Le problème de Berlekamp : résolution théorique, résolution pratique

Théorie des codes et recuit simulé

Clémentine Laurens
Année scolaire 2016-2017
Lycée Louis le Grand, classe de MP5

Sommaire

1 Position du problème : principe du *Berlekamp's Switching Game*

Le *Berlekamp's Switching Game* est un problème d'optimisation, dont l'énoncé peut se présenter sous la forme d'un jeu opposant deux personnes.

Considérons un tableau comportant n lignes et n colonnes, dont chaque case est occupée par une ampoule qui est soit éteinte, soit allumée. Le premier joueur doit imposer une configuration initiale, dans laquelle il choisit le nombre d'ampoules allumées et leur positionnement sur le tableau. Le second joueur n'a quant à lui accès qu'à des interrupteurs situés au bout de chaque ligne et de chaque colonne du tableau ($2n$ interrupteurs en tout), et dont le basculement a pour effet d'intervertir les états des ampoules sur la ligne ou la colonne en question (les ampoules qui étaient allumées s'éteignent, et celles qui étaient éteintes s'allument). Le second joueur peut manipuler les interrupteurs à l'infini, dans l'ordre qu'il souhaite.

L'objectif du second joueur est d'amener le tableau dans une situation où le nombre d'ampoules allumées est minimal. L'objectif du premier joueur est donc d'imposer une configuration initiale qui maximise le nombre minimal d'ampoules allumées que peut obtenir son adversaire, après manipulation illimitée des interrupteurs.

Considérons un exemple avec $n = 3$. Supposons que le premier joueur impose la configuration ci-dessous, dans laquelle les ronds blancs correspondent à des ampoules éteintes, et les ronds noirs à des ampoules allumées :

$$\begin{pmatrix} \bullet & \circ & \bullet \\ \circ & \bullet & \bullet \\ \bullet & \bullet & \circ \end{pmatrix}$$

Le second joueur peut alors obtenir successivement les situations suivantes, en actionnant un à un les interrupteurs marqués en bout de rangées dans les schémas ci-dessous :

$$\rightarrow \begin{pmatrix} \bullet & \circ & \bullet \\ \bullet & \circ & \circ \\ \bullet & \bullet & \circ \end{pmatrix}$$

$$\rightarrow \begin{pmatrix} \bullet & \circ & \bullet \\ \bullet & \circ & \circ \\ \circ & \circ & \bullet \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} \bullet & \circ & \circ \\ \bullet & \circ & \bullet \\ \circ & \circ & \circ \end{pmatrix}$$

$$\rightarrow \begin{pmatrix} \bullet & \circ & \circ \\ \circ & \bullet & \circ \\ \circ & \circ & \circ \end{pmatrix}$$

Cette dernière situation est en fait la meilleure que puisse obtenir le second joueur, à partir de la situation initiale qui lui était imposée.

La taille n du tableau étant fixée, deux problèmes peuvent alors être formulés :

Problème 1 (problème intermédiaire) : Étant donnée une configuration de départ quelconque, quelles sont les configurations optimales que le second joueur peut obtenir après manipulation illimitée des interrupteurs (i.e. quelles sont les configurations accessibles à partir de cette configuration de départ qui minimisent le nombre d'ampoules allumées) ? Résoudre ce problème revient en fait à déterminer une méthode algorithmique de recherche des configurations optimales pour le second joueur, à partir d'une situation de départ donnée.

Problème 2 (« Problème de Berlekamp ») : Quelle sont les configurations initiales optimales pour le premier joueur (i.e. quelles sont les configurations initiales qui maximisent le nombre minimal d'ampoules allumées que peut obtenir le second joueur, après manipulation illimitée des interrupteurs) ?

Nous aborderons tout d'abord ces deux questions par le biais de la théorie des codes qui, comme nous le verrons, est un cadre parfaitement adapté pour modéliser et résoudre ce problème d'un point de vue théorique. Toutefois, la complexité temporelle de l'algorithme déterminé par cette première étude nous contraindra à trouver une solution algorithmiquement plus efficace, permettant de traiter ces questions des des cas concrets (pour des tableaux de taille n raisonnable). Nous implémenterons donc une version adaptée de l'algorithme du « recuit simulé », en Python.

2 Quelques notions de théorie des codes

2.1 Aspect historique : approche intuitive

La théorie des codes s'est majoritairement developpée au XX$^{\text{ème}}$ siècle, avec la nécessité croissante de transmettre de l'information de manière fiable. Le problème auquel cette théorie apporte une solution peut être resumé ainsi : étant donnée une information (par exemple, un ensemble de huit lettres), comment faire pour la transmettre à un interlocuteur de la manière la plus fiable possible, c'est-à-dire en minimisant le risque de perte d'information (lié à un « brouillage »lors de la transmission) ?

La solution envisagée consiste à rajouter de l'information supplémentaire au moment de la transmission [1]. Pour la transmission d'un ensemble de huit lettres, cela reviendrait à ajouter, par exemple, quatre lettres à l'ensemble initial. L'enjeu est alors, pour le récepteur de l'information, de retrouver, à partir de l'ensemble de douze lettres reçu, l'ensemble de douze lettre initialement émis (qui est *a priori* différent de celui reçu, en raison du brouillage de l'information lors de la transmission), afin de pouvoir remonter à l'ensemble de huit lettres constituant l'information initiale.

2.2 Formalisation : définitions fondamentales

Soit $\mathbb{F}$ un corps fini, de cardinal $q \in \mathbb{N}$. Soit $n \in \mathbb{N}$.

Définition 1 (Alphabet)**.** *Les scalaires de $\mathbb{F}$ composent l'alphabet.*

L'alphabet permet d'exprimer l'information que l'on souhaite transmettre.

