# AlPro
# Tp3 - Ants

## 1 Context

This TP aims to *informatically* models ants behaviour when they are searching for food. We will simulate the individual actions of ants when they search for food in a given environment, in order to see the global behaviour of the anthill. We will build step-by-step a simplified version of these behaviours.

## 2 Init and printing

Ants evolve in an *environment* (a map) which will be represented as a structure `t_world` containing the size of the map, the position of the anthill and a matrix of integer values depending on the content :

- OBSTACLE (constant defined to $-1$) if there is an obstacle
- EMPTY (constant set to 0) if there is no obstacle and no food in this square
- a value between 1 and FOOD_MAX (set to 50) to specify the amont of food at this position.

Ants may evolve freely in this environment, but they start from the anthill and have to go search for food. They can move in every 8 directions (up, down, left, right and diagonals) but they can not step on obstacles. When they find food, they take 1 and go straight back home to drop it off. So they can be in two modes : in search of food or on their way back to the anthill.

We provide you with some utility functions to print the environment with ants, food and obstacles as well as a function to draw random lots with some given examples of environment map. These functions are given in the following files:

- `constants.h` for the constants and types
- `display.c` and `display.h` for the printing functions and their headfile:
  - `init_display` initialize the window for the environment. This function should be called only once before every other and return 1 if the initialization went well.
  - `update_ant` allows us to update the position of an ant and its mode (from "searching for food" to "going back to the anthill" and back).
  - `update_environment` is a function to update the environment i.e. the obstacles and the food
  - `update_pheromones` is a function that updates the amount of pheromones in the environment
  - `display_` the last and mandatory function to be called after every ant and environment updates. This allows us to make a break before the next display and to manage graphical window (redisplay if hidden, manage key pressed, ...). Return 0 if the user pressed ESC and 1 otherwise. (You can also make this simulation go faster or slower using "+" or "-" of your keyboard.)
- `proba.c` and `proba.h` contains functions for drawing random lots and their headfile:

- `nalea(n)` return a random integer between 0 and $n-1$.
- `nalea_weighted` return a random integer between 0 and 7, weighted by a vector of integers.

- `world1.dat`, `world2.dat`, `world3.dat` and `world4.dat` (and a really small `world0.dat`) are map descriptions for environments with:
  - the two first values corresponding to width and height of the environment.
  - the two next values gives the position (abscissa and ordinate) of the anthill.
  - the next values gives the value in each square (line by line). They could be OBSTACLE, EMPTY or an integer value between EMPTY and MAX_FOOD.

The main program implements the following algorithm:

**Algorithm 1** Main algorithm

```
r ← init_display(width, height)
while r=1 do
    ...
    for all ant a do
        update_ant(...)
    end for
    update_environment(...)
    r← display_()
end while
```

In the Makefile we added the library X11 (option –lX11) and the math library (–lm in CFLAGS) that are needed for this program.

## 2.1 Let's start !

**Question 1.** Write the algorithm and then code the function `read_environment` in a new file `world.c`. This function takes a path to a map file as argument and return a structure `t_world` corresponding to the environment described in the file. (Don't forget to write the corresponding header file to develop good coding habits).

**Question 2.** Using the previous algorithm, write the main function in a file `main.c` in order to display a world.

**Question 3.** Create a file `ant.h` containing a structure `t_ant` to describe an ant.

**Question 4.** Write a function `move_ant` to randomly move ant in the environment. For this first move function we will consider that ants could go in every adjacent position (using the `nalea` function) without caring of the obstacle nor the map border.

**Question 5.** Now you can put your first ant in the environment ! Make it move and don't forget to print it after every move.

