Université de Lille

# Modèle de Potts pour la classification

Master Mathématiques et Applications

par Khalifa Naïl et Briouat Farid

Département de Mathématiques
Faculté des sciences et technologies

Mémoire présenté à la Faculté des sciences et technologies en vue de l'UE TER

Mai 2023

KHALIFA Naïl et BRIOUAT Farid

# Table des matières

# Remerciements

Nous tenons à remercier toutes les personnes qui nous ont aidés lors de la rédaction de ce mémoire.

Nous remercions Monsieur Nicolas Wicker professeur de mathématiques appliquées à l'Université de Lille, pour son encadrement, sa disponibilité et son aide précieuse.

Nous remercions Madame Charlotte Baey maître de conférence à l'Université de Lille, notre enseignante de statistique computationnelle.

Nous remercions également toute l'équipe pédagogique de l'Université de Lille et les intervenants professionnels responsables de notre formation, pour avoir assuré la partie théorique de celle-ci.

# Introduction

Étant étudiants en master Mathématiques et Application dans la section Ingénierie Statistique Numérique, l'étude d'évolution de cluster et de simulation grâce à des algorithmes MCMC [1] nous semble assez logique. Le but de notre TER est de réaliser une classification non supervisée d'un ensemble de points en utilisant le modèle de Potts. Nous avons donc procédé en deux étapes : une première étant l'implémentation d'un programme informatique en C++ et en Python. Ces deux langages sont très utiles pour la modélisation qui se fait en Python et pour l'algorithmique qui se fait en C++. La deuxième étant l'étude des algorithmes de Metropolis-Hasting et Swendsen-Wang, leurs convergences et leurs utilités de ces algorithmes dans un tel modèle. Dans ce mémoire, nous allons ainsi expliquer dans un premier temps le travail que nous avons fourni en programmation, ensuite, à l'aide de nos connaissances statistiques, l'utilité des algorithmes de Metropolis-Hasting et Swendsen-Wang.
L'argumentation sera basée sur notre cours de statistique computationnelle et de nos recherches référencées en fin de document.

Pour commencer, qu'est-ce que le modèle de Potts ?
D'après l'encyclopédie libre Wikipedia, le modèle cellulaire de Potts [2] est un modèle informatique de cellules et de tissus. Il est aussi connu sous le nom de modèle Glazier-

1. Markov chain Monte Carlo
2. CPM

Graner-Hogeweg. Le CPM est composé d'une grille rectangulaire (Figure 1) où chaque pixel peut appartenir soit à une cellule, soit au milieu. Une cellule se compose donc d'un ensemble de pixels qui partagent le même état. L'algorithme qui met à jour les états de chaque pixel minimise cette énergie en suivant un algorithme de type MCMC. Ici, nous utiliserons donc les algorithmes de Metropolis-Hasting et Swendsen-Wang.

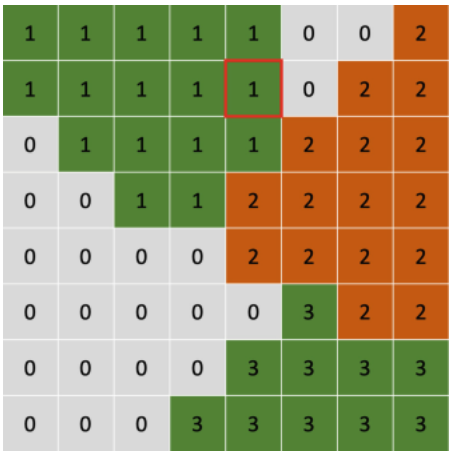


Figure 1 – Représentation d'une grille de modèle celullaire de Potts à 4 états

Nous avons commencé par la programmation et donc l'implémentation de plusieurs classes et de l'affichage d'une cellule. Puis arrivés à la mise en place des algorithmes, nous avons commencé la théorie des algorithmes et étudié comment les appliquer à cette fonction cible. La partie programmation nous a pris un certain temps, dû à l'utilisation de nouveaux objets pour nous. L'algorithme de Metropolis-Hasting a été étudié lors du semestre dans une UE de statistique computationnelle, c'est aussi pour cela que nous avons laissé la théorie en second plan. Dans les trois chapitres suivants, nous allons vous expliquer le procédé et les démarches que nous avons suivies afin d'implémenter Metropolis-Hasting et Swendsen-Wang.

# Chapitre 1

# Théorie

## 1.1 Clustering

Le principe du clustering est d'identifier des groupes (cluster) dans un jeu de données et de classer les observations de ce jeu de données dans ces groupes selon des critères que nous verrons par la suite.
L'algorithme de clustering le plus répandu est celui des K-moyennes (K-means). Soit $x_i \in \mathbb{R}^d, i = 1, \ldots, n$, on souhaite répartir ces n points en q clusters. Soit $z_{ki} = 1$ si $x_i$ appartient au k-ème cluster, 0 sinon.

L'algorithme des K-moyennes consiste à :

(i) Trouver les centres des clusters que l'on note $\{m_k\}_{k=1}^q$ ainsi que les membres $z_{ki}$ de ces clusters. Ces membres sont obtenus en minimisant la quantité :

$$\frac{1}{n} \sum_{k=1}^{q} \sum_{i=1}^{n} z_{ki}(x_i - m_k)^t(x_i - m_k)$$

ce qui revient à maximiser :

$$\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} \langle x_i, x_j \rangle \sum_{k=1}^{q} \frac{z_{ki} z_{kj}}{n_k}$$

où $n_k$ est le nombre de points du k-ème cluster, $k = 1, \ldots, q$ et $\langle \cdot, \cdot \rangle$ le produit scalaire sûr $\mathbb{R}^d$.

(ii) Les poids $w(i, j, \{z_{ki}\})$ qui sont égaux à $\frac{1}{n_k}$ si $x_i$ et $x_j$ sont dans le même cluster k, zéro sinon.

Finalement, (i) et (ii) nous donnent la quantité suivante à maximiser :

$$\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} w(i, j, \{z_{ki}\}) \langle x_i, x_j \rangle \tag{1}$$

L'équation (1) peut être généralisée en modifiant le produit scalaire par une mesure de similarité $s(x_i, x_j)$ qui sera à définir. On obtient ainsi :

$$\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} w(i, j, \{z_{ki}\}) s(x_i, x_j) \tag{2}$$

Enfin, dans ce TER, nous étudions un système d'états et non un système de clusters, c'est pourquoi on peut remplacer la quantité $w(i, j, \{z_{ki}\})$ par $\delta_{ij}$ qui vaut 1 si deux sommets voisins sont dans le même état, zéro sinon. On maximise alors cette quantité :

$$\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} s(x_i, x_j) \delta_{ij} \tag{3}$$

D'après nos recherches, il existe une fonction $H$ qui mesure la qualité de la classification, on pourra ainsi trouver les membres d'un état souhaité en minimisant H. En appliquant cette fonction à notre problème, maximiser (3) revient à minimiser :

$$H(z_{ki}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}(1-\delta_{ij})s(x_i, x_j) \tag{4}$$

De plus, cette fonction $H$, lorsque $s(\cdot,\cdot)$ est une fonction positive et symétrique, correspond à l'hamiltonien d'un modèle de Potts qui décrit le phénomène de ferromagnétisme (modèle d'Ising). Plus généralement, le modèle de Potts est un modèle probabiliste basé sur un système de particules et leurs interactions sont données par une mesure de similarité. La distribution du système dépend de la température $T$. Le modèle se résume alors à l'équation ci-dessous :

$$p_T(z) \propto \exp\left\{ -\frac{1}{T}\sum_{i=1}^{n}\sum_{j=1}^{n}(1-\delta_{ij})s(x_i, x_j) \right\} \tag{5}$$

l'idée du modèle est que plus la similarité entre deux points est grande, plus cela sera pénalisant de les avoir dans des états différents.

## 1.2 Modélisation

### 1.2.1 Première approche

Une première approche assez naïve serait de sélectionner un pixel sur la grille aléatoirement et modifier son état.

