Université Nice SOPHIA ANTIPOLIS

Master 1 Informatique

# Travail d'Etude et de Recherche : Le Problème des fusiliers


**Auteurs**

Fulconis Angélique

Benouali Hamine

Bouhlel Oualid

Casanova Pierre

**Encadrants**

Verel Sébastien

Clergue Manuel


Le 26 mai 2008

# Table des matières


**1 Présentation générale du problème** **2**
- 1.1 Sujet . . . 2
  - 1.1.1 Problématique . . . 2
  - 1.1.2 Description du sujet . . . 3
  - 1.1.3 Définitions des automates cellulaires . . . 3
  - 1.1.4 Finalités . . . 4
- 1.2 Etat de l'art . . . 4
- 1.3 Gestion du projet . . . 7
  - 1.3.1 Planning . . . 7
  - 1.3.2 Problèmes rencontrés . . . 7
  - 1.3.3 Autres travaux réalisés . . . 7

**2 Description de l'étude : Les différentes approches** **8**
- 2.1 Algorithme à solution unique . . . 8
  - 2.1.1 Principe . . . 8
  - 2.1.2 Environnement et Algorithmes . . . 10
  - 2.1.3 Algorithmes . . . 12
  - 2.1.4 Résultats expérimentaux . . . 14
  - 2.1.5 Analyse comparative des différents algorithmes . . . 20
- 2.2 Algorithme évolutionnaire . . . 21
  - 2.2.1 Principe . . . 21
  - 2.2.2 Définitions . . . 21
  - 2.2.3 Environnement et algorithme . . . 24
  - 2.2.4 Implémentation . . . 24
  - 2.2.5 Résultats . . . 25
- 2.3 Backtracking . . . 27
  - 2.3.1 Principe . . . 27
  - 2.3.2 Algorithme . . . 28
  - 2.3.3 Implémentation . . . 30
  - 2.3.4 Résultats . . . 30
  - 2.3.5 Pistes explorées . . . 32
  - 2.3.6 Statistiques obtenus . . . 33
- 2.4 Approche par signaux . . . 35

2.4.1 Principe et définitions . . . . . 35
2.4.2 Formes caractéristiques attendues pour le problème des fusiliers : . . . . 38
2.5 Approches combinées . . . . . 39
2.5.1 Combinaison avec le backtracking . . . . . 39
2.5.2 Algorithme évolutionnaire et recherche tabou . . . . . 39
2.5.3 Résultats . . . . . 39

**3 Conclusion** **41**
3.1 Perspectives . . . . . 41
3.2 Intérêts du TER . . . . . 41

# Remerciements

Nous tenons à remercier nos encadrants M. VEREL et M. CLERGUE, qui nous ont accueillis de nombreuses fois dans leur bureau. Le temps qu'ils nous ont consacré et nos discussions sur le sujet nous ont aidés à nous orienter dans la résolution du problème.

# Chapitre 1

# Présentation générale du problème

Ce chapitre donne les définitions du problème lui-même et des concepts sur lesquels il repose, l'état de l'art à son sujet et l'exposé du travail demandé.

## 1.1 Sujet

Dans cette première partie nous détaillons la problématique et le sujet et donnons les définitions fondamentales.

### 1.1.1 Problématique

On constate l'essor grandissant de nouveaux types d'ordinateur : les machines dites parallèles qui possèdent plusieurs processeurs et une ou plusieurs mémoires. Il existe aujourd'hui une grande diversité de parallélismes pour lesquels beaucoup de travaux restent à faire. Ce projet concerne un type particulier de parallélisme : les automates cellulaires qu'avait inventé Von Neumann à la fin des années 40. L'automate cellulaire est un modèle pertinent pour l'étude des machines parallèles. En effet, il présente des atouts non négligeables :

- le nombre important de processeurs mis en parallèle.
- son fonctionnement synchrone qui permet de visualiser simplement la progression des calculs.

Le problème de la synchronisation d'une ligne de fusiliers est une manière d'aborder la programmation d'ordinateur massivement parallèle à fonctionnement synchrone. Ce TER permet en réalité de s'intéresser à certaines problématiques :

- la découverte d'une méthode de synchronisation de machines communiquant de manière locale,
- découverte qui se fait de manière automatique : possibilité de déterminer automatiquement des protocoles de communication locaux,
- étude de la faisabilité sur un modèle **simple et universel** de machines parallèles : l'automate cellulaire.

### 1.1.2 Description du sujet

Il s'agit de résoudre un problème classique de synchronisation dans le domaine des automates cellulaires (AC), proposé par J.Mihill en 1957 : comment synchroniser une ligne de fusiliers de façon à ce qu'ils se mettent à tirer ensemble, alors que l'ordre donné par le général depuis l'un des deux bords de l'escadron met un certain temps à se propager ?
Chaque fusilier est représenté par une cellule d'un automate cellulaire. Celles-ci peuvent être dans un nombre fini d'états parmi lesquels on trouve les états : repos (ou quiescent) , général (1), feu ou dans l'un des états intermédiaires.
Quel que soit le nombre d'états, le problème consiste à trouver des règles de transition permettant la synchronisation des cellules dans le même état « feu ». Ces règles d'inférence, définissant l'état futur de la cellule, sont de tailles 3, i.e. le voisinage de chaque cellule contient cette même cellule ainsi que celle qui est à sa droite et celle qui est à sa gauche (qui peuvent correspondre aux extrémités). Le but du problème est de trouver les règles de transition qui vont permettre d'obtenir une configuration dans laquelle toutes les cellules se trouvent ensemble et pour la première fois dans l'état feu. Une solution est optimale quand le temps de synchronisation est minimal, c'est à dire $2n-2$ pour un automate à $n$ cellules.

### 1.1.3 Définitions des automates cellulaires

Les automates cellulaires sont à la fois un modèle de système dynamique discret et un modèle de calcul. Un automate cellulaire consiste en une grille régulière de cellules pouvant chacune prendre à un instant donné un état parmi un ensemble fini. Le temps est également discret et l'état d'une cellule au temps $t+1$ est fonction de l'état au temps $t$ d'un nombre fini de cellules appelé son voisinage.

On appelle *Automate Cellulaire Linéaire Fini* (ACLiF) tout doublet $\mathcal{A} =< \mathcal{Q}, \delta >$, où :

- $\mathcal{Q}$ est un ensemble fini d'états, dans lequel les plus importantes sont :
  $q_l$ : dit état latent, quiescent ou de repos.
  $q_i$ : dit état initial, départ ou général (car il donne l'ordre).
  $q_f$ : dit état de feu ou de synchronisation.

  **Remarque :** On utilisera par la suite un état appelé état de bord : $q_b \notin \mathcal{Q}$ (seul état du système que l'on considèrera comme n'étant pas interne).

- $\delta$ est appelé fonction de transition locale et vérifie :

$$\delta : \mathcal{Q} \cup q_b \times \mathcal{Q} \times \mathcal{Q} \cup q_b \rightarrow \mathcal{Q}$$

- On impose que l'état $q_l$ vérifie la condition :

$$\delta(q_b, q_l, q_l) = \delta(q_l, q_l, q_l) = \delta(q_l, q_l, q_b) = q_l$$

**Définition**
Une *ligne de n automates cellulaires* (ACLiF), notée $\mathcal{A}_n$, est constituée par n machines $\mathcal{A}$ numérotées de 1 à $n$. Une ligne est appelée automate cellulaire fini de longueur n.

**Définition** Une *configuration* d'un ACLiF de longueur $n$ est une fonction :

$$\mathcal{C}^n : [1, n] \to \mathcal{Q}$$

**Définition** On appelle *fonction de transition globale*, la fonction induite par la fonction de transition locale $\delta$ sur $\mathcal{Q}^\star = \bigcup_{i=1}^{\infty} \mathcal{Q}^i$ :

$$\Delta : \mathcal{Q}^\star \to \mathcal{Q}^\star$$

### 1.1.4 Finalités

Ce problème qui jusqu'à présent a été étudié à la main, nous nous proposons de l'aborder de toute autre manière en développant des méthodes qui permettront de rechercher des solutions de façon automatique. Ce qui nous amènera aux finalités suivantes :

- l'analyse comparative des différentes méthodes existantes pour traiter ce genre de problèmes,
- le développement de nouveaux algorithmes de résolution du problème des fusiliers de façon générale et plus particulièrement pour cinq états (en temps minimal),
- la mise en place d'une page web permettant de visualiser les résultats obtenus,
- la création d'une bibliothèque d'algorithmes qui s'appliquent à la résolution du problème des fusiliers à p état,
- finalité pédagogique : apprendre les méthodes d'optimisation stochastique (métaheuristiques) et aborder un problème fondamental des systèmes dynamiques discrets,
- la résolution du problème pour six états si l'état d'avancement dans le planning nous le permet.

## 1.2 Etat de l'art

Cette partie présente les études précédentes ainsi que les prestations demandées.
En 1957 J.Myhill inventa le problème dit « Firing squad »(Ligne de fusiliers) mais ce n'est qu'en 1964 que E.F Moore publia le probléme des fusiliers :

Soit un $\mathcal{A}_n$ un ACLiF de taille $n$ dont toutes les cellules sont dans l'état quiescent à l'exception de la cellule d'index 1 qui est dans l'état général. Trouver une *fonction de transition globale* telle que toutes les cellules rentrent dans l'état "feu" pour la première fois et simultanément. Cette *fonction de transition globale* doit permettre la synchronisation pour tout $n$. Le nombre d'itérations optimal pour optenir la synchronisation est $n - 2$, le temps pour l'information de se

propager de gauche à droite puis de droite à gauche.

Un an plus tard Minsky et Mc Carthy résolvent ce problème avec la méthode « Diviser pour régner »avec une complexité en temps de $3n$.
La première solution non publiée en temps optimal fût découverte par E.Goto. L'automate comportait plusieurs milliers d'états.
Puis en 1967 Waksman et Balzer découvrirent parallèlement une solution optimale : l'une pour 16 états et l'autre pour 8 états.
Plus tard Balzer réussit à montrer que pour 4 états le problème n'avait pas de solution.
Puis en 1986, Mazoyer trouva une solution en temps minimal pour 6 états[MAZ87].
En 1993 Yunes a également effectué plusieurs travaux : en temps quelconque, il a démontré que pour 3 états il n'y a pas de solution et qu'il y en a pour 7 et 8 états[Yun93].

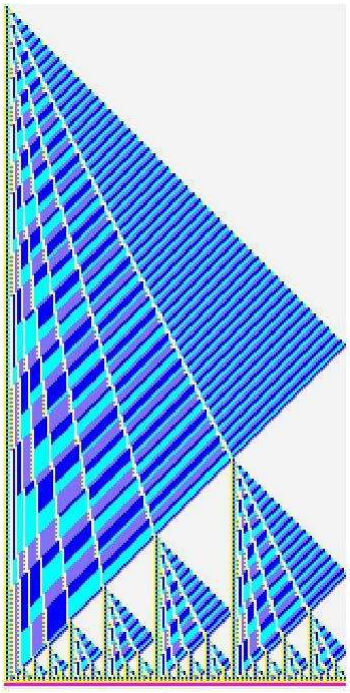

FIG. 1.1 – La solution de Mazoyer à 6 états.

**Récapitulatif :**

| Nombre d'états | Temps optimal | Temps quelconque |
|---|---|---|
| 3 états | Pas de solution Balzer | Pas de solution Yunes |
| 4 états | Pas de solution Balzer | Ouvert |
| 5 états | Ouvert | Ouvert |
| 6 états | Une seule solution Mazoyer | Ouvert |
| 7 états | Solution Mazoyer | Solution Yunes |
| 8 états | Solution Balzer-Mazoyer | Solution Yunes |

Jusqu'à présent, les solutions ont été trouvées « à la main ».
Enfin en 2007, Mirela Frandes a également travaillé sur ce problème en utilisant une nouvelle approche, et obtenu la synchronisation des $n$ fusiliers avec 5 états pour $n$ entre 2 et 15 en temps non optimal.

Pour 5 états, le problème reste ouvert : aucune solution n'a été trouvée que ce soit en temps optimal ou non. C'est la raison pour laquelle c'est à ce cas particulier que nous nous intéressons.

Nous assignons à chaque état une couleur qui lui sera propre et par laquelle il sera représenté sur les diagrammes espace-temps :

- le blanc pour l'état 0 (quiscent),
- le bleu pour l'état 1 (repos),
- le jaune pour l'état 2,
- le vert pour l'état 3,
- et le rouge pour l'état 4 (feu).

### Nature de prestations demandées

Au cours du TER, nous nous emploierons à réaliser les tâches suivantes :

- analyse comparative des différentes stratégies pour traiter ce genre de problème. Le but est de développer et d'analyser les différentes approches. On devra comparer ces approches de façon statistique sur la base d'un échantillon significatif d'expérimentations, en fonction des résultats qu'elles auront obtenus, mais aussi en fonction de leur facilité de conception. Celles-ci sont détaillées dans la section suivante. Pour cela, il est nécessaire d'implémenter les algorithmes et de comparer les résultats obtenus,
- mise en place d'une page web permettant de visualiser les résultats obtenus,
- création d'une archive d'algorithmes qui s'appliquent au problème des fusiliers à $p$ états.

## 1.3 Gestion du projet

Nous avons équilibré le temps alloué au développement et à l'expérimentation de chaque approche de manière à ce que les échantillons de statistique que nous disposons permettent de les comparer efficacement.

