PROJET D'ALGORITHMIQUE

«Same Gnome»

LAPLACE Jordan
CISSE Coumba
Tuteur: M.KESSACI

GIS3
2017-2018

Sommaire

Introduction

Le démineur, le solitaire ou encore le mahjongg sont autant de jeu fournie de base avec un system d'exploitation. Nous pouvons constater qu'entre plusieurs systems d'exploitation des aspects du jeu varie . Comme par exemple le démineur où l'on pouvait tomber au premier clic sur une bombe dans les anciennes versions de Windows . Ceci prouve qu'en informatique il existe une infinité de façons de programmer une même chose . Notre objectif lors de ce projet était de programmer le jeu same gnome de la façon la plus optimale possible. C'est-à-dire avec un minimum de temps de calcul . Nous allons, au cours de ce rapport vous expliquer notre analyse du problème ainsi que ça résolution .

Cahier des Charges

1) Présentation du jeu Same GNOME

Same Gnome est un jeu de réflexion. Le but de la version que nous devons réaliser consiste à enlever toutes les billes d'un plateau en marquant le plus de points possible. Un groupe de billes identiques adjacentes disparaît d'un seul bloc. Les billes restantes s'écroulent dans les espaces libérées et de nouveaux groupes se forment.

Au début, le plateau (10 lignes et 15 colonnes) est entièrement rempli aléatoirement de billes de couleurs différentes. Il y a 3 couleurs de billes (rouge, vert , bleu). Le déroulement d'un coup de jeu se décompose de la manière suivante :

- L'utilisateur sélectionne une bille
- Le groupe de bille de même couleur dont fais partie la bille sélectionnée et qui se touchent (verticalement ou horizontalement) seront enlevées. Plus il y a de billes dans un groupe, plus vous obtenez de points en le faisant disparaître. À chaque coup, le joueur gagne $(n-1)^2$ points, n étant le nombre de billes enlevées.
- Lorsque des billes sont enlevées, celles qui se trouvent au-dessus tombent dans les espaces vides.
- Si une ou plusieurs colonnes entières du plateau sont libérées, les billes sont poussées vers la gauche pour combler l'espace ainsi crée.

Le jeu s'arrête lorsqu'il n'y a plus de billes

2) contrainte d'affichages

L'affichage de l'état de la matrice de Jeu se fait à chaque étape d'un coup de Jeu.

Étape 1:

On affiche le plateau de jeu ainsi que les points du joueur. Si le joueur à au moins jouer il s'agit aussi de l'afficha après décalage des colonnes vides.

Étape 2:

Après sélection de la bille à retirer par l'utilisateur. un nouveau plateau s'affiche obtenu après enlèvement du groupe de billes connexes de même couleur. Les billes retirées sont représentées par le caractère "X".

Étape 3:

Affichage du plateau après l'écroulement.

3) contrainte saisie

La saisie des coups du joueur doit respecter le format: colonne ligne.

4) contrainte programmation

Pour visualiser le plateau de jeu ,nous avons remplacé caractères "R'", "V'"," B", "." et "x" par des couleurs:

- Le caractère R est remplacé par des espaces sous fond rouge.
- Le caractère V est remplacé par des espaces sous fond vert.
- Le caractère B est remplacé par des espaces sous fond bleu.
- Le caractère x est remplacé par des espaces sous fond jaune.
- Il n' y a pas d'affichage dans le cas où le caractère est "." .

Analyse du problème

1) matrice pur ou structure ?

La première question à se poser est "comment stocker efficacement les valeurs du jeu". Une matrice semble à première vue la meilleure solution. Mais n'utiliser qu'une matrice a pour effets de rendre le code moins lisible. (nous ferons un parallèle matrice/structure au moment problématique). C'est pour cela que nous avons utilisons deux structures représentant notre jeu. Une structure qui représentera une colonne, et une autre qui contiendra une liste de ces colonnes. Au final ceci revient à avoir une matrice "cacher" dans les structures. C'est pour cela que dans les prochaines parties nous réfléchiront comme si nous traitions une matrice.

Nous allons maintenant détailler les données présentes dans notre pseudo matrice. Une bille à cinq états possible, donc de même pour les cases de notre matrice :

- La bille est rouge
- La bille est bleu
- La bille est verte
- La bille vient d'être sélectionnée suite à un coups
- La case est vide

Nous allons donc utiliser une matrice de caractère. Pour stocker dans la matrice les états des billes ("R","V","B","X",".").

2) Sélectionner les billes connexes

Maintenant que nous savons comment ranger les données relatives au jeu, il nous faut les traiter. Le premier traitement principal est la sélection des billes connexes à la bille sélectionnée par l'utilisateur. Pour cela, puisque nous utilisons un semblant de matrice. Il nous suffit de regarder la couleur de la bille à côté de la bille sélectionnée. En d'autres mots, de regarder si en haut, en bas, à gauche ou à droite de cette dernière il y a une bille de la même couleur. En effectuant cette action nous avons trois résultats possibles :

- La bille est de même couleurs. Nous rappelons notre algorithme de façon récursive avec les indices de la bille trouvée en paramètre.
- La bille n'est pas de même la même couleur. Rien ne se passe, l'algorithme continue.
- Les indices à tester sont hors de la matrice. Rien ne se passe, l'algorithme continue.

Il nous reste trois problèmes à traiter. Tout d'abord la suppression des billes sélectionnées . Pour cela il suffit de stocker la valeur originelle de notre bille, puis de la mettre à vide avant de lancer les possibles récursions. Ensuite le second problème est la fin de la récursion. Ceci est un faux problème, une bille ne peut être sectionnée deux fois par l'algorithme puisque son état change lors de la première sélection (passe à l'état vide être "."). Enfin, connaître le nombre de billes sélectionnées au total pour le calcul des points. Puisque le nombre d'appels de notre algorithme indique également le nombre de billes sélectionnées. Il nous suffit d'incrémenter une variable à chaque récursion.

Nous profitons aussi de cette fonction pour déterminer les indices "limGauche" et "Lim Droite". Ce dernier corresponde à la colonne la plus à gauche/droite ou au moins une bille a été supprimée. Ces variables nous serons utiles pour les fonctions écoulement et décalage.

3) affichage

Nous allons maintenant nous attarder sur notre affichage car celui-ci à une fonction double.

Tout d'abord il permet d'afficher le jeu et l'état de chacune des cases. Pour cela il nous suffit d'afficher une couleur spécifique selon l'état de la case (voir le cahier des charges).

