AlgPr – Report

TP n°4

— *Ants* —

The goal of this TP is to simulate the emergence of *collective intelligence* with a simple model: an ant. In this model an ant has to bring food (located somewhere in the world) to its anthill. A single ant can only carry one unit of food, so the ants have to work together in order to supply the anthill with food. To achieve this, the ants use *pheromones* to communicate with each other.

Though this may be a simple model, it's the basis of many optimisation algorithms, for the *Travelling Salesman Problem* as an example. We find a path between two points, and we try to minimise its length by nudging it. If done right, the length will decrease until a local minimum is reached.

Our code can be found on *Replit* via the following URL :

https://replit.com/join/zqdgvdnyxa-hugos29.

I. Initializing and displaying the world.

The first step is to read the world file. This file contains the world size, and the coordinates of the food reserves, and of the anthill. We will code the function `LireEnvironnement`, corresponding to the algorithm bellow (Algorithm 1).

Input. fileName: string
Output. world: `t_monde`

world.L ← read(fileName)
world.H ← read(fileName)

world.F_x ← read(fileName)
world.F_y ← read(fileName)

For $i = 0$ **to** world.$L - 1$ **do**
  For $j = 0$ **to** world.$H - 1$ **do**
    world.mat$[j][i]$ ← read(fileName)
  End for
End for

Return world

Algorithm 1. The algorithm for the `LireEnvironnement` function, responsible for reading the world file

Now that we can read the `.dat` file containing the world data, we can code the `main` function—in the `main.cpp` file—, correspond to the algorithm described bellow (Algorithm 2).

Variables
i, return, n_{ants}: three integers,
world: `t_monde`,
ants: a list of `t_fourmi`.

world ← LireEnvironnement(`"monde1.dat"`)

For $i = 0$ **to** n_{ants} **do**
ants[i].x ← world.F_x
ants[i].y ← world.F_y
ants[i].mode ← 1
ants[i].direction ← nalea(8)
End for

return ← InitAffichage(world.L, world.H)

While return $\neq 1$ **do**
For $i = 0$ **to** n_{ants} **do**
MoveAnt(world, ants[i])
UpdateAnt(ants[i].x, ants[i].y, ants[i].mode)

If world.mat[ant.y][ant.x] > 0 and ant.mode $= 1$
and ant.$x \neq$ world.F_x and ant.$y \neq$ world.F_y **then**
world.mat[ant.y][ant.x] ← world.mat[ant.y][ant.x] $- 1$
ant.mode ← 0
Else if ant.$x =$ world.F_x and ant.$y =$ world.F_y and ant.mode $= 0$ **then**
world.mat[ant.y][ant.x] ← world.mat[ant.y][ant.x] $+ 1$
ant.mode ← 1
End if
End for

UpdateEnvironment(world)
return ← Display()
End While

Algorithm 2. The algorithm for the `main` function

To represent an ant, we use the following `t_fourmi` type, as shown in Listing 1.

```
typedef struct {
  int x, y; // position of the ant in the world
  int mode; // 0 -> come back to the anthill;
            // 1 -> search for food
  int direction; // between 0 and 7
} t_fourmi;
```

Listing 1. The C++ code for type `t_fourmi`, representing a single ant

One of the most important part of this simulation is the function that'll move an ant, it will be called `DeplaceFourmi`. This function will be changed multiple times later on. As a first step, we start by making the ant take steps in random direction, without taking account the world border, nor the obstacles (Algorithm 3).

Input. ant: `t_fourmi` [shared]
Output. nothing

Variables. k: integer

$k \leftarrow \mathrm{nalea}(8)$

$\mathrm{ant}.x \leftarrow \mathrm{ant}.x + \mathrm{td}x[k]$
$\mathrm{ant}.y \leftarrow \mathrm{ant}.y + \mathrm{td}y[k]$

Algorithm 3. The algorithm for the `DeplaceFourmi` function, version 1

In the algorithm above, we use the *globally defined* constant arrays $\mathrm{td}x$ and $\mathrm{td}y$:

$$\mathrm{td}x = (+1, +1, 0, -1, -1, -1, 0, +1) \quad \text{and} \quad \mathrm{td}y = (0, +1, +1, +1, 0, -1, -1, -1)$$

As said in the description of Algorithm 3, the algorithm doesn't take the world border, nor the obstacles in account. Thus, the ant may "leave the world", or "walk on an obstacle." The `UpdateAnts` function warns us that one of the simulated did just that.

