CONCOURS CENTRALE•SUPÉLEC

Sujet 10 — MPI

Le jeu de la vie dans le jeu de la vie

Durée : 3h

Centrale-Supélec

Préambule

Ce sujet d'oral d'informatique est à traiter, sauf mention contraire, en respectant l'ordre du document. Votre examinatrice ou votre examinateur peut vous proposer en cours d'épreuve de traiter une autre partie, afin d'évaluer au mieux vos compétences.

Le sujet comporte plusieurs types de questions. Les questions sont différenciées par une icône au début de leur intitulé :

- les questions marquées avec ⌨ nécessitent d'écrire un programme dans le langage demandé. Le jury sera attentif à la clarté du style de programmation, à la qualité du code produit et au fait qu'il compile et s'exécute correctement ;
- les questions marquées avec 🧍 sont des questions à préparer pour présenter la réponse à l'oral lors d'un passage de l'examinatrice ou l'examinateur. Sauf indication contraire, elles ne nécessitent pas d'appeler immédiatement l'examinatrice ou l'examinateur. Une fois la réponse préparée, vous pouvez aborder les questions suivantes ;
- les questions marquées avec ✍ sont à rédiger sur une feuille, qui sera remise au jury en fin d'épreuve.

Votre examinatrice ou votre examinateur effectuera au cours de l'épreuve des passages fréquents pour suivre votre avancement. En cas de besoin, vous pouvez signaler que vous sollicitez explicitement son passage. Cette demande sera satisfaite en tenant également compte des contraintes d'évaluation des autres candidates et candidats.

1 Introduction

Bien que défini par des règles très simples, il a été montré que le jeu de la vie pouvait faire émerger des comportements complexes, au point même de pouvoir simuler l'exécution de n'importe quel programme informatique. L'objectif de ce sujet est d'illustrer ce propos sur un exemple ambitieux : simuler une version grande échelle du jeu de la vie, dans le jeu de la vie.

Les trois parties de ce sujet sont à traiter en langage C, et permettront de progressivement construire un algorithme pour la simulation efficace de grandes instances du jeu de la vie : d'abord en implémentant une structure de données nommée quadtree, puis en canonisant les arbres pour un partage mémoire maximal, et enfin en définissant un algortihme d'évolution récursif exploitant le principe de mémoïsation.

1.1 Règles du jeu de la vie

Le jeu de la vie se déroule sur une grille carrée bidimensionnelle, dont les cases sont appelées *cellules* et sont soit mortes (en blanc), soit vivantes (en noir). À chaque nouvelle *génération*, les cellules sont simultanément mises à jour en fonction de leur état et de celui de leurs (au plus) huit cellules voisines (orthogonalement et diagonalement).

Règles du jeu de la vie

- ☐ une cellule morte devient vivante si elle a exactement 3 cellules voisines vivantes ;
- ☐ une cellule vivante devient morte si elle n'a pas exactement 2 ou 3 cellules voisines vivantes ;
- ☐ sinon, la cellule garde son état.

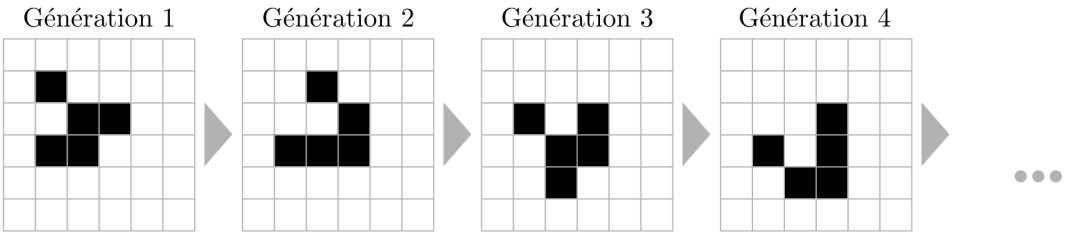
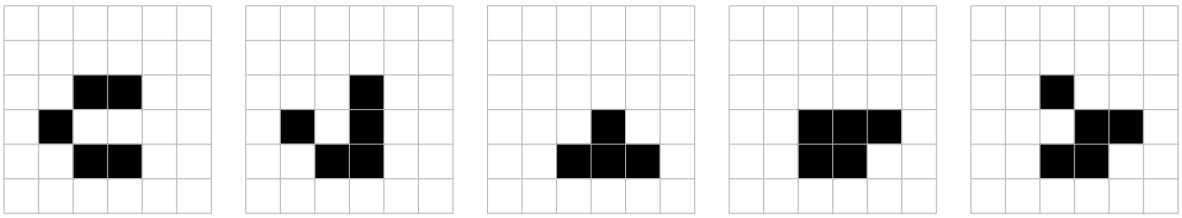


