UNIVERSITÉ DE ROUEN

Stage 2015-2016

Stage au sein du département informatique du Madrillet

Rapport de stage 1.1

26 août 2016

Auteur(s): Tristan Rodriguez
Relecteur(s): Tristan Rodriguez
Approuvé par: Carla Selmi et Jean-philippe Dubernard

Version	Date	Changelog
0.1	du 9 au 14 mai	recherche bibliographique (Théorème de Spague-Grundy et différents jeux de Nim) et étude de l'article de J. Ulehla
0.1.1	du 16 au 21 mai	recherche technique sur le python et implantation d'un jeu de Nim ordinaire
0.1.2	du 23 au 27 mai	étude du travail de Samuel Giraudo et Cédric Leroy sur les listes préfixes et suffixe dans les arbres
0.2	du 30 mai au 2 juin	fin de l'implantation de la méthode Ulehla et une forêt gérée grâce aux listes préfixe et suffixe
0.3.1	du 6 au 10 juin	début de la rédaction du rapport et mise en forme de la preuve sur les listes préfixe et suffixe
0.3.2	du 12 au 16 juin	étude du jeu de Chomp revisité
1.0	du 20 au 30 juin	rédaction du rapport de stage
1.1	du 4 au 9 juillet	finalisation du rapport et rendu du travail

Remerciements

Avant toutes choses, je tiens à remercier Carla Selmi, Jean-Phlippe Dubernard qui m'ont donné la possibilité de découvrir le métier d'enseignant-chercheur et m'ont fait confiance durant ce stage. Je tiens également à remercier Yacine Hmito qui a travaillé avec moi, qui m'a aidé et qui a partagé son travail et ses connaissances.

Ce stage a été proposé et encadré par Carla Selmi et Jean-Phlippe Dubernard.

Ce rapport a été rédigé en LaTeX.

Table des matières

1 Introduction

Durant la formation de licence informatique, nous avons obtenu de fortes bases théoriques et pratiques, et nous avons la possibilité de pouvoir mettre à profit nos connaissances durant un stage de deux mois. Le but de ce dernier est d'avoir une première approche du milieu professionnel dans le monde de l'informatique et de pouvoir évaluer nos connaissances acquises.

C'est dans ce but que j'ai obtenu un stage au sein du département informatique de l'université de Rouen.

Ce stage fut pour moi une bonne occasion de pouvoir découvrir le milieu de la recherche informatique et de pouvoir valider mes choix d'orientation. En effet, ce stage m'a permis de pouvoir parler et travailler avec une équipe d'enseignants-chercheurs.

A l'issue de cette formation, j'ai pour but de continuer mes études en sécurité informatique afin de pouvoir devenir intervenant extérieur.

Durant ce stage, j'ai dû, avec l'aide de mes encadrants, étudier les jeux de Nim dans leur généralité et plus particulièrement le jeu de la forêt et sa méthode de résolution proposée par Josef Ulehla en 1979.

Dans ce rapport, je vais vous présenter le travail effectué durant ce stage et aussi vous présenter les implantations des différents jeux étudiés.

1.1 Présentation du département informatique

Le département informatique de l'université de Rouen est découpé en deux équipes de recherche, la première est au technopole du Madrillet. Cette équipe du département est spécialisée dans la recherche algébrique de l'informatique comme la cryptographie, la théorie des graphes et la théorie des automates ; alors que le second département est situé à Mont-Saint-Aignan et est spécialisé dans la recherche en bio-informatique et la théorie du texte.

Le département informatique de l'université est dirigé par M. Ayoub Otmani compte une trentaine d'enseignants-chercheurs permanents et fait appel à une vingtaine d'intervenants professionnel extérieurs. Le département forme aussi des doctorants : à ce jour deux doctorants sont présents, M. Vlad Dragoi et M. Clément Miklarz. M. Ali Chouria a quant à lui validé sa thèse durant la période de ce stage.

Ce stage fait suite à un projet effectué au cours de l'année dans le cadre du cours de théorie des graphes de Madame Selmi au cinquième semestre de licence informatique.

2 Généralités

Durant le stage, nous avons commencé par étudier théoriquement le sujet. Comme nous avons étudié l'article de J Ulehla, nous avons commencé par faire une étude bibliographique.

Dans cette partie, nous allons présenter les outils théoriques dans l'ordre d'étude. Ces outils ont été nécessaires pour pouvoir comprendre et mieux implanter le jeu de la forêt.

2.1 Jeu de Nim

Un jeu de Nim est un jeu de stratégie pure, c'est-à-dire que ce jeu ne laisse aucune part au hasard et ne permet pas la fin sur une égalité entre les deux joueurs.

Par exemple, un jeu avec un tas d'allumettes où l'on enlève une ou plusieurs allumettes dans ce tas et où le gagnant est celui qui a retiré la dernière allumette est un jeu de Nim.

2.1.1 Jeu de Nim simple

Un jeu de Nim simple est un jeu de Nim comme défini ci-dessus à ceci près qu'il peut être résolu de façon rapide. Ces jeux ont été résolus par Charles Bouton en 1901[JP10] qui a trouvé un algorithme permettant le gain.

