SAE 1.02
Sujet : fourmis et intelligence collective

1 Objectifs et pré-requis

L'objectif de ce projet est de simuler l'intelligence collective d'une colonie de fourmis en quête de nourriture.

Un peu de lecture saine pour commencer...

Lire l'article **« L'intelligence collective »** de Fabrice Rossi publié dans le GNU/Linux Magazine n°51 de juin 2003 et disponible à l'adresse http://apiacoa.org/publications/2003/rossi2003swarn-intelligence.pdf ou en papier sur demande auprès des enseignants.

Dans la suite de ce sujet de projet, nous désignerons ce document par « l'article ». Pour un programmeur en C moyen à confirmer, la lecture de l'article est suffisante pour coder toute cette simulation.

2 Guide pour la réalisation du projet

Algorithme

Vous devez construire votre fonction principale au fur et à mesure pour que là où les fourmis fassent le travail...

1. **Le monde :** on représente l'environnement explorable par une grille de `N×N` cases aux bords infranchissables. Pour commencer, on peut utiliser le petit exemple de la page 3 de l'article, c'est-à-dire un tableau de 25×25.

```c
// Taille d'un côté du monde
#define N 25
// La grille explorable
int monde[N][N] = {0};
```

   Les cases sont désignées par leurs coordonnées `(ligne, colonne)`. La case de coordonnées `(0,0)` est celle en haut à gauche (Nord-Ouest). Les dernières cases à droite (complètement à l'Est) ont comme coordonnées de colonne `N-1` (on commence à numéroter à partie de 0) ; les dernières cases complètement au Sud ont `N-1` en coordonnée de ligne.

   Les entiers dans ce tableau auront pour signification, par exemple :

   – **0 :** case vide (ni nourriture, ni obstacle), une fourmi peut se positionner sur cette case ;
   – $>$ **0 et** $\leqslant$ **40 :** présence de nourriture. Les cases de nourriture peuvent contenir jusqu'à 40 unités de nourriture et sont regroupées par tas ;
   – **-1** : obstacle infranchissable.

   Il est intéressant de déclarer des constantes globales, afin de ne plus avoir à réfléchir sur certaines de ces valeurs :

```c
#define VIDE 0  //case vide (ni nourriture, ni obstacle)
#define OBSTACLE -1
#define FOOD_MAX 40 // FOOD, c'est plus court que NOURRITURE
```

(a) En globale, déclarer le tableau du `monde`. Le reste du code est fait dans le `main()` ou des fonctions. La déclaration du tableau en global simplifiera l'utilisation des fonctions.

(b) Remplir tous les bords de `monde` avec des `OBSTACLE`. Cette partie doit fonctionner pour n'importe quelle taille de tableau.

(c) Créer une zone de nourriture à la main.

(d) Programmer une fonction permettant d'afficher cette grille dans le terminal (par exemple : `void printMonde()`). Afficher les obstacles et les zones de nourriture uniquement. Bien entendu, on n'affichera pas les entiers, mais à la place, des caractères (par exemple, `'*'` pour un obstacle).

2. **La fourmilière :** dans la grille, une des cases sera la fourmillère. La position de la fourmilière pourra être représentée par deux constantes globales `FOURMILIERE_L` et `FOURMILIERE_C` pour la ligne et la colonne de la grille. La nourriture récoltée sera représentée par une simple variable globale.

```c
#define FOURMILIERE_C 12
#define FOURMILIERE_L 12

int score = 0; //nourriture récoltée
```

(a) Programmer la fourmilière comme-dessus.

(b) Modifier la fonction `printMonde()` pour quelle affiche la fourmilière.

3. **Les fourmis :** pour le moment, on utilisera le modèle le plus simple d'une fourmi, décrit dans la section 2 de l'article. Les fourmis étant contrôlées par plusieurs variables (position, mode recherche ou retour àla fourmilière…), il est intéressant d'utiliser une variable structurée (chapitre 9 du cours). Cette structure contiendra :
   - deux entiers pour ses coordonnées sur la grille : généralement `i` ou `l` pour la ligne et `j` ou `c` pour la colonne. Sa position de départ est la fourmilière ;
   - un entier `orientation` pour son orientation (section 2.2 de l'article). On pourra utilisera comme représentation :
     - orientation Nord : 0,
     - orientation Nord-Est : 1,
     - …
     - orientation Nord-Ouest : 7 ;
   - un entier `rotation` de valeur +1 ou -1 suivant si la fourmi préfère tourner à droite ou à gauche lorsqu'elle est confrontée à un obstacle ;
   - un entier `mode` pour le mode dans lequel elle se trouve : 0 pour recherche de nourriture et 1 pour le retour à la fourmillière à la fourmilière.

```c
//des constante peuvent être utiles
#define MODE_RECHERCHE 0
#define MODE_RETOUR 1
/* ... */
typedef struct _fourmi fourmi;
struct _fourmi
{
   int i, j, orientation, rotation, mode;
};
```

(a) Programmer le code minimum pour gérer des nombres aléatoires et créer une fonction `int randn(int a, int b)` qui retourne un nombre aléatoire entier entre `a` et `b` (voir pages 113 et 114 du cours).