FIGURE 1 – Un exemple d'instance du jeu de la vie qui évolue sur quatre générations.

▷ **Question 1.** ✍ Parmi les cinq grilles suivantes, laquelle correspond à la génération 5 de la figure 1 ?



1.2 La métacellule

En 2006, Brice Due parvient à construire une *métacellule* nommée OTCAMP (Outer Totalistic Cellular Automaton Meta Pixel). Cet assemblage vertigineux de 2048×2048 cellules peut visuellement simuler un état de vie et un état de mort, à l'image d'une cellule standard. Lorsque l'on juxtapose côte à côte plusieurs de ces blocs, un réseau de circuits permet de mettre à jour l'état des métacellules en fonction de ses voisines, en suivant les règles du jeu de la vie. Il faut exactement 35328 générations pour qu'une telle *métagénération* se produise.

1.3 Fichiers et éléments fournis

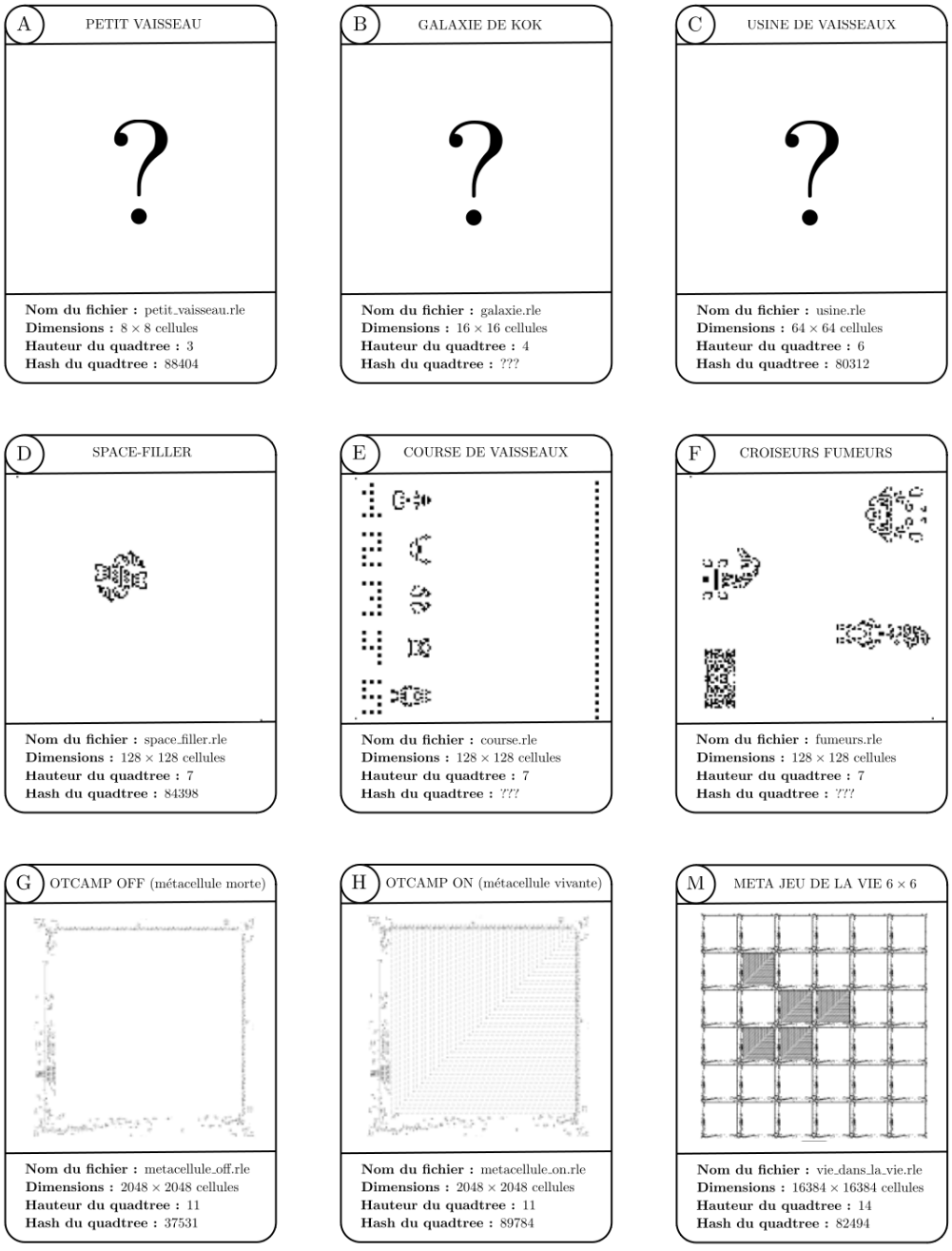


FIGURE 2 – Collection de grilles du jeu de la vie mises à disposition dans ce TP.