**Question 6.** Since your ant is starting to feel a bit lonely, you can now add a bunch of ants (start with 100).

**Question 7.** Explain what happens with your program when an ant steps on an obstacle or go out of the map.

**Every time you create a new .c file, don't forget to write the corresponding header file to develop good coding habits.**

# 3 Cool moves: avoid obstacles and stay in the world

As you may have noticed, this is not a very realistic way for ants to move. We will now fix probabilities for each of the 8 directions in order to make them avoid obstacles or the borders of the map. The next move will then be decided by a draw respecting these probabilities. For that we will use a rotation with respect to the previous direction. We arbitrarily fix the following directions between $-4$ and 4:

| | | |
|---|---|---|
| 3 | 2 | 1 |
| $\pm 4$ | $\rightarrow$ | 0 |
| $-3$ | $-2$ | $-1$ |

We use the 0 rotation to keep the same direction, $\pm 4$ to go backward, $\pm 1$ to go diagonally forward, etc. To avoid erratic ant movements, the new direction to follow is chosen according to the old one. To do this, we give a higher probability to the movement that is globally in the same direction as the one the ant had previously. For example, we use the following *weights* (depending on the rotation to be made) for the changes of direction:

| **rotation** | 0 | $\pm 1$ | $\pm 2$ | $\pm 3$ | $\pm 4$ |
|---|---|---|---|---|---|
| **weights** | 12 | 2 | 1 | 1 | 0 |

This table indicates that the ant has 12 chances out of 20 (12+22+21+21) to keep the same direction as in the previous movement, and that it will not be able to go back. For the draw, the function `nalea_weighted` returns a number drawn at random (reduced by a modulo operation between 0 and 7), with a draw not equiprobable but weighted by a given vector.

The *direction* is given by an integer between 0 and 7, corresponding to the modulo 8 rotation, which gives the value of the index of the displacement vectors (`tdx` and `tdy`). For example, in the algorithm of the function `move_ant` below, the rotation $-1$ is represented by the direction 7, such that `tdx[7]`$=+1$ and `tdy[7]`$=-1$.

For each of the moves, we therefore determine the final weighting, using the following criteria:

- first, we draw the move favoring the "straight ahead" direction, thanks to the weighting vector whose values belong to the set {12, 2, 1, 0}.
- if the move is not possible (obstacles or exit from the world), the corresponding weighting is cancelled.

We will use the following arrays for directions in the file `ant.c`:

```
const int tdx[8]={+1,+1,0,-1,-1,-1, 0,+1};
const int tdy[8]={0,+1,+1,+1, 0,-1,-1,-1};
const int straight[8]={12,2,1,1,1,1,1,2};
```

The movement of an ant is thus done in 3 steps:

1. determining the direction to move (examining all directions to set the probability for each, then drawing lots).
2. move in this direction (if it is possible).
3. eventually perform some functions after moving (take the food, etc...).

## 3.1 Work to do

**Question 1.** Write the algorithm and the code of a function (`possible_position`) allowing to know if a given position can be occupied by an ant. We assume that a position can be occupied by several ants.

**Question 2.** Write the code for the function to move an ant (`move_ant`) using the algorithms below. If necessary, the data structure representing an ant can be completed in order to keep the direction (between 0 and 7).

**Question 3.** Test the behavior and also test by changing the probabilities exaggeratedly (turn left, right, etc...) and comment on the tests in your report (if you write one).

Algorithms to use:

**Algorithm 2** move_ant algorithm: t_ant a ▷ random moves

```
d ← nalea(8)                                  ▷ get a random direction
a.x← a.x+tdx[d]                               ▷ change ant position in this direction
a.y← a.y+tdy[d]
```

**Be careful to choose to pass the arguments as pointers if needed.**

**Algorithm 3** move_ant2 algorithm: t_ant a, t_world environment

```
t_weight weightings;

for all i from 0 to 7 do                      ▷ choice of a direction, favouring "straight ahead"
    Integer direction← (i-(a.dir)+8) modulo 8;
    weightings[i]=straight[direction];
end for
for all i from 0 to 7 do                      ▷ zero weighting when movement is not possible
    if not (possible_position(a.x+tdx[i],a.y+tdy[i],environment)) then
        weightings[i]←0;
    end if
end for
a.dir←nalea_weighted(weightings);             ▷ determine the direction by weighted draw
a,x← a.x+tdx[a.dir];
a.y← a.y+tdy[a.dir];
```

# 4 Clever moves: search for food and return to the anthill

As you may have noticed, random search is not the most efficient way to attain food. Now that our ants are able to move correctly, to avoid obstacles, they have to be more clever in the sens that they have to look for food and bring it back to the anthill.

The ants will be able to be in two states, corresponding to two different modes:

- searching for food
- going back to the anthill

To take this into account, we complete the calculation of the weighting as follows.

