1 Présentation

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/pacman.git*. Il est composé de 9 fichiers .java.

AStarPa.java Une implementation de l'algorithme A*, la distance utilisé est la distance de manhattan et le cout est la fonction constante 1.

LabyGenerator.java Classe qui implémente la transformation d'un graphe «grille» en «labyrinthe». Voir compteRenduTp2.pdf pour plus d'explication.

ToolkitPa.java Une bibliothèque de fonction manipulant les graphes.

Config.java Contient toutes les variables utiles à l'ensemble des autres classes. Voir les commentaires de la sources pour plus de details.

PacmanGame.java Classe principale, s'execute en temps qu'applet ou en temps qu'application. Implémentée en utilisant processing en tant que librairie.

Perso.java Classe abstraite definissant les personnages.

Monstre.java Classe fille de Perso qui gère l'affichage et le comportement des monstres.

Pacman.java Classe fille de Perso qui gère l'affichage et le deplacement de pacman.

BenchMark.java Classe permettant de comparer l'efficacité en temps de differents algorithme de recherche de chemin. (fonctionne très mal sous windows [voir ici pourquoi])

2 Description des classes

PacmanGame.java La classe est composé de deux fonctions,*setup()* et *draw()*. Ce sont les deux fonctions obligatoirement présente dans un sketch processing.

setup() est automatiquement appelé au lancement de l'application. Elle initialise l'environement de jeu et dessine le labyrinthe

draw() est appelé periodiquement, à chaque appel la fonction efface les sprites des personnages (*Perso.enlever()*), les deplace (*Perso.deplacer()*), les reaffiche à leur nouvelle position (*Perso.afficher()*), verifie les eventuelles collisions puis affiche les«dialogues» i.e. score ou message de fin.

Perso.java

x, y Coordonée du personnage.

direction Direction du mouvement(UP, DOWN, LEFT, RIGHT)

$frame$ Compteur de frame.

vitesse Nombre de frame necessaire pour arriver à la prochaine cases.

deplacer()
1: **if** $frame = vitesse$ **then**
2: deplace le personnage si cela est possible selon *direction* ;
3: $direction \leftarrow orienter()$;
4: **else**
5: $frame \leftarrow frame + 1$;
6: **end if**

orienter() Recupère la direction de la prochaine case,via un evenement clavier pour Pacman et selon le comportement à adopter pour les Monstres. C'est dans cette fonction que reside «l'inteligence» des P.N.J.s

$affiche r()/enlever()$ Affiche/Efface le personnage l'ecran, la position est determiné gràce a x, y et $frame/vitesse$, le rapport $frame/vitesse$ represente le niveau d'avancement entre deux cases(e.g. la vilaine Pinky est en *(3,2)*, sa valeur de $frame$ est 12 et sa *vitesse* 24. Cela veut dire qu'elle en est à la moitié de son deplacement vers la prochaine case.Elle seras donc affichée entre les deux cases.)

Monstre.java

C c'est une pile de case représentant le chemin que le monstre à envie de suivre

$orienter()$

```
1: if malade = true then
2:   orientation ← hunted();
3: else
4:   \\ (ia = 1) ⇒ passif, se règle lors de la création du monstre.
5:   if monstreEstPassif() then
6:     orientation ← welcomeToRandomLand();
7:   else
8:     orientation ← aStarIsBorn();
9:   end if
10: end if
```

$hunted()$ Le monstre suit le chemin stocké tant qu'il n'apercoit pas Pacman. Lorsqu'il l'apercoit il essaye de s'en eloigner le plus possible.

$welcomeToRandomLand()$ Comportement passif du monstre. Il choisit une case au hasard dans le labyrinth et s'y rend (le chemin étant stocké dans C). Une fois arrivé à sa destination, il s'en choisit une nouvelle toujours au hasard...