Définition 2 (Code linéaire)**.** *Soit $k \in [\![0, n]\!]$. On appellera code linéaire de longueur n et de dimension k sur $\mathbb{F}$ un sous-espace vectoriel C de $\mathbb{F}^n$ de dimension k. On notera : $C(n, k)$.*

1. C'est ainsi que les aviateurs prononcent « Alpha Tango Charlie » pour transmettre les seules lettres « ATC », afin de minimiser la perte d'information liée au brouillage des ondes radio.

Soit un tel code linéaire $C(n,k)$.

Définition 3 (Mots). *Les éléments de $C(n,k)$ sont appelés mots du code.*

Pour faire le lien avec l'approche intuitive proposée dans le paragraphe précédent, l'information que l'on souhaite transmettre est un vecteur de $\mathbb{F}^k$, et la méthode de transmission consiste à lui associer un mot du code $C(n,k)$, qui sera l'information effectivement émise.

Définition 4 (Codage d'un vecteur). *On appelle codage d'un vecteur l'opération consistant à associer à un vecteur de l'espace vectoriel $\mathbb{F}^k$ un mot du code $C(n,k)$.*

Définition 5 (Décodage d'un vecteur). *On appelle décodage d'un vecteur l'opération consistant à associer à un vecteur de l'espace vectoriel $\mathbb{F}^n$ un mot du code $C(n,k)$.*

L'enjeu, pour l'émetteur de l'information, est donc de coder de manière unique le vecteur de $\mathbb{F}^k$ qu'il souhaite transmettre, en lui associant un (et un seul) mot du code $C(n,k)$. Pour le récepteur de l'information, il s'agit ensuite de décoder le vecteur de $\mathbb{F}^n$ reçu, afin de retrouver le mot du code $C(n,k)$ que l'émetteur a envoyé, et donc, par unicité, le vecteur initial de $\mathbb{F}^k$.

Munissons $\mathbb{F}^n$ d'une distance d.

Définition 6 (Poids d'un vecteur). *On appelle poids d'un vecteur colonne u de $\mathbb{F}^n$, et on note $w(u)$, sa distance au vecteur nul.*

Définition 7 (Rayon de recouvrement). *On appelle rayon de recouvrement d'un code $C(n,k)$, et on note $\rho(C)$, la plus grande distance entre un vecteur de $\mathbb{F}^n$ et un mot de $C(n,k)$:*

$$\rho(C) = \max_{u\in\mathbb{F}^n}\left(\min_{c\in C}(d(c,u)) \right) = \max_{u\in\mathbb{F}^n}(d(C,u))$$

3 Du *Berlekamp's Switching Game* à la théorie des codes

3.1 Modélisation mathématique du problème

Soit $n \in \mathbb{N}$. Notons $\mathcal{M}(n)$ l'ensemble des matrices carrées binaires de taille n : $\mathcal{M}(n) = \mathcal{M}_n(\mathbb{F}_2)$. $\mathcal{M}(n)$ est un $\mathbb{F}_2$-espace vectoriel de dimension n^2.

Les tableaux d'ampoules de taille n du *Berlekamp's Switching Game* peuvent être modélisés par des matrices de $\mathcal{M}(n)$, dans lesquelles chaque coefficient 1 représente une ampoule allumée, et chaque coefficient 0 une ampoule éteinte. Plus précisément, ceci définit une bijection de l'ensemble de ces tableaux d'ampoules sur $\mathcal{M}(n)$.

Considérons le sous-espace vectoriel $\mathcal{C}$ de $\mathcal{M}(n)$ engendré par les matrices de $\mathcal{M}(n)$ comportant exactement une ligne ou une colonne de 1, et des 0 partout ailleurs. $\mathcal{C}$ peut être considéré comme un code linéaire de longueur n^2 et de dimension $2n-1$ (en identifiant, au besoin, $\mathcal{M}(n)$ à $(\mathbb{F}_2)^{n^2}$). En effet, notons, $\forall (i,j) \in (\llbracket 1,n \rrbracket)^2$, A_i la matrice de $\mathcal{M}(n)$ comportant des 0 partout sauf sur la i-ème ligne (qui ne comporte, elle, que des 1), et B_j la matrice comportant des 0 partout sauf sur la j-ème colonne (qui ne comporte que des 1). On vérifie alors facilement que la famille $\{A_i, B_j \mid i \in \llbracket 1,n \rrbracket,\ j \in \llbracket 1, n-1 \rrbracket\}$ est une base de $\mathcal{C}$.

Soit une configuration initiale pour notre tableau d'ampoules, modélisée par une matrice S de $\mathcal{M}(n)$. Les positions que peut atteindre le second joueur à partir de cette position initiale, en manipulant à sa guise les interrupteurs en bout de lignes et de colonnes, sont donc exactement les positions modélisées par les matrices de l'ensemble :

$$\overline{S} = S + \mathcal{C} = \{(S+C) \mid C \in \mathcal{C}\}$$

3.2 Une distance adaptée au problème : la distance de Hamming

Munissons $\mathcal{M}(n)$ de la distance de Hamming, définie ci-dessous.

