Travaux Encadrés de Recherche : Valeurs de Shapley

Gwennael CANNENPASSE
Antoine LEURIDAN

Encadrant : Joseph DESQUAIRES

Master 1 Informatique - TER - MIN15221
Année 2021 - 2022

20 mai 2022
Github du projet

TABLE DES MATIÈRES

I Introduction

Lors de jeux collaboratifs, des joueurs sont amenés à mettre en commun leurs ressources afin de maximiser un gain commun. Le problème est alors la répartition de ce gain entre les différents joueurs. Shapley a proposé une répartition "équitable" des gains de la coalition des joueurs[1].
Le but du projet est d'implémenter une modélisation de jeux collaboratifs afin d'étudier une répartition des gains utilisant les valeurs de Shapley.
Dans un premier temps, nous élaborons une modélisation de jeux collaboratifs simple.
Dans un second temps, nous implémenterons une méthode de répartitions des gains traditionnelle (Simple), une méthode utilisant les valeurs de Shapley et une méthodes utilisant une approximation des valeurs de Shapley.
Enfin, le tout nous permettra de comparer ces différentes méthodes en terme de vitesse et de précision de calcul mais aussi en terme d'équité.

II Modélisation

2.1 Joueurs et Boss

Notre modélisation est composée d'une coalition de joueurs et d'une entité Boss.
Un joueur est défini par son nom et possède un nombre de dégâts.
Un boss est lui défini par son nom mais aussi ses points de vie (PV) et son expérience (XP).
Chaque joueur participe exclusivement a la grande coalition[2].

2.2 Jeux

A tour de rôle, les joueurs vont "attaquer" le boss avec leur nombre de dégâts respectif. Chaque joueur ne peut attaquer qu'une seule fois. Si le total des dégâts mis au boss est supérieur a ses PV, alors il répartira son XP aux différents joueurs.

[1] Ensemble de joueurs réunie dans la poursuite d'un intérêt commun.
[2] La grande coalition correspond a la coalition de l'ensemble des joueurs.

III Les valeurs de Shapley

3.1 Qui était Lloyd Shapley ?

Lloyd Shapley (1923-2016) est un mathématicien et économiste américain. Il a contribué aux domaines de l'économie mathématique et surtout de la théorie des jeux. Il est considéré par de nombreux experts comme le plus grand théoricien des jeux depuis 1940. Sa thèse et son travail post-doctoral ont prolongé les idées de F.Edgeworth en introduisant en théorie des jeux **la valeur de Shapley**[3].

3.2 Théorème et Axiomes

En théorie des jeux, plus précisément dans un jeu collaboratif, les valeurs de Shapley donnent une répartition équitable des gains aux joueurs. La valeur de Shapley pour le joueur i est notée $\varphi_i(F)$ et est définie par :

$$\varphi_i(\mathcal{F}) = \sum_{c \subseteq J \setminus \{i\}} \frac{|c|!(n-|c|-1)!}{n!}(\mathcal{F}(c \cup \{i\}) - \mathcal{F}(c))$$

avec c : la grande coalition, n: le nombre de joueurs, $F(c)$: la fonction caractéristique du jeu[4].
Cette valeur correspond a la moyenne des contributions marginales de chaque joueurs dans toutes les coalitions possibles.

Les valeurs de Shapley, nous permettent de définir une répartition équitable des gains d'un jeu collaboratif. En ce sens, Shapley a énoncé au travers de 4 axiomes son théorème[5] :

- **Efficacité :** La somme des gains attribuées aux joueurs doit être égale à ce que la coalition de tous les joueurs peut obtenir.
- **Symétrie :** Deux joueurs apportant une même contribution dans toutes les coalitions dans lesquelles ils apparaissent, doivent percevoir le même gain.
- **Joueur nul :** Un joueur ne contribuant en rien au succès de chaque coalition ne doit percevoir aucun gains.
- **Additivité :** La répartition de l'union de deux collaborations doit être la somme des répartitions des deux collaborations : $\varphi_i(v+w) = \varphi_i(v) + \varphi_i(w)$

Shapley a prouvé qu'il existe une et une seule solution φ satisfaisant les axiomes.

