 100000 sets

Algo	Nombre de victoires totales	Nombre de victoires partielles
Esp	22456	36974
Hum	11540	27154

Résultat :

- Esp a gagné 40943,0 à 25117,0.
- Au risque de 5%, sur l'ensemble des parties jouées, `Toutes_les_esperances` (esp) bat `Décision_Une` (hum) avec une probabilité de 0.552±0.003 et avec un écart de 11.98±0.35.
- Au risque de 5%, hum bat esp avec proba 0.442±0.003 et avec un écart de -11.70±0.35 points.

La variance de la probabilité de victoire a bien été réduite.

6 Troisième approche : `Perturbation`

Cette dernière approche prendra en compte deux aspects négligés jusqu'ici : la PRIME alloué à la SECTION SUPÉRIEUR, et l'état de la partie de l'adversaire. Pour ce qui concerne la PRIME, celle ci n'a pas été prise en compte dans le calcul des espérances pour une question

de temps de calcul, il est donc nécessaire de l'inclure en encourageant certaines prises de décision sur les cases de la SECTION SUPÉRIEUR.

En incluant la règle de la prime, on pourrait se dire que l'algorithme est le plus performant possible, dans le sens où la stratégie mise en place maximise le nombre de points moyen obtenu sur un grand nombre de partie, et qu'il n'est donc plus possible de trouver une stratégie meilleure. Cependant, cette stratégie ne prends pas en compte la manière de jouer de l'adversaire, et si celui-ci est en bonne ou mauvaise posture dans une partie en cours. En effet en cas de retard trop conséquent dans une partie, la meilleur stratégie consiste à minimiser son espérance de points, tout en augmentant ses chances de marquer beaucoup de points (en forçant le Yahtzee ou les suites par exemple). Dans le cas où au contraire, nous aurions beaucoup d'avance sur notre adversaire, on cherchera à limiter la variance dans la partie, et donc à limiter les décisions risquées pour obtenir un nombre de points suffisants pour battre l'adversaire. Le but de cette partie est de mettre en place ces stratégies afin d'augmenter la performance de notre stratégie.

Maintenant que nous considérons les stratégies les unes par rapport aux autres, regarder le nombre de points moyen obtenus en fin de partie n'est plus suffisant pour évaluer les performances de notre algorithme. Nous devons donc regarder le taux de victoire de notre stratégie contre d'autres stratégies. Nous réaliserons cette évaluation par rapport à la stratégie de notre deuxième approche, qui maximise l'espérance des décisions. Le fait qu'une stratégie batte une autre n'assure pas que cette stratégie domine l'ensemble des stratégies. Ainsi on pourrait trouver des stratégie particulièrement adaptées pour battre une autre stratégie.

6.1 Nouvelles évaluations des décisions

Pour construire cette nouvelle stratégie, nous repartons de notre ancienne stratégie et notamment de la structure de données qui contient les espérances dans chacune des situations possibles du jeu. Nous ajoutons maintenant une somme de fonctions prenant en paramètres les différents éléments propres à chaque situations du jeu. Cette somme de fonctions à pour but de changer les décision dans certaines situations précises où l'approche par les espérances était défaillantes, en pondérant plus fortement certains relancers ce qui va combler le retard en terme d'espérances pour ces dits-relancers. On a donc :

$$\mathbb{E}'(Decision) = \mathbb{E}(Decision) + \sum_i \alpha_i f_i(Decision) \tag{10}$$

où les f_i sont des fonctions qui représentent les différentes modifications des règles à prendre en compte et les α_i leurs pondérations.

Nous avons écrit 12 fonctions f_i :

1. 6 fonctions dont le but est de "forcer" les décisions vers la prime. Chacune des ces fonctions reposent sur une condition de "jouabilité" de la prime. En effet, ces fonctions ne renverront pas 0 seulement dans le cas où le score actuelle sur la SECTION

SUPÉRIEUR permet d'espérer atteindre SEUILPRIMESUP (65 points) à la fin de la partie. Bien évidemment, si ce seuil est déjà dépassé, ces fonctions renvoient 0, la prime n'étant plus un objectif dans la partie. On adopte alors un comportement différent en fonction du nombre de case à remplir dans SECTION SUPÉRIEUR (respectivement 1,2,3,4). Nous attribuons un score plus important aux relancers qui permettent de compléter les cases manquantes dans la SECTION SUPÉRIEURE. Il y a également deux fonctions parmi ces 6 qui interviennent au moment de rendre la décision finale de la case à jouer. En effet, le travail réalisé par les autres fonctions n'auraient pas beaucoup de sens si la SECTION SUPÉRIEURE n'étaient pas également favorisé au moment de choisir la case à remplir

