8INF957–Programmation objet avancée
TP2

Sacha Guyot
GUYS07119908

Alan Guivarch
GUIA24119800

Nous avons choisi de traiter les questions 1 et 4. Le code se trouve dans le dossier src, dans Q1 pour la question 1 et dans Q4 pour la question 4 (common ne contient qu'un listener et une interface servant à arrêter les programmes).

Question 1 :

Le thread principal et la fonction *main* se trouvent dans la classe *PigeonMap*. Cette classe contient l'ensemble des pigeons et de la nourriture qui devra être affichée à l'écran.
La fonction *main* initialise l'affichage, lance le nombre de thread de pigeons donné en argument, et lance la boucle de traitement de la nourriture, des pigeons et de l'affichage.
À chaque tour de boucle on vérifie la durée de vie de la nourriture. Si elle est trop vieille, elle est supprimée. On fait ensuite un test de probabilité pour savoir si pendant ce tour les pigeons doivent être effrayés, et enfin on met à jour l'affichage de la carte.
Pour ajouter de la nourriture sur la carte, on a utilisé un MouseListener afin que dès que l'on clic sur la carte, de la nourriture soit ajoutée aux coordonnées de la souris. La fonction d'ajout de nourriture dans *PigeonMap* fait en sorte de réveiller les pigeons s'ils sont endormis.

La nourriture est modélisée par la classe *Food* qui stocke les coordonnées de la nourriture et le temps du système au moment où elle apparaît.

Les pigeons sont modélisés par la classe *Pigeon* qui étend la classe Thread pour que chaque pigeon ait son propre thread.
Un pigeon commence sur une position aléatoire sur la carte. Le thread d'un pigeon exécute une boucle infinie. Au début de cette boucle, le deltatime est mis à jour, puis le pigeon vérifie s'il y a de la nourriture sur la carte. Si non, il s'endort en attendant qu'il y en ait. Si oui, le pigeon se dirige vers la nourriture la plus fraîche grâce à la fonction *moveToFreshestFood()*. Si le pigeon est suffisamment proche de la nourriture après son déplacement, alors il essaye de la manger grâce à la fonction *eatFood(Food food)*. Si il réussit à avoir la nourriture (aucun autre pigeon ne la mange avant lui), son score augmente de 1.

Pour exécuter le programme, il faut donner en argument le nombre de pigeons que l'on souhaite afficher. Le fichier Q1.jar dans le dossier out/artifacts/TP2_Q1_jar permet d'exécuter le programme.

On peut le lancer avec la commande :
`java -jar Q1.jar <nombre de pigeons (int)>`
Exemple : `java -jar Q1.jar 5`

Question 4 :

Le main se trouve dans *Main4*. Il crée un graph connexe aléatoire et la fenêtre et démarre le graph.

La classe *Graph* contient les méthodes *sendMessage(Node a, Node b)* et *sendMessage(Message m)* qui permettent d'envoyer un nouveau message entre a et b et de poster un message existant. *repeatLast()* envoie une copie du dernier message envoyé. *getRandomNode()* et *getNode(int i)* renvoient respectivement un nœud aléatoire du graph et le nœud numéro i.
La fonction run effectue 3 tâches principales :

- Envoyer le nombre de messages spécifié en paramètre du programme (si il y en a un)
- Démarrer les nœuds
- Exécuter la boucle d'affichage

Elle a aussi 2 fonctions auxiliaires : réveiller les nœuds lorsqu'ils ont tous fini le remplissage de leurs tables de routage (si applicable, voir les algorithmes de routage plus bas), et envoyer des messages à chaque tour de boucle si l'utilisateur le demande.

Les nœuds du graph sont rangés dans un *QuadTree* (plus efficace qu'une liste ou un tableau pour trouver les nœuds assez proche d'un nœud donné pour s'y connecter).
Ce *QuadTree* stocke au maximum 100 nœuds par niveau. Si ce nombre est dépassé, les nœuds sont répartis dans le niveau inférieur.

La dernière classe digne d'intérêt de ce projet est *Node*, qui implémente le comportement des nœuds du graph. Chaque nœud fonctionne dans un thread à part.
Sa boucle consiste à attendre la réception d'un message, puis à le traiter selon le principe suivant :

- Si le message est trop vieux (ce qui signifie qu'il est perdu dans un trou noir), il est détruit
- Sinon, le nœud cherche à quel nœud accessible transférer le message pour qu'il s'approche de sa destination. S'il trouve un nœud, il lui envoie le message. Sinon, cela signifie que le message est bloqué et il est détruit. Le choix du nœud auquel transférer le message se fait selon l'un des 7 algorithmes détaillés plus bas (au choix de l'utilisateur).

Si l'utilisateur choisit d'utiliser l'algorithme de routage réseau, le run de Node commence par une séquence d'initialisation des tables de routage avant de lancer la boucle principale.

Lors de l'arrêt du programme, *Graph* affiche le numéro du nœud ayant gagné le plus de points et le taux de perte des messages.

Pendant l'exécution, le programme offre des boutons permettant d'envoyer des messages manuellement d'un nœud à un autre, ou d'envoyer plusieurs messages d'un coup entre des nœuds choisis aléatoirement, ainsi qu'une option pour envoyer des messages en continu et un leaderboard.

Pour exécuter le programme, il faut lancer out/artifacts/TP2_Q4_jar/Q4.jar.

Le programme peut prendre les paramètres suivants (tous les paramètres sont optionnels) :

`[n/nodes <int>]`
Définit le nombre de nœuds du graph.
Valeur par défaut : 10

`[d/distance <double>]`
Définit la distance maximale entre deux nœuds pour qu'ils puissent se connecter.
Valeur par défaut : 1

`[m/messages <int>]`
Le nombre de messages à envoyer automatiquement dès le lancement du programme.
Valeur par défaut : 0

`[s/size <double>]`
La taille du graph (dans la même unité que le paramètre de distance)
Valeur par défaut : racine carrée du nombre de nœuds

`[a/algo <String>]`
L'algorithme à utiliser pour le routage des messages.
Valeur par défaut : pp
Valeurs possibles :

- pp : nœud le plus proche du nœud courant
- ppnr : nœud le plus proche sans demi-tour
- ppu : nœud le plus proche sans boucle (passage unique)
- dd : nœud accessible le plus proche de la destination du message (à vol d'oiseau)
- ddnr : nœud accessible le plus proche de la destination du message (à vol d'oiseau) sans demi-tour
- ddu : nœud accessible le plus proche de la destination du message (à vol d'oiseau) sans boucle (passage unique)
- r : routage réseau (ne pas utiliser avec plus de 2000 nœuds)

`[fd/forcedisplay]`
Si le paramètre est présent, le graph sera affiché même s'il contient plus de 200 nœuds.

Exemples : `java -jar Q4.jar n 250 d 1 m 0 s 5 a ddu fd`
`java -jar Q4.jar n 50 a r`

Attention : les paramètres ne sont pas modifiables pendant l'exécution.