[3] L. S. Shapley. "A value for n-person games". In: (1953).
[4] Michel PAUL et Hatem SMAOUI CEMOI Dominique LEPELLEY. *Introduction à la théorie des jeux (2) : les jeux coopératifs1.*
[5] frederic koessler. *7C-cooperatifs-Shapley-article.*

Dans notre études, nous abordons seulement la grande coalition où dans ce cas, la contribution marginale du joueur i selon l'ordre A, est définie par son apport à la coalition constituée des joueurs qui l'ont précédé.

3.3 Mise en situation

Voici nos données de base, on a donc 3 joueurs et 1 boss.

	Dégats	
Alice	2	
Bob	6	
Charlie	3	
Boss	10 HP	30 XP

Dans un premier temps, nous allons calculer l'ensemble des combinaisons d'attaques possibles de ces 3 joueurs.

Coalitions	Répartitions des dégâts
A, B, C	2, 6, 2
A, C, B	2, 3, 5
B, C, A	6, 3, 1
B, A, C	6, 2, 1
C, A, B	3, 2, 5
C, B, A	3, 6, 1

Dans un second temps, nous calculons les valeurs de Shapley de chaque joueur.
Alice effectue 2 de dégâts dans 4 sur 6 configuration et 1 de dégât dans 2 sur 6 configurations. Nous avons donc $2 * \frac{4}{6} + 1 * \frac{2}{6} = 1,67$.
Pour Bob, nous avons $6 * \frac{4}{6} + 5 * \frac{2}{6} = 5,67$.
Pour Charlie, nous avons $3 * \frac{4}{6} + 2 * \frac{2}{6} = 2,67$.
Enfin, il suffit de calculer la répartition de l'XP du boss en prenant en compte les valeurs de Shapley. Soit $Gains = \frac{ValeurdeShapley * XP}{HP}$.
Donc Alice obtient 5 XP, Bob obtient 17 XP et Charlie lui obtient 8 XP.

Dans notre modélisation, les valeurs de Shapley représentent les dégâts que chaque joueurs doit effectuer afin d'obtenir une répartition des gains équitable.

Nous observons que le temps de calcul va résider dans le nombre de joueurs et donc le nombre de combinaisons possibles d'attaques a traiter.

IV Implémentation algorithmique

Nous avons décidé d'implémenter notre modélisation avec le langage C++.
Nous souhaitions un langage de programmation orienté objet afin d'avoir une classe joueur où stocker ses informations(nom, dégât, gain).
De plus, nous souhaitions faire une interface graphique simple et épuré d'où l'utilisation de QT (notamment QT Designer qui nous permet de crée facilement des fenêtres graphiques).
Enfin, il nous fallait un langage de programmation avec de bonne performance de calcul au vu de notre sujet qui repose sur de la comparaison de précision et de temps de calcul d'algorithme.

4.1 Les algorithmes

Notre modèle est une attaque par des joueurs sur un boss unique, on veut que les dégâts des joueurs soient répartis de façon à ce que les gains après l'attaque soient équitable. Nous avons donc pensé à plusieurs algorithmes :

- Répartition simple : La première répartition qui nous ai venu à l'esprit est une répartition basée sur le pourcentage de participation de chaque joueur. Au sein de la grande coalition, chaque joueur effectue un pourcentage de dégât sur les dégâts totaux de la coalition. Chaque joueur va donc faire ce même pourcentage de dégât des points de vie du boss et percevra ce même pourcentage des gains.

- Répartition grâce au calcul des valeurs de Shapley : Dans cet algorithme, nous allons utiliser les valeurs de Shapley pour répartir les dégâts et les gains. Pour chaque joueurs, nous calculons sa valeur de Shapley et il percevra le pourcentage de dégâts fait par rapport aux points de vie du boss. Cette méthode permet d'ignorer l'ordre d'attaque de la grande coalition.