Le graphe suivant permet de représenter tous les cas possibles pour un jeu de Nim décrit ci-dessus avec un tas à 3 allumettes.

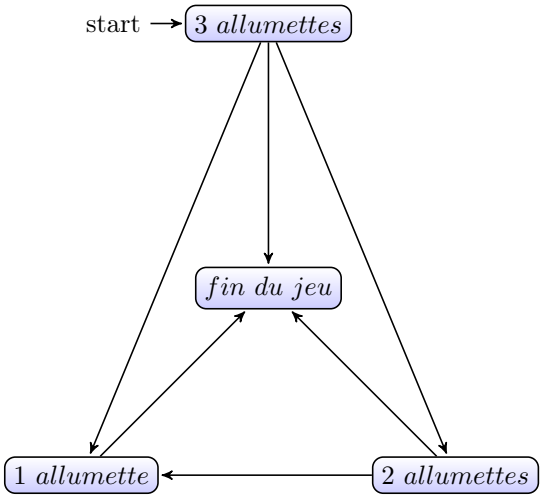

FIGURE 1 – Graphe du jeu de Nim simple avec un tas à 3 allumettes

Le principe de l'algorithme de Bouton est de donner une valeur à chacun des états du graphe représentant le jeu appelé *Nimber*. Dans le cas d'un jeu à un tas unique, le *nimber* d'un état est la valeur de l'état elle-même.

L'attribution de ce *nimber* est récursive, tous les états finaux (un seul état final dans le cas d'un jeu à un tas) ont comme *nimber* 0. Ensuite, pour tout état, le *nimber* est égal au plus petit entier qui n'est pas égal au nimber de chacun de ces fils. Nous pouvons bien facilement voir alors que ce nimber est une interprétation du noyau d'un graphe. En effet, tous les éléments du noyau du graphe d'un jeu seront les éléments ayant un nimber égal a 0.

Reprenons l'exemple précédent : l'état où le jeu est fini aura comme valeur de nimber 0, ensuite pour l'état avec une allumette, nous aurons comme valeur de nimber 1. Comme nous connaissons le nimber de tous les fils de l'état avec deux allumettes, nous pouvons donner comme valeur de nimber le plus petit entier qui n'est égal à aucune valeur de nimber de ses fils, c'est pourquoi nous lui attribuons la valeur 2. Enfin nous appliquons le même procédé pour l'état avec 3 allumettes et nous lui attribuons la valeur 3.

Ce *nimber* permet de pouvoir savoir si oui ou non on peut gagner à partir d'une position donnée de jeu.

Si cette valeur est égale a 0, alors on dit que la position est une *position gagnante*, sinon on dit que cette position est une *position perdante*. Si jamais un joueur commence à partir d'une position perdante, alors il possède une stratégie gagnante, en effet lorsque le joueur est dans une position perdante, alors il y a forcément un état successeur de la position dans laquelle se trouve le joueur qui possède une valeur de nimber valant 0 qui permettra au joueur de gagner.

Dans le cas contraire, la position dans laquelle se trouve le joueur a un nimber valant 0 alors il n'a pas d'état successeur permettant de gagner. En effet comme la valeur de nimber est le plus petit entier n'étant pas égal aux nimber de ses successeurs, alors il n'y a pas de valeur de nimber de ses successeurs égal a 0. Cela ne permet pas de gagner car jamais le joueur ne pourra atteindre l'état final.

2.1.2 Jeu de Nim complexe

Un jeu de Nim complexe est un ensemble de jeux de Nim simples. Ils ont été résolus indépendamment par Roland Sprague en 1935 et Patrick Grundy en 1939[Gru64].

Par exemple, un jeu de Nim avec plusieurs tas d'allumettes où l'on enlève une ou plusieurs allumettes dans un tas à la fois et où le gagnant est celui qui retire la dernière allumette est un jeu de Nim complexe.

Le graphe suivant est le graphe d'un jeu avec deux tas de 2 allumettes.

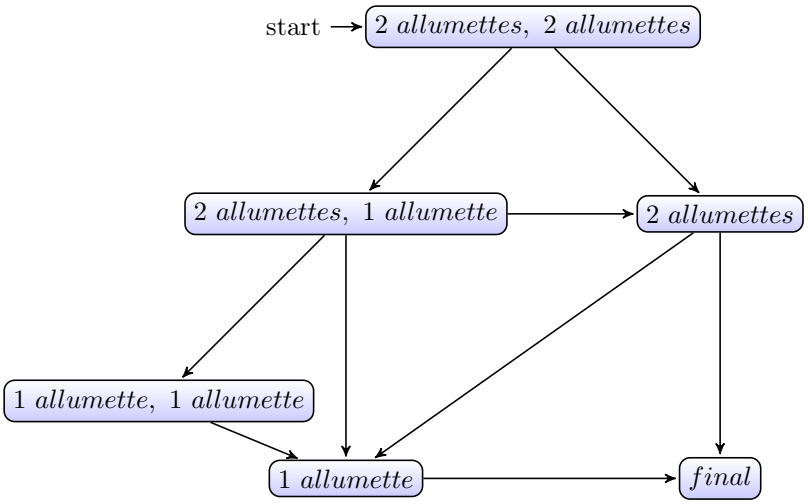

FIGURE 2 – Graphe du jeu de Nim complexe avec 2 tas de 2 allumettes

La résolution de ces jeux est une généralisation de la méthode de résolution des jeux de Nim simples. Le principe est de *diviser pour mieux régner*. En effet, nous divisons le jeu en autant de jeux de Nim simples qu'il y a de tas.

