# Documentation Viab-Cell 1.0

## Modèle

L'idée est de construire des ensembles atteignables à partir d'un état initial (cellules ou forme) et en utilisant tous les contrôles possibles. Dans ce modèle, les contrôles représentent les différentes directions de division que peuvent emprunter une cellule lors de sa mitose (gauche, droite, haut et bas). Successivement, à partir des ensembles atteignables de l'étape précédente, on construit tous ceux de l'étape présente. Chaque ensemble atteignable définit une forme. Une taille maximale est définie pour que si on arrive à une étape où les ensembles atteints sont de cette taille, le programme s'arrête et ces ensembles constitueront ainsi les ensembles finaux.
Pour ce faire, nous disposerons d'objets `Cell` qui évolueront dans des objets `Grid` placés dans un objet `Environment`. Pour avoir une vue sur l'historique des formes finales, à chaque étape courante, nous mettons à jour un graphe dont les sommets d'origines sont les formes obtenus à l'étape précédente et les sommets cibles les formes nouvellement créées. Entre deux sommets, nous établissons un arcs qui mémorise la position de la cellule ayant déclenché la mitose, le contrôle qu'elle a appliqué et le pas de temps où cette transition a eu lieu.

## Fonctionnement

### Initialisation

On définit un environnement avec la taille maximale et éventuellement les formes finales spécifiquement recherchées dans les ensembles finaux(carré par exemple). Puis, on crée une grille avec les dimensions souhaitées en longueur et en largeur. Cette grille peut évidemment défini carrée ou rectangulaire. On crée un `Cellule`et on lui assigne une position dans la grille. Ce sera la position initiale. Enfin, on ajoute la grille nouvellement créée dans l'environnement. On initialise les variables qui vont nous permettre de récupérer à chaque étape le nombre de formes générées.
Il est aussi possible de démarrer le programme en ayant à la place d'une cellule unique un groupe de cellules (une forme). Dans ce cas-ci, l'objectif sera d'étudier les formes atteignables à partir de cette forme initiale.

### Boucle de Simulation

La grille nouvellement ajoutée est parcourue. A chaque position où l'on identifie une cellule, on évalue toutes les dynamiques possibles de cette cellule. C'est à dire, peut-elle déclencher une mitose en haut, en bas, à gauche ou à droite ?

Dans chaque cas possible, on choisit le contrôle adéquat qui permet à la cellule de faire sa mitose. Et comme nous sommes en échelle de temps double, chaque cas de mitose correspond à un ensemble atteignable à partir de cette cellule. En effet, on ne considère pas le pas de temps global(voir figure **??**) où toutes les cellules se sont divisées mais plutôt chaque pas de temps intermédiaire (voir figure **??**) lors duquel une seule cellule de la forme se divise.

FIGURE 1 – Single time scale

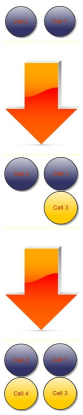


FIGURE 2 – Double time scale

La possibilité d'une mitose dépend d'une part de la position de la cellule mère par rapport aux frontières de la grille et d'autre part de son voisinage. Si la mitose est permise, une cellule fille est créée à la position adéquate par rapport à la position mère. Et nous vérifions que cet ensemble nouvellement atteint n'a pas déjà été préalablement atteint par une autre transition antérieure. Si c'est le cas, on ne l'inscrit pas une seconde fois. On l'identifie dans le graphe et on crée un nouvel arc le reliant à cet autre ensemble d'origine. Par contre, s'il n'existe pas encore, on crée un nouveau noeud dans le graphe le représentant et un nouvel arc le reliant à son ensemble d'origine. Comme toute l'historique de l'évolution est stockée dans le graphe, à la fin de chaque étape, nous supprimons de l'environnement toutes les grilles de l'étape précédente et repartons sur celles de l'étape courante où une grille identifiée par

la forme qu'elle contient n'est présente qu'une seule fois.

### Arrêt

Après avoir atteint tous les ensembles possibles dans une étape, on passe à l'étape suivante en partant de celles-ci. On arrête la construction des formes atteignables lorsque que l'on récupéré toutes les formes possibles composées de nombre de cellules égales à la taille maximale définie à l'initialisation de l'environnement.
A ce stade, on affiche les grilles contenant les formes finales atteintes soit à partir des dernières grilles ajoutées à l'environnement, soit à partir des derniers noeuds ajoutés dans le graphe. L'affichage peut être faite sous console ou par sortie de fichiers.