II. Linear movements with obstacle avoiding.

As described before, an ant can be in a *valid* position if it's not outside of the world, nor on an obstacle. The test performed by `PositionPossible` is the following one:

$$\begin{gathered} \mathrm{ant}.x \in [\![0, \mathrm{world}.L - 1]\!] \\ \text{and} \\ \mathrm{ant}.y \in [\![0, \mathrm{world}.H - 1]\!] \\ \text{and} \\ \mathrm{world.mat}[\mathrm{fourmi}.y][\mathrm{fourmi}.x] \neq \texttt{OBSTACLE} \end{gathered}$$

An algorithm for this function returns the result of this test as a boolean.

Using the given algorithm, we implement the two function `modulo8` and `DeplaceFourmi` (version 2). These two algorithm will make the ant more likely to go straight, instead of rotating at every step.

By changing the weight values for each rotation, we can make the ants more likely to turn left. For example, we can set the weight for rotation $+1$ to 12, whilst setting the weight for rotation 0 to 2. After implementing this change, we can see that the ants tend to move in circles, always turning left. Doing the same procedure to make the ants turn right will yield expected results.

III. Looking for food, and going back to the anthill.

Now that the ants have a more *realistic* scattering procedure, we can work on helping the ants go back to the anthill. Since the anthill has a fixed position $(\mathrm{F}_x, \mathrm{F}_y)$, the ants can more straight to the anthill when they want to bring food. Thus, we implement this in the `DirFourmiliere` function, corresponding to the algorithm bellow (Algorithm 4).

Input. $x, y, \mathrm{F}_x, \mathrm{F}_y$: four integers,
Output. dir: integer

Variables.
$\mathrm{d}x, \mathrm{d}y$: two integers,
norm: a floating-point number,
i: an integer.

$\mathrm{d}x \leftarrow \mathrm{F}_x - x$
$\mathrm{d}y \leftarrow \mathrm{F}_y - y$
$\mathrm{norm} \leftarrow \sqrt{(\mathrm{d}x)^2 + (\mathrm{d}y)^2}$
$\mathrm{d}x \leftarrow \left\lfloor \frac{\mathrm{d}x}{\mathrm{norm}} \right\rceil$
$\mathrm{d}y \leftarrow \left\lfloor \frac{\mathrm{d}y}{\mathrm{norm}} \right\rceil$
For $i = 0$ **to** 7 **do**
  If $\mathrm{d}x = \mathrm{td}x[i]$ and $\mathrm{d}y = \mathrm{td}y[i]$ **then**
    Return i
  End if
End for

Return -1

Algorithm 4. The algorithm for the `DirFourmiliere` function[1]

With this function, we now know which direction the ant has to take to get to the anthill. Thus, we can now update the algorithm behind the `DeplaceFourmi` function, as shown bellow in Algorithm 5.

With this change, one of the two “actions” an ant can do has been optimized: the ant can now reach its nest faster. However, the ant still has to randomly stumble on the food in order to bring it home. We need to make the ants remember *where* the food here, and *how* to get there. That’s what will be implemented next: “*pheromones*.”

IV. Pheromones

In this part, we will model the pheromones used by ants in the real world to communicate with each other. Every 70 simulation steps, the pheromones will evaporate (1% of the pheromone is removed, everywhere on the grid). The ants coming back to the nest with food will disperse pheromones on its current cell, and the neighboring ones. The ants looking for food will “adjust” the weights used to move in order to follow the pheromone trail. Each ant has can diverge from the path, and this allows the length of the path to converge to a local minimum.

We implement this change by updating the `DeplaceFourmi` function, corresponding to Algorithm 6.

Finding food becomes easier for the ants. For example, in the “`monde3.dat`” world, even though the entrance of the “cave” is quite narrow, most of the ants tend to go in this direction to find food, after some time.

[1]In this function, the C++ `round` will be denoted as $\lfloor \cdot \rceil$. This way, `round(x)` will be written as $\lfloor x \rceil$.

Input. ant: `t_fourmi` [shared], world: `t_monde`
Output. nothing

Variables.
$x, y, \mathrm{d}x, \mathrm{d}y$: four integers,
weights: an array of 8 integers,
i, k: two integers.

weights $\leftarrow (0, 0, 0, 0, 0, 0, 0, 0)$

$x \leftarrow$ ant.x
$y \leftarrow$ ant.y

If ant.mode $= 1$ **then** (the ant is looking for food)
 For $i = 0$ **to** 7 **do**
  $\mathrm{d}x \leftarrow \mathrm{td}x[i]$
  $\mathrm{d}y \leftarrow \mathrm{td}y[i]$