Comme les jeux de Nim simples, on commence par donner les valeurs de nimber à tous les états du graphe du jeu, puis pour savoir si une position du jeu est gagnante, nous devons faire une opération spéciale appelée *addition de Nim*.

Cette opération est une addition bit à bit des nimbers en binaire sans retenue. Ceci revient à faire un *ou exclusif* entre les valeurs des nimbers de tous les tas de l'état du jeu.

Comme le jeu de Nim simple, lorsque le nimber d'une position est égal à 0, on se trouve dans une position gagnante du jeu et dans tous les autres cas, nous nous trouvons dans une position perdante.

Dans l'exemple d'un jeu avec un tas de 3 allumettes et un tas de 2 allumettes, le joueur qui commencera à jouer à partir de la position (3,2), pourra gagner puisqu'il pourra arriver suite à un mouvement à une position gagnante. Pour savoir quel est ce coup gagnant, il suffit de trouver une valeur pour un des deux tas qui permet d'obtenir un nimber a 0. Ce coup est facile à trouver puisqu'il suffit de retirer une allumette dans le tas avec 3 allumettes (2 ou exclusif 2 est bien égal a 0). Le joueur qui commence a donc bien une stratégie gagnante.

Cette méthode de résolution est très utilisé puisque nous pouvons trouver une fonction permettant d'attribuer une valeur de nimber en fonction du jeu que nous étudions.

C'est ce qu'a fait J. Ulehla en 1979 pour trouver une méthode de résolution rapide du jeu de Hackendot de J. Von Neumann.

2.2 Hackendot

Ce jeu, créé par John Von Neumann et John Horton Conway indépendamment, est un jeu de Nim sur les arbres où l'on retire dans un arbre du jeu le chemin de la racine à un noeud choisi. Le dernier joueur à enlever un nœud du jeu gagne.

Par exemple, l'ensemble d'arbres suivant est une situation initiale d'un jeu de Hackendot

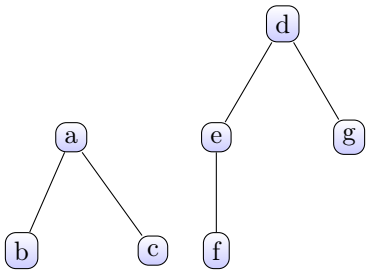

FIGURE 3 – Forêt d'arbres d'un jeu de hackendot

La résolution proposée en 1979 par Josef Ulehla est un dérivé de la méthode de résolution des jeux de Nim complexes. En effet, dans son papier [Ule79] Ulehla cherche simplement une fonction qui permettrait de donner une valeur de nimber à une situation de jeu.

Dans la suite, nous verrons un arbre comme étant un graphe. Le noyau d'un arbre est construit de la même façon que le noyau d'un graphe, tout élément n'ayant pas de fils appartiendra au noyau et tout élément ayant aucun fils n'appartenant pas au noyau appartiendra au noyau.

Dans son papier, Ulehla défini plusieurs fonctions :

1. `l(f)` qui permet de *dénoyauter*, c'est-à-dire enlever le noyau du graphe du jeu. $l^n(f)$ est la fonction `l` appliquée n fois à la forêt f.
2. `rip(f)` (pour Root-ImParity) qui est égal à 0 (resp. 1) si le nombre de racines blanches de f est pair (resp. impair).

A l'aide de ces deux fonctions, on peut maintenant définir la méthode de résolution du jeu. En effet, pour savoir si on a bien une stratégie gagnante, on commence par *colorer* la foret du jeu. Colorer signifie que tous les nœuds appartenant au noyau sont *blanc* et tous les autres sont *noir*. Ensuite, on *dénoyaute* la forêt, ce qui permet d'obtenir une nouvelle forêt *f'*. Nous répétons ce procédé jusqu'à obtenir une forêt f^n vide. La suite des forêts f^n permet de savoir s'il y a une stratégie gagnante pour la forêt du jeu. En effet, lorsqu'il y a au moins un $rip(f^i)$ avec i compris entre *1* et n vaut 1, alors il y a une stratégie gagnante.

Dans cette méthode, la fonction pour donner une valeur de nimber a une position de jeu est la fonction *rip* et l'addition de Nim est la suite des valeurs de *rip* de toutes les forêts f^n.