### 1.3.1 Planning

La mise en oeuvre et l'expérimentation des différentes approches ont été réalisées, ainsi que le site web.

Nous avons respecté les limites en ne consacrant pas trop de temps à développer une seule approche. Nous nous sommes tenus assez rigoureusement à notre planning au détail prés que nous avons consacré plus de temps que prévu à l'approche combinée et au site internet.

### 1.3.2 Problèmes rencontrés

Pour les approches par métaheuristique il a fallu se familiariser avec les librairies paridiseo et trouver une structure de données satisfaisante, qui permette de parcourir efficacement l'espace de recherche. Il a été nécessaire de redéfinir certaines classes contenues dans les librairies afin d'améliorer l'efficacité de la fonction d'évaluation. Nous avons rencontré des difficultés pour résoudre les problèmes de segmentation fault. Nous avons utilisé un debugger gdb avec la commande "bt" et supprimé de nombreux appels récursifs qui sont mal gérés en c++ pour résoudre ces problèmes.
Un autre problème a été l'espace mémoire. Pour palier celui-ci, nous avons rajouté des "deletes", mais le temps d'exécution a doublé pour n allant de 2 à 12 (1220 au lieu de 778). Pour éviter la création et suppression des tableaux trop fréquentes qui font perdre du temps, nous avons donc modifié l'implémentation de la pile : au lieu d'avoir un "stack" de tableau, nous avons un tableau à deux dimensions (96 correspond au nombre de règles et 5 correspond aux trois nombres formant la règle, à la valeur attribuée à la règle et au "n" où la règle a été crée). Cette nouvelle implémentation nous permet de savoir combien de règles sont utilisées. De plus par rapport à l'ancienne version, nous avons une trace de la pile et nous connaissons donc l'ordre d'apparition des règles.

### 1.3.3 Autres travaux réalisés

Nous avons crée un site internet sur lequel nous stockons tous les résultats obtenus. Ainsi la visualisation des résultats est plus évidente.

# Chapitre 2

# Description de l'étude : Les différentes approches

Dans le cadre de ce TER, quatre approches de résolution ont été envisagées :

## 2.1 Algorithme à solution unique

### 2.1.1 Principe

Les métaheuristiques à base de solution unique sont des algorithmes de recherche stochastique de voisinage. Ils partent d'une solution initiale (générée aléatoirement), puis itérativement, substituent la solution courante par une autre solution du voisinage, le plus souvent meilleure, jusqu'à ce qu'un critère d'arrêt soit vérifié.
Il existe plusieurs types de métaheuristique à base de solution unique telle que le Hill Climbing, la Recherche Tabou et le Recuit Simulé .
Nous définissons maintenant le principe de chaque algorithme.

#### Hill Climbing

Hill Climbing est une technique de recherche qui appartient à la famille des métaheuristiques de recherche locale. Cette technique consiste à sélectionner une solution aléatoire au départ (configuration initiale) et poursuivre la recherche en raffinant cette solution récursivement. A un certain point l'algorithme n'arrive pas à faire des améliorations, à ce moment l'algorithme utilise une class d'équivalence qui permet d'accepter des solutions de même performance que la solution courante en espérant trouver une meilleure solution ultérieurement.
Cette technique d'exploration permet de progresser dans l'espace de recherche palier par palier. Le hill climbing est limité par les maximums locaux mais il est souvent utilisé à cause de son efficacité.

**Recherche Tabou**

La recherche tabou [4] est une méthode de recherche locale avancée qui fait appel à un ensemble de règles et de mécanismes généraux pour guider la recherche vers les solutions le plus prometteuses. Un élément fondamental de la recherche tabou est l'utilisation d'une mémoire (liste tabou) à court terme, qui garde une certaine trace des dernières opérations effectuées.

Cette liste permet d'empêcher les blocages dans les minima locaux en interdisant de passer à nouveau sur des configurations de l'espace de recherche précédemment visitées.

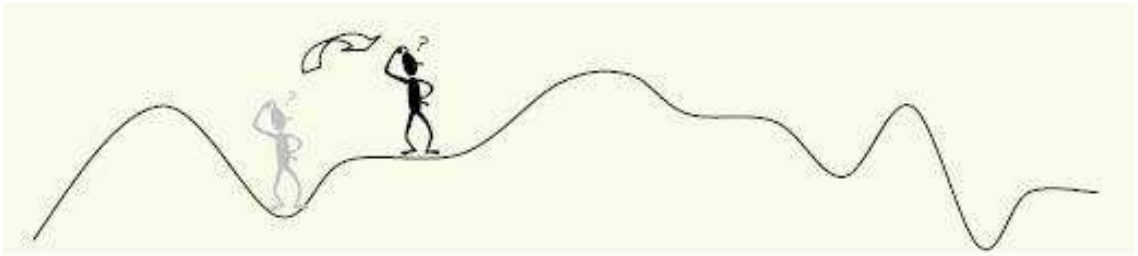

FIG. 2.1 – effectuer un mouvement

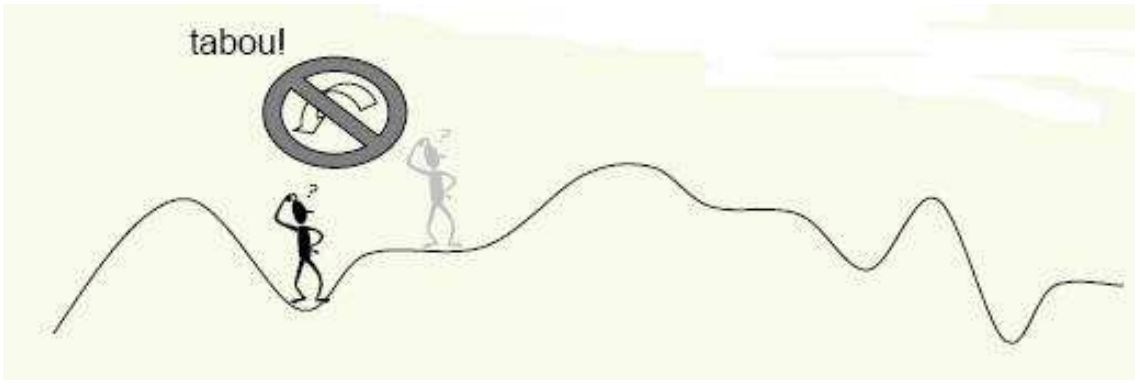


FIG. 2.2 – interdire un mouvement déjà effectué pendant un nombre d'itérations

Si on passe d'une configuration `s` à une configuration `s´` il faut interdire ce mouvement. Il sera donc tabou pendant un certain nombre d'itérations.

La durée de cette interdiction appelée longueur tabou (K), est l'un des paramètres importants de cette métaheuristique. Elle est liée à la nature du problème. Si sa valeur est trop élevée alors des solutions non visitées seront inaccessibles et la capacité de la méthode à exploiter le voisinage sera réduite. Inversement, si sa valeur est trop faible alors la méthode risque fortement d'être bloquée dans un minimum local. La longueur tabou permet donc d'éviter tous les cycles de longueur inférieure ou égale à k.

#### Recuit Simulé

La méthode du recuit simulé est inspirée de la physique des matériaux et plus spécialement de la métallurgie. Le recuit est une opération consistant à laisser refroidir lentement un métal pour améliorer ses qualités. L'idée physique est qu'un refroidissement trop brutal peut bloquer le métal dans un état peu favorable (alors qu'un refroidissement lent permettra aux molécules de s'agencer au mieux dans une configuration stable). C'est cette même idée qui est à la base du recuit simulé. Pour éviter que l'algorithme reste piégé dans des minima locaux, on fait en sorte que la température T = T(n) décroisse lentement en fonction du temps.

### 2.1.2 Environnement et Algorithmes

Dans cette section nous définissons les notions de base pour les méthodes métaheuristiques .

#### Fonction objectif

Pour comparer, en terme de qualité, deux configurations **s** et **s'** de **S**, on définit une fonction de coût (fitness), qui est en fait une relation d'ordre sur S.

Eval(s), est la fonction d'évaluation d'une solution. Cette fonction permet de donner la taille de l'automate synchronisé en appliquant la solution **s**.

Pour mieux comprendre cette fonction nous proposons l'exemple suivant :

Soit S une solution et b=Eval(S). Il faut que le nombre de cellules (fusiliers) à l'état feu obtenu en appliquant la solution sur une ligne de fusiliers de taille i, égale à i en temps optimal (2N-2) pour i de 1 à 5.

#### Définition du voisinage

Pour parcourir le bon espace de recherche du problème des fusiliers, on a défini un ensemble de voisinage qui est représenté par un tableau à deux dimensions de taille égale au nombre de Règles possibles (216) et au nombre d'états. Cette structure permet d'interdire ou autoriser un ou plusieurs états pour une règle donnée. Pour trouver une solution voisine, il suffit de trouver un mouvement voisin et de l'appliquer à la solution courante. Un mouvement est représenté par le numéro de la règle et un état. Un mouvement voisin est généré de la façon suivante :

- Si une règle peut prendre plusieurs états, il suffit de prendre un état qui n'été pas pris précédament.
- Si tous les états sont déjà pris on passe à la règle suivante.
- Si une règle est fixée on passe à la règle suivante.

Pour réduire l'espace de recherche et permettre aux algorithmes d'obtenir des solutions plus prometteuses, nous avons fixé des règles de [Tab 1], et interdit les règles contenant 2 ou 3 états bord. Les règles différentes des règles de [Tab2] doivent prendre comme valeur FIRE (feu).

| gauche | centre | droite | | état |
|---|---|---|---|---|
| GENERAL | REPOS | REPOS | | A |
| BORD | GENERAL | REPOS | | GENERAL |
| GENERAL | REPOS | BORD | | GENERAL |
| A | REPOS | REPOS | | GENERAL |
| BORD | GENERAL | GENERAL | | FIRE |
| BORD | GENERAL | GENERAL | | FIRE |
| GENERAL | GENERAL | BORD | | FIRE |
| REPOS | REPOS | REPOS | | REPOS |
| BORD | REPOS | REPOS | | REPOS |
| REPOS | REPOS | BORD | | REPOS |

Tab 1 : Les règles fixes.

| gauche | centre | droite | | état |
|---|---|---|---|---|
| BORD | GENERAL | GENERAL | | FIRE |
| GENERAL | GENERAL | GENERAL | | FIRE |
| GENERAL | GENERAL | BORD | | FIRE |

Tab 2 : Les règles autorisées à prendre l'état FIRE (feu).

### Choix d'un meilleur voisin

Les algorithmes métaheuristiques examinent la totalité du voisinage pour effectuer un mouvement. Il est donc impératif de pouvoir évaluer rapidement les coûts des voisins d'une solution, pour choisir une des meilleures solutions voisines (c'est-à-dire choix du meilleur mouvement).
Une solution est meilleure que d'autres si sa fonction d'évaluation donne une valeur supérieure. En cas d'égalité nous avons défini une classe d'équivalence, qui permet d'assembler tous les mouvements (solutions) qui ont la même performance, et en suite, de choisir une solution aléatoirement. Cette configuration permet d'accepter d'autres solutions de même performance.

### Les outils utilisés

Pour le développement des différents algorithmes, c'est la bibliothèque **Paradiseo** qui a été utilisée, les paquets paradiseo-mo et paradiseo-eo en particulier.
Cette bibliothèque représente une interface développée en C++ qui permet de mettre en œuvre le parallélisme et la distribution des différents algorithmes métaheuristiques pour accroître leurs efficacités.
**Le paquet paradiseo-eo :** fournit un concept général sur des métaheuristiques pour l'optimisation combinatoire.
**Le paquet paradiseo-mo :** décrit le concept général des métaheuristiques à base de solution unique.

### Terminologie

Pour mieux comprendre les algorithmes définis ultérieurement, nous avons défini les méthodes et les objets suivants : neighbors : l'ensemble de voisinage.

initNeighbors() : initialise l'ensemble de voisinage.
solution : c'est la solution du problème des fusiliers, qui est constituée par l'ensemble des règles.
initSolution() : correspond à la solution initiale.
move : c'est un mouvement, qui dans notre cas est représenté par la règle et sa valeur.
regleMove() : applique un mouvement à une solution.
initMove() : donne une valeur initiale à un mouvement.
nexMove() : générer le mouvement suivant si c'est possible et retourne vrai, faux sinon.
nexRandMove () : générer le mouvement aléatoire .
eval () : évalue une solution.
evalMove() : évalue un mouvement sans l'appliquer.
listTabou : liste qui permet d'interdire ou d'autoriser le changement de la valeur d'une règle donné.
initlistTabou() : initialise une liste tabou.
addTabou() : interdit un mouvement pendant un nombre d'itérations donné.
updateTabou() : met la liste tabou à jour.
isTabou () :indique si une règle est tabou ou pas.
continu() : indique si la condition d'arrêt est vérifiée ou pas.
initContinu() : initialise la condition d'arrêt.
initialTemperature : la température initiale.
finalTemperature : température finale.
initTemp() : initialise la méthode qui permet de mettre à jour la température.
updateTemperature() : met à jour la température et rend vrai, faux sinon.

### 2.1.3 Algorithmes

Cette partie présente les différents algorithmes.