Ce sujet contient deux squelettes de code `quadtree.c` et `hashlife.c`, leur en-têtes respectives `quadtree.h` et `hashlife.h`, et un fichier de compilation `Makefile` (qu'il n'est pas nécessaire de lire) qui permettra de compiler le code en un exécutable `life`. Pour cela, il suffit de taper la commande `make safe` (pour un exécutable plus lent mais utlisant des sanitizers), ou `make fast` pour un exécutable plus rapide.

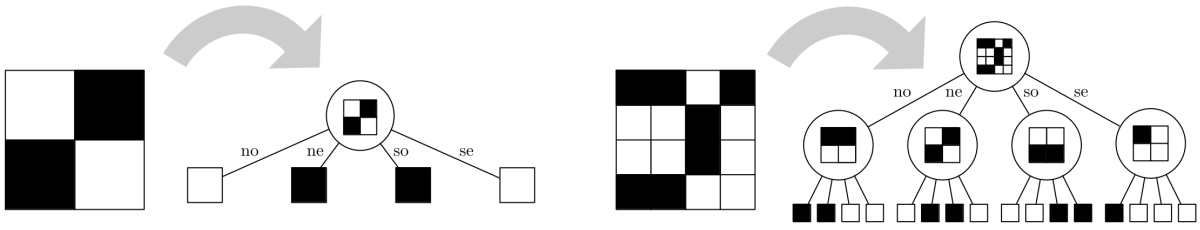
Le dossier `data` contient un exemple de fichier image `.pbm` nommé `course.pbm`, un GIF `metacellules.gif` et une collection de neuf grilles détaillées dans la figure 2.

▷ **Question 2.** ✍ Regarder le GIF `metacellules.gif` situé dans le dossier `data`, illustrant l'objectif de ce TP : une simulation du jeu de la vie à l'aide de 36 metacellules (grille (M) de la collection). Estimer rapidement au brouillon le temps d'exécution que prendrait un algorithme naïf pour simuler une vingtaine de métagénérations sur cette grille.
Vous reste-t-il assez de temps avant la fin de l'épreuve ?