Enfin, dans sa méthode, Ulehla permet de savoir quel est le coup gagnant. Comme dans la résolution des jeux de Nim complexes, pour trouver quel est le coup gagnant, il suffit de trouver le coup permettant d'avoir une forêt *f"* qui, lorsqu'on applique le procédé décrit précédemment, a un nimber égal à 0 (c'est-à-dire une suite de *rip* égaux à 0).

Pour trouver ce coup gagnant lorsqu'il existe, il faut prendre la dernière forêt de la suite notée f^k tel $rip(f^k) = 1$. A partir de cette forêt, on prend l'ensemble des racines blanches et leurs successeurs blancs directs et on regarde quel suppression permet d'obtenir $rip(f^k) = 0$. On remonte la suite des forêts en prenant à chaque fois l'ensemble constitué du coup gagnant de la forêt précédente et ses successeurs.

Reprenons l'exemple du début, la première question que nous nous posons est : est-ce qu'on a une stratégie gagnante en jouant ?

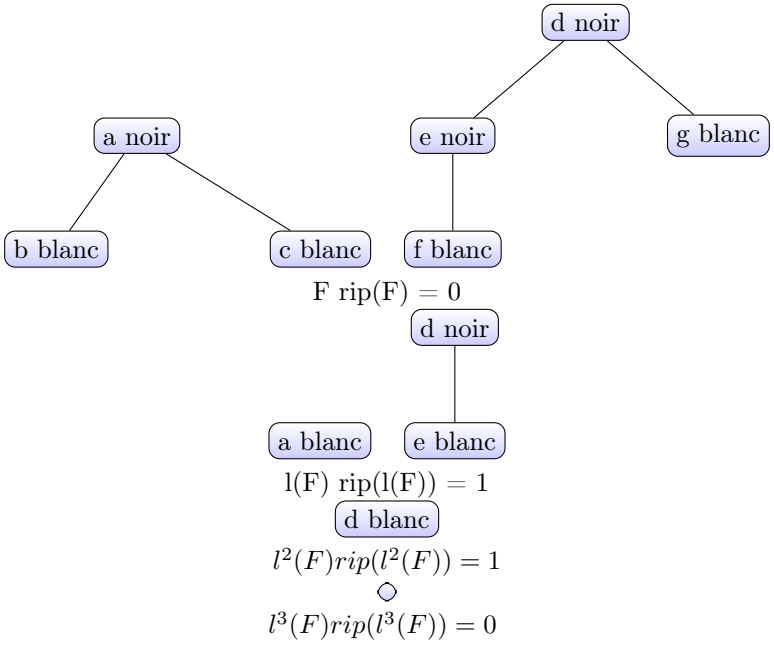

FIGURE 4 – Suite des forêts dénoyautées

Nous savons qu'il existe un coup gagnant en faisant la suite des forêts dénoyautées et que nous devons chercher dans $l^2(F)$. Nous savons que dans cette forêt, il suffit de retirer le nœud `d` *qui permet d'obtenir rip égal à 0.*

Nous cherchons maintenant dans la forêt l(F) quel est le coup permettant d'obtenir un rip égal à 0. Nous savons que dans la forêt précédente, le coup gagnant était `d` *donc ici, nous regardons l'ensemble constitué de* `d, e`*. Nous voyons que c'est le nœud* `d` *qu'il faut supprimer pour avoir un rip égal à 0.*

Enfin, nous appliquons le même procédé à la forêt F et nous trouvons que le coup gagnant est `g`*.*

2.3 Parcours sur les arbres

Durant l'implantation du jeu de Hackendot, nous nous sommes retrouvé face à une difficulté : comment implanter des arbres quelconques en python ?

Pour répondre à ce problème, Madame Selmi m'a fourni le travail d'un ancien étudiant[SG06] qui a utilisé uniquement le parcours préfixe et suffixe sur les arbres et a permis de gérer l'intégralité de ce problème.

Quelque soit le parcours utilisé, nous regardons de gauche à droite les nœuds dans l'arbre, c'est-à-dire que lorsque nous sommes sur un nœud donné, nous cherchons d'abord à découvrir récursivement tout son fils gauche puis nous allons découvrir son fils droit.

Tout d'abord, nous avons utilisé le parcours préfixe sur une forêt. Ce parcours est effectué dans l'ordre de *découverte* des nœuds dans la forêt. Nous pouvons obtenir la liste préfixe, notée P, d'un arbre en mettant les nœuds dans l'ordre de découverte d'après le parcours préfixe.

Par exemple, dans l'arborescence suivante, la liste préfixe est (1, 2, 3, 4)

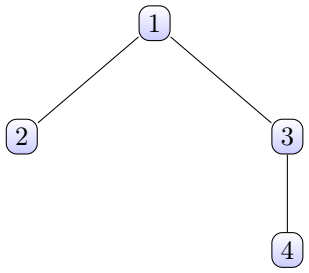

FIGURE 5 – Arborescence de liste préfixe (1, 2, 3, 4)

Ensuite, nous avons utilisé le parcours suffixe sur une forêt. Ce parcours est effectué dans l'ordre de fin de découverte des nœuds dans la forêt. Un nœud est entièrement découvert lorsque nous avons entièrement exploré tous ses successeurs. Grâce à ce parcours, nous pouvons créer la liste suffixe, notée S, en mettant dans l'ordre de découverte par rapport à ce parcours les nœuds de la forêt.