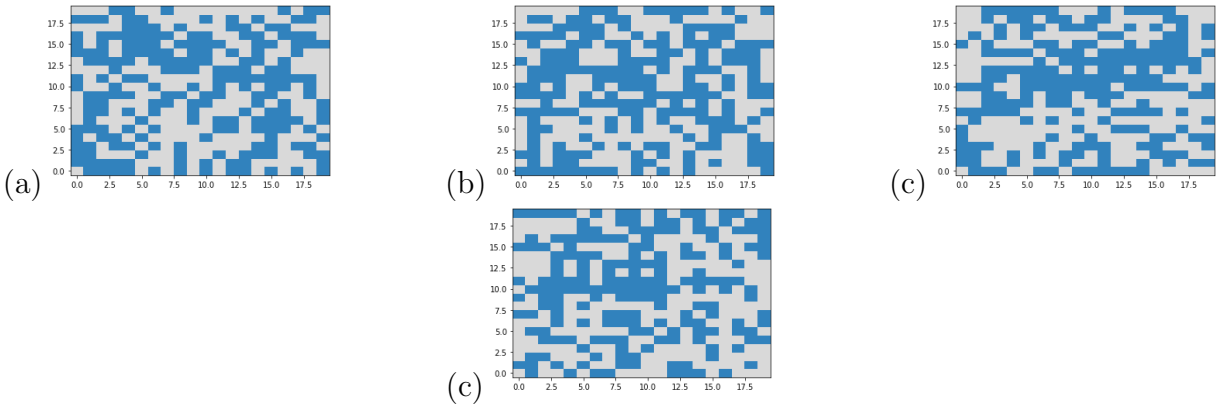


Figure 1.1 – (a) Initialisation (b) 250 itérations (c) 600 itérations (d) 1000 itérations

On remarque que cette approche assez simpliste ne nous permet pas de tirer une quelconque conclusion. Aucun cluster ne semble se former, les points semblent juste se réarranger différemment sans suivre une loi spécifique.

### 1.2.2 Metropolis-Hastings (cas général)

Partant d'une loi cible de densité $f$ , on commence par choisir une densité conditionnelle $q(\cdot|x)$, dont on sait facilement simuler des réalisations aléatoires, ou symétrique (c'est-à-dire telle que $q(x|y) = q(y|x)$). Puis, chaque itération de l'algorithme consiste alors à simuler, à partir de la configuration courante de la chaine $X_n$, un candidat qui sera accepté comme la nouvelle configuration de la chaîne avec une certaine probabilité. Si le candidat est rejeté, la chaîne reste dans le même état.

L'algorithme est le suivant :

**Algorithm 1** Algorithme de Metropolis-Hasting

**Initialisation :** on initialise les états de la grille avec $X_0$

**Pour** $n = 1, ..., N$ **faire**

On génère un candidat $Y_n \sim q(\cdot|X_{n-1})$

on pose

$$X_n = \begin{cases} Y_n & \text{avec une probabilité } \alpha(X_{n-1}, Y_n) \\ X_{n-1} & \text{avec une probabilité } 1 - \alpha(X_{n-1}, Y_n) \end{cases}$$

avec

$$\alpha(x, y) = min\left(1, \frac{f(y)}{f(x)} \frac{q(x|y)}{q(x|y)}\right)$$

On utilise la loi instrumentale $q(\cdot|x)$ pour générer des réalisations de la loi $f$ donc, plus le rapport $f/q$ est faible pour le candidat par rapport à l'état courant de la chaine, plus la probabilité de le rejeter est forte. Intuitivement, la probabilité d'acceptation $\alpha(x, y)$ permet de faire un compromis entre les deux conditions suivantes : d'une part, on souhaite que l'algorithme se dirige vers des régions de plus forte probabilité sous $f$ , ce qui est contrôlé par le rapport $f(y)/f(x)$ (plus celui-ci est haut, plus on accepte le candidat), et d'autre part, on souhaite éviter que l'algorithme ne reste trop longtemps dans une région spécifique de trop forte probabilité sous $q$, ce qui est contrôlé par le rapport $q(x|y)/q(x|y)$.

### 1.2.3 Méthode naïve

Il convient dans cette partie de définir plusieurs éléments. Notamment la loi cible et la loi instrumentale, afin d'adapter l'algorithme à notre problème.

— **Choix de la loi cible**

Il est évident que notre loi cible ici sera issue de l'équation (5) qui nous donne la probabilité que notre système soit dans un certain état donné. C'est la distribution de notre système, c'est cette loi qui régit sa configuration.

— **Choix de la loi instrumentale**

Notre loi instrumentale ici sera une loi uniforme, dans le même esprit que dans la partie *1.2.1*, ici l'idée est de tirer aléatoirement un point de la grille à l'aide d'une réalisation uniforme et de changer son état aléatoirement à l'aide d'une autre réalisation d'une loi uniforme sur l'espace des états. Ce qui est nécessaire de comprendre ici, contrairement à la partie précédente, ce changement ne s'opèrera pas de manière systématique, il sera validé ou non via la probabilité d'acceptation $\alpha$.

Les étapes de l'algorithme dans notre cas seront les suivantes :

1. Initialisation : Tout d'abord, nous commençons par une configuration initiale des sommets, où chaque sommet est attribué à l'un des E états possibles de manière aléatoire.
2. Propagation : Pour chaque itération de l'algorithme, nous sélectionnons un sommet au hasard dans la configuration actuelle selon une loi **uniforme sur** $[\mathbf{0}, \mathbf{n}]$.
3. Proposition de modification : Nous proposons une modification de la valeur de l'état du sommet sélectionné à l'étape précédente, généralement en choisissant une nouvelle valeur d'état au hasard parmi les E états possibles selon une loi **uniforme sur** $[\mathbf{0}, \mathbf{E}]$.
4. Calcul des probabilités : Nous calculons la probabilité de transition entre la configuration actuelle et la nouvelle configuration proposée. Cette probabilité est basée sur le calcul de la fonction évaluée entre la configuration actuelle et la configuration proposée.

5. Acceptation ou rejet de la modification : nous décidons d'accepter ou de rejeter la proposition de modification selon une règle de décision probabiliste. Pour cela, nous comparons la probabilité de transition calculée à l'étape précédente avec un nombre aléatoire entre 0 et 1. Si la probabilité de transition est plus grande que le nombre aléatoire, nous acceptons la modification et passons à la nouvelle configuration. Sinon, nous rejetons la modification et restons dans la configuration actuelle.

6. Répétition : Les étapes 2 à 5 sont répétées pour un certain nombre d'itérations ou jusqu'à ce que le système atteigne un état d'équilibre.

Soit $Z \in \mathbb{R}^n$ où n est notre nombre de points, E le nombre d'états :

On choisit la position du point à changer $\mathbf{U_1} \sim \mathcal{U}([\mathbf{0}, \mathbf{n}])$.

On choisit l'état qu'il va prendre $\mathbf{U_2} \sim \mathcal{U}([\mathbf{0}, \mathbf{E}])$.

On a donc l'algorithme suivant :

**Algorithm 2** méthode naïve de Metropolis-Hastings

**Initialisation :** On initialise les états de la grille avec $Z_0$

**Pour** $k = 1, ..., N$ **faire**

On a bien $Y_k|Z_{k-1} \sim \mathcal{U}(\frac{1}{nE}\mathbf{1}_{\{i\in[0,n]\cap Z_{k-1,i}\in[0,E]\}}) = q(\cdot|Z_{k-1})$ où $Z_{k-1,i}$ correspond à la ième composante du vecteur $Z_{k-1}$.

$$Z_k = \begin{cases} Y_k & \text{avec une probabilité } \alpha(Z_{k-1}, Y_k) \\ Z_{k-1} & \text{avec une probabilité } 1 - \alpha(Z_{k-1}, Y_k) \end{cases}$$

avec

$$\alpha(x, y) = min\left(1, \frac{p_T(y)}{p_T(x)}\right)$$

*Remarque : le rapport* $\frac{q(x|y)}{q(y|x)} = 1$ *car la loi uniforme est symétrique.*

### 1.2.4 Swendsen-Wang

Voici les étapes principales de l'algorithme de Swendsen-Wang pour le modèle de Potts :

1. Initialisation : Tout d'abord, nous commençons par une configuration initiale des états des sommets, où chaque sommet est attribué à l'un des E états possibles de manière aléatoire.