## Architecture

### Environnement

- composé de une ou plusieurs grilles
- identifié par :
  - les formes finales à atteindre et
  - et le nombres de cellules maximal qu'elles doivent contenir
- peut être réinitialisé
- vérifie si une grille donnée existe déjà dans le graphe ou parmi les grilles courantes de l'environnement.
- effectue les transformations géométriques de forme contenue dans une grille

### Grille

- a une dimension (longueur, largeur)
- contient des cellules
- met à jour les liens entre les cellules
- dispose d'une position courante
- possède une taille courante
- identifie sa première cellule et sa dernière cellules ajoutées
- peut être copiée
- assigne une cellule à une de ses positions
- héberge les mitoses :
  1. on vérifie la cellule pour savoir si elle est allumée. Seules les cellules allumées peuvent initier une mitose.
  2. On vérifie si les conditions de déclenchement d'une mitose sont vérifiées. C'est à dire que la position induite par le contrôle choisi

soit non occupé et soit dans les limites de la grille

3. On crée la cellule fille et met à jour les propriétés de la grille (taille et position courantes)
4. lie la cellule fille nouvellement créée à ses voisins s'ils existent et à sa mère
5. mis à jour du graphe
6. Si la cellule n'est parvenu à se diviser par aucun des contrôles possible, on l'éteint.

- peut être affichée
- peut être évaluée pour vérifier si la forme qu'elle contient est celle définie à l'initialisation de l'environnement.

### Cellule

- dispose de voisins de droite, de gauche, d'en haut et d'en bas
- a une mère cellule
- possède une position
- est soit allumée soit éteinte
- peut désigner et renvoyer ses voisins et sa mère

## Graphe

Nous n'avons pas eu à implémenter une classe de graphes pour notre modèle. Boost propose une bibliothèque évoluée mais relativement très générique où les méthodes et algorithmes peuvent être utilisés sur un large choix de données.
Par exemple, nous avons choisi de représenter les formes par un type qu'elle définit, il s'agit des `dynamic_bitset`. Comme leur nom l'indique, ils fonctionnent comme les bitset de la STL mais leur taille est définie à l'exécution et non à la compilation. Ils permettent de représenter l'occupation de la grille de façon optimale avec une suite de bits où 1 signifie que la position est occupée et 0 qu'elle est libre. Ils offrent aussi un certain nombre de méthodes dont celles qui permettent d'accéder à un bit particulier, de définir ou modifier sa valeur, de la tester... et aussi de faire des opérations de bits usuelles. C'est un choix qui se justifie car nous avons lancé des simulations avec différentes structures de données mais les `dynamic_bitset` semblent être les meilleurs dans le rapport temps/mémoire.

### Structure de données

Le graphe est constitué de noeuds et d'arcs. En fonction de notre architecture logicielle et de l'objectif que nous visons dans ce modèle, nous avons défini les structures de données du graphe de la manière suivante :

- Noeud : contient uniquement une forme, qui peut être soit une forme de départ(racine), soit atteinte (feuille), soit les deux (intermédiaire). Cette forme est représentée par un `dynamic_bitset`. Un noeud peut avoir 0 ou plusieurs arcs d'entrée mais aussi 0 ou plusieurs arcs de sortie.
- Arc : contient les informations de la transition d'une forme à une autre. Elle est représentée par une structure dont chaque champ définit une propriété de la transition (responsable, contrôle, temps). Un arc symbolise une et une seule transition.