- For the ant looking for food:
  1. we choose a direction:
     - we favor the straight movement,
     - if the displacement leads to food (and it is not the anthill), the weighting is very high (for example 100000),
     - if the movement is not possible (obstacles or exit from the world), the weighting is null,
  2. we move in this direction
  3. if we get to the food, we take some (quantity taken=1), and we pass in mode "return to the anthill"

- For the ant looking to return to the anthill, the choice of direction is a little different, because the ant can remember the direction of the anthill. We will give a higher probability to the movement that is in the direction of the anthill and we will use the following weights:

| **rotation** | 0 | ±1 | ±2 | ±3 | ±4 |
|---|---|---|---|---|---|
| **weights** | 200 | 32 | 8 | 2 | 0 |

  The movement of the ant returning to the anthill is then :

  1. we choose a direction:
     - we favor the movement in the direction of the anthill, with the weighting$\in \{200, 32, 8, 2, 1\}$ (with declaration `const int dir_anthill[8]={200,32,8,2,1,2,8,32};` in the `ant.c` file),
     - if the movement leads to the anthill, the weighting is very high (for example 100000).
     - if the move is not possible (obstacle or exit from the world), the weighting is null.
  2. we move in this direction
  3. if we get to the anthill, we put down the food (we increment the environment table at this place), we turn around (to go back in the right direction) and we go back to the "search for food" mode.

### 4.1 Work to do

**Question 1.** Complete the algorithm and the code for the function dir_anthill (given below) that calculates in which direction the anthill is located (between 0 and 7) from the position of an ant and the position of the anthill.

**Algorithm 4** direction_anthill algorithm: int ax, int ay, int hx, int hy
▷ (ax,ay) position of the ant, (hx,hy) position of the anthill

```
int direction;
int dx=ax-x;
int dy=ay-y;
float norm=sqrt(dx*dx + dy*dy);
for all i from 0 to 7 do                          ▷ check the best direction to go to the anthill
    if round(dx/norme) == tdx[i] && round(dy/norme) == tdy[i] then
        ...
    end if
end for
return result;
```

**Question 2.** Modify the algorithm and the code of the function to move an ant (or write a new function move_ant3) so that the ants go to get the food, and bring it back to the anthill.

**Question 3.** Comment on the performance, i.e. the number of food units brought back to the anthill per number of iterations. How could it be improved ?

## 5 Pheromones

In order to get closer to the real behavior of ants in their search for food, we will use the same tools as them ! Ants that find food have the particularity to deposit on their way back a hormone (the pheromone) that informs the other ants. The pheromone evaporates naturally with time.

We use in this part a table of type `t_world` which represents the concentration (in percentage: integer from 0 to 100) of pheromone (for each cell of the world). To display the pheromone, we will use the function `evaporation_pheromones(t_world)` (to use just after the display of the environment).

An ant "returning to the anthill" deposits pheromone (10 units) on its current cell and 5 units on the 8 neighboring cells.

A ant "looking for food" will have a higher probability to go in the direction where there is more pheromone. The problem in this case is that we must also favor the direction farthest from the anthill (otherwise we could come back to the anthill!). Let `diff_anthill` be the difference between the direction of the anthill and the direction of the ant, all brought back between $-4$ and 3 (0 indicates that the ant is heading towards the anthill and $-4$ that it is heading in the opposite direction). We then add the weighting: `rate_pheromone*(abs(diff_anthill))` to the previously calculated weights.

The maximum weighting is then : 4(opposite direction to the anthill)$*$100(pheromone saturation) = 400.

## 5.1 Work to do

**Question 1.** Write a function (`evaporation_pheromones`) to model the "evaporation" of the pheromone. This function must be called every 70 rounds, and evaporate 1 unit of pheromone.

**Question 2.** Modify the behavior of the ants in "return to the anthill" mode in order to deposit pheromone. Be careful, the value of the pheromone cannot exceed 100 (saturation phenomenon).

**Question 3.** Modify the behavior of the ants in "search for food" mode in order to privilege a direction with pheromone, opposite to the direction of the anthill.

**Question 4.** Compare the efficiency with the previous question.