- Répartition grâce à une approximation des valeurs de Shapley : Cette méthode est similaire à celle des valeurs de Shapley en tout point, sauf qu'ici nous allons utiliser une approximation des valeurs de Shapley à la place des valeurs de Shapley exactes.

Nous allons dans cette partie voir comment nous les avons implémenté, dans l'objectif de les comparer par la suite.

4.2 Répartition simple

Ici comme nous l'avons évoqué ci-dessus, ce n'est pas des valeurs de Shapley que nous allons calculer mais un pourcentage de participation.
On calcul donc $\frac{d_i}{dt}$;
avec d_i les dégâts du joueur i et dt les dégâts totaux infligé au boss.
On obtient donc une fraction qui représente le taux de participation à l'attaque.
Plus on a de dégâts par rapport aux dégâts totaux plus notre taux de participation est élevé.
Les dégâts du joueur i sont calculés ainsi : $\frac{d_i}{dt} * bpv$;
avec bpv les points de vie du Boss.
Et les gains sont redistribués de la manière suivante : $\frac{d_i}{dt} * bpv$

Algorithme : On va dans un premier temps calculer les dégâts max en additionnant les dégâts de tous les joueurs, on va ensuite simplement assigner une valeur dans un tableau $S[n]$ à chaque joueur (avec n le nombre de joueurs), tel que $S[i] = \frac{d_i}{dt}$ (en reprenant les notations ci-dessus).

Algorithm 1: Répartition simple

Entrée : $J[n]$ un tableau de joueurs.
Sortie : $S[n]$ le tableau des valeurs de répartition.
Variables : $S[n]$, i et dt des entiers.
(avec n = nombre de joueurs)

```
begin
    for i ← 0 to n − 1 ;                          // Initialisation de dt
     do
        dt ← dt+ dégâts de J[i];
    end
    for i ← 0 to n − 1 ;                  // Calcul du taux de participation
     do
        S[i] ← dégâts de J[i]/dt;
    end
    return S;
end
```

Complexité : Ici la complexité est $O(n)$, avec n le nombre de joueurs. On a donc une complexité linéaire pour cet algorithme.

4.3 Calcul avec les valeurs de Shapley

Algorithme : Nous prenons un tableau a qui est un tableau d'indice. Chaque case de ce tableau a a une valeur entre 1 et n (avec n le nombre de joueurs). Ce tableau représente l'ordre d'attaque des joueurs, le joueur $a[0]$ attaque en premier, si $a[0] = 1$, le joueur1 attaque en premier, etc... Grâce a l'algorithme *QuickPerm*[6], ce tableau a aura les $n!$ combinaisons au cours de l'exécution du programme, ce qui nous permettra de calculer tous les ordres d'attaques possible et d'avoir la valeur de Shapley exacte pour chaque joueur. Cette valeur sera stockée dans un tableau $S[n]$, avec $S[i]$ la valeur de Shapley du joueur i. Ce tableau S va ensuite nous permettre de redistribuer les gains avec la formule suivante : $G_i = S[i]/bpv * bxp$; avec G_i le gain du joueur i, bpv les points de vie du boss et bxp l'expérience du boss.

Pourquoi QuickPerm ? Nous avions de base opté pour un algorithme récursif de permutation[7] mais nous nous sommes rendu compte après plusieurs recherches que ce n'était pas le plus performant et nous avons décidé d'implémenter QuickPerm qui est en général 2 fois plus rapide. En effet, l'algorithme initialement utilisé fait $n!$ fois un appel récursif, or QuickPerm fait $n!$ opérations seulement. C'est pourquoi on observe un gain considérable de temps.

Comment fonctionne QuickPerm ? QuickPerm est un algorithme de permutation qui utilise un tableau $p[n]$, avec n le nombre d'éléments a permuter. Au début du programme, le tableau p est initialisé et va contrôler les itérations faites sur le tableau a, qui on le rappel est un tableau d'indice. On va ensuite faire $n * (n-1)!$ opérations, qui vont permuter les éléments du tableau a. Grâce à cette suite d'opérations, le tableau d'indice a est permuté avec à chaque fois une nouvelle permutation sur les $n!$. Nous avons donc bien un algorithme qui effectues $n!$ opérations avec une complexité de $O(n!)$ qui permet d'obtenir toutes les permutations d'un tableau.