2 Représentation de la grille sous forme de quadtrees

L'objectif de cette partie est de compléter le squelette de code `quadtree.c`, qui permet l'importation, la manipulation et l'exportation de grilles de jeu sous forme de *quadtrees*.

Un *quadtree* est soit une feuille t_0 associée à une cellule morte, une feuille t_1 associée à une cellule vivante, soit un nœud interne représentant une grille carrée de dimension $2^h \times 2^h$, pour $h \geq 1$ un entier appelé la *hauteur* du quadtree. Chaque nœud interne est d'arité exactement quatre, ses quatre fils représentant une partition de la grille en quatre quadrants carrés de dimensions $2^{h-1} \times 2^{h-1}$ appelés Nord-Ouest (no), Nord-Est (ne), Sud-Ouest (so) et Sud-Est (se), comme illustré en figure 3.



(a) (b)

FIGURE 3 – En (a), un quadtree de hauteur 1 représentant une image de dimension 2×2. En (b), un quadtree de hauteur 2 représentant une image de dimension 4×4. Chaque nœud partitionne l'image en quatre quadrants de mêmes dimensions.

Le fichier d'en-tête `quadtree.h` contient un type structuré `quadtree`, dont les champs sont quatre pointeurs vers ses quadtrees fils `no`, `ne`, `so`, `se` (`NULL` si ce quadtree est une feuille), mais également le nombre de cellules vivantes de la sous-grille représentée par ce nœud (`int nb_vivantes`), la hauteur du nœud (`int hauteur`) et un hash associé au sous-quadtree dont ce nœud est la racine (`int hash`), défini par :

$$H(t_0) = 0, \qquad H(t_1) = 1,$$

$$H(t(no, ne, so, se)) = (a \times H(no) + b \times H(ne) + c \times H(so) + d \times H(se) + e) \mod 100000$$

avec a, b, c, d et e des constantes prédéfinies dans la fonction `hash` de `quadtree.c`.

2.1 Importation des grilles sous forme de quadtrees

Les grilles mises à disposition durant cette épreuve sont des fichiers compressés au format RLE (Run Length Encoding). Le squelette `quadtree.c` contient une fonction de signature `quadtree* importe_quadtree_rle(char* nom_de_fichier)` permettant de construire un quadtree depuis un fichier RLE dont le nom a été donné en argument. Il n'est pas demandé de lire cette fonction, ni de chercher à comprendre le principe de la compression RLE. On pourra se contenter d'utiliser cette fonction pour importer des quadtrees, comme dans la fonction `main` présente dans le fichier `hashlife.c` qui propose un exemple minimal que vous pourrez modifier tout au long du sujet.

▷ **Question 3.** Expliquer le principe de fonctionnement de deux algorithmes de compression qui auraient pu être utilisés, et discuter de leur pertinence sur ce type de données.

▷ **Question 4.** Modifier la fonction `assemble_quadtree` afin de calculer et stocker le nombre de cellules vivantes, la hauteur et le hash du nouveau quadtree ainsi formé. Quel sont les hashs des quadtrees obtenus en important les grilles (A) et (B) ?

▷ **Question 5.** En revanche, les hashs obtenus pour les quadtrees des grilles (C) et (D) sont incorrects. Pourquoi la fonction `hash` renvoie-t-elle parfois un résultat erroné, voire négatif ? Proposer une modification simple et comparer les nouveaux hashs obtenus avec ceux attendus. Noter les hashs obtenus pour (E) et (F).

2.2 Exportation des quadtrees sous forme d'images

▷ **Question 6.** Compléter l'implémentation de la fonction de prototype `int couleur_cellule(int x, int y, quadtree* t)` qui renvoie la couleur du pixel de coordonnées (x, y) dans l'image représentée par le quadtree `t`. Vous pourrez vérifier à la question suivante que votre orientation des axes convient.