#### Algorithme de Hill Climbing

```
begin
initNeighbors (neighbors) solution = initSolution
bestSolution = solution
currentFitness = eval(solution)
bestFitness = currentFitness
initMove(move)
repeat
  currentFitness = evalMove(move,solution)
  if currentFitness ≤ bestFitness then
    regleMove(move , solution )
    bestSolution = solution
    bestFitness = currentFitness
  end if
until nextMove(move ) and continu ()
```

return Solution

**Algorithme de Recherche Tabou**

```
begin
initlistTabou(listTabou)
initNeighbors (neighbors)
solution = initSolution
bestSolution = solution
currentFitness = eval(solution)
bestFitness = currentFitness
initMove(move )
repeat
  repeat
    nextMove(move )
  until isTabou(move)
  currentFitness = evalMove ( move, solution)
  if  bestFitness ≥ currentFitness  then
    regleMove(move , solution )
    bestSolution = solution
    bestFitness = currentFitness
    addTabou(mov, listTabou)
  end if
  update(listTabou)
  nextMove(move)
until  continu()
return Solution
```

**Algorithme de Recuit Simulé**

```
begin
initNeighbors (neighbors)
solution = initSolution
bestSolution = solution
currentFitness = eval(solution)
bestFitness = currentFitness
temperature = initialTemperature
repeat
  initContinu()
  repeat
    nextRandMove(move)
    currentFitness = evalMove ( Move, solution)
    delta = currentFitness - bestFitness
    if  currentFitness > bestFitness then
```

```
        regleMove(move,solution)
        bestSolution = solution
        bestFitness = currentFitness
        else
        if andom(0,1) < exp(-delta / temperature) then
        regleMove(move,solution)
        bestSolution = solution
        bestFitness = currentFitness
      end if
    until continu()
  until  updateTemperature()
  return solution
```

### 2.1.4 Résultats expérimentaux

Cette section présente les résultats expérimentaux des algorithmes définis précédemment pour un nombre d'itérations égale à 5000.

#### Résultats expérimentaux de Hill Climbing

Les résultats ont été obtenu dans cette section pour 5000 exécutions de l'algorithme. La figure suivante montre la distribution des résultats obtenus, c'est à dire le nombre d'exécution de HC en fonction de la taille du problème résolu.

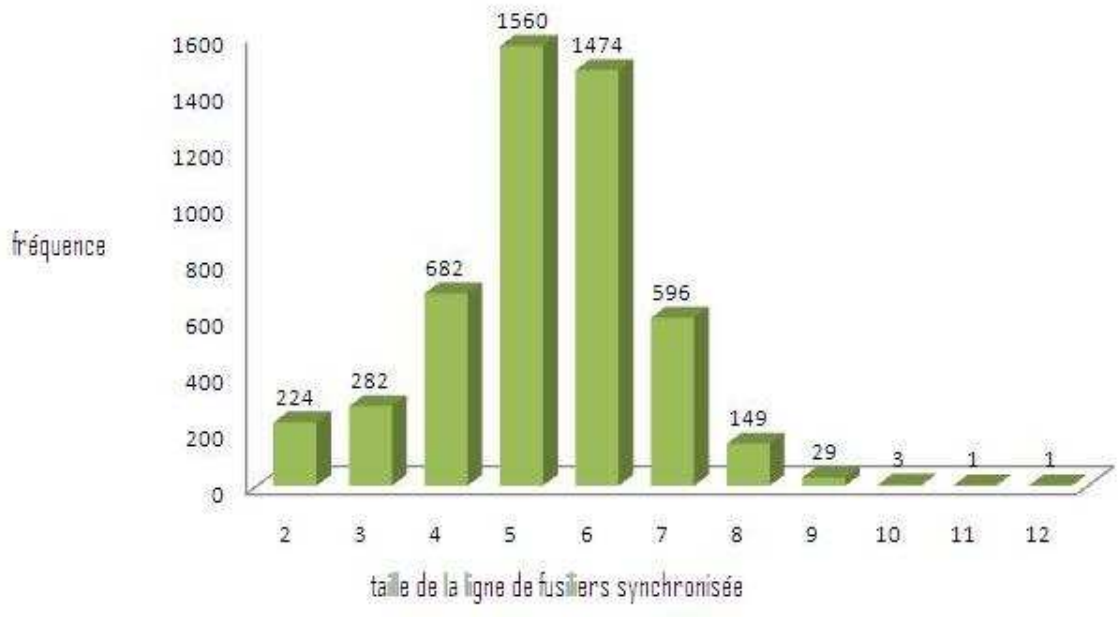


FIG. 2.3 – nombre d'exécutions de HC en fonction de la taille du probème résolu

À partir du graphe 2.3 qui représente la fréquence d'apparitions des solutions pour chaque taille de ligne fusiliers synchronisés en utilisent le Hill Climbing on remarque :

- La fréquence des solutions qui permet de synchroniser une ligne de fusiliers de taille 5 ou 6 est plus grande que pour les autres solutions.
- La fréquence des solutions qui permet de synchroniser une ligne de fusiliers de taille 10, 11 et 12 est très faible comparée aux autres solutions.

A partir des ces résultats on a pu déterminer :

- La moyenne = 5,27
- L'écartype = 1,36

La courbe suivante représente la moyenne de temps pour trouver la solution qui synchronise la taille de ligne de fusiliers.

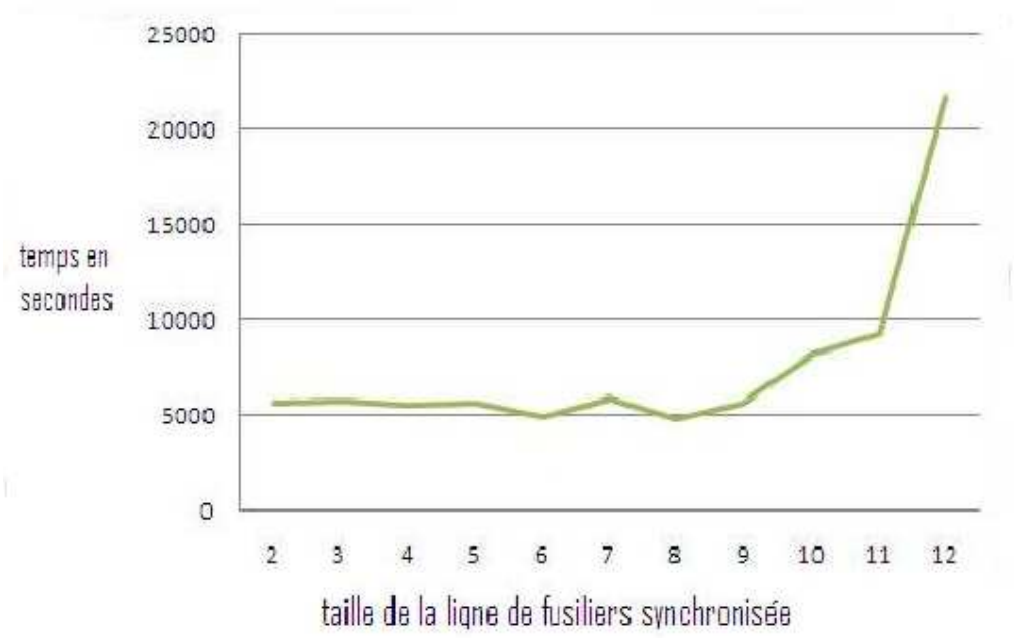


FIG. 2.4 – Temps moyen pour chaque taille de ligne de fusiliers synchronisés

En analysant la courbe précédence 2.4 on remarque qu'elle est presque constante pour des lignes de fusiliers synchronisés de taille 2 jusqu'à 9 puis qu'elle croît exponentiellement. Donc on peut déduire qu'on peut trouver des solutions qui synchronisent une ligne de fusiliers de taille supérieure à 12 mais il faut lancer l'application pendant une période de temps très grande qui tend ver l'infini.
La meilleure solution 2.5 trouvée permet de synchroniser une ligne de fusiliers de taille 12 en temps minimal« 2N-2 »avec les mêmes règles.

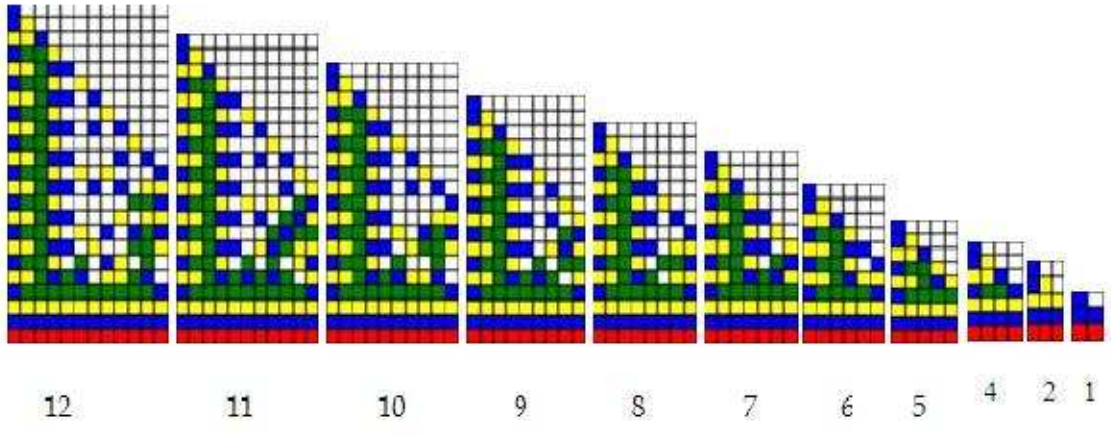


FIG. 2.5 – Diagramme espace temps des résultats de Hill Climbing

### Résultats expérimentaux de Recherche Tabou

Les résultats obtenus dans cette section pour 1000 exécutions de l'algorithme, une liste taboue de taille égale au nombre des règles possibles 216 et une longueur tabou = 30. La figure suivante montre la distribution des résultats obtenus, c'est à dire le nombre d'exécutions de RT en fonction de la taille du problème résolu.

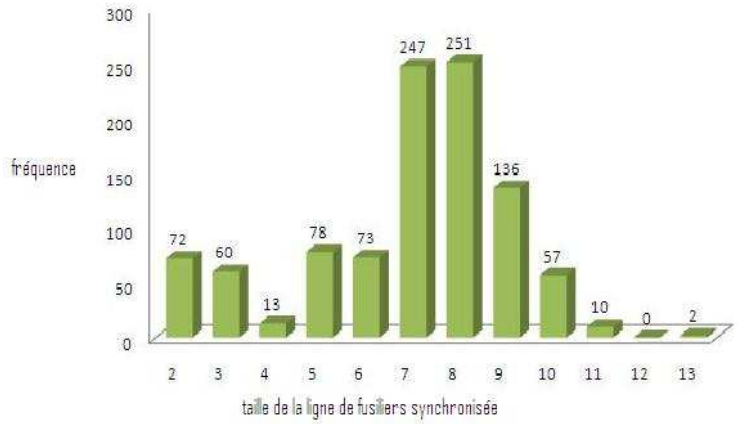


FIG. 2.6 – nombre d'exécutions de RT en fonction de la taille du problème résolu

A partir du graphe 2.6 qui représente la fréquence d'apparitions des solutions pour chaque taille de ligne fusiliers synchronisés en utilisant la Recherche Tabou on remarque :

- La fréquence des solutions qui permet de synchroniser une ligne de fusiliers de taille 7, 8 et 9 plus grande que pour les autres solutions.

- La fréquence des solutions qui permet de synchroniser une ligne de fusiliers de taille 11, 12 et 13 est très faible comparée aux autres solutions.

A partir des ces résultats on a pu déterminer :

- La moyenne = 7,1
- L'écartype = 2,19

La courbe suivante représente la moyenne de temps pour trouver la solution qui synchronise la ligne de fusiliers.

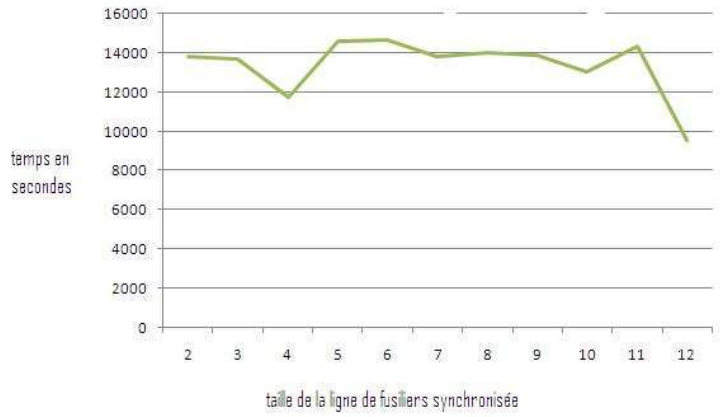


FIG. 2.7 – Temps moyen pour chaque taille de ligne de fusiliers synchronisés

En analysant la courbe précédence 2.7 on remarque qu'elle est presque constante pour toutes les lignes de fusiliers synchronisés. Donc on peut déduire qu'on peut trouver des solutions qui synchronisent une ligne de fusiliers de taille supérieure à 13 en augmentant le nombre d'exécutions, mais l'augmentation du nombre d'itération n'a pas d'influence sur les résultats. Les meilleures 2.8 et 2.9 solutions trouvées permettent de synchroniser des lignes de fusiliers de taille 13 en temps minimal (2N-2) avec les mêmes règles.