La seconde fonction et de remplacer l'état "les billes viennent d'être sélectionnées à la suite d'un coup " (="X), par l'état "case vide"(="."). L'état "X" n'existe que pour un seul but, afficher les billes sélectionnées juste après l'algorithme de sélection des billes connexes. C'est pour cela qu'après l'affichage de cet état, nous donnons l'état " case vide" à la bille.

4) écoulement des billes

Après avoir "enlevé" les billes grâce à notre affichage. Il nous faut faire tomber les billes en lévitation .

La solution la plus triviale est de déplacer toutes les cases avec un état "vide" une par une vers le haut de leurs colonnes respectives . Mais cette méthode est loin d'être optimal car elle ne tient pas comptes des autres cases vide de la colonne. Voici les idées d'amélioration de cette solution pour la rendre optimale.

Premièrement nous utilisons une variable "limDroite" et "limGauche" qui indique les colonnes affecter par la variable rCouleurCon. de "limGauche" et "limDroite" sont les indices de la colonne la plus à gauche/droite ou au moins une bille a été retirée.

Deuxièmement nous utilisons variable dans notre structure colonne qui indique le nombre de billes encore en jeu dans la colonne, nommons la "indlimvide". Cette variable permet de ne pas parcourir entièrement les colonnes à chaque coup. C'est cette variable qui nous poussait à utiliser des structures. En effets pour gérer ce problème sans structure nous aurions eu besoin d'un vecteur en plus. Ce dernier aurait servi pour stocker les valeurs des variables pour les colonnes de notre pseudo matrice. Cette variable est bien sur mise à jour pour chaque colonne à chaque utilisation de l'algorithme écoulement .

Dernièrement nous effectuons les permutations entre les cases vides et les billes de façon optimale. L'idée est d'avoir une variable stockant l'indice vers où déplacer une bille. Si cette variable est égale à l'emplacement actuel de notre bille, alors ça ne sert à rien de déplacer notre bille. L'astuce est d'incrémenter cette variable uniquement quand l'élément est une bille, uniquement quand l'élément est différent de ".".C'est aussi grâce à cette variable que nous mettons à jour "indLimVide". Voici un exemple pour illustrer le fonctionnement de cette variable :

1	2	3	4	5	6
B	B	B	B	B	vide
vide	vide	vide	vide	vide	vide
V	V	V	vide	vide	B
vide	vide	vide	V	V	V
R	R	R	R	R	R

- 1) Il s'agit de l'état de base d'une colonne avant l'utilisation de l'algorithme. Nous allons, pour les points suivants utiliser deux variables. "i" un entier qui désigne l'indice de l'élément que nous traitons actuellement (le parcours se fait du bas vers le

haut). Et une variable "indEchange " qui indique vers où déplacer une case qui est non vide. "indEchange "= 1 et "i" = 1 avant le début de la récursion .

- 2) la case 1 est une bille. Aucun échange n'est nécessaire car i = "indEchange ". Il nous reste plus qu'à incrémenter de un "indEchange ", "indEchange " = 2.
- 3) la case 2 est vide. Nous ne faisons rien. "i" = 2 , "indEchange" = 2.
- 4) la case 3 est une bille et i ≠ indEchange. Nous échangeons la case 2(="indEchange") et 3(="i") et incrémentons "indEchange" : "indEchange" = 3
- 5) la case 4 est vide. Nous ne faisons rien. "i" = 4 , "indEchange" = 3
- 6) la case 5 est une bille et i ≠ indEchange. Nous échangeons la case 3(="indEchange") et 5(="i") et incrémentons "indEchange" : "indEchange" = 4
- Nous avons fini notre parcours. "indLimVide" = "indEchange"

Mais nous pouvons faire plus optimale . Nous ne sommes pas obligés d'intervertir les valeurs. Il suffit d'écraser les valeurs d'arriver et à la fin de rectifier les cases qui doivent être vide. Voici un exemple de cette amélioration :

1	2	3	4	5	6	7	8
B	B	B	B	B	B	B	vide
vide	vide	vide	vide	vide	vide	vide	vide
V	V	V	V	V	B	B	B
vide	vide	vide	V	V	V	V	V
R	R	R	R	R	R	R	R

- 1) Il s'agit de l'état de base d'une colonne avant l'utilisation de l'algorithme. Nous allons, pour les points suivants utiliser deux variables. "i" un entier qui désigne l'indice de l'élément que nous traitons actuellement (le parcours se fait du bas vers le haut). Et une variable "indEchange " qui indique vers où déplacer une case qui est non vide. "indEchange "= 1 avant le début de la récursion et "i" = 1 .
- 2) la case 1 est une bille. Aucun échange n'est nécessaire car i = "indEchange ". Il nous reste plus qu'à incrémenter de un "indEchange ", "indEchange " = 2.
- 3) la case 2 est vide. Nous ne faisons rien. "i" = 2 , "indEchange" = 2.

- 4) la case 3 est une bille et i ≠ indEchange.. La case 2(="indEchange") = la case 3(="i") et nous incrémentons "indEchange" : "indEchange" = 3
- 5) la case 4 est vide. Nous ne faisons rien. "i" = 4 , "indEchange" = 3
- 6) la case 5 est une bille et i ≠ indEchange. La case 3(="indEchange") = 5(="i") et nous incrémentons "indEchange" : "indEchange" = 4
- Nous avons finit notre parcours , "indLimVide" = "indEchange"
- 7 et 8) pour j de indEchange à finDuTableau(5) la case j = "vide"

5) Décalage

Notre but est de déplacer les colonnes entièrement vide du plateau sur la droite. En d'autres mots de poussées les billes vers la gauche pour combler les espaces vides. Notre méthode de résolution est proche de celle d'écoulement.

Premièrement nous utilisons une variable "indDecalage" contenant le nombre de colonnes encore en jeu. C'est à dire non vide. Nous utilisons aussi la variable "limGauche" . Celle-ci correspond à l'indice de la 1er colonne à gauche ou au moins une bille a été retirée lors de ce tour de jeu . Nous pratiquons donc le décalage sur les colonnes entre "limGauche et "indDecalage" première colonnes.