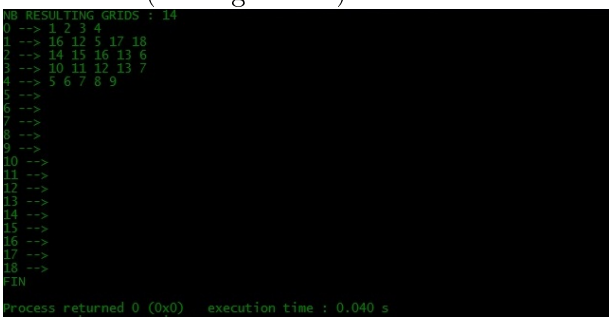
### Construction

Nous désignons comme racine du graphe la grille initiale contenant la forme ou la cellule d'origine. A partir de cette grille initiale, de nouvelles grilles contenant les ensembles atteignables à partir de cette forme d'origine sont générées, chacune représentant un nouveau noeud dans le graphe. Au début de chaque étape de recherche de nouveaux ensembles atteignables, on récupère un itérateur sur les noeuds du graphe créés dans l'étape précédente. Et comme ils ont été créés au fur et à mesure que les grilles sont générées, on fait avancer l'itérateur en même temps que le curseur sur les grilles de l'étape précédente. Ainsi, à chaque fois qu'une nouvelle grille est atteinte à partir d'une grille de l'étape précédente, on sait créer un nouveau noeud cible et un nouvel arc qui le reliera à son noeud d'origine qui n'est autre que le noeud courant indexé par l'itérateur. Cet itérateur ainsi pensé, nous évite à chaque étape de génération de nouvelles formes atteignables, de devoir rechercher dans la liste des noeuds du graphe, celui qui va constituer le noeud de départ, c'est à dire contenant la forme correspondant à celle contenue dans la grille en traitement.
Il peut arriver qu'une grille nouvellement atteinte existe déjà dans l'environnement. Ce qui signifie qu'il existe déjà dans le graphe un noeud qui a la même forme que celle contenue dans cette grille. Donc, il faut juste créer un nouvel le reliant au noeud d'origine considéré. Comme nous disposons d'une variable qui nous compte le nombre de noeuds ajoutés dans une étape, en récupérant un itérateur sur les noeuds du graphe, en commençant par le dernier ajouté et en itérant qu'autant de fois qu'il y a de noeuds ajoutés, nous sommes sûrs de garantir une certaine efficience. C'est à dire, chercher et être sûr de trouver le noeud que dans l'espace de recherche adéquatement défini. Mais cette technique ne marchera pas dès lors qu'on aura introduit l'apoptose comme transition. Car en cas d'apoptose, la forme obtenue ne figurera pas parmi les noeuds de l'étape courante mais parmi ceux précédant. Donc, nous avons envisagé un nouveau moyen de tester l'existence des formes créées non en fonction de leur temps de génération mais plutôt suivant le nombre de cellules qu'elles contiennent (voir Section **??**).

## Résultats

Nous avons testé ce modèle en utilisant d'abord une grille 3x3 où on a placé une cellule unique initiale à différents endroits pour voir quelles sont toutes les formes atteignables à 3 cellules :

– au milieu : 14 formes (voir figure **??** )




– à l'angle : 5 formes (voir figure)


Nous l'avons testé aussi en prenant aussi une forme comme configuration initiale et nous avons choisi de disposer 8 cellules dans la grille initiale tel que sont disposées les premières 8 cellules fondatrices du C-Elegans. Et nous avons observé toutes les évolutions possibles de cette forme pour atteindre 13 cellules.

Nous avons constaté qu'il y a 40.478 formes composées de 13 cellules qui sont atteignables à partir des 8 cellules fondatrices du C-Elegans. Cette fois-ci, nous avons des grilles $10 \times 10$. Pour générer toutes ces grilles, cela a pris **41 minutes** (voir figure **??**).

```
0     0     0     0     0     0     0     0     0     0

0     0     0     0     0     0     0     0     0     0

0     0     0     0     0     0     0     0     0     0

0     0     1     1     0     0     0     1     0     0

0     0     0     0     1     1     1     1     0     0

0     0     1     1     0     0     0     0     0     0

0     0     0     0     0     0     0     0     0     0

0     0     0     0     0     0     0     0     0     0

0     0     0     0     0     0     0     0     0     0

0     0     0     0     0     0     0     0     0     0
==================
N° : 11
==================
```

## Améliorations

Nous avons constaté que les ensembles atteignables croissent de manière exponentielle.
A partir des 8 cellules fondatrices du C-elegans, voici ce que nous obtenons en voulant rechercher toutes les possibilités d'évolution pouvant menant à des formes de nombre de cellule égal à :

- $9 -> 17$
- $10 -> 164 : 0.140s$
- $11 -> 1.197 : 1.638s$
- $12 -> 7.375 : 66.488s$
- $13 -> 40.478 : 2447.750s$

Ainsi, pour ne pas arriver à une explosion combinatoire avec des nombres de cellules relativement abordables, nous préconisons 2 stratégies :

1. Introduire la translation, la rotation et la symétrie.
2. Paralléliser les recherche des ensembles : à chaque étape de création de nouveaux ensembles, la lecture des ensembles de départ se fait parallèlement tandis que ces nouveaux ensembles sont écrits via une mémoire commune. Car l'évolution de chaque ensemble est indépendant de celle

d'une autre alors qu'il doit être garanti la non-redondance entre les nouveaux ensembles.

