UNIVERSITÉ DE ROUEN

UFR Sciences et Techniques

Rapport de Projet

Thème : solveur des grilles de Hashiwokakero

Réaliser par :

EL KOUAY El Mehdi
BERHAIL Khalid
BENDAOU NASSIM
AAZZA IDRISS

Licence 3 Informatique | 2020 - 2021

Plan :

- Présentation du jeu :
  - ✓ Grilles de Hashiwokakero
  - ✓ Hashi Puzzle Solver
- Choix lié à la programmation
- Difficulté rencontré
- Algorithmes et Structures :
  - ✓ Island
  - ✓ Connection
  - ✓ Algorithmes de résolution d’énigmes : Backtracking
  - ✓ Algorithmes de génération de puzzle
- Annexe

I. Présentation du jeu :

1. Grilles de Hashiwokakero

Le ***Hashiwokakero*** se joue sur une grille rectangulaire sans grandeur standard. On y retrouve des nombres de 1 à 8 inclusivement. Ils sont généralement encerclés et nommés *îles*. Le but du jeu est de relier toutes les îles en un seul groupe en créant une série de *ponts* (simples ou doubles) entre les îles.

- Tout pont débute et finit sur une île.
- Aucun pont ne peut en croiser un autre.
- Tous les ponts sont en ligne droite.
- Le nombre de ponts qui passent sur une île est le nombre indiqué sur l'île.
- Toutes les îles doivent être reliées entre elles.

(Extrait *Wikipédia*)

2. Hashi Puzzle Solver :

Hashi Puzzle Solver est le nom de notre jeu. Il s'agit d'un solveur de grilles de ***Hashiwokakero.***
Le solveur peut générer des grilles aléatoirement et qui ont toujours une solution, comme il peut laisser le choix à l'utilisateur pour créer sa propre grille. Après, notre solveur va tester si la grille créer admet une solution ou pas selon les règles définis au paragraphe précédent.

II. Choix lié à la programmation :

Nous avons choisi le langage JAVA pour coder ce jeu, premièrement parce qu'on est habitué à travailler avec ce langage, ainsi qu'il est basé sur la notion d'objet, chose qui nous facilite le travail. Et deuxièmement parce qu'il va nous faciliter le travail sur le côté graphique du jeu en utilisant la bibliothèque d'interfaces graphique JavaFX, et plus précisément avec l'outil Scène Builder.

III. Difficulté rencontré :

Pour mettre en place notre jeu, on a rencontré des difficulté lié à l'implémentation des algorithmes avec le langage java, pour résoudre cette difficulté on a adapté nos algorithmes et ajouté des bout de code qui vont faciliter le déroulement des fonction qui trouve la solutions. Une autre difficulté été lors de la transformation de la solution à un affichage graphique, et pour la résoudre on a utilisé l'objet "connection" qui est dans les cases vides du coup quand l'objet "connection" à comme valeur 0, on dessine rien mais quand il a une valeur on dessine une ligne spécifier (Horizontale, Verticale, et spécifier par une direction). Le dernier problème est lié au temps nécessaire pour résoudre l'énigme, surtout lors ce qu'on commence a dépassé 15 iles , pour cette tache on a pas encore trouvé une solution mais prochainement on peut améliorer notre algorithme pour qu'elle résolue les puzzle plus rapidement .

IV. Algorithme et structures :

Pour travailler sur notre jeu nous avons choisi d'utiliser les structures suivant :

Island :

Il représente les iles dans le jeu, ou les sommets s'ils ont considère comme un graphe.

Connection :

Il représente les ponts dans le jeu, ou les arrêts s'ils ont considère comme un graphe.

```
Type Island N = Enregistrement

                MaxDegree : Entier
                Degree : Entier
                NorthIsland : Island
                EasthIsland : Island
                SouthhIsland: Island
                WestIsland  : Island
                NorthConnection : Entier
                EastConnection : Entier
                SouthConnection: Entier
                WestConnection :  Entier

              Fin;
```

```
Type Connection E = Enregistrement

            ConnectionType :Entier
            ConnectionDirection :Entier

          Fin;
```

Les structures citées sont juste un outil utilisé pour construire nos deux principaux algorithmes.

Algorithmes de résolution d'énigmes : Backtracking

Remarque : Les algorithmes des fonctions qu'on utilisera dans cet algorithme seront dans la partie Annexes.

La complexité de cet algorithme est : O (n^4)

Dans cette algorithme, qui prend en entrée deux Nodes et un tableau de deux dimension qui représentera notre grille va retourner un booléen pour indiquer si l'algorithme a réussi a trouvé une solution ou pas.