Dans l'exemple précédent, la liste suffixe de l'arborescence est (2, 4, 3, 1)

Nous avons aussi défini les listes *S(n)* et $P^{-1}(n)$ comme étant respectivement la liste suffixe à partir d'un nœud n et la liste des nœuds du début de la liste préfixe jusqu'au nœud n.

Dans l'exemple précédent, prenons le nœud 2, alors la liste S(2) est la liste (4, 3, 1) et la liste $P^{-1}(2)$ *est la liste (1).*

Grâce à ces différentes listes, nous avons pu connaitre le père direct d'un nœud n de la forêt. En effet, nous remarquons que le père direct du nœud est le premier élément de *S(n)* commun à la liste $P^{-1}(n)$.

Nous avons aussi remarqué que tous les nœuds du chemin de la racine à un nœud n étaient tous les nœuds commun à la liste S(n).

Durant ce stage, la preuve de ces deux éléments a été fournie.

Ces listes préfixes et suffixes ont été utilisées sur des mots de Dyck générés de façon aléatoire avec une grammaire dont les règles sont :

— S -> () qui permet de créer une racine sans fils
— S -> ()S qui permet de créer une racine avec une seconde arborescence
— S -> (S) qui permet de créer une racine avec un sous arbre
— S -> (S)S qui permet de créer une racine avec un sous arbre et une seconde arborescence

Par exemple, dans l'arborescence précédente, le mot de Dyck est généré est : `((()()))`

2.4 Jeu de Chomp

Le dernier jeu que nous avons étudié est le jeu de Chomp revisité. Comme le Chomp, nous jouons sur une tablette carrée ou rectangle, mais nous avons cependant modifié les règles pour faciliter le jeu. Nous devons choisir une case et supprimer soit la ligne, soit la colonne de la case choisie. Le dernier joueur à enlever une case est le vainqueur.

Voici une situation de jeu avec une tablette rectangle 3x4

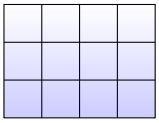

FIGURE 6 – Situation de jeu du Chomp

Au démarrage, ce jeu devait être sur les polyominos convexe (sans trou) cependant, nous nous sommes rendus compte que travailler sur une tablette ou sur un polyomino revenait au même. En effet, nous pouvons voir facilement qu'un polyomino convexe est compris dans une tablette, c'est pourquoi nous avons généralisé directement sur une tablette.

Ensuite, nous avons cherché à diviser ce jeu de façon à pouvoir utiliser le théorème de Sprague-Grundy, nous avons donc pensé que le but était de calculer le nombre de lignes et de colonnes. Nous avons pensé que si le nombre de lignes ou le nombre de colonnes était impair, le coup était gagnant.

Puis nous avons implanté le jeu avec cette idée, cependant nous sommes tombé face à un contre exemple. C'est pourquoi nous avons essayé de faire en sorte de revoir la façon de calculer si le coup est gagnant. Nous avons donc pensé que lorsque nous avons plusieurs composantes connexes, nous pouvons les mettre sous la forme d'une tablette avec comme nombre de ligne la somme du nombre de ligne de toutes les composantes connexes, et de la même façon le nombre de colonnes la somme des colonnes de toutes les composantes connexes.

A la fin de ce stage, nous n'avons pas réussi à savoir comment faire pour savoir s'il existe une stratégie gagnante à partir d'une position de jeu donnée, et si oui, comment calculer le coup gagnant.

Cependant, nous avons conjecturé que, pour une position donnée, le nimber valait :
— `m` dans le cas d'une tablette 1xm ou mx1 (prouvé)
— `0` si la tablette est de la forme 2nx2m (prouvé)
— `1` si dans une tablette nxm avec m ou n impair m + n est impair
— `2` sinon
Nous avons mis au point un petit programme permettant la vérification de cette conjecture et à ce jour, nous n'avons pas trouvé de contre exemple.

3 Travail effectué

Suite à l'étude théorique des jeux de Nim et de leurs méthodes de résolutions, il a été temps de les implanter.

Dans ce but, j'ai décidé d'apprendre une nouveau langage, le *python*, qui est un langage fortement typé, interprété et est aussi orienté objet.

Ce langage est de plus en plus présent dans le milieu professionnel et est très utilisé dans le milieu de la sécurité informatique. C'est pour ces raisons que j'ai pensé que ce stage était l'occasion de pouvoir apprendre ce nouveau langage.

Toutes les implantations effectuées durant ce stage sont disponibles sur GitHub [1] en téléchargement libre.

3.1 Jeu de Nim simple

La première implantation effectuée était principalement pour apprendre le python.

