Faculté des Sciences
Université de Montpellier
Lirmm

Rapport de stage
Ouvertures et finales d'Eternity II

Auteur :
Fati Chen

Référent :
Eric Bourreau

31 août 2016

Table des matières

Table des figures

Liste des tableaux

Remerciements

Je tiens tout d'abord à remercier Eric BOURREAU, mon encadrant pour m'avoir proposé ce stage, je tiens évidemment à remercier le LIRMM et l'UM pour l'accueil et l'hospitalité.

Je remercie aussi toute l'équipe MAORE,la ribambelle de stagiaire et toutes les autres personnes dont j'ai fait la connaissance durant ces deux courts mois. Grâce à eux j'ai découvert de nouvelles choses.

Enfin, je remercie ma famille et mes amis pour leur aide, leur intérêt et leur soutien, notamment Raphaël, Anthony et Jean–François qui m'ont aidé, entre autres, à la relecture de ce rapport.

1 Introduction

Les puzzles et casses-têtes nous ont toujours passionnés, pour faire passer le temps ou pour se poser des défis. Eternity II est un de ces jeux où le principe peut être compris par tous, mais pourtant sa résolution est extrêmement complexe. Ce genre de paradigme est à l'heure actuelle l'un des problèmes mathématiques qui régissent notre monde. La plupart des systèmes informatiques et méthodes de chiffrement reposent sur ce genre de mécanisme : une simplicité de mise en place, mais une complexité de destruction.

Eternity II n'est solvable à l'heure actuelle qu'en testant toutes les combinaisons par méthode de bruteforce. Ce qui nous fait poser une question importante : comment, avec l'augmentation exponentielle des données et des nouvelles technologies, sommes-nous réduit à utiliser une méthode aussi simple. Par extension, est-il plus efficace d'accumuler des données avant de le résoudre plutôt qu'essayer d'accélérer la résolution basique.

Dans un premier temps, nous verrons les origines du jeu, la difficulté à laquelle nous sommes confrontés et l'état de l'art des méthodes de résolution.

Ensuite, nous présenterons la problématique, ce qui à déjà été fait tout au long de l'année et l'approche initiale du problème.

Pour conclure, nous évoquerons les résultats et réflexions qui peuvent en être tirés.

Par ailleurs, ce compte rendu comporte un manuel d'utilisation et un manuel technique fourni, car les applications développées, ou tout du moins leur logique, est destinée à être réutilisée ou améliorée.

2 Eternity II

2.1 Les origines

Eternity II est le fier successeur d'Eternity.

La première version sortie en 1999, était composée de 159 pièces de différentes formes, cependant ces formes peuvent être décomposés en forme de triangles équilatéraux (ou leur moitié) qui doivent être placés sur un plateau octogonal.

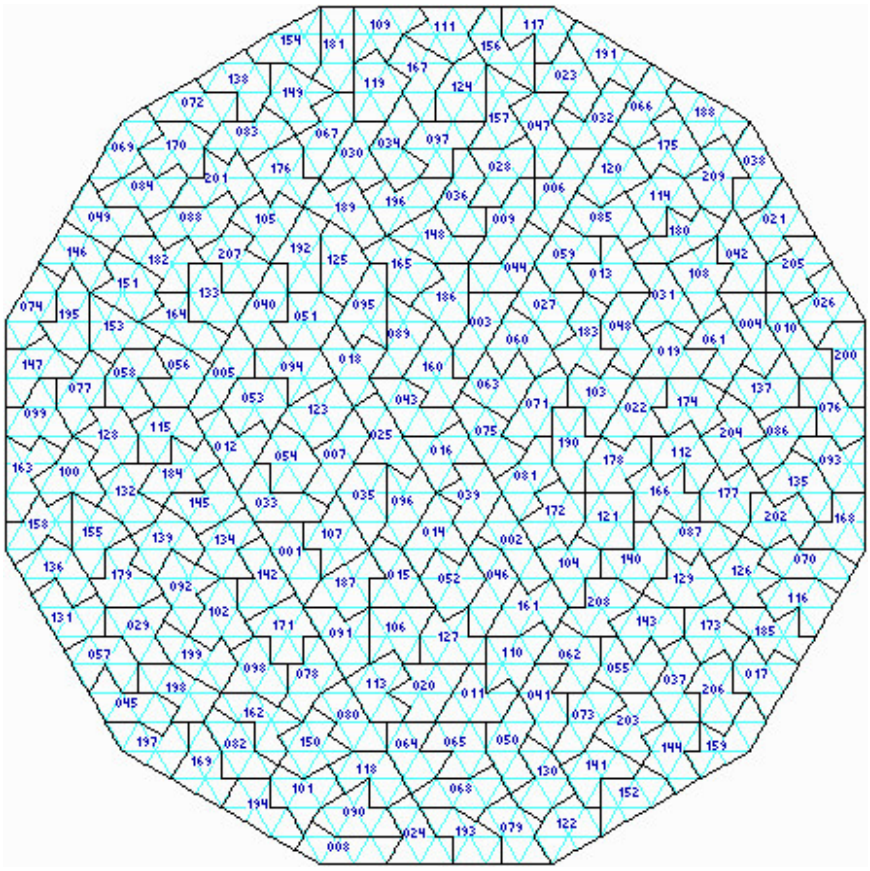

FIGURE 1 – Eternity I

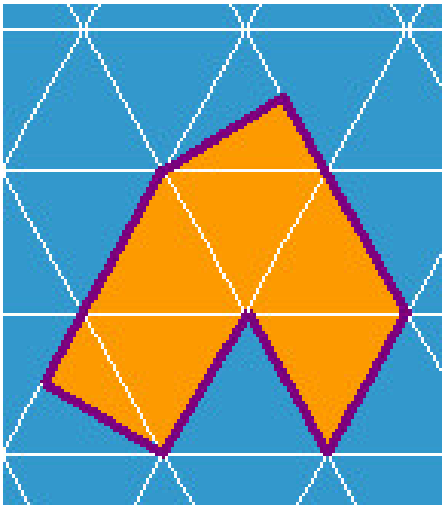

FIGURE 2 – Forme d'une pièce d'Eternity I

Son point faible se trouvait dans la disposition de ses pièces sur le plateau : il était possible de pré-calculer des régions, puis de remplir l'espace restant en s'assurant que celui-ci possède une forme adaptée aux pièces restantes [1]. De cette façon, le puzzle fut résolu en à peine un an (contrairement aux 3 ans prévus par le créateur), par deux mathématiciens, qui ont ainsi empoché la récompense s'élevant à 1000000£.

Après cet « échec », Christopher Monckton, le créateur d'Eternity [2], décide en 2008 de sortir une deuxième version, bien plus complexe, avec à la clé 2000000$pour celui qui arriverait à la résoudre au bout de deux ans.

FIGURE 3 – La boite et les pièces d'Eternity II

C'est un puzzle de 16 par 16 qui sort sous le nom d'Eternity II. Ce puzzle est composé de 256 pièces carrées, qui ont chacune 4 faces colorées (ou demi-motifs).

Ces pièces peuvent être classées en trois catégories suivant le nombre de faces grises qu'elles possèdent :

FIGURE 4 – **pièce de coin :** 2 faces grises

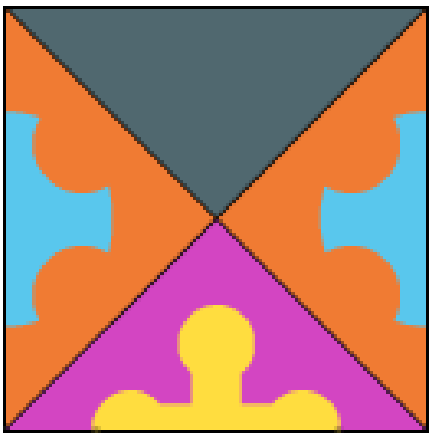

FIGURE 5 – **pièce de bord :** 1 faces grise

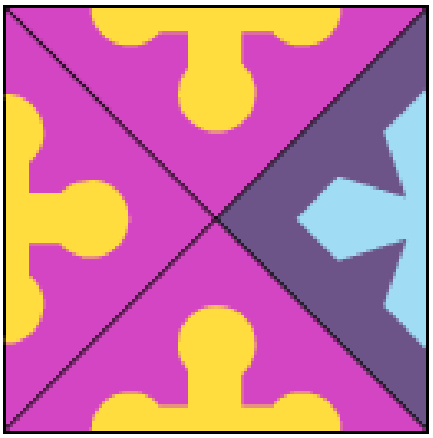

FIGURE 6 – **pièce d'intérieur :** toutes les faces de couleur

Ces pièces ne possèdent pas de formes comme dans un puzzle classique. Afin de les faire correspondre les unes avec les autres, il est nécessaire que les faces adjacentes de chaque pièce voisine soient de la même couleur, dés lors les pièces « matchent [Figure 7] » .

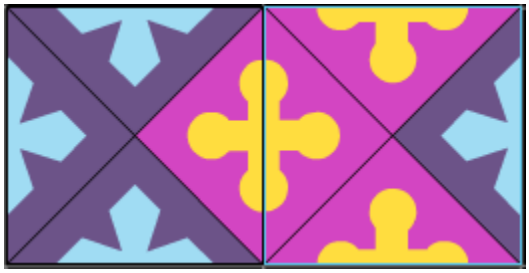

FIGURE 7 – Deux pièces correctement placés (matchés)

Par conséquent, une pièce peut quasiment être placée n'importe où sur le plateau car son placement dépend des couleurs des pièces adjacentes. De plus, les pièces n'ont pas d'orientation prédéterminée (elles peuvent être placées à n'importe quelle rotation).

Pour résumer, la plupart des pièces peuvent être posées n'importe où sur le plateau à différentes rotation car la position dépends entièrement des pièces adjacentes posées auparavant.

Enfin, comme leur nom l'indique, les pièces de coins sont les seules à pouvoir être posées dans les coins du plateau, c'est aussi valable pour les pièces de bord qui ne peuvent être placées que sur les bords du plateau, ces deux types de pièces ne peuvent pas être ailleurs car leurs faces grises doivent « matcher [Figure 7] » avec les bords du plateau.

2.2 Le défi

Malgré le fait que la récompense ait expiré le 31 décembre de l'année 2010, le problème et l'enthousiasme qu'a engendré Eternity II ne s'est pas calmé pour autant (enfin si...un peu). Car loin d'être juste un jeu avec une importante cagnotte il recel en son cœur des secrets d'une certaine valeur.

En effet, jusqu'à maintenant, personne n'a réussi à résoudre ce puzzle, même pas effleuré la solution, malgré l'aide de supercalculateurs et de nombreux spécialistes, que ce soit des mathématiciens ou des informaticiens.

Pourquoi ? Car derrière ce jeu anodin se cache l'un des plus grand problème du monde actuel : les problèmes NP-complets [3]. Ceux-ci sont fait de telle sorte que même en connaissant leur structure ou fonctionnement, il est pratiquement impossible d'en déduire un algorithme (moyen de résoudre) efficace afin de trouver la solution. Ce type de problème est communément appliqué dans le chiffrement. Le meilleur moyen de cacher une aiguille (solution) est de la placer dans gros paquet d'aiguilles, plus le tas est gros, plus on met de temps à la [l'aiguille] trouver.

Exemple 2.1. Le nombre de combinaisons possibles pour Eternity II s'élève à 10^{545}, c'est à dire environ 10^{450} fois le nombre d'atomes dans l'univers connu (estimé à au plus 10^{80}) ! ! ! Ca fait un gros tas d'aiguilles ! !

2.3 Les II lois d'Eternity II

Pour rendre Eternity II complexe et combinatoire, il est nécessaire de respecter les deux lois d'Eternity II :

Loi 1. Chaque pièce est unique

L'unicité des pièces est indispensable pour complexifier le problème, car sinon, on peux considérer qu'une pièce peux être placée à plusieurs endroits en fonction du nombre de « clones » qu'elle possède. Ce qui réduit grandement l'espace de recherche [de la solution].

Loi 2. La quantité de couleurs et de pièces est finement calculée

En effet, si l'on augmente le nombre de couleurs, on obtient des couplage uniques : une pièce ne peux être couplée qu'avec une autre pièce (ou dans le meilleur des cas limite le couplage des pièces). A l'inverse, si il n'y a pas assez de couleurs, on obtient des doublons, les pièces ne sont plus uniques, ce qui va à l'encontre de la première loi.

Par ailleurs, certaines couleurs sont exclusives aux pièces de coin et de bord, car ceux-ci étant liés en eux mais seulement sur le périmètre extérieur du plateau il est nécessaire d'ajouter des couleurs supplémentaires tenant compte qu'ils n'ont que 2 ou 3 faces disponibles (le reste étant des faces grises).

2.4 État de l'art

Un grand nombre de méthodes ont été mis en place afin de résoudre ce problème.

Il serait trop long de présenter et décrire les différentes méthodes mise en place car il requièrent une certaine connaissance dans les domaines auxquels ils sont appliqués. Malgré tout, les différentes approches seront notés ici à titre informatif.

Pour commencer, il existe plusieurs solveurs graphiques afin de pouvoir résoudre le puzzle manuellement ou assisté par l'ordinateur. Certains d'entre eux permettent même l'import export de la progression actuelle.

- E2_manual : `https://sourceforge.net/projects/e2manual/`(anglais)
- E2Lab : `http://eternityii.free.fr/`(anglais)

Ensuite, il existe un très grand nombre de solveurs bruteforce plus au moins rapides. Certains d'entre eux peuvent agréger plusieurs machines afin d'augmenter la puissance de calcul via le réseau.

Ce problème a été abordé de façon très varié au niveau théorique et applicatif.

Par exemple, Eternity à été adopté sous forme de graphe [4] ou de sous forme de contraintes [5].

On note aussi un procédé intéressant de résolution grâce à une approche nommée **SAT** (satisfaisabilité booléenne) s'appuyant sur la logique propositionnelle [6] [7].

3 Problématique

Le but du stage est donc de déterminer si, malgré les dires, on pourrait trouver une méthode de résolution plus efficace que la bruteforce, qui reposerait sur une grande quantité d'informations pré-calculées, cette méthode sera appelée smartforce par la suite.

Ces informations pré-calculées serviront différentes causes, mais deux objectifs principaux peuvent en être explicités :

- les ouvertures
- les finales

Afin de comprendre le principe des ouvertures et des finales, il est important de pouvoir se représenter un arbre des possibilités où chaque branche de l'arbre est une combinaison spécifique.