(b) Programmer la structure de la fourmi puis créez une `fourmi` en global et initialisez ses données dans le `main()` :
- `i` et `j` sur la fourmilière,
- `mode` recherche de nourriture,
- `orientation` et `rotation` aléatoirement.

(c) Modifier la fonction `printMonde()` pour afficher aussi la fourmi. On pourrait s'intéresser au caractère unicode `0x1F41C`, mais ce caractère n'est que dans un seul sens. Les fourmis ayant une orientation, il est assez simple et surtout très intéressant d'utiliser des caractères « flèches ». Par exemple : https://en.wikipedia.org/wiki/Arrows_(Unicode_block).

4. **Faire bouger la fourmi :** programmer une fonction qui fait avancer la fourmi d'une case par rapport son orientation actuelle si elle n'est pas face à un obstacle.
5. **Oups, un osbtacle :** programmer une fonction qui fait tourner la fourmi d'un cran dans son sens préféré si elle est face à un obstacle. Elle peut être toujours face à un obstacle à la sortie de la fonction et il faudra peut-être attendre plusieurs pas de temps avant qu'elle puisse avancer. En combinant avec la fonction précédente, à chaque pas de temps, soit la fourmi avance d'un pas, soit elle tourne d'un cran pour éviter un obstacle.
6. **La fourmi n'avance pas tout droit :** programmer une fonction qui, en mode « recherche » pour le moment, modifie l'orientation de la fourmi aléatoirement en fonction d'une distribution. Voir l'article et l'annexe de ce document.
7. **Trouver de la nourriture :** programmer une fonction qui teste si la fourmi est sur une case de nourriture. En monde « recherche », elle passera alors en mode « retour » (et dans ce mode, elle transporte une unité de nourriture).
8. **Ramenez sa nourriture :**
   (a) Modifier la fonction de la question **6.** pour inclure la distribution dans le cas « retour ». On rappelle que dans le cas « retour », la fourmi s'oriente d'abord vers la fourmilière (car elle connaît la position de la fourmilière) puis sa position varie avec la distribution.
   (b) Programmer une fonction qui détecte que la fourmi est sur la fourmilière. En mode « retour », elle passera alors en mode « recherche ».
9. **Plus de fourmis :** à partir du travail réalisé et qui sera modifier, gérer l'ensemble de la colonie de fourmis.
10. **Des fourmis plus évoluées :** programmer la capacité des fourmis à produire et détecter des phéromones. On s'inspirera de la section 4 de l'article. Modifier toutes les fonctionnalités en conséquence.

2.1 Pour aller plus loin

– Pour obtenir des cartes plus complexes et plus grandes, réfléchir et programmer des fonctionnalités pour importer un fichier (tableau CSV réalisé à partir d'un logiciel de tableur, image bitmap...) et qui le convertit dans le format de notre tableau d'environnement.

– Programmer un affichage textuel amélioré avec la bibliothèque ncurses ou plus compliqué, graphique (version complexe la plus aboutie).

Annexe : gérer un distribution de probabilité en programmation

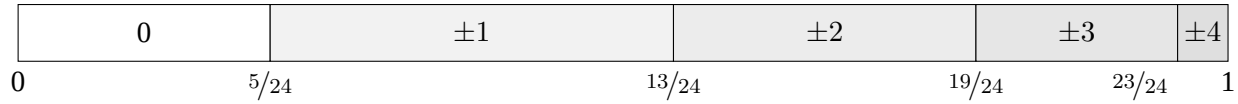
Soit le tableau de probabilité suivant pour l'exemple (Distribution 1 page 7 de l'article)

Rotation	0	± 1	± 2	± 3	± 4
Distribution 1	5/24	8/24	6/24	4/24	1/24

Cela signifie que la fourmi a 5 chances sur 24 d'aller tout droit, 8 chances sur 24 d'effectuer une rotation d'un cran dans son sens préféré… Une des méthodes pour tirer aléatoirement un choix en fonction de ce tableau de probabilté est d'utiliser la fonction de répartition ou tableau des probabilités cumulées :

Rotation	0	± 1	± 2	± 3	± 4
Répartition 1	$5/24$	$13/24$	$19/24$	$23/24$	$24/24 = 1$

Graphiquement, on aurait la représentation ci-dessous représentation, chaque intervalle à la longueur de la probabilité associée. Il suffit alors de tirer un nombre aléatoire réel entre 0 et 1 (tirage uniforme) et de vérifier dans quel intervalle il se trouve pour déterminer quelle orientation choisir.



Algorithme de tirage aléatoire avec distribution de probabibilité

```
REEL DISTRI_CUMUL[4] = tableau {5/24, 13/24, 19/24, 23/24, 1}
REEL n = alea(0,1)
ENTIER i = 0
TANT QUE n>DISTRI_CUMUL[i]
   i=i+1
FIN TANT QUE
RETOURNER i
```