Pour ceci :

- On commence à tester si les deux nœuds sont égaux.
- On teste si on peut ajouter un pont entre les deux nœuds :
  - ✓ Si oui, on ajoute le pont.
  - ✓ Sinon on retourne un False.

```
//L'ajout des bords
-Si CanConnect(n1, n2,Largeur,Hauteur) alors
|
|   addConnection(n1, n2);
|
|
| Sinon
|       //si ce bord n'est pas un ajout valide, il suffit
|       Retourner false;
 -FinSi
```

- On teste si on a gagné ou pas :
  - ✓ Si oui, on return un True pour indiquer qu'on vient de gagner
  - ✓ Sinon, on teste s'il n'existe plus d'autre chemin à prendre et qu'on n'a toujours pas encore gagné :
    - ➢ Si oui, on retourne un False

```
-Si SolutionWasFound(puzzleElements,Largeur,Hauteur) alors
|   //Si le jeu est résolu
|   Retourner true;
|
-Sinon Si !hasPotentielMoves(puzzleElements) alors
|
|       // s'il n'y a pas de coups possibles, mais que la partie n'est pas gagnée, retourne false
|       removeConnection(n1, n2);
|       Retourner false;
|
```

- Sinon :
  - On commence à remplir un tableau Nodes qu'on vient de déclarer au début, et contiendra tous les nœuds existantes dans la grille.

```
Pour x de 0 à Largeur faire
 Pour y de 0 à Hauteur faire
   Debut
       Si puzzleElements[x][y] == Island alors
        Debut
           Islands[num] <- puzzleElements[x][y];
           num <- num + 1;
        Fin
   Fin
```

  - Définir la position des deux nœuds dans le tableau.

```
//définir les positions des deux noeuds n1 et n2 dans le tableau des noeuds.
Pour i de 0 à num -1 faire
 Debut
    Si (n1 == Islands[i]) alors
        pIsland1 <- i;
    Si (n2 == Islands[i]) alors
        pIsland2 <- i;
 Fin
```

  - Si le Degree de l'ile qui a l'indice pIsland1 dans le tableau Islands est 0 ou on a arrivé à la dernière case du tableau
  - On entre dans une boucle qui incrémente l'indice pIsland1 et l'indice pIsland2 reçoit le premier, tant que la dégrée d'ile qui est dans la case de l'indice pIsland1 dans notre tableau égale à 0 et l'indice pIsland1 n'a pas dépassé la taille du tableau Islands.
  - Sinon on incrémente la position du deuxième indice pour passer à l'ile suivante.
  - On test dans tous les directions si l'ile identifier par l'indice pIsland2 et un voisin de l'ile qui a l'indice pIslan1.

- si cette condition et valider en test le retour de la fonction récursive findSolution on passe les deux iles qui on l'identifiant pIsland1 et pIsland2 si la fonction retourne Vrai on quitte la fonction et on retourne Vrai ce qui signifie qu'on a trouvé en solution.
- on doit refaire cette procédure jusqu'à passer par toutes les iles dans notre tableau.

```
  Faire
--  Debut
|          //si le degree du noeud n1 == 0 et n2 est le dernier nœud dans le tableau.
|         -Si (Islands[pIsland1].Degree == 0) ou (pIsland2 == num - 1) alors
|          |    -Faire
|          |     |   pIsland1 <- pIsland1 + 1; // incrémenter la position du n1 pour passer au noeud suivant.
|          |     |   pIsland2 <- pIsland1;
|          |     |
|          |     -Tant que(Islands[pIsland1].Degree == 0 Et pIsland1 < num - 1);
|          |
|          |  Sinon
|          |        pIsland2 <- pIsland2 + 1; // Sinon incrémenter la position du n2 pour passer au noeud suivant.
|          |  -FinSi
|          |
|          |  //puis appelez récursivement ces ensembles de nœuds
|          |  -Si (Islands[pIsland1].NorthIsland == Islands[pIsland2]) ou (Islands[pIsland1].EastIsland == Islands[pIsland2]) ou
|          |  |       (Islands[pIsland1].SouthIsland == Islands[pIsland2]) ou (Islands[pIsland1].WestIsland == Islands[pIsland2]) alors
|          |  |
|          |  |    Si FindSolution(Islands[pIsland1], Islands[pIsland2]) alors
|          |  |    |   Retourner true;
|          |  |    FinSi
|          |  -FinSi
|    // si n1 est la dernier dans le tableau ou égal à une position qui n'est pas dans le tableau -> quitter la boucle
--Tant que(!((pIsland2 >= (num - 2)) Et pIsland1 >= (num -1)));


    //si tous les résolveurs avec cette arête ne retournent jamais vrai, vérifiez si c'est le cas avec les doubles arretes
    Si !solve(n1, n2) alors
    -Si (!FindSolution(n1, n2)) alors
    |   removeConnection(n1, n2);
    |   Retourner false;
    -FinSi
```

Algorithmes de génération de puzzle :

Remarque : Les algorithmes des fonctions qu'on utilisera dans cet algorithme seront dans la partie Annexes.