3. Imposer des contraintes de passage : cette méthode permet de ne calculer qu'un sous-ensemble de l'ensemble atteignable. Elle nous permet d'automatiser la sélection des formes obtenues après un certain nombre de division parmi toutes celles ayant le nombre de cellules fixé au départ. Ainsi, si on connait la structure ou les structures potentielles que peuvent présenter la forme recherchée à un nombre de divisions donnée, on définit un catalogue des formes au début du programme contenant ces structures. Et cette catalogue sera une contrainte de passage pour tous les ensembles atteints au pas de temps correspondant au nombre de divisions attendu. Seule les formes ayant une structure identique à au moins une des formes du catalogue seront retenues pour poursuivre la génération des prochains ensembles. Il est possible de poser des contraintes de passage à plusieurs temps de division, c'est une manière de trouver tous les chemin menant vers chaque étape de développement d'une forme bien ciblée. Ou encore les différentes manière de créer une forme étape par étape. Sous Linux 64 bits, voici le nombre de formes à 9 cellules dans une grille $10 \times 10$ à partir d'une cellule initiale en position 55 :
   - sans aucune contrainte de passage -> 2489 en 1351 s
   - avec les contraintes de passage suivantes(2) : 1) toutes les formes obtenues après 2 divisions complètes seront confrontées à un catalogue composée d'une seule forme à structure carrée 2)après la 3eme division, une deuxième contrainte de passage imposée par un catalogue composé d'une seule forme dont la structure est 2 rangées de 4 cellules parallèles. Nous avons obtenu juste 3 formes générées en 1 s.

4. Implémenter les algorithmes de viabilité sur une structure abstraite : il s'agit d'abstraire l'architecture de sorte que les algorithmes de viabilité ne soient implémentés que par le graphe. Nous avons donc fait une version du programme où est supprimé presque toute la nation d'objet. Il n'y a pas d'objet `Grid` ni d'objet `Cell`. Mais juste un environnement contenant un graphe. Toutes les opération que l'on effectuait sur les grilles sont désormais appliquées directement aux noeuds (mitose, transformations géométriques, affichage etc.). Et toutes les opérations que l'on appliquait aux cellules sont désormais supportées par les bits de la chaîne contenue dans le noeud. De ce fait, il y a plus deux structures parallèles (l'environnement et ses éléments pour la simulation et le graphe pour la sauvegarde de l'historique des formes). Le graphe fait les deux. Exemple : le graphe renvoie la chaîne de bits d'un noeud comme entrée pour l'opération de simulation de la mitose. En sortie nous en obtenons une nouvelle chaîne de bits. Puis on retourne au

graphe pour vérifier son unicité. Soit on l'a identifié comme déjà présent dans un noeud soit on crée un nouveau noeud pour l'y insérer. Cela nous permet de stocker moins de données dans la mémoire durant l'exécution. De ce fait, nous avons eu un gain considérable en espace($-67.67\%$) bien que ça l'est moins en temps ($1.25\%$). Cependant, il faut noter que l'abstraction appauvrit le modèle en termes de détails et de prise en compte de la réalité biologique.

5. Faire tourner l'algorithme sous une autre plateforme : bien que nous ne soyons pas confrontés à un problème de mémoire mais plutôt de temps d'exécution, il est à signaler que sous windows 64 bits, nous sommes limités à 2Go de mémoire. Et comme le code est portable, nous l'avons testé alors sous Linux 64 bits. Nous avons obtenu quelques améliorations en temps d'exécution avec une même configuration :
   - Windows
     2489 formes en 1910 s
   - Linux
     2489 formes en 1351 s

En plus du graphe, nous avons aussi créé un tableau associatif dont la clé est le nombre de cellules et les données les références de tous les noeuds du graphe contenant autant de nombre de cellules. De ce fait, lorsque seront ajoutés au modèle des mécanismes cellulaires plus évolués que la mitose tels que l'apoptose et la différentiation, on n'aura pas à modifier encore la structure des algorithmes.
Ainsi, avant d'ajouter un noeud, on vérifie grâce au nombre de cellules de la forme qu'elle contient, s'il n'existe pas parmi les noeuds existant dans le tableau et associés à ce nombre de cellules, un contenant la même forme.
Pour ce faire, on préconise la méthode suivante :