Définition 8 (Distance de Hamming)**.** *Soient M_1 et M_2 deux vecteurs de $\mathcal{M}(n)$. La distance de Hamming $d_H(M_1, M_2)$ entre M_1 et M_2 est le nombre de composantes pour lesquelles ces deux vecteurs diffèrent.*

L'intérêt de cette distance est double, pour le problème qui nous préoccupe. Tout d'abord, $\mathcal{M}(n)$ étant muni de d_H, le poids d'un vecteur M de $\mathcal{M}(n)$ correspond exactement au nombre d'ampoules allumées dans le tableau modélisé par M. On notera : $\forall M \in \mathcal{M}(n),\ w_H(M) = d_H(M, 0_{\mathcal{M}(n)})$. Les deux problèmes énoncés en introduction peuvent donc être reformulés de la manière suivante :

Problème 1, reformulation : Étant donnée une configuration de départ quelconque, modélisée par une matrice S de $\mathcal{M}(n)$, on recherche les configurations modélisées par les matrices S' de $\overline{S}$ vérifiant :

$$w_H(S') = \min_{M \in \overline{S}} (w_H(M))$$

Autrement dit, on recherche l'ensemble des matrices de $\overline{S}$ de poids minimal.

Problème 2, reformulation : On recherche l'ensemble des configurations modélisées par des matrices S de $\mathcal{M}(n)$ telles que, $\forall S' \in \overline{S}$ vérifiant $w_H(S') = \min_{M \in \overline{S}} (w_H(M))$, on ait :

$$w_H(S') = \max_{M \in \mathcal{M}(n)} \left(\min_{P \in \overline{M}} (w_H(P)) \right)$$

Il s'agit de l'ensemble des matrices S de $\mathcal{M}(n)$ qui maximisent le poids minimal d'une matrice de $\overline{S}$.

Pour toute matrice S de $\mathcal{M}(n)$ notons $F(S)$ le poids minimal d'une matrice de $\overline{S}$:

$$F(S) \ = \ \min_{S' \in \overline{S}} (w_H(S')) \ = \ \min_{C \in \mathcal{C}} (w_H(S + C))$$

D'après ce qui précède, ce poids correspond au nombre minimal d'ampoules allumées que peut obtenir le second joueur en manipulant à sa guise les interrupteurs en bout de lignes et de colonnes à partir de la configuration modélisée par la matrice S. Résoudre le problème 1 revient donc à chercher, à partir d'une configuration initiale quelconque, modélisée par une matrice S de $\mathcal{M}(n)$, les matrices de $\overline{S}$ de poids $F(S)$.

Notons de plus :

$$f(n) = \max_{S \in \mathcal{M}(n)} (F(S)).$$

Le second avantage que présente la distance de Hamming réside dans la propriété suivante, dont la démonstration ne pose aucune difficulté :

Proposition 1 (Invariance par translation)**.** *La distance de Hamming est invariante par translation :*

$$\forall (M_1, M_2, M) \in \mathcal{M}(n)^3,\ d_H(M_1, M_2) = d_H(M_1 + M, M_2 + M)$$

Ceci permet de démontrer le théorème fondamental ci-dessous :

Théorème 1. *$f(n)$ est le rayon de recouvrement du code $\mathcal{C}$:*

$$f(n) = \rho(\mathcal{C})$$

Preuve. Soit une configuration initiale modélisée par une matrice S de $\mathcal{M}(n)$. Soit C_S le mot du code $\mathcal{C}$ le plus proche de S (au sens de d_H). Par construction et d'après la proposition 1 :

$$\begin{aligned} d_H(S, C_S) &= w_H(S - C_S) \\ &= \min_{C \in \mathcal{C}} (w_H(S - C)) \\ &= d_H(S,\ \mathcal{C}) \\ &= F(S) \end{aligned}$$

Donc :

$$\begin{aligned} f(n) &= \max_{S \in \mathcal{M}(n)} (F(S)) \\ &= \max_{S \in \mathcal{M}(n)} (d_H(S, \mathcal{C})) \\ &= \max_{S \in \mathcal{M}(n)} \left(\min_{C \in \mathcal{C}} (d_H(S, C)) \right) \\ &= \rho(\mathcal{C}) \end{aligned}$$

□

3.3 Réduction du problème

Le *Berlekamp's Switching Game* est équivalent à un problème de théorie des codes, puisqu'avec les notations introduites ci-dessus, et grâce à la bijection de l'ensemble des tableaux d'ampoules de taille n du *Berlekamp's Switching Game* sur $\mathcal{M}(n)$, les deux problèmes qui nous préoccupent peuvent être résumés ainsi :

Réduction du problème 1 : Soit une matrice S de $\mathcal{M}(n)$, on recherche les matrices S' de $\mathcal{M}(n)$ vérifiant :
- $\exists C \in \mathcal{C},\ S' = S + C$
- $w_H(S') = \min_{C \in \mathcal{C}} (w_H(S + C)) = F(S)$

Réduction du problème 2 : On recherche l'ensemble des matrices S de $\mathcal{M}(n)$ vérifiant :

$$\begin{aligned} \min_{C \in \mathcal{C}} (w_H(S + C)) &= \max_{M \in \mathcal{M}(n)} \left(\min_{C \in \mathcal{C}} (w_H(M + C)) \right) \\ &= \max_{M \in \mathcal{M}(n)} (F(M)) \\ &= f(n) \\ &= \rho(\mathcal{C}) \end{aligned}$$

Ces deux problèmes peuvent donc être résolus théoriquement avec les outils de la théorie des codes.

4 Résolution théorique : groupe quotient et tableau standard

4.1 Résolution théorique

L'utilisation de tableaux standards est une méthode classique de décodage d'un vecteur, en théorie des codes. On propose ci-dessous une adaptation de cette méthode classique pour résoudre le problème qui nous préoccupe.