Deuxièmement de la même façon que "indEchange " dans l'algorithme écoulement nous échangeons les colonnes de place. C'est aussi grâce a cette variable indice échange que nous mettons a jour la valeur de "indDecalage"s

6) fonction principal

Maintenant que nous avons détaillé toutes les fonctionnalités, il nous faut les utiliser ensemble dans un même algorithme. Cet algorithme aura aussi pour but le calculent des points, la saisie des données par l'utilisateur et de tester si le jeu est fini . Nous allons détailler ses actions :

- Regarder si le jeu est fini, pour cela nous avons deux possibilités, regarder si la première case de la première colonne est vide ou si "indDecalage" est égale à 0 . Nous avons choisi la seconde option arbitrairement.
- afficher le jeu et les points du joueur

- lire les données de l'utilisateur
- sélectionner les billes connexes
- calculer les points
- afficher
- faire l'écoulement
- afficher
- faire le décalage
- revenir à l'étape 1

Quand le jeu est fini il nous faut afficher les points du joueur

Description des structures

Dans notre programme nous avons utilisé deux structures, PJeu et colonne

1) Structure colonne

Cette structure a pour objectif de stocker les billes d'une colonne ainsi que le nombre de billes encore en jeu dans cette colonne (= case total - case vide). Cette seconde variable appelée "indLimVide" nous permet d'optimiser notre code dans l'algorithme écoulement. Si nous aurions utilisé une simple matrice, il nous aurait fallu un vecteur à part pour pouvoir arriver à la même finalité. C'est-à-dire avoir un algorithme écoulement aussi optimal sans structure. En utilisant des structures notre code est plus lisible c'est pour cela que nous en avons utilisées.

```
typedef colonne: structure
          eleLig[NLB] Vecteur de caractère de [1..NLB]
{NLB est une constante représentant le nombre de lignes. Cette
variable contient les états des billes d'une colonne}

          indLimVide Entier ;
{nombre de case "vide" = "." dans le vecteur eleLig}
```

2) Structure Pjeu

Cette structure a quatre rôles à jouer. Premièrement stocker le plateau de jeu. C'est-à-dire un vecteur de colonnes. Deuxièmement stocker la variable "indDecalage" qui indique le nombre de colonnes non vides . C'est-à-dire le nombre de colonnes contenant encore des billes .Troisièmement la variables "points" qui est le nombre de points du joueur. Dernièrement garder en mémoire "le secteur de suppression de bille" via limGauche et limDroite

Nous aurions pu nous passer de cette structure en les définissant dans l'algorithme Jeu. Mais encore une fois, il est plus lisible d'utiliser une structure pour regrouper ses cinq variables. De plus ceci permet de réduire les paramètres donnés aux algorithmes.

```
typedef PJeu: structure
          col[NBC] Vecteur de colonne de [1..NBC]
{NBC est une constante représentant le nombre de colonnes.
Cette variable contient les colonnes du jeu}

          indDecalage Entier
{nombre de case colonnes contenant encore des billes dans le
vecteur col }

          points Entier
{nombre points de l'utilisateur }

          limGauche Entier
{Lorsque nous utilisons la fonction rCouleurCon. Nous gardons
en mémoire l'indice de la colonne là plus au gauche affecté par
la suppression d'une bille. Cette variable est utilisée dans la
fonction écoulement et décalage pour un parcours plus optimal }

          limDroite Entier
{Lorsque nous utilisons la fonction rCouleurCon. Nous gardons
en mémoire l'indice de la colonne là plus au droite affecté par
la suppression d'une bille.Cette variable est utilisée dans la
fonction écoulement pour un parcours plus optimal}
```

Algorithmes

1) Remplissage aléatoire de la matrice jeu

L'objectif de cet algorithme est de remplir aléatoirement notre plateau de jeu.

```
Action remplissageAlea(NBL,NBC,pjeu) :

D/R: PJeu pjeu {plateau de jeu à remplir}

D: NBL , NBC Entier {nombre de colonnes et de lignes du jeu}

VL: i , j Entier {indices de parcours}

     pour i de 1 à NBC faire :

          pour j de 1 à NBL faire :
               #1
                pjeu.col[i].eleLig[j] ← aléatoire('R','B','V')

          FinPour
          #2
          pjeu.col[i].indLimVide ← NBL
     finPour

     #3
     *pjeu.indDecalage ← NBC

finAction
```

#1 nous initialisons aléatoirement le vecteur "deeleLig" de chacune des colonnes de "col". C'est deux vecteurs correspond à notre plateau de jeu.

#2 nous initialions "indLimVide" qui est le nombre de billes dans la colonne à son maximum.

#3 nous initialions "indDecalage" qui est le nombre de colonnes non vides à son maximum.

2) Remplissage avec SG_math.c de la matrice jeu

Le but de cet algorithme est d'initialiser notre jeu avec les valeurs générées par SG_math.c .

SG_math.c contient une fonction f2. Cette fonction nous renvoie un réel que l'on doit interpréter de la façon suivante :

- si la valeur renvoyée est strictement inférieure à 0 la couleur de la bille sera rouge.
- si la valeur renvoyée est strictement inférieure à 0.5 la couleur de la bille sera verte.
- dans tous les autres cas la couleur de la bille sera bleue.

```
Action remplissageSG_mat(NBL,NBC,pjeu) :

D/R: PJeu pjeu {plateau de jeu à remplir}

D: NBL , NBC Entier {nombre de colonnes et de lignes du jeu}

VL:   - i , j Entier {indices de parcours}
      - z réel {variable tampon contenant la valeur donnée par
la fonction f2 de SG_math.c}

     pour i de 1 à NBC faire :
          pour j de 1 à NBL faire :

               z ← f2(i,j)
               #1
               Si z < 0 faire
                    pjeu.col[i].eleLig[j] ← 'R';
               Sinon
                    Si z < 0.5 faire
                    pjeu.col[i].eleLig[j] ← 'V';
                    Sinon
                    pjeu.col[i].eleLig[j] ← 'B';
                    Fin si
               Fin si
          FinPour
          #2
          pjeu.col[i].indLimVide ← NBL
     finPour

     #3
     pjeu.indDecalage ← NBC

finAction
Faction
```

#1 nous initialisons en fonction des résultats de SG_math.c le vecteur "deeleLig" de chacune des colonnes de "col". C’est deux vecteurs correspond à notre plateau de jeu.

#2 nous initialions "indLimVide" qui est le nombre de billes dans la colonne à son maximum.

#3 nous initialions "indDecalage" qui est le nombre de colonnes non vides à son maximum.