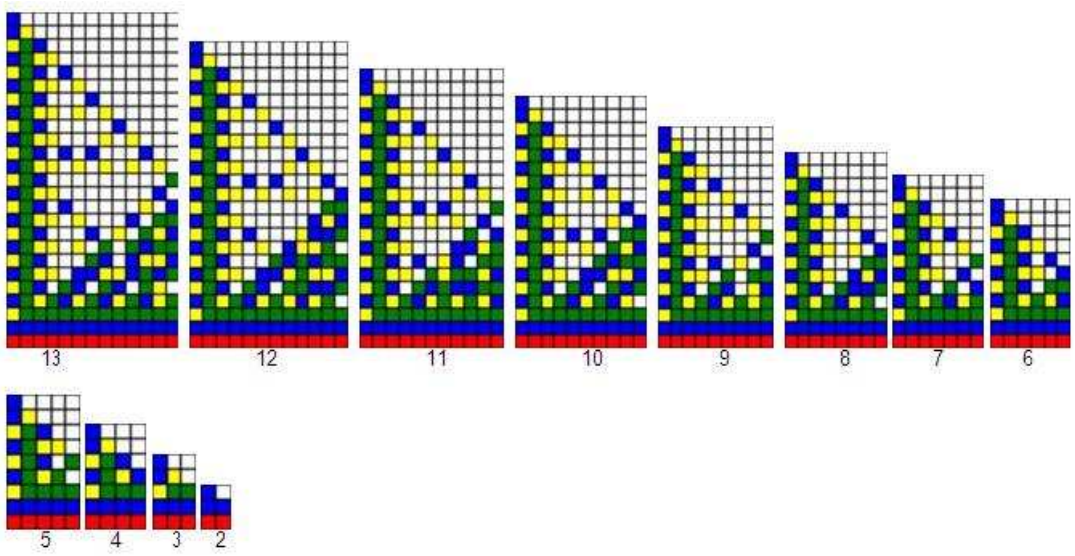


FIG. 2.8 – Diagramme espace temps des résultats de la Recherche Tabou

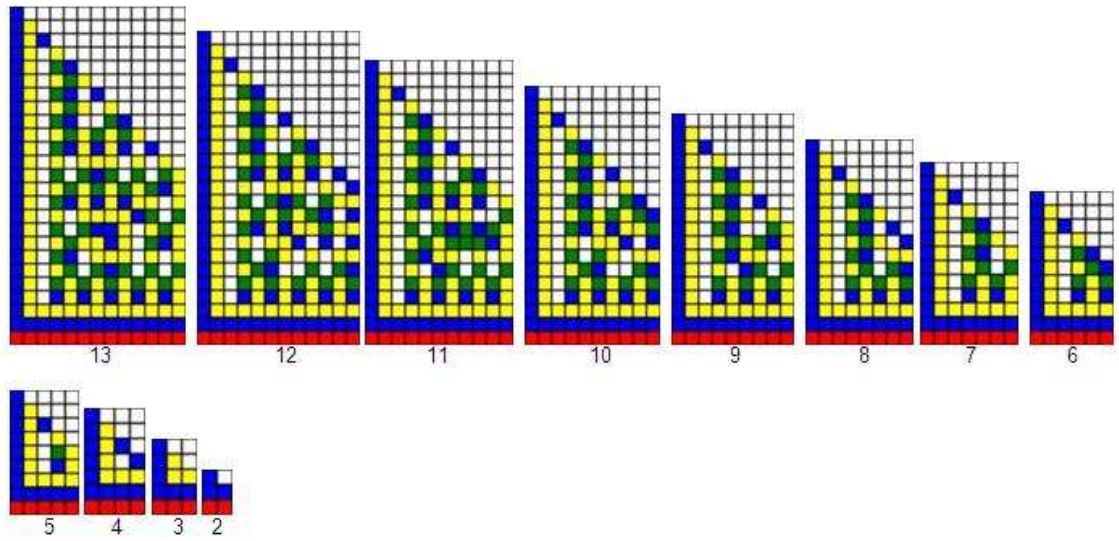


FIG. 2.9 – Diagramme espace temps des résultats de la Recherche Tabou

**Résultats expérimentaux de Recuit Simulé**

Les résultats obtenus après avoir testé plusieurs valeurs pour les paramètres de l'algorithme. Voici les valeurs des paramètres pour les meilleurs résultats obtenus pour le recuit simulé :

- Nombre d'exécutions =75
- température initiale 10
- température finale 0.001

- coefficient multiplicatif pour la décroissance de température 0.999

La figure 2.10 montre la distribution des résultats obtenus, c'est-à-dire le nombre d'exécutions de RS en fonction de la taille du problème résolu.

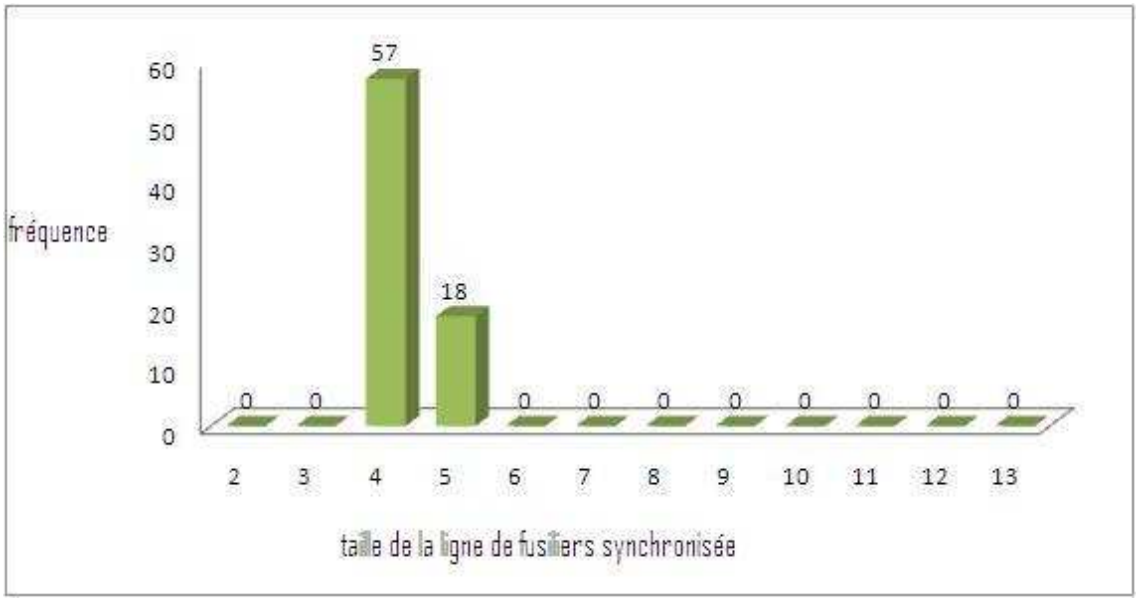


FIG. 2.10 – nombre d'exécutions de RT en fonction de la taille du probème résolu

A partir du graphe 2.10 qui représente la fréquence d'apparition des solutions pour chaque taille de ligne fusiliers synchronisés en utilisent le Recuit Simulé on remarque :

- La fréquence des solutions qui permet de synchroniser une ligne de fusiliers de taille 4 et 5 est plus grande que pour les autres solutions.
- La fréquence des solutions qui permet de synchroniser une ligne de fusiliers pour les autres tailles est très faible.

A partir des ces résultats on a pu déterminer

- La moyenne = 4,24
- L'écartype = 0,43

Les meilleurs solutions 2.11 trouvées permettent de synchroniser des lignes de fusiliers de taille 5 en temps minimal (2N-2) avec les mêmes règles.

Les résultats obtenus dans cette section ne nous semble pas convaincants et ceci dû au mauvais choix de paramètres, car les principaux inconvénients du recuit simulé résident dans le choix de ces paramètres, tels que la température initiale, la loi de décroissance de la température, les critères d'arrêt ou la longueur des paliers de température. Comme nous avons trouvé de bons résultats pour la recherche tabou, que le temps d'une exécution du RS est grand (416s) et que pour évaluer les paramètres il faut faire plusieurs exécutions, nous avons décidé d'abandonner les tests ainsi que l'exploration de cette piste.

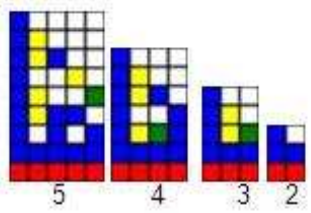


FIG. 2.11 – Diagramme espace temps des résultats du Recuit simulé

### 2.1.5 Analyse comparative des différents algorithmes

Nous présentons dans cette partie les résultats comparatifs des différents algorithmes, et voici le tableau récapitulatif des résultats obtenus précédemment.

| Méthode | nb itérations | nb Run | | meilleur résultat | temps pour un Run | total total | moyenne | écartype |
|---|---|---|---|---|---|---|---|---|
| Hill Climbing | 5000 | 5000 | | 12 | 28s | 140000s(38h53) | 5,27 | 1,36 |
| Recherche Tabou | 5000 | 1000 | | 13 | 22s | 27517s(7h64) | 7,1 | 2.12 |
| Recuit Simulé | 5000 | 75 | | 5 | 416s | 32021s(8h53) | 4,24 | 0,42 |

Tab 3 : Tableau récapitif des résultats obtenus

A la vue de ces résultats, nous remarquons que la recherche tabou est plus efficace pour un même nombre d'itérations. C'est pour cela que nous l'avons utilisée en approche combinée avec les algorithmes évolutionnaires.

## 2.2 Algorithme évolutionnaire

### 2.2.1 Principe

Cette méthode est basée sur la théorie de l'évolution de Darwin. Il s'agit d'effectuer une sélection des générations futures puis des variations sur celles-ci. Ces variations consistent à faire muter la nouvelle population et à la croiser deux à deux .

### 2.2.2 Définitions

#### Individus et population

Dans le monde des algorithmes évolutionnaires, un **individu** est une solution, plus ou moins performante, à un problème donné.

L'ensemble des individus traités par l'algorithme évolutionnaire, se nomme une **population**. Parmi les individus d'une population, on distingue les individus $\mu$ (parents) et les individus $\lambda$ qui sont leurs enfants.

#### Fonction d'évaluation

**La fonction objectif** (fonction de **fitness**) est la fonction qui sert à mesurer la **performance** d'une solution.

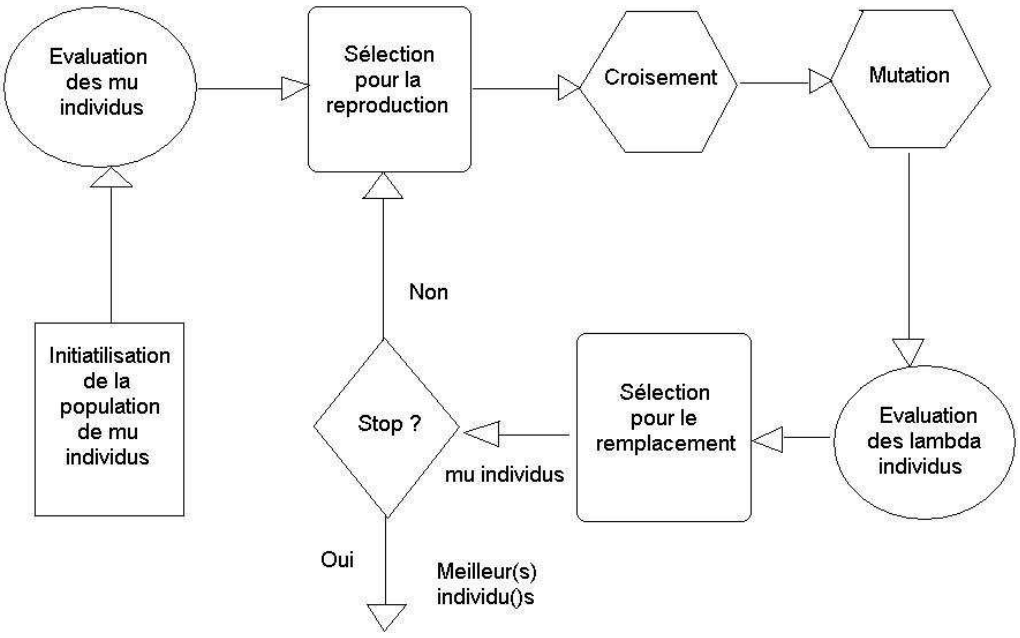


FIG. 2.12 – boucle évolutionnaire

### Initialisation de la population initiale

Dans l'étape d'initialisation de la population, on prend $\mu$ individus. La manière de choisir les individus dépend bien entendu du problème donné.

### Evaluation des $\mu$ individus

Dans cette étape de la boucle évolutionnaire, on évalue chaque indivu $\mu$ de la population avec la fonction de fitness.

### Sélection pour la reproduction

Le principe de la sélection pour la reproduction, est de choisir parmi les parents d'une génération, ceux qui vont se reproduire. La sélection pour le remplacement suppose que parmi une population d'individus $\mu$, on applique un certain taux de reproduction. Par exemple, si le taux de reproduction est de 80%, cela voudra dire que 80% des individus $\mu$ vont être sélectionnés pour se reproduire. Nous allons tout de suite définir quelques méthodes de sélection.

### Sélection par tournois déterministes

Le principe de la sélection par tournois déterministes, consiste à effectuer $n$ tournois de $k$ individus. On suppose que $n$ est le nombre d'individus choisis pour se reproduire.Le principe du tournoi est le suivant : pour chaque tournoi de $k$ individus sélectionnés aléatoirement dans la population d'individus $\mu$, on prend le meilleur. Et on refait la même opération $n$ fois ($n$ tournois). On peut choisir s'il s'agit d'un tournoi avec ou sans remise, à savoir si l'on autorise ou non de remettre un individu qui a remporté un tournoi précédent.