On définit sur $\mathcal{M}(n)$ la relation $\mathcal{R}$ par :

$$\forall (M_1, M_2) \in (\mathcal{M}(n))^2, \quad M_1 \ \mathcal{R} \ M_2 \iff \exists C \in \mathcal{C}, \ M_1 = M_2 + C \iff (M_1 - M_2) \in \mathcal{C}$$

On vérifie très facilement que $\mathcal{R}$ définit une relation d'équivalence sur $\mathcal{M}(n)$. De plus, on constate que deux matrices M_1 et M_2 de $\mathcal{M}(n)$ sont dans la même classe d'équivalence de $\mathcal{R}$ si et seulement si le second joueur peut, à partir de la configuration modélisée par la matrice M_1, atteindre la configuration modélisée par la matrice M_2 en actionnant les interrupteurs de bout de lignes et de colonnes. Ainsi, la classe d'équivalence d'une matrice S de $\mathcal{M}(n)$ pour la relation $\mathcal{R}$ est exactement l'ensemble $\overline{S}$ définit dans la partie 3.1.

Proposition 2. *Les classes d'équivalence de $\mathcal{R}$ sont toutes de même cardinal que $\mathcal{C}$, à savoir 2^{2n-1}.*

Preuve. Soit $M \in \mathcal{M}(n)$. Alors la classe d'équivalence de M est l'ensemble $\overline{M} = \{M + C \mid C \in \mathcal{C}\}$. Donc $\forall M \in \mathcal{M}(n)$, $Card\left(\overline{M}\right) = Card\left(\mathcal{C}\right)$, et $\mathcal{C}$ étant un $\mathbb{F}_2$-espace vectoriel de dimension $2n-1$, avec $\mathbb{F}_2$ de cardinal 2, on a bien $Card\left(\mathcal{C}\right) = 2^{2n-1}$.

□

Proposition 3. *$\mathcal{R}$ admet exactement $2^{\left(n^2-2n+1\right)}$ classes d'équivalence, et l'ensemble de ces classes d'équivalence forme le groupe quotient $\mathbb{F}^n/\mathcal{C}$.*

Preuve.

— $Card\left(\mathcal{M}(n)\right) = Card\left(\mathbb{F}_2\right)^{dim(\mathcal{M}(n))} = 2^{n^2}$. Ainsi, $\mathcal{R}$ admet bien $\frac{Card(\mathcal{M}(n))}{Card(\mathcal{C})} = 2^{\left(n^2-2n+1\right)}$ classes d'équivalence.

— $(\mathcal{M}(n), +)$ est un groupe abélien (cf. structure d'espace vectoriel), et $(\mathcal{C}, +)$ est un sous-groupe distingué de $(\mathcal{M}(n), +)$ (cf. commutativité de la loi $+$). L'ensemble des classes d'équivalence de la relation $\mathcal{R}$ est donc, par définition, l'ensemble quotient $\mathbb{F}^n/\mathcal{C}$.

— Démontrons enfin que cet ensemble quotient peut être muni d'une structure de groupe. Définissions sur $\mathbb{F}^n/\mathcal{C}$ la loi interne $+_q$ par :

$$\forall (M_1, M_2) \in (\mathcal{M}(n))^2, \overline{M_1} +_q \overline{M_2} = \overline{M_1 + M_2}$$

Cette loi (dite « loi induite de $(\mathcal{M}(n), +)$ sur $\mathbb{F}^n/\mathcal{C}$ ») est bien définie, car $(\mathcal{C}, +)$ étant un sous-groupe distingué de $(\mathcal{M}(n), +)$, on a :

$$\forall (M_1, M_2, M_1', M_2') \in (\mathcal{M}(n))^4 \text{ tel que } \overline{M_1} = \overline{M_1'} \text{ et } \overline{M_2} = \overline{M_2'}, \text{ alors } \overline{M_1 + M_2} = \overline{M_1' + M_2'}$$

Soit en effet $(M_1, M_2, M_1', M_2') \in (\mathcal{M}(n))^4$ tel que $\overline{M_1} = \overline{M_1'}$ et $\overline{M_2} = \overline{M_2'}$.

Alors $\exists (c_1, c_2) \in \mathcal{C}^2$ tel que $M_1 = M_1' + c_1$ et $M_1 = M_1' + c_2$, i.e. $(M_1 - M_1') = c_1$ et $(M_2 - M_2') = c_2$.

Alors : $M_1 + M_2 - (M_1' + M_2') = M_1 - M_1' + M_2 - M_2' = c_1 + c_2$.

Donc $M_1 + M_2 - (M_1' + M_2') \in \mathcal{C}$, donc $\overline{M_1 + M_2} = \overline{M_1' + M_2'}$.

Ainsi, $0_{\mathcal{M}(n)}$ est neutre pour la loi $+_q$, et $\forall\ \overline{M} \in \mathbb{F}^n/\mathcal{C}$, $\overline{M}$ possède un inverse pour cette loi, donné par $\overline{-M}$. L'associativité de $+_q$ étant immédiate, on obtient donc bien que $(\mathbb{F}^n/\mathcal{C}, +_q)$ est un groupe.

□

Définition 9 (Chef de classe)**.** *On appelle chef de classe d'une classe d'équivalence de $\mathcal{R}$ l'élément de poids minimal de cette classe (en prenant, au besoin, un élément au hasard parmi les éventuels ex-æquo).*

Ainsi, pour toute matrice S de $\mathcal{M}(n)$, le chef de la classe d'équivalence $\overline{S}$ est une matrice S' de $\mathcal{M}(n)$ vérifiant : $w_H(S') = F(S)$.

Notons $\mathcal{C} = \{C_j \mid j \in [\![1, 2^{2n-1}]\!]\}$ avec $C_1 = 0_{\mathcal{M}(n)}$, et notons $\{M_i \mid i \in [\![1, 2^{\left(n^2-2n+1\right)}]\!]\}$ l'ensemble des chefs de classe des classes d'équivalence de $\mathcal{R}$, avec $M_1 = 0_{\mathcal{M}(n)}$ (cette dernière notation ayant bien un sens, puisque le vecteur nul est, de manière évidente, le chef de sa propre classe d'équivalence).