La complexité de cet algorithme est : O (n)

Dans cet algorithme, qui prend en entré les nombres de lignes, colonnes et iles que l'utilisateur souhaite générer, on va pouvoir créer aléatoirement un puzzle qui aura toujours une solution.

Le principe est de prendre une ile aléatoire dans la grille, et commencer à explorer à partir d'elle des nouvelles iles.

Pour ceci, on suit l'algorithme suivant :

- On fixe des coordonnées aléatoires pour notre première ile.
- On augmente le degré de notre ile et on la marque comme Locked.

- On ajoute cette ile à un tableau d'iles « allIslands [] » qui contiendra les iles qu'on va générer.

```
  Faire
--  Debut
|          //si le degree du noeud n1 == 0 et n2 est le dernier nœud dans le tableau.
|         -Si (Islands[pIsland1].Degree == 0) ou (pIsland2 == num - 1) alors
|          |    -Faire
|          |     |   pIsland1 <- pIsland1 + 1; // incrémenter la position du n1 pour passer au noeud suivant.
|          |     |   pIsland2 <- pIsland1;
|          |     |
|          |      -Tant que(Islands[pIsland1].Degree == 0 Et pIsland1 < num - 1);
|          |
|           Sinon
|          |        pIsland2 <- pIsland2 + 1; // Sinon incrémenter la position du n2 pour passer au noeud suivant.
|           -FinSi
|
|          //puis appelez récursivement ces ensembles de nœuds
|          -Si (Islands[pIsland1].NorthIsland == Islands[pIsland2]) ou (Islands[pIsland1].EastIsland == Islands[pIsland2]) ou
|          |       (Islands[pIsland1].SouthIsland == Islands[pIsland2]) ou (Islands[pIsland1].WestIsland == Islands[pIsland2]) alors
|          |
|          |    Si FindSolution(Islands[pIsland1], Islands[pIsland2]) alors
|          |    |   Retourner true;
|          |    FinSi
|           -FinSi
|   // si n1 est la dernier dans le tableau ou égal à une position qui n'est pas dans le tableau -> quitter la boucle
--Tant que(!((pIsland2 >= (num - 2)) Et pIsland1 >= (num -1)));


   //si tous les résolveurs avec cette arête ne retournent jamais vrai, vérifiez si c'est le cas avec les doubles arretes
   Si !solve(n1, n2) alors
   -Si (!FindSolution(n1, n2)) alors
   |   removeConnection(n1, n2);
   |   Retourner false;
   -FinSi
```

- Grace à la fonction « findRandomNgb » on va récupérer un voisin pour la première ile créé. Cette fonction choisi arbitrairement la direction et la distance du voisin par rapport à l'ile initial.
- On augmente le degré de l'ile voisin, et l'ajoute au tableau d'iles.
- On connecte les deux iles. Ceci implique que tous ce qui se trouve entre l'ile initiale et le voisin vont être marqué comme Locked, car ceci entrainera des problèmes lors de la résolution.

```
| x<---- RandomNB(Lines);
| {retourne un nombre aléairatoire entre 0 et le nombre passer dans les paramètre}
| y<---- RandomNB(Columns);
| i<---- 0;
| board[x][y].Degree<---- board[x][y].Degree+1
| board[x][y].Locked<---- Vrai
| allIslands[i]<----board[x][y]
| i<---- i+1
| randomNgb<---findRandomNgb(board[x][y]);
| temp<---- randomNgb
| temp.Degree<---- temp.Degree+1
| connect(temp,board[x][y],board)
| allIslands[i]<---- temp
| i<---- i+1
```

- La sélection du voisin, l'augmentation du degré et la connexion entre les deux vont se répéter tant qu'on n'arrive pas à atteindre le nombre d'ile que l'utilisateur veut générer. Par contre il faut noter qu'à chaque fois on choisira arbitrairement une ile à partir des iles générer, pour être comme ile initiale.

```
tant que (i<Nbile) faire
|
| temp<---- SelectRandomNgb(allIslands,i)
| random<----findRandomNgb(temp)
| temp.Degree<---- temp.Degree+1
| randomNgb.Degree<---- randomNgb.Degree+1
| connect(temp,randomNgb,board)
| allIslands[i]<---- temp
| i<---- i+1
|
|Fin Tant que
```

Annexes :