$aStarIsBorn()$ Le monstre suis un comportement passif tant qu'il n'apercoit pas Pacman. Une fois pacman repéré il calcul le plus court chemin pour y arriver. Une fois le chemin calculé il le suivra tant que son prochain mouvement le raproche de pacman. Si le prochain mouvement ne le rapproche pas, il recalcule sont chemin et le suit même si cela l'eloigne temporairement de pacman. Afin de simuler une stratégie d'encerclement, lorsque le mouvement d'un monstre l'amene sur une case ou est deja présent un autre monstre, il va recalculer son chemin en simulant un monstre sur la première case de son trajet initial.

3 Questions

3.1

Dans Config.java, **mettre SUPERGOMME, TUNNELHORI et TUNNELVERT à 0 ainsi que MONSTERCOST à 1**, afin de jouer au jeu basique.

3.2

Mettre SUPERGOMME à 0, MONSTERCOST à 1 ainsi que TUNNELHORI et TUNNELVERT au dessus de 0.

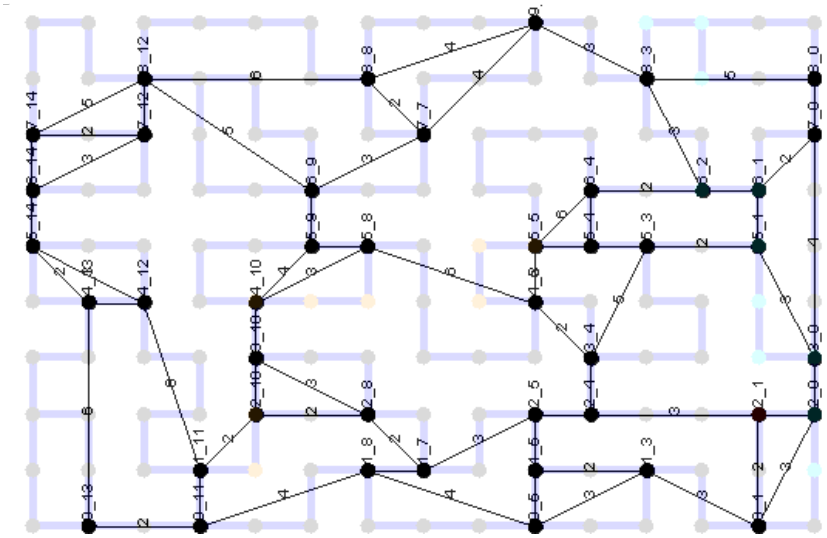
Le jeu ne gère pas les tunnels quelconques mais seulement ceux d'un bord vers le bord opposé. Pour gerer les tunnels quelconques, il aurait fallu que je rajoute une orientation «Tunnel» afin de gérer ce type de déplacement particulier.

3.3

Executer Benchmark.java. N'ayant pas un environement linux sous la main, je n'ai put avoir de resultat utilisable à l'aide de la classe Benchmark. Cependant en théorie l'algorithme A* est plus rapide qu'un algorithme exact, c'est pourquoi on l'utilise dans les jeux car le chemin doit être souvent recalculé et que l'on as pas besoin d'un resultat parfait mais d'un très bon resultat. De plus le benchmark permet quand même de voir que sur les graphes que j'utilise pour le labyrinthe la longueur des chemins sont identiques dans 99% des cas (je n'ai pas put tester sur des graphes plus grand que 100×100 à cause de la vetusté de ma machine)

3.4

La réponse à cette question se trouve dans les paragraphes précedents ou je décrit les heuristiques des differents comportements des monstres. Cependant je vais completer en expliquant deux heuristiques que je n'ai pas eu le temps d'implementer et qui peuvent reduire la somme de calcul necessaire. Lors de la conception du labyrinthe en Tp de Graphe, j'ai à un moment utilisé un «graphe simplifié» qui contient très peu de chaîne.(les lignes 119 à 165 de *PacmanGame.java* construisent et affichent ce graphe)



Le graphe simplifié et pondéré est en noir, et l'on voit par transparence le graphe d'origine en bleu. Le poid de chaque arrete étant la taille du plus petit chemins reliant les deux somments. Donc l'heuristique utilisé avec ce graphe serait la distance de manhattan et le poid de chaque arrete comme cout.