2. Construction du graphe : Ensuite, nous construisons un graphe basé sur les intéractions entre les sommets. Chaque sommet est représenté par un point du graphe, et les arêtes du graphe relient les sommets voisins qui sont dans le même état. Ainsi, si deux sommets voisins ont la même valeur, ils sont connectés par une arête.

3. Génération des clusters : Nous utilisons une procédure de propagation sur le graphe pour générer des clusters de sommets connectés. Dans le cas de Swendsen-Wang, cette procédure utilise une probabilité de liaison entre deux sommets voisins, qui est donnée par p = 1 - exp($\beta$), où $\beta \in \mathbb{R}^+$ est le paramètre d'inverse de température $T$. Chaque arête du graphe est activée indépendamment avec une probabilité p, ce qui conduit à la formation de clusters des sommets.

4. Attribution de nouvelles valeurs : Après avoir généré les clusters, chaque cluster est attribué à une nouvelle valeur d'état choisie uniformément parmi les E états possibles. Ainsi, tous les sommets appartenant à un même cluster prennent la même nouvelle valeur d'état.

5. Répétition : Les étapes 3 et 4 sont répétées pour un certain nombre d'itérations ou jusqu'à ce que le système atteigne un état d'équilibre thermodynamique.

**Algorithm 3** Fonction generate_clusters

**Entrées :** Grille Z, $\beta$

**Sortie :** Clusters

**Fonction** *generate_clusters* **:**

Clusters ← la matrice avec les réparition des clusters dans le graphe

**Pour chaque** *sommet*

On le connecte à son voisin avec probabilité $1 - \exp(-\beta)$.

On assigne un numéro de cluster à chaque composantes connexes du graphe.

**retourner** Clusters

**Algorithm 4** Fonction update_configuration

**Entrées :** Grille Z, Clusters

**Sortie :** Nouvelle Grille

**Fonction** *update_configuration* **:**

Clusters ← la matrice avec les réparition des clusters dans le graphe

**Pour chaque** *cluster*

On change l'état de tous les points du cluster de manière aléatoire parmi tous les états possibles.

**retourner** Nouvelle Grille

**Algorithm 5** Fonction swendsen_wang

**Entrées :** Grille Initial Z, N nombre d'itérations

**Sortie :** Grille Final

**Fonction** *swendsen_wang* **:**

**Pour** $1, \ldots, N$

Clusters←*generate_clusters*(Z,$\beta$)

Z←*update_configuration*(Z,Clusters)

**retourner** Z

# Chapitre 2

# Programmation

## 2.1 Préambule

Pour implémenter le modèle de Potts, nous allons suivre le schéma que nous avons mis en place afin de pouvoir générer des modèles le plus simplement possible.
Nous avons donc divisé les taches selon ce schéma :

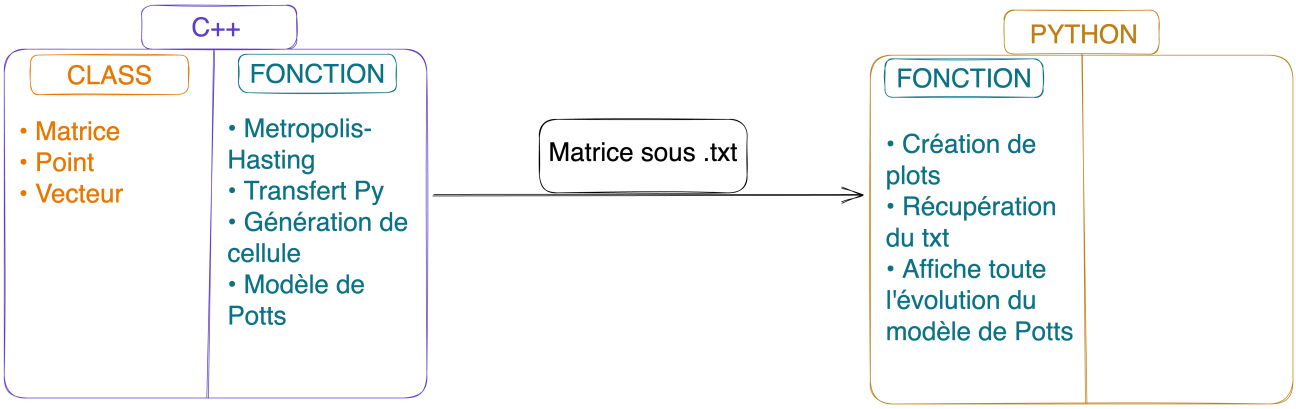


Figure 2.1 – Schéma de programmation

On a, selon ce schéma, deux langages de programmation différents, le C++ et le Python.

Par la suite, les fonctions seront toutes trouvables sur notre GitHub à l'adresse : `https://github.com/gtk-gh/ter_potts_model` .

## 2.2 Prérequis

### 2.2.1 Préambule en C++

Nous allons donc implémenter 3 classes différentes :

**point** Une classe de point avec 2 coordonnées dont chacun ont un état, dans cette classe, nous aurons des fonctions qui modifieront les propriétés des points.

**matrice** Cette classe va nous permettre d'avoir une matrice de point.

**vecteur-template** Ce n'est pas vraiment une classe, car ici, nous allons utiliser la classe déjà présente en C++, vector [1]. Nous allons ainsi ajouter des fonctions propres à cette classe pour pouvoir l'utiliser avec les points.

On pourra donc grâce à ces classes implémenter le modèle de Potts.
Nous avons choisi de faire des classes et donc de se rajouter une charge de travail au début, car cela semblait plus approprié et nous permettait aussi de parfaire notre niveau de programmation dans le langage C++.

### 2.2.2 Détails sur les classes

**Classe point :**

Fichiers `point.h` et `point.cpp` sur le Github.

Pour la classe point, nous avons créé un objet avec trois principaux attributs : ses deux coordonnées et son état.

Nous ajoutons des fonctions qui permettent de modifier les attributs pour plus tard. La

1. Documentation officielle de la librairie sur : `https://cplusplus.com/reference/vector/vector/`

classe est surtout utile pour la modélisation et l'affichage sur un graphique qu'on verra après.

Nous y ajoutons aussi les fonctions telles que la comparaison d'états entre deux sommets par exemple, etc. Cette classe n'a donc pas été trop compliquée à programmer.

**Classe vector :**

Fichiers `vector_double.h` sur le Github.

Pour la classe vector, nous avons donc utilisé la librairie vector, comme dit plus tôt, que nous avons adapté pour prendre en charge la classe point et donc avoir des vecteurs de points.


Figure 2.2 – Exemple d'un vecteur de point

**Classe matrice :**

Fichiers `matrice.h` et `matrice.cpp` sur le Github.

Le but de cette classe, comme son nom l'indique, est de créer une classe qui représente une matrice où chaque élément est un sommet du modèle. On pourra, à l'aide d'un quadrillage, modéliser le schéma et son évolution.

Le principal problème que nous avons rencontré est la gestion d'adresse mémoire entre les points et la matrice.

Nous avons pris l'initiative d'utiliser la librairie shared_ptr [2], cette librairie permet d'avoir des pointeurs intelligents qui gèrent eux-mêmes leurs constructions et leurs destructions.

2. Documentation officielle de la librairie sur : `https://en.cppreference.com/w/cpp/memory/shared\_ptr`

On a donc une classe avec des attributs et les points (sommets) sont stockés grâce à un vecteur de pointeur intelligent. Grâce à la classe matrice, on a donc accès à tous les points, tous les états possibles du modèle et aux dimensions du modèle.

## 2.3 Implémentation des algorithmes

Maintenant que nous avons tous les outils en main, nous pouvons donc implémenter les deux algorithmes : Metropolis-Hasting et Swendsen-Wang.