Exemple 3.1. Prenons pour exemple l'arbre des possibilités d'un lancé de monnaie (en supposant que la monnaie ne tombe pas sur la tranche [8]). A chaque étape, deux choix s'offrent à nous :

- la pièce tombe sur pile
- la pièce tombe sur face

Supposons que l'on lance la pièce trois fois. On a donc un arbre ressemblant à ça :

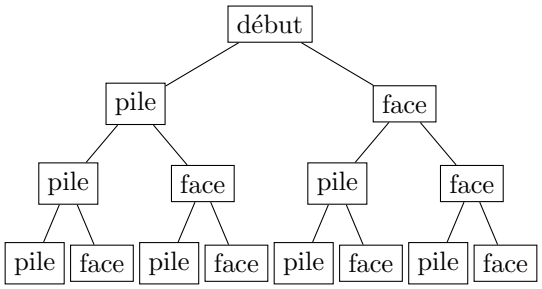

FIGURE 8 – Arbre de possibilités d'un lancé de monnaie

Grâce à cet arbre, à chaque fois qu'une pièce sera lancée jusqu'à trois fois d'affilée, la combinaison (ou chemin) figurera dans l'arbre.

Chaque nœud de l'arbre possède un sous-arbre (s'il est pris comme racine), ce sous-arbre peut être vide.

Note. Pour l'arbre de résolution d'EternityII, c'est à peu près la même chose mais l'arbre est bien plus grand. Les nœuds de celui-ci ne peuvent pas être prédits (on ne connait que les nœuds suivants du chemin emprunté).

Cet arbre sera représenté par la suite sous cette forme :

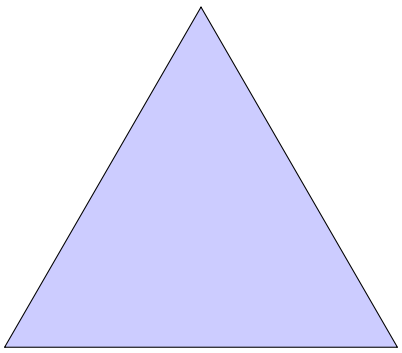

FIGURE 9 – Représentation simplifiée d'un arbre des possibilités

3.1 Ouvertures

L'idée est de pré-calculer tous les débuts jusqu'à un certaine profondeur de l'arbre pour ensuite créer des sous-arbres pouvant être parcourus parallèlement ou pondérer les sous-arbres générés pour les classer (par taille, ...).

Exemple 3.2. Dans le cas du lancé de monnaie, je sais que le premier lancer, me donne soit pile soit face. En connaissant cela, je peux séparer l'arbre en deux sous-arbres. Cela m'évite de recalculer la première profondeur.

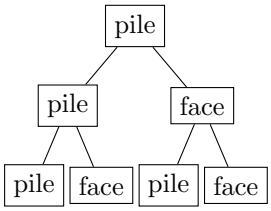

FIGURE 10 – Sous-arbre de pile

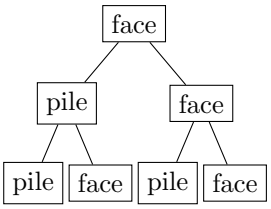

FIGURE 11 – Sous-arbre de face

Remarque. L'utilité des ouvertures dans un lancé de pièce (de monnaie) est très discutable, mais elle prends son sens lorsque :

- le calcul de chaque nœud est gourmand en ressources.
- la quantité de nœuds est importante.

L'arbre des possibilités dans le cas des ouvertures est représenté comme ceci :

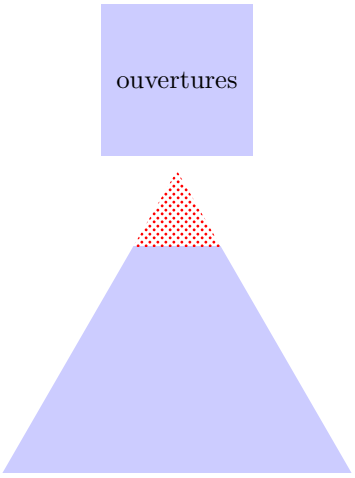

FIGURE 12 – Représentation simplifiée des ouvertures

Le triangle est tronqué car le début a déjà été pré-calculé, c'est comme si l'on avait placé tout les sous-arbre côte à côte. Les ouvertures étant stockées sous forme de données, elles sont représentés par un carré.

3.2 Finales

Les finales permettent de prédire si le chemin emprunté dans l'arbre mène à une solution. Par conséquent, toutes les finales possibles sont pré-calculés et stockées sous forme de données.

Grâce à cela, on peux connaître si le chemin choisi (ou combinaison actuelle) est possible sans avoir à finir le chemin en entier. L'arbre est donc tronquée en bas, réduisant l'espace à énumérer.

Les finales sont représentés comme ceci :

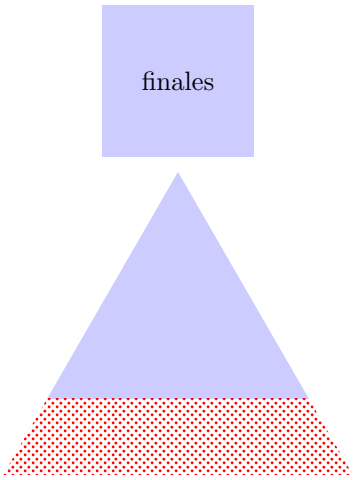

FIGURE 13 – Représentation simplifiée des finales

3.3 Objectif

Grâce à l'aide de ces deux approches. on peux donc diminuer drastiquement la quantité de calcul nécessaire en augmentant la quantité des données stockées. Par ailleurs, en adaptant la méthode de résolution on peux aussi réduire la profondeur totale de l'arbre.

Par conséquent, on réduit non seulement ce qui reste à déterminer, mais aussi la taille de l'arbre dans son ensemble.

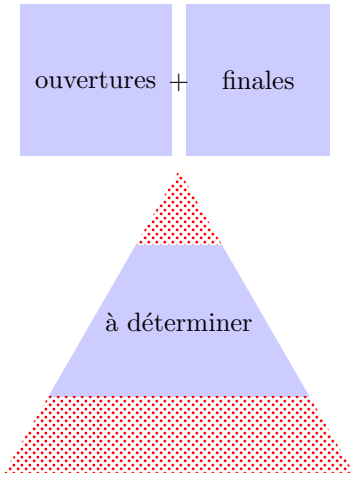

FIGURE 14 – Représentation de la fusion des ouvertures et des finales

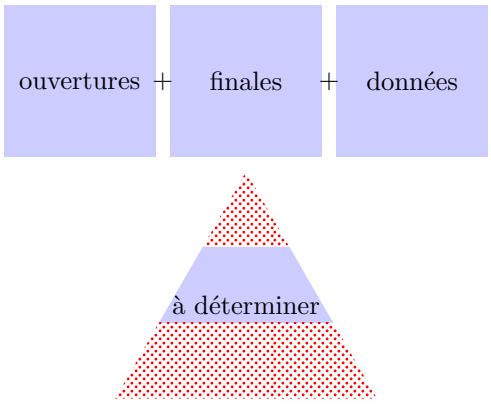

FIGURE 15 – Représentation simplifiée de l'objectif comprenant les ouvertures, les finales et les données

Par la suite, en supposant que l'emploi des données pré-calculées est longue et fastidieuse, le but sera d'aussi utiliser des technologies avancées afin d'accélérer la résolution.

4 Approche

Dans cette partie, nous expliquerons quel sont les différents outils et stratégies mis en place pour permettre la résolution du problème.

La principale difficulté c'est que le jeu de base est trop complexe et ne permet pas de déterminer si l'approche actuelle est mieux adapté. Par conséquent, on utilisera des instances plus petites, qui sont des plateaux de plus petite taille ($4 \times 4, 5 \times 5, \dots$) qui respectent les deux lois d'Eternity II.

En supposant que la smartforce devient de plus en plus bénéfique suivant la complexité du problème, il est nécessaire de mettre en place une valeur étalon, reposant sur un programme bruteforce qui fournit différentes données permettant d'estimer si cette supposition est vraie.

Chaque nœud de l'arbre des possibilités du jeu doit satisfaire deux données :

variable : une des cases

valeur : une des pièces.

Lors de la progression dans l'arbre, il faut définir quelle valeur sera définie pour quelle variable (choisir telle case sur laquelle sera placée telle pièce). Pour cela, chaque variable (case) possède un domaine de valeur, un tableau contenant contient toutes les valeurs possibles (toutes les pièces qui peuvent être mise dessus) que l'on appellera **domaine**.

Et inversement, chaque valeur (pièce) possède un domaine contenant toutes les variable à laquelle elle peux être appliquée.

Il est possible de déterminer théoriquement, suivant l'instance actuelle, la taille du domaine pour chaque pièce ou case. En effet, en connaissant la moyenne de distribution des couleurs par pièces, il est possible d'évaluer la quantité de pièces à chaque case ou la quantité de pièces adjacentes à une autre pièce.

Exemple 4.1. Pour un plateau de taille 4. La case en position (O, O) a 4 pièces posables. La distribution des couleurs de ces pièces est équitable (voir Figure 23), par conséquent, elles possèdent toutes les couleurs possibles pour les pièces de bord. Par conséquent, si l'on ne définit pas la pièce qui est placé sur la case en $(0, 0)$, jusqu'à 8 pièces peuvent être placés en $(0, 1)$ et $(1, 0)$.

Remarque. Dans la pratique, chaque couleur n'à pas la même quantité de pièces. Par conséquent, ce résultat peux varier. Mais il permet d'évaluer le plafond supérieur du nombre de cas possible pour une corolle ou le nombre de cas possibles total.

4.1 Bruteforce

Le programme de bruteforce nous fournit une valeur étalon. Il se contente de parcourir tout l'arbre des possibilités.

Il permet aussi de savoir quelle stratégie de parcours du plateau est la plus bénéfique. Les différentes stratégies de choix de variable étalonnées sont :

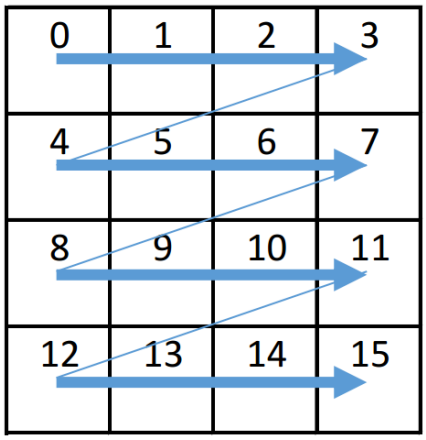

FIGURE 16 – **rowscan** : parcours horizontal du plateau

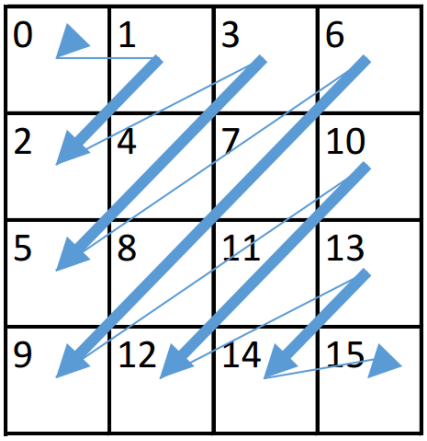

FIGURE 17 – **diagonal** : parcours diagonal du plateau

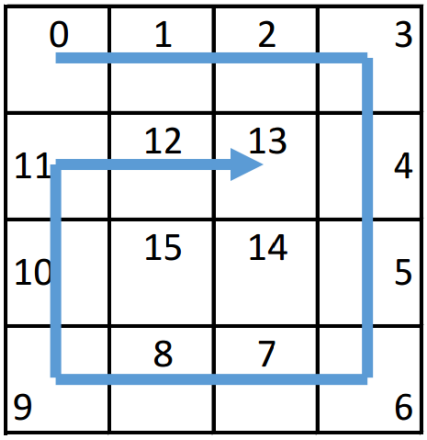

FIGURE 18 – **spiral-in** : parcours en spirale fermente du plateau

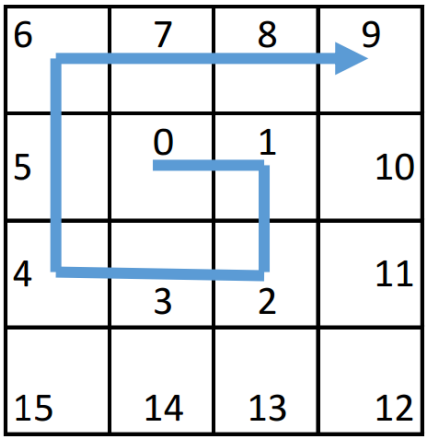

FIGURE 19 – **spirale out** : parcours en spirale ouvrante du plateau

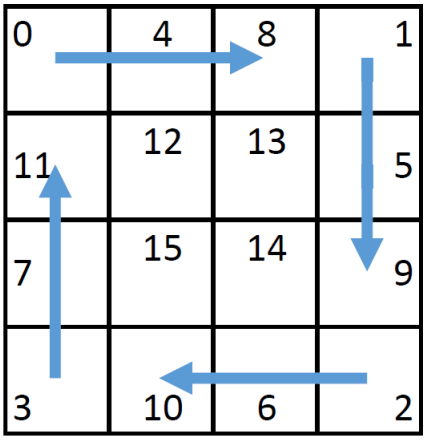

FIGURE 20 – **quad spiral out** : parcours avec quatre spirales fermentes

Le choix des pièces (valeurs) est fait dans l'ordre lexicographique (ordre par défaut).

Les données utilisés pour l'étalonnage sont les suivants :

Première solution : le temps et la quantité de nœuds nécessaires pour trouver la première solution (chaque instance peux en avoir plusieurs)

Nombre total de solutions : pour estimer les placement des solutions dans l'arbre

Nombre total de nœuds : pour pouvoir estimer la rapidité de l'algorithme pour trouver la première solution

Temps total : pour connaître si l'algorithme utilisé est plus performant que celui d'avant

4.2 Smartforce

Le principe de la smartforce repose sur la quantité des données qu'elle possède. Par conséquent, il est important que ces données soient bien organisées afin de pouvoir les traiter avec aise. Ces données sont organisées pour former des modèles. Chaque modèle peut être interprété comme un point de vue différent du problème. La liaison de plusieurs modèles variés fait la puissance de la smartforce.

On note aussi l'introduction de nouvelles méthodes de parcours :

- Choix de variable :
  - **Optimiste** : on choisit la variable qui à le plus grand domaine
  - **Pessimiste** : on choisit la variable qui à le plus petit domaine
- Choix de valeur :
  - **Lexicographique** : utilisé par défaut par la bruteforce, on choisit la valeur suivante dans l'ordre par défaut.
  - **Optimiste** : la valeur dont le domaine est le plus grand.
  - **Pessimiste** : la valeur dont le domaine est le plus petit.

Différents modèles seront donc présentés par ordre de difficulté, car, afin de fournir des types de données variés, il est nécessaire d'abstraire le problème initial.

4.2.1 CaPi

CaPi est le plus simple modèle représentant le problème. Il a pour variable les cases (identifiées par un numéro ou par ses coordonnées sur le plateau) et pour valeur les pièces (identifiées par un numéro et par leur rotation). C'est aussi lui qui est utilisé comme modèle par défaut pour le choix des valeurs et variables.