1. Une structure `map` où on ajoute au fur et à mesure les références des noeuds du graphe avec comme clé le nombre de cellule qu'ils contiennent.
   Avec cette structure, nous avons pu réduire le temps d'exécution de $65\%$ par rapport à la technique de recherche précédemment établie.
2. Lorsqu'un nouvel ensemble atteignable est généré, avant de créer un noeud correspondant dans le graphe, on récupère d'abord à partir du tableau les références de tous les noeuds ayant un même nombre de cellules comme clé.
3. Si le nombre de référence récupéré est nul, on crée alors le noeud dans le graphe et une nouvelle clé dans le `map` correspondant au nombre de cellules du noeud. Ensuite, on ajoute la référence du noeud dans le tableau en l'associant à cette nouvelle clé.
4. Si le nombre de référence est non nul, alors on vérifie si la forme atteinte correspond à une forme parmi celles déjà référencées.

5. Si oui, on crée juste un arc entre le noeud référencé et le noeud d'origine courant. Sinon, on procède à la même vérification mais cette fois pour toutes les transformations géométriques possibles de la forme. En effet, à chaque étape, nous devons avoir la forme une et une seule fois dans le graphe, quelque soit la transformation, du moment où la conservation des angles et des distances (isométrie) est respectée. Nous considérerons la translation. Ensuite, les rotations et retournements. Et enfin les translations de toutes les rotations et retournements.
   - Translations : en plus de conserver les distances et les angles dans les formes qu'elle transforme, elle respecte l'orientation d'origine. On considère qu'il s'agit d'un déplacement de la forme dans la grille. Pour réaliser la translation dans notre modèle, il suffit de faire toutes les opérations de `shift and rotate` du `dynamic_bitset` contenant la forme de 1 à `maxSize-1` bits.
     Avec cette technique, nous avons pu réduire le nombre de grilles à stocker de 29% pour la configuration $10 \times 10$ avec une cellule initiale à la position 10 et 7 comme nombre de cellules des grilles finales à atteindre. Cependant, le temps d'exécution est $10\times$ plus long.
   - Rotations : il faut d'abord définir le barycentre de la forme tel que quelque soit la rotation, toutes les cellules de la forme maintiennent leur distance par rapport à elle.
     - 90 :
       - pour une cellule de position inférieure à celle du barycentre. C'est à dire entre 0 et `(centroid_pos DIV width) * width`
         On pose :
         `spread = (centroid_pos DIV width) * width − (cell_pos DIV width) * width`
         `n = spread DIV width`
         Ainsi :
         `cell_new_col = col_centroid + n`
         Si `cell_col` $<$ `centroid_col` : `cell_new_row = centroid_row − centroid_col + cell_col`
         Sinon : `cell_new_row = centroid_row − centroid_col + cell_col`

       - pour une cellule de position supérieure à celle du barycentre entre `(centroid_pos DIV width) * width` et `maxSize`
         On pose :
         `spread = (cell_pos DIV width) * width − (centroid_pos DIV width) * width`
         `n = spread DIV width`
         Ainsi :
         `cell_new_col = col_centroid − n`
         Si `cell_col` $\leq$ `centroid_col` : `cell_new_row = centroid_row − centroid_col + cell_col`

Sinon : `cell_new_row = centroid_row − centroid_col + cell_col`
Enfin :
`cell_new_pos = cell_new_row * width + cell_new_col`
Cette technique, combinée avec la translation et la rotation à 180 degré a réduit les formes atteintes de 66% avec notre configuration test. Cependant, le temps d'exécution 34× plus important.