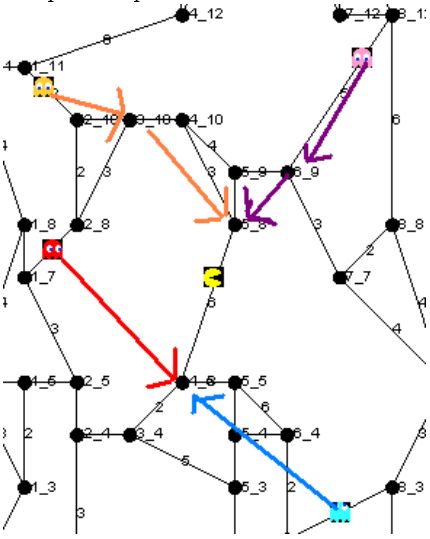
L'autre heuristique à laquelle j'ai pensé, c'est de calculé les chemins à l'aide de l'algorithme de dijkstra en partant de pacman et s'arretant lorsque les 4 fantomes sont atteint. Je pense que pour un petit graphe c'est une bonne idée (mais qui me demandait de devoir mettre la gestion des monstres en dehors de la classe monstre, donc de revoir pas mal de choses dans le programme), cependant dans un grand graphes si les monstres sont disséminés, un unique dijkstra me semble moins avantageux que plusieur A*, surtout vu la topologie de mes labyrinthes.

3.5

Mettre MONSTERCOST entre min(H,W) et H+W. la valeur MONSTERCOST sert dans le calcul du cout d'un deplacement dans l'heuristique de A*, plus la valeur est forte plus

les monstres considereront que suivre leur copains n'est pas une bonne idée, à l'inverse plus elle est faible et plus ils auront tendances à se mettre à la file indienne. Ce n'est pas vraiment une «collaboration» dans le sens ou ils n'établissent pas une veritable stratégie d'encerclement, ils evitent juste de se géner[1], cependant le fait qu'ils n'aiment pas se suivre les amenent à chercher un chemin different du précedent et donc ils réalisent en general un encerclement sans vraiment l'avoir planifié. C'est l'heuristique d'encerclement que j'ai choisit d'utiliser car elle est très simple à mettre en oeuvre(juste changer l'heuristique de l'algorithme A*), relativement efficace (surtout avec les tunnels), modulable (on pourrait par exemple attribuer une valeur différente pour chaque monstre, e.g. avoir un ou plusieurs«fonce au but» avec une valeur basse et un ou plusieurs«rabatteur» avec des valeurs plus hautes) (Cette solution m'a obligé à rajouter le recalcul du chemin lorsque deux monstres sont sur la même case cf explication de $aStarIsBorn()$)

L'autre stratégie à laquel j'ai pensé, necessite d'utiliser le graphe simplifié afin de determiner plus facilement les sommets propice à un encerclement(voir dessin). comme je n'ai pas eu le temps d'implementer l'utilisation du graphe simplifié, j'en suis resté la pour l'encerclement.



3.6

Il existe des version dynamiques de A*, e.g D* ou D*Lite [wikipedia D*] [Article sur l'algorithme D*Lite] [Article sur l'algorithme LPA*] Elles peuvent être interessante car elles se basent sur la solution precedente pour reduire le nombre de calcul demandé lors de la recherche repeté de chemin, ce qui est bien evident le cas dans un jeux vidéo ou les npcs recalculent constamment leurs routes. Pour le cas de D*Lite, son implementation irait très bien avec le graphe simplifié, en rajoutant deux sommets aux graphes, un sommet representant le monstre qui serait connecté à au plus deux sommets tel que les deux sommet soient les extrémités de la chaine de case ou se trouve le fantome, et un sommet representant pacman pareillement connecté, on a en faisant ainsi toujours le même sommet de depart et d'arrivée, seul le poid des aretes change ce qui nous permet d'appliquer l'algorithme (de ce fait on considère que le sommet monstre et le sommet pacman sont implicitement relié à tout les autres sommets du graphes avec un poid infini sauf pour au plus deux aretes)

3.7

Mettre SUPERGOMME au dessus de 1.