En somme, chaque pièce est associé aux cases sur laquelle elle peut être posée et inversement, chaque case contient la liste des pièces qu'elle peux avoir.

Remarque. Le domaine des cases est un peu plus complexe : une pièce donnée peux être placée sur la case « jusqu'à 4 fois » (à plusieurs rotation différentes). Par conséquent, lorsque l'on met à jour le domaine de la case (en posant la pièce sur une autre case), on supprime toutes les occurrences de cette pièce du domaine en question.

Par conséquent, chaque pièce possède en vérité quatre domaines distincts (correspondant à chaque rotation de la pièce).

4.2.2 BoCo

Le modèle BoCo (Bordures/Couleurs) est une approche bien plus fine, elle découle de l'abstraction de CaPi.

Si l'on connait le domaine d'une case. On connaît les pièces qui peuvent être posés dessus. Par conséquent, on définit une **Bordure** comme une sous-partie d'une case.

Exemple 4.2. Prenons la case 0, en $(0, 0)$ sur le plateau. Elle peux contenir 4 pièces (n° 0,1,2,3 ; ce sont toutes des pièces de coin) en la rotation 0.

Si l'on décompose la case, celle-ci contient 4 faces :

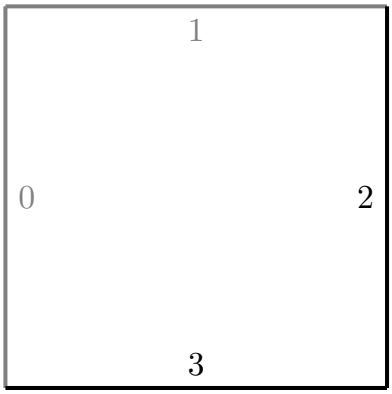

FIGURE 21 – Détails d'une case

Les bords 1 et 2 étant gris, il n'est pas nécessaire de les représenter.

On décompose ensuite les 4 pièces :

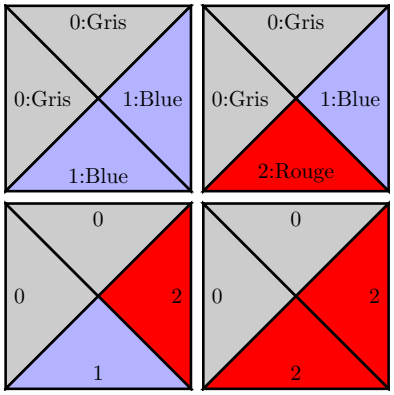

FIGURE 22 – Couleurs des 4 pièces

On associe ensuite les couleurs et leur occurrence à chaque bord :

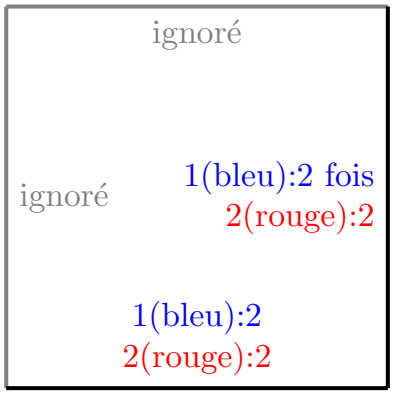

FIGURE 23 – Occurrences et couleurs de bordures

Cette simplification du problème nous permet de connaître plus simplement si deux cases adjacentes possèdent des matching non possibles. Il suffit ensuite d'éliminer les pièces possédant la couleur manquante.

Exemple 4.3. Dans cet exemple, pour le deuxième cas [Figure 25], toutes les pièces (orientées) ayant du rouge sur la face 2 ne peuvent être posées sur la case. Cela vaut aussi pour la case adjacente où la couleur verte ne peux être posée.

Les deux cases ont des matching possibles :

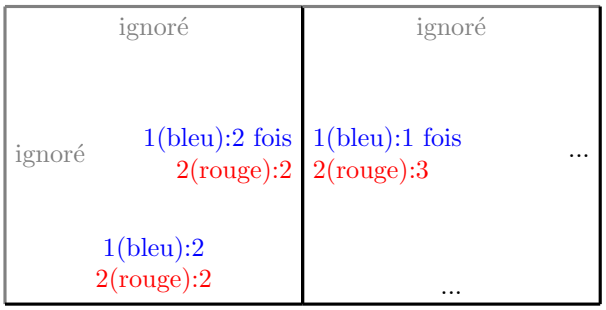

FIGURE 24 – Cases adjacentes avec des matching possibles

Les deux cases ont des matching non possibles

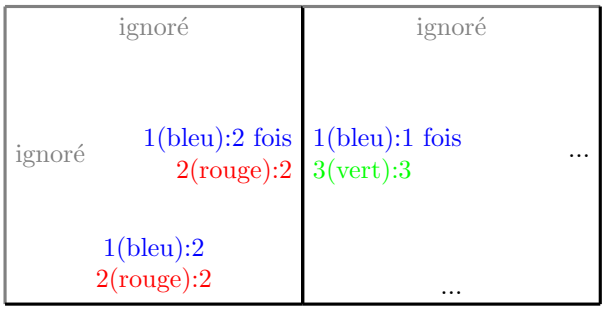

FIGURE 25 – Cases ayant des matching non possibles

Les bordures sont identifiées par un numéro. Chaque **bordure** possède un domaine contenant les **couleurs** uniques qu'elle peux avoir et leur cardinalité [Figure 23].

Les couleurs sont aussi identifiés par un numéro. Chaque **couleur** contient l'ensemble des bordures où elle peux être placée, et combien de fois elle peux l'être (cardinalité).

L'utilité de ce modèle réside dans la propagation des données. Le but étant de propager une information à travers les cases, en ne mettant à jour que les données concernées.

Dans une disposition CaPi, pour appliquer la propagation des données, on est obligé de faire le produit des domaines des cases adjacentes pour chaque matching (comparer les pièces possibles de chaque case entre eux). Ce qui s'avère très couteux et beaucoup de ces vérifications sont inutiles.

Par contre, grâce à **BoCo**, il suffit de vérifier que la cardinalité des couleurs reste positive pour les deux bordures adjacentes. Seulement lorsque le nombre d'occurrences d'une couleur tombe à zéro, l'autre est invalidée (disons, la couleur rouge). Il suffit ensuite de récupérer les autres couleurs de la pièce qui disparait (car elle avait la couleur rouge), et de décrémenter les autres bordures de la case concernée. Si leur occurrence tombe à zéro, on met à jour leur bordure adjacente, sinon, la propagation s'arrête.

Note. Le fait que la propagation soit plus efficace en BoCo est due à la relation qu'à la bordure avec son domaine (couleurs). Contrairement à CaPi où, à la fin de la résolution, une case ne possède plus qu'une seule pièce (et vice versa). Plusieurs bordures vont posséder une même couleur. Le modèle CaPi est en $1 \Leftrightarrow 1$ alors que le modèle BoCo est en k (bordures) $\Leftrightarrow 1$ (couleur).

Exemple 4.4. Si une couleur disparait, alors toutes les pièces l'ayant ne peuvent plus être placées à cette case, par conséquent, les autres bords de la case ont (probablement) des couleurs qui disparaissent aussi (propagation de la disparition).

4.2.3 Corolles

Une corolle est une surface du plateau qui est pré-calculée afin de pouvoir être réutilisée par la suite.

La corolle possède une pièce centrale, de laquelle sont calculées les pièces avoisinantes. Par conséquent, la taille de la corolle est définie par la distance de la pièce la plus au bord par rapport à la pièce centrale, cette distance sera appelée hamming (H).

Remarque. Une corolle de hamming 0 est la pièce elle-même.

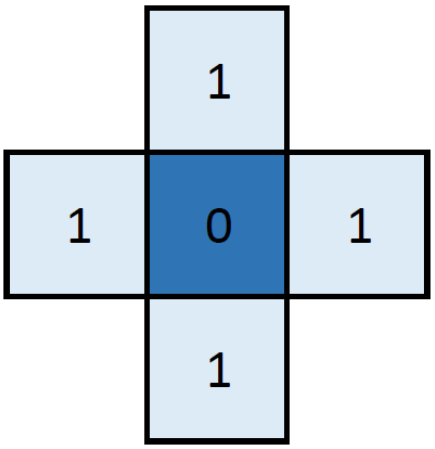

FIGURE 26 – Corolle de hamming 1

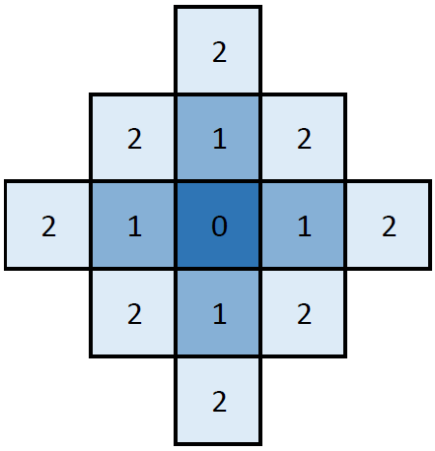

FIGURE 27 – Corolle de hamming 2

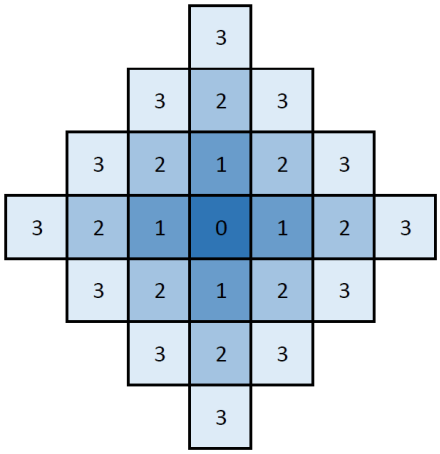

FIGURE 28 – Corolle de hamming 3

Son nom, corolle, vient de sa forme. Suivant son placement, cette forme change. Les différentes formes dépendent de la taille de la corolle (« rayon » de la corolle) et de la position sur le plateau.

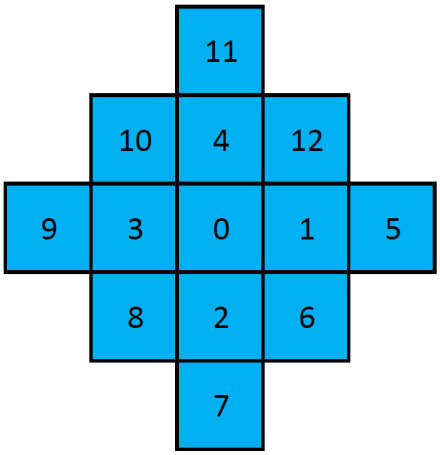

FIGURE 29 – Ordre de placement des pièces d'une corolle de Hamming 2

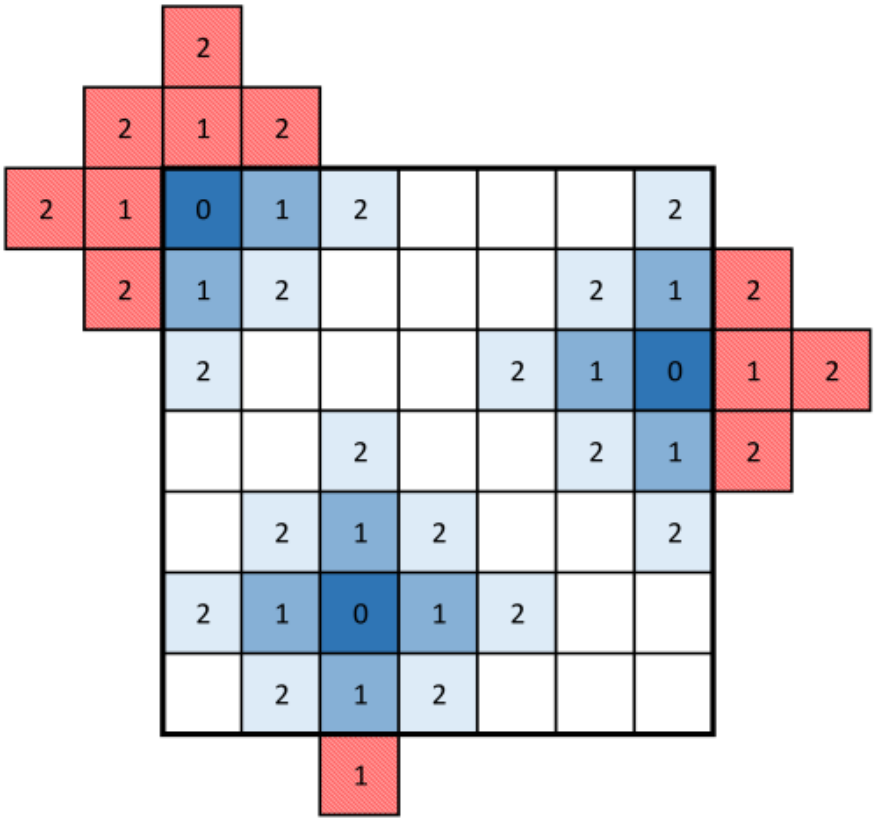

FIGURE 30 – Différentes formes de la corolle de hamming 2 sur une instance de taille 7

Exemple 4.5. Supposons qu'une corolle est seulement définie par sa forme. Prenons comme coordonnées $(0, 1)$ et $(1, 0)$ avec un hamming de 1. La forme des corolles sur ces coordonnées est identique, elle a une forme de tétris. Les deux corolles se superposent en $(0, 0)$ et en $(1, 1)$.

Par conséquent, les corolle en coordonnées $(0, 1)$ peuvent aller en $(1, 0)$ et inversement.

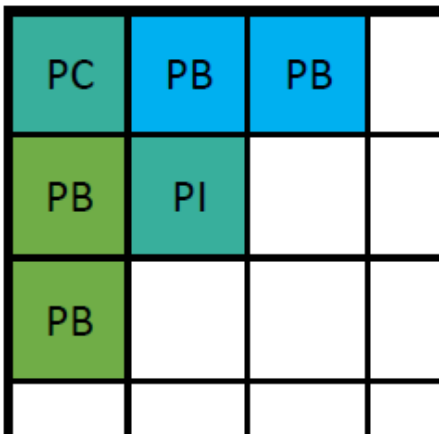

FIGURE 31 – Corolle en $(0,1)$ (blue)et en $(1,0)$ (vert)

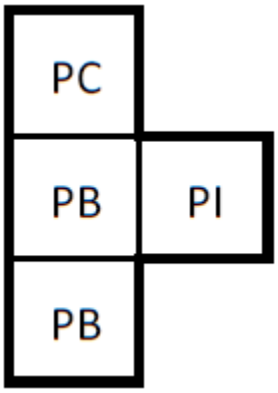

FIGURE 32 – Corolle tetris en $(0,1)$

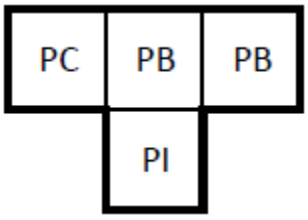

FIGURE 33 – Corolle tetris en $(1,0)$

PC pièce de coin
PB pièce de bord
PI pièce d'intérieur

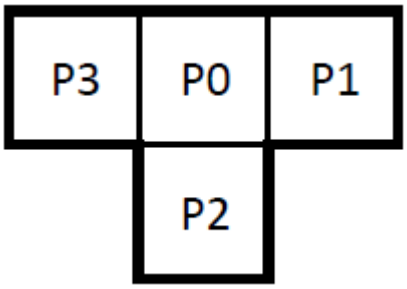

FIGURE 34 – Noms des pièces dans la corolle tetris

Or, pour les types de pièces, il y a un problème. en $(1,0)$ P3 est une pièce de coin, alors qu'en $(0,1)$ c'est une pièce de bord (même problème pour P1).

