Projet de Programmation

Simulation de termites rassemblant des brindilles. Soutenance la semaine du 2 mai¹.

1 Principe de l'algorithme mis en œuvre par les termites.

La stigmergie est une méthode de communication indirecte dans un environnement émergent auto-organisé, où les individus communiquent entre eux en modifiant leur environnement.

La stigmergie a d'abord été observée dans la nature - Les termites, par exemple, utilisent des phéromones pour construire de grandes et complexes structures de terre à l'aide d'une simple règle décentralisée. Chaque termite ramasse un peu de boue autour de lui, y incorpore des phéromones, et la dépose par terre. Comme les termites sont attirés par l'odeur, ils déposent plus souvent leur paquet de boue là où d'autres ont déjà déposé le leur, ce qui forme des piliers, des arches, des tunnels et des chambres.

Dans ce projet, nous allons simuler un comportement particulièrement simple qui ne nécessite pas de phéromone. La tâche est de rassembler des brindilles éparpillées, pour en faire un seul gros tas. Pour ce faire, un termite itère la suite des quatre étapes suivantes : 1- Une marche aléatoire mène le termite vers une brindille. 2- La brindille est ramassée par le termite. 3- Une deuxième marche aléatoire conduit vers un autre tas de brindilles. 4- La brindille portée est posée.

Les petits tas diminuent alors que les gros ont tendance à augmenter statistiquement. Une fois qu'un seul gros tas est obtenu, il reste en place. L'algorithme est décentralisé et auto-stabilisant. Ce projet est un exemple de simulation de système complexe fait d'un grand nombre d'agents simples. La complexité naît de l'interaction du grand nombre. La compréhension de ce type de système est un défi majeur de l'informatique. La puissance de l'ordinateur moderne est indispensable pour les étudier en autorisant leur simulation.

2 TD Analyse descendante

On cherche à décomposer le problème en sous problèmes plus simples. Il faut analyser le problème pour le préciser en faisant des choix d'implémentation. On fera ces choix de façon à simplifier la programmation finale, tout en permettant la mise en œuvre d'un type abstrait **termite**, raisonnablement étoffé.

2.1 Architecture du Programme principal

- Décomposer le programme principal final en deux phases et préciser l'interaction avec l'utilisateur qui a lieu dans la deuxième.

2.2 Sur l'initialisation

- Qu'est-ce qu'il faut initialiser ?
- Comment programmer où l'on pose des termites et des brindilles ?
- Doit-on définir statiquement un nombre de termites/brindilles donné ?
- Comment procéder pour remplir aléatoirement le terrain de façon simple ?

¹sauf pour les demi groupe A2 et B3 avec Gruau, qui soutiennent la semaine du 18 avril

- A quoi est à peu près égal le nombre de termites final ?
- Une borne sur le nombre de termites est-elle connue statiquement ?
- Doit-on définir un type abstrait **ensemble de termites** ?
- On recommande d'utiliser le moins possible de variables globales pour éviter les effets de bord. Mais, les variables réputées uniques sur l'ensemble de la simulation, peuvent se déclarer globales pour éviter de les passer en paramètre. Quelles sont ces variables ?
- Comment fait un termite pour accéder aux cases voisines ?
- Où sont stockées les coordonnées d'un termite ?
- Ecrire la procédure qui initialise le terrain.

2.3 2me phase : itération en interaction avec l'utilisateur

- Doit-on itérer sur le terrain ? oui, non et pourquoi ?
- Comment afficher l'évolution des termites sur leur terrain, de façon interactive ?
- Comment facilement demander au simulateur d'afficher la configuration suivante du terrain après une passe sur les termites ?
- Comment faire pour demander au simulateur d'afficher successivement 10 passes ?
- Et si on veut connaître l'état du terrain au bout de 1000 passes ?
- écrire le programme principal, en utilisant les réponses aux questions précédentes.

3 TD Spécification de l'algorithme de rassemblement de brindilles.

Le sujet du projet annonce le but de l'algorithme : regrouper des brindilles. Il explicite que les termites itèrent quatre étapes. Il reste un travail important à faire pour préciser cet algorithme.

- Autorise-t-on les termites à marcher sur les brindilles ?
- Deux termites peuvent-ils occuper la même case ?
- Sachant qu'un termite ne voit que la case devant lui, quand peut-il saisir une brindille ?
- Quand un termite peut-il poser une brindille ?
- Comment faire pour la poser à côté d'un tas ?
- Lorsqu'un termite prend (resp. pose) une brindille, comment faire pour qu'il s'éloigne un peu afin de trouver un autre tas avant de la poser (resp. ramasser une autre brindille ?)

4 TD sur les types abstraits.

L'objectif de ce TD est de faire une analyse ascendante. On cherche la liste des fonctionnalités élémentaires que doivent pouvoir remplir les places et les termites. Puis, en les combinant, on programme des primitives plus complexes pour aboutir finalement à *rassemblerBrindille*.

4.1 Type abstrait élémentaire

Ce sont des types abstraits tellement simples, qu'ils ne méritent pas vraiment qu'on les traite comme tels. Cependant, il faut s'efforcer de les programmer proprement. En fait, l'analyse réelle commence logiquement par se poser des questions sur le type concret sous-jacent : qu'est-ce qu'on stocke dans le *struct* ?