3) Algorithme d'écoulement

Voici l'algorithme d'écoulement de notre jeu :

```
Action ecoulement(NBL,NBC,pjeu) :

D/R: PJeu pjeu {plateau de jeu}

D: NBL , NBC Entier {nombre de lignes  et de colonnes du jeu}

VL:   - ligne , colonne , k Entier {indices de parcours}
      - indEchange Entier {indice vers où nous devons déplacer
la bille. Cette variable sert aussi à mettre à jour  la
variable indLimVide dans pour chaque colonne de pjeu}


                          #1
     pour colonne de pjeu.limGauche à pjeu.limDroite faire :
          indEchange ← 1
                               #2
          pour ligne de 1 à pjeu.col[colonne].indLimVide
faire :
               #3
               si  pjeu.col[colonne].eleLig[ligne] ≠ '.'
faire :
                    #4
                    si  indEchange ≠  ligne faire
                         pjeu.col[colonne].eleLig[indEchange]
                         ← peu.col[colonne].eleLig[ligne]

                    Fin si
                    indEchange ←  indEchange +1
               Fin si
          Fin pour

     #5
          pour k de indEchange à pjeu.col[colonne].indLimVide
faire :

          pjeu.col[colonne].eleLig[k] ← '.'


          Fin pour
          #6
          pjeu.col[colonne].indLimVide ← indEchange

     Fin pour
Faction
```

.#1 Nous parcourons seulement la zone où des billes ont été enlevées

#2 pjeu.col[colonne].indLimVide est le nombre de billes dans la colonne. Si pjeu.col[colonne].indLimVide = 0 alors la colonne est vide. Nous parcourons cette zone pour effectuer l'écoulement, car nous savons que les cases au-dessus sont vides . Et donc qu'il n'y a aucune bille à faire tomber.

#3 et #4 Si la case courante est vide nous ne faisons rien, grâce à ce stratagème "indEchange" ne s'incrémentera pas. Lorsque "indEchange" et différent de l'indice de boucle "ligne." Cela signifit que la bille à la case "ligne." dois aller à la case "indEchange". ("indEchange" – "ligne." Corresponds au nombre de case vide entre la case 1 et j).

#5 puisque nous n'échangeons pas les valeurs mais les écrasons au fur et à mesure pour écouler les billes. Nous devons parcourir les cases qui doivent être vide pour changer leur état. Ces cases sont entre "indEchange" et " pjeu.col[colonne].indLimVide". L'écart entre ces deux valeurs correspond au nombre de nouvelle case vide dans la colonne.

#6 "indEchange" devient la nouvelle limite entre les cases vides et les billes dans la colonne "pjeu.col[colonne]"

4) Algorithme de décalage

Voici l'algorithme de décalage de notre jeu :

```
Action décalage(NBL,NBC,pjeu) :

D/R: PJeu pjeu {plateau de jeu}

D: NBL , NBC Entier {nombre de lignes  et de colonnes du jeu}

VL:   - i , j  Entier {indices de parcours}
      - indEchange Entier {indice vers où nous devons déplacer
la colonne. Cette variable sert aussi à mettre à jour  la
variable indDecalage dans pjeu}

     indEchange ← pjeu.limGauche
                        #1
     pour i de pjeu.limGauche à pjeu.indDecalage faire :


          #2
          si  pjeu.col[i].indLimVide  ≠ 0 faire :
               #3
               si  indEchange ≠  i faire
                    #4
                    pour j de 1 à NBL faire :
                         pjeu.col[indEchange].eleLig[j] ←
                         pjeu.col[i].eleLig[j]
                    Fin pour
                    #5
                    pjeu.col[indEchange].indLimVide ←
                    pjeu.col[i].indLimVide

               Fin si
                    indEchange ← indEchange + 1
          Fin si
     Fin pour

     #6
          pour i de indEchange à pjeu.indDecalage faire :
               #7
               pour j de 1 à pjeu.col[i].indLimVide faire :
                    pjeu.col[i].eleLig[j] ← '.'
               Fin pour
          Fin pour
          #8
          pjeu.indDecalage ← indEchange
     Fin pour
Faction
```

.#1 pjeu.indDecalage correspond nombre de colonnes non vides. Nous parcourons donc toutes les colonnes qui sont non vides. Nous commençons le parcours à partir la colonne modifier la plus à gauche .

#2 et #3 pjeu.col[i].indLimVide est le nombre de billes dans la colonne "col[i]". Si pjeu.col[i].indLimVide = 0 alors la colonne est vide. Si la colonne est vide alors nous ne faisons rien, grâce à ce stratagème "indEchange" ne s'incrémentera pas. Lorsque "indEchange" et différent de l'indice de boucle "i." Cela signifie que la colonne à la case "i." dois aller à la case "indEchange". ("indEchange" – "i." Corresponds au nombre de colonne vide entre la case 1 et j).

#4 et #5 Nous déplaçons la colonne de la case pjeu.col[i] vers la case pjeu.col[indEchange]. Pour cela nous devons deplacer tous les éléments du tableau ainsi que la variable "indLimVide".

#6 puisque nous n'échangeons pas les colonnes mais les écrasons au fur et à mesure. Nous devons parcourir les colonnes qui doivent être vide pour changer leurs variables. Ces colonnes sont entre "indEchange" et " pjeu.indDecalage ". L'écart entre ces deux valeurs correspond au nombre de nouvelles colonnes vides.

#7 nous parcourons les colonnes a vidé jusqu'à "pjeu.col[i].indLimVide". Les cases aprés la variable "pjeu.col[i].indLimVide" sont forcément vides. "pjeu.col[i].indLimVide" est la limite entre les billes et les cases vides d'une colonne.

#8 "indEchange" devient la nouvelle limite entre les colonnes vides et colonnes non vides

5) Affichage de la matrice jeu

L'action affichage prend comme paramètre une structure, pour la résolution nous commençons à définir deux indices de parcours i (correspondant à l'indice de parcours des lignes de la matrice) et j(indice de parcours de la colonne), nous effectuons d'abord deux parcours séquentiels pour i de NBL-1 à 0 pas de -1 on affiche i puis pour j de 0 à NBC (nombre de colonnes de la matrice) on affiche le caractère correspondant à la case d'indices i et j de la matrice. Pour afficher la partie en dessous de la matrice on calcule tout d'abord le nombre de traits '_' nécessaires puis on fait un parcours séquentiel pour chaque indice i de 0 au nombre de traits on affiche '_' ,pour l'affichage des indices de colonnes,on affiche d'abord '|' puis on affiche l'indice i pour chaque i de 0 à NBL (nombre de lignes de la matrice) ensuite on affiche l'indice i pour chaque i de 0 à NBC.

```
Action affichage(NBL , NBC , pjeu):

D/R: pjeu PJEu

D: NBL , NBC Entier {nombre de lignes  et de colonnes du jeu}

V\L:
     -i,j entier{indices de parcours des lignes de
la matrice}

     -nbTrait  entier {nombres de "_" à afficher }

     Pour i de NBL à 1 par pas de --1 faire :

          afficher(i)
          afficher("| ")
          Pour j de 0 à NBC faire
               switchCouleur(NBL , NBC , pjeu,i,j)
          fpour
          afficher("\n")
     Fpour

     nbTrait= 3*NBC +3
     Pour i de 0 à nbTrait faire :
          afficher('_')
     Fpour

     afficher( "\n | " )
     Pour i de 0 à 10 faire

          afficher(i)
          afficher("  ")
     Fpour

     Pour i de 10 à NBC faire

          afficher(i)
          afficher(" ")
     Fpour


Fin Action
```

