CC2 : Random Path Project

Data Structure and Algorithms 1

L1 Computer Science

The aim of this project is to write a program exhibiting a *mobile* (point or snake) moving at random in a limited finite space called an *arena*. The *arena* on figure 1 is composed of 10×8 cells. The *mobile* (point or snake) can only step on cells.
We will go through this project step by step, beginning with the easier parts and finishing with more difficult tasks. However, this project will not be evaluated in itself. So, feel free not to follow exactly my guidelines and to make your own experiments and your own mistakes. All of your functions must be tested.

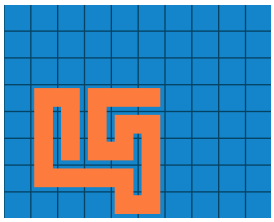

FIGURE 1 – A snake in a 10×8 arena.

1 The `cell` structure

What we call *cells* are the blue squares on figure 1. They define the positions were the *mobile* can step. They are characterized by a position (row and column). The overall idea is to put the mobile on a start cell c and, next, to move to one of its neighbors n, and then to one of the neighbors of n and so on.

1. Create a file `cell.h`, with an include guard.
2. Define the `cell` structure with two integer fields : `row` and `col`. For all the following functions, you will write the prototype in file `cell.h` and the source code in file `cell.c`.
3. Write a function with prototype :

   ```
   struct cell C_new(int row, int col);
   ```

   which for two given integers, `row` and `col`, return a `cell` structure whose `row` and `col` fields are equal to the function's arguments.
4. Write a function with prototype :

   ```
   void C_print(struct cell c, char *label);
   ```

   which prints on the terminal a label and the coordinates of the cell. For example, in the following code :

   ```
   struct cell c = C_new(3,5);
   C_print(c,"my first cell");
   ```

   The first line creates a cell with position (3,5). And the second line outputs :

   ```
   my first cell : (3,5)
   ```
5. Write a function with prototype :

   ```
   void C_printNeighbors(struct cell c);
   ```

   which prints on the terminal the position of the four neighbors of cell `c`. At this stage, we consider that our *arena* has no borders and all cells have 4 neighbors : upper, lower, left and right. For example, the following code :

   ```
   struct cell c = C_new(3,5);
   C_printNeighbors(c);
   ```

   should output :

   ```
   Upper : (2,5)
   Lower : (4,5)
   Left  : (3,4)
   Right : (3,6)
   ```

2 The `cellList` structure

Now the global idea is to choose a start cell and print its position, then pick one of its neibhbors n (print its position) and pick a neighbor of n and so on. But in order to visualize the path of the point, we need to draw something like a curve. But that would need to specify the order in which the cells were explored. For this, we need to make cell lists.

In the context of this project, we will not need to remove any element from the list. And inserting is done only at the end of the list. Furthermore, the number of cells in a path is limited. It cannot grow indefinitely. We can reasonably assume that we will never have a path of more than 1000 cells. What data structure would you use to make these lists ? An array ? or a linked list ?

3 The `cellList` structure

1. Create a file `cellList.h`, with an include guard.
2. Define the `cellList` structure with its size and whatever data structure you chose.
3. Write a function with prototype :

   ```
   struct cellList CL_new();
   ```

   which creates an empty `cellList` structure.
4. Write a function with prototype :

   ```
   struct cellList CL_add(struct cellList cl, struct cell c);
   ```

   which adds cell `c` to the `cl` cell list and returns the resulting list.
5. Write a function with prototype :

   ```
   void CL_print(struct cellList cl, char *label);
   ```

   which prints on the terminal a label and the list of its contents (the position of each cell).
6. test the following functions. The following code :

   ```
   struct cellList cl = CL_new();
   struct cell c = C_new(3,5);
   cl = CL_add(cl,c); // You can put the cell in a variable
   cl = CL_add(cl,C_new(4,5)); // or create it directly like this.
   cl = CL_add(cl,C_new(4,6));
   CL_print(cl,"my first list");
   ```

   should output something like :

   ```
   my first list
   0 : (3,5)
   1 : (4,5)
   2 : (4,6)
   ```
7. Write a function with prototype :

   ```
   struct cell CL_get(struct cellList cl, int ind);
   ```

   which returns the value of cell index `ind` in the `cl` cell list. The first cell in the list has a 0 index. If the cell list contains no cells of index `ind`, an error message should be issued.
8. Write a function with prototype :

   ```
   struct cell CL_random(struct cellList cl);
   ```

   which returns the value of cell picked at random in `cl`. If the cell list is empty, an error message should be issued.
9. Write a function with prototype :

   ```
   struct cellList CL_neighbors(struct cell c);
   ```

   which returns a list composed of the neighbors of cell `c`.The difference between this function and function `C_printNeighbors(st` written in section 1 is that this function does not print anything. It returns the list of neighbors.