- Comment repérer un endroit sur le terrain ?
- Quelle(s) information(s) doit fournir une place du terrain ?
- Quels sont les constructeurs pour une place ?
- Quels sont les deux tests élémentaires que l'on souhaite pouvoir faire sur une place ?
- Quel test peut-on vouloir faire, qui combine ces tests élémentaires ? Le programmer ;
- Quand est-ce qu'on modifie une place ?
- Que doit-on pouvoir faire d'autres avec une place ?
- Que doit appeler cette procédure ?

4.2 Les primitives élémentaires du type abstrait termite.

- Quelles informations doit contenir un termite?
- Quel(s) paramètre(s) passer pour construire un termite ?
- Quand un termite est-il modifié ?
- Quand il avance ou charge/décharge des brindilles, le termite modifie une ou deux places, pourquoi n'y a t-il pas de places passées en Donnée-Résultat dans la signature des primitives correspondantes ?
- Quelles sont les primitives dont on a besoin sur un termite, pour récupérer des infos sur le terrain ?
- Les primitives avancer, charger, decharger peuvent s'écrire à partir de primitives plus élémentaires des types abstraits 'place' et 'termite', le faire.
- L'introduction des procédures placeVide/poserTermite, placerVide/poserBrindille et set-Brindille comme des primitives élémentaires peut être sujet à controverse, pourquoi ?
- Qu'est-ce qu'on veut pouvoir faire d'autre d'élémentaire avec un termite ?

4.3 Les primitives termite programmées avec les primitives élémentaires.

- Quelles sont les deux choses que le termite teste sur le terrain, pour choisir entre ses différents comportements ? Programmer ces primitives.
- L'objectif de la marche aléatoire est de chercher un nouveau tas ; Préciser comment elle doit être effectuée pour remplir cet objectif et en déduire une première programmation de celle-ci.
- Que va-t-il se passer avec cet algorithme simple ?
- Comment éviter cela ?

4.4 L'algorithme de la primitive rassemblerBrindille.

C'est toujours une primitive du type abstrait **termite** ; la seule qui soit réellement un peu étoffée. Ce sera à vous de tâtonner pour la trouver tout seul. Les questions qui suivent servent à préciser comment utiliser et mettre à jour les variables d'état du termite qui peuvent sembler difficiles.

- Comment un termite renverse-t-il son sablier, le met à jour, teste si le temps est écoulé ?
- Pourquoi faut-il une variable d'état booléenne *tournerSurPlace* ?
- Ecrire l'algorithme qui permet aux termites de rassembler les brindilles. Le corrigé NE SERA PAS donné. Les TDs contiennent suffisamment d'informations pour que cette tâche soit largement faisable, de plus, cela justifie le programme lui-même : Il est difficile de voir l'algorithme directement, par contre il est facile de tester différentes façons de faire, d'observer le résultat, puis de corriger graduellement son programme.

4.5 Mise en forme du programme

- On recommande de mettre en début de programme toutes les constantes utilisées, pour pouvoir les voir et les modifier facilement. Quelles sont ces constantes ?
- Comment faut-il faire pour faire apparaître clairement le type abstrait **termite** au niveau du programme C ?
- Que faut-il inclure pour rendre le programme lisible ?

5 TP démarrage.

L'objectif de ce TP est de démarrer le projet. Le but est d'arriver à produire un seul fichier C, avec un *Main* qui fasse l'affichage du terrain initialisé.

- Ecrire les type coord, place et une ébauche minimale du type concret **termite**, avec juste les coordonnées.
- Déclarer les variables globales : terrain de 40 places de coté, termites, nbTermites.
- Programmer une fonction qui prend un flottant p entre 0 et 1, et renvoie un booléen vrai avec une probabilité p .
- Programmer les constructeurs de : coord, place et termite, et leur affichage (un brindille s'affiche par '0', pour l'instant, les termites sont affichés par un 'T')
- Initialiser un terrain de 40x40 cases avec 5 pour cent de brindilles, et 1 pour cent de termites.
- Pour afficher une place, quel mode de passage utiliser, donnée, ou donnée-résultat ?
- A quoi s'apparente la procédure *afficheTerrain* ?
- Ecrire la procédure *afficheTerrain*.
- Faire le programme principal qui initialise le terrain aléatoirement, puis l'affiche.

6 Travail personnel demandé et noté.

Pour vous aider, nous indiquons une suite d'étapes permettant d'avancer progressivement, et aboutissant à un effet testable le jour de la soutenance. D'autre part, deux bugs sont fréquents. Si votre programme plante inopinément, vous avez sans doute tenté d'adresser une place dont les coordonnées sont en dehors du terrain. Si votre programme ne s'arrête jamais, c'est probablement que l'un de vos termites est entré dans une boucle qui ne s'arrête pas. Pour éviter cela, éviter les boucles dans les primitives du termite.