2. 6 fonctions autour de l'écart de points entre les deux joueurs au cours d'une partie. Ces fonctions n'interviennent que dans les cas où l'écart est suffisamment important entre les deux joueurs (¿50 points). Dans certains cas il s'agit de conserver son avance, dans d'autres il s'agit de tenter de rattraper un retard conséquent. Lorsque l'adversaire à plus de 50 points de retard, on cherche à relancer le moins de dés possible en assurant d'avoir une combinaison à la fin des lancers. Lorsque l'adversaire à plus de 50 points d'avance, on cherche à conserver les dés dont la valeur est la plus présente parmi les 5 dés. On cherche également à favoriser 3 cases (Grande Suite, Petite Suite et Yahtzee). La encore, il est nécessaire d'avoir des fonctions au niveau du choix de relancer et au niveau du choix de case.

6.2 Optimisation des α_i

Une fois l'ensemble des fonctions créées, il est nécessaire de déterminer les paramètres α_i qui maximisent le taux de victoire contre `Toutes_les_esperances`. Pour ce faire, nous avons d'abord regardé individuellement chaque fonction f_i pour voir si il existait une zone tel qu'avec

$$\mathbb{E}'(Decision) = \mathbb{E}(Decision) + \alpha_i f_i(Decision) \tag{11}$$

il existe un intervalle $[\alpha_{inf}, \alpha_{sup}]$ pour lequel le taux de victoire de cette dernière stratégie était supérieur à 50 %. Dans le cas contraire, la fonction n'était pas retenue/était modifiée.

Le problème étant que l'ajout d'une fonction au modèle modifie assez fortement les autres coefficients. En effet le modèle que nous avons choisi et additif, et si $\sum_i \alpha_i f_i(Decision)$ prends des valeurs trop grandes, le taux de victoire de la stratégie chute brusquement. Pour une mauvaise famille $(\alpha_1, ..., \alpha_{12})$, le taux de victoire de la nouvelle stratégie peut être inférieur à 40 % . L'ajout d'une nouvelle fonction au modèle entraîne souvent une diminution de la valeur des paramètres α_i. Il n'est donc pas possible de calculer les α_i au fur et a mesure en ajoutant des fonctions.

Il est donc nécessaire de trouver $(\alpha_1, ..., \alpha_{12})$ "d'un seul coup". Pour ce faire, nous n'avons pas réussi à mettre en place des techniques de régression qui permettraient d'obtenir les α_i maximisant le taux de victoire de la stratégie. En effet pour effectuer ce type de technique, il

faut un nombre suffisant de points (au moins 10 fois plus que le nombre de paramètre), et le calcul du taux de victoire pour $(\alpha_1, ..., \alpha_{12})$ nécessite au grand minimum 100 000 parties pour obtenir une précision suffisante sur le taux de victoire (+ 0.2%). Sachant que la simulation de 100 000 parties prend environ une demi-heure, il faudrait donc environ deux jours pour le calcul des points. Et ce pour une précision qui n'est de plus que peu satisfaisante.

Nous avons donc procédé par "marche aléatoire". Voici la méthode que nous avons utilisé :

1. Choix d'une valeur initiale pour $(\alpha_1, ..., \alpha_{12})$
2. t = $tauxdevictoire(\alpha_1, ..., \alpha_{12})$
3. Tant que $(\alpha_1, ..., \alpha_{12})$ n'améliore pas t ou que $t < objectiftaux$:
4. $(\alpha_1, ..., \alpha_{12}) = (\alpha_1, ..., \alpha_{12}) + (r_1, ..., ...r_{12})$ (les r_i sont choisis aléatoirement dans un intervalle)

A noter que $objectiftaux$ était souvent défini autour de 0.52%, et que l'intervalle dans lequel on prend les r_i peut varier. En effet on autorise plus de variations après une amélioration de t, pour affiner l'intervalle à chaque tour. En effet, on laisse la chance de sortir d'un maxima locale avec cette variation de l'intervalle, tout en affinant $(\alpha_1, ..., \alpha_{12})$ à chaque itération.

6.3 Résultats et analyse

Nous avons repris la méthode des variables antithétiques et fait jouer `Perturbation` contre `Decision_Une` et `Toutes_les_esperances` sur 1 match de 100000 sets, soit 200000 parties.