Afin d'exporter les grilles représentées par des quadtrees, nous utiliserons le format `PBM` permettant d'encoder les images par un texte structuré de manière simple. La première ligne d'un fichier `PBM` ne contient que le mot "P1", précisant le format d'encodage. La deuxième ligne est formée d'un entier désignant la largeur de l'image, d'un espace, puis d'un second entier désignant la hauteur de l'image. Les lignes suivantes correspondent aux pixels de l'image écrits en clair : un 0 pour un pixel blanc et un 1 pour un pixel noir. En guise d'exemple, le fichier `course.pbm` est fourni.

▷ **Question 7.** Consulter les fonctions de manipulation de fichiers `fopen`, `fprintf` et `fclose` dans la documentation mise à disposition. Compléter la fonction de prototype `void exporte_quadtree(quadtree* t, char* nom)` afin de générer un fichier `nom.pbm` depuis un quadtree `t`. **Ne pas exporter l'image (M) qui est bien trop grande.** Garder une trace de l'exportation des images mystères de la collection (A), (B) et (C).

2.3 Manipulation de quadtrees

Le *centre* d'un quadtree de hauteur $h \geq 2$ est le quadtree de hauteur $h-1$ représentant la partie centrale d'une image, comme illustré dans la figure 4.

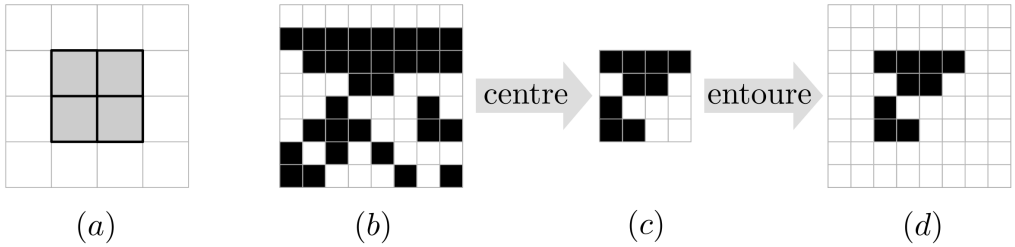


FIGURE 4 – Le centre d'une image correspond à l'aire grisée sur le schéma (a). Sur le quadtree de hauteur 3 représentant l'image (b), la fonction `centre` renvoie le quadtree de hauteur 2 représentant l'image (c). La fonction `entoure` appliquée à ce dernier renvoie le quadtree représentant l'image (d).

▷ **Question 8.** ⌨✍ Implémenter une fonction de prototype `quadtree* centre(quadtree* t)` qui renvoie le centre d'un quadtree. On veillera à n'allouer qu'un seul nouveau nœud. Quel est le hash du centre de la grille (D) ?

▷ **Question 9.** ⌨✍ En utilisant la fonction `feuille`, implémenter une fonction `quadtree* blanc(int hauteur)` qui alloue et initialise un quadtree d'une hauteur donnée, représentant une grille toute blanche. Chaque fils doit être alloué séparément. Noter le hash d'un tel quadtree de hauteur 10.

▷ **Question 10.** ⌨✍ Implémenter une fonction de prototype `quadtree* entoure(quadtree* t)` qui renvoie un quadtree dont le centre est le quadtree `t`, et dont la bordure restante est entièrement blanche. Quel est le hash de (D) entourée ? La libération mémoire de ces nouveaux quadtrees sera assurée dans la partie suivante.

3 Canonisation des quadtrees

Tous ces quadtrees peuvent prendre beaucoup de place en mémoire. L'objectif de cette partie est de modifier les fonctions définies précedemment dans `quadtree.c` afin de ne jamais avoir à stocker deux fois un même quadtree (ou sous-quadtree) durant toute l'exécution du programme. Pour cela, tous les nœuds de quadtrees seront stockés dans une table de hachage, afin de rapidement identifier si un quadtree est déjà présent. L'unique nœud stocké représentant un quadtree sera appelé sa version *canonique*.