6) Sous-programme 'rCouleurCon'

L'action rCouleurCon prend en paramètres une structure 'pjeu',deux entiers 'col','lig' représentant respectivement le numéro de colonne et de la ligne de la bille sélectionnée par le joueur et une variable de type Donnée\Résultat correspond au nombre de billes connexes et de même couleur que la bille sélectionnée. Pour la résolution ,nous avons commencé à définir une variable 'couleur' pour stocker la couleur de la bille sélectionnée puis on vérifie si 'limDroite' est plus petite que 'col' si c'est le cas ,on affecte à 'limDroite' la valeur 'col', si

'limGauche' est plus grand que 'col' on affecte à 'limGauche' la valeur 'col'. Ensuite on remplace le caractère de la bille sélectionnée par 'x' . Si le numéro de la ligne 'lig' de la bille est plus grand que zéro et que la couleur de la case en dessous de la bille est la même que 'couleur' dans ce cas on appelle la fonction rCouleurCon en diminuant le troisième paramètre 'lig' de -1. Si le numéro de la ligne 'lig' est inférieur au nombre de lignes de la matrice et la couleur de la case au dessus de la bille sélectionnée est la même que 'couleur' dans ce cas on appelle la fonction rCouleuCon en incrémentant le troisième paramètre 'lig' de 1. Si le numéro de la colonne 'col' est inférieur au nombre de colonnes de la matrice et la couleur de la case à

droite de la bille sélectionnée est la même que 'couleur',on appelle à la fonction rCouleurCon en incrémentant le deuxième paramètre 'col' de 1. Si le numéro de la colonne 'col' est plus grand que zéro et la couleur de la case à gauche de la bille sélectionnée a la même couleur que 'couleur' ,on appelle à la fonction rCouleurCon en diminuant le deuxième paramètre 'col' de -1.

```
Action rCouleurCon(NBL , NBC , pjeu ,col,lig,nbCouleurCon):

D/R:
     -pjeu Pjeu
     -nbCouleurCon,entier {nombre de billes supprimées}

D:
     -NBL , NBC Entier {nombre de lignes  et de colonnes du
jeu}
     -col,lig {indice de la bille sélectionnée par
l'utilisateur }

V\L: couleur caractère

     nbCouleurCon ← nbCouleurCon + 1
     Si pjeu.limDroite < col alors
          pjeu.limDroite ← col
     FinSi
     Si pjeu.limGauche > col alors
          pjeu.limGauche ← col
     FinSi

     couleur=pjeu.col[col].eleLig[lig]

     Si lig > 1 et pjeu.col[col].eleLig[lig--1] = couleur
alors :
          rCouleurCon(pjeu,col,lig --1,nbCouleurCon )
     FinSi

     Si lig < NBL et pjeu.col[col].eleLig[lig+1] =
couleur alors :
          rCouleurCon(pjeu,col,lig+1,nbCouleurCon )
     FinSi

     Si col < NBC et pjeu.col[col+1].eleLig[lig]
=couleur) alors :
          rCouleurCon(pjeu,col+1,lig,nbCouleurCon )
     FinSi

     Si col > 1 et pjeu.col[col--1 ].eleLig[lig] = couleur
alors :
     CouleurCon(pjeu,col-1,lig,nbCouleurCon )
     FinSi
```

7) Sous-programme 'switchCouleur'

L'action 'switchCouleur' prend en paramètres une structure 'pjeu', deux entiers donnés 'lig' et 'col' correspondant respectivement au numéro de ligne et de colonne d'une bille.Elle permet de remplacer tous les caractères par des couleurs. Si 'lig' et 'col' donnés correspond à une bille de caractère 'B' ,alors on affiche des espaces sous fond bleu. Si 'lig' et 'col' correspond à une bille de caractère 'V' ,alors on affiche des espaces sous fond vert. Si 'lig' et 'col' correspond à une bille de caractère 'R' ,alors on affiche des espaces sous fond rouge. Si 'lig' et 'col' correspond à une bille de caractère 'x' ,alors on affiche des espaces sous fond jaune. Sinon il n'y a pas d'affichage.

```
Action swicthCouleur(NBL , NBC , pjeu ,col , lig):

D/R:
     -pjeu Pjeu

D:
     -NBL , NBC Entier {nombre de lignes  et de colonnes du
jeu}
     -col,lig {indice de la bille à afficher }


     Si pjeu.col[col].eleLig[lig] = "R" alors :
          afficher("   "){trois espaces de couleur rouge }
     Sinon
          Si pjeu.col[col].eleLig[lig] = "V" alors :
               afficher("   "){trois espaces de couleur verte }
          Sinon
               Si pjeu.col[col].eleLig[lig] = "B" alors :
                    afficher("   "){trois espaces de couleur
bleu }
               Sinon
                    Si jeu.col[col].eleLig[lig] = "X" alors :
                         afficher("   "){trois espaces de
couleur jaune }
                         jeu.col[col].eleLig[lig] ← "."
                    Sinon
                         afficher("   ")
                    FSI
               FSI
          FSI
     FSI
FA
```