### Sélection aléatoire

L'idée de cette méthode est simple : on sélectionne $n$ individus aléatoirement parmi la population d'individus $\mu$ pour se reproduire. Cette méthode simple privilégie le phénoméne de diversification des solutions. On peut noter qu'en prenant la méthode de la sélection par tournois déterministes, il suffit de prendre comme paramétre $k = 1$ pour ainsi obtenir la sélection aléatoire.

### Sélection élitiste

Le principe de la sélection élitiste est aussi trés simple. Il suffit de choisir les $m$ meilleurs individus parmi les individus $\mu$. Cette méthode privilégie ainsi le phénoméne d'intensification.

### Opérateurs de variation

Pour revenir à notre boucle évolutionnaire, à l'issu de l'étape de sélection pour la reproduction, on a besoin d'opérateurs de variation pour obtenir de nouvelles solutions qu'on appellera **individus** $\lambda$ (enfants). Ces opérateurs se nomment **croisement** et **mutation**.

### Croisement

On va prendre les individus $\mu$ (parents) précédemment sélectionnés, pour se reproduire deux à deux et donner des individus $\lambda$ (enfants). L'idée du croisement est proche de celle d'un brassage génétique, où on mélange les caractéristiques de chacun des individus. Pour chacun des deux individus, on choisit le même point de croisement. Les caractéristiques qui se trouvent à gauche de ce point pour le premier individu, et les caractéristiques qui se trouvent à droite de ce point pour le deuxième individu, vont être raccollées pour donner naissance à un individu $\lambda$ (figure de droite). On fait de même pour les caractéristiques qui se trouvent à droite du point pour le premier individu, et les caractéristiques qui se trouvent à gauche du deuxième individu. On obtient ainsi un deuxième enfant.

### Notre étude

Dans notre étude, une population est composée d'un ensemble de règles représentées par un vecteur. Les mutations consistent à modifier une règle aléatoirement et les croisements consistent à prendre un nombre aléatoire correspondant à l'indice des vecteurs qu'on va interchanger.
Voici un exemple montrant une mutation puis un croisement d'une génération à l'autre :

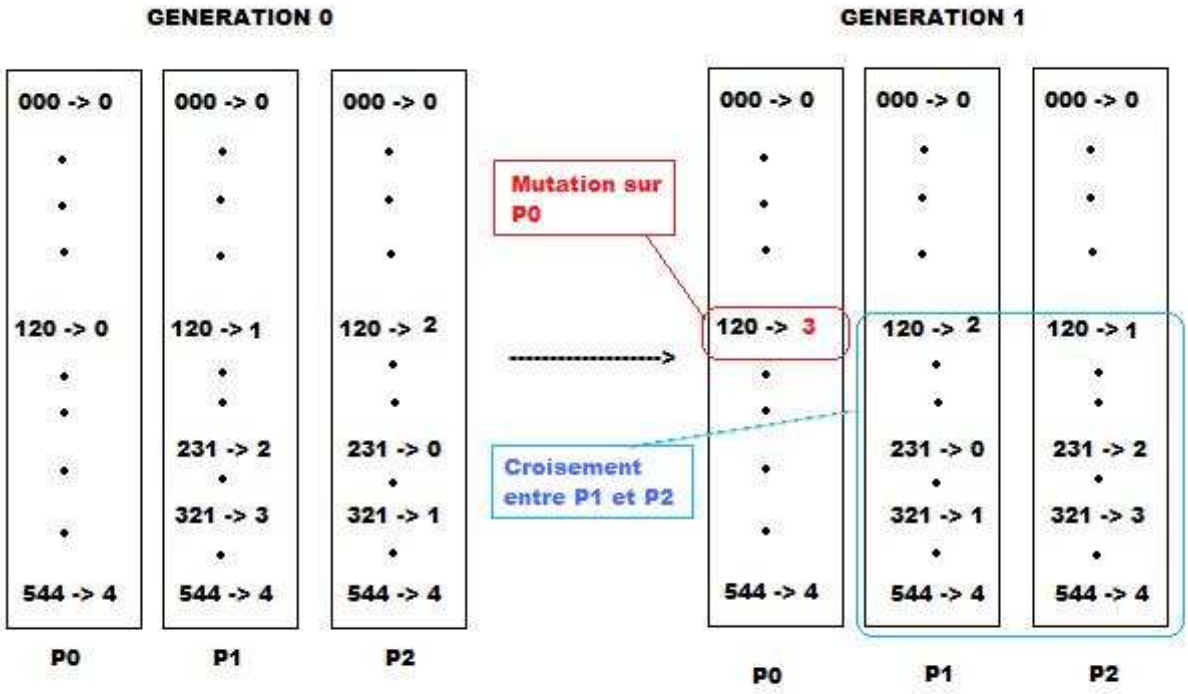


**Illustration d'une mutation et d'un croisement d'une génération à l'autre**

### 2.2.3 Environnement et algorithme

Pour le développement de cet algorithme nous avons utilisé la bibliothéque Paradiseo et le package paradiseo-eo en particulier. Cette bibliothéque représente une interface qui permet de mettre en oeuvre le parallélisme et la distribution des différents algorithmes métaheuristiques pour accroître leurs efficacités. Enfin, paradiseo-eo fournit des concepts permettant le développement d'algorithmes génétiques.

- neighbors : l'ensemble des voisins possibles.
- InitRegle : opérateur d'initialisation pour l'ensemble des règles.
- PopEval : opérateur d'évaluation correspondant à la fonction objectif.
- Pop : population de règles
- TournamentSelect : opérateur de sélection de population
- SGATransform : opérateur de transformation pour une population en fonction de la mutation, du croisement
- croisement , mutation : opérateur de croisement et de mutation
- MAXGEN : nombre de génération maximum

Algorithme :
begin
InitRegle init (neighbors)
Pop population
PopEval eval
TournamentSelect select, SGATransform transform
croisement cross
mutation mut
**repeat**
  transform(pop, mut, cross)
  select(pop)
**until** gen $\geq$ MAXGEN
end

### 2.2.4 Implémentation

Chaque solution, (ensemble de règles) est representée par un vecteur de taille 216. Cette taille étant justifiée par les règles de 000 à 555. Bien sur certaines règles de ce vecteur ne sont jamais utilisées (exemple 555 ou 515). Pour cela, nous avons besoin d'un voisinage pour chaque règle representé par un vecteur à deux dimensions permettant d'initialiser les valeurs possibles et celles interdites. Enfin, la fonction objectif (ou d'évaluation) consiste à choisir comme meilleure

solution, celle qui synchronise le plus grand nombre de fusiliers. Celle-ci doit aussi synchroniser pour les automates de taille inférieure.

### 2.2.5 Résultats

Au cours des différentes expérimentations, le meilleur résultat obtenu pour cette approche est une synchronisation jusqu'à 8, illustrée ci dessous :

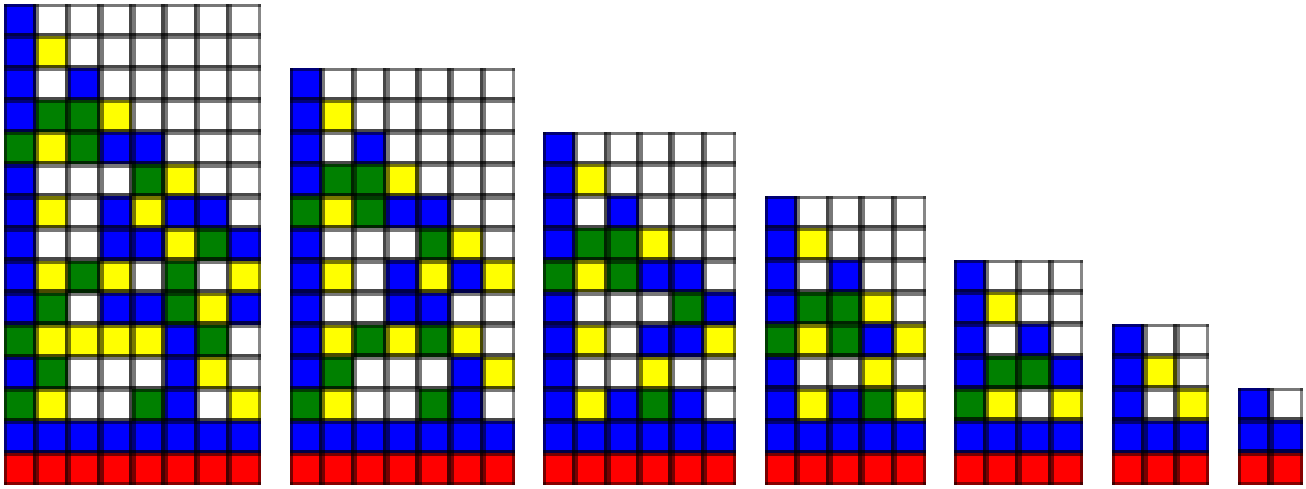

FIG. 2.13 – La meilleure solution obtenue

Voici le tableau significatif représentant la moyenne des fusiliers synchronisés en fonction de la population pour 500 générations, le taux de mutation étant fixé à 0.3 et celui du croisement à 0.7.

On remarque que plus la population augmente, plus le nombre de fusiliers synchronisés aug-

| Nombre d'individus dans la population | Moyenne du nombre de fusiliers synchronisés |
|---|---|
| 5 | 2 |
| 10 | 2.3 |
| 20 | 2.2 |
| 30 | 2.3 |
| 40 | 3.2 |
| 50 | 3 |
| 60 | 3.3 |
| 70 | 3.8 |

FIG. 2.14 – Variation de la population (15 runs pour chaque valeur)

mente.

Voici les tableaux significatifs représentant la moyenne des fusiliers synchronisés en faisant varier le taux de mutation et de croisement pour 500 générations :

| Taux de croisement | Moyenne du nombre de fusiliers synchronisés |
|---|---|
| 0.1 | 3.11 |
| 0.2 | 2.55 |
| 0.3 | 3.11 |
| 0.4 | 2.66 |
| 0.5 | 2.66 |
| 0.6 | 2.77 |
| 0.7 | 3.66 |
| 0.8 | 2.55 |
| 0.9 | 3.22 |

| Taux de mutation | Moyenne du nombre de fusiliers synchronisés |
|---|---|
| 0.1 | 2.12 |
| 0.2 | 2.91 |
| 0.3 | 3.47 |
| 0.4 | 3.37 |
| 0.5 | 3.61 |
| 0.6 | 4.95 |
| 0.7 | 3.96 |
| 0.8 | 5.31 |
| 0.9 | 5.21 |

FIG. 2.15 – Variation du taux de mutation et de croisement (15 runs pour chaque valeur)

Enfin, ici le taux de croisement est fixé à 0.7 et celui de mutation à 0.3. On remarque que les valeurs maximums sont pour un taux de mutation de 0.9 et un taux de croisement de 0.8

## 2.3 Backtracking

Approche ascendante et analytique : à partir de la configuration finale, conception des règles locales de proche en proche et utilisation du backtracking en cas de conflit.

### 2.3.1 Principe

Le backtracking est une technique d'énumération de l'espace de recherche dans un ordre particulier jusqu'à trouver une solution adéquate et qui tente d'éviter l'énumération exhautive de l'espace de recherche.
L'idée maîtresse est d'essayer chaque alternative (combinaison) jusqu'à trouver la bonne, ou à revenir légérement en arrière sur des décisions prises lorsque l'alternative ne mène pas à la bonne solution.
Pour ce problème, la recherche consiste à trouver des règles permettant de synchroniser les $n$ cellules de la configuration initiale au bout de $2n - 2$ itérations (temps minimal).
Points qui font échouer une solution :

- L'état feu symbolisant la synchronisation n'est pas obtenu au bout de 2N-2 itérations.
- L'état feu est obtenu avant ce nombre précis d'itérations.

- Ces règles ne synchronisent pas les automates pour une configuration initiale allant de 2 à $n$ cellules.