- 180 :
  On pose : `spread = centroid_pos - cell_pos`
  Ainsi, `cell_new_pos = centroid_pos + spread`
  Cette technique combinée à la translation a réduit le nombre des formes atteintes de 65% avec notre configuration test. Cependant, le temps d'exécution est 35× plus important.
- 270 : pour ce faire, il faudra faire la composition de ces trois transformations :
  (a) une rotation à 90
  (b) un retournement selon l'axe vertical
  (c) et enfin un retournement suivant l'axe horizontal

  si `cell_col` < `centroid_col` alors on pose :
  `spread = centroid_col − cell_col`
  et `cell_new_row = centroid_row + spread`

  Sinon
  `spread = cell_col − centroid_col`
  et `cell_new_row = centroid_row − spread`

  si `cell_row` < `centroid_row` alors on pose :
  `spread = centroid_row − cell_row`
  et `cell_new_col = centroid_col − spread`

  Sinon
  `spread = cell_row − centroid_row`
  et `cell_new_col = centroid_col + spread`

- Retournement : il s'agit de symétrie orthogonale par rapport à une droite. Donc, il faut d'abord définir un axe de symétrie. L'axe est une ligne de la grille choisie judicieusement telle que toute position de cellule s'y situant reste invariante par la transformation sinon il sera la médiatrice du segment formé par l'ancienne position et la nouvelle position de la cellule
  - axe horizontal : cet axe est déterminé par la position minimale occupée par une cellule dans la forme. On pose :
    `sym_inf = (centroid_pos DIV width) * width` et

`sym_sup` = `(centroid_pos DIV width +1) * width`

Si `cell_pos` < `sym_sup`, on pose :
`spread` = `(sym_sup − cell_pos) DIV width`
Si `(sym_sup − cell_pos) MOD width` $\neq 0$
alors `spread` = `spread` +1
Ainsi, `cell_new_row` = `sym_inf DIV width + spread`

Par contre, si `cell_pos` $\geq$ `sym_sup`, on pose :
`spread` = `(cell_pos - sym_sup) DIV width`

Si `(cell_pos − sym_sup) MOD width` $\neq 0$
alors `spread` = `spread` +1

`cell_new_row` = `sym_sup DIV width − spread`

`cell_new_pos = cell_new_row * width + cell_col` Avec cette technique, nous avons obtenu une réduction du nombre de formes finales de 84% avec un temps d'exécution 10× plus long.

– axe vertical : la symétrie orthogonale de la forme suivant cet axe est donnée par la composition de :
  (a) la rotation à 180 degrés et
  (b) la symétrie orthogonale par rapport à l'axe horizontal
  `sym_inf` = `centroid_pos % width` et
  `sym_sup` = `centroid_pos % width` +1

  Si `cell_col` $\leq$ `sym_inf`, on pose :
  `spread` = `sym_inf − cell_col`

  Ainsi, `cell_new_col` = `sym_sup + spread`

  Par contre, si `cell_col` $\geq$ `sym_sup`, on pose :
  `spread` = `cell_col - sym_sup`

  Et, `cell_new_col` = `sym_inf − spread`

  `cell_new_pos = cell_row * width + cell_new_col` Avec cette technique, nous avons obtenu une réduction du nombre de formes finales de 86% avec un temps d'exécution 15× plus long.

A chaque transformation, nous veillerons à ce que toutes les nouvelles positions calculées pour chaque cellule de la forme soient valides. c'est à dire que la transformation n'a pas projeté la forme hors

de la grille. Pour ce faire, nous vérifierons les extrema de la forme (`position_min` et `position_max`).
- Si `position_min` $< 0$, alors pour `n` allant de $1 \rightarrow$ `width`, si `position_min` + `n * width` $\geq 0$, alors les nouvelles positions par translation seront : `cell_new_pos` = `cell_pos` + `n * width`.
- Si `position_max` $\geq$ `maxSize`, alors pour `n` allant de $1 \rightarrow$ `width`, si `position_max` − `n * width` $<$ `maxSize`, alors les nouvelles positions par translation seront : `cell_new_pos` = `cell_pos` − `n * width`.

Lorsque toutes les transformations géométriques de la forme sont prises en compte avant de la considérer comme distincte et l'ajouter, on obtient une réduction du nombre final de formes de 87% par rapport à celui obtenu avec la même configuration mais sans considérer ces transformations. Avec un temps d'exécution relativement plus important 10× (2 secondes).

6. Si ni la forme, ni ses translations et rotations ne correspondent à une forme déjà référencée, là seulement on crée un noeud et ajoute sa référence dans le `map` en l'associant à la bonne clé.

## Études de performance

- 7 cellules : 107 grilles en 2 s
- 8 cellules : 311 grilles en 20 s
- 9 cellules : 958 grilles en 208 s