Définition 10 (Tableau standard)**.** *On appelle tableau standard du code $\mathcal{C}$ le tableau de taille $2^{\left(n^2-2n+1\right)} \times 2^{2n-1}$ dont l'élément en position (i, j) est $M_i + C_j$.*

Le tableau standard du code $\mathcal{C}$ est donc un tableau contenant les 2^{n^2} vecteurs de $\mathcal{M}(n)$, dont les lignes correspondent aux classes d'équivalence de $\mathcal{R}$. Grâce aux notations choisies, sa première colonne regroupe l'ensemble des chefs de classe de ces classes d'équivalence, et sa première ligne les vecteurs du code $\mathcal{C}$.

Une fois construit le tableau standard du code $\mathcal{C}$, les deux problèmes qui nous préoccupent peuvent donc être résolus par simple lecture de ce tableau.

Résolution du problème 1 par lecture du tableau standart du code $\mathcal{C}$: Étant donnée une matrice S de $\mathcal{M}(n)$, pour trouver l'ensemble des matrices S' recherché, il suffit de trouver S dans le tableau standard du code $\mathcal{C}$, puis de rechercher sur la ligne sur laquelle se trouve S dans le tableau standard les matrices de même poids que le chef de la classe d'équivalence de S, i.e. les matrices de même poids que la première matrice de cette ligne.

Résolution du problème 2 par lecture du tableau standard du code $\mathcal{C}$: L'ensemble des matrices S de $\mathcal{M}(n)$ recherché est exactement l'ensemble des matrices de même poids que le chef de classe de poids le plus élevé. On trouve ce chef de classe de poids le plus élevé en lisant la première colonne du tableau standard, et il suffit ensuite de rechercher dans le reste du tableau les matrices de même poids.

Notons que le poids maximal d'un chef de classe est donc, par définition, $f(n) = \rho(\mathcal{C})$.

4.2 Limites de cette résolution : problèmes de complexité

Le premier obstacle à la résolution effective d'un problème de Berlekamp par lecture de tableau standard est la construction du tableau en question. En effet, comme précisé ci-dessus, celui-ci contient les 2^{n^2} vecteurs de $\mathcal{M}(n)$: pour des tableaux d'ampoules de taille $n = 10$, le tableau standard contient donc plus de 10^{30} vecteurs ! Définir, modéliser et trier par classes d'équivalence une telle quantité de

vecteurs représente donc un travail bien trop long et fastidieux pour être réalisé en pratique par un ordinateur.

De plus, même une fois le tableau standard construit, sa lecture est un procédé algorithmique présentant une très mauvaise complexité, et pour cette raison, il s'agit d'une méthode de décodage très peu utilisée. Il est donc nécessaire, afin de pouvoir étudier des cas concrets, de trouver un compromis algorithmique qui permette la résolution de notre problème en alliant une fiabilité convenable, un temps de calcul correct et des besoins en mémoire raisonnables.

5 Une solution algorithmique efficace

Le langage de programmation utilisé dans toute cette partie est Python.

5.1 Modélisation informatique et définition d'une classe adaptée (section légèrement incomplète)

Afin d'éviter tout ambiguïté, commençons par préciser que, dans la suite de ce document, on pourra parler de la « première ligne » pour évoquer la ligne du haut d'un tableau, de la « dernière ligne » pour évoquer la ligne du bas, de la « première colonne » pour évoquer la colonne de gauche et de la « dernière colonne » pour évoquer la colonne de droite.

On représente chaque vecteur de $\mathcal{M}(n)$ par un entier naturel, dont la représentation binaire correspond à l'ensemble des n lignes de la matrice en question mises bout à bout, en commençant par la ligne du haut.

Considérons par exemple le tableau représenté par la matrice de $\mathcal{M}(3)$ ci-dessous :

$$\begin{pmatrix} \mathbf{1} & \mathbf{0} & \mathbf{1} \\ 1 & 1 & 1 \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{pmatrix}$$

Ce vecteur est alors représenté de manière unique par l'entier dont la représentation binaire est :

$$\overline{\mathbf{101}111\mathbf{001}}^2$$

à savoir 377.

Une telle modélisation numérique possède deux avantages majeurs : d'une part, le faible espace en mémoire nécessaire au stockage des entiers (ce qui autorise à garder en mémoire un grand nombre d'informations sans qu'il n'y ait de problème de saturation), et d'autre part la facilité d'implémentation de certaines opérations sur les tableaux d'ampoules, comme on pourra le constater dans la suite de cette section.

Pour simplifier le travail de programmation, on crée une nouvelle classe afin de regrouper les informations utiles à la résolution du problème dans un unique objet informatique :

```
1 class panel :
2     def __init__(self, nbl, nbc) :
3         self.nbl = nbl
```

```
4           self.nbc = nbc
5           self.n = nbl*nbc
6           self.max = 2**self.n-1
7
8
9           self.shift = 2**nbc
10          self.lastlgn = self.shift-1
11          self.lastcol = sum(2**(nbc*i) for i in range(nbl))
12
13          lgn = self.lastlgn
14          res = []
15          for _ in range(nbl) :
16              res.append(lgn)
17              lgn <<= nbc
18          self.lgns = list(reversed(res))
19
20          col = self.lastcol
21          res = []
22          for _ in range(nbc) :
23              res.append(col)
24              col <<= 1
25          self.cols = list(reversed(res))
26
27          self.nbones = { 0:0 }
28          for i in range(nbc) :
29              tmp = {}
30              for elem in self.nbones :
31                  tmp[elem] = self.nbones[elem]
32                  tmp[elem+(1<<i)] = self.nbones[elem] + 1
33              self.nbones = tmp
34
35          dic2 = { 0:0 }
36          for i in range(nbl) :
37              tmp = {}
38              for elem in dic2 :
39                  tmp[elem] = dic2[elem]
40                  tmp[elem+(1<<(i*nbc))] = dic2[elem] + 1
41              dic2 = tmp
42
43          self.nbones.update(dic2)
```