[6] Phillip Paul Fuchs. *QuickPerm.*

[7] GeeksForGeeks. *Write a program to print all permutations of a given string.*

Algorithm 2: Répartition avec les valeurs de Shapley

Entrée : $J[n]$ un tableau de joueurs et Bpv les points de vie du boss.
Sortie : $S[n]$ le tableau des valeurs de Shapley.
Variables : $S[n]$, $a[n]$, $p[n+1]$, i, j, tmp et pvr des entiers.
(avec n = nombre de joueurs)

---

begin
  for $i \leftarrow 0$ *to* $n-1$; `// Initialisation des tableaux` a `et` p `(QuickPerm)`
  do
    $a[i] \leftarrow i+1$;
    $p[i] \leftarrow i$;
  end
  $p[n] \leftarrow n$;
  $pvr \leftarrow Bpv$;
  for $i \leftarrow 0$ ***to*** $n-1$; `// Calcul pour la permutation initiale`
  do
    if *dégâts de* $J[a[i]-1] \geq pvr$; `// [1]`
    then
      $S[a[i]-1] \leftarrow S[a[i]-1]$ + dégâts de $J[a[i]-1]$;
      $pvr \leftarrow pvr$ - dégâts de $J[a[i]-1]$;
    else
      $S[a[i]-1] \leftarrow S[a[i]-1] + pvr$;
      break ; `// [2]`
    end
  end
  $i \leftarrow 1$;
  while $i < n$; `// Génération des permutation et calculs`
  do
    $p[i] \leftarrow p[i]-1$; `// Génération des permutations (QuickPerm)`
    $j \leftarrow i \bmod 2 * p[i]$;
    $tmp \leftarrow a[j]$; `// Swap` $a[i]$, $a[j]$
    $a[j] \leftarrow a[i]$;
    $a[i] \leftarrow tmp$;
    $pvr \leftarrow Bpv$;
    for $i \leftarrow 0$ ***to*** $n-1$; `// Calcul pour la permutation généré`
    do
      if *dégâts de* $J[a[i]-1] \geq pvr$ **then**
        $S[a[i]-1] \leftarrow S[a[i]-1]$ + dégâts de $J[a[i]-1]$;
        $hpr \leftarrow pvr$ - dégâts de $J[a[i]-1]$;
      else
        $S[a[i]-1] \leftarrow S[a[i]-1] + pvr$;
        break;
      end
    end
    $i \leftarrow 1$;
    ; `// Reset du tableau p (QuickPerm)`
    while $p[i] = 0$ **do**
      $p[i] \leftarrow i$;
      $i \leftarrow i+1$;
    end
  end
  for $i \leftarrow 0$ **to** $n-1$ **do** $S[i] \leftarrow S[i]/n!$;
  return S;
end

[1] : Ici on accède à J et S à la position $a[i] - 1$ car le tableau a contient des valeurs de 1 à n ($a[0] = 1$ jusqu'à $a[n-1] = n$) et que les tableaux J et S sont de 0 à $n - 1$.

[2] : Ici on sort de la boucle, car les *pvr* (points de vies restants) sont à 0, donc personne ne peut plus attaquer : on passe à la permutation suivante.

Complexité : Ici la complexité est de $O(n * n!)$, avec n le nombre de joueurs. Car nous utilisons QuickPerm qui a une complexité de $O(n!)$ et nous faisons le calcul de dégâts des joueurs($O(n)$) pour chaque permutation. La complexité de cet algorithme est donc factorielle.

