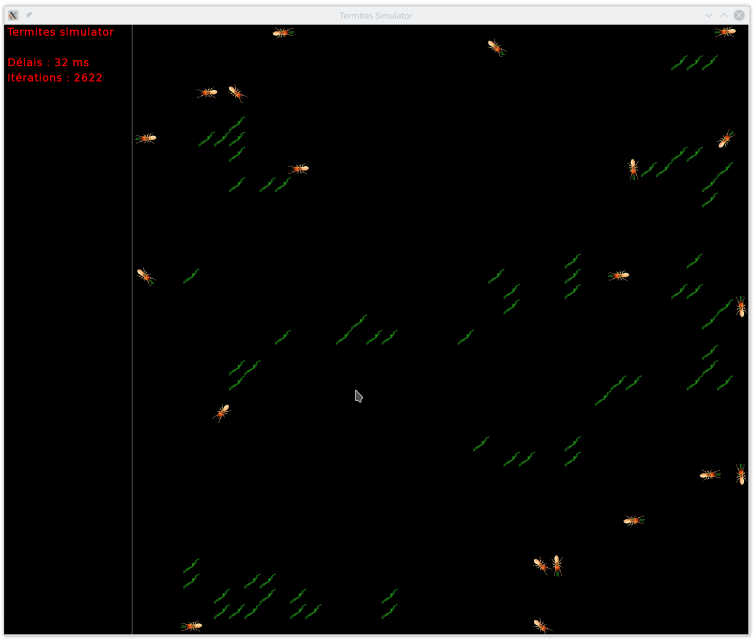
université PARIS-SACLAY

Projet de programmation
Cours de programmation modulaire
—Licence MI/IM - Info 3—

Simulation de colonies de termites



Dans ce projet, nous allons simuler des *colonies de termites* en compétition pour créer un nid en rapportant des brindilles. Malgré des règles de fonctionnement très simples, et un comportement aléatoire, les termites réussissent à rassembler des brindilles pour construire leur nid.

La *stigmergie* est une méthode de communication indirecte dans un environnement émergent auto-organisé, où les individus communiquent entre eux en modifiant leur environnement. D'après la page Wikipedia :

> En biologie, la *stigmergie* est un mécanisme de coordination indirecte entre les agents. Le principe est que la trace laissée dans l'environnement par l'action initiale stimule une action suivante, par le même agent ou un agent différent. De cette façon, les actions successives ont tendance à se renforcer et conduisent ainsi à l'émergence spontanée d'une activité cohérente, apparemment systématique.

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 :

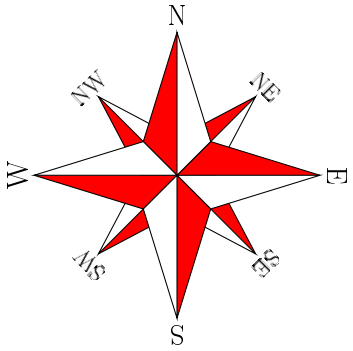
1. Une marche aléatoire mène le termite vers une brindille.
2. La brindille est ramassée par le termite.
3. Une deuxième marche aléatoire conduit vers un autre tas de brindilles.
4. La brindille portée est posée.

Statistiquement, les petits tas diminuent alors que les gros ont tendance à augmenter. 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 permettant leur simulation.

Comportement des termites (partie 1)

Le terrain où évoluent les termites est une grille, implémentée par un tableau à deux dimensions de cases, sur laquelle on pose aléatoirement un certain nombre de termites et de brindilles. On n'autorisera pas les termites à marcher sur les brindilles, ni deux termites à partager une même case, ce qui facilitera l'affichage de la grille.