Ainsi, un tableau d'ampoules est modélisé par un objet de la classe panel, lequel est entièrement défini par les dimensions du tableau modélisé (nombre de lignes et nombre de colonnes). Alors, si M est un objet de la classe panel, M.nbl retourne le nombre de lignes du tableau d'ampoules représenté par M, M.nbc son nombre de colonnes, M.n le nombre total d'ampoules du tableau en question, et M.max l'entier représentant la configuration dans laquelle toutes les ampoules de ce tableau sont allumées. Remarquons que, dans la mesure où on ne s'intéresse ici qu'à des tableaux carrés, on aura toujours M.nbl = M.nbc. Toutefois, notre étude est généralisable sans difficulté à des tableaux non carrés, aussi est-il utile de faire cette distinction dans le programme proposé.

De plus, comme illustré ci-dessous, M.shift retourne l'entier tel que, si l'on considère l'unique entier modélisant le tableau représenté par M **dans une configuration dans laquelle la première ligne**

d'ampoules est éteinte, en multipliant cet entier par M.shift, on obtient l'unique entier modélisant le tableau représenté par M dans la configuration dans laquelle on a, à partir de la configuration initiale, « fait monter » toutes les lignes du tableau d'un cran, et inséré une ligne d'ampoules éteintes en bas du tableau.

Étudions par exemple le cas où M représente un tableau carré de taille 5, et considérons la configuration simple modélisée par la matrice de $\mathcal{M}(n)$ suivante (configuration dans laquelle seule la dernière ligne du tableau est allumée) :

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Cette matrice est représentée par l'entier dont la représentation binaire est :

$$\overline{000\cdots0011111}^2$$

à savoir 31. Il suffit alors de multiplier cet entier par M.shift $= 2^5$ pour obtenir 992, dont la représentation binaire est :

$$\overline{000\cdots001111100000}^2$$

992 représente donc la matrice de $\mathcal{M}(5)$:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

laquelle représente bien un tableau d'ampoules obtenu à partir du tableau initial en appliquant le procédé décrit ci-dessus.

Par ailleurs, M.lastlgn retourne l'unique entier représentant la configuration dans laquelle seule la dernière ligne du tableau représenté par M est allumée, et M.lastcol retourne l'unique entier représentant la configuration dans laquelle seule la dernière colonne du tableau représenté par M est allumée. Ainsi, d'après ce qui précède, si M représente un tableau carré de taille 5, alors M.lastlgn $= 31$. De même, la configuration dans laquelle seule la dernière colonne du tableau représenté par M est allumée est alors représentée par la matrice de $\mathcal{M}(5)$:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

elle-même représentée par l'entier M.lastcol dont la représentation binaire est :

$$\overline{0000100001\cdots00001}^2$$

Donc M.lastcol = 1082401.

Les trois grandeurs M.shift, M.lastlgn et M.lastcol permettent alors de définir M.lgns, qui renvoie une liste contenant tous les entiers représentant les configurations dans lesquelles seule une ligne du tableau représenté par M est allumée, en commençant par l'entier représentant la configuration dans laquelle seule la **première** ligne du tableau en question est allumée (et en terminant par la configuration dans laquelle seule la **dernière** ligne du tableau est allumée).

Pour illustrer la méthode de construction de la liste M.lgns, considérons que M modélise un tableau de taille 4. On commence par créer une liste res dans laquelle on place l'entier représentant la matrice de $\mathcal{M}(4)$:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

c'est-à-dire l'entier dont la représentation binaire est :

$$\overline{00\cdots00001111}^2$$

à savoir M.lastlgn = 15.

On rajoute ensuite, « à la fin » de la représentation binaire de cet entier, M.nbc = 4 zéros. On obtient donc l'entier dont la représentation binaire est :

$$\overline{00\cdots011110000}^2$$

à savoir 240. Cet entier représente la matrice de $\mathcal{M}(4)$:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Remarquons que, par construction, définir un entier en « rajoutant M.nbc zéros à la fin de la représentation binaire de l'entier M.lastlgn » revient à définir l'entier M.lastlgn × M.shift.

On ajoute alors à la liste res l'entier 240 ainsi obtenu, et on réitère le processus pour rajouter à la liste les entiers modélisant les configurations représentées par les matrices de $\mathcal{M}(4)$:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

et

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Pour finir, on inverse l'ordre des éléments dans la liste res, et on affecte cette valeur à la variable M.lgns.

De même, M.cols renvoie une liste contenant tous les entiers représentant les configurations dans lesquelles seule une colonne du tableau représenté par M est allumée, en commençant par l'entier représentant la configuration dans laquelle seule la **première** colonne du tableau en question est allumée (et en terminant par la configuration dans laquelle seule la **dernière** colonne du tableau est allumée).

Illustrons comme précédemment la méthode de construction de la liste M.cols par un exemple où M modélise un tableau de taille 4. On commence ici aussi par créer une liste res, dans laquelle on place l'entier représentant la matrice :

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

c'est à dire l'entier dont la représentation binaire est :

$$\overline{00010001\cdots0001}^2$$

à savoir M.lastcol $= 4369$.