FindSolution : Partie 1 :

```
Fonction FindSolution(n1 : Island ,n2 : Island, puzzleElements  : Tab[][], Largeur:Entier ,Hauteur :Entier) : Boolean

   Var     num,x,y : Entier
           pIsland1 : Entier
           pIsland2 : Entier
           Islands   : Tab [] // Tableau de tous les nœuds du jeu

-Debut
|
|  //On teste si les deux noeuds sont égaux
|  Si (n1 == n2) alors
|      Retourner false;
|
|
|  //L'ajout des bords
|  -Si CanConnect(n1, n2,Largeur,Hauteur) alors
|  |
|  |   addConnection(n1, n2);
|  |
|  |
|  | Sinon
|  |       //si ce bord n'est pas un ajout valide, il suffit
|  |       Retourner false;
|  |  -FinSi
|  |
|  |
|  -Si SolutionWasFound(puzzleElements,Largeur,Hauteur) alors
|  |   //Si le jeu est résolu
|  |   Retourner true;
|  |
|  -Sinon Si !hasPotentielMoves(puzzleElements) alors
|  |
|  |       // s'il n'y a pas de coups possibles, mais que la partie n'est pas gagnée, retourne false
|  |       removeConnection(n1, n2);
|  |       Retourner false;
|  |
```

Partie 2 :

```
|
-Sinon Si !hasPotentielMoves(puzzleElements) alors
|
|       // s'il n'y a pas de coups possibles, mais que la partie n'est pas gagnée, retourne false
|       removeConnection(n1, n2);
|       Retourner false;
|
|
 Sinon
 -Debut
|   //Remplissage du Tableau Islands par tous les noeuds existants dans le bord.
|
|   num <- 0 ;
|   x <- 0 ;
|   y <- 0 ;
|
|    Pour x de 0 à Largeur faire
|     Pour y de 0 à Hauteur faire
|       Debut
|           Si puzzleElements[x][y] == Island alors
|            Debut
|               Islands[num] <- puzzleElements[x][y];
|               num <- num + 1;
|            Fin
|       Fin
|
|
|   //définir les positions des deux noeuds n1 et n2 dans le tableau des noeuds.
|   Pour i de 0 à num -1 faire
|    Debut
|       Si (n1 == Islands[i]) alors
|           pIsland1 <- i;
|       Si (n2 == Islands[i]) alors
|           pIsland2 <- i;
|    Fin
|
```

Partie 3 :

```
    Faire
  --  Debut
      //si le degree du noeud n1 == 0 et n2 est le dernier nœud dans le tableau.
     -Si (Islands[pIsland1].Degree == 0) ou (pIsland2 == num - 1) alors
        -Faire
           pIsland1 <- pIsland1 + 1; // incrémenter la position du n1 pour passer au noeud suivant.
           pIsland2 <- pIsland1;

         -Tant que(Islands[pIsland1].Degree == 0 Et pIsland1 < num - 1);

       Sinon
           pIsland2 <- pIsland2 + 1; // Sinon incrémenter la position du n2 pour passer au noeud suivant.
      -FinSi

      //puis appelez récursivement ces ensembles de nœuds
      -Si (Islands[pIsland1].NorthIsland == Islands[pIsland2]) ou (Islands[pIsland1].EastIsland == Islands[pIsland2]) ou
           (Islands[pIsland1].SouthIsland == Islands[pIsland2]) ou (Islands[pIsland1].WestIsland == Islands[pIsland2]) alors

        Si FindSolution(Islands[pIsland1], Islands[pIsland2]) alors
          Retourner true;
        FinSi
       -FinSi
    // si n1 est la dernier dans le tableau ou égal à une position qui n'est pas dans le tableau -> quitter la boucle
  --Tant que(!((pIsland2 >= (num - 2)) Et pIsland1 >= (num -1)));


    //si tous les résolveurs avec cette arête ne retournent jamais vrai, vérifiez si c'est le cas avec les doubles arretes
    Si !solve(n1, n2) alors
    -Si (!FindSolution(n1, n2)) alors
       removeConnection(n1, n2);
       Retourner false;
    -FinSi

  -FinSi
       Retourner false;
Fin
```

CanConnect : Partie 1 :

```
Fonction CanConnect(Island n1, Island n2, puzzleElements:Tab[][],Largeur:Entier,Hauteur:Entier): Boolean


      Var x,y,k,x2,y2,y1 : Entier


-Debut
      x <- 0 ;
      y <- 0 ;

      //si l'un des noeud est null
      Si (n1 == null) ou (n2 == null) faire
       Retourner false;
       -FinSi
      //Si l'un des noeuds n'as aucun arête supplémentaires à ajouter.
      Si (n1.Degree == 0) ou (n2.Degree == 0)
       Retourner false;
       -FinSi


      //Si Il n'y a aucun arrete entre les deux noeuds.
      Si(n1.NorthIsland != n2) Et (n1.EastIsland != n2) Et (n1.SouthIsland != n2) Et (n1.WestIsland != n2) alors
       Retourner false;
       -FinSi


      //trouver la position de n1 sur le puzzle.
      Tantque(puzzleElements[x][y] != n1) faire
      -Debut
       y <- 0;

        Tantque (puzzleElements[x][y] != n1 Et y < Largeur-1) faire
          y <- y + 1;
        -Fintantque

        Si puzzleElements[x][y] != n1 alors
```