Un termite ne voit que la case située devant lui mais peut modifier sa direction pour s'orienter vers les 8 points cardinaux (nord-ouest, nord, nord-est, est, sud-est, sud, sud-ouest, ouest). Il ne peut saisir une brindille que si elle se trouve en face de lui, et ne pourra la reposer que sur une case vide en face de lui :



Les termites se déplacent "en même temps" (on n'attend pas qu'un termite ait trouvé une brindille pour déplacer les autres termites). Cependant, comme il n'est pas possible de traiter tous les termites exactement en même temps, on procède étape par étape : on fait faire à chaque termite une étape élémentaire, puis on recommence.

Le comportement des termites est contrôlé par deux constantes.

```
const float probaTourner = 0.1;  // 10%
const int dureeSablier = 6;
```

Les valeurs données sont indicatives. L'idée du projet est de les modifier pour voir comment elles influent sur le comportement.

Voici les règles concernant le comportement des termites :

— Lors du déplacement aléatoire, si la case devant est libre, le termite a `probaTourner` de chance de tourner dans une direction au hasard, sinon il avance tout droit.

— Quand il a ramassé une brindille, il doit attendre un certain nombre de déplacements (`dureeSablier`= 6 dans l'exemple ci-dessus) avant de pouvoir reposer la brindille.

— Après s'être déplacé au moins `dureeSablier` fois, dès qu'il trouve devant lui une autre brindille, il tourne sur place jusqu'à avoir trouvé une case libre et pose sa brindille dans cette case.

  Le termite ne tourne que d'un 8e de tour à chaque étape de simulation ; il est donc possible qu'il tourne sur place pendant un moment avant de trouver une case où poser la brindille. Il est également possible qu'il n'y ait plus aucune case libre car un autre termite l'a enfermé. Dans ce cas, il va tourner sur place jusqu'à ce qu'un autre termite le libère.

  Il est aussi possible qu'en posant sa brindille le termite s'enferme. Dans une amélioration possible de l'algorithme, le termite ne pose la brindille que si il ne s'enferme pas en le faisant, c'est-à-dire s'il existe une autre case voisine libre.

— Il repart ensuite à la recherche d'une autre brindille, mais il ne ramassera pas de nouvelle brindille avant `dureeSablier` nouveaux déplacements. Ainsi, il ne va pas reprendre la brindille qu'il vient juste de poser.

Remarque : On fera attention à utiliser les constantes ci-dessus, et pas des nombres, pour compter le nombre minimal de déplacements et la probabilité de tourner, de manière à pouvoir les changer facilement et voir si cela modifie le comportement.

Gestion des Nids (partie 2)

Cette partie est destinée à ceux qui veulent faire un bon projet (pour avoir une note supérieure à 15). Il ne faut s'y attaquer que si l'on a déjà fait bien fonctionner le déplacement de base d'une seule colonie de termites.

Après avoir fait une simulation où des termites se déplacent au hasard et rassemblent les brindilles, on veut tester si le fait de varier les paramètres de comportement peut rendre les termites plus efficaces. Pour ceci, on va faire plusieurs colonies de termites, chacune avec des paramètres différents. Pour chaque colonie, on choisira une case de la grille qui sera le nid de la colonie. Pour simplifier, on considère que la case du nid est inaccessible.

Au début du jeu, les brindilles posées sur la grille sont «neutres». Quand un termite cherche où poser une brindille sur la grille, il va poser sa brindille dans les trois cas suivants (après avoir attendu `dureeSablier` étapes) :

1. il rencontre une autre brindille «neutre» : dans ce cas il pose sa brindille normalement ;
2. il rencontre son nid : dans ce cas il pose sa brindille, mais avant de la poser, il dépose une odeur dessus pour la marquer comme appartenant à son nid. Un termite ne ramassera jamais une brindille appartenant à son nid.
3. il rencontre une brindille appartenant à son nid. Comme dans le cas précédent, il marque alors la brindille pour agrandir son nid et la dépose.

Comme annoncé, on va changer aussi la règle de ramassage des brindilles : un termite ne ramassera que les brindilles «neutres» ou bien celle du nid adverse (à discuter).