8) Sous-programme 'jeu

Le sous-programme 'jeu' permet de jouer le jeu Same Gnome.Pour la résolution ,on commence par définir une structure 'pjeu',puis à remplir la matrice de jeu. On initialise à zéro le nombre de points marqués par le joueur,on définit ensuite trois variables correspondantes respectivement à la colonne saisie par le joueur,à la ligne saisie par le joueur et le nombre de billes enlevés(billes connexes) dans la matrice. Tant qu'il reste des billes dans la matrice jeu ,on affiche le nombre de points marqués par le joueur ,on affiche l'état de la matrice et le jeu continue par la saisie d'une nouvelle colonne et d'une nouvelle ligne par le joueur et si la case correspondante n'est pas vide ,la variable correspondante au nombre de billes est initialisée à zéro ,nous recherchons ensuite les billes connexes de même couleurs que la bille sélectionnée puis on calcule le nombre de points marqués et on affiche la nouvelle matrice obtenue après l'enlèvement des billes connexes,on appelle ensuite les fonctions permettant l'écroulement des billes et le décalage des colonnes. Dans le cas où la colonne et la ligne saisies par le joueur correspondent à une case vide ,un message d'erreur s'affiche le joueur doit saisir à nouveau une nouvelle colonne et une nouvelle ligne. On affichera au final le nombre de points marqués par le joueur .'

```
Action jeu():

D/R:
     -pjeu Pjeu

VL:
     - pjeu Pjeu
     - points,col,lig,nbBilleConnexes entier

     {ou autre méthode de remplissage}
     remplissageAlea(NBL,NBC,pjeu)
     points=0
     Tant que pjeu.inDecalage ≠ 0 faire :


          afficher('Vous avez ' points 'points\n')
          affichage(NBL , NBC , pjeu)
          afficher('Choisir une colonne: ')
          lire(col)
          afficher('Choisir une ligne: ')
          lire(ligne)
          si pjeu.col[colonne].eleLig[ligne]!= '.' alors :

               nbBilleConnexes=0
               pjeu.limDroite=colonne
               pjeu.limGauche=colonne


               rCouleurCon(NBL , NBC , pjeu
,col,lig,nbCouleurCon

               points ← (nbBilleConnexes1)* (nbBilleConnexes-1)
+ points

               affichage(NBL , NBC , pjeu)

               ecoulement(NBL,NBC,pjeu)
               affichage(NBL , NBC , pjeu)
               décalage(NBL,NBC,pjeu)
          sinon
               afficher('choix invalide ')

          Fin Si
     FinTantQue
     afficher('Vous avez fini le jeu avec ')
     afficher(points)
     afficher(' points')
Fin Action
```

Tests effectués

Nous avons testé chacune des fonctions qui ont été programmées tout au long de ce projet.

D'abord nous avons testé le programme 'remplissageAlea' qui remplit une matrice aléatoirement avec les caractères ‘R’,’V’ et ‘B’ .Puis nous avons obtenu une matrice de couleurs rouge,vert et bleu,dans ce cas les programmes ‘affichage’ et ‘switchCouleur’ se sont bien déroulés .

Avec cette même matrice déjà remplie avec des caractères ‘R’,’V’ et ‘B’ nous avons testé le programme ‘rCouleurCon’ en choisissant une colonne et une ligne correspondant à la bille qu’on veut sélectionner. Il s’affiche une nouvelle matrice dont toutes les cases connexes et de même couleur que la bille sélectionnée sont remplacées par le caractère ‘x’ (couleur jaune).Donc le programme ‘rCouleurCon’ s’est bien déroulé.

À l'aide de cette matrice obtenue, nous avons également testé le programme ‘ecoulement’ .On a obtenu une nouvelle matrice dont toutes les billes qui étaient au dessus de la bille sélectionnée par le joueur tombent dans les espaces vides donc le programme s’est bien déroulé.

Nous avons testé le programme ‘remplissage’ qui remplit manuellement une matrice de jeu. Tant que la matrice n’est pas remplie totalement le programme demande à l’utilisateur de saisir un caractère.

Nous avons testé le programme ‘decalage’ en utilisant le plateau initial de la version 7(SG_7.c).Dans un premier temps nous avons choisi la colonne 0 et la ligne 0,puis les programmes ‘rCouleurCon’ et‘ecoulement’ se sont bien déroulés et après nous avons obtenue à l’aide du programme ‘decalage’ une nouvelle matrice dont les colonnes 11,12,13 et 14 sont entièrement libérées.

Nous avons également fait le test du programme ‘rCouleurCon’ en utilisant le plateau initial donné à la version 3 du sujet (SG_3.c) où toutes les cases du plateau contient que des billes de caractères ‘V’ , on a testé qu’on obtient bien une matrice remplie qu’avec des caractères ‘x’(couleur jaune)et que le nombre affiché correspond bien à 150(nombre total de cases dans la matrice) .Cet programme est également testé en utilisant le plateau initial de la

version 6(SG_6.c) et 8(SG_8.c) du sujet . On a sélectionné une bille de caractère ‘B’ dans chaque version et obtient dans chaque cas une matrice dont tous les caractères ‘B’ ont été remplacés par ‘x’ (couleur jaune).

Les programmés ‘ecoulement’ et ‘decalage’ sont aussi testés en utilisant le plateau initial de la version 4 (SG_4.c) où il n’ y a qu’une seule bille de caractères ‘B’ et toutes les billes restantes sont de caractères ‘V’.On a sélectionné une bille de caractère ‘V’ , et toutes les billes de caractères 'V' se sont remplacés par 'x'(à l'aide du programme 'rCouleurCon') et avec les programmes ‘ecoulement’ et ‘decalage’ ,on obtient bien une matrice contenant qu’une seule bille de caractère ‘B’ se retrouvant à la colonne 0 et à la ligne 0 de la matrice.

On a également testé le programme ‘decalage’ en utilisant le plateau initial donné à la version 7 (SG_7.c), on a sélectionné une bille ‘B’ et après le déroulement des programmes 'rCouleurCon' et 'decalage' puis 'décalage' on a obtenu une matrice dont la dernière colonne est entièrement libérée donc le programme 'decalage' s’est bien déroulé.

Le programme ‘jeu’ regroupe toutes les fonctions qui ont été programmées et à chaque coup de jeu on affiche l’état de la matrice et le nombre de points marqués par le joueur à la fin du jeu(Lorsqu’il n’y a plus de billes dans la matrice).

Avancement du projet et problèmes rencontrés

Nous nous sommes focalisé sur le pseudo-code avant de programmer véritablement Same gnome. Une fois le pseudo-code réalisé nous avons réparti les tâches de programmation et de rapport. Jordan c'est occupée de la partie analyse du problème, description des structures , programmation et l'écriture des pseudo-codes. Coumba c'est occupée s'occuper de la partie cahier des charges et des tests à effectuer.L'analyse des pseudo-codes a été faites à deux.

Nous avons au cours du projet décider d'utiliser des structures plutôt qu'une simple matrice pour simplifier notre code et stocker plus efficacement nos variables. Ceci a impliqué la modification de tous les algorithmes.