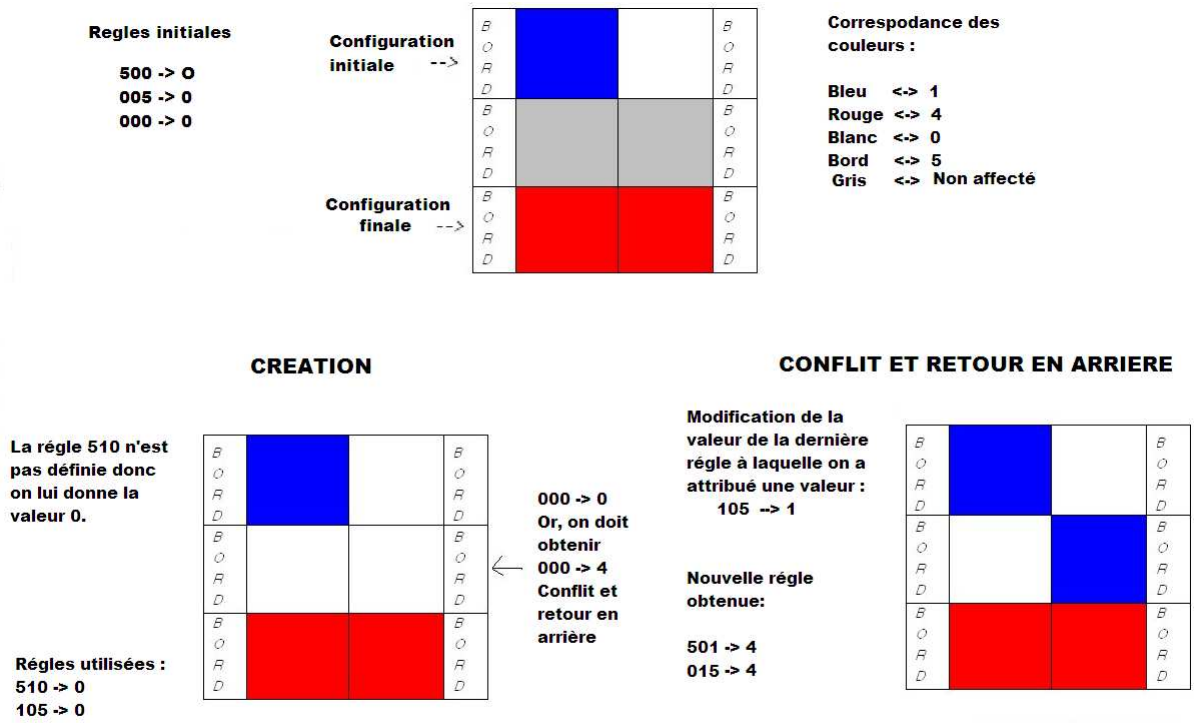


FIG. 2.16 – Affectation et modification d'une règle par le bactracking

Ainsi, lors de l'affectation d'une valeur du conséquent d'une règle, si une possibilité essayée (attribution d'une valeur à une règle) ne satisfait plus une contrainte, un retour sur trace à un point où d'autres alternatives s'offraient à nous est effectué et la possibilité suivante est choisie (attribution d'une autre valeur pour la règle). S'il n'y a plus de tels points, la recherche échoue. La force du backtracking est que beaucoup de ses réalisations évitent d'essayer beaucoup de combinaisons partielles, diminuant ainsi le temps d'exécution.

### 2.3.2 Algorithme

- InitRegles(fichier) : initialise les règles à partir d'un fichier passé en argument.
- InitPile(fichier) : initialise la pile à partir du même fichier (passé en argument).
- Init (minTaille,maxTaille) : initialise la taille de la première configuration "minTaille" (en général 2) et de la dernière "maxTaille"(<=12 pour obtenir une solution dans un temps raisonable) avec deux paramètres passés en argument.
- InitConfiguration(taille) : initialise la configuration initiale

- NbFire : Nombre de cellules à l'état feu
- T : numéro de la configuration courante
- i : numéro de la cellule courante de la configuration courante
- r : valeur de la règle locale
- regle($\delta$, Config,i) : retourne la valeur de la règle correspondant à la configuration Config[i], c'est à dire que l'on utilise la configuration courante pour obtenir la configuration suivante. i correspond au positionnement de la cellule traitée.

| configurtion courante x = ... | $\mathcal{X}_{i-1}$ | $\mathcal{X}_i$ | $\mathcal{X}_{i+1}$ |
|---|---|---|---|
| configurtion suivante x' = ... | ... | $\mathcal{X'}_i$ | ... |

- NonAffecte : règle non initialisée par le fichier (aucune valeur)
- CreateFire (Config,i,N) : donne la valeur Fire correspondant à l'état feu à la règle et l'empile en sommet de pile avec le N (taille de l'automate) correspondant au moment où la règle a été crée.
- Fire : valeur correspondant à l'état feu
- backtrack(N) : dépile la règle se trouvant en sommet de pile tant que la valeur est égale à fire. Ensuite, modifie la règle se trouvant en sommet de pile en incrémentant sa valeur de 1 et retourne le N correspondant à la création de la règle en question.

Algorithme :

```
InitRegles (fichier)
InitPile (fichier)
Init (minTaille,maxTaille)
for N = minTaille to maxTaille do
  InitConfiguration(N)
  NbFire ← 0 ;
  T ← 1 ;
  while T <= 2 * N − 2 & NbFire ≠ 0 do
    for i = 1 to N do
      r ← regle(δ, Config,i) ;
      if T = 2*N-2 then
        if r = NonAffecte then
          CreateFire (Config,i,N)
          NbFire ← NbFire+1 ;
        else if r = Fire then
          Config [i] ← r
          NbFire ← NbFire+1 ;
```

```
            else
              Backtrack(N)
            end if
          else
            if r ≠ NonAffecte & r ≠ Fire then
              Config [i] ← r
            else if r = Fire then
              Backtrack(N)
            else
              Create (Config,i,N)
            end if
          end if
        end for
      end while
    end for
```

### 2.3.3 Implémentation

Une pile permet d'implémenter le backtrack. Chaque nouvelle règle crée est enregistrée dans la pile. Le choix d'une pile n'est pas anodin, en effet : la règle à modifier doit être la dernière à avoir été crée (il y a en quelque sorte des règles plus prioritaires que d'autres), ainsi, lorsqu'un conflit se produit la règle au dessus de la pile est modifiée dans l'ordre de l'énumération des états. De plus, le changement de la dernière règle affectée permet de conserver la validitée de la solution sur des tailles plus petites.
Donc plus la règle a été crée tardivement (donc positionnée en sommet de pile) et moins il faudra revenir en arrière (à un automate avec un nombre initial de cellule plus petit).
Les résultats sont copiés dans un fichier et des diagrammes espaces temps sont obtenus à la fin de l'exécution.

### 2.3.4 Résultats

Grâce à l'algorithme implémenté, nous avons obtenu des premiers résultats (prometteurs pour la dernière approche) :
synchronisation en temps minimal (2n-2) d'une ligne de fusiliers avec un nombre initial de cellule allant de 2 à 12 (et bien entendu avec les mêmes règles).
Ce sont les meilleurs résultats trouvés jusqu'alors mais la complexité est importante pour cet algorithme : 788 secondes pour l'exécution ayant fournie la meilleure solution.

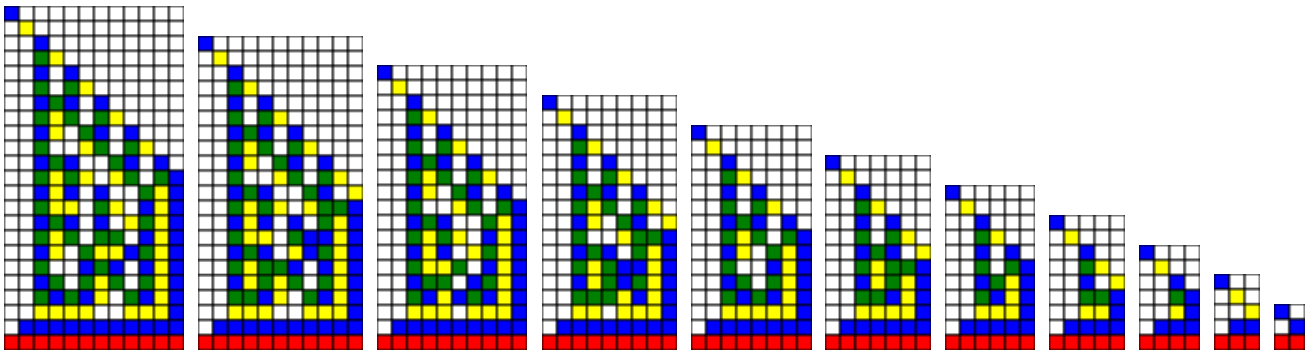

FIG. 2.17 – Une des meilleures solutions obtenues de 2 à 12

Nous avons effectué des mesures de temps pour avoir une idée de la complexité de l'algorithme en fonction du nombre de cellules initiales. Le diadramme de temps montre quelques mesures de temps et l'on remarque que la courbe croît exponentiellement dès que l'on dépasse 9. Comme prévu, en voyant la taille de l'espace de recherche, on constate que le temps pose problème.
Le temps pour résoudre le problème de taille n en suposant que la même forme de courbe est conservée tendrait vers l'infini.

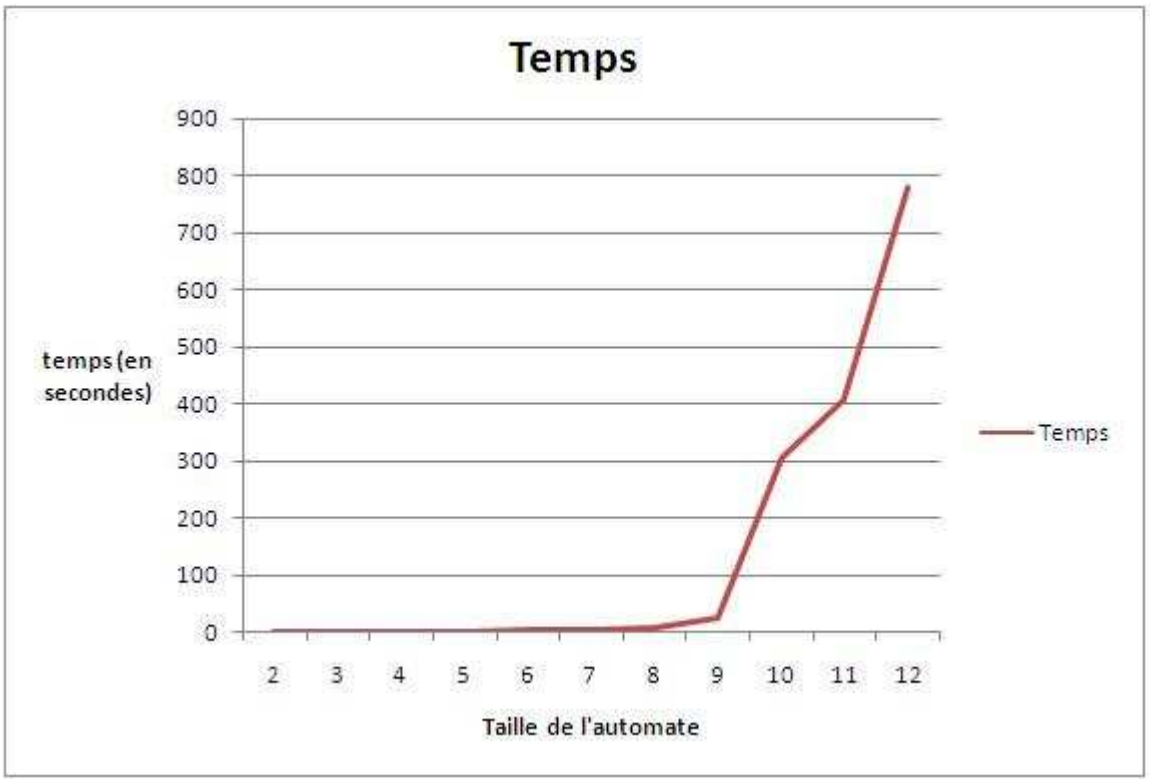


FIG. 2.18 – mesure de temps

Nous avons lancé l'exécution sur le serveur de calcul : Pour n allant de 2 à 13, plusieurs

heures de calculs sont nécessaires et malgrés cela, aucune solution n'est trouvée.

Notre but a donc été d'optimiser au maximum l'algorithme pour diminuer le temps d'exécution :

- sortir les variables de boucles
- utiliser les versions des règles obtenues qui sont le plus rapides...

### 2.3.5 Pistes explorées

Nous avons tenté d'explorer les résultats obtenus par l'algorithme du backtracking avec :

- l'approche par signaux. Nous avons voulu déceler l'apparition des signaux sur les diagrammes espaces temps fournis.
- l'approche combinée. Nous avons essayé de les utiliser comme base (règle qu'ils vont donner en entrées aux algorithmes).

Ces deux pistes n'ont pas abouti : avec l'approche par signaux, il faut un nombre de cellules initial plus grand pour envisager la détection d'un signal. 12 est trop petit.
Le départ pris, i.e les règles crées au départ par l'algorithme doivent certainement correspondre à une partie de l'espace de recherche qui ne contient pas la solution.

D'autres idées ont été explorées :

- Partir de n = 13, et essayer de backtracker vers n = 2. Cette approche a vite été abandonnée, en accord avec nos encadrants : en effet, dans l'espace de recherche, sans orienter les recherches (en fournissant déjà quelques règles), le temps pour trouver les règles qui synchronisent les cellules (pour n = 13) est trop important.
- Au lieu d'initialiser une règle nouvellement crée à 0 (puis 1,2,3 et 4 en cas de conflit), nous avons commencé à 1 (puis 0,2,3 et 4 en cas de conflit). Nous avons choisi de tester cela car lors de nos premières expérimentations à la main (où nous avions obtenu une synchronisation pour n allant de 2 à 8), l'une des premières règles crées était initialisée à 1. Pourquoi 0 ou 1 : En fait c'est pour tenter de minimiser le nombre de règles (et donc de diminuer les contraintes).
  Au départ, la configuration initiale n'est composée que de 0 (état repos) et de 1 (état général), donc, en initialisant une règle (lors de sa création) à 0 ou à 1, on augmente les chances de retrouver des règles déjà crées par la suite (composée de 0 ou de 1) et on évitera ainsi la création de nouvelles règles. Le fait qu'il y ai moins de règles signifie qu'il y a moins de contrainte (avec d'autre valeurs).