En effet, le jeu de Nim simple était facile à programmer car il ne demande que très peu de mémoire et la méthode de résolution est basée sur l'opération XOR [2] sur l'ensemble des entiers.

Pour implanter ce jeu, nous avons eu besoin de deux fichiers : le premier `ia.py` permet de pouvoir calculer s'il existe un coup gagnant et de savoir quel est ce coup, et le second `nim.py` est une implantation du jeu de Nim complexe à plusieurs tas.

Pour pouvoir implanter la méthode de résolution, nous avons eu besoin de plusieurs fonctions :
— `getwinningStrat(l)` qui retourne le coup à jouer pour gagner s'il existe
— `numAdd(l)` qui retourne le nimber d'une situation de jeu
L'algorithme pour avoir le coup gagnant d'une situation de jeu est la suivante :

1. https://github.com/wampixel/nim-games
2. ou exclusif

Algorithm 1 Calcul le coup gagnant s'il existe

Require: l une liste de tas représentant une situation de jeu
Ensure: un tuple de la forme (t, n) avec t le tas où jouer, n le nombre à enlever

```
s ← numAdd(l)
if s = 0 then
    return (0, 1)
else
    i ← 0
    while i < len(l) do
        f ← f xor l[i]
        if f = 0 then
            return (i, l[i])
        else
            k ← 0
            while k < l[i] do
                if f xor k = 0 then
                    return (i, (l[i] − k))
                else
                    k ← k + 1
                end if
            end while
        end if
        i ← i + 1
    end while
end if
```

Le but de cet algorithme est de simplement tester si on est dans une position gagnante au démarrage (numAdd(l) = 0). Si c'est le cas, comme présenté dans les généralités, on ne peut pas gagner, c'est pourquoi on joue un coup dont on est sûr de l'existence (tas 0, 1 allumette).

Si jamais on n'est pas dans une position gagnante, alors on est sûr et certain qu'il existe un coup gagnant, c'est pourquoi on commence à calculer ce coup.

On commence à tester en enlevant chaque tas complètement et on regarde l'addition de Nim pour ce fils. Si jamais on a un résultat de 0 pour l'addition de Nim, alors on a trouvé le coup gagnant et on le retourne. Sinon, on va commencer à enlever les allumettes une à une dans chaque tas et on trouvera forcément le coup gagnant que l'on retournera.

Cet algorithme est en temps linéaire sur la somme des éléments de la liste. Dans le pire des cas, on aura du calculer tous les coups possibles pour trouver le coup gagnant.

Cet algorithme repose sur une addition spéciale appelée addition de Nim. Elle a été implantée dans la fonction `numAdd` et fonctionne de la façon suivante :

Algorithm 2 Addition de Nim

Require: l une liste de tas représentant une situation de jeu
Ensure: le nimber de la situation l

```
i ← 0
s ← 0
for all i valeurs de la liste l do
    s ← s xor i
end for
return s
```

Cette fonction calcule simplement une addition sans retenue entre toutes les valeurs des tas en base 2. Ceci revient à faire un XOR (ou exclusif) entre ces valeurs. Cette fonction est en temps linéaire par rapport à la longueur de la liste l.

Ces algorithmes permettent d'implanter le théorème de Sprague-Grundy. Suite à ceci, nous avons cherché à utiliser ce théorème dans plusieurs autres jeux pour vérifier si nous pouvions (à une méthode de calcul de nimber près) toujours résoudre les jeux.

3.2 Hackendot

Après avoir réussi à implanter le jeu de Nim et le théorème de Sprague-Grundy, nous avons attaqué le travail principal de ce stage. Nous avons donc étudié la méthode de résolution du jeu de Hackendot proposé par J. Ulehla.

Nous avons décidé d'utiliser ce coup ci la partie orientée objet du python. Ceci m'a permis de pouvoir apprendre à utiliser les objets en python.

La première classe que nous avons créée est la classe `forest`, celle-ci permet de gérer les forêts. Nous ne détaillerons pas ici le contenu de classe car c'est une gestion de listes préfixe et suffixe simples avec les éléments démontrés dans les généralités

Nous avons ensuite créé une classe `iaUlehla` permettant de créer une intelligence artificielle simplifiée pour calculer le coup gagnant dans ce jeux.

Dans cette classe, il y a les méthodes :
— `getWinningStrat(forest)` qui permet de connaitre le coup gagnant pour la forêt *forest*.
— `lfunction(forest)` qui permet d'effectuer la fonction l sur la forêt *forest*.
— `ripFunction(forest)` qui permet d'effectuer la fonction rip sur la forêt *forest*.
— `getNodeForWin(forestL)` qui permet d'implanter le raisonnement de Ulehla pour trouver le coup gagnant à partir de la liste des dénoyautages successifs de la forêt initiale *forestL*.

Cette classe utilise aussi deux méthodes privées qui permettent de *colorer* la forêt.