Partie 2 :

```
    Si puzzleElements[x][y] != n1 alors
       x <- x + 1;
     -FinSi
  -Fintantque

  //si l'arête à ajouter est déjà à une valeur de 2
  Si (n1.NorthIsland == n2) alors
  - Debut
     Si(n1.NorthConnection == 2) alors
        Retourner false;

     Sinon

        //s'il y a déjà un croisement d'arête entre ces deux nœuds
        Pour k de y-1 à faire
         -Debut
            Si(puzzleElements[x][k] == n2) alors
               break;

             Sinon

               Si(puzzleElements[x][k].ConnectionDirection == 1) alors

                  Retourner false;
               -FinSi
            -FinSi
         -Finpour
     -FinSi
  -FinSi

  //si l'arête à ajouter est déjà à une valeur de 2
   Si (n1.EastIsland == n2) alors
   -Debut
     -Si (n1.EastConnection == 2) alors
        Retourne false;

     Sinon
```

Partie 3 :

```
    Sinon


       //renvoie false s'il y a déjà un croisement d'arête entre ces deux nœuds
       Pour x1 de x+1 à Largeur faire
          Si(puzzleElements[x2][y] == n2) alors
             break;
           Sinon
            Si(puzzleElements[x2][y].ConnectionDirection == 2) alors
               Retourner false;
            -FinSi
          -FinSi
       -Finpour
    -FinSi
  -FinSi

 //si l'arête à ajouter est déjà à une valeur de 2
 Si (n1.SouthIsland == n2) alors
  Debut
    Si (n1.SoutConnection == 2) alors
       Retourner false;

     Sinon

       //renvoie false s'il y a déjà un croisement d'arête entre ces deux nœuds
       Pour y1 de y+1 à Largeur faire
          Si(puzzleElements[x][y1] == n2) alors
             break;
           Sinon
              Si(puzzleElements[x2][y].ConnectionDirection == 1) alors
                 Retourner false;
            -FinSi
         -Finpour
     FinSi
  -FinSi
```

Partie 4 :

```
|       //si l'arête à ajouter est déjà à une valeur de 2
|       -Si (n1.WestIsland == n2) alors
|       | Debut
|       |   Si (n1.WestConnection == 2) alors
|       |       Retourner false;
|       |
|       Sinon
|       |
|       |   //renvoie false s'il y a déjà un croisement d'arête entre ces deux nœuds
|       |    -Pour x2 de x-1 à faire
|       |    |
|       |    |  -Si(puzzleElements[x2][y] == n2) alors
|       |    |  |    break;
|       |    |   Sinon
|       |    |  |     Si(puzzleElements[x2][y].ConnectionDirection == 1) alors
|       |    |  |        Retourner false;
|       |    |   -FinSi
|       |    -Finpour
|       -FinSi
|
|   Retourner true;
-Fin
```

SolutionWasFound :

```
Fonction SolutionWasFound(baord : Tab[][],Largeur :Entier, Hauteur:Entier) : Boolean

    Var x,y :Entier;

-Debut
|       x <-0;
|       y <-0;
|
|      -Pour  x de 0 à Largeur  faire
|      |    -Pour y de 0 à  Hauteur faire
|      |    |
|      |    |    -Si puzzleElements[x][y] == Island alors
|      |    |    |
|      |    |    |     Si puzzleElements[x][y].Degree() > 0 alors
|      |    |    |        Retourne false;
|      |    |     -FinSi
|      |     -Finpour
|       -Finpour
|   Retourn true;
|
|
-Fin;
```

HasPotentielMoves :

```
Fonction hasPotentielMoves(baord : Tab[][],Largeur :Entier, Hauteur:Entier) : Boolean
-Debut
|
|   -Pour  x de 0 à Largeur  faire
|   |   Pour y de 0 à  Hauteur faire
|   |
|   |    -Debut
|   |   |
|   |   |       Si(puzzleElements[x][y] == Island) alors
|   |   |
|   |   |       -Debut
|   |   |       |    Si (puzzleElements[x][y].Degree > 0) alors
|   |   |       |    -Debut
|   |   |       |   |   // s'il y a un déplacement valide avec ce nœud, renvoie true
|   |   |       |   |   -Si CanConnect(puzzleElements[x][y],puzzleElements[x][y].NorthIsland) alors
|   |   |       |   |   |
|   |   |       |   |   |   Retourne true;
|   |   |       |   |    -FinSi
|   |   |       |   |   Si CanConnect(puzzleElements[x][y],puzzleElements[x][y].EastIsland) alors
|   |   |       |   |   |
|   |   |       |   |   |   Retourne true;
|   |   |       |   |    -FinSi
|   |   |       |   |   Si CanConnect(puzzleElements[x][y],puzzleElements[x][y].SouthIsland) alors
|   |   |       |   |   |
|   |   |       |   |   |   Retourne true;
|   |   |       |   |    -FinSi
|   |   |       |   |   Si CanConnect(puzzleElements[x][y],puzzleElements[x][y].WestIsland) alors
|   |   |       |   |  |
|   |   |       |   |  |   Retourne true;
|   |   |       |   |   -FinSi
|   |   |       |    -FinSi
|   |   |       -FinSi
|   |    -FinPour
|   -FinPour
|       Retourne false;
|
-Fin
```

