1 Utilisation

Le projet se trouve dans l'archive jointe avec ce compte rendu, c'est la copie du workspace d'eclipse. Il est aussi disponible en tant qu'archive git à cette adresse : *git ://github.com/Ekareya/Tp2.git*. Il est composé de 6 fichiers .java.

AStarPa.java Une implementation de l'algorithme A*, la fonction heuristique utilisé étant la fonction nulle, il est équivalent à un algorithme de Dijkstra

LabyGenerator.java Classe qui implémente la transformation d'un graphe «grille» en «labyrinthe».

ToolkitPa.java Une bibliothèque de fonction manipulant les graphes.

OldTp2.java La première version du tp. Créer un labyrinthe de taille n×n , et trouve le plus cours chemin à l'aide de l'algorithme de Dijkstra de la bibliothèque GraphStream.

Tp2.java La deuxième version du tp. Génère un graphe «grille» de taille n×m à l'aide du produit cartésien de deux chaînes, puis le labyrinthe est crée à l'aide de LabyGenerator.

Menu.java La version finale du tp. Un menu crée à l'aide de Processing permet de régler certain paramètre pour la génération du labyrinthe sans avoir à les changer dans le code. (S'exécute en tant qu'applet.)

2 Représentation

J'ai choisit de représenter le labyrinthe en prenant comme arête les «non-murs» et comme nœud les cases. J'avais commencé par représenter le labyrinthe en prenant comme arête les murs, et les coins des murs comme nœuds mais cela c'est très très vite révélé peu pratique. Cependant je me suis quand même servit de cette représentation dans OldTp2 afin que le résultat généré soit plus facilement compréhensible pour un œil humain.

3 Création

Pour la génération du labyrinthe, je suis parti d'un graphe grille crée à l'aide du générateur intégré à graphstream (GridGenerator) auquel j'ai donné un poids aléatoire (entre 0 et 1) à chaque arête. Une fois obtenu ce graphe je lui ai simplement appliqué l'algorithme de Prim, cet algorithme lorsqu'il est appliqué à un graphe connexe renvoie un arbre couvrant de poids minimal. L'arbre étant couvrant cela veux dire que tout le graphe reste connexe qui est une propriété que l'ont souhaitait garder. De plus les poids des arcs étant aléatoire, il est difficile d'obtenir un labyrinthe qui ne ressemble pas à un labyrinthe enfin ce que je veux dire par la c'est que quand j'oubliais de mettre les poids sur les arêtes j'avais un arbre couvrant avec de longue branche ce qui faisait que la solution était facile à trouver pour un humain, alors qu'avec des poids aléatoire il y a beaucoup de petite branche donc c'est plus dur pour l'œil de trouver une solution.

Mettre le paramètre DeadEnd à 0 afin d'obtenir un tel labyrinthe.

Donc je m'étais arrêté la avant de commencer à travailler sur le projet d'i.a. J'avais besoin d'utiliser un labyrinthe pour faire le plateau de jeu, cependant le labyrinthe que j'avais développe précédemment est plus d'injouable, du fait que c'était un arbre (donc beaucoup de cul-de-sac et surtout pas de possibilité de faire «demi-tour» sans repasser par le même chemin, ce qui est assez gênant lorsqu'on est poursuivit).

Dans le graphe représentant les chemins possible dans un labyrinthe, un cul-de-sac est simplement un nœud de degré 1. Pour éliminer les culs-de-sac, je recherche dans le graphe tout les nœuds de degré 1 que je connecte à un autre nœud, de préférence un autre cul-de-sac.

Mettre DeadEnd entre 1 et 10, et 4-cycle à *false*

Donc en enlevant tout les culs-de-sac, je me retrouvait avec beaucoup de 4-cycle dans mon graphe ce qui une fois que l'on transformait le graphe des chemins en plateau de jeu donnait un gros trou de 2×2 ce qui n'est pas très joli.