10. Test the two previous functions. For any given cell, they should enable you to pick one neighbor at random.
11. Write a function with prototype :

    ```
    struct cellList CL_randomPath(struct cell start, int nb_cells);
    ```

    which produces a list of cells. The first cell in this list is the `start` cell. The following cell must be a neighbor of `start` chosen at random. Then we must have a neighbor of this neighbor and so on, until we have `nb_cells` cells in the list. Thus each pair of consecutive cells in this list must be neighbors.
12. Let us visualize this first array. Download, from the *Moodle* platform, the archive file called *snake project toolkit*. The `cellList.c` file contains two functions `CL_draw` and `CL_animate` which make it possible for you to visualize your cell lists. This requires that all consecutive cells in your list are neighbors.

    — `void CL_draw(struct cellList cl, int nb_rows, int nb_cols, char *ppm_name);`
    Draws the contents of the `cl` cell list in a ppm format image file. For example :

    ```
    CL_draw(cl, 10, 10, "snake");
    ```

    creates an image file called `snake.ppm` representing a grid with 10×10 cells and the the cell list in orange (as on figure 1).

    — `void CL_animate(struct cellList cl, int nb_rows, int nb_cols, char *ppm_name);`
    If `cl` contains `n` cells, this function creates `n` ppm format image files. The first file represents the first cell of `cl`, the second file represents the first and the second cell, and the last cell represents the whole cell list. Looking at this sequence of images enables you to see the construction of the random path.

    ```
    CL_animate(cl, 10, 10, "snake");
    ```

    creates image files called `snake_00.ppm`, `snake_01.ppm`, `snake_02.ppm` and so on.

    Try these function on the list that you made in question 6 of this section. If every thing works as you expect, then you can try it on the result of function `CL_randomPath`.

4 The `arena` structure

As you probably noticed, there are two problems with our implementation : the path may often go outside of the image, and next, it is difficult to see the path since the can come back on positions it already occupied before. In this section, our aim is to limit the region in which the mobile point can walk. To achieve that, we define a finite *arena* with a finite number of rows and columns. The arena in figure 1 has 8 rows and 10 columns.

1. In this limited region, do all cells have 4 neighbors ?
2. Create a new file called *arena.h* with an include guard and, in this file, define the `arena` structure characterized by two integer fields : `nb_rows` and `nb_cols`. In the rest of this section, the prototypes of your functions must be written in `arena.h` and the source codes shoud be written in `arena.c`.
3. Write a function with prototype :

    ```
    struct arena A_new(int nb_rows, int nb_cols);
    ```

    which, for two given integers, `nb_rows` and `nb_cols`, return an `arena` structure whose `nb_rows` and `nb_cols` fields are equal to the function's arguments.
4. Write a function with prototype :

    ```
    int A_isInside(struct cell c, struct arena ar);
    ```

    which returns 1 if cell `c` in inside `ar` and returns 0 in ALL other cases.
5. Write a new version of files `cellList.h` and `cellList.c` in which the `CL_neighbors` function has the following prototype :

    ```
    struct cellList CL_neighbors(struct cell c, struct arena ar);
    ```

    This function returns a list of cells representing the neighbors of cell `c` who must also be inside the arena `ar`.
6. Write a new version of files `cellList.h` and `cellList.c` in which the `CL_randomPath` function has the following prototype :

    ```
    struct cellList CL_randomPath(struct cell start, int nb_cells, struct arena ar);
    ```

    The behaviour of this function is the same as before, except for the fact that the returned list of cells must all belong to the arena `ar`.

5 Do not step on already explored cells

Now, we would like the mobile not to walk over already explored cells. What data structure would you suggest ? Here are two possibilities, but feel free to propose others.

1. During the neighbor search, test whether the neighbors already belong to the cell list. Do not select neighbors who were already explored.
2. Prepare a 2D grid of integers (one integer for each cell) initialized at 0. Each time a cell is explored, set the corresponding value to 1. During the neighbor search, do not select neighbors with a value of 1.

If you have other ideas, feel free to propose. But prepare your arguments : why do you choose one solution or the other ? What is the time or memory complexity of each solution ?

One of the consequences of this new configuration is that the random path may be blocked because all of the neighbors of the mobile point have already been explored (as in figure 1). This is why I suggest to change the specifications and prototype of the `CL_randomPath` function :

```
struct cellList CL_randomPath(struct cell start, struct arena ar);
```

Now this function will stop exploring new cells only when the mobile point is blocked and cannot move any more.

6 Snake ?

Now, we would like to simlate the movement of a snake of length `L` (integer). The principle is that, as soon as the cell list created in the previous questions is longer than `L`, we must get them out of the list and make it possible for the snake to explore those freed cells. What data structure do we need to make such a movement ? Which are the removed cells ? the oldest cells ? or the newest ones ?

Here (`http://dpt-info.u-strasbg.fr/~ahabibi/07_automatique/CC3.html`) is one example of this implementation (in javascript).

7 Avoid blocking situations

Here is the last step. We would like to teach this snake not to get blocked. One possible solution must be carried out during the neighbor search. For each possible neighbor, explore the possible paths stemming from each neighbor. It would be like sending invisible snakes from each neighbor. If one of the snakes is blocked, then we must not choose this neighbor. What data type do we need to carry out this algorithm ? Explain why.

Here (`http://dpt-info.u-strasbg.fr/~ahabibi/10_refont/CC3.html`) is one example of this implementation (in javascript).