AddConnection : Partie 1 :

```
Function addConnection(Island n1, Island n2,puzzleElements :Tab[][])


   Var  x,y2 :Entier


-Debut
|    x <- 0;
|    y <- 0;
|
|       //decrements the Islands' needed degree count
|       n1.degree <- n1.degree - 1;
|       n2.degree <- n2.degree - 1;
|
|       //finds the position of n1 on the puzzleElements
|        -Tantque(puzzleElements[x][y] != n1) faire
|       |   y <- 0;
|       |    -Tantque(puzzleElements[x][y] != n1) Et (y < 9) faire
|       |    |   y<-y+1;
|       |    -Fintantque
|       |
|       |  Si(puzzleElements[x][y] != n1) alors
|       |       x<-x+1;
|       |
|        -Fintantque
|
|       //updates the Connection's value (0, 1, or 2)
|        -Si (n1.NorthIsland == n2) alors
|       |
|       |   n1.NorthConnection  <- n1.NorthConnection + 1;
|       |   n2.SouthConnection  <- n2.SouthConnection + 1;
|       |
|       |   -Pour y2 de y-1 à 0 faire
|       |   |   -Si(puzzleElements[x][y2] == n2) alors
|       |   |   |   break;
|       |   |   | Sinon{
|       |   |   |
```

Partie 2 :

```
    -Pour y2 de y-1 à 0 faire
    |   -Si(puzzleElements[x][y2] == n2) alors
    |   |   break;
    |   | Sinon{
    |   |
    |   |   puzzleElements[x][y2].ConnectionType <- puzzleElements[x][y2].ConnectionType + 1;
    |   |   puzzleElements[x][y2].ConnectionDirection <- 2;
    |   |  }
    |   -FinSi
    |  y2 <- y2 - 1;
    -Finpour
|
-FinSi


-Sinon Si (n1.EastIsland == n2) alors
|
|   n1.EastConnectionn <- n1.EastConnection + 1;
|   n2.WestConnection  <- n2.WestConnection + 1;
|
|   Pour x2 de x+1 à Largeur faire
|       Si(puzzleElements[x2][y] == n2) alors
|           break;
|        Sinon{
|
|           puzzleElements[x2][y].ConnectionType <- puzzleElements[x2][y].ConnectionType + 1;
|           puzzleElements[x2][y].ConnectionDirection <- 1;
|        }
|       FinSi
|   Finpour
|
-FinSi
```

Partie 3 :

```
|       -Sinon Si (n1.SouthIsland == n2) alors
|       |   n1.SouthConnectionn <- n1.SouthConnection + 1;
|       |   n2.NorthConnection  <- n2.NorthConnection + 1;
|       |
|       |   -Pour y2 de y+1 à Largeur faire
|       |   |   -Si(puzzleElements[x][y2] == n2) alors
|       |   |   |   break;
|       |   |   |  Sinon{
|       |   |   |
|       |   |   |   puzzleElements[x][y2].ConnectionType <- puzzleElements[x][y2].ConnectionType + 1;
|       |   |   |   puzzleElements[x][y2].ConnectionDirection <- 2;
|       |   |   | }
|       |   |   -FinSi
|       |    -Finpour
|       -FinSi
|
|       -Sinon Si (n1.WestIsland == n2) alors
|       |
|       |   n1.WestConnection <- n1.WestConnection + 1;
|       |   n2.EastConnection  <- n2.EastConnection + 1;
|       |
|       |   -Pour x2 de x+1 à 0 faire
|       |   |   -Si(puzzleElements[x2][y] == n2) alors
|       |   |   |   break;
|       |   |   |
|       |   |    Sinon
|       |   |   |   puzzleElements[x2][y].ConnectionType <- puzzleElements[x2][y].ConnectionType + 1;
|       |   |   |   puzzleElements[x][y2].ConnectionDirection <- 1;
|       |   |   |
|       |   |    -FinSi
|       |   |   x2 <- x2 - 1;
|       |    -Finpour
|       |
|       -FinSi
|       Sinon
|         Ecrire("n2 n'est pas un paramètre valide pour n1")
-Fin
```