  If not PositionPossible($x + \mathrm{d}x$, $y + \mathrm{d}y$, world)
   or ($x + \mathrm{d}x =$ world.F_x and $y + \mathrm{d}y =$ world.F_y) **then**
   weights$[i] \leftarrow 0$
  Else if world.mat$[y + \mathrm{d}y][x + \mathrm{d}x] > 0$ **then**
   weights$[i] \leftarrow 100\,000$
  Else
   weights$[i] \leftarrow w^{\longrightarrow}_{\text{straight}}$[modulo8($i -$ ant.direction)][2]
  End if
 End for
Else (the ant is going back to the anthill)
 For $i = 0$ **to** 7 **do**
  $\mathrm{d}x \leftarrow \mathrm{td}x[i]$
  $\mathrm{d}y \leftarrow \mathrm{td}y[i]$

  If not PositionPossible($x + \mathrm{d}x$, $y + \mathrm{d}y$, world) **then**
   weights$[i] \leftarrow 0$
  Else
   weights$[i] \leftarrow w^{\longleftarrow}_{\text{straight}}$[modulo8($i -$ ant.direction)][2]
  End if
 End for
End if

$k \leftarrow$ nalea_pondere(weights)

ant.$x \leftarrow$ ant.$x + \mathrm{td}x[k]$
ant.$y \leftarrow$ ant.$y + \mathrm{td}y[k]$

ant.direction $\leftarrow k$

Algorithm 5. The algorithm for the `DeplaceFourmi` function, version 3

[2]The w_{straight} arrays correspond to the `p_toutdroit` C++ vector. The $w^{\longrightarrow}_{\text{straight}}$ corresponds to the vector when the ant seeks for food. The $w^{\longleftarrow}_{\text{straight}}$ corresponds to the vector when the ant goes to the nest.

Input.
ant: `t_fourmi` [shared],
world: `t_monde`,
pheromones: `t_matrice` [shared]
Output. nothing

Variables.
$x, y, \mathrm{d}x, \mathrm{d}y$: four integers,
weights: an array of 8 integers,
i, k: two integers.

weights $\leftarrow (0, 0, 0, 0, 0, 0, 0, 0)$

$x \leftarrow \text{ant}.x$
$y \leftarrow \text{ant}.y$

If ant.mode $= 1$ **then** (the ant is looking for food)
  For $i = 0$ **to** 7 **do**
    $\mathrm{d}x \leftarrow \mathrm{td}x[i]$
    $\mathrm{d}y \leftarrow \mathrm{td}y[i]$

    If not PositionPossible($x + \mathrm{d}x$, $y + \mathrm{d}y$, world)
      or ($x + \mathrm{d}x = \text{world.F}_x$ and $y + \mathrm{d}y = \text{world.F}_y$) **then**
        weights$[i] \leftarrow 0$
    Else if world.mat$[y + \mathrm{d}y][x + \mathrm{d}x] > 0$ **then**
        weights$[i] \leftarrow 100\,000$
    Else
        opp $\leftarrow$ DirFourmiliere(world.F$_x$, world.F$_y$, $x + \mathrm{d}x$, $y + \mathrm{d}y$)
        opp $\leftarrow$ opp $- 4 -$ ant.direction

        weights$[i] \leftarrow w_{\text{straight}}^{\rightarrow}$[modulo8($i -$ ant.direction)]
        weights$[i] \leftarrow$ weights$[i]$ + pheromones$[y + \mathrm{d}y][x + \mathrm{d}x] \times |\text{opp}|$
    End if
  End for
Else (the ant is going back to the anthill)
  pheromones$[y][x] \leftarrow$ min(pheromones$[y][x] + 10, 100$)

  For $i = 0$ **to** 7 **do**
    $\mathrm{d}x \leftarrow \mathrm{td}x[i]$
    $\mathrm{d}y \leftarrow \mathrm{td}y[i]$

    If not PositionPossible($x + \mathrm{d}x$, $y + \mathrm{d}y$, world) **then**
        weights$[i] \leftarrow 0$
    Else
        weights$[i] \leftarrow w_{\text{straight}}^{\leftarrow}$[modulo8($i -$ ant.direction)]
    End if

    pheromones$[y + \mathrm{d}y][x + \mathrm{d}x] \leftarrow$ min(pheromones$[y + \mathrm{d}y][x + \mathrm{d}x] + 5, 100$)
  End for
End if

$k \leftarrow$ nalea_pondere(weights)

ant.$x \leftarrow$ ant.x + td$x[k]$
ant.$y \leftarrow$ ant.y + td$y[k]$

ant.direction $\leftarrow k$

Algorithm 6. The algorithm for the `DeplaceFourmi` function, version 4

As stated before, the length of the ants' path to the food will tend towards a *local* minimum. If anther shorter path exists, they may not always find it. This gives an *approximation* of the path. In order to find an *exact* shortest path, we can use other algorithms like Dijkstra's or $\mathbf{A}^{\star}$.

Similar algorithms may be used when an approximation is acceptable, or when the exact solution is too computationally expensive (for example, the Travelling Salesman Problem).