Donc, non seulement, une corolle est définie par sa forme, mais aussi par sa position.

Par conséquent, une corolle est définie par le type de pièces (coin, bord, intérieur) qui la composent et de sa forme. il n'est pas nécessaire de pré-calculer les corolles sur toutes les cases du plateau, il suffit de partager le plateaux en zones. Toutes les corolles d'une même zone sont identiques.

Remarque. Plus le hamming de la corolle augmente, plus il y a des zones différentes à cause de l'augmentation des types de pièces qui la compose.

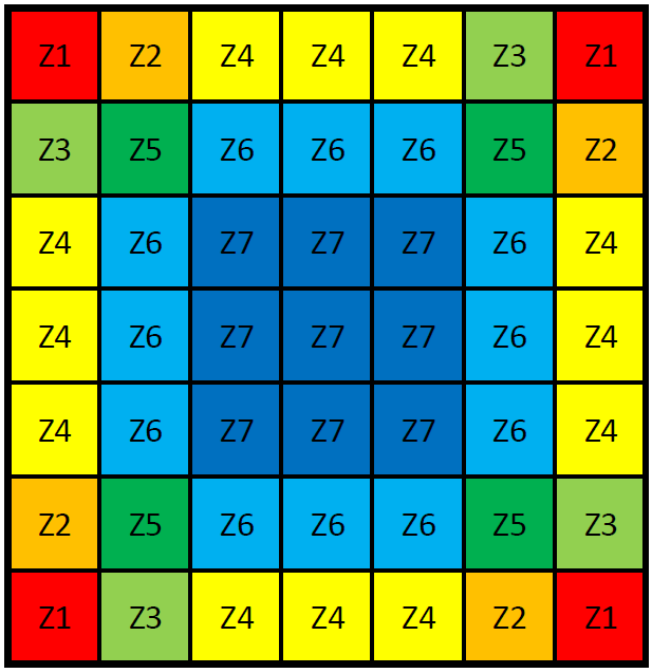

FIGURE 35 – Quantité de zones differentes sur un plateau de taille 7, pour une corolle de hamming 1

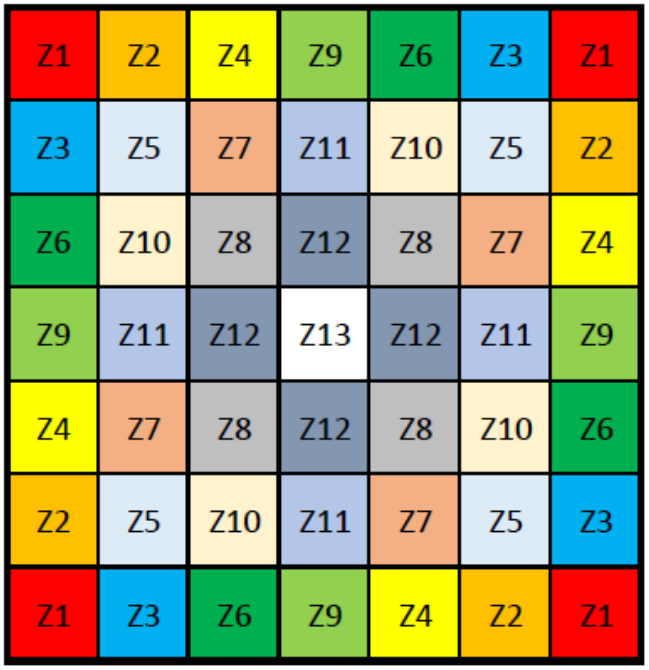

FIGURE 36 – Quantité de zones différentes sur un plateau de taille 7, pour une corolle de hamming 2

Il est possible de déterminer le nombre de zones qui composent un plateau. Elle contient 2 variables, T la taille du plateau et H le hamming. La quantité de zones z est définie par la relation suivante :

$$\begin{cases} z = (H+1)^2 + H + 2 & \text{si } M > H+1 \text{ avec } M\frac{T+T\%2}{2} \\ z = \frac{T+T\%2}{2}^2 \end{cases}$$

Il y a deux cas :

1. Soit M la moitié (arrondie au supérieur) de la taille du plateau. Si $M > H+1$.
   Dans ce cas, Toutes les cases de $(0,0)$ jusqu'en (H,H) sont des zones différentes (toutes ces cases sont en contact avec au moins deux bords du plateau), plus toutes les cases de $(H+1,0)$ jusqu'à $(H,H+1)$ (ceux la ne sont en contact qu'avec un seul bord) et enfin la pièce en $(H+1,H+1)$ qui ne contient que des pièces d'intérieur.
2. , si le $H+1$ est supérieur ou égal, alors toutes les cases de $(0,0)$ jusqu'en (M,M) sont des zones distinctes.

L'espace réel dépend aussi de la taille du hamming.

On a donc deux cas :

Définition 4.1. Pour tout corolle ayant pour pièce centrale P et de taille H d'une zone Z_1 donnée, elle découle de l'expansion d'une corolle de taille inférieure $H-1$ ayant pour zone Z_0 englobant la zone Z_1 avec P identique.

Exemple 4.6. Dans la Figure 36, les zones 8,12,13 sont englobées par la zone 7 de H1 [Figure 35].

Par conséquent, toutes les corolles des zones 9,12,13 en hamming 1 sont une extension des corolles de la zone 7 en hamming 1.

La frontière définit les couleurs au bord de la corolle, ils sont nécessaires pour connaitre les pièces qui peuvent être placées à côté.

Exemple 4.7. Les faces d'une pièce sont la frontière d'une corolle de hamming 0.

Rotation de la corolle

Les corolles concernées par les rotations sont les corolles qui ont au moins une pièce de bord ou de coin. Les pièces de bords et de coin ayant une orientation définie à cause de leur face grise.

Exemple 4.8. Prenons la zone $Z5$ de la Figure 26, suivant l'endroit où elle se trouve, la composition de la corolle est la même, mais « tournée » :

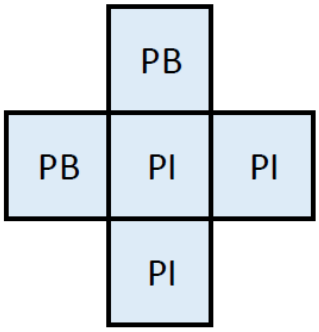

FIGURE 37 – Corolle de hamming 1 avec les types de pièces en $(1, 1)$

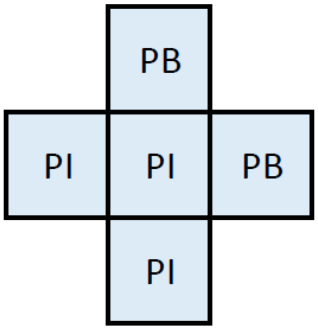

FIGURE 38 – Corolle de hamming 1 avec les types de pièces en $(5, 1)$

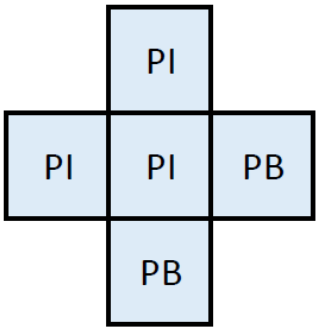

FIGURE 39 – Corolle de hamming 1 avec les types de pièces en $(5, 5)$

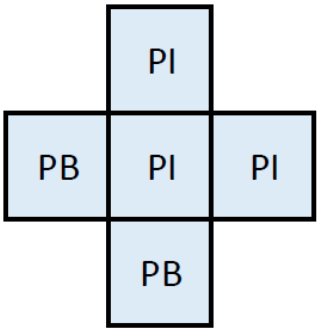

FIGURE 40 – Corolle de hamming 1 avec les types de pièces en $(1, 5)$

La composition et la forme est la même, mais l'ordre est différent (voir Figure 29).

Pour résumer. Une corolle est une surface du plateau d'une certaine forme qui est pré-calculée. Une corolle pouvant être à plusieurs endroit, elle est dépendante d'une zone. Elle est composée d'une pièce principale qui la définit ainsi que d'une frontière. Une corolle est composée d'une corolle de hamming inférieur, se trouvant dans une zone englobant celle de la corolle actuelle. Certaines corolles d'une même zone peuvent être différentes à cause de leur « rotation ».

Afin que la corolle soit correcte, il est nécessaire de connaître les informations suivantes :
- taille du plateau
- zone de la corolle
- la rotation de la corolle
- son hamming
- les pièces qui la composent

4.2.4 BoCoDiag

BoCoDiag est un modèle permettant de lier des couples de couleurs. Son principe est presque le même que pour BoCo, d'où le nom. La différence est que le couple est indirect.

Exemple 4.9. Prenons un zone de 4 cases.

Le principe de BoCoDiag est de récupérer les liaisons diagonales. On en obtiens un couple de couleurs non identique. et pourtant, il contraint 4 couleurs.

Dans cet exemple, le couple (Bleu/Blue) est lié à (Rouge/jaune) en direction Sud-Est et le couple (Blue/-Jaune) est lié au couple (Bleu/Rouge).

Note. La lecture des couleurs se fait dans le sens horaire en commencent par la face à l'ouest de la pièce.

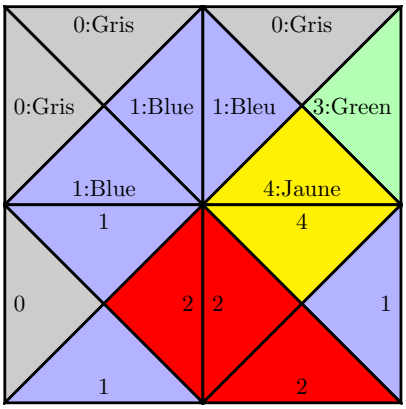

FIGURE 41 – 4 pièces matchés

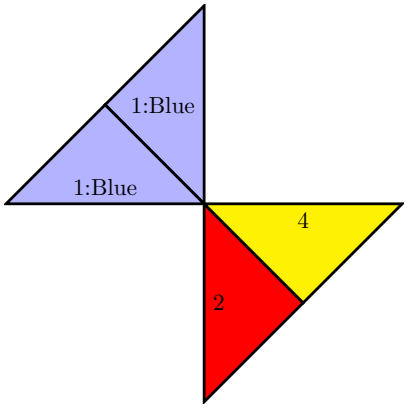

FIGURE 42 – Premier couple BoCoDiag

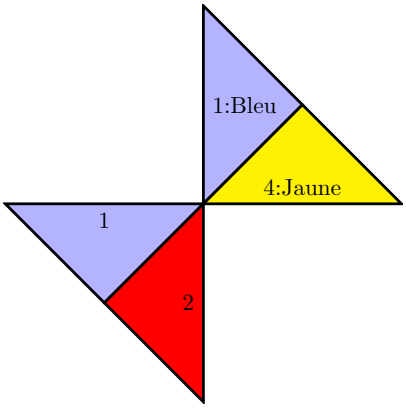

FIGURE 43 – Deuxième couple BoCoDiag

4.3 Corolles dynamiques et marécage

La limitation principale des corolles est que l'espace pré-calculé est souvent l'ensemble des pièces possibles à ses cases. Par conséquent, le fait de traiter une donnée qui nous informe que tout est possible est inutile. D'où l'intérêt des corolles dynamiques. Les corolles dynamiques sont polymorphes et n'ont pas forcément un hamming déterminé. Elle s'agrandissent lorsque qu'une case adjacente [à la corolle dynamique] voit son domaine restraint.

Exemple 4.10. Prenons un plateau de taille 7 dans sa disposition initiale. Plaçons la corolle en $(1, 2)$

4	20	20	20	20	20	4
20	100	100	100	100	100	20
20	100	100	100	100	100	20
20	100	100	100	100	100	20
20	100	100	100	100	100	20
20	100	100	100	100	100	20
4	20	20	20	20	20	4

FIGURE 44 – Répartition des pièces sur un plateau de taille 7

Note. Il y a 25 pièces intérieures, 4 rotation sont possibles par pièce, par conséquent, 100 « pièces » sont posables sur une case d'intérieur.

Dans cette figure, il n'est pas nécessaire de calculer la corolle en Hamming 1 ou plus, car ce ne sera que tout les cas possibles, on appelle donc H1 le marécage.

Plaçons maintenant la pièce 0 en $(0, 0)$

1	7	20	20	20	20	3
7	58	100	100	100	100	20
20	100	100	100	100	100	20
20	100	100	100	100	100	20
20	100	100	100	100	100	20
20	100	100	100	100	100	20
3	20	20	20	20	20	3

FIGURE 45 – Répartition des pièces sur un plateau de taille 7 avec la pièce 0 en $(0, 0)$

La case en $(1, 1)$ voit son domaine réduit. Par conséquent il est malin d'agrandir la corolle vers cette case car elle pourrait nous fournir plus de données.

Note. Cet exemple est naïf mais illustre bien le principe de corolle dynamique et de marécage.

4.4 Ouvertures

Grâce à l'approche Smartforce et surtout grâce au modèle **Corolle** il est possible d'initialiser efficacement les ouvertures.

Pour ce faire, il suffit de placer plusieurs corolles (donc d'occuper une surface du plateau), de déterminer quels couples (de corolles) sont possibles (ne pas utiliser deux corolles qui placent la même pièce à plusieurs endroits distincs). Exporter toutes les combinaisons possible pour créer toutes les ouvertures possibles (Figure 12).

Exemple 4.11. Prenons un plateau de taille 7. Plaçons y 5 corolles de Hamming 2 de façon à occuper le plus de place possible.

Nous avons donc fait 5 étapes de placement. Mais actuellement, nous venons d'énumérer tous les placements possibles de 45 pièces.

Il ne reste plus qu'à déterminer dans quel cas il est possible de placer les quatres pièces restantes pour terminer le plateau de taille 7.