Pour les enlever, j'ai d'abord recherché sur internet les algorithmes de détection de cycle d'une certaines longueur, c'était un peu compliqué à mettre en œuvre. J'ai finit par remarquer que la plupart des 4-cycles du graphe de chemin possède au moins un nœud d'ordre 2 et qu'il ne peux pas exister de cycle d'ordre inférieur du fait de la structure de départ du graphe. Donc l'heuristique consiste à visiter tout les sommets d'ordre 2 puis de regarder si l'union des voisins des voisins du sommet contient un autre sommet que le sommet de départ, si c'est le cas on a un 4-cycle. Cet heuristique ne détecteras pas les 4-cycles n'ayant pas de sommet égal à deux, mais cela n'arrive que rarement, car soit les 4-cycles sont collés les uns aux autres soit il ne contient que des sommets d'ordre supérieur à 2, ce qui est fort improbable car les cycles sont crées lors de l'étape d'élimination des culs-de-sac dans un arbre qui est par définition acyclique. Comme on ne rajoute qu'un nombre assez limités d'arêtes et qu'on connecte en priorité les culs-de-sac entre eux on va avoir du mal a créer un gros tas de 4-cycle. Quand aux 4-cycles avec que des nœuds d'ordre supérieur à 2, je pense que l'on peu affirmer qu'ils sont impossibles à obtenir à la suite des manipulations décrite précédemment. Une fois un 4-cycle détecte, on cherche à enlever une arête du cycle tout en ne fabriquant pas de cul-de-sac. Si cela n'est pas possible on rajoute une arête vers l'extérieur du cycle depuis le sommet d'ordre 2 et on recommence. Le rajout d'une arêtes vers l'extérieur du cycle peu produire un nouveau cycle dans des sommets déjà vérifié, c'est pourquoi l'heuristique vérifie plusieurs fois tout les sommets jusqu'à ce qu'il ne trouve aucun 4-cycle.

Mettre DeadEnd à 10, 4-cycle à *true*,BottleNeck à *false* et prendre un graphe de taille inférieur à 10×10

Donc à partir de la, j'ai un labyrinthe sans cul de sac ni 4-cycle (sauf très rarement). Il me reste cependant un problème qui arrive de temps en temps dans les labyrinthe de petite taille (enfin dans les grands aussi mais beaucoup moins souvent). J'obtenais ce que j'appelais un cul-de-sac cyclique. Je suis donc retourné sur internet pour voir ce que je pouvais y trouver, et j'ai appris que ce que je cherchais s'appelait un nœud maître et que leur détection n'était pas évidente donc j'ai choisit de refaire une heuristique afin de régler mon problème. Celle ci va utiliser une copie du graphe et le réduire en remplaçant toutes les chaînes par des arêtes. En gros je prend un sommet d'ordre 2, je supprime ses arêtes avec ses deux voisins et je connecte les voisins entre eux. Si les voisins sont déjà connecté, je ne les reconnecte pas afin de conservé un graphe simple. Une fois la simplification faite, j'ai remarqué que le cycle attaché au n oe ud maître se résumait à un nœud d'ordre 1 relié au nœud maître. En fait cet heuristique permet de détecter les nœuds maîtres dont les esclaves forment un cycle sans corde (parfois avec, mais il faut une configuration bien particulière). Et une fois détecté je rajoute simplement une arête depuis le cycle vers l'extérieur afin de faire disparaître le nœud maître. (et je reteste pour savoir si je n'ai pas crée de 4-cycle...)

decommenter les lignes 313 à 323 de LabyGenerator en mettant BottleNeck à *true* pour afficher le graphe sans chaine

Maintenant j'ai donc un labyrinthe sans cul de sac, avec très très peu de 4-cycle et sans nœud maître gênant pour la jouabilité du pac-man.

4 Parcours

Pour le parcours j'utilisais l'algorithme de Dijkstra de graphstream, sauf que cela m'obligeait à enlever, une fois le labyrinthe généré, toutes les arêtes ou l'on ne pouvait pas passer, du coup si je voulais construire mon labyrinthe en 3D et que je n'ai pas réussit à passer le viewer de graphstream en 3D cela donnait des résultats assez bizarre. J'ai donc légèrement modifier mon implementation de l'algorithme A* que j'utilise pour le projet d'i.a. pour qu'il agisse comme un Dijkstra (l'heuristique étant la fonction nulle) et qu'il ne prennent pas de raccourci non autorisé.
ligne 24 de Tp2 mettre laby.setPruning à false pour voir ce que je veux dire