Conclusion

Au final, le jeu fonctionne bien et effectue les opérations demandées. Ce projet s'est révélé très enrichissant dans la mesure où il a consisté en une approche concrète du métier d'ingénieur. En effet, la prise d'initiative, le respect des consignes et le travail en binôme seront des aspects essentiels de notre futur métier. De plus, il nous a permis d'appliquer nos connaissances en algorithme et programmation à un domaine très pratique.
Il y avait également une cohésion avec mon binôme, personne n'est restée en retrait .
Ce projet nous a également permis d'avoir un aperçu de tous les problèmes qui peuvent se poser lors d'un projet de ce genre.

Programme en c

```
#include <stdio.h>

#include <stdlib.h>

#include <time.h>


#define NBL 10 //Nombre de lignes de la matrice

#define NBC 15 //Nombre de colonnes de la matrice

#define reset "\033[0m"

#define reverse "\033[7m"

#define rouge "\033[;31m"

#define vert "\033[;32m"

#define bleu "\033[;34m"

#define jaune "\033[;33m"



double f1 (int i, int j){

    return sin ((double)i*j + (double)j);

}


double f2 (int i, int j){

    double z = f1 (i, j);

    return z * z;

}
```

/**

*

typedef colonne: structure

eleLig[NLB] Vecteur de caractère de [1..NLB]

{NLB est une constante représentant le nombre de lignes. Cette variable contient les états des billes d'une colonne}

indLimVide Entier ;

{nombre de case "vide" = "." dans le vecteur eleLig}

*

*/

typedef struct Colonne{

char eleLig[NBL];

int indLimVide ;

}Colonne;

/**

*

typedef PJeu: structure

col[NBC] Vecteur de colonne de [1..NBC]

{NBC est une constante représentant le nombre de colonnes. Cette variable contient les colonnes du jeu}

indDecalage Entier

{nombre de case colonnes contenant encore des billes dans le vecteur col }

points Entier

{nombre points de l'utilisateur }

limGauche Entier

{Lorsque nous utilisons la fonction rCouleurCon. Nous gardons en mémoire l'indice de la colonne là plus au gauche affecté par la suppression d'une bille. Cette variable est utilisée dans la fonction écoulement et décalage pour un parcours plus optimal }

limDroite Entier

{Lorsque nous utilisons la fonction rCouleurCon. Nous gardons en mémoire l'indice de la colonne là plus au droite affecté par la suppression d'une bille.Cette variable est utilisée dans la fonction écoulement pour un parcours plus optimal}