### 2.3.1 Metropolis-Hasting en C++

Avant de commencer à coder Metropolis-Hasting, nous allons d'abord nous occuper de la fonction cible et la fonction instrumentale. Les fonctions seront toutes dans les fichiers `metropolis_hasting.h` et `metropolis_hasting.cpp`.

#### Fonction instrumentale

Comme expliqué dans le chapitre 1, nous devons créer une fonction qui choisit un sommet uniformément dans notre liste de sommets et qui change uniformément son état parmi tous les états possibles.
Pour cela, nous allons donc créer notre distribution uniforme grâce à la fonction rand() de la librairie cstdlib [1]. Le seed (graine) de l'aléatoire est indexé sur le nombre de secondes écoulées depuis le 1er janvier 1970 au moment de l'exécution du script. On a donc une génération complètement aléatoire et donc la fonction rand() nous renvoie un entier compris entre 0 et RAND_MAX (une constante qui vaut le plus grand entier en 32 bits donc : 2147483647) nous garderons juste le reste de la division euclidienne par le nombre de sommets ou le nombre d'états.

1. Documentation officielle de la librairie sur : `https://cplusplus.com/reference/cstdlib/`

On a donc notre fonction instrumentale qui suit bien une double loi uniforme comme expliqué plus tôt.

**Fonction cible**

Rappelons la fonction cible :

$$p_T(z) \propto \exp\left\{ -\frac{1}{T} \sum_{i=1}^{n} \sum_{j=1}^{n} (1 - \delta_{ij}) s(x_i, x_j) \right\} \tag{5}$$

Nous devons donc implémenter cela en C++.

Traduisons les termes de l'équation avec nos données en C++ : $z$ représente donc notre matrice d'états (une configuration), pour cela transformons notre matrice en vecteur où les coefficients sont les états des points et les coefficients sont rangés lignes par lignes, par exemple pour la matrice : $\left(\begin{smallmatrix} 1 & 2 \\ 0 & 1 \end{smallmatrix}\right)$ nous avons le vecteur $z = (1, 2, 0, 1)$, où les coefficients sont donc des états.

Nous avons donc déterminé $z$. $T$ est la température associée au modèle et donc il nous reste $\delta_{ij}$ et $s(x_i, x_j)$. Pour $\delta_{ij}$ on a juste à comparer si les coefficients sont les mêmes, car $\delta_{ij} = 1$ si l'état de $x_i$ est égal à l'état de $x_j$ et 0 sinon.

La fonction $s(x_i, x_j)$ dépend des similarités. Nous allons donc devoir étudier les similarités entre les points et pour cela, nous allons définir les similarités dans notre modèle.

Les similarités sont par définition des poids qui valent 1 ou 0 si des voisins sont dans le même cluster ou pas. Quand nous parlons de voisins, nous parlons de sommets présents proches de notre sommet étudié, c'est-à-dire par exemple si nous nous trouvions dans une grille, les points au-dessus, en dessous, à gauche ou à droite. Donc pour chaque sommet $x_i$ nous allons déterminer seulement les similarités avec les points collés à lui.

Il est donc important ici de différencier les clusters dans Metropolis-Hasting de ceux de

Swendsen-Wang que nous verrons plus tard. Ici les clusters ne vont pas évoluer, ils sont fixés au début de la simulation et ne changeront pas pendant toutes les itérations.
Nous avons donc défini toutes les fonctions et nous pouvons mettre en place l'algorithme de Metropolis-Hasting.

**Metropolis-Hasting**

Dans le chapitre 1 nous avons vu la version théorique et la version appliquée de l'algorithme de Metropolis-Hasting. Nous pouvons donc grâce aux fonctions précédentes l'implémenter en C++. L'algorithme nous renvoie donc un échantillon de matrice que nous pouvons utiliser pour modéliser le modèle.

### 2.3.2 Swendsen-Wang en C++

**Fonction generate_clusters :**

Cette fonction fonctionne avec 3 matrices, la matrice initiale *config*, une matrice *clusters* et une matrice *visited* pour savoir si un sommet a été visité ou non lors de la procédure. L'idée ici est de parcourir la matrice initiale, une fois qu'un sommet est visité, on l'indique par un 1 dans la matrice *visited*. À partir de ce point, on regarde les composantes connexes de ce point parmi ses voisins et récursivement, on regarde si on peut étendre le cluster à partir de ces composantes connexes. Si cela n'est pas possible, alors, on continue cette procédure en itérant le numéro du cluster. La fonction va nous retourner une matrice avec les différents sommets et le cluster auquel ils appartiennent.

**Fonction update_configuration :**

Cette fonction prend en paramètre une configuration avec les clusters associés à cette configuration, la fonction parcours la matrice et change l'état de tous les sommets d'un même cluster parmi tous les états possible.

### 2.3.3 Modélisation graphique en Python

Toute la modélisation du modèle est réalisée grâce aux bibliothèques numpy [2] et matplotlib [3].

On récupère les échantillons de matrice via un fichier texte écrit grâce à la fonction save_matr2 en C++ puis on affiche ces matrices grâce à la fonction read_opti en Python.

2. Documentation officielle de la librairie sur : `https://numpy.org/`
3. Documentation officielle de la librairie sur : `https://matplotlib.org/`

# Chapitre 3

# Simulation

Nous avons deux algorithmes qui ont le même but, mais qui ne le font pas de la même manière, nous allons donc, dans ce chapitre, étudier les cas particuliers des deux algorithmes et comparer leurs vitesses de convergence.

## 3.1 Effet de la température

### 3.1.1 Cas de Metropolis-Hasting

À chaque itération, Metropolis-Hasting teste alpha qui dépend de la fonction cible, dans la fonction cible le "poids" de la température $T$ peut en fonction de sa grandeur impacter beaucoup plus que $H(z_{ki})$ (équation 4 chapitre 1), les travaux de Mr. H. DUMINIL-COPIN (annexe) sur le sujet nous prouvent ces dires (attention dans le cas de Swendsen-Wang, on parle de $\beta$ qui est l'inverse de la température, ici, on parle donc bien de la température $T = \frac{1}{\beta}$).

Nous allons donc partir d'une même base et nous allons varier la température, une sous-critique, une critique et une supercritique.

Nous partons d'un modèle spécial pour mettre en valeur les similarités lors des cas où

elles sont impactantes. Dans notre modèle, les points présents en bas à gauche sont de même similarité et nous leur attribuons le même état de départ, sachant qu'il y a 3 états possibles : $(0, 1, 2)$.

Voici un exemple pour un modèle 20 par 20.

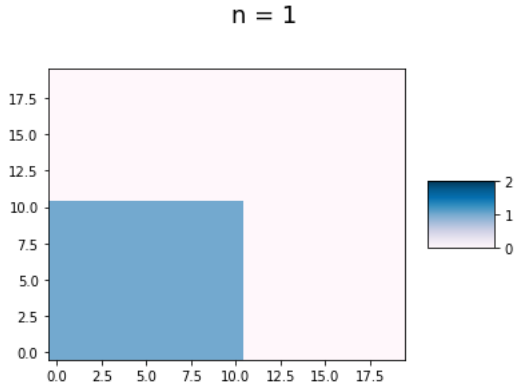


Figure 3.1 – Exemple d'un candidat avec des similarités en bas à gauche

Nous allons donc faire 1000 itérations avec trois températures différentes : $T_1 = 2$, $T_2 = 7$ et $T_3 = 50$, voici les résultats :

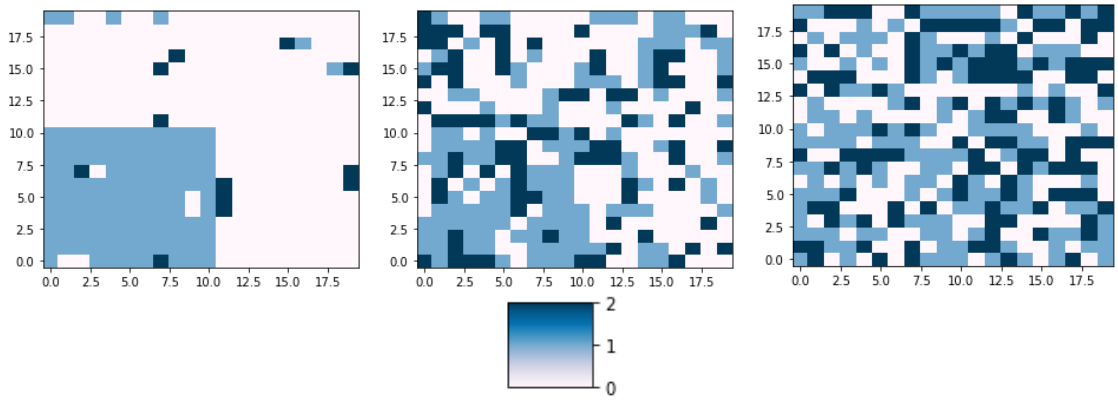


Figure 3.2 – Évolution en fonction de la température à la 1000 ème itération. De gauche à droite $T_1$ $T_2$ $T_3$.

