Projet de semestre - jeu de la vie

Le programme lit les paramètres de jeu et initialise deux tableaux de nbL fois nbC. Il imprime ensuite l'en-tête et le tableau initial. La fonction ***start*** est appelée et elle va appeler nbJ fois la fonction ***analyse***, pour les étapes impaires, on analyse le tableau 1 et la mise à jour est écrite dans le tableau 2, pour les étapes paires, c'est l'inverse, ceci en inversant simplement les tableaux lors de l'appel de ***analyse***. La fonction ***analyse*** parcourt chaque case du tableau et à chaque fois, elle détermine où la case se trouve (dans un coin, une bordure, ou au centre). Une fois la position déterminée, elle appelle la fonction ***caseViv*** avec un nombre compris entre 0 et 15 en décimal, qui est une sorte de code déterminant la position de la case basée sur un système binaire proche de la commande *chmod* de linux. Le nombre est donc composé de 4 chiffres binaires, chacun exprimant si oui ou non la case touche un bord. Le nombre s'écrit donc **[haut|droite|gauche|bas]**. Par exemple, le code 1010 correspond à la case dans le coin en haut à gauche du tableau.[1] La fonction ***caseViv*** va donc appeler la fonction correspondant à la position de la case à analyse qui permet de compter le nombre de voisins vivants de la case en sachant à l'avance quelles sont les cases non définies et gagner un coût calcul non négligeable. Ces fonctions de comptages sont nommées ***voisin*Position**** et retournent le nombre de voisins vivants. ***caseViv*** retourne ensuite si la cellule actuelle doit être vivante à l'étape suivante à ***analyse***.

Une fois tout le tableau analysé, la fonction ***start*** détermine s'il convient d'afficher l'étape en fonction de nbS et si oui, on dissocie pour les cas pairs et impairs le tableau à imprimer (pair = tableau 1). Cette fonction d'impression (***output***) imprime une ligne noire avant d'imprimer le tableau agrandi zoom fois (sauf à la première impression) en respectant le nombre de caractères maximum.

Pseudocode

Start
```
Étape ← 1
tant que étape <= nbJ
  si étape paire
    analyse(tab2, tab1)
  sinon
    analyse(tab1, tab2)
  si nbS !=0 et nbS divise étape
    si étape paire
      output(tab1, étape)
    sinon
      output(tab2, étape)
```

Analyse(t1, t2)
```
Pour chaque case i de t1
  si case n'est pas bordure
    t2[i]=caseViv(t1, centre)
  sinon
    trouver type de bordure
    t2[i]=caseViv(t1, type)
```

Voisin*position*(t1)
```
count=nb cases viv. Autours
sortir count
```

CaseViv(t1, type)
```
switch(type)
  cas CENTRE :
    count=voisinCentre(t1)
  cas BAS :
    count=voisinBas(t1)
  …
si t1[i] vivant
  sortir (count==3)?vivant : mort
sinon
  sortir (count==2|3)?vivant :mort
```