4.4 Calcul avec des valeurs approximatives de Shapley

Algorithme : Tout comme pour l'algorithme 2, nous prenons un tableau a qui est un tableau d'indice. Chaque case de ce tableau aura une valeur entre 1 et n (avec n le nombre de joueurs). Grâce à l'algorithme Fisher-Yates shuffle[8], nous allons générer m permutations de l'ensemble $n!$ (avec m une constante). Ces m permutations nous permettrons de calculer une valeur approximative des valeurs de Shapley qui en est plus ou moins proche. Les valeurs de Shapley seront également stockées dans un tableau $S[n]$ et les gains redistribués avec la même formule que pour l'algorithme 2. Cet algorithme est linéaire car il ne calculera que m permutations même si n est grand.

Pourquoi Fisher-Yates shuffle ? De base nous voulions tirer aléatoirement entre 1 et n chaque case de a, mais vu que si un nombre est déjà apparu il ne peut pas être réutilisé, nous aurions eu besoin de tirer aléatoirement un nombre beaucoup plus de n fois (complexité $> O(n)$). Nous avons donc choisi l'algorithme Fisher-Yates shuffle pour sa complexité en $O(n)$, il tirera toujours n nombres aléatoires pour le même résultat final.

Comment fonctionne Fisher-Yates shuffle ? L'algorithme Fisher-Yates shuffle est un algorithme de permutation aléatoire. Cet algorithme a besoin d'une variable j qui représente un indice. Pour j allant de la fin du tableau à 1, on va tirer un nombre aléatoire entre 0 et j. On va ensuite permuter la case j avec le nombre tirer aléatoirement. j va ensuite se décrémenter. On a donc à chaque tour, un élément de $[0; j]$ qui va se retrouver à la position j. Au tour d'après, l'élément à la case j devient l'élément à la case $j + 1$ et ne peut donc plus être permuté. Pour résumer, l'algorithme tire aléatoirement un élément de l'avant du tableau pour le mettre à l'arrière. On a donc bien un algorithme de permutation aléatoire en $O(n)$.

[8]Ronald Fisher and Frank Yates. *Fisher–Yates shuffle.*

Algorithm 3: Répartition avec les valeurs approximatives de Shapley

Entrée : $J[n]$ un tableau de joueurs et Bpv les points de vie du boss.
Sortie : $S[n]$ le tableau des valeurs de Shapley.
Variables : $S[n]$, $a[n]$, $p[n+1]$, m, r, i, j, tmp et pvr des entiers.
(avec n = nombre de joueurs et m une constante)

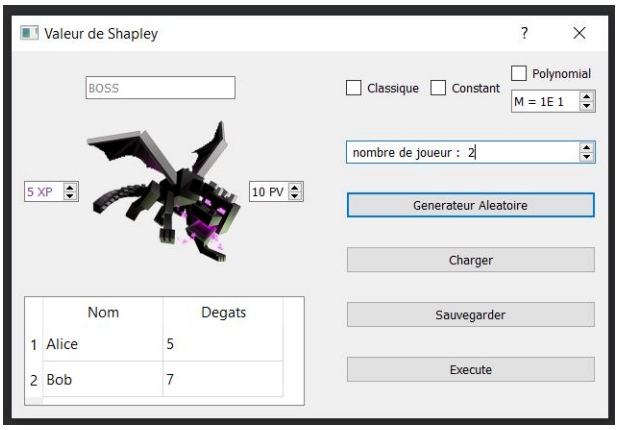
```
begin
    for i ← 0 to n − 1;                          // Initialisation du tableau a
     do
        a[i] ← i + 1;
    end
    for i ← 0 to m ;                  // Génération des permutation et calculs
     do
        pvr ← Bpv;
        for j ← n − 1 downto 1 ;      // Permutation aléatoire Fisher-Yates
         do
            r ← un nombre aléatoire entre 0 et j;
        end
        if j ≠ r then
            tmp ← a[j] ;                                        // Swap a[j], a[r]
            a[j] ← a[r];
            a[r] ← tmp;
        for j ← 0 to n − 1 ;              // Calcul pour la permutation généré
         do
            if dégats de J[a[j] − 1] ≥ pvr then
                S[a[j] − 1] ← S[a[j] − 1] + dégats de J[a[j] − 1];
                hpr ← pvr - dégats de J[a[j] − 1];
            else
                S[a[j] − 1] ← S[a[j] − 1] + pvr;
                break;
            end
        end
    end
    for i ← 0 to n − 1 do S[i] ← S[i]/n!;
    return S;
end
```

Complexité : Ici la complexité est $O(m * 2 * n)$, avec m la constante choisie car nous utilisons m fois Fisher-Yates ($O(n)$) et le calcul des dégâts des joueurs ($O(n)$). La complexité de cet algorithme est donc linéaire.