On voit que pour une température basse $T_1$ le modèle évolue très peu, le bloc en bas à gauche de voisin avec la même similarité reste présent. Nous avons un taux d'acceptation d'environ 0.3 (nombre de fois où $Y_n$ est accepté divisé par $n$) ce qui explique ce résultat.

On appellera cette température : sous-critique.

Pour $T_2$ on voit que des groupes de voisins de même état se forment, le taux d'acceptation est d'environ 0.6 ce qui est très bon pour Metropolis-Hasting. On appellera cette température : critique.

Pour $T_3$ tout devient chaotique, la température l'emporte sur $H(z_{ki})$, le taux d'acceptation est d'environ 0.98 proche de 1, $Y_n$ est tout le temps accepté la température est dite supercritique.

Nous allons maintenant optimiser la fonction cible pour faire beaucoup moins de boucle et donc utiliser des modèles plus grand et avoir plus d'itérations.

En effet, on sait que $s(x_i, x_j) = 0$ quand les sommets ne sont pas voisins, on peut alors remplacer la somme des j allant de 1 à n par la somme des j sur les sommets voisins de $x_i$. Soit l'ensemble $v(x_i) = \{j \in \mathbb{N} \mid x_j \text{ est un voisin de } x_i\}$, la fonction cible devient :

$$p_T(z) \propto \exp\left\{ -\frac{1}{T} \sum_{i=1}^{n} \sum_{j \in v(x_i)} (1 - \delta_{ij}) s(x_i, x_j) \right\} \tag{6}$$

On a donc une nouvelle fonction cible et maintenant, on peut travailler avec des plans plus grands et plus d'itérations. On ne voit pas vraiment de changement sur l'ordre de grandeur des températures.

Le modèle 30 par 30.

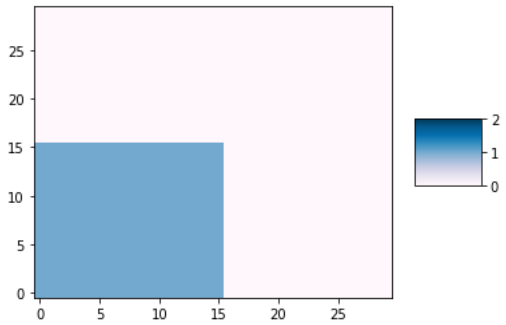

Figure 3.3 – Exemple d'un candidat avec des similarités en bas à gauche

Nous allons faire 2500 itérations avec trois températures différentes : $T_1 = 2$, $T_2 = 7$ et $T_3 = 50$, voici les résultats :

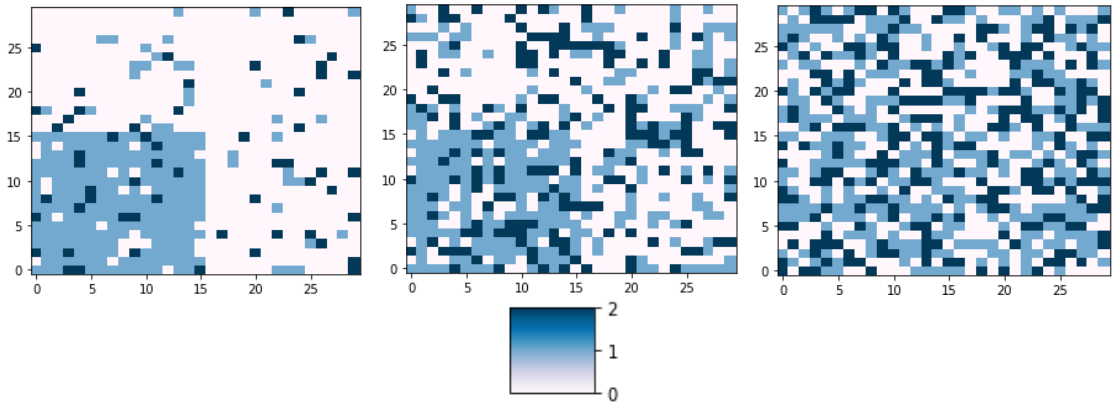

Figure 3.4 – Évolution en fonction de la température à la 2500 ème itération. De gauche à droite $T_1$ $T_2$ $T_3$.

On a presque les mêmes résultats, le modèle se stabilise donc quel que soit la taille du modèle et le nombre d'itérations.

On peut donc essayer des modèles de départ différents et voir leurs résultats.

Cas d'un quadrillage de similarités.

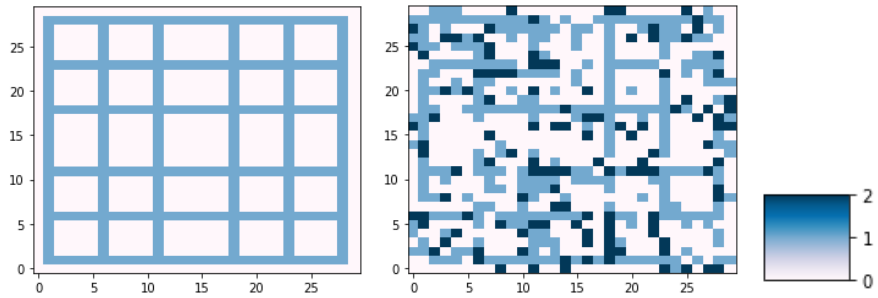


Figure 3.5 – Évolution en fonction du modèle (quadrillage) à la 2500 ème itération. $T = 7$, taux d'acceptation $= 0.744$.

Cas d'une génération aléatoire, ici tous les points sont dans un état et une similarité choisis aléatoirement.

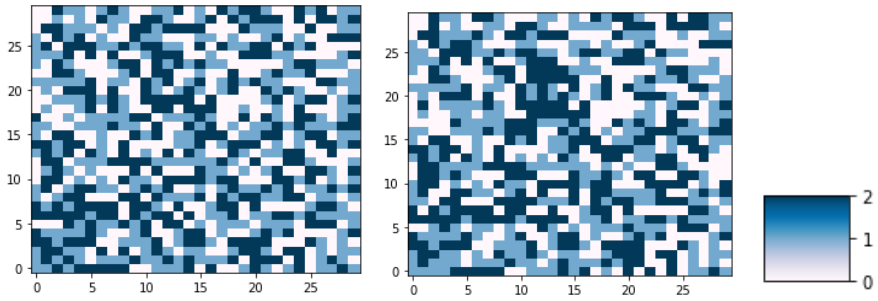


Figure 3.6 – Évolution en fonction du modèle (chaos) à la 2500 ème itération. $T = 1$, taux d'acceptation $= 0.516$.

Ici, il faut fortement baisser la température pour arriver à un état critique.
Comme vu plus tôt dans le chapitre 1, le choix d'un bon candidat de départ joue sur le résultat voulu et attendu.

### 3.1.2 Cas de Swendsen-Wang

Dans cette partie et, plus généralement, pour notre algorithme, nous parlerons de paramètre $\beta$ qui est tout simplement l'inverse du paramètre de température $T$.