  En lançant l'exécution sur le serveur de calcul, nous avons obtenu comme résultat la synchronisation des cellules pour n allant de 2 à 5. Pour n = 6, l'exécution ne permet pas pour l'instant d'avoir une solution : le temps de calcul est trés important. Nous avons essayé

d'y parvenir en fournissant plus de règles dés le départ, et notre algortihme a obtenu une synchronisation de 2 à 11.

### 2.3.6 Statistiques obtenus

En sauvegardant des différentes versions de règles permettant la synchronisation de 2 à 12, nous avons obtenu 3299 versions différentes pour n = 12 (d'au moins une règle) – en commençant à initialiser une nouvelle règle à 0 – , et avons donc mis en place un nouveau programme pour compter le nombre de fois qu'une règle prend une certaine valeur.

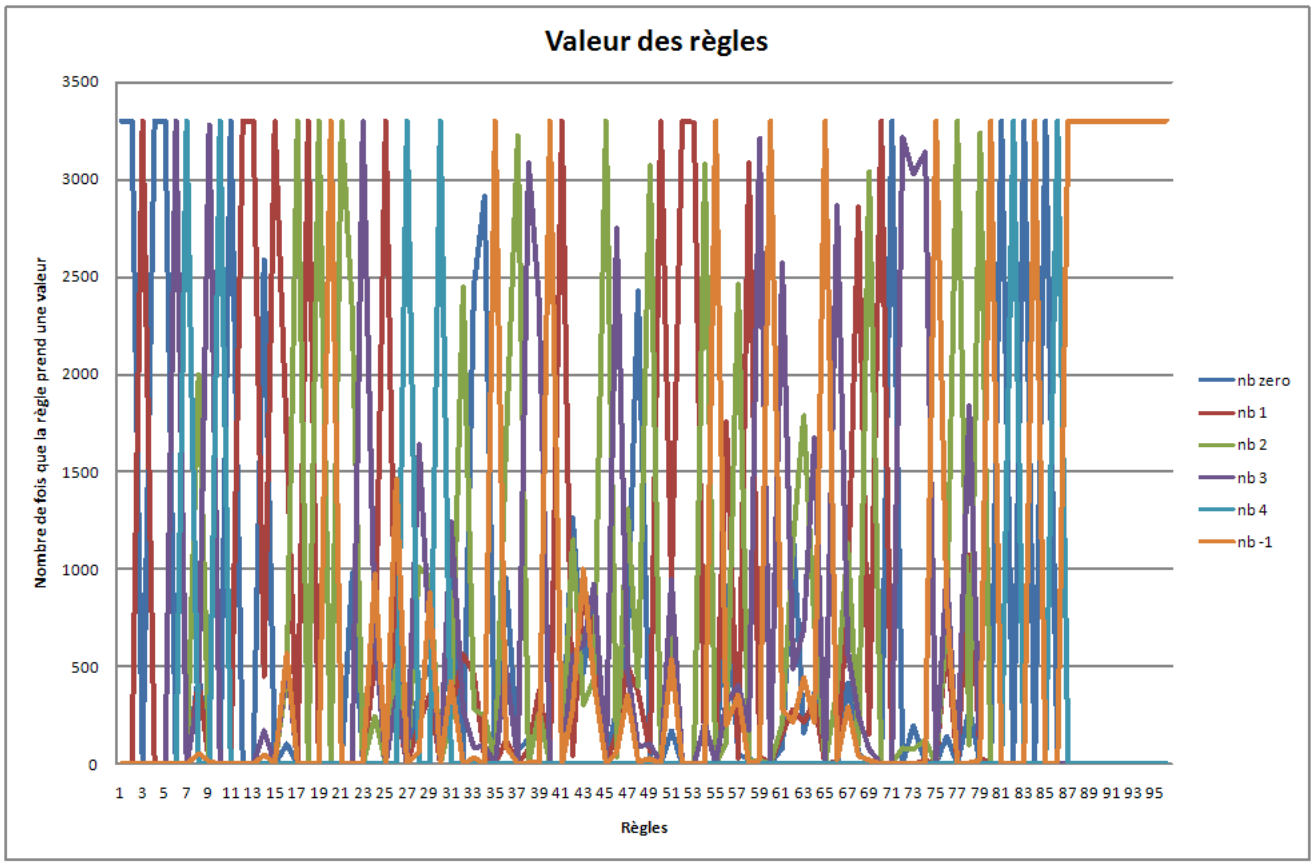


FIG. 2.19 – valeur pour chaque règle

Ainsi nous avons aussi la valeur la plus souvent utilisée pour une règle. Bien sur cela ne nous donne un indice que pour une certaine partie de l'espace de recherche (en effet, les différentes versions appartiennent toutes à une même zone).

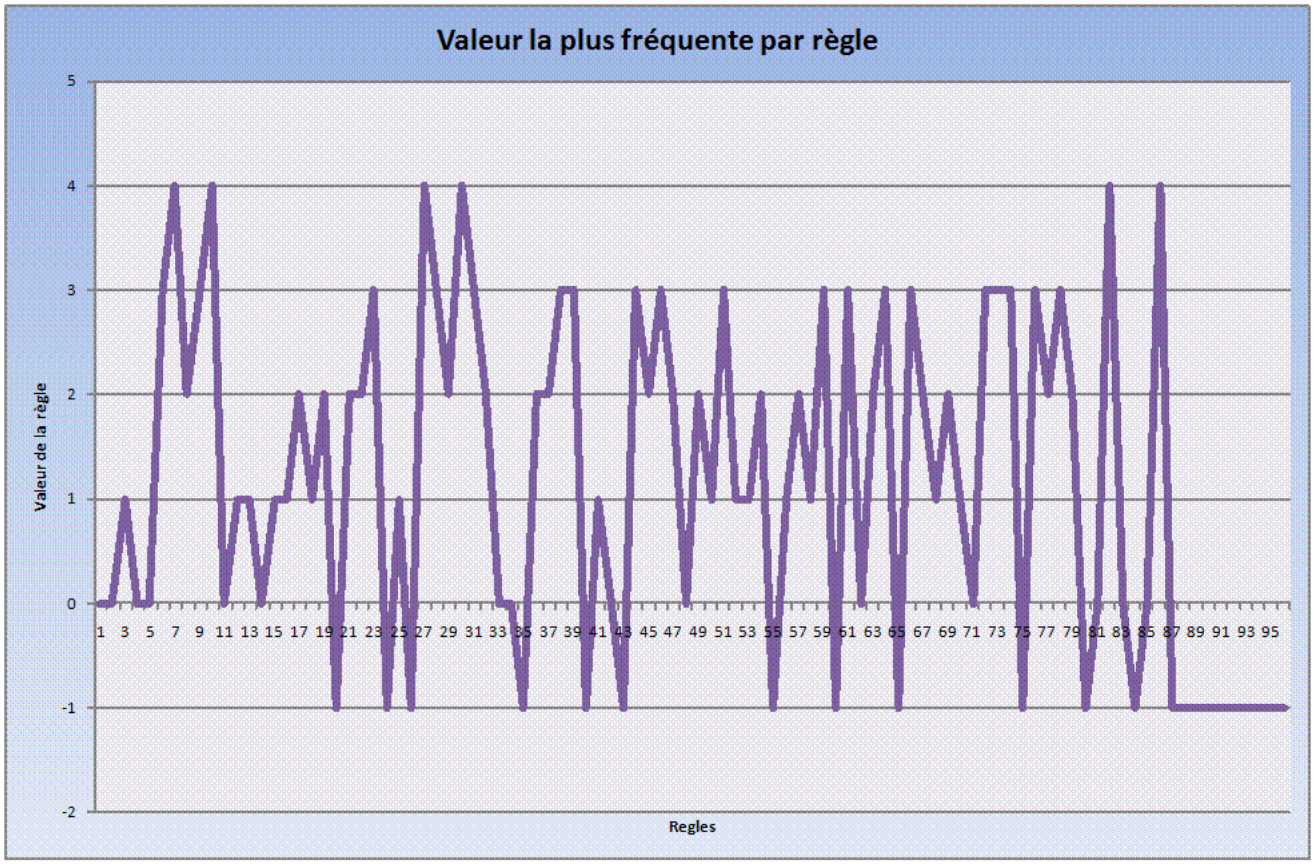


FIG. 2.20 – valeur la plus fréquente pour chaque règle

Une autre programme nous permet d'obtenir le diagramme espace temps moyen pour les 3299 versions de 12 trouvées : il compte le nombre de fois qu'une case du diagramme espace temps, pour n=12, va prendre la valeur 0,1, ... ,4 et retourne le diagramme espace temps correspondant à la moyenne.

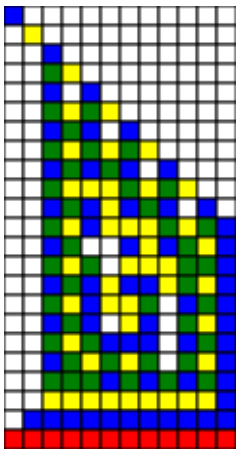

FIG. 2.21 – diagramme espace temps moyen

## 2.4 Approche par signaux

### 2.4.1 Principe et définitions

A ce jour, toutes les solutions au problème des fusiliers sont basées sur ce que l'on appelle les signaux, i.e. la propagation d'une information élémentaire au sein d'une ligne d'automates. Des stratégies les plus simples ("diviser pour régner") aux stratégies plus complexes utilisées par exemple par Balzer et Mazoyer plus tard, toutes propagent des signaux selon des règles bien précises afin de générer des comportements sur l'automate qui se produiront quelle que soit la taille de la ligne de fusiliers.
La compréhension de ces signaux est donc la clé de la découverte de certaines règles nécessaires à la synchronisation des fusiliers en temps optimal. Connaître ces règles nous permet de réduire l'espace de recherche sur les autres approches et ainsi améliorer leur efficacité.

**Quelques définitions se rapportant à ces signaux[MazTer] :**

Comme énoncé plus haut, par signal on entend propagation d'une information élémentaire au sein d'une ligne d'automates.

Un automate tel qu'on en utilise pour le problème des fusiliers (i.e. dont toutes les cellules au temps $t = 0$ sont dans l'état quiescent à l'exception de la première, qui est dans l'état général) est appelé ICA (impulse cellular automaton).

Pour préciser la définition intuitive d'un signal, considérant un diagramme espace-temps, un signal S est un ensemble de sites (i.e. de cellules d'index $k$ à un instant $t$) tel que chaque site à partir du temps $t = 1$, a reçu l'information, soit de celui situé au même index que lui à l'instant précédent, soit de l'un des deux voisins de ce dernier. En d'autres termes, un signal est une propagation continue d'information. Dans notre cas, un signal vers la droite se déplace dans le sens croissant des indexs et un signal vers la gauche dans le sens décroissant.

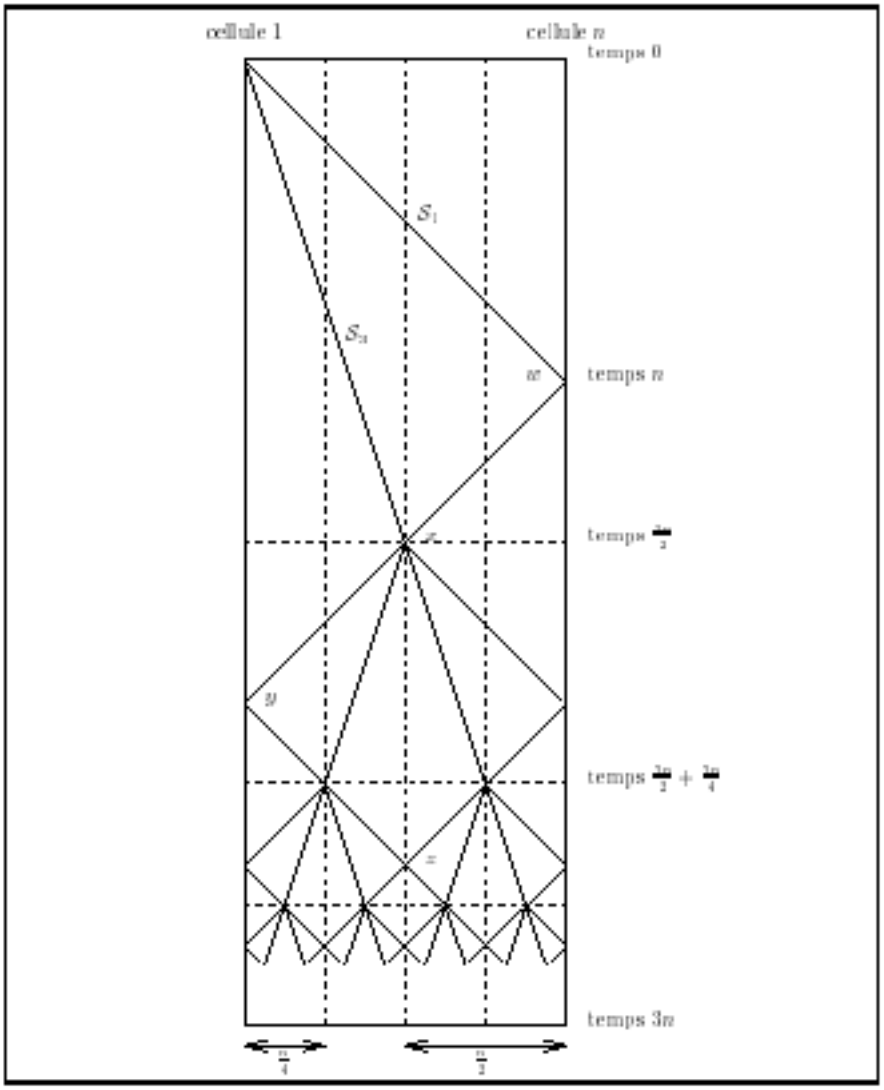


FIG. 1.1 - *Solution de type Minsky*

FIG. 2.22 – Diagramme de signaux basés sur la stratégies "diviser pour régner".