6.3.1 Contre `Decision_Une`

Au risque de 5%, `Perturbation` bat `Decision_Une` avec une probabilité 0.553±0.003 et avec un écart moyen de 10.674±0.349 points et `Decision_Une` bat `Perturbation`. `Perturbation` gagne 41606,0 à 25844,0.

Algo	Nombre de victoires totales	Nombre de victoires partielles
Hum	11649	28391
Perturb	23018	35176

On a bien la version améliorée de `Toutes_les_esperances` qui continue de dominer `Decision_Une`.

6.3.2 Contre `Toutes_les_esperances`

En termes de parties simples, `Perturbation` a gagné 99529 fois contre 98618, avec 1855 parties nulles. Au risque de 5%, `Perturbation` bat `Toutes_les_esperances` avec une probabilité de 0.497±0.003 et avec un écart de -1.44±0.32 points, et `Toutes_les_esperances` bat `Perturbation` avec une probabilité de 0.493±0.003 et avec un écart de 0.900±0.324 points.

La probabilité est certes inférieure à 50%, mais il faut prendre aussi en compte toutes les parties nulles (que nous pouvons en fait ne pas prendre en compte dans la définition de performance, ce qui donne 50,2% sans les matchs nuls).

Algo	Score moyen	1er quartile	Médiane	3eme quartile
Esp	236.76 ± 1,54	208.0	224,0	264,0
Perturb	236.43 ± 1,53	193,0	219,0	256,0

Ecart-type	Ecart moyen des scores quand victoire
52.86	0,66±0,32
51,53	0,53±0,32

Algo	Nombre de victoires totales	Nombre de victoires partielles
Esp	4693	27428
Perturb	4985	32649

Cette nouvelle version de `Toutes_les_esperances` apporte peu à la "performance" telle que nous l'avons défini et en finalement très proche (l'écart des probabilités de victoire n'est pas significatif). Cependant, dans le cadre des variables anti-thétiques, `Perturbation` domine clairement son adversaire, ce qui rend au final l'ajout de ces 12 fonctions pertinentes. La probabilité de victoire aurait pu être amélioré si nos avions trouvé les α_i optimaux.

On remarque aussi que `Perturbation` fait plus de victoires totales contre `Decision_Une` que contre `Toutes_les_esperances`.

7 Gestion du projet et conclusion

Nous avons travaillé sur `Decision_Une` d'octobre à début décembre 2021, puis à partir des vacances de Noël sur `Toutes_les_esperances` jusqu'à début mars. Puis de fin mars à début juin, nous nous sommes attelés à la troisième approche `Perturbation`. Nous nous étions fixé la règle d'un après-midi consacré au projet, à rattraper sur une autre semaine en cas de problème ou d'imprévu. Nous avons aussi intensifié le travail certaines vacances, où nous nous retrouvions à distance quasiment un jour sur deux. Nous n'avons malheureusement pas réussi à maintenir un rythme de travail constant, notamment durant les mois de janvier et de mai . Nous pensons cependant pouvoir considérer que le volume de temps passé en terme de séances reste assez similaire (notamment grâce aux vacances).

La contrainte principale de ce projet, qui nous a notamment empêchés d'obtenir de meilleurs résultats pour `Perturbation` (nous pensons qu'un taux de victoire de 52/53 % aurait pu être atteignable par cette méthode) a été le temps. Le projet étant fortement tourné autour de l'aléatoire, il est nécessaire de simuler un grand nombre de parties, et ces simulations prennent un temps considérable. Ainsi chaque mauvaise idée, erreur de programmation coûte relativement cher en terme de temps. Nous avons de cette manière perdu 2 jours

de calculs en devant recalculer le tableau ESPERANCE pour des erreurs dans la partie allouée à la règle du joker lorsque il reste 1 relancer. Nous aurons au moins appris l'importance d'effectuer de petits échantillons avant de lancer une simulation de plus grande ampleur. Ce temps de simulation, même si il ne fait pas partie du temps de travail, nous a plutôt ralenti et demandé de la présence autour du projet.

Ce projet nous a aussi sur le plan de la gestion de projet permis de nous rendre plus précisément compte de la nature précise du travail d'ingénieur :

— Organiser et commenter son travail, grâce à des codes clairs et des variables nommées à la place de simples chiffres.
— Savoir communiquer et travailler en équipe, les forces de l'un compensant les lacunes de l'autre.