▷ **Question 11.** 🕴 Rappeler le principe d'une table de hachage et proposer un exemple pertinent permettant d'illustrer son intérêt.

Dans ce sujet, nous proposons une implémentation simple, dont la signature est déjà présente dans le fichier d'en-tête `quadtree.h`. Une table de hachage sera un tableau, de taille `HASH_MAX` (100 000) ici, dont chaque case pointe vers une liste simplement chaînée de quadtrees, implémentée par la structure `noeud`. En plus de cela, la structure de table de hachage contient un entier désignant sa taille, deux pointeurs représentant les deux quadtrees canoniques de hauteur 0 (la feuille blanche et la feuille noire), ainsi que le nombre de quadtrees contenus dans la table, sans compter les deux feuilles, qui ne sont pas stockées dans le tableau.

▷ **Question 12.** ⌨ Implémenter une fonction permettant d'allouer et initialiser une telle table de hachage, intialement vide hormis les deux quadtrees canoniques de hauteur 0 alloués avec la fonction `allocation_feuille` (et non `feuille`).

▷ **Question 13.** ⌨ Implémenter une fonction permettant la libération mémoire d'une telle table de hachage. Cette dernière devra également libérer tous les quadtrees présents dans la table.

Les autres fonctions qui auront besoin d'une feuille devront désormais récupérer la feuille canonique depuis la table de hachage.

▷ **Question 14.** ⌨ Modifier la fonction `feuille` afin de ne pas allouer de nouvelle feuille mais renvoyer directement l'une des deux feuilles canoniques. Il n'est pas demandé de modifier la signature de la fonction `feuille` pour prendre une table de hachage en argument, mais plutôt d'ajouter la ligne `extern table_hachage* ht;` en début de fonction, qui récupérera une table de hachage définie globalement dans `hashlife.c`. Il n'est pas demandé de comprendre finement l'emploi du mot-clé `extern`.

Afin de pouvoir correctement libérer les quadtrees, il reste à prendre en compte la fonction `assemble_quadtrees`. Il suffira ensuite de libérer la table et son contenu, et il ne sera plus nécessaire d'utiliser la fonction `free_quadtree`. Il est normal que la libération ne fonctionne pas tant que vous n'avez pas traité la question suivante.

Lorsqu'un nouveau quadtree est sur le point d'être créé par `assemble_quadtrees`, on regarde d'abord la case de la table de hachage dont l'indice est le hash de ce quadtree. Cette case pointe vers la liste de tous les quadtrees canoniques déjà créés qui partagent ce même hash. Si jamais le quadtree en cours de création est déjà dans cette liste, on se contente de renvoyer le pointeur trouvé en question. Sinon, c'est que l'on rencontre ce quadtree pour la première fois : on l'alloue alors comme avant, et on l'insère dans la liste.

Il est donc nécessaire de tester l'égalité entre le quadtree en cours de création, et chaque quadtree présent dans la table ayant le même hash que lui. On pourra remarquer ici que ce test peut se faire en temps constant, grâce à l'égalité physique des pointeurs : deux quadtrees ayant les mêmes quatre fils canoniques sont égaux.

▷ **Question 15.** ⌨ Modifier la fonction `assemble_quadtrees`, en y ajoutant la ligne `extern table_hachage* ht;`, afin de ne pas allouer de nœud qui serait déjà présent dans la table de hachage. Cette dernière doit être mise à jour si ce nouveau nœud n'y était pas.

▷ **Question 16.** ⌨✍ Proposer une modification de `blanc` permettant d'améliorer sa complexité temporelle en exploitant la table de hachage des quadtrees canoniques. Quelle nouvelle complexité obtient-on ?

▷ **Question 17.** ⌨✍ Avec cette méthode, combien de nœuds sont nécessaires pour représenter en mémoire les images (E) et (F) ? Quel ratio de compression obtenez-vous ?