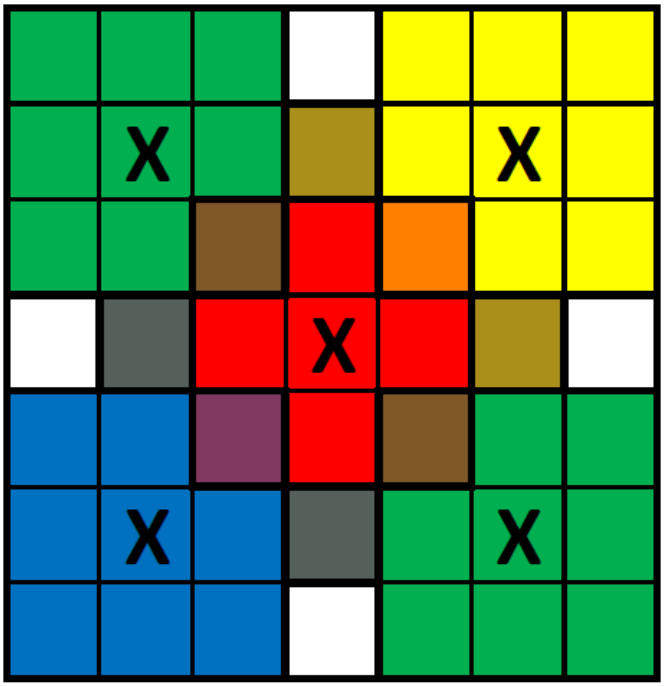

FIGURE 46 – Placement de 5 corolles sur un plateau de taille 7 (ouvertures)

Lorsque plusieurs corolles se superposent, il y à une très forte contrainte à cet endroit, car la pièce qui s'y trouve doit être possible dans toutes ces corolles.

Remarque. Actuellement, il est plus bénéfique de placer moins de corolles sur le plateau pour laisser place aux finales.

4.5 Finales

Pour les finales, le principe est le même que pour les ouvertures, mais le but est différent : le but est de déterminer facilement si la résolution actuelle est viable. Par conséquent, il faut limiter au plus le nombre de finales, afin de déterminer la réalisation du problème le plus rapidement possible.

Par conséquent, le placement des corolles se fait sur les coins (car il n'y à que 4 pièces qui peuvent y être placés)

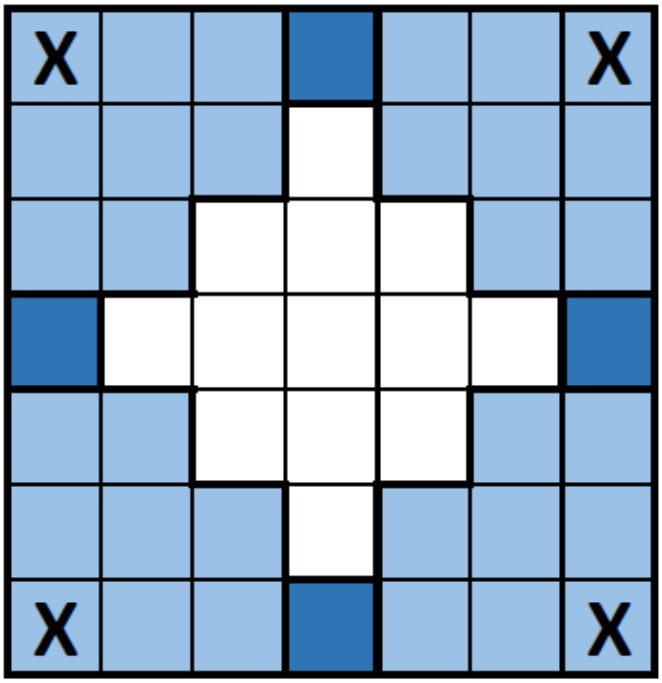

FIGURE 47 – Placement des corolles sur un plateau de taille 7 (finales)

5 Manuel d'utilisation

Dans cette partie nous verrons comment utiliser les différents outils développés pendant la durée du stage.

5.1 Pré-requis

Afin de pouvoir manipuler avec aise l'ensemble des outils présentés ici, il est vivement conseillé de connaître le fonctionnement global d'un Git [9].

L'ensemble des outils présentés ici sont hébergés sur un dépot privé de l'UM [10].

5.2 EternityII

Pré-requis : Afin de pouvoir utiliser ce logiciel il est nécessaire de pouvoir compiler du code C++11 et utiliser la commande « `make` ».

Le programme **EternityII** est le programme principal développé le long de l'année et du stage. Le code source peux être récupéré ici : `https://gitlab.info-ufr.univ-montp2.fr/EternityII/EternityII` (dépôt privé).

Le dépot contient deux versions du programme :

- La version basique pour la résolution bruteforce
- la résolution smartforce

5.2.1 Bruteforce

La version bruteforce se trouve sur la branche v0.2.2-bruteforce [11]. L'interaction avec l'application se fait dans le main :

Listing 1 – Parcours bruteforce sur plusieurs tailles du plateau, en affichant tous les résultats des différents parcours

```
int main()
{
    for (int i = 4; i < 8; ++i) {
        ostringstream str;
         // nom du fichier d'entree contenant l'instance
        str << "assets/pieces_" << i << "x" << i << ".txt";
        FileIn file_in(str.str().c_str());
        Jeu jeu = file_in.initJeu(); // initialise le jeu
        // initialise la classe chargee de la resolution
        Generator generator(jeu);
        // Affiche les resultats pour tous les parcours
        generator.multipleGeneration();
    }
}
```

Pour effectuer un parcours spécifique, la dernière ligne doit être changée :

Listing 2 – Parcours bruteforce sur plusieurs tailles du plateau, en affichant tous les résultats du rowscan

```
int main()
{
    for (int i = 4; i < 8; ++i) {
```

```
4           ostringstream str;
5           // nom du fichier d'entree contenant l'instance
6           str << "assets/pieces_" << i << "x" << i << ".txt";
7           FileIn file_in(str.str().c_str());
8           Jeu jeu = file_in.initJeu(); // initialise le jeu
9           // initialise la classe chargee de la resolution
10          Generator generator(jeu);
11          // Affiche les resultats du parcours
12          generator.parcoursBruteForce(Generator::PARCOURS_ROW,0);
13      }
14  }
```

Les différentes constantes pour le parcours sont :

- `Generator::PARCOURS_ROW`
- `Generator::PARCOURS_DIAGONAL`
- `Generator::PARCOURS_SPIRALE_IN`
- `Generator::PARCOURS_SPIRALE_OUT`

Les fichiers d'instances se trouvent dans le dossier « assets ».

Le résultat est affiché en Markdown [12].

5.2.2 Smartforce

La résolution smartforce est en cours de développement, malgré le fait que le framework (outil de développement) soit fini. L'implémentation de CaPi est incomplète et renvoie des données non cohérentes.

Le fonctionnement du framework est explicité dans le manuel technique.

5.3 EternityII–corolle_generator

Le code de ce programme peux être trouvé ici : `https://gitlab.info-ufr.univ-montp2.fr/EternityII/EternityII-corolle_generator` (dépôt privé).

Le code source est inspiré du programme de bruteforce.

Utilisation

Compiler le programme en utilisant make (linux).

Le programme accepte deux arguments :

- Le fichier d'instance (qui peux être trouvé dans le dossier assets)
- Le hamming maximal à générer (2 par défaut).

Exemple 5.1. La commande `$ ./main pieces_10x10.txt 2`

Génèrera les corolles de toutes les zones en hamming 1 et 2 pour un 10.

Personnalisation

Il est malgré tout possible de forcer le programme à ne générer qu'une zone et un hamming spécifique.

En modifiant les lignes (10-14) du fichier main.cpp :

```
int main(int argc, const char *argv[])
{

    if (argc > 1) {
        string str(argv[1]);
        FileIn file_in(str);
        Jeu jeu = file_in.initJeu();
        Generator generator(jeu);

        if (argc > 2) {
            generator.multipleGeneration(atoi(argv[2]));
        } else {
            generator.multipleGeneration(2); //Genere toutes les corolles
                ↪ possible pour une taille de plateau
        }

        return 0;
    }

    cerr << "The input file was not specified" << endl;

    return 1;

}
```

en

```
generator.initGeneration(0,0,1) // x,y,hamming
```

5.4 EternityII–cardinality_counter

Le code de ce programme peux être trouvé ici : `https://gitlab.info-ufr.univ-montp2.fr/EternityII/EternityII-cardinality_counter` (dépôt privé).

Ce programme est en python, il compte le nombre de pièces uniques à chaque position dans la corolle. Il prends deux arguments :

- Le fichier corolle.
- (optionnel) « –o » sauvegarde le résultat dans un fichier du même nom avec « stat_ » préfixé

Exemple 5.2. `$ python main.py corolle.txt -o` Traite le ficher et sauvegarde les comptes dans un fichier nommé `stat_corolle.txt`

5.5 EternityII–corolle_rotator

Le code de ce programme peux être trouvé ici : `https://gitlab.info-ufr.univ-montp2.fr/EternityII/EternityII-corolle_rotator` (dépôt privé).

Description

Génère un fichier de la corolle à la rotation voulue, en fonction du fichier initial.

Usage

```
$ python main.py file_path [-o output_file, -r rotation]
```

- `file_path` est le chemin vers le fichier de corolle
- `-o` ou `--output` est le chemin vers le fichier généré
- `-r` ou `--rotation` est la rotation souhaitée (entre 0 et 3)

Remarque. si `-o` est spécifié, alors `-r` doit l'être aussi.

Remarque. si `-r` n'est pas spécifié, alors il générera les 3 rotation complémentaire au fichier

Le fichier de sortie est généré dans le même répertoire que le fichier d'entrée.

6 Manuel Technique

Dans cette partie, sera expliqué le fonctionnement de chaque application, en particulier, le framework développé pour la smartforce.

Le but est de fournir assez d'information afin de faciliter le maintient et la manipulation des applications développées.

6.1 EternityII (bruteforce)

Cette application a été bâtie de façon à pouvoir rapidement intégrer de nouvelles méthodes de parcours. Elle intègre aussi différents outils de manipulation de fichiers.

Son fonctionnement est simple. Chaque classe accomplit une certaine tâche spécifique [13].

D'abord, on charge le fichier d'instance (main.cpp) grâce à la classe `FileIn`.

Une fois les données récupérés du fichier, le `Jeu` est initialisé (les différentes `Pieces` ainsi que le plateau).

Le jeu est ensuite envoyé à `Generator` qui va initialiser les données nécessaires à la résolution.

Grâce à la fonction `Genetrator::multipleGeneration` ou directement `G::parcoursBruteforce`, on définit le type de parcours. En fonction du type de parcours, le chemin est initialisé par `G::coordinatesCreator`.

Une fois le chemin sur le plateau initialisé (voir Figure 16), le solveur (`G::generationRecursive`) est lancé . Celui-ci va, en fonction des coordonnées, tester si les pièces peuvent être placés (les pièces de coins ne seront testés que sur les coins, ...).

Lorsqu'une solution est trouvée, l'évenement `G::solutionFoundEvent` est déclenché.

6.2 EternityII–corolle_generator

Le générateur de corolle est directement inspiré de l'application de bruteforce. Au lieu de résoudre tout le plateau, il résout une sous-partie de celui-ci (une corolle).

Lorsqu'une corolle est trouvée, au lieu de déclencher l'évènement, il va copier les pièces dans un objet `Corolle`, la frontière de la corolles est aussi déterminée à ce stade. La corolle et la frontière sont envoyés à la fonction `Generator::writeInFile`.

Si un fichier est déjà ouvert, celle-ci vérifie si ces données sont identiques :

- si la taille du plateau est identique
- le hamming de la corolle
- la pièce centrale (et son orientation)
- la zone de la corolle (représentée sous forme de coordonnées (x, y))

Si aucun fichier n'est ouvert ou que l'une de ces données est différente, un nouvel objet `FileOut` est créé et remplace le précédent.

`FileOut` va alors créer ou ouvrir (en supprimant le contenu) le fichier et écrire 4 lignes contenant toutes les informations de la corolle.

Les deux premières lignes sont sous cette forme :

Exemple 6.1.

```
# taille rotation_corolle no_piece rotation x y hamming nb_pieces
# 6 0 0 0 0 0 1 3
```

Sera ensuite écrit la ligne contenant les information de la corolle :

Exemple 6.2. `0:0;5:1;6:0;;|3;4;5;6;7;;;;;;;0`

Si les données du fichier sont identiques, alors la corolle est ajoutée au fichier grâce à la fonction `FileOut ↪ ::put`.

6.3 EternityII–cardinality_counter

On parcours d'abord le fichier contenant les corolles (`count()`) comptant le nombre de pièces uniques à chaque position (`count_pieces()`).

Vient ensuite `put_on_plate()` qui se charge de convertir chaque position en coordonnées (x, y) et en y mettant le nombre d'occurences.

Enfin `display_plate()` qui affiche ou enregistre, en fonction des options, le plateau avec les occurences.

6.4 EternityII–corolle_rotator

Lorsqu'une corolle est tournée, non seulement les pièces sont tournées, mais l'ordre de parcours change aussi (voir Figure 37 et Figure 29). Par conséquent, il suffit non seulement de tourner les pièces `pieces_rotate()` mais aussi de « shifter » les pièces.

Exemple 6.3. Prenons une corolle en $(1, 1)$ (Z5 dans la Figure 35). on obtient donc :

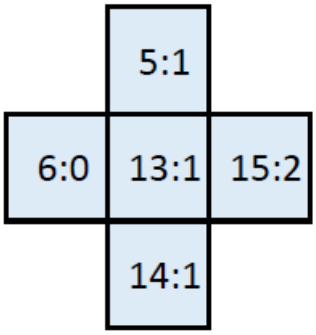

FIGURE 48 – Exemple de corolle de hamming 1 en $(1, 1)$

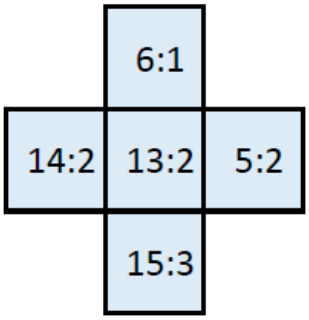

FIGURE 49 – Exemple de corolle de hamming 1 en $(5, 1)$

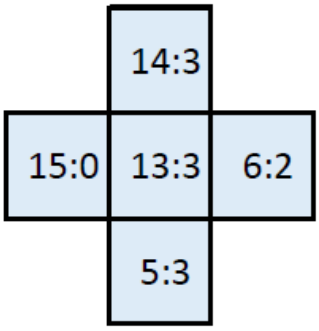

FIGURE 50 – Exemple de corolle de hamming 1 en $(5, 5)$

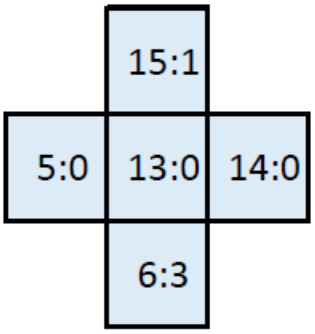

FIGURE 51 – Exemple de corolle de hamming 1 en $(1, 5)$

Sous forme de texte, en excluant la frontière on à donc :

```
13:1;15:2;14:1; 6:0; 5:1|...
13:1; 5:2;15:3;14:2; 6:1|...
13:1; 6:2; 5:3;15:0;14:3|...
13:1;14:0; 6:3; 5:0;15:1|...
```

On remarque que les 4 dernières pièces (appartenant au hamming 1) bouclent en se décalent de 1 à chaque fois (d'où le fait de « shifter » les cases).