6.1 Affichage de la direction des termites (1,5pt)

Modifier le type concret des termites pour stocker leur direction (entier dans 0 .. 7), initialiser celle-ci aléatoirement, et afficher le termite avec le caractère '/', '\', '-', '|' suivant sa direction. Les points de cette question ne peuvent être donnés que si la question suivante est également traitée, de sorte qu'on puisse vérifier visuellement la cohérence du mouvement avec la direction affichée.

6.2 Mouvement des termites (3 pt)

Bouger les termites aléatoirement, en modifiant leur direction avec une probabilité de 1 sur 10. Ainsi, en moyenne, le termite va tout droit dix fois de suite. Les termites ne peuvent se déplacer vers une case déjà occupée par une brindille, ou un autre termite. Il faut mettre en place la boucle qui itère une passe, puis demande à l'utilisateur une saisie d'un caractère de contrôle avant de continuer (`getcar()`). Le programme termine si l'utilisateur tape '.' sinon il continue une passe. Une simple pression sur return provoque directement une nouvelle passe.

6.3 Contournement des obstacles(1pt)

Observer que les termites restent bloqués longtemps sur les obstacles. En particulier la majorité se retrouvent coincés au bord de la grille. Pour éviter cela, les termites mutent, la nouvelle génération décide systématiquement de tourner à droite si un obstacle survient en face.

6.4 Contrôle de l'exécution (1pt)

On souhaite vérifier l'exécution après un grand nombre d'itérations, pour pouvoir s'assurer directement si le regroupement en un seul tas fonctionne, sans avoir à taper un grand nombre de fois return, pour itérer un grand nombre de passes. Utiliser une variable *nbPasse*, et réaliser *nbPasse* entre chaque affichage. Initialiser d'abord directement cette variable à 10,100,1000, puis donner la possibilité à l'utilisateur de multiplier par deux (resp. diviser par deux) cette variable, s'il saisit le caractère '*' (resp. '/'). Cela permet d'accélérer et de ralentir l'exécution.

6.5 Rassembler les brindilles (5 pt)

Modifier le type concret des termites pour ajouter les variables d'état du termite (sablier, brindille, tourneSurPlace). Implémenter *chargerBrindille* et *dechargerBrindille*. Puis *rassemblerBrindille*, en prenant progressivement en compte ces nouvelles variables d'état.

6.6 Séparation du type abstrait (2pt)

Séparer le code relatif aux termites dans un fichier `termite.c`, en ajoutant également un fichier `termite.h` pour les signatures des primitives. Cela vous oblige peut-être à passer le terrain en paramètre pour les fonctionnalités du termite qui en ont besoin.

6.7 Préserver la liberté du termite (1pt)

Observer, ou tester que les termites peuvent s'enfermer tout seul dans un mur de brindilles, en posant une brindille sur la case par où ils sont arrivés. Modifier *rassemblerBrindille* pour éviter cela. (Cela prend trois lignes de programmes).

6.8 Optionnel (0pt)

Ce qui suit est réellement difficile, et n'est pas du tout demandé. C'est la suite logique, et si certains étudiants aiment bien programmer, et ont le temps, ils peuvent s'y frotter. On observe que le tas ne reste pas forcément connexe, une fois qu'il est formé. La raison est que un termite peut enlever une brindille qui sépare le tas en deux. Modifier le comportement pour éviter cela. Tester la connexité du tas, vérifier qu'il reste connexe, utiliser le temps moyen de mise en tas connexe, optimiser les constantes du modèle (durée du sablier, probabilité de tourner) en fonction des densités en termites et brindilles, pour obtenir une mise en tas la plus rapide possible. Commenter les résultats obtenus (comparer avec des vrais termites ?!).

6.9 Présentation (1,5 pt)

Ces points récompensent la qualité de présentation orale et du code : modularité, lisibilité, l'originalité : gestion des problèmes rencontrés, affichage d'informations intermédiaires, choses traitées en plus.

6.10 Contrôle continu pendant les TPs (4pt)

la note de contrôle continu vaut le projet, sur 16 plus ces 4 points qui évaluent votre présence en TPs, votre activité durant les TPs , et vos résultats aux TP notés.

7 Soutenance

L'évaluation du projet se fait par la remise d'une impression papier du programme, avec les deux noms du binôme, et pendant une soutenance. Tout étudiant absent à la soutenance aura la note 0, même si il a fait le programme, car l'examinateur ne serait pas en mesure d'évaluer le travail fait. Il est interdit de donner le listing de son programme à un autre binôme. Une note inférieure à 8 sanctionne systématiquement deux listings trop proches qui révèlent à l'évidence une recopie.

L'organisation de la soutenance est la suivante :

- Vous vous installez sur une machine, chargez votre programme et vérifiez rapidement que tout fonctionne comme la dernière fois que vous l'avez testé ;
- les deux membres du binôme ont 10 mn pour présenter le résultat de leur travail (organisation du code, visualisation des résultats, discussion sur les problèmes rencontrés...);
- L'examinateur a 5 mn pour questionner, en commençant par le moins fort des deux binômes. Il évalue votre implication réelle individuelle. Un étudiant n'a pas les points sur le code qu'il ne maîtrise pas. Les notes des deux binômes peuvent donc différer.