On pourra alors, à chaque tour, parcourir la grille et compter combien de brindilles ont été marquées pas les termites de chaque nid. Le nid ayant le plus de brindilles marquées est le plus compétitif. Une variante où le jeu s'arrête est que les termites refusent aussi de ramasser les brindilles de l'équipe adverse. Dans ce cas, au bout d'un certain temps toutes les brindilles auront été marquées et il n'y aura plus de brindille à ramasser. On peut alors arrêter le jeu et voir quel est le nid gagnant.

1 Simulation

La simulation consiste essentiellement à initialiser la grille en plaçant les termites et les brindilles (et les nids), puis à itérer un certain nombre de fois le déplacement des termites.

Initialisation de la grille

On initialise la grille de manière aléatoire : on place d'abord les nids (s'il y en a), puis pour chaque case non occupée par un nid on place une brindille avec la probabilité

```
const float densiteBrindille=0.05;
```

Ensuite pour chaque nid, on crée

```
const int nbTermites=20;
```

termites (stockés dans un vecteur) que l'on place au hasard dans la grille sur une case vide.

Déplacement des termites

Une étape de simulation consiste à déplacer une fois chaque termite, selon les règles de comportement décrites plus haut. Pour ceux qui cherchent la difficulté, on peut les déplacer dans un ordre aléatoire, sans jamais déplacer deux fois le même termite. On conseille de ne faire cette amélioration que si le reste du projet marche déjà parfaitement.

Affichage de la direction des termites

L'affichage pourra se faire dans le terminal en affichant les brindilles par un caractère `'*'`, les nids par un `'\#'` et les termites avec le caractère `'/'`, `'\'` , `'-'` , `|`* suivant la direction.

Vous devriez avoir un affichage ressemblant à la figure 1.

Dans un deuxième temps, pour que l'on puisse voir quels termites portent une brindille ainsi que l'équipe d'un nid ou d'un termite, on peut mettre de la couleur dans le terminal grâce aux *séquences d'échappement ANSI*. Ce sont des chaînes de caractères qui ne sont pas affichées à l'écran mais qui sont comprises par le terminal comme une demande de changement de configuration de l'affichage. Par exemple, la ligne

```
cout << "\033[31;1m" << "Bonjour" << "\033[0m" << endl;
```

affiche le texte en gras et en rouge. Une séquence commence par `\033[` suivi d'une suite de codes numériques séparés par des virgules, et se termine par le caractère `m`. Dans l'exemple ci-dessus, le

FIGURE 1 – Affichage simple d'une grille sans nid ni équipe

code `31` bascule en rouge et le code `1` en gras. La séquence finale `\033[0m` revient en mode normal grâce au code `0`

Voici une liste non exhaustive de codes :

0	normal	1	gras
2	pâle	3	italique
4	souligné	5	clignotant
7	video inverse	9	barré
30–37	change la couleur de texte	40–47	change la couleur de fond

Les couleurs peuvent dépendre de la configuration de votre terminal et sont par défaut noir, rouge, vert, jaune, bleu, violet, cyan, gris. On peut trouver plus d'information sur Wikipedia (`https://en.wikipedia.org/wiki/ANSI_escape_code`).

Affichage graphique

Si vous avez le temps, vous pouvez faire un affichage graphique, par exemple avec la bibliothèque `SFML`. N'y passez pas trop de temps, surtout si le reste de la simulation n'est pas terminé. Concernant la bibliothèque SFML, vous trouverez des informations de démarrage dans votre cours de premier semestre :

`https://nicolas.thiery.name/Enseignement/Info111/Assignments/Semaine10/index.html#exercice-premiers-graphiques-avec-sfml`

Pour compiler avec la SFML, il faut rajouter les trois options

```
-lsfml-system -lsfml-window -lsfml-graphics
```

au moment de l'édition des liens (variable `LDFLAGS` du `Makefile` proposé en cours).