On rajoute ensuite, « à la fin » de la représentation binaire de cet entier, un zéro. On obtient donc l'entier dont la représentation binaire est :

$$\overline{00100010\cdots0010}^2$$

à savoir 8738. Cet entier représente la matrice de $\mathcal{M}(4)$:

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

On ajoute alors à la liste res l'entier 8738 ainsi obtenu, et on réitère le processus pour rajouter à la liste les entiers modélisant les configurations représentées par les matrices de $\mathcal{M}(4)$:

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

et

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Pour finir, on inverse l'ordre des éléments dans la liste res, et on affecte cette valeur à la variable M.cols.

Enfin, M.nbones, qui renvoie un dictionnaire qui associe à chaque entier représentant une configuration du tableau M le nombre d'ampoules allumées dans cette configuration.

[Reste à expliciter la méthode de construction de M.nbones$\cdots$]

5.2 Implémentation des fonctions de base (section incomplète)

On définit dans cette partie un certain nombre de fonctions qui seront utiles à la résolution informatique du *Problème de Berlekamp*.

Notons qu'on aura au préalable réalisé les imports suivants :

```
from textwrap import wrap
from random import random, randint
from matplotlib.pyplot import plot, matshow, show, hist
from math import exp
import numpy as np
```

La fonction ci-dessous prend en argument un entier x représentant une configuration pour un tableau représenté par un objet *data* de la classe panel, et affiche dans la console ce tableau dans la configuration représentée par l'entier x :

```
def Disp(x, data) :
    print("\n".join(wrap(bin(x)[2:].zfill(data.n), data.nbc)
        ))
```

La fonction ci-dessous prend en argument un entier x représentant une configuration pour un tableau représenté par un objet *data* de la classe panel, et affiche graphiquement (grâce aux fonctions de la bibliothèque **matplotlib.pyplot**) ce tableau dans la configuration représentée par l'entier x :

```
def Draw(x, data) :
    m = np.array( [ int(c) for c in bin(x)[2:].zfill(data.n)
        ] )
    m.shape = (data.nbl, data.nbc)
    matshow(m, cmap='gray')
    show()
```

La fonction ci-dessous prend en argument un entier x représentant une configuration pour un tableau dans une certaine configuration, et renvoie le nombre de lampes allumées dans cette configuration :

```
def Score(x) :
    return bin(x).count("1")
```

La fonction ci-dessous prend en argument un entier x représentant une configuration pour un tableau dans une certaine configuration pour un tableau représenté par un objet *data* de la classe panel ainsi qu'un indice i (entier compris entre 0 inclus et le nombre de lignes du tableau exclus), et renvoie la i-ième ligne dans l'état représenté par l'entier x :

```
def GetLine(x, i, data) :
    return (x >> (data.nbc*(data.nbl-i-1))) & data.lastlgn
```

```
def GetCol(x, i, data) :
    return (x >> (data.nbc-i-1) & data.lastcol)
```

```
def CountLine(x, i, data) :
    return data.nbones[GetLine(x, i, data)]
```

```
def CountCol(x, i, data) :
    return data.nbones[GetCol(x, i, data)]
```

```
def CountSet(x, i, data) :
    if i<data.nbl :
        return data.nbones[GetLine(x, i, data)]
    return data.nbones[GetCol(x, i-data.nbl, data)]
```

```
def ChangeSet(x, i, data) :
    if i<data.nbl :
        return data.nbc - 2*data.nbones[GetLine(x, i, data)]
    return data.nbl - 2*data.nbones[GetCol(x, i-data.nbl,
       data)]
```

```
def SwitchLine(x, i, data) :
    return x ^ data.lgns[i]
```

```
def SwitchCol(x, i, data) :
    return x ^ data.cols[i]
```

```
def SwitchSet(x, i, data) :
    if i<data.nbl :
        return x ^ data.lgns[i]
```

```
4       return x ^ data.cols[i-data.nbl]
```

5.3 Principe général du « recuit simulé »

On se propose d'appliquer au *Berlekamp's Swiching Game* une méthode algorithmique inspirée de la physique statistique : celle du « recuit simulé » (« simulated annealing », en Anglais), qui fournit d'excellents résultats pour de très nombreux problèmes d'optimisation mathématique.

Il s'agit plus précisément d'une méthode récursive visant à déterminer les extrema d'une fonction numérique sur un certain ensemble, au moyen d'une distribution de probabilités bien choisie.

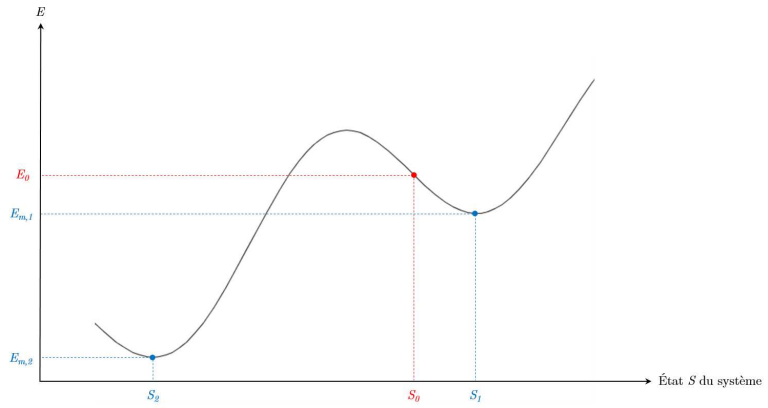
Supposons sans perte de généralité que l'on cherche à minimiser une fonction numérique E, dépendant d'un certain nombre de paramètres définissant l'état d'un système. Définissons également un paramètre numérique T à valeur dans $\mathbb{R}_+^*$, qui ne dépend pour sa part que du temps écoulé depuis le lancement de l'algorithme (on reviendra ultérieurement sur la définition et l'utilité d'un tel paramètre). Le principe du « recuit simulé » est alors le suivant :

Initialisation : On choisit un état initial du système, dans lequel la fonction E a une valeur E_0, et on fixe la valeur initiale T_0 de la fonction T.