Amélioration pour la méthode d'approximation : Pour la méthode d'approximation des valeurs de Shapley, le tirage au sort de m permutations ne prend pas en compte les hits (permutation déjà utilisée plusieurs fois). Ceci peut donc rendre l'approximation plus éloignée de la valeur de Shapley. Nous avons donc essayé de prendre en compte ces hits et donc ne jamais utiliser une permutation plus d'une fois. Nous nous sommes rendu compte que l'approximation obtenue était plus proche des valeurs de Shapley, mais la complexité de l'algorithme explose.
En effet, avec $m = 10^8$ par exemple, il faut stocker 10^8 permutations et vérifier cette liste à chaque nouvelle génération.
De plus, si la permutation est déjà utilisée, il faut en régénérer une nouvelle et recommencer le processus. Nous avons fait un test avec $n = 9$ et $m = 10^5$, ici $n!$ est environ 3,6 fois plus grand que m mais nous avons quand même eu 17000 hits, donc 17000 générations en plus et 17000 parcours de la liste en plus.
Cette méthode est certes plus intéressante pour la précision des valeurs mais on s'éloigne du but recherché d'avoir un algorithme linéaire et efficace.

V Interface Graphique

Pour l'interface graphique, nous avons un ensemble de fenêtres liées entre elles par des slots.
Nous avons une fenêtre principale sur laquelle l'utilisateur sera amené a rentrer les informations des joueurs (Dégâts, nom) et du boss (point de vie, expérience, nom).
Il aura également la possibilité de générer aléatoirement les joueurs, de charger un fichier contenant le descriptif des joueurs, de sauvegarder les données des joueurs saisies et le choix de l'algorithme a utiliser.
Par la suite, une fois toutes les informations saisies, il lancera le programme (bouton execute) et obtiendra une nouvelle fenêtre contenant les résultats (temps de calcul, gains de chaque joueurs par algorithme utilisé) avec la possibilité de sauvegarder ces derniers au format PDF.



5.1 Fonctionnalité secondaire

- **Chargement :** Nous avons implémenté la possibilité de charger un fichier qui contient les données des joueurs et du boss. Cette fonctionnalité prend un charge un fichier .txt contenant le nombre de joueurs puis pour chaque joueur leur nom et nombre de dégâts et enfin le boss ses PV et son XP.
- **Sauvegarde :** L'application offrira la possibilité de sauvegarder les résultats sous forme de tableau dans un fichier PDF.
- **Génération aléatoire des joueurs :** Nous avons décidé de laisser le choix a l'utilisateur de saisir le nombre de joueurs, les dégâts maximum des joueurs, les PV et l'XP maximum du boss. Par la suite nous générons les noms, les dégâts des joueurs, les PV et l'XP du boss aléatoirement.

VI Expérimentation

Nous avons effectué un ensemble de tests sur nos algorithmes décrits ci-dessus. Ces tests ont été effectués sans l'interface graphique afin d'obtenir des résultats ne prenant en compte que nos implémentations algorithmiques.
Enfin, ces tests ont été effectués sur une machine avec un processeur Intel i5.