Pour aller plus loin

Le barême prévoit 2 points (voir plus) pour récompenser les initiatives personnelles, l'originalité, ou les caractéristiques qui distinguent le programme des autres. Voici des suggestions :

— affichage graphique ;

— comportement des termites amélioré (les termites peuvent se donner des brindilles de l'un à l'autre) ;

— grille plus compliquée, tunnels qui passent directement d'une case à l'autre...

— combat entre les termites avec naissances et morts des termites.

2 Implémentation de la simulation

Méthode de développement

Vous devez réaliser le projet en utilisant les outils et méthodes vus en cours de programmation modulaire :

— programmation objet et classes ;

— tests rigoureux des fonctionnalités élémentaires ;

— encapsulation des structures de données pour assurer leur cohérence.

Pour vous aider, il y aura une séance d'amphi et un TP encadrés, puis une séance de suivi en TP.

Organisation du travail

Afin de vous aider à réaliser le projet, nous vous donnons une analyse de la suite d'actions à mettre en œuvre pour implémenter la simulation.

Après l'écriture de chaque fonction, il faut vérifier soigneusement qu'elle fonctionne bien. Soit (quand c'est possible) dans un cas de test de `doctest`, sinon avec une autre procédure de tests avec un affichage, que l'on **conservera**. Dans ce second cas, on essayera également de conserver dans une procédure le code qui permet de reproduire ce test, sans le lancer automatiquement.

Les quatre premiers points ci-dessous seront travaillés en amphi et en TP (voir sujets correspondants).

1. Spécifier les **types abstraits** suivants :
   — `Coord` pour contenir les coordonnées d'un point de la grille ;
   — `Direction` pour contenir une direction ;
   — `Termite` pour coder un termite ;
   — `Grille` pour contenir l'état de la grille à un moment donné.

   Pour les termites, on peut simplement les stocker dans un vecteur.

   Attention ! Si vous essayez de stocker les termites dans un `array`, le compilateur va refuser de créer le tableau car le type `Termite` n'a pas de constructeur par défaut : le compilateur ne sait donc pas comment initialiser les éléments du tableau.

2. Décider des types concrets associés.

   Aide 1 : un termite doit être associé à ses coordonnées dans la grille. Pour cela, utiliser le type concret `coord` défini dans le TP.

   Aide 2 : une case de la grille peut soit être vide, soit contenir un termite. On identifiera un termite par sa position dans le vecteur des termites et on utilisera `Vide = -1` pour indiquer qu'une case est vide.

3. **Initialiser une grille** 20×20 avec le contenu de chaque case choisi aléatoirement.

4. Afficher le contenu de cette grille dans le terminal pour vérifier que l'initialisation a correctement été effectuée.

5. Écrire une fonction qui déplace aléatoirement les termites (sans prendre ni poser de brindilles) : 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 pas 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. 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, on modifie les termites : la nouvelle version décide systématiquement de tourner (au hasard) si un obstacle survient en face.

7. On souhaite vérifier l'exécution après un grand nombre d'itérations, afin de s'assurer directement que 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.

8. Coder le comportement complet des termites : modifier le type concret des termites pour ajouter les variables d'état du termite (sablier, brindille, tourneSurPlace). Implémenter `chargerBrindille` et `dechargerBrindille`, puis l'algorithme principal `rassemblerBrindille`, en prenant progressivement en compte ces nouvelles variables d'état.

9. Observer qu'un termite peut s'enfermer tout seul dans un mur de brindilles, s'il pose une brindille sur la case par où il est arrivé. Modifier `rassemblerBrindille` pour éviter cela (cela prend trois lignes de programme).

10. Coder les différents nids. Ceci demande de modifier la structure de données `Termite` en changeant le type de certains attributs ou en ajoutant des attributs.