RemoveConnection : Partie 1 :

```
Function removeConnection(Island n1, Island n2,puzzleElements :Tab[][],Largeur :Entier, Hauteur:Entier)
    Var  x :Entier
         y :Entier
-Debut
|    x <- 0;
|    y <- 0;
|
|       //incrémente le nombre de degrés nécessaires aux nœuds
|       n1.degree <- n1.degree + 1;
|       n2.degree <- n2.degree + 1;
|
|       //trouver la position de n1 sur le puzzle
|        -Tantque(puzzleElements[x][y] != n1) faire
|        |   y <- 0;
|        |    Tantque(puzzleElements[x][y] != n1) Et (y < Largeur-1) faire
|        |    |   y<-y+1;
|        |    -Fintantque
|        |
|        |   Si(puzzleElements[x][y] != n1) alors
|        |    |   x<-x+1;
|        |    -FinSi
|        -Fintantque
|
|       //mettre à jour la valeur de l'arête (0, 1 ou 2)
|        -Si (n1.NorthIsland == n2) alors
|        |
|        |   n1.NorthConnection  <- n1.NorthConnection - 1;
|        |   n2.SouthConnection  <- n2.SouthConnection - 1;
|        |
|        |   -Pour y2 de y-1 à 0 faire
|        |   |   -Si(puzzleElements[x][y2] == n2) alors
|        |   |   |   break;
|        |   |   | Sinon{
```

Partie 2 :

```
|   -Pour y2 de y-1 à 0 faire
|   |   -Si(puzzleElements[x][y2] == n2) alors
|   |   |   break;
|   |   | Sinon{
|   |   |
|   |   |   puzzleElements[x][y2].ConnectionType <- puzzleElements[x][y2].ConnectionType - 1;
|   |   |
|   |   |       Si puzzleElements[x][y2].ConnectionType == 0 alors
|   |   |           puzzleElements[x][y2].ConnectionDirection <- 0;
|   |   |  }
|   |    -FinSi
|   |   y2 <- y2 -1;
|    -Finpour
|
|-FinSi
 -Sinon Si (n1.EastIsland == n2) alors
|
|   n1.EastConnectionn <- n1.EastConnection - 1;
|   n2.WestConnection  <- n2.WestConnection - 1;
|
|  Pour x2 de x+1 à Largeur faire
|       Si(puzzleElements[x2][y] == n2) alors
|           break;
|    - Sinon
|   |
|   |   puzzleElements[x2][y].ConnectionType <- puzzleElements[x2][y].ConnectionType - 1;
|   |
|   |       Si puzzleElements[x2][y].ConnectionType == 0 alors
|   |           puzzleElements[x2][y].ConnectionDirection <- 0;
|   |
|    -FinSi
|  Finpour
 -FinSi
```

Partie 3 :

```
        -Sinon Si (n1.SouthIsland == n2) alors
        |   n1.SouthConnectionn <- n1.SouthConnection - 1;
        |   n2.NorthConnection  <- n2.NorthConnection - 1;
        |   -Pour y2 de y+1 à Largeur faire
        |   |    -Si(puzzleElements[x][y2] == n2) alors
        |   |   |   break;
        |   |   |  Sinon{
        |   |   |   puzzleElements[x][y2].ConnectionType <- puzzleElements[x][y2].ConnectionType - 1;
        |   |   |
        |   |   |      Si puzzleElements[x][y2].ConnectionType == 0; alors
        |   |   |         puzzleElements[x][y2].ConnectionDirection <- 0;
        |   |   | }
        |   |    -FinSi
        |    -Finpour
        -FinSi

        -Sinon Si (n1.WestIsland == n2) alors
        |   n1.WestConnection <- n1.WestConnection - 1;
        |   n2.EastConnection  <- n2.EastConnection - 1;
        |   -Pour x2 de x+1 à 0 faire
        |   |   -Si(puzzleElements[x2][y] == n2) alors
        |   |   |   break;
        |   |   |
        |   |    Sinon
        |   |   |
        |   |   |   puzzleElements[x2][y].ConnectionType <- puzzleElements[x2][y].ConnectionType - 1;
        |   |   |      Si puzzleElements[x][y2].ConnectionDirection == 0 alors
        |   |   |           puzzleElements[x][y2].ConnectionDirection <- 0;
        |   |    -FinSi
        |   |   x2 <- x2 - 1;
        |    -Finpour
        |
        |-FinSi
         Sinon
           Ecrire("n2 n'est pas un paramètre valide pour n1")
|
-Fin
```