Un signal est dit basique si la séquence des mouvements élémentaires qui le compose est périodique, même si la période n'apparaît qu'après un certain temps. C'est à dire qu'il est possible qu'un signal basique contienne une séquence non périodique avant de finir par devenir effectivement périodique.

Un signal basique est CA constructible[1]. Ce qui signifie qu'il peut être construit par un ICA et donc apparaître dans le problème des fusiliers.

[1]Terme emprunté à l'anglais

Si on nomme $T$ la période d'un signal basique et $U$ la somme des mouvements élémentaires qu'il effectue durant cette période, alors $\frac{T}{U}$ représente la pente de ce signal. On remarque que cette pente est nécessairement supérieur ou égale à 1.

Si $\rho$ est une fonction croissante de $\mathbb{N}$ dans $\mathbb{N}$, $\rho$ désigne le ratio du signal $S$ si $S$ atteint la cellule $n$ précisément au temps $t = r(n)$. La vitesse d'un signal vers la droite est donnée par $\frac{n}{r(n)}$ et est au maximum de 1 puisque $r(n) \geq n$.

**Conséquences pour le problème des fusiliers :**

Des signaux de plusieurs types de ratio sont constructibles :

- quadratiques de la forme $n^k$,
- contenant des racines de la forme $xn + \lfloor\sqrt{n}\rfloor$, $x \in \mathbb{N}^*$,
- contenant des logarithmes de la forme $n + \lfloor\log_x n\rfloor$,
- factorielle de la forme $2(n!)$.

Si ces derniers signaux sont bien constructibles, c'est malheureusement au prix de l'utilisation d'un nombre d'états supérieur à celui auquel nous nous sommes astreint, si bien qu'il apparaît que si solution il y a, elle ne contiendra pas de signal de ces formes-là.

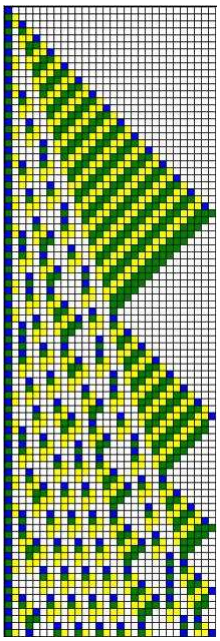

FIG. 2.23 – Signaux proches de ceux utilisés par Mazoyer, avec 5 états seulement.

### 2.4.2 Formes caractéristiques attendues pour le problème des fusiliers :

On se représentera un diagramme espace-temps sur lequel le temps s'écoule vers le bas et les automates sont indexés vers la droite.

Notre but étant la synchronisation en temps optimal, cela implique clairement que l'information doit se propager le plus vite possible le long de la ligne de fusiliers.

Première phase, de la gauche vers la droite :

Peu importe ce qui se produit ailleurs sur le diagramme, un signal doit exister tel qu'il se propage vers la droite et soit de pente 1 pour respecter le temps minimal. Ce signal ne peut prendre que $n-3$ formes (ou $n$ est le nombre d'états dont on dispose) selon les $n-3$ périodes possibles. Dans le cas où l'on n'utiliserait que 5 états comme nous le faisons, les deux formes possibles sont une succession d'états 1 ou une succession des états 1 et 2 alternés. En effet, outre les états 4 et 0 qui sont bien entendu inadaptés à être utilisés à cette fin, il convient de conserver un dernier état en réserve de manière à pouvoir revenir vers la gauche, faute de quoi ce signal repartirait invariablement vers le bord droit immédiatement après l'avoir quitté. Il est aisé de forcer l'apparition de la forme de son choix puisque ce signal est de pente 1 et qu'il va à l'encontre de cellules dans l'état quiescent. Par conséquent l'état de la cellule $k$ à l'instant $t$ ne dépend que de l'état de la cellule $k-1$ à l'instant $t-1$. Il suffit d'une règle de transition par mouvement élémentaire souhaité dans la période.

Ainsi, si l'on souhaite obtenir un signal de période 1 il suffit de fixer la règle suivante :

$1\,0\,0 \rightarrow 1$

Si c'est le signal de période 2 que l'on désire voir apparaître, les deux règles qui suivent y suffisent :

$1\,0\,0 \rightarrow 2 \quad 2\,0\,0 \rightarrow 1$

C'est cette dernière forme que l'on a choisi de fixer pour les différentes métaheuristiques car elle est seule à même de permettre la génération de signaux transversaux et donc de favoriser l'apparition de motifs qui rendront possible la sychronisation pour un nombre élevé de fusiliers. Cette thèse a depuis été confirmée expérimentalement par les résultats obtenus par l'algorithme de backtracking.

Seconde phase, de la droite vers la gauche :

Le retour de l'information s'opère bien différemment puisque de nombreux autres signaux que celui de pente 1 pourront avoir été générés ce qui donnera lieu à des interactions complexes imprévisibles à moins de connaître tous les signaux et leurs interactions. Or, leur connaissance implique à priori qu'ils ont été créés à dessein et que l'on sait déjà le comportement qu'ils adop-

teront, ce qui revient à résoudre le problème à la main.

Il n'est donc pas possible de déduire des contraintes générales sur la seconde phase.

## 2.5 Approches combinées

### 2.5.1 Combinaison avec le backtracking

Cette approche est justifiée car elle permet au programme du backtracking d'avoir une initialisation (règles définies pour n petit, c'est à dire $< 5$) différentes de celle obtenue qu'avec ce dernier. En effet, l'algorithme modifie les règles crées qui se trouvent dans la pile mais ne remonte pas suffisamment haut dans celle-ci pour changer la valeur des règles obtenues au début de l'exécution. Nous avions envisagé (hypothèse lors de nos tests réalisés avec le backtracking uniquement) de modifier manuellement les règles crées avant n=5 pour parcourir une autre zone de l'espace de recherche, mais nos encadrants nous avaient indiqué que c'était plutôt le rôle de la recherche tabou de se charger de cela. En fait, le programme peut le faire mais là encore le temps pour obtenir un résultat est inconnu mais très important (vu la taille de l'espace de recherche).En fournissant des règles différentes au départ, nous allons donc explorer une autre partie de l'espace de recherche.

**C**ela explique l'importance de cette approche.

Pour cette dernière approche, où le backtracking est combiné avec un autre algorithme, il faut que le programme soit exécutable avec seulement un fichier de règles (le format de règles n'était pas compatible avec celles utilisés par le backtracking).

### 2.5.2 Algorithme évolutionnaire et recherche tabou

Nous avons développé un opérateur de recherche de tabou qu'on a combiné avec l'opérateur de mutation normal dans l'algorithme évolutionnaire. Ainsi, nous avons testé cet algorithme en appliquant la recherche tabou (1000 itérations) sur 1% puis 5 % des populations.

### 2.5.3 Résultats

Pour ce dernier algorithme, il n'a pas pu améliorer la solution à 16. Cependant, il nous a permis de trouver d'autres solution à 14 et 16, comme le backtracking. Historique de la solution à 16 : Tout d'abord le Hill Climbing a donné une règle qui synchronise un automate de 2 à 12. Puis cette solution est utilisée par l'algorithme évolutionnaire. Une nouvelle solution permet la synchronisation de 2 à 13. Enfin, cette dernière solution récupérée pour passer en argument au backtracking a permis d'avoir la meilleure solution connue à ce jour qui synchronise en temps minimal de 2 à 16.

C'est avec cette approche, grâce à l'exploitation combinée des résultats obtenus par les différents algorithmes, que nous avons obtenu un résultat inédit : la synchronisation en temps optimal pour $n$ compris entre 2 et 16.

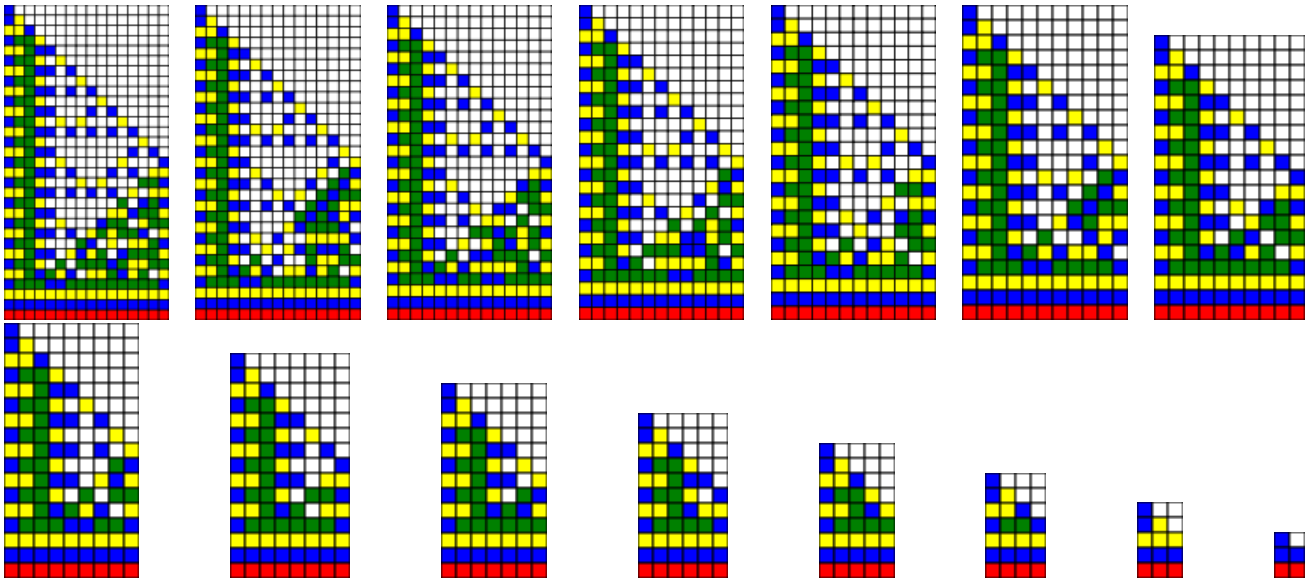

FIG. 2.24 – Une des meilleures solutions obtenues de 2 à 16

# Chapitre 3

# Conclusion

## 3.1 Perspectives

Nous pouvons aussi envisager de faire plus de tests afin de trouver de bons paramètres pour le Recuit Simulé. Aussi, pour l'approche évolutionnaire nous pouvons prévoir une implémentation des opérateurs d'Hill Climbing et de Recuit Simulé.

Amélioration de l'algorithme du backtracking. Cette amélioration n'a pas été réalisée pendant le ter en accord avec nos encadrants, nous avons préféré ne pas approfondir davantage car malgrés le fait que cela paraisse être une bonne idée, nous risquions de manquer de temps pour aborder la dernière approche (il faudrait attendre le nouveau code). L'une des limites de notre cahier des charges étant de ne pas trop creuser la même approche.
L'algorithme peut être amélioré de deux manières :

- Lors de la création des règles. En fait il faudrait faire un premier passage où l'on attribue à chaque cellule de la configuration suivante, la valeur de la règle correspondante (règle formée des trois cellules de la configuration courante). Si la règle a pour valeur NonAffect, on continue quand même l'affectation des valeurs pour les autres cellules.

- lors du retour en arrière, au lieu de modifier la règle se trouvant en sommet de pile (qui ne résout pas forcement le conflit), il faudrait partir de la cellule de la configuraution d'où provient le conflit et remonter dans les configurations pré¸edentes (à chaque fois prendre les trois cellules d'où est issue la cellule posant problème) et remonter également dans la pile jusqu'à cette règle pour la modifier directement.

Ces deux améliorations évitent de nombreux tests inutiles.

## 3.2 Intérêts du TER

Ce travail d'étude et de recherche a été une expérience enrichissante qui nous a permis d'explorer plusieurs métaheuristiques et les algorithmes évolutionaires. Nous avons également utilisé de nouvelles approches pour ce problème telles que le backtracking ou les signaux. Enfin, les résultats obtenus ont dépassé nos espérances puisque nous ne pensions pas parvenir à synchroniser plus

12 ou 13 fusiliers.
Objectif pédagogique : apprentissage du travail en équipe et première expérience d'un travail plutôt orienté recherche.

# Bibliographie

[MAZ87] . Mazoyer, *A six-state minimal solution to the firing squad synchronization problem*, Theoretical Computer Science, Volume 314, Issue 3, 10 April 2004, Pages 303-334.

[Yun93] .B. Yunes, *Synchronisation et automates cellulaires : la ligne de fusiliers*, Thèse(1993).

[MazTer] . Mazoyer, Véronique Terrier, *Signals in one dimensional cellular automata*, Research Report N 94-50, December 1994.

[Ham] ean-Philippe Hamiz et jin-Keo. *12 Congrès on reconnaissance et l'intelligence artificielle -RFIA 2000,Paris(recherche tabou et Planification de rencontres sportive* ,janvier 2000.

[AyMa] oiseph Ayas, MarcAndré Viau.*Recherche Tabou.* 16 janvier 2004.

[Pic] Abien Picarougne. *Thèse : recherche d'information sur Internet par Algorithmes métaheuristiques*,19 novembre 2004.