Ce comportement marche sur tous les Hammings. Les pièces appartenant au hamming 2 de la corolle « shiftent » de 2, et ainsi de suite.

Cette action est gérée par `pieces_shift` pour les pièces et `color_shift` pour la frontière.

6.5 EternityII(smartforce)

Le code source se trouve sur : `https://gitlab.info-ufr.univ-montp2.fr/EternityII/EternityII/commits/feature/framework_refactoring`

Cette application est composée de deux parties :

- le core de l'application (le framework)
- l'app qui contient les différentes données nécessaires à la résolution

Cette application à été mise en place pour plusieurs raisons :
- Pouvoir organiser un grand nombre de données
- interconnecter les données entre eux pour optimiser la mise à jour des informations
- fournir une platforme modulaire afin de pouvoir facilement intégrer de nouvelles données
- mettre en place des systèmes de résolutions alternatifs
- implémenter facilement et rapidement de nouvelles façon de parcourir l'arbre

6.5.1 Principe

Avant d'aborder l'implémentation, il est important de comprendre le principe de l'application.

Le framework est décomposé en 4 parties distinctes :

1. Les Modèles
2. Les Contraintes
3. Le « PathFinder »
4. Le Solveur

Les contraintes Les modèles sont contraints par les contraintes, ceux-ci sont chargés de la propagation des données entre les modèles.

Les modèles Un modèle est un objet chargé de la manipulation d'une donnée lors de la résolution du problème. Il avertit les modèles auquel il est connecté lors d'un changement utile dans ses données via les contraintes.

Le Pathfinder Le PathFinder est connecté à des modèles de référence. Grâce à eux, il détermine le choix de valeur et le choix de variable.

Le solveur Le solveur progresse dans l'arbre, à chaque nœud il récupère grâce au PathFinder le nœud suivant à atteindre. Il envoie ensuite la décision aux modèles de références. Si le Pathfinder n'en trouve pas, il envoie alors un ordre de rollback vers le nœud précédent.

6.5.2 Fonctionnement

Solveur et PathFinder Le **solveur** à un rôle simple, c'est celui qui donne l'ordre de continuer ou non, c'est aussi celui qui est chargé de parcourir l'arbre des possibilités. Il utilise les ordres suivants :

resolve qui lance la résolution du problème

resolve(profondeur) qui détermine le choix de variable (utilise le pathfinder)

resolve(variable, profondeur) teste récursivement toutes les valeurs et rappelle `resolve(profondeur +1)`. À la fin de résolution des noeuds fils, un rollback partiel est effectué (on dénie le nœud fils qui vient de se terminer). Une fois que toutes les valeurs ont été testés pour la variable actuelle, on fait un rollback total pour revenir au nœuds parent (qui déniera le nœuds actuel comme choix de variable).

Il contient deux modèles qui sont les points d'entrée vers les autres modèles. C'est à ces deux modèles que les données relatives aux choix de variable et de valeur vont être envoyées.

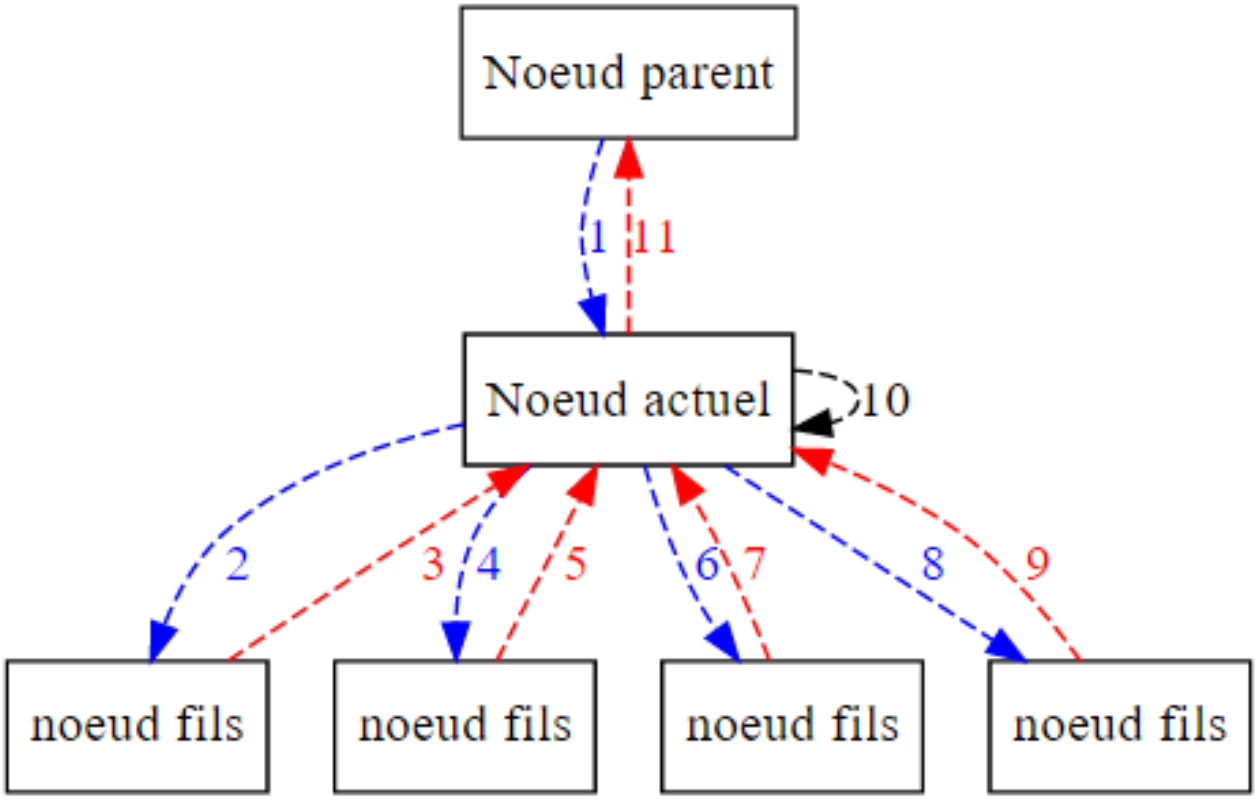

FIGURE 52 – Fonctionnement d'un solveur.
En bleu, la progression d'un noeud à l'autre.
En rouge, un rollback partiel.
En noir, un rollback total.

Le **pathfinder** aide le solveur à prendre des décisions. Il est associés à classes, l'un pour le choix de variable (ex : cases en rowscan), l'autre pour le choix de valeur (ex : pièces en lexico). Il contient 4 ordres :

hasNextVariable qui renvoi vrai s'il existe une variable à une profondeur donnée.

nextVariable qui renvoi une variable suivant la stratégie de variable utilisée.

hasNextValue renvoi vrai s'il existe une valeur à la variable et la profondeur actuelle

nextValue renvoi une valeur suivant la stratégie de utilisée.

Il est important de noter que le pathfinder et le solveur doivent utiliser les mêmes données pour les valeurs et variables, lorsque ceux-ci sont envoyés aux modèles.

Exemple 6.4. Afin d'éviter, par exemple, d'envoyer les identifiants de BoCo à un solveur CaPi.

6.5.3 Modèles et contraintes

Les modèles sont interconnectés entre eux grâce aux contraintes.

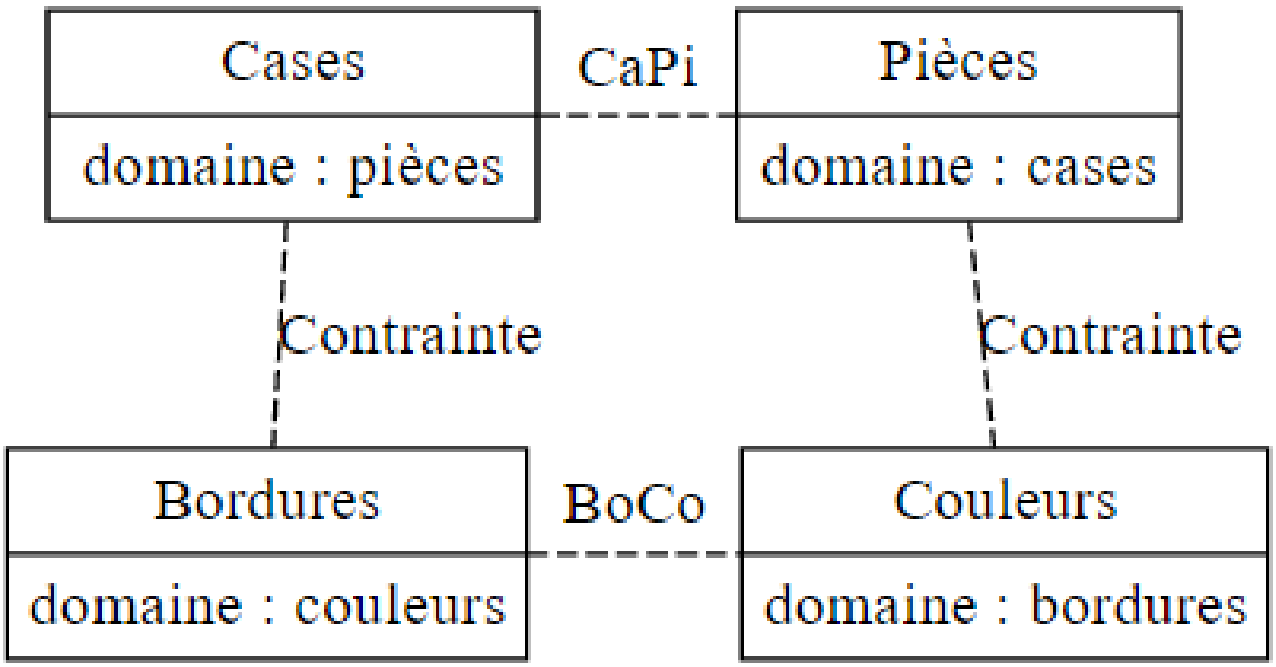

FIGURE 53 – Structure des modèles et des contraintes pour EternityII

Chaque modèle doit pouvoir effectuer ces actions :

allow autorise un couple de données (ex : CaPi) en tant que variable et valeur pour le nœud actuel. Cela déclenche 2 actions :
- dénie toutes les autres valeurs pour la variable et réciproquement.
- propage l'autorisation du couple aux autres modèles.

denyOne dénie le couple de valeur/variable pour la donnée actuelle. on peux aussi dénier définitivement un couple (utile lors d'un parcours en largeur de l'arbre).

addOne Utilisé lors de l'initialisation, il permet d'ajouter des données

rollback(profondeur, total/partiel) permet de rétablir le modèle à une profondeur antérieure partiellement ou totalement.

6.5.4 Fonctionnement global

L'ensemble des éléments est gérée par la classe EternityII, elle est aussi chargée de l'hébergement de tous les autres objets.

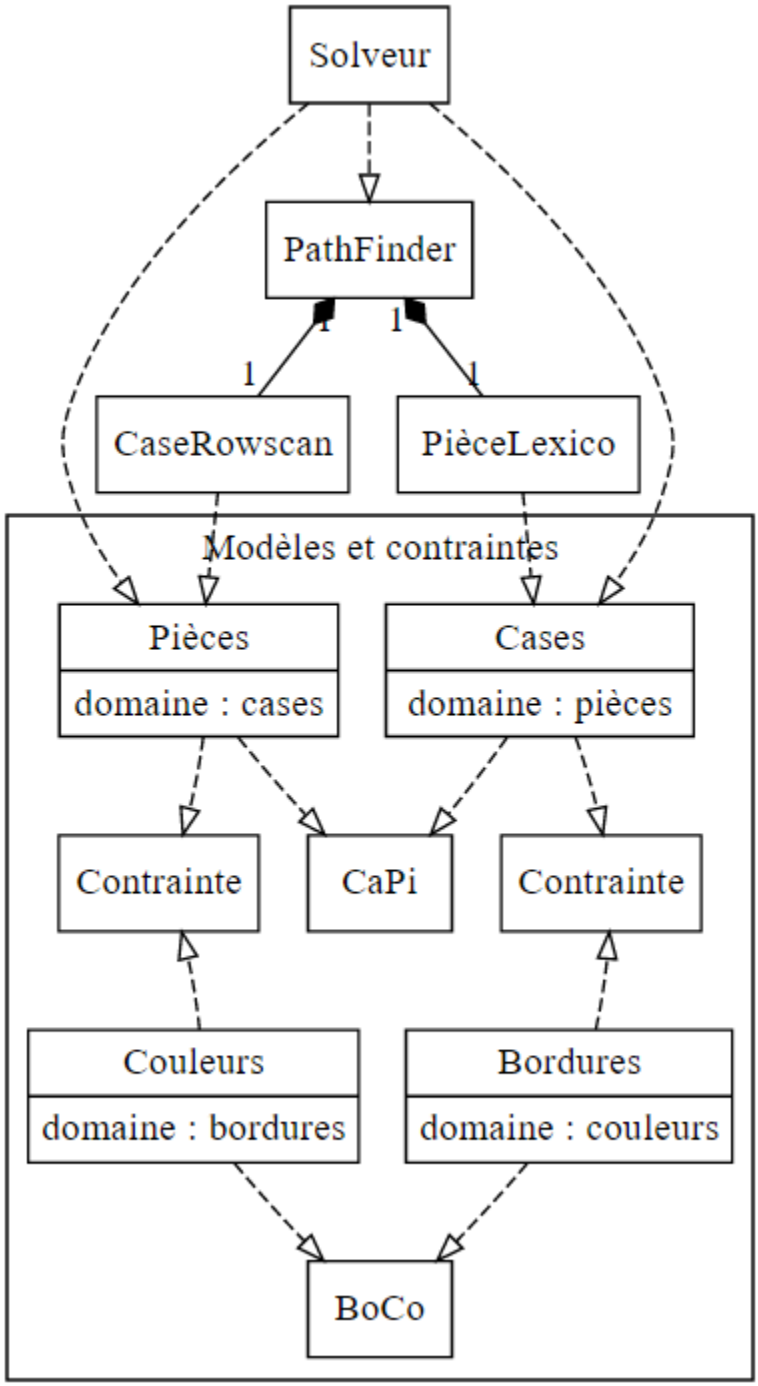

FIGURE 54 – Modèle simplifié du fonctionnement du framework

Le solveur demande et envoie le couple CaPi depuis le pathfinder aux modèles d'entrées (CaPi), ceux-ci sont chargés de propager l'information aux autres modèles via les contraintes.