4 Mise à jour de la grille avec Hashlife

Les quadtrees canonisés permettent une optimisation mémoire notable, mais vont de surcroît servir à concevoir un algorithme extrêmement efficace pour le calcul des générations suivantes d'une grille, nommé *Hashlife*.

4.1 Une approche diviser pour régner

L'objectif de cette partie est de compléter le squelette de code `hashlife.c` afin d'obtenir une fonction qui prend en entrée un quadtree et qui renvoie la génération suivante **de son centre**. Pour calculer la génération suivante d'une grille, il suffira donc d'entourer cette grille par une bordure blanche, puis d'appeler la fonction d'évolution dessus.

En tant que cas de base, les quadtrees de hauteur 2 sont traités dans la fonction `evolution_4x4`, qui calcule la génération suivante du centre de manière naïve en suivant les règles du jeu de la vie. Pour les quadtree de hauteur plus grande, on suit le schéma récursif illustré en figure 5, en commençant par diviser la grille en 9 sous-grilles pouvant se chevaucher.

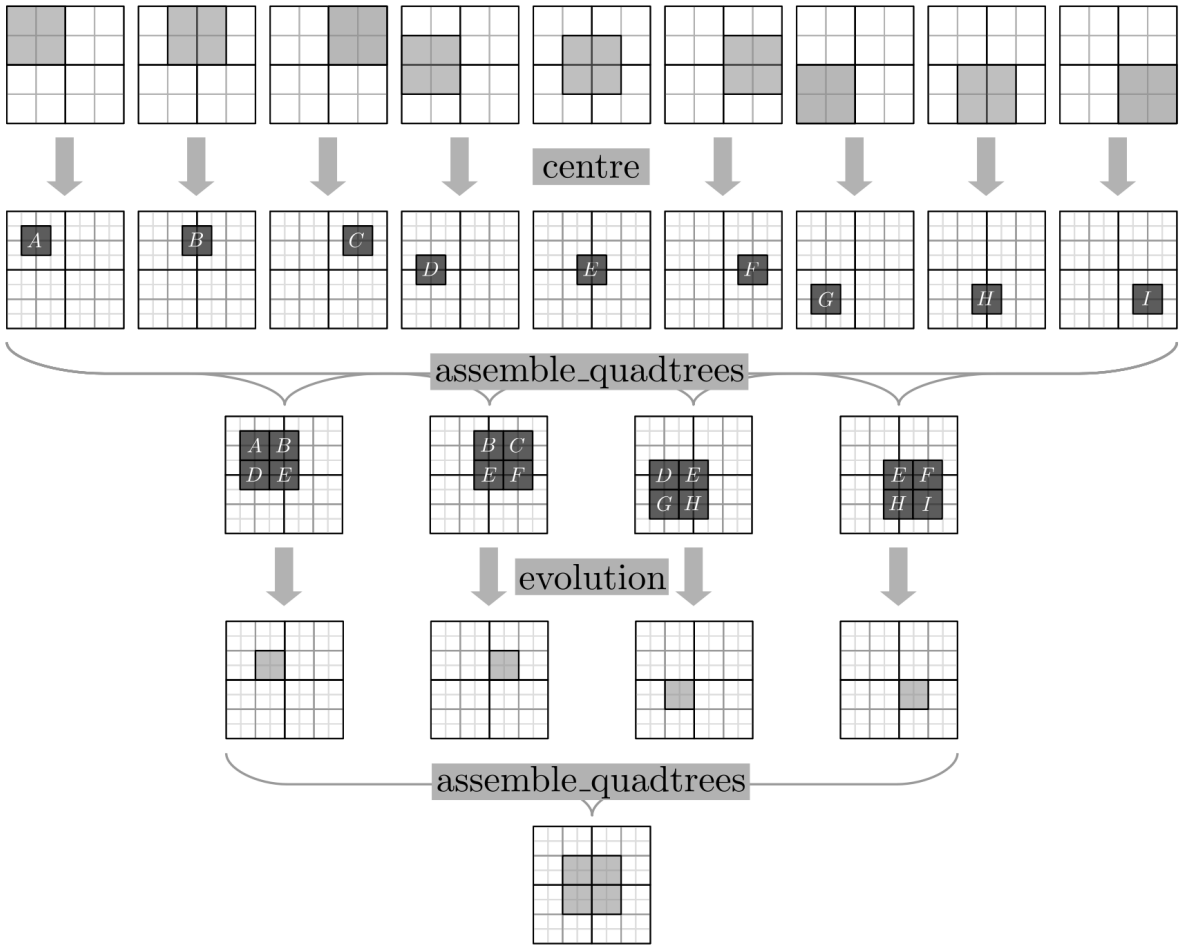


FIGURE 5 – Fonction d'évolution récursive à implémenter dans la fonction `evolution` du fichier `hashlife.c`.

▷ **Question 18.** ⌨✍ Compléter l'implémentation de la fonction de prototype `quadtree* evolution(quadtree* t)`, en suivant le schéma récursif proposé. Quel hash obtient-on pour le centre de la génération suivante de la grille (D) ?

▷ **Question 19.** ⌨ Compléter la fonction de prototype `void simulation(quadtree* t, int nb_etapes, char* nom_dossier)` qui fait évoluer une grille sur `nb_etapes` générations, en exportant une image de chaque génération dans le dossier nommé `nom_dossier`.

Une fois toutes ces images générées dans un dossier, il est possible de s'y rendre avec un terminal, et de générer un GIF avec la commande :

```
ffmpeg -i %d.pbm output.gif
```

▷ **Question 20.** ✍ Qui remporte la course de vaisseaux de la grille (E) ?