6.1 Test avec 10 joueurs

Test avec 10 joueurs pour voir les différences de gains.
Paramètre de notre test :

	Dégats	
Alice	7	
Bob	9	
Charlie	11	
Damien	2	
Eric	3	
Fabien	8	
Gabriel	8	
Herve	15	
Ilona	10	
Julia	11	
Boss	50 HP	100 XP

Résultats :

	Joueurs	Simple	Shapley Normal	Shapley Appro
Temps		0	0	2
1	Alice	8,3333	8,2904	8,2895
2	Bob	10,7142	10,688	10,6873
3	Charlie	13,0952	13,1214	13,1214
4	Damien	2,3809	2,3507	2,3497
5	Eric	3,5714	3,5333	3,5315
6	Fabien	9,5238	9,4801	9,4777
7	Gabriel	9,5238	9,4801	9,4815
8	Herve	17,8571	18,0325	18,0388
9	Ilona	11,9047	11,9015	11,9007
10	Julia	13,0952	13,1214	13,1214

On observe que la répartition des gains est très proches de l'ordre du centième entre nos différents algorithmes. De plus, le temps de calcul commence a augmenter en Shapley du faite du M = 1e7.

6.2 Test avec 10 joueurs et une configuration spéciale

Test avec 10 joueurs où l'on va observer la répartition des gains

Paramètre de notre test :

	Dégats	
Alice	5	
Bob	5	
Charlie	500	
Damien	5	
Eric	5	
Fabien	5	
Gabriel	5	
Herve	5	
Ilona	5	
Julia	5	
Boss	50 HP	100 XP

Résultats :

	Joueurs	Simple	Shapley Normal	Shapley Appro
Temps		0	0	3
1	Alice	0,9174	5,5555	5,5571
2	Bob	0,9174	5,5555	5,5567
3	Charlie	91,7431	50	49,9888
4	Damien	0,9174	5,5555	5,5574
5	Eric	0,9174	5,5555	5,5554
6	Fabien	0,9174	5,5555	5,5573
7	Gabriel	0,9174	5,5555	5,5577
8	Herve	0,9174	5,5555	5,5565
9	Ilona	0,9174	5,5555	5,5563
10	Julia	0,9174	5,5555	5,5562

On observe que la répartition des gains est très éloignée entre Algo-Simple et les algorithmes Shapley et Shapley-Approx. De plus, en Algo-Simple Charlie a plus de 91% de gains or si dans une configuration particulière où Charlie attaquerait en dernier le boss serait déjà mort et donc l'apport de Charlie aurait été nul.

6.3 Test avec 14 joueurs

Test avec 14 joueurs pour voir les limites des algorithmes.
Paramètre de notre test :

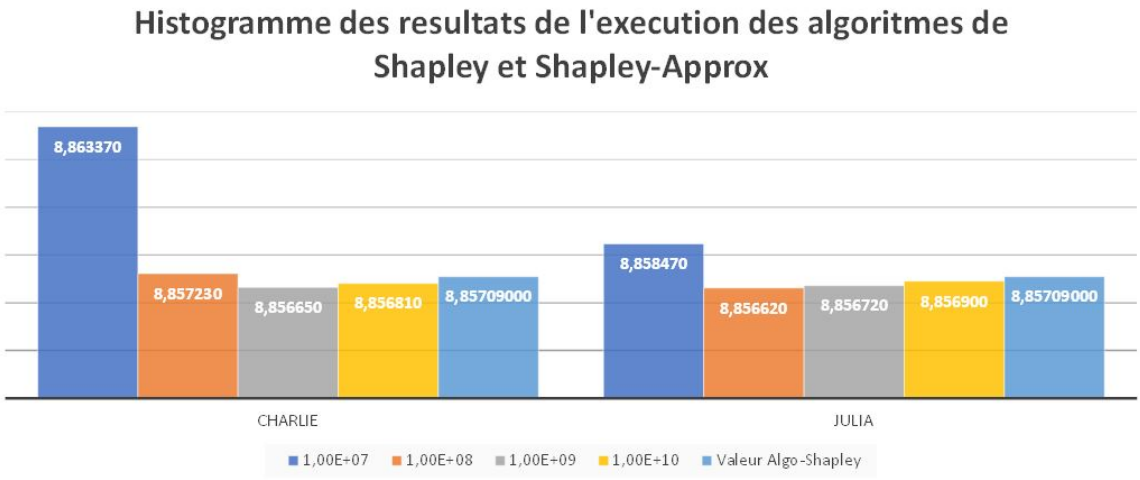
	Dégats	
Alice	7	
Bob	9	
Charlie	11	
Damien	2	
Eric	3	
Fabien	8	
Gabriel	8	
Herve	15	
Ilona	10	
Julia	11	
Kimberley	9	
Loic	10	
Moise	7	
Noe	14	
Boss	50 HP	100 XP