6.5.5 Conseils pour le développement de l'application

Tout solveur doit hériter de core/SolverInterface.h

Tout choix de variable doit implémenter VariableInterface

Tout choix de valeur doit implémenter ValueInterface

Toute contrainte doit implémenter ConstraintInterface

Tout modèle doit implémenter ModelInterface

Toute donnée transmise d'une classe à l'autre doit implémenter DataInterface

La création de toutes les classes ont lieu dans `EternityII::bootstrap`

Tous les fichiers liées à l'application doivent se trouver dans le dossier app/.

Certains commentaires sont à lire lors de toute modification, ils sont préfixés par différents mots-clés :

unused : la fonction n'est pas utilisée
entrypoint : lire le commentaire si le point d'entrée à changé
minimal importance : La tache à faire n'est pas indispensable au fonctionnement de l'application
advice : indique la marche à suivre
dangerous : la fonction ou l'algorithme est fragile (dangereuse à utiliser ou a modifier).

7 Resultats

Dans cette partie, nous verrons les résultats et déductions qui ont été faites à partir des différentes approches. Elle évoquera aussi ce qui à été accompli et ce qui reste à faire.

Les résultats et déductions seront cités par ordre chronologique, il est donc normal que certaines approches soient invalidés.

L'implémentation et le fonctionnement interne des application sont explicités dans le manuel technique, malgré tout, certaines implémentations seront expliquées ici pour justifier les décisions prises durant le stage.

De plus, grâce au développement en parallèle d'un solveur en programmation par contraintes [14] par Mr Bourreau nous apporte un certain recul sur les données reçus.

7.1 Bruteforce

Les résultats de la bruteforce suivent ce qui avait été prédit. Malgré un certain niveau d'optimisation du programme, la plus grande instance résolue est de taille 6.

Le puzzle de taille 7 n'as pas pu être parcouru en 9 heures de calcul. Malgré tout, la première solution de l'instance se trouvait au nœud 10548049 et à été trouvée en 31.7344 secondes en rowscan.

D'après les résultats le choix de variable le plus performant est le « rowscan » [sous-section A.1]. On remarque aussi que le nombre de nœuds total et le temps écoulé augmente de façon exponentielle en fonction de la taille de l'instance.

Note. La programmation par contrainte à l'air d'être plus performante que le parcours naïf en bruteforce d'après la Figure 58.

7.2 Smartforce

Les valeurs étalons pour les choix de variables dynamiques (optimiste/pessimiste) qui ont été implémentés sur un modèle **CaPi**, elles se sont avérées moins performantes qu'un parcours basique.

La raison est simple : il est possible de recréer le même cas en parcourant de différentes façons. Par conséquent, le nombre de nœuds augmente sans pour autant parcourir de nouvelles possibilités.

Exemple 7.1. Soit un plateau composé de deux cases et par conséquent, de deux pièces. Si je pose d'abord ma pièce sur la première case, puis sur la deuxième. Le résultat sera le même que si je pose d'abord l'autre pièce sur la deuxième case puis que je fasse la première.

Donc, j'ai deux façons de trouver le même résultat.

Le choix de valeur dynamique, décident du chemin à chaque nœuds, il fait plusieurs fois la même combinaison, mais avec un chemin différent.

Les tests effectués sur les choix de valeur dynamiques ont obtenus les mêmes résultats que pour le **rowscan** car aucune mise à jour par propagation n'a pas été implémentée pour CaPi (non performant).

Le modèle **BoCo** est en cours d'implémentation, mais ne fournit pas des résultats pertinent. Le problème se trouve dans le mécanisme de mise à jour par propagation.

7.3 Corolles

Afin de pouvoir assimiler le fonctionnement des corolles dans la pratique, il est important de tenir compte de la quantité volumineuse d'information que génère ce modèle (voir sous-section A.2).

A titre d'exemple, le nombre de corolles en hamming 2 pour une instance de taille 6 (non rotationnées) est de 850694772. Avec un stockage non optimisé des corolles, l'espace occupé par ces corolles s'élève à environ 90GB. Par extension, on peux en déduire que pour une instance de taille 7, l'espace occupé par l'ensemble des corolles avoisinera au moins 1TB (en prenant la valeur de la Tableau 7 comme valeur max, et en supposant que l'espace occupé par une corolle est la même).

Ce stockage avait un défaut important, il n'était pas optimisé pour le stockage mais pour pouvoir manipuler avec aise les corolles.

Une corolle etait stockée de cette manière : chaque pièce était identifiée par un numéro et sa rotation, l'ensemble des pièces était séparés par un charactère (;) et disposés dans un ordre spécifique (voir Figure 29). Suivait ensuite la frontière, composée de couleurs qui étaient aussi séparés par un charactère (;). Évidemment, si la corolle etait réduite, les cases et frontières manquantes étaient remplacées par un vide.

Exemple 7.2. `0:0;5:1;6:0;;|3;4;5;6;7;;;;;;;0`

Une corolle étant une sous-partie du plateau, un ensemble de corolles peux être représenté sous forme d'arbre. Lors de la création de la corolle, on parcours tout simplement l'arbre en profondeur. Ainsi, dans le fichier, deux lignes adjacentes se ressemblent fortement.

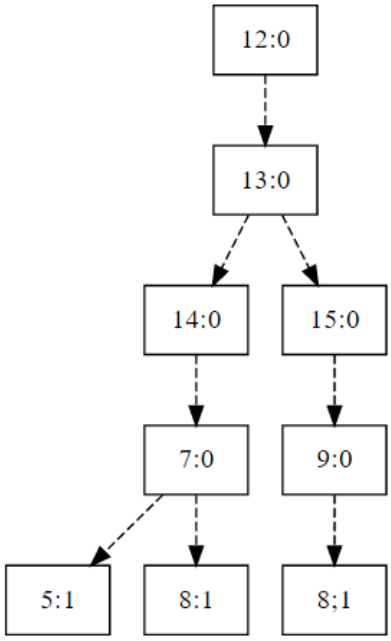

FIGURE 55 – Représentation sous forme d'arbre d'un exemple de corolles possibles

Exemple 7.3.

```
1 0:0;5:1;6:0;;|3;4;5;6;0;;;;;;;0
2 0:0;5:1;7:0;;|3;4;5;3;0;;;;;;;0
```

On peux donc optimiser en spécifiant à partir quelle pièce se fait l'optimisation.

Nous obtenons donc :

Exemple 7.4.

```
0>0:0;5:1;6:0;;|3;4;5;6;0;;;;;;;;0
1>7:0;;|3;4;5;3;0;;;;;;;;0
```

Grâce à ce procédé, le fichier de 90GB est passé à 20GB, ce qui nous donne une optimisation de 70% environ. Ceci étant, il est intéressant de noter que la compression de ces fichier grâce à LZMA [15] permet de réduire de 98% le volume du fichier. Mais la compression exige un certain délai afin de pouvoir exploiter les fichiers compréssés, ce qui limite son utilisation.

Il existe une dernière approche, qui se base sur un constant simple : dans une corolle de hamming 1, les pièces adjacentes à la pièce centrale sont indépendantes les uns des autres. Il suffit donc de lister à chaque position les pièces qui peuvent y être posés, puis lorsqu'il y a nécessité de créer une corolle, parcourir les positions en fixant une pièce. Toutes les corolles sont possibles, tant que le chemin ne possède pas deux fois la même pièce.

Exemple 7.5. Reprenons l'exemple précédent en y rajoutant une ligne :

nous avons donc

Positions	0	1	2
	0 : 0	5 : 1	6 : 0
	0 : 0	5 : 1	7 : 0
	0 : 0	6 : 1	7 : 0

il suffit donc d'enregistrer dans le fichier :

```
0:0;5:1;6:0
   ;6:1;7:0
```

Tant que l'on ne prend pas `0:0;6:1;6:0` tous les autres cas sont possibles.

Plus théoriquement, on fait un produit cartésien entre les ensembles de chaque position et l'on en obtient une corolle.

Le soucis, c'est que cette méthode fonctionne en hamming 1 car les pièces de H1 sont indépendantes de les unes des autres. En H2, l'approche est plus complexe : les pièces de H2 sont dépendants des pièces de H1. Or, on vient de déterminer un moyen de stocker les corolles. Il suffit de spécifier quel chemin emprunter en H1 (l'identifier) puis d'appliquer la même méthode des ensembles, mais pour les pièces de H2.

Ainsi, on peux obtenir, en minimisant la taille des fichiers, toutes les corolles possibles.

Malheureusement cette dernière approche n'as pas pu être mise en place faute de temps.

7.4 Liens dansants [16]

Le principe des liens dansants est à la fois complexe au niveau théorique que pratique. L'intérêt de cette approche est de réduire la quantité de données à consulter lors du parcours de l'arbre sans pour autant supprimer ou libérer ces données (pour pouvoir les réutiliser plus tard). Son principe est simple, il repose sur une base de liste doublement chainée : un élément est relié à l'élément suivant et précédent. Tous les éléments sont aussi répertoriés dans un tableau. Lorsque l'on souhaite masquer un élément du parcours, il suffit de chainer l'élément qui le précède à celui qui le suit (et vice-versa). De cette façon, l'élément actuel est ignoré car il n'est plus dans la chaine, mais, il est toujours répertorié. Lorsque l'on souhaite le remettre dans la chaine, il suffit de lire quel est l'élément qui le suit, puis de chainer cet élément à soi-même (pareil pour l'élément qui le précède ; il faut se chainer comme l'élément qui le suit).

Définition 7.1. Soit $S[x]$ et $P[x]$ pointant respectivement vers le successeur et le prédécesseur d'un élément. L'opération ci-dessous supprime un l'élément :

$$L[R[x]] \leftarrow L[x], R[L[x]] \leftarrow R[x]$$

Et à l'inverse, cette opération le rétablit :

$$L[R[x]] \leftarrow x, R[L[x]] \leftarrow x$$

Son emploi est très bénéfique dans des application de parcours d'arbre. Mais son implémentation peux être faite à plusieurs niveaux : la liaison des domaines dans les différents modèles (CaPi et BoCo) mais surtout dans le modèle des corolles.

On chaine une corolle C avec la corolle qui la précède et qui la suit (principe des liens dansants). On fait la même chose avec les pièces qui composent C : on chaine la pièce avec la position suivante et précédente de la corolle (on « met » C sous forme de liens dansants). Ensuite, on chaine chaque pièce avec la prochaine (et précédente) occurrence de cette pièce (à la même position) dans la liste des corolles. Au final on obtient une liste 6-tuplement chainée.

Grâce à cette quantité de liaison, lors de la pose d'une corolle, on peux rapidement l'enlever du parcours, on peux aussi dénier toutes les corolles qui ont des occurrences des pièces déjà posées. Toutes ces mises à jour sont fait en k étapes, k représentant le nombre de corolles à dénier.

7.5 GPU

Une fois le système de corolles et de liens dansants mis en place, le but était d'utiliser les capacités d'un GPU (carte graphique) afin de paralléliser les calculs. En effet, le GPU se distingue par sa quantité astronomique de cœurs, ce qui le rends imbattable sur les problèmes où les calculs peuvent être parallélisés.

8 Conclusion

8.1 Rétrospective

EternityII est un sujet qui m'a beaucoup passionné car le fait d'utiliser une grande quantité de données pour vaincre un problème combinatoire m'a paru très intéressante, mais il m'a fallu un certain temps d'adaptation pour assimiler la complexité de certaines approches. Il est difficile de se représenter à quel point c'est complexe. Heureusement, grâce aux données que l'on à trouvé, on peux maintenant savoir que le problème ne peux absolument pas être pris à la légère. Malgré le travail effectué (notamment de débroussaillage) j'aurais aimé pouvoir investir plus de temps dans ce défi.

En un an de TER et deux mois de stage, nous n'avons pas réussi à résoudre le puzzle de taille 8 (ou un parcours total du 7x7), l'objectif initial était le 10x10. Ce qui nous permet de comprendre pourquoi personne n'à pu résoudre le 10x10 en pratiquement 9ans.

Malgré cela, je suis très satisfait du travail que j'ai effectué et de tout ce que j'ai pu apprendre. J'ai essayé de partager ce que j'ai appris et compris durant ce stage.

8.2 L'avenir d'EternityII

Les outils présentés et développés par mes soins (grâce à l'aide de bon nombre de personnes) sont destinés à être réutilisés. Même si l'objectif final n'a pas pu être accompli, on se rapproche doucement de la solution. Comme pour le principe de la smartforce, c'est en accumulant des données, de la connaissance et du savoir que l'on pourra un jour résoudre les problèmes de ce monde.

Annexes

A Résultats

A.1 Bruteforce

Type Parcours	Première solution		Toutes les solutions		Nombre de solutions
	Nombre de nœuds	Temps (sec)	Nombre de nœuds	Temps (sec)	
row	166	0,000201	1202	0,001595	9
diagonal	457	0,000553	2126	0,002564	9
spirale_in	1357	0,001518	3216	0,003629	9
spirale_out	129	0,000158	881	0,001084	9

TABLE 1 – Résultats de la bruteforce pour une instance de taille 4

Type Parcours	Première solution		Toutes les solutions		Nombre de solutions
	Nombre de nœuds	Temps (sec)	Nombre de nœuds	Temps (sec)	
row	10194	0,02347	23759	0,056545	4
diagonal	56767	0,119206	75486	0,160089	4
spirale_in	33702	0,086007	61931	0,167123	4
spirale_out	66724	0,14307	84248	0,181349	4

TABLE 2 – Résultats de la bruteforce pour une instance de taille 5

Type Parcours	Première solution		Toutes les solutions		Nombre de solutions
	Nombre de nœuds	Temps (sec)	Nombre de nœuds	Temps (sec)	
row	145233	0.505312	16004458	36.9561	65
diagonal	2212939	4.43591	115333679	276.519	65
spirale_in	5048507	16.2233	97937587	364.801	65
spirale_out	637196	1.42648	103645373	214.629	65