Algorithme Generate :

```
Generate(Nbile:Entier,Lines: Entier,Columns: Entier, board: Node[][]) : boolean
            var allIslands : Node[]
            var x,y,i: Entier
            var temp, randomNgb: Node

  Début
  |
  | x<---- RandomNB(Lines);
  | {retourne un nombre aléairatoire entre 0 et le nombre passer dans les paramètre}
  | y<---- RandomNB(Columns);
  | i<---- 0;
  | board[x][y].Degree<---- board[x][y].Degree+1
  | board[x][y].Locked<---- Vrai
  | allIslands[i]<----board[x][y]
  | i<---- i+1
  | randomNgb<---findRandomNgb(board[x][y]);
  | temp<---- randomNgb
  | temp.Degree<---- temp.Degree+1
  | connect(temp,board[x][y],board)
  | allIslands[i]<---- temp
  | i<---- i+1
  |
  | tant que (i<Nbile) faire
  | | |
  | | temp<---- SelectRandomNgb(allIslands,i)
  | | random<----findRandomNgb(temp)
  | | temp.Degree<---- temp.Degree+1
  | | randomNgb.Degree<---- randomNgb.Degree+1
  | | connect(temp,randomNgb,board)
  | | allIslands[i]<---- temp
  | | i<---- i+1
  | | |
  | |Fin Tant que
  |Fin
```

Connectable :

```
Connectable (a: Island, b: Island, puzzleElements: Tab[][]) :Boolean

 Début
 |   Si (a.X == b.X) alors
 |   |   Pour I de Min ( a.Y, b.Y) à Max(a.X, b.X) faire
 |   |   | Si (puzzleElements[a.X][i].Locked == true) alors
 |   |   |   |   retourner false ;
 |   |   |   | FinSI
 |   |   |FinPour
 |   |FinSI
 |
 |
 |   Si (a.Y == b.Y) alors
 |   | pour I de Min(a.X, b.X) à Max(a.Y, b.Y) faire
 |   |   | Si (puzzleElements[i][a.Y].Locked == true) alors
 |   |   |   |   retourner false ;
 |   |   |   |
 |   |   |   |Fin SI
 |   |   |Fin Pour
 |   |Fin SI
 -Fin
```

Connect :

```
Connect (a: Island, b: Island, puzzleElements: Island[][]){
Début
|   Si (a.X == b.X) alors
|   |   pour I de Min ( a.Y, b.Y) jusqu'à Max(a.X, b.X)
|   |   |
|   |   |  puzzleElements[a.X][i].Locked<----Vrai ;
|   |   |
|   |   |Fin Pour
|   |Fin SI
|
|
|   Si (a.Y == b.Y) alors
|   | pour I de Min ( a.X, b.X) à Max(a.Y, b.Y) faire
|   |   |
|   |   | puzzleElements[i][a.Y].Locked<---Vrai ;
|   |   |
|   |   |Fin Pour
|   |Fin SI
|Fin
```

SelectRandomNgb : Partie 1 :

```
SelectRandomNgb(allIslands: Tab[] , i: Entier) : Island
{retourne un sommet des sommets qui sont  déjà fixés}
   var randomNgb;
   var j;
  Début
  |
  | j<---- RandomNB(i);
  | Retourne allIslands[j];
  |
  Fin


findRandomNgb(ile: Island, puzzleElements: Island[][], Lines: Entier, Columns: Entier, )
{retourne un voisin aléairatoire du sommet ile passer dans les paramètre }

                var ok: boolean
                var x,y: Entier
                var randomNgb: Island
                var Direction: entier
               {0 West, 1 North , 2 East , South}
    Début
    |  y<---- ile.y;
    |  x<---- ile.x;
    |  ok<---- Faux;
    |   Tantque (!ok) alors
    |   | Direction<---- RandomNB(3);
    |   | si (Direction == 0) alors
    |   |  | y<----ile.Y;
    |   |  | x<---- RandomNB(ile.X);
    |   |  |Fin SI
    |   |
    |   | si (Direction == 1) alors
    |   |  | x<----ile.X;
```

Partie 2 :

```
    si (Direction == 1) alors
      x<----ile.X;
      y<---- RandomNB(ile.Y);
     Fin SI

    si (Direction == 2) alors
      y<----ile.Y;
      x<---- 1+ ile.X + RandomNB((Lines-2)-ile.X);
     Fin SI

    si (Direction == 3) alors
      x<----ile.X;
      y<---- 1+ ile.Y + RandomNB((Columns-2)-ile.Y);
     Fin SI

    si( x > 0 et x < Lines-1) alors

      si(y > 0 et y < Columns-1 ) alors

          si (puzzleElements[x][y].Degree == 0 et Connectable(ile puzzleElements[x][y])) alors

             puzzleElements[x][y].Locked<-True;
             randomNgb <- puzzleElements[x][y];
             ok<- True
          Fin SI
       Fin SI

     Fin SI
   Fin Tantque
  Retourner randomNgb;
Fin
```