La première `colorNodeSucc(succList)` qui permet de colorer un nœud grâce à la liste de ses successeurs. Cette fonction prend en entrée une liste de couleurs qui représente la liste des couleurs de ses successeurs et elle retourne `BLANC` si et seulement si toutes les couleurs de la liste sont en `NOIR`. Si jamais la couleur d'au moins un des successeurs est en `BLANC`, la méthode retourne `NOIR`.

La seconde méthode privée `colorNode(forest, node)` permet de colorer le noeud `node` dans la forêt `forest`.Cette méthode construit récursivement la liste des couleurs des successeurs du noeud `node` et utilise la méthode `colorNodeSucc(succList)`.

L'avantage de cette méthode est de ne pas colorer entièrement la forêt, en effet, pour savoir si un noeud est blanc, nous devons être sûr que tous les fils du noeud que l'on veut colorer sont bien `NOIR`, cependant, pour qu'un noeud soit `NOIR`, il suffit de savoir qu'un fils est bien `BLANC`.

Dans la suite, nous allons présenter les algorithmes permettant de savoir si un coup gagnant existe, puis nous présenterons ensuite les algorithmes des fonctions propre à la méthode Ulehla (`ripFunction` et `lFunction`)

Algorithm 3 Calcul si le coup gagnant existe

Require: f une forêt quelconque
Ensure: le noeud à supprimer pour entrer dans une stratégie gagnante si c'est possible, le premier noeud de la liste sinon

```
forestL ← [f]
while f is not empty do
    f ← lfunction(f)
    forestL ← forestL.append(f)
end while
i ← −1
k ← len(forestL) − 1
while k > 0 and i = −1 do
    if ripFunction(forestL[k]) = 1 then
        i ← k
    end if
    k ← k − 1
end while
if i = −1 then
    return f.getPrefixeList()[0]
else
    return getNodeForWin(forestL[0...i + 1])
end if
```

Cet algorithme a besoin de la fonction `getNodeForWin(forestL)` qui implante l'algorithme suivant :

Algorithm 4 Calcul le coup gagnant dans la suite des forêts dénoyautées

Require: forestL la suite des forêts dénoyautées
Ensure: le noeud à supprimer pour entrer dans une stratégie gagnante

```
f ← forestL[len(forestL) − 1]
tmp ← f.getRoots()
root ← []
for all r in tmp do
  if colorNode(f, r) = BLANC then
    root ← root.append(r)
  end if
end for
i ← len(forestL) − 1
possibility ← root
while i >= 0 do
  f ← forestL[i].getForest()
  tmp ← possibility
  for all p in possibility do
    tmp ← tmp.extend(f.getSuccNode(p))
  end for
  j ← 0
  b ← TRUE
  while j < len(tmp) and b do
    f ← f.delNodeToRoot(tmp[j])
    if ripFunction(f) = 0 then
      b ← FALSE
      tmp ← [tmp[j]]
    end if
    j ← j + 1
  end while
  possibility ← tmp
end while
```

Cet algorithme est linaire sur le nombre de fils des coups possibles. En effet, nous pouvons voir qu'au premier tour de boucle, nous devons trouver tous les fils des coups dit possibles (c'est-à-dire les racines blanches de la dernière forêt), puis nous devons parcourir cette liste de successeurs pour trouver le coup qui permet de donner une suite de forets avec un rip a 0.

Ceci entraine que l'algorithme pour savoir si un coup gagnant existe est quadratique au pire des cas car on doit parcourir la liste complètement une fois pour trouver quelle est la dernière forêt avec un rip égal à 1, puis nous devons parcourir tous les fils possibles pour chaque coup gagnant possibles pour trouver le coup gagnant. Si jamais ce coup gagnant n'existe pas, cet algorithme est linéaire sur la longueur de la liste des forêts dénoyautées.

Les fonctions précédentes permettent, à partir du nimber d'un position, de savoir où est le coup gagnant. Ce nimber est attribué à une situation grâce à la fonction `rip` et la fonction `l` présentée dans l'article de Ulehla.

L'algorithme de ces fonctions sont les suivantes :

Algorithm 5 fonction rip

Require: f une forêt
Ensure: 1 si un nombre de racines blanches impair, 0 sinon
  $nbRW \leftarrow 0$
  for all i in f.getPrefixList() **do**
    if *i est une racine et* $colorNode(f, i) = BLANC$ **then**
      $nbRW \leftarrow nbRW + 1$
    end if
  end for
  return $(nbRW modulo 2)$

Cet algorithme est linéaire sur le nombre de nœud de la forêt. En effet, pour trouver le nombre de racines blanches, il faut parcourir la forêt complète et trouver tous les nœuds sans prédécesseurs(les racines) et trouver la couleur du nœud.

Algorithm 6 fonction l

Require: f une forêt
Ensure: f dénoyauté
  $whiteNode \leftarrow []$
  for all i in f.getPrefixList() **do**
    if $colorNode(f, i) = BLANC$ **then**
      $whiteNode \leftarrow whiteNode.append(i)$
    end if
  end for
  for all i in whiteNode **do**
    $f \leftarrow f.delNode(i)$
  end for

Cet algorithme est linéaire sur le nombre de nœud de la forêt. En effet, nous devons parcourir toute la forêt pour trouver les nœuds blancs et les supprimer, c'est immédiat(nous gérons les forêts avec des listes).