Partons d'une configuration aléatoire de la grille qui sera dans chaque cas similaire à la figure suivante :

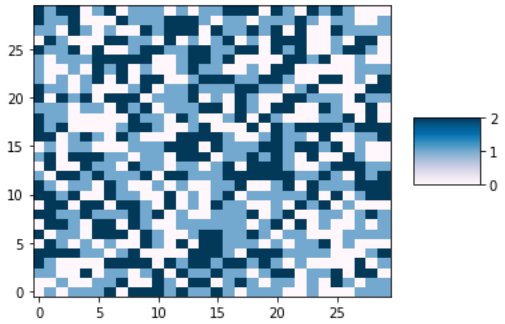


Figure 3.7 – caption

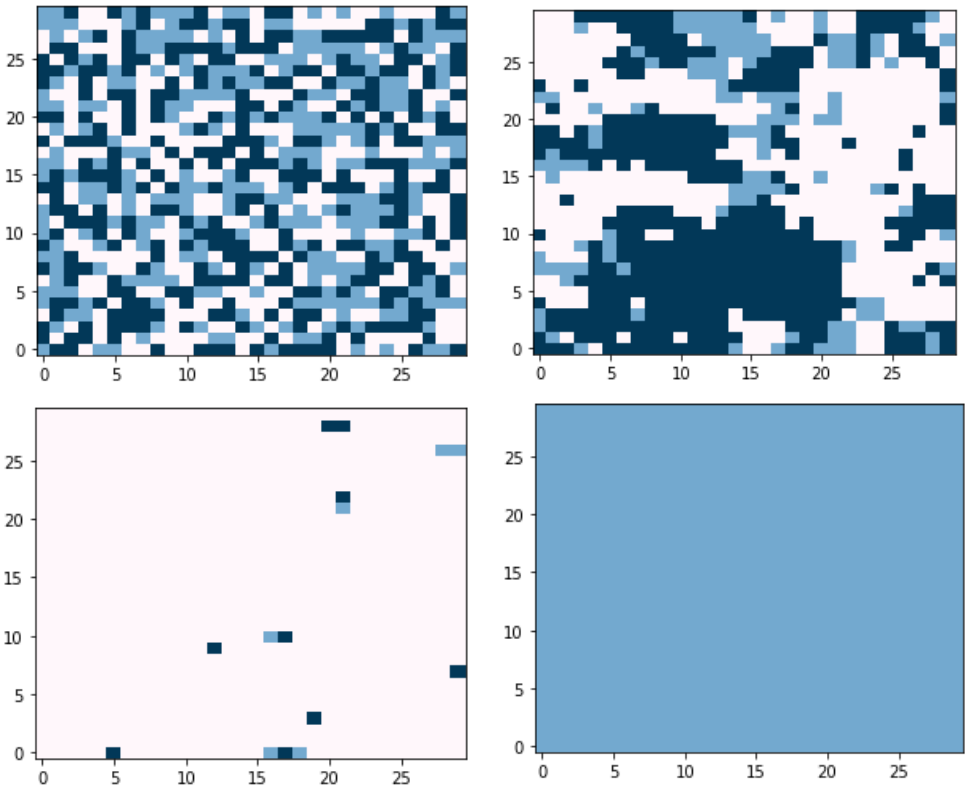


Figure 3.8 – État de la grille après 1000 itérations pour des valeurs de $\beta$ valant 0.1, 1, 1.5 et 3 respectivement.

On remarque ici que pour des petites valeurs de $\beta$ (donc une température élevée) les états que prennent les points sur la grille semblent équirépartis. Une température élevée

excite les sommets, ils changent tous constamment d'état même après de nombreuses itérations, on retombe toujours sur une configuration similaire à la configuration initiale. En augmentant la valeur de $\beta$, au fil des itérations, on remarque que des clusters se forment entre les sommets, la configuration change, mais ce sont très souvent les mêmes sommets qui restent connexes ensemble. De plus, d'après nos simulations, nous avons remarqué que pour $\beta \geq 3$ nous aboutissons toujours à la même configuration finale où tous les sommets de la grille sont dans le même état. Cela s'explique par une température très faible, une température trop faible vient figer notre grille très rapidement. Prenons maintenant pour paramètre $\beta = 1$, le paramètre pour lequel nos sommets du graphe se regroupent entre eux comme sur la figure (3.8).

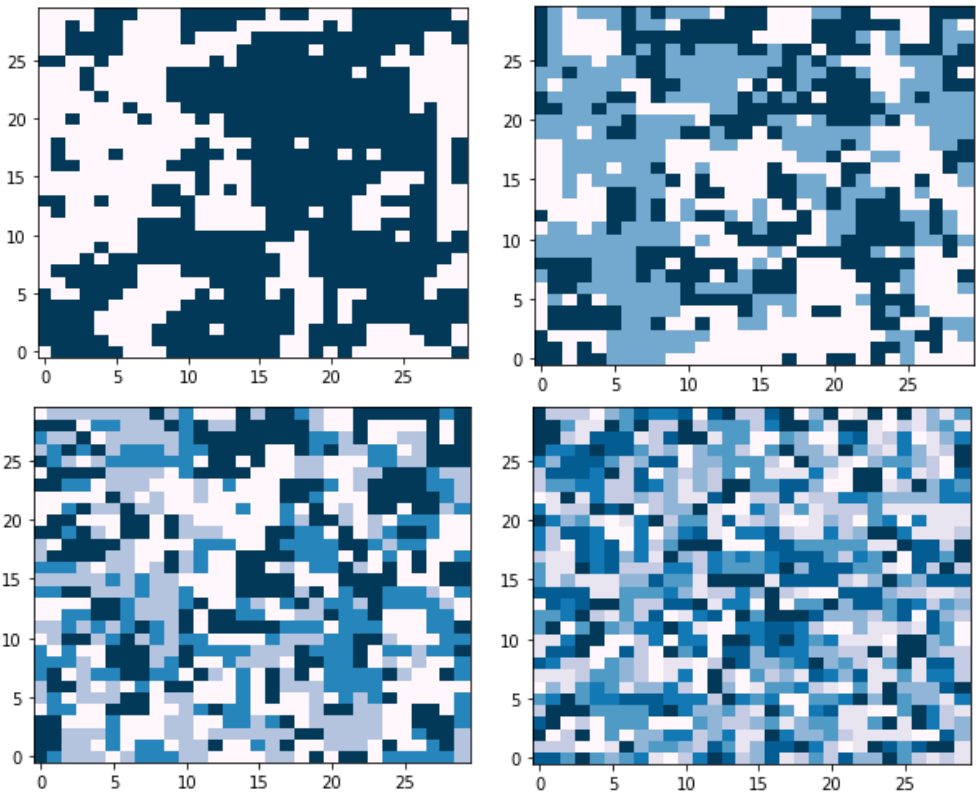

Figure 3.9 – État final de la grille après 1000 itérations, $\beta = 1$ pour 2,3,4 et 8 états respectivement.

Ce qu'on peut remarquer ici, c'est que plus notre nombre d'états augmente, plus, il est

difficile de regrouper les sommets ayant le même état entre eux.

Intéressons-nous maintenant au cas où dans notre configuration initiale il y a des similarités entre les sommets, prenons par exemple le cas de la figure (3.13) avec des similarités entre les sommets dans le coin inférieur gauche.