TABLE 3 – Résultats de la bruteforce pour une instance de taille 6

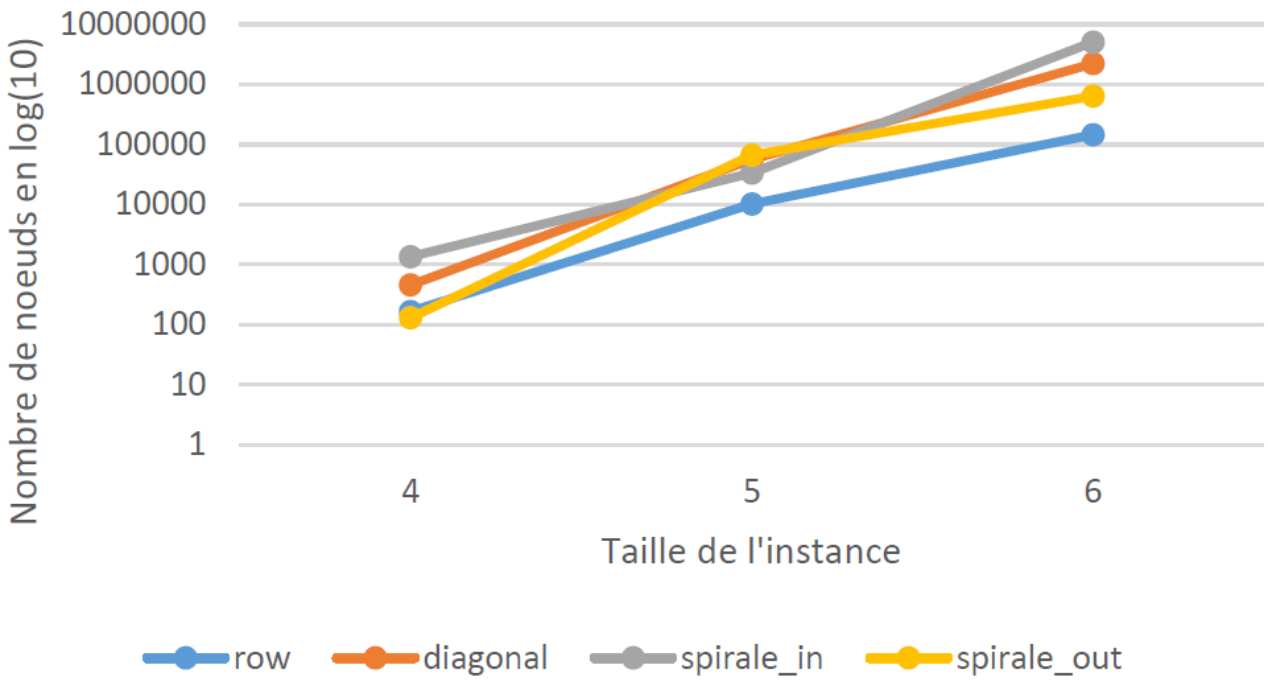

FIGURE 56 – Nombre de nœuds à la première solution en fonction de la taille de l'instance

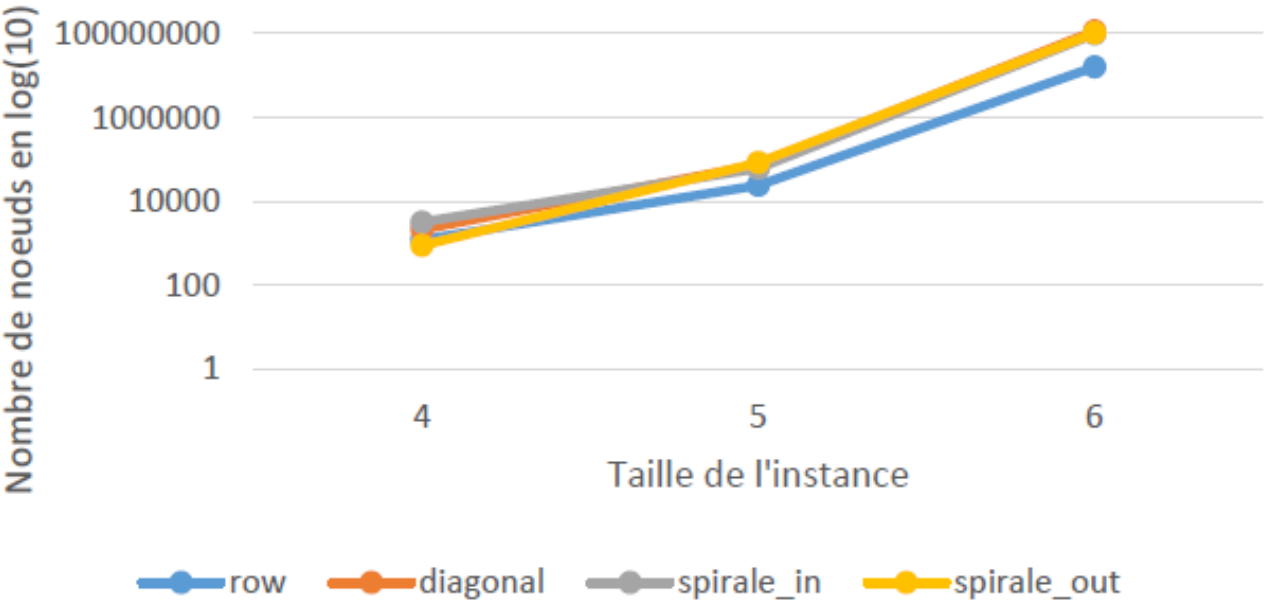

FIGURE 57 – Nombre total de nœuds en fonction de la taille de l'instance

Taille de l'instance	Toutes les solutions	
	Nombre de nœuds	Temps (sec)
4	72	0,14
5	869	0,58
6	351000	120

TABLE 4 – Résultats de la bruteforce (rowscan) en programmation par contrainte

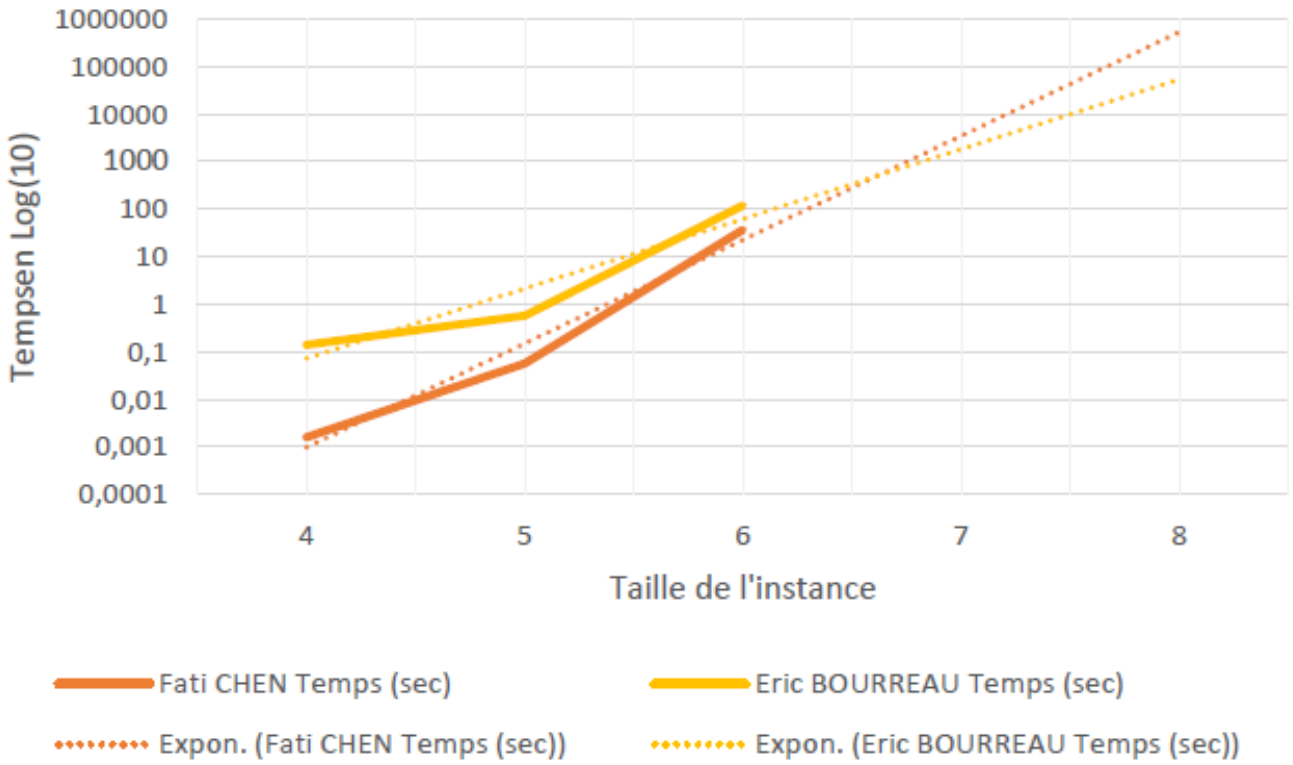

FIGURE 58 – Temps nécessaire au parcours complet de l'arbre

A.2 Smartforce

Nombre de corolles

Zones (x|y)	4	5	6	7	8	9	10
0|0	56	73	124	173	213	160	202
1|0	294	548	1452	2835	3824	3790	5925
2|0		1287	4665	12470	23471	26205	48001
3|0							
0|1	294	616	1378	2810	3863	3805	5886
1|1	797	13810	72092	235507	312594	735818	1422453
2|1		28194	249070	1051601	1873013	4910491	11127769
3|1							
0|2							
1|2							
2|2		45288	732248	4303016	10243916	30666984	82233160
3|2							
0|3							
1|3							
2|3							
3|3							
	1441	**89816**	**1061029**	**5608412**	**12460894**	**36347253**	**94843396**

TABLE 5 – Quantité de corolles de hamming 1 en fonction de la taille de l'instance

Nombre de nœuds

Zones (x|y)	4	5	6	7	8	9	10
0|0	76	167	276	377	460	350	437
1|0	477	1541	3987	7882	10988	11835	18454
2|0		3019	10413	27152	50282	56665	102606
3|0							
0|1	402	1430	3069	6111	8278	8223	12553
1|1	1356	33579	171409	550771	748091	1734204	3336615
2|1		62347	525365	2182959	3868929	10083550	22747247
3|1							
0|2							
1|2						11499792	25951789
2|2		102341	1552229	8949273	21186059	63012778	168162571
3|2							
0|3							
1|3							
2|3							
3|3							
	2311	**204424**	**2266748**	**11724525**	**25873087**	**74907605**	**194380483**

TABLE 6 – Nombre de nœuds afin de déterminer les corolles de hamming 1

Nombre de corolles

Zones (x|y)	4	5	6	7	8	9	10
0|0	708	1719	6353	16207	20458	15593	29339
1|0	1268	19492	151387	620172	732580	711438	1592663
2|0		37295	1450492	8645614	13495885	15781152	44295837
3|0				34047965	74307364	100670165	340437373
0|1	1301	20038	150788	612641	753429	723436	1631595
1|1	320	109610	4394668	40911075	39826328	63184631	201669254
2|1		284922	85986347	2003391758	?	?	?
3|1				7744390324	?	?	?
0|2		37295	1354518	8592952	13514064	96595319	43957175
1|2		284922	87741140	?	?	?	?
2|2		115104	669459079	?	?	?	?
3|2				?	?	?	?
0|3				34047965	74307364	?	?
1|3							
2|3							
3|3				?	?	?	?
	3597	**910397**	**850694772**	**9841228708**	**142650108**	**277681734**	**633613236**

TABLE 7 – Quantité de corolles de hamming 2 en fonction de la taille de l'instance

Nombre de nœuds

Zones (x|y)	4	5	6	7	8	9	10
0|0	1235	4327	14884	36521	45612	15593	64291
1|0	3956	66795	494221	1947920	2539399	711438	5923686
2|0		141848	4425208	26542329	44089658	15781152	157898139
3|0				77347031	165712616	100670165	750181211
0|1	3836	52214	355962	1382805	1669066	723436	3592864
1|1	5454	835300	28339858	272914777	325801787	63184631	2718512429
2|1		1627756	347268489	7209889289	?	?	?
3|1				24323914694	?	?	?
0|2		141848	3199393	19394585	30010234	16141066	96595319
1|2		2184372	425818301	?	?	?	?
2|2		2317658	2647617373	?	?	?	?
3|2				?	?	?	?
0|3				77347031	165712616	?	?
1|3							
2|3							
3|3				?	?	?	?
	14481	**7372118**	**3457533689**	**31933369951**	**569868372**	**197227481**	**3732767939**

TABLE 8 – Nombre de nœuds afin de déterminer les corolles de hamming 2

Nombre de corolles

Zones (x|y)	4	5	6
0|0	1178	33248	432731
1|0	253	91167	13090178
2|0		67912	40377690
0|1	254	85814	13162247
1|1	56	33547	177550498
2|1		2720	227378442
0|2		67912	42936840
1|2		2720	222784993
2|2		48	63298607
	1741	**385088**	**801012226**

TABLE 9 – Quantité de corolles de hamming 3 en fonction de la taille de l'instance

Nombre de nœuds

Zones (x|y)	4	5	6
0|0	5946	96585	1071794
1|0	6295	445948	48934834
2|0		517247	174146773
0|1	7311	450476	33372427
1|1	5973	1240725	728227848
2|1		1732673	3155244298
0|2		517247	175258592
1|2		2306560	3063431244
2|2		2365209	6631113798
	25525	**9672670**	**14010801608**

TABLE 10 – Nombre de nœuds afin de déterminer les corolles de hamming 3

Références

[1] Mark WAINWRIGHT. *Prize specimens*. 1er jan. 2001. URL : https://plus.maths.org/content/os/issue13/features/eternity/index.

[2] Sunday Star TIMES, éd. *The maker of Eternity II*. 27 juil. 2007. URL : http://www.stuff.co.nz/sunday-star-times/features/feature-archive/older-features/51606/The-maker-of-Eternity-II.

[3] WIKIPEDIA. *Problème NP-complet — Wikipedia*. 2016. URL : https://fr.wikipedia.org/wiki/Probl%C3%A8me_NP-complet.

[4] Ludovic PATEY et Sylvain GRAVIER. « Eternity II et variantes ». In : (2010).

[5] Thierry BENOIST et Eric BOURREAU. « La programmation par contraintes à l'attaque d'Eternity II ». In : *JFPC 2008-Quatrièmes Journées Francophones de Programmation par Contraintes*. 2008, p. 105–114.

[6] Joffrey CUVILLIER et Rémi SZYMKOWIAK. « Résolution du jeu Eternity 2 avec les technologies SAT ». In : ().

[7] Marijn JH HEULE. « Solving edge-matching problems with satisfiability solvers ». In : *SAT* (2009), p. 69–82.

[8] Daniel B MURRAY et Scott W TEARE. « Probability of a tossed coin landing on edge ». In : *Physical Review E* 48.4 (1993), p. 2547.

[9] GIT. *Git Documentation*. URL : https://git-scm.com/.

[10] Fati CHEN. *Dépot des ressources EternityII — [Privé] Gitlab*. URL : https://gitlab.info-ufr.univ-montp2.fr/groups/EternityII.

[11] Fati CHEN. *Dépot des ressources EternityII — [Privé] Gitlab*. Version 0.2.2-bruteforce. URL : https://gitlab.info-ufr.univ-montp2.fr/EternityII/EternityII/tree/v0.2.2-bruteforce.

[12] GITHUB. *Mardown Documentation*. URL : https://guides.github.com/features/mastering-markdown/.

[13] WIKIPEDIA. *KISS Principle — Wikipedia*. 2016. URL : https://en.wikipedia.org/wiki/KISS_principle.

[14] Charles PRUD'HOMME, Jean-Guillaume FAGES et Xavier LORCA. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. 2015. URL : http://www.choco-solver.org.

[15] WIKIPEDIA. *Lempel–Ziv–Markov chain algorithm — Wikipedia*. 2016. URL : https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Markov_chain_algorithm.

[16] Donald E KNUTH. « Dancing links ». In : *arXiv preprint cs/0011047* (2000).