Jeu de la vie - corrigé

Guillaume Vernade - guillaume.vernade@ens.fr

Compter le nombre de cellules

Il s'agit d'un bête parcours de liste[1] dont on va faire la somme des éléments (car si on a une cellule on a un 1 et sinon un 0). Ainsi on commence par faire une fonction de somme sur une liste :

```
sommeliste := proc(liste,n)
  local somme,i;
  somme := 0;
  for i from 1 to n do
    somme := somme + liste[i];
  done;
  somme;
end;
```

On cré ensuite une fonction de somme sur une liste de liste :

```
sommelisteliste2liste := proc(liste2liste,m,n)
  local somme,i;
  somme := 0;
  for i from 1 to m do
    somme :=  somme + sommeliste(liste2liste[i],n);
  done;
  somme;
end;
```

On a plus qu'à créer la fonction demandée :

```
nb_vivantes := tableau -> sommeliste2liste(tableau,N,N);
```

On aurait aussi pu programmer directement la fonction $nb_vivantes$ mais faire plusieurs fonctions auxiliaires permet de faire du code plus lisible et d'avoir des fonctions réutilisables par la suite (*sommeliste* et *sommeliste2liste*).

```
nb_vivantes := proc(tableau)
  local somme,i,j;
  global N; # On utilise N en global pour pouvoir le changer une seule
            # fois et pas dans tous les programmes
  somme := 0;
  for i from 1 to N do
```

[1] d'une liste de liste en fait

```
      for j from 1 to N do
        if tableau[i][j]=1
          then somme := somme + 1; #On peut aussi ajouter tableau[i][j]
                                   #à somme dans tous les cas
fi;
      od;
    od;
    somme;
  end;
```

Position des cellules vivantes

L'algorithme est le même que dans le précédent, il s'agit de parcourir toutes les cases d'un tableau. La seule différence réside dans la forme du résultat. On veut une liste de coordonnées, les coordonnées sont sous la forme $[x, y]$, et on se rappelle que deux suites (des choses de la forme a, b, c) peuvent être concaténées (ie. collées) en les notant l'une à la suite de l'autre séparés par une virgule (si $L1{=}a, b, c$ et $L2{=}d, e, f$, $L1, L2{=}a, b, c, d, e, f$). Cela donne donc[2] :

```
  coordonnées := proc(tableau)
    local coord,i,j;
    global N;
    coord := op([]); #Une petite ruse étant donné qu'on ne peut
                     #pas noter de suite vide (on écris donc
                     #''La suite équivalente à la liste vide'')
    for i from 1 to N do
      for j from 1 to N do
        if tableau[i][j)=1
          then coord := coord,[i,j]; #Au lieu d'ajouter 1 on ajoute
                                     #un couple de coordonnées
fi;
      od;
    od;
    [coord]; #Et on transforme la suite en liste pour la renvoyer
  end;
```

Pour afficher une liste de points il suffit de donner à *plot* la liste des coordonnées et de lui dire $style = point$, ce qui donne :

```
  affiche := proc(tableau)
    global N;
    plot(coordonnées(tableau),x=1..N, y=1..N, axes=none, style=point);
  end;
```

Le changement de variables des coordonnées

On va maintenant s'attaquer au calcul des états successifs du tableau. Pour cela on aura besoin de calculer le nombre de voisins de chaque cellule. On risque alors d'avoir des problèmes

[2]En un seul algorithme pour mieux voir les points communs avec le précédent mais on peux encore le faire en plusieurs

au bord. Plutôt que de faire des cas particuliers dans l'algorithme qui comptera le nombre de voisins on faire faire un "changement de variable" qui se débrouillera juste pour que 0 soit vu comme un N et que N+1 soit vu comme un 1. En gros c'est un calcul de modulo :

```
f := proc(i)
  if i<=0
    then i+N;
    else if i>=N+1
      then i-N;
      else i;
    fi;
  fi;
end;
```

ou plus simplement (pour ceux qui ont fait une spé maths...) :

```
f := i -> (i-1 mod N)+1;
```

Calcul de l'évolution du système

Encore une fois il est plus simple de faire plusieurs algorithmes auxiliaires qu'un gros programme.

Calcul du nombre de voisins

```
nb_voisins := (tableau,i,j) ->
  tableau[f(i-1),f(j-1)] + tableau[f(i-1),f(j)] + tableau[f(i-1),f(j+1)] +
  tableau[f(i)  ,f(j-1)] +                      + tableau[f(i)  ,f(j-1)] +
  tableau[f(i+1),f(j-1)] + tableau[f(i+1),f(j)] + tableau[f(i+1),f(j+1)];
```

Mise en aplication des lois de vie ou de mort

```
vie_mort := proc(ancienne_valeur,voisins)
  if ancienne_valeur=0
    then
      if voisins=3
        then 1; #Une cellule nait
  else 0; #La cellule reste morte
fi;
      else
        if voisins=2 or voisins=3
  then 1; #La cellule survit
  else 0; #Le cellle meurt
fi;
    fi;
  end;
```

Calcul de l'état suivant

On va créer un tableau initialement vide qui représentera l'état suivant du système puis on va parcourir le tableau de l'état présent et ainsi remplir le nouveau tableau.

```
suivant := proc(tableau)
  local i,j,etat_suivant;
  global l0,N; # On va utiliser l0 pour initialiser le tableau vide
  etat_suivant:=l0;
  for i from 1 to N do
    for j from 1 to N do
      etat_suivant[i][j] := vie_mort(tableau[i][j],nb_voisins(tableau,i,j));
    od;
  od;
  etat_suivant;
end;
```

Jouer

Pour tester notre programme on commence par générer un tableau aléatoire de 0 et de 1 :

```
tableau := seq(seq(rand(2)(),i=1..N),j=1..N);
```

Comme tout à l'heure on affiche les cellules en tapant :

```
plot(coordonnées(tableau),style=point);
```

Puis on va calculer l'état suivant et l'afficher :

```
tableau := suivant(tableau);
plot(coordonnées(tableau),style=point);
```

Et on recommence...