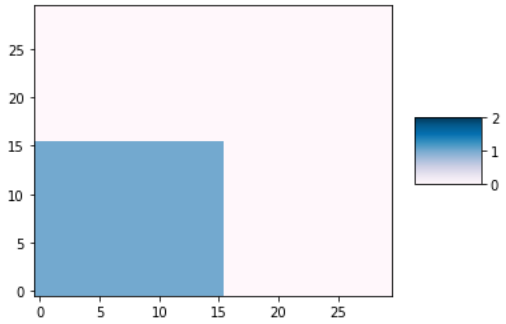


Figure 3.10 – Configuration initiale avec similarités

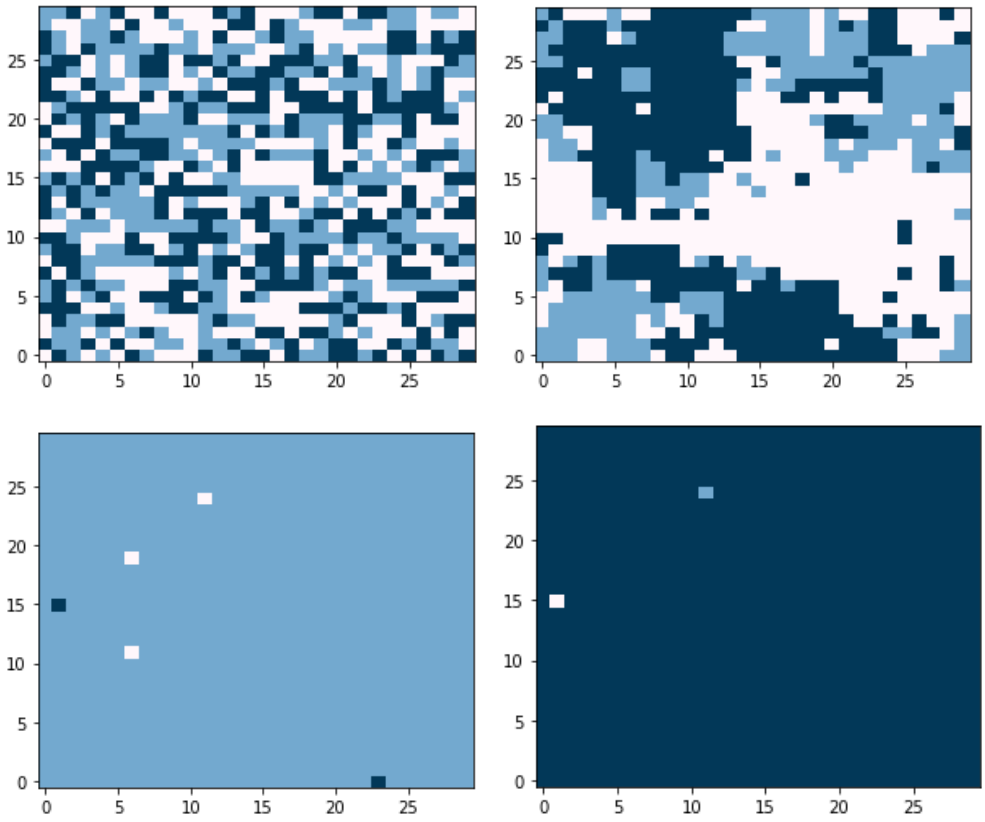


Figure 3.11 – État final de la grille après 1000 itérations, $\beta$ valant 0.1,1,1.5 et 3 respectivement.

Il est clair ici que la configuration initiale ne joue aucun rôle, les similarités de départ disparaissent dès les toutes premières itérations. Ceci s'explique par l'absence de paramètre qui régit les similarités sur le voisinage d'un sommet contrairement à Metropolis-Hastings.

## 3.2 Rapidité et convergence

### Rapidité d'exécution

Dû à son écriture plus simple, Swendsen-Wang se compile beaucoup plus vite que Metropolis-Hasting, comparons l'exécution des deux algorithmes pour 1000 itérations en partant de la même matrice :

```
Pour 1000 itérations Métropolis-Hasting s'est exécuté en 31 seconde(s).
Pour 1000 itérations Swendsen-Wang s'est exécuté en 8 seconde(s).
```

Figure 3.12 – Différence de temps d'exécution entre les deux algorithmes en secondes

### Vitesse de convergence

Prenons ici une matrice commune et une température équivalente pour les deux algorithmes. Nous allons prendre $T_{MH} = 7$ et $\beta_{SW} = 0.9$, et comparons après 500 itérations l'état des deux modèles :

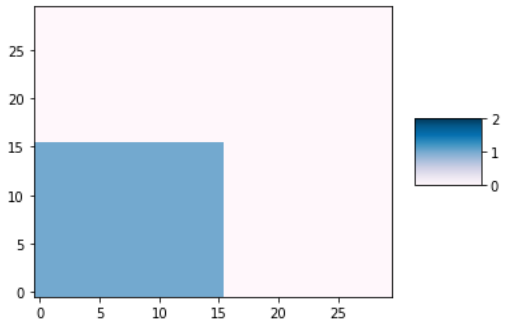


Figure 3.13 – Matrice candidate

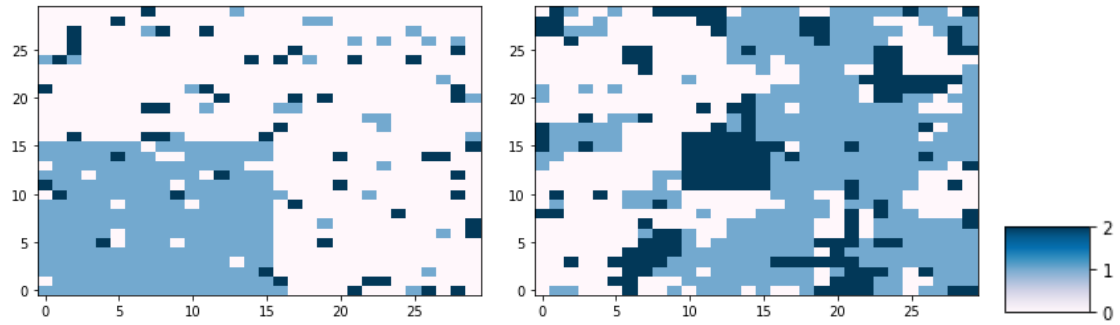


Figure 3.14 – Comparaison des deux algorithmes à la 500 ème itération. À gauche Metropolis-Hasing, à droite Swendsen-Wang

On voit bien que le cas de Metropolis-Hasting n'est qu'au debut de son évolution alors que Swendsen-Wang est à un stade déjà avancé. On remarque aussi que Swendsen-Wang a perdu toutes les propriétés dues aux similarités en bas à gauche, contrairement à Metropolis-Hasting qui les conserve tout au long de son évolution.

**Conclusion**

Swendsen-Wang est un algorithme plus rapide et plus efficace que Metropolis-Hasting, il se compilera plus rapidement et convergera plus rapidement que Metropolis-Hasting. Contrairement à Metropolis-Hasting qui conservera les similarités du modèle et gardera plus d'informations au fur et à mesure des itérations.

Les deux algorithmes ont leurs avantages, dans un modèle où chaque sommet sont équivalents il sera plus judicieux de choisir Swendsen-Wang pour une convergence plus rapide. Dans le cas où le modèle est spécifique et que les sommets ont des similarités entre eux, il vaudrait mieux se tourner vers Metropolis-Hasting.

# Chapitre 4

# Code

Voici le code de Metropolis-Hasting et de Swendsen-Wang, le reste du code comme les classes ou la partie Python se trouve sur le Github : `https://github.com/gtk-gh/ter_potts_model` .