Intégrité de la modélisation

L'état de la simulation est stocké dans plusieurs structures de données : les tableaux des termites (il peut y en avoir plusieurs s'il y a plusieurs colonies) et la grille. Ces structures de données sont *redondantes*, c'est-à-dire que *la même information est stockée plusieurs fois*. C'est une manière de programmer dangereuse, car en cas d'erreur de programmation, les deux informations peuvent devenir incohérentes. Voici les redondances :

— **Indice de termite :** chaque termite enregistre son indice dans le tableau. On pourrait avoir stocké par erreur un termite à un autre indice.

De plus, chaque termite enregistre ses coordonnées dans la grille qui elle-même sait quel termite est dans une case donnée.

— **Cohérence termite-grille :** on peut donc avoir un termite qui «pense» être dans une case qui est vide ou qui contient en fait une brindille.

— **Cohérence grille-termite :** ou bien une case qui contient un termite qui n'existe pas ou qui «pense» être dans une autre case.

Nous conseillons d'écrire une procédure qui teste que tout est bien cohérent. En cas de problème, cette procédure doit lever une exception qui décrit le problème.

Cette procédure pourra être appelée après chaque mouvement de l'ensemble des termites pour vérifier la cohérence de la simulation.

Note : tous ces problèmes viennent de l'utilisation des indices et des coordonnées. En général, on évite ces redondances en utilisant des méthodes de programmation plus avancées comme les références ou les pointeurs.

3 Organisation et évaluation du projet

Deux séances de TP (l'une dirigée, l'autre libre) porteront sur le projet. N'hésitez pas à poser des questions à vos encadrants de TP pour régler les problèmes que vous ne parvenez pas à surmonter. Il est indispensable de commencer à travailler sur le projet dès maintenant.

L'évaluation du projet se fera au travers d'une soutenance orale. L'organisation de la soutenance est la suivante :

— Avant votre horaire de passage, vous vous installez sur une machine, chargez votre programme et vérifiez rapidement que tout fonctionne comme la dernière fois que vous l'avez testé (il est de votre responsabilité de vous être assuré avant la soutenance que votre projet fonctionne sur la machine que vous utilisez pour la soutenance) ;

— les deux membres du binôme ont 3 minutes pour présenter ensemble le résultat de leur travail (organisation du code, visualisation des résultats, discussion sur les problèmes rencontrés...) ;

— ensuite l'examinateur a 7 minutes pour juger de votre maîtrise de la programmation. À partir de ce moment, vous serez considérés individuellement, et un effort important sera fait pour mettre en avant clairement votre implication dans le projet. Un étudiant ne peut prétendre être noté sur un code qu'il ne réussit pas à expliquer.

— Un étudiant absent à la soutenance aura zéro sur vingt.

Le principal critère d'évaluation est le bon fonctionnement de votre projet.

ATTENTION : Il vaut mieux avoir écrit moins de code s'il se comporte correctement, que toutes les fonctionnalités demandées où rien ne marche vraiment !

En particulier, rappelez-vous que «le mieux est l'ennemi du bien». Si vous souhaitez améliorer votre projet, soumettez-le (*submit* en anglais) dans Gitlab de manière à pouvoir revenir en arrière si vous n'arrivez pas à faire marcher la suite. Git retient toutes les versions que vous avez soumises. En cas de besoin, n'hésitez pas à demander sur le forum comment revenir en arrière dans l'historique.

Nous prendrons également en compte :

— Qualité de présentation...

— Types abstraits (tous les attributs sont privés) : pertinence, choix alternatifs, utilisation dans le code...

— Qualité du code : modularité, lisibilité, pertinence des commentaires...

— La qualité et la pertinence des tests...

— Originalité : gestion de problèmes rencontrés, affichage d'informations intermédiaires...

Attention ! Les critères ci-dessus ne seront pas évalués en regardant seulement le code, mais également par la manière dont vous répondez aux questions lors de la soutenance. Il est autorisé de se faire aider à comprendre ou débugguer certains points, mais il faut avoir *écrit soi-même, compris et être capable d'expliquer le code que l'on présente.* La note des deux membres du binôme est individuelle et dépend de leurs réponses aux questions et de leur implication dans le projet.