Récursion : On effectue une « modification élémentaire » de l'état du système, et on calcule les valeurs de E et T dans le nouvel état ainsi atteint. Alors :

1. Si la nouvelle valeur de E est inférieure à la précédente, on conserve ce nouvel état et on réitère récursivement le processus.
2. Si la nouvelle valeur de E est supérieure de $\Delta E > 0$ à l'ancienne, on conserve ce nouvel état avec une probabilité $p = \exp\left(-\frac{\Delta E}{T}\right)$, et on réitère récursivement le processus.

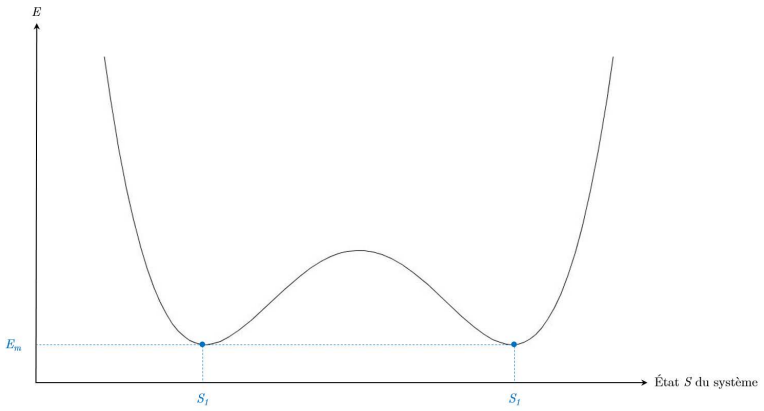
L'acceptation avec une probabilité non nulle d'une modification de l'état du système induisant une augmentation de E permet de s'assurer que l'algorithme recherche bien un minimum global de la fonction, et non un simple minimum local au voisinage de l'état initial du système. Considérons en effet la situation représentée sur la figure ci-dessous, sur laquelle est indiqué l'état S_0 à partir duquel on exécute l'algorithme.



Si on ne conservait que les modifications de l'état du système induisant une diminution de E, on ne pourrait détecter que le minimum local $E_{m,1}$, atteint pour l'état du système S_1, et non le minimum

global $E_{m,2}$, atteint pour l'état du système S_2.

Toutefois, avec un profil pour la fonction E comme celui présenté dans la figure ci-dessous, cette acceptation de modifications de l'état du système induisant une augmentation de E pourrait, *a priori*, avoir pour conséquence que l'algorithme passe sans cesse de la détection d'un minimum local à un autre, voisin, sans parvenir à s'arrêter sur l'un d'entre eux...



Dans une telle situation, l'algorithme semble en effet pouvoir se rapprocher alternativement des états S_1 et S_2 du système, sans jamais se fixer sur l'un des deux.

C'est ici que le paramètre T trouve toute son importance : pour éviter cet écueil, on impose une décroissance de T en fonction du temps écoulé depuis le début de l'exécution de l'algorithme. Ceci permet, à variation $\Delta E > 0$ égale, une décroissance au cours du temps de la probabilité p avec laquelle on conserve la modification de l'état du système induisant cette augmentation de E. Autrement dit, plus le temps s'écoule, moins on a de chances de s'éloigner d'un état du système fournissant un minimum local « satisfaisant ».

Le mode de décroissance de la fonction T au cours du temps est souvent choisi de manière empirique. Il fera ici l'objet d'un travail d'optimisation informatique, décrit dans la partie 5.6 de ce document.

5.4 Quelques améliorations (section incomplète)

5.5 Algorithme de résolution (section incomplète)

```
1  def Minimize(pos, data, nbsteps=10000, maxcounter=30,
       earlyabort=0, coeffA = 1.0, coeffB = 1.0, coeffC = 2.0) :
2
3      def T(i) :
4          return (1-i/nbsteps)
5
6      coeffC *= (data.nbc + data.nbl)
7      def P(Efrom, Eto, Ebest, T) :
8          diff = coeffA * (Eto - Ebest) + coeffB * (Eto -
               Efrom)
9          return exp(-diff / (coeffC*T))
10
11     score = Score(pos)
```

```
12      best, bestscore, counter = pos, score, 0
13      lst = [ score ]
14
15      for i in range(nbsteps) :
16          set = randint(0, data.nbl + data.nbc - 1)
17          nextscore = score + ChangeSet(pos, set, data)
18
19          if nextscore < score or random() < P(score,
                nextscore, bestscore, T(i)) :
20              pos = SwitchSet(pos, set, data)
21              score = nextscore
22
23          if score < bestscore :
24              best, bestscore = pos, score
25              counter = 0
26
27          if score > bestscore :
28              counter += 1
29              if counter > maxcounter :
30                  counter, pos, score = 0, best, bestscore
31
32          lst.append(score)
33
34          if score <= earlyabort :
35              break
36
37      return best, bestscore, lst
```

5.6 Optimisation (section incompplète)

5.7 Résultats (section incomplète)

6 Conclusion (section incomplète)

7 Annexes

7.1 Programme informatique complet (Python) (section incomplète)

On trouvera ci-dessous le programme informatique complet, en Python, qui est le fruit de ce travail.

7.2 Documents SCEI

Ci-après se trouvent le MCOT et le rapport de TIPE déposés sur le site SCEI dans le courant de l'année 2017.