Résultats :

	Algo Shapley-Approx			
Temps (s)	3	29	293	627
Echantillonage (M)	1,00E+07	1,00E+08	1,00E+09	1,00E+10
Alice	5,683190	5,684280	5,685480	5,685260
Bob	7,277130	7,276960	7,278290	7,278250
Charlie	8,863370	8,857230	8,856650	8,856810
Damien	1,636360	1,636550	1,636550	1,636460
Eric	2,450520	2,450240	2,450330	2,450330
Fabien	6,480940	6,481700	6,483250	6,483060
Gabriel	6,480430	6,482230	6,483050	6,483280
Herve	11,965500	11,965500	11,963900	11,964000
Ilona	8,070470	8,069570	8,070040	8,070190
Julia	8,858470	8,856620	8,856720	8,856900
Kimberley	7,277810	7,279300	7,277960	7,277960
Loic	8,069150	8,070540	8,070210	8,070260
Moise	5,684240	5,685150	5,685090	5,685130
Noe	11,202400	11,204100	11,202500	11,202100

	Algo Shapley	
Temps (s)	296	3990,00
Joueurs	13,00	14,00
Alice	6,379520	5,685170
Bob	8,18876000	7,27823000
Charlie	9,99020000	8,85709000
Damien	1,83233000	1,63650000
Eric	2,74792000	2,45026000
Fabien	7,28500000	6,48327000
Gabriel	7,28500000	6,48327000
Herve	13,55300000	11,9639
Ilona	9,08991000	8,07003000
Julia	9,99020000	8,85709000
Kimberley	8,188760	7,278230
Loic	9,089910	8,070030
Moise	6,379520	5,685170
Noe		11,2017

On observe que la répartition des gains est très proches suivant les instances d'algorithmes utilisées. Le temps de calcul quand a lui augmente de façons exponentielle avec l'algorithme Shapley passant de 256s pour 13 joueurs a 3990s (1h06) pour 14 joueurs. On observe le même phénomène en utilisant l'algorithme Shapley-Approx.
En se concentrant sur les valeurs :



On observe que plus nous augmentons la taille de l'échantillonnage (M) plus les répartitions des gains se rapprochent des répartitions obtenues avec l'algorithme Shapley.

VII Conclusion

A la suite de notre implémentation, nos tests nous ont permis d'observer que la répartition des gains utilisant les valeurs de Shapley peut être calculée plus ou moins rapidement suivant le nombre de joueurs. L'utilisation de l'algorithme Shapley-Approx permet d'accélérer le calcul de la répartition des gains pour des grandes instances de joueurs tout en gardant une précision de l'ordre du millième.

Notre application offre des possibilités d'amélioration futur. Nous souhaitons implémenter une version multi-thread de nos algorithmes afin d'améliorer leurs performances notamment en terme de temps de calcul. Nous avons aussi pensé a un mode où le boss aurait des points de vie illimités afin de maximiser les dégâts de chaque joueurs.

Les valeurs de Shapley sont devenus très utiles pour la mesure du pouvoir de vote dans les processus de décision collective (par exemple au conseil de l'ONU). Par ailleurs ces valeurs restent en grande partie utilisées dans le partage des coûts (ratio coûts/utilisation).

References

[1] L. S. Shapley. “A value for n-person games”. In: (1953).

[2] Michel PAUL et Hatem SMAOUI CEMOI Dominique LEPELLEY. *Introduction à la théorie des jeux (2) : les jeux coopératifs1*.

[3] frederic koessler. *7C-cooperatifs-Shapley-article*.

[4] Phillip Paul Fuchs. *QuickPerm*.

[5] GeeksForGeeks. *Write a program to print all permutations of a given string*.

[6] Ronald Fisher and Frank Yates. *Fisher–Yates shuffle*.