*/

```
typedef struct PJeu{
   Colonne col[NBC];
   int indDecalage ;
   int points;
   int limGauche;
```

```
    int limDroite;
}PJeu;


/**
 * @but Permets d'afficher la couleur correspondant au caractère envoyé
 *
 * @Donnée lig,col {indice de la bille affichée actuellement  }
 *
 * @DonnéeRésultat pjeu
 *
 *
 */
void switchCouleur(PJeu *pjeu , int lig , int col ){
    switch ((*pjeu).col[col].eleLig[lig]){
        case 'R':
            printf("%s%s   %s",rouge,reverse,reset);
            break ;
        case 'B':
            printf("%s%s   %s",bleu,reverse,reset);
            break ;
        case 'V':
            printf("%s%s   %s",vert,reverse,reset);
            break ;
```

```
        case 'X':
            printf("%s%s   %s",jaune,reverse,reset);
            (*pjeu).col[col].eleLig[lig] = '.';
            break ;
        default:
            printf("   ");
            break ;
    }
}

/**
 * @but : Afficher une matrice  donnée
 *
 * @Donnée:  pjeu PJeu
 *
 * @Variable_local:
 *  -i,j Entier {indices de parcours }
 *  -nbTrait Entier {nombre de '_' à écrire pour rentrer l'affichage plus esthétique }
 */
void affichage(PJeu *pjeu){
   int i , j ;

   // affichage de la partie haute du jeu
```

```
for( i=NBL-1;i>=0;i--){
   //numéro des lignes
   printf("%d| ",i);
   for(j=0;j<NBC;j++){
      // affichage de la couleur ij de matJeu
      switchCouleur(pjeu ,i ,j );
   }
   printf("\n");
}
//affichage de la partie en dessous  du jeu

// nombre de '_' à afficher
int nbTrait = NBC + NBC + NBC + 3 ;
for(i=0;i<nbTrait;i++){
   printf("_");
}

//affichage des indices de colonnes
printf("\n | ");

for(i=0;i<10;i++){
   printf("%d  ",i);
}
```

```
    for(i=10;i<NBC;i++){
        printf("%d ",i);
    }
    printf("\n");

}

/**
 *
 * @but : avec une matrice de caractère donné, déplacer tous les caractères "." vers le haut
 *
 * @DonnéeRésultat pjeu PJeu
 *
 * @Variable_local:
 *  -colonne, ligne,k   Entier {indice de boucles dans matJet }
 *  -indEchange Entier
 *
 *
 *
 */
void ecoulement(PJeu *pjeu ){
    int indEchange   ;
    for (int colonne = pjeu->limGauche ; colonne <= pjeu->limDroite ; ++colonne) {
```

```
//for (int colonne = 0; colonne < (*pjeu).indDecalage ; ++colonne) {
    indEchange = 0 ;

     //on fais décendre toutes les cases de couleur du nombre de case vide en dessous d'eux
    for ( int ligne = 0; ligne < (*pjeu).col[colonne].indLimVide ; ++ligne) {
       if ((*pjeu).col[colonne].eleLig[ligne] != '.') {
          if (indEchange != ligne){
                                                    (*pjeu).col[colonne].eleLig[indEchange]  = (*pjeu).col[colonne].eleLig[ligne] ;

          }
          indEchange++;
       }
    }

    //les cases entre  indEchange et pjeu.col[colonne].indLimVide sont mis à vide
     //l'écart entre les deux valeurs correspond au nombre de case vide apparue lors de ce tour de jeu
    for (int k = indEchange ; k <(*pjeu).col[colonne].indLimVide ; ++k) {
       (*pjeu).col[colonne].eleLig[k] = '.' ;
    }

    //on met a jour la indLimVide dans notre colonne
```

```
        (*pjeu).col[colonne].indLimVide = indEchange ;
    }
}

/**
 * @but décaler les colonnes "vide" (remplie de '.') vers la droite du jeu
 *
 * @DonnéeRésultat: PJeu pjeu
 *
 *  @Variable_local
 *      -i,j, ii ,jjEntier { indice de boucle dans matJet }
 *      -indEchange Entier {indice d'échange entre les colonnes
 */
void decalage(PJeu *pjeu ){
    int indEchange = (*pjeu).limGauche ;
    int i , j ;

    for ( i = (*pjeu).limGauche; i < (*pjeu).indDecalage ; ++i) {
        //si indLimVide == 0  alors toute la colonne est constituée de '.'
        //il nous faut donc décaler la ligne non vide la plus proche sur cette dernière.
        if((*pjeu).col[i].indLimVide != 0){
            if (indEchange != i){
```

```
            //decalage de la Ligne
            for ( j = 0; j < NBL ; ++j) {
                (*pjeu).col[indEchange].eleLig[j] = (*pjeu).col[i].eleLig[j] ;
            }
            //on modifie indLimVide de la colonne d'arriver
            (*pjeu).col[indEchange].indLimVide = (*pjeu).col[i].indLimVide ;
        }
        indEchange++;

    }
}

//clean colonne
for ( i = indEchange;  i < (*pjeu).indDecalage ; ++i) {
    for ( j = 0 ; j <(*pjeu).col[i].indLimVide ; ++j) {
        (*pjeu).col[i].eleLig[j] = '.' ;
    }
}

//nouvelle indice limite entre les colonnes jouables et vides
(*pjeu).indDecalage = indEchange ;
```

```
}


/**
 * @but : initialiser le jeu aléatoirement
 *
 * @DonnéeRésultat PJeu pjeu
 *
 * @Variable_local i , j Entier { indice de boucle}
*/
void remplissageAlea(PJeu *pjeu){
   char chaine[3]="RVB";


   srand(time(NULL));
   for (int i=0;i< NBC;i++){
      for( int j=0;j<NBL;j++){
         (*pjeu).col[i].eleLig[j] = chaine[rand()%3]  ;
      }
      (*pjeu).col[i].indLimVide = NBL;


   }
   (*pjeu).indDecalage = NBC ;
}
```

```
/**
 * @but : initialiser le jeu
 *
 * @DonnéeRésultat: PJeu pjeu
 *
 * @Variable_local :i,j  Entier { indice de boucle}
 * @Variable_local: z  Réel{Carré de sin(i*j+j)}
*/
void remplissageSG_mat(PJeu *pjeu){
   int i,j;
   double z;
   for (i=0;i< NBC;i++){
      for( j=0;j<NBL;j++){
         z=f2(i,j);
         if(z<0) (*pjeu).col[i].eleLig[j]='R';
         else{
            if(z<0.5) (*pjeu).col[i].eleLig[j]='V';
            else (*pjeu).col[i].eleLig[j]='B';
         }
      }
      (*pjeu).col[i].indLimVide = NBL;

   }
```

```
    (*pjeu).indDecalage = NBC ;
}

/**
 *
 * @but : avec 2 indices col  et lig qui désignent l'emplacement d'une couleur dans pjeu :
 *  -Remplace toutes les cases connexes et  de même couleurs par une case vide (='.')
 *  -donne le nombre de case modifier avec la variable nbCouleurCon
 *
 * @DonnéeRésultat
 *      -PJeu *pjeu
 *      -nbCouleuron entier {Nombre total de billes connexes}
 *
 * @Donnée col,lig entier {indices de la bille que l'on doit "vider" et chercher les billes de sa couleur autour d'elle }
 *
 *  @Variable_local couleur caractère {variable permettant de stocker la couleur de la case sélectionnée }
 */
void rCouleurCon(PJeu *pjeu ,int col,int lig, int *nbCouleurCon){
    (*nbCouleurCon )++;
    char couleur = pjeu->col[col].eleLig[lig];
```

```
if  (pjeu->limDroite < col) {
    pjeu->limDroite = col ;
}

if  (pjeu->limGauche > col) {
    pjeu->limGauche = col ;
}
pjeu->col[col].eleLig[lig] = 'X';



//récursion bas
if(lig > 0 &&  pjeu->col[col].eleLig[lig-1] == couleur){
    rCouleurCon(pjeu,col,lig-1,nbCouleurCon );

}
//récursion haut
if(lig < NBL &&  pjeu->col[col].eleLig[lig+1] == couleur){
    rCouleurCon(pjeu,col,lig+1,nbCouleurCon );
}
//récursion droite
if(col < NBC && pjeu->col[col+1].eleLig[lig] == couleur){
    rCouleurCon(pjeu,col+1,lig,nbCouleurCon );
}
```

```
    //récursion gauche
    if(col > 0 &&  pjeu->col[col-1].eleLig[lig] == couleur){
        rCouleurCon(pjeu,col-1,lig,nbCouleurCon );
    }
}

/**
 *  @but Jouer au jeu SameGnome
 *
 *  @Variable_local
 *      -pjeu PJeu
 *      -points Entier {points du joueur }
 *      -colonne,ligne Entier {indice de la bille choisi par le joueur  }
 *      -nbBilleConnexes Entier {variable servant au calcul des points }
 */
void jeu() {
    PJeu pjeu ;

    //initialisation du jeu
    remplissageAlea(&pjeu);

    pjeu.points = 0 ;
    int colonne,ligne,nbBilleConnexes ;
```

```
//tant que le jeu n'est pas fini
while (pjeu.indDecalage != 0){
    printf("Vous avez %d points \n",pjeu.points );
    affichage(&pjeu);

    //Choix d'une colonne
    puts("choisir une colonne : ");
    scanf("%d",&colonne );

    //choix d'une ligne
    puts("choisir une ligne : ");
    scanf("%d",&ligne );

    //si la case selectionné est une case non vide
    if (pjeu.col[colonne].eleLig[ligne] != '.'){
        nbBilleConnexes = 0 ;
        pjeu.limDroite = colonne ;
        pjeu.limGauche = colonne ;
        //recherche des billes connexes de même couleur à la bille choisi par le joueur
        rCouleurCon(&pjeu,colonne,ligne,&nbBilleConnexes);
        //calcul des points
```

```
                pjeu.points  = (nbBilleConnexes-1)*(nbBilleConnexes-1) + pjeu.points  ;


                affichage(&pjeu);


                //écoulement des billes
                ecoulement(&pjeu);
                affichage(&pjeu);


                //décalage des billes
                decalage(&pjeu);




            } else{
                //message d'erreur lorsque la case selectionné est vide
                  printf("la case colonne : %d  ligne : %d  est une case vide ! choisissez d'autre indices \n",colonne,ligne );
            }


        }


        printf("vous avez fini le jeu avec %d points!! Bravo\n",pjeu.points  );


    }
```

```
//lancement du JEU SAME GNOME

int main(){


   jeu();

   return 0;


}
```