## 4.1 Metropolis-Hasting

```
#include "metropolis_hastings.h"
#include <iostream>
using namespace std;


matrice q(const matrice & M){
    matrice res;
    res = M;
    int n = res.getFullSize();
    vector<double> etat = res.getVecEtat();
    int e = etat.size();
    int randU = rand() % n; // choix d'un sommet alatoire uniformment
```

```
    int randV = rand() % e; // choix d'un tat alatoire uniformment
    shared_ptr<point> sommetPtr = (res.getSommet(randU));
    double etatValue = etat[randV];
    changeEtat(sommetPtr, etatValue);
    return res;
}


vector<int> voisin(int i, matrice & M){
    int n = M.getSize1();
    int m = M.getSize2();
    vector<int> voisin;
    if (i<n){ // premiere ligne
        if (i == 0){
            voisin.push_back(1);
            voisin.push_back(n);
        }
        else if (i == (n-1)){
            voisin.push_back(i-1);
            voisin.push_back(i+n);
        }
        else {
            voisin.push_back(i-1);
            voisin.push_back(i+1);
            voisin.push_back(i+n);
        }
    }
```

```
    else if (i>=n*m){} // hors de la matrice
    else if (i >= (n*(m-1))){ // derniere ligne
        if ((i%n) == 0){
            voisin.push_back(i-n);
            voisin.push_back(i+1);
        }
        else if (i%n == (n-1)){
            voisin.push_back(i-n);
            voisin.push_back(i-1);
        }
        else {
            voisin.push_back(i-1);
            voisin.push_back(i+1);
            voisin.push_back(i-n);
        }
    }
    else {// le reste de la matrice
        if ((i%n) == 0) { // cot gauche
            voisin.push_back(i-n);
            voisin.push_back(i+1);
            voisin.push_back(i+n);
        }
        else if ((i%n) == (n-1)) { // cot droit
            voisin.push_back(i-n);
            voisin.push_back(i-1);
            voisin.push_back(i+n);
```

```
        }
        else { // le reste
            voisin.push_back(i-n);
            voisin.push_back(i-1);
            voisin.push_back(i+n);
            voisin.push_back(i+1);
        }
    }
    return voisin;
}

int sim(int i, int j, vector<int> classes){
    // le but ici est de construire des similarits entre les sommets
    // int i et j reprsentent lse sommets i et j respectivement que l'on
   veut tester
    // vector<double> classes reprsente un vector de sommet avec comme
   valeur sa classe par dfaut la classe 0 ne
    // reprsente aucune similarit
    // exemple : [ 0 , 1, 0 ,1 ,2 ,2] ici le sommet 0 et 2 n'ont aucune
   similarit avec d'autres sommets,
    // le sommet 1 et 3 ont les memes similarits et le 4 et 5 aussi.
    int res = 0;
    if (classes[i] == classes[j]){
        res = 1;
    }
    return res;
```

```
}


double p_T_z2(double T, matrice & M, vector<int> simil){
    // Initialisation de variable
    double delta , res;
    int s;
    res = 0;
    int n = M.getSize1();
    int m = M.getSize2();
    // on rcupre toutes les valeurs de la matrice + sa taille
    vector<shared_ptr<point>> AllSommet = M.getAllSommet();
    int f = M.getFullSize();
    // on cre deux variables de point pour parcourir tous les points de la
    matrice
    shared_ptr<point> point_i;
    shared_ptr<point> point_j;
    for (int i = 0;i<f;i++){
        point_i = M.getSommet(i);
        vector<int> voisinage = voisin(i,M);
        for (int j = 0;j<voisinage.size();j++){
            delta = 0;
            point_j = M.getSommet(voisinage[j]);
            // on calcule delta
            if(abs(point_i->getEtat() - point_j->getEtat() ) < pow(10,-9))
    delta = 1;
            s = sim(i, voisinage[j], simil);
```

```
            res = res + (1-delta)*s;
        }
    }
    return exp(-(1/T)*res); // Renvoi 0, car trop grande valeur dans l'
   exponentielle
}

vector<matrice> mh1(int n, matrice & X0, double T,vector<int> simil){
    /// INITIALISATION ///
    vector<matrice> res;
    res.push_back(X0);//initialisation de X0 dans le vecteur rsultat
    int accept = 0; // compteur pour le taux d'acceptation
    /// BOUCLE ///
    for (int i = 1 ; i<n; i++){
        matrice Y; // matrice Y gnr par la loi instrumentale
        Y = q(res[i-1]); // generation du candidat depuis X_(n-1) grce  la
    loi instrumentale
        // CALCUL DE ALPHA
        double alpha = 1;
        double temp = p_T_z2(T,Y,simil) / p_T_z2(T,res[i-1],simil);
        if(temp<alpha) alpha = temp;
        double u = ((double)rand()/(double)RAND_MAX); //gnration d'un
   double compris entre 0 et 1 de manire uniforme
        if (u < alpha){ // cas o on accepte Y
            res.push_back(Y);
            accept++;
```

```
        }
        else{
            res.push_back(res[i-1]);
        }
    }
    cout << "Il y a eu "<<accept<<" acceptations." << endl <<"Ce qui fait
   un taux d'acceptation de "<<double(accept)/n<<".";
    return res;
}
```

## 4.2 Swendsen-Wang

```
#include "swendsen_wang.h"
#include <iostream>
using namespace std;

// Fonction pour gnrer les clusters via la procdure de percolation
matrice generate_clusters(matrice& config, double beta ){
    int n = config.getSize1(); // taille de la grille
    int E = normeinf(config.getVecEtat()); // nb d'tats
    matrice clusters = matrice(n,n); // nous indiquera  quel cluster est
   assign chaque point
    matrice visited = matrice(n,n); // est ce que le point est visit ?
    int current_cluster = 1;
```

```
 vector<pair<int,int>> active;
 random_device rd;
 mt19937 gen(rd());
 uniform_real_distribution<>dis(0.0,1.0);
 auto etendre_cluster = [&](int row, int col, int cluster){ // fonction
 lambda
     visited(row,col)->changeEtat(1.0);
     clusters(row,col)->changeEtat(cluster);
     vector<pair<int,int>> voisins; // on stockera les voisins de
chaque point dans cette variable
     if(row>0) voisins.emplace_back(row-1,col); //voisin ouest
     if(row<n-1) voisins.emplace_back(row+1,col); // voisin est
     if(col>0) voisins.emplace_back(row,col-1); // voisin sud
     if(col<n-1) voisins.emplace_back(row,col+1); // voisin nord
     for(const auto& voisin : voisins){
         int voisin_row = voisin.first;
         int voisin_col = voisin.second;
         // Conditions pour ajouter le voisin au cluster
         if((visited(voisin_row,voisin_col)->getEtat() == 0.0) &&
config(row,col)->getEtat() == config(voisin_row,voisin_col)->getEtat()
){
             double p = 1.0 - exp(-beta);
             if(dis(gen)<=p){
                 active.emplace_back(voisin_row,voisin_col);  // On
ajoute le voisin pour le visiter plus tard
```

```
                    visited(voisin_row,voisin_col)->changeEtat(1.0); // On
    le marque comme visit
                }
            }
        }
    };
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            if(visited(i,j)->getEtat()==0){
                etendre_cluster(i,j,current_cluster); // On regarde si on
   peut crer un cluster  partir de ce point
                // On regarde si on peut tendre le cluster  partir des
   voisins du point prcdent
                while(!active.empty()){
                    pair<int,int> current = active.back();
                    active.pop_back();
                    etendre_cluster(current.first, current.second,
   current_cluster);
                }
                current_cluster++;
            }
        }
    }
    return clusters;
}
```

```
matrice update_configuration( matrice& config, matrice& clusters){
    int n = clusters.getSize1();
    int E = normeinf(config.getVecEtat());
    cout << E;
    matrice new_config(n,n,config.getVecEtat());
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(0, E);
    int new_etat;
    for(int cluster = 1; cluster<= clusters.getEtatSom().back(); cluster
   ++){
        new_etat = dis(gen);
        for(int i = 0; i<n; i++){
            for(int j = 0; j<n; j++){

                // On vrifie qu'on est dans le bon cluster
                if(clusters(i,j)->getEtat() == cluster){
                    new_config(i,j)->changeEtat(new_etat);
                }
            }
        }
    }
    return new_config;
}
```

# Bibliographie


[AMS08] L. Stanberry A. Murua and W. Stuetzle. On potts model clustering, kernel k-meansand density estimation. *Journal of Computational and Graphical Statistics*, 2008.

[ARAAR07] Mark A. J. Chaplain Alexander R. A. Anderson and Katarzyna A. Rejniak. *Single-Cell-Based Models in Biology and Medicine*. Birkhäuser, 2007.

[Bae21] Charlotte Baey. Statistique computationnelle. *https://baeyc.github.io/files/Poly_StatComp.pdf*, 2021.

[DC15] Hugo Duminil-Copin. Order/disorder phase transitions : the example of the potts model. *https://www.unige.ch/~duminil/publi.html*, 2015.

[Wik] Wikipedia. Potts model. Online free encyclopedia, `https://en.wikipedia.org/wiki/Potts_model`.