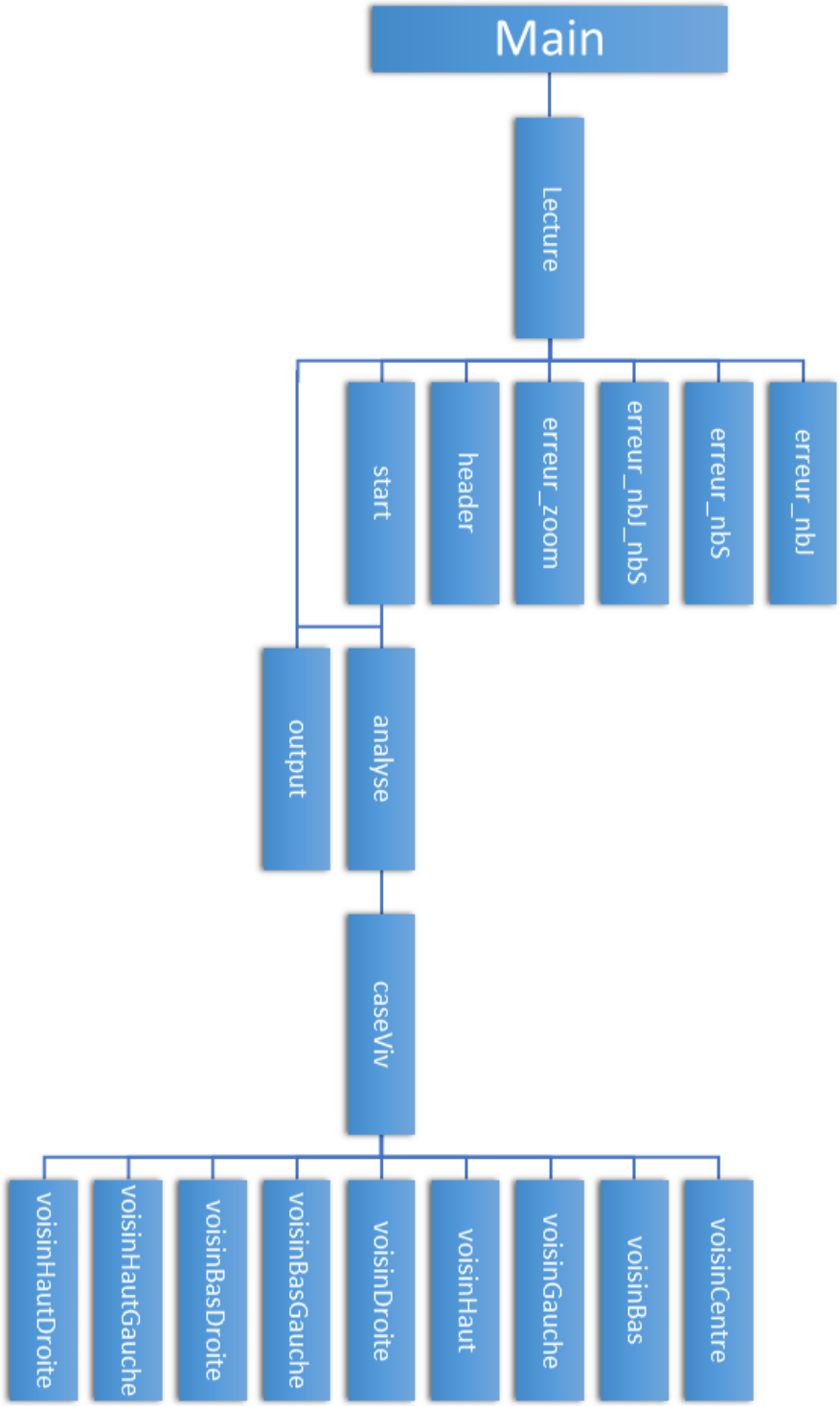
Output(t1, étape)
```
char=0
si étape différent de 0
  pour i à (nbc · zoom)
    si char == MAX
      retour à la ligne
    imprime '1 '
  char=0
pour i à ligne
  pour k à zoom
    pour j à colonne
      pour l à zoom
        si char == MAX
      retour à la ligne
    imprime tab[i*col+j]
```

Complexité

L'algorithme a une complexité $\mathcal{O}(nbJ \cdot nbL \cdot nbS + \frac{nbJ^2}{nbS} \cdot nbC \cdot zoom + \left(\frac{nbJ^2}{nbS} + 1\right) \cdot nbC \cdot nbL \cdot zoom^2)$. Avec la première partie correspondant aux mises à jours pour les nbJ étapes entre les deux tableaux, le deuxième membre pour l'affichage de lignes de 1, et le dernier membre pour l'affichage des tableaux. La méthode utilisée dans cette version est probablement la meilleure si on prend en compte le coût calcul et le coût temporel, les différences sont notables particulièrement lors de grandes matrices carrées.

[1] Pour des informations concrètes avec un schéma ASCII, voir le fichier doc.txt qui est inclus dans mon rendu conway.zip sur https://github.com/dxs/EPFL/blob/master/ba-1/prog/projet1/final/doc.txt

Call graph


Main
Lecture
start
header
erreur_zoom
erreur_nbJ_nbS
erreur_nbS
erreur_nbJ
output
analyse
caseViv
voisinHautDroite
voisinHautGauche
voisinBasDroite
voisinBasGauche
voisinDroite
voisinHaut
voisinGauche
voisinBas
voisinCentre