Une fois ce travail terminé, et mes encadrants étant content de mon travail, Madame Selmi m'a proposé une recherche nouvelle, le jeu de Chomp revisité. Cette recherche est un travail non effectué, nous avons donc dû chercher comment le résoudre et nous avons cherché à appliquer le théorème de Sprague-Grundy.

3.3 Jeu de Chomp

Avant tout, ce jeu n'existait pas à notre connaissance, c'est pourquoi nous avons dû chercher comment faire pour pouvoir appliquer le théorème de Sprague-Grundy. Nous avons conjecturé plusieurs résultats et prouvé les plus simples. Cependant, nous n'avons, à ce jour, toujours pas trouvé comment faire pour trouver le coup gagnant et gagner à tous les coup.

Nous avons quand même implanté le jeu et la solution que nous avions trouvée.

Cette implantation est faite en deux fichiers :
— `chomp.py` qui est le jeu en lui même
— `resolver.py` qui est l'implantation de la méthode de résolution conjecturé.

Dans ce jeu, nous jouons, comme le Chomp standard, sur une tablette. Nous choisissons un carré et nous supprimons soit la ligne, soit la colonne.

La méthode conjecturée était que, lorsque nous avions un nombre de colonnes ou de lignes impair, nous avions un nimber à 1 et un nimber à 0 sinon.

Cependant, nous nous sommes rendus compte que dans certains cas faciles, la méthode de résolution ne prenait pas en compte les composantes connexes identiques.

Nous avons donc cherché à redéfinir notre fonction pour attribuer le nimber et nous avons conjecturé les éléments suivants :

— le nimber d'une tablette 1xm ou mx1 est égal à m (prouvé)
— le nimber d'une tablette 2mx2n est égal à 0 (prouvé)
— le nimber d'une tablette nxm avec n ou m impair est égal à 1 si m+n est impair et 2 sinon (non prouvé)

La preuve de ce dernier point n'est pas évidente comme les deux premiers et nous avons travaillé avec Madame Selmi sans arriver à voir comment nous y prendre.

4 Apports du stage

Durant ce stage, j'ai eu l'occasion de pouvoir mettre en pratique la théorie acquise durant mes 3 années de licence.

Ceci m'a permis de voir que même si je pensais que certains cours ne seraient pas utiles car trop théoriques, ils avaient leur importance dans la logique et la gestion d'un problème technique et mathématique.

Ce stage a été aussi pour moi l'occasion d'apprendre un nouveau langage et de pouvoir valider mes connaissances théoriques dans les différents paradigmes de programmation comme la programmation impérative, la programmation orientée objet et la programmation fonctionnelle.

4.1 Connaissances acquises durant le stage

Ce stage m'a permis de mieux appréhender la gestion d'un projet. En effet, grâce à lui, j'ai eu l'occasion de pouvoir gérer mon travail seul et de pouvoir demander de l'aide aux moments où j'étais bloqué.

J'ai aussi pu élargir mes connaissances dans le milieu de l'informatique théorique au cours de différents séminaires et présentations au sein du laboratoire informatique.

4.2 Connaissances sur le métier d'enseignant chercheur

Durant ces deux mois de stage, j'ai pu avoir un aperçu du métier d'enseignant-chercheur. J'ai pu assister à différents événements au sein du laboratoire comme le séminaire de Normastic ou encore la soutenance de thèse de M. Ali Chouria.

Durant ces événements, j'ai pu voir qu'il nous restait encore beaucoup de travail avant de pouvoir être totalement apte à entrer dans le milieu de la recherche. Même si les exposés étaient clair et précis, nous nous sommes vite rendu compte qu'il nous restait encore beaucoup de bases à appréhender.

5 Conclusion

Durant ce stage, j'ai appris beaucoup sur le métier d'enseignant-chercheur et sur la recherche. Ce stage m'a aussi permis de pouvoir mieux appréhender mes inquiétudes sur mes choix d'orientation et de pouvoir être plus sûr de mon envie d'aller en sécurité informatique.

Je pense que ce stage m'a permis de voir un nouveau milieu professionnel que je ne connaissais pas et grâce à mes encadrants, les enseignants-chercheurs du département et mon collègue Yacine Hmito, j'ai pu faire un stage sans être perdu et ils m'ont beaucoup aidé et proposé des solutions qui m'ont permis d'avancer sans me perdre dans le travail.

Références

[Gru64] P.M. Grundy. Mathematics and games. Eureka, 1964.

[JP10] Delahaye Jean-Paul. Jeux finis et infinis. Éd. du Seuil, 2010.

[SG06] Cédric Leroy Samuel Giraudot. Rapport de Projet. 2006.

[Ule79] J. Ulehla. A complete analysis of von neumann's hackendot. Int. Journal of Game Theory, 1979.