Garder le GIF généré et noter le classement final des cinq participants.

4.2 Hyperspeed

Dans le jeu de la vie, la *vitesse de la lumière* correspond à la vitesse maximale à laquelle une information peut se propager, et est d'un déplacement diagonal de celulle par génération.

Ainsi, si l'on connait une bordure suffisament grande autour d'un centre, on pourra prédire ce que deviendra ce centre au bout d'un grand nombre de générations, peu importe ce qui existe au-delà de la bordure. En effet, ce qui existe en dehors de la bordure n'aura même pas le temps d'interagir avec ce centre.

En exploitant le fait que la vitesse de la lumière soit limitée, il est donc possible d'avancer à grands pas vers le futur pour atteindre des générations vertigineusement lointaines **sans avoir à calculer chaque génération intermédiaire**. C'est ce qui permet l'explosion des performances obtenue avec l'*Hyperspeed*, qui sera implémenté dans cette partie.

▷ **Question 21.** ⌨✍ Modifier la fonction `evolution` en remplaçant les appels à la fonction `centre` par d'autres appels récursifs à `evolution`. De combien de générations dans le futur cette nouvelle fonction avance-t-elle à chaque appel ?

▷ **Question 22.** ⌨✍ Proposer et implémenter une modification simple de la structure de quadtree qui permette à la fonction `evolution` de directement renvoyer le résultat si ce quadtree a déjà été traité auparavant. Obtient-on un gain de temps important ?

▷ **Question 23.** ⌨ Modifier les fonctions `export_quadtree` et `couleur_cellule` afin de pouvoir exporter des images avec une définition plus faible. Attention à ne jamais générer d'images ayant plus de 256 pixels de côté.

Vous pouvez désormais simuler le jeu de la vie dans le jeu de la vie.

5 Ouverture

▷ **Question 24.** ⌨✍ Proposer une optimisation de la table de hachage afin de limiter l'empreinte mémoire du programme. Noter toute initiative prise, ou toute idée qui aurait pu être implémentée avec davantage de temps.