CC2 : Random Path Project

Data Structure and Algorithms 1

L1 Computer Science

FIGURE 1 – A bumblebee

The aim of this project is to write a program exhibiting a point moving at random in a limited finite space called an *arena*. The *arena* on figure 2 is composed of 10×10 cells. The point can only step on cells.
You have already worked on a snake project. We will call this one the *bumblebee* project. The difference is that the bumblebee is represented by only one point (yellow cell on figure 2) and that it cannot move on obstacle cells (orange cells).

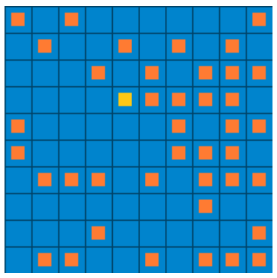

FIGURE 2 – An arena of 10×10 cells, a bumblebee (yellow cell) and obstacle cells (orange cells).

You can use the toolkit for the bumblebee project (on the *Moodle* platform). Make sure you see the animation folder. This is what we want to get in the end. Some of the functions that you must write for this project are the same as those of the snake project. Feel free to use them. Some other functions are similar but not identical. If you copy/paste the functions of the snake project without adapting them, it will bring you no points. When the function prototypes are exhibited in a question, you must use that prototype (the same name, the same parameters, with the same types and coming in the same order). Make sure that your final program compiles. If it does not, it will be very poorly evaluated. At the end of the exam, if one of your functions is preventing the correct compilation, comment out that function, so that the program compiles correctly. This means that you should compile and test your program very regularly during the exam.

1 Obstacle generation

1. Write a function of prototype

   `struct cellList CL_randomFill(int nb_cells, int nb_rows, int nb_cols);`

   which returns a cell list structure containing `nb_cells` cells chosen at random in a `nb_rows` × `nb_cols` arena. Some cells may have the same position. At this stage, this is not a problem. You can see your cell list by calling the `CL_draw2` function in your toolkit. For example figure 3 represents the `obst1.ppm` image file obtained by the following instructions :

```
int nb_rows = 10, nb_cols = 10;
struct cellList obst = CL_randomFill(50,nb_rows,nb_cols);
CL_draw2(obst, nb_cols, nb_rows, "obst1");
```

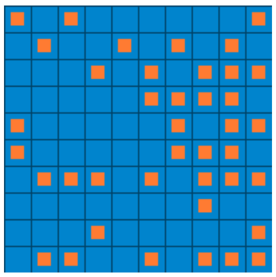

FIGURE 3 – 50 cells chosen at random. Some have the same position. Only 40 cells can be seen here.

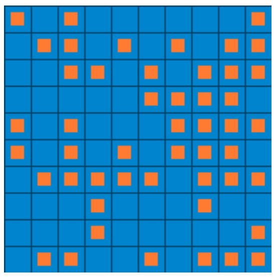

FIGURE 4 – 50 cells chosen at random. Each cell has a different position.

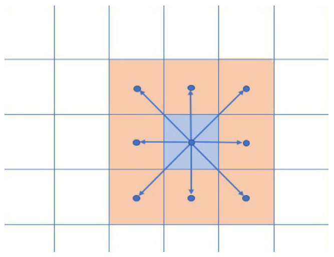

FIGURE 5 – In this project, each non-border cell has 8 neighbors.

2. Write a function with prototype :

   ```
   int CL_isInList(struct cell c, struct cellList cl);
   ```

   which returns 1 if the `cl` list contains a cell located at the same position as `c`. Otherwise, the function should return 0.

3. Write a function with prototype :

   ```
   struct cell CL_randomCellNotInObst(int nb_rows, int nb_cols, struct cellList obst);
   ```

   which returns one cell chosen at random in an arena of `nb_rows` × `nb_cols` cells, but this cell must not be located at the same position as any of the cells of the `obst` cell list. Here is the algorithm : choose a cell `c` at random in the arena (as before). If `c` belongs to `obst` (use the `CL_isInList` function written above), then choose another cell at random and so on until the chosen cell does not belong to `obst`.

4. There may be a situation (a specific value of `nb_rows`, `nb_cols` and `obst`) in which the above algorithm could loop infinitely. Make sure your implementation of `CL_randomCellNotInObst` detects that situation and returns a cell at row -1 and column -1.

5. Write a function with prototype :

   ```
   struct cellList CL_randomFill2(int nb_obstacles, int nb_rows, int nb_cols);
   ```

   which returns a cell list structure containing `nb_obstacles` cells chosen at random in a `nb_rows` × `nb_cols` arena. Each cell must have a different position as in figure 4. Use the `CL_randomCellNotInObst` function written above. If the situation mentioned in the previous question should occur, make sure your function detects it, prints an error message and returns an empty list.

This cell list will be used in the rest of the project as a set of forbidden cells.

2 Neighbors

In the snake project, each cell that is not located at the border of the arena had 4 neighbors. In this project, each non-border cell has 8 neighbors as illustrated in figure 5.

1. Write a function with prototype :

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

   which, for a given cell `c` returns a list of cells corresponding to the neighbors of `c` belonging to the arena `ar`. If `c` is not on the border, the returned list should contain 8 cells. If `c` is a corner cell, there should be 3 neighbors. If `c` is another border cell, then it should have 5 neighbor cells.

2. Write a function with prototype :

   ```
   struct cellList CL_neighborsObst(struct cell c,
                                    struct arena ar,
                                    struct cellList obst);
   ```

   which does the same job as the previous function, but which discards from the neighbor list any cells belonging to the obst cell list. For example if `obst` is the cell list represented on figure 4, and `c` is cell (0,1) (i.e. located at row 0 and column 1) then the returned neighbor list should contain only one cell : (1,0). But if `c` is cell (1,0), then the returned neighbor list should contain 3 cells : (0,1),(2,0) and (2,1).

3. Write a function with prototype :

   ```
   struct cellList CL_randomFlight(struct cell start,
                                   int nb_steps,
                                   struct arena ar,
                                   struct cellList obstacles)
   ```

   which is the counterpart of the `CL_randomPath` function that you wrote for the snake project. This function starts at cell `start` and puts it in a list. Then it chooses `neighb`, one of the non-obstacle neighbors of `start` chosen at random and puts it also in the same list. Next it does the same thing with a non-obstacle neighbor of `neighb` and so on, until the list contains `nb_steps` cells. During the neighbor search, you can choose cells that were already explored. In this way, there is no blocking situation.

4. The cell list returned by the previous function represents the flight of our bumblebee. We would like to see that list in the form of an animation composed of several images. Each image should represent one of the cells of that list. But we want the obstacle cells to appear on all images. Use the `CL_animate2` function (in your toolkit) of prototype :

   ```
   void CL_animate2(struct cellList obst,
                    struct cellList anim,
                    int nb_rows,
                    int nb_cols,
                    char *ppm_name);
   ```

   which produces a series of ppm image files representing an arena of `nb_rows` × `nb_cols` cells. The number of produced files equals the number of cells in the `anim` cell list.
   — `obst` is a cell list that appears in all images ;
   — `anim` is a cell list. The `n`th cell of this list appears on the `n`th image file.
   — `nb_rows` and `nb_cols` represent the size of the arena.
   — `ppm_name` is the name of the image files. For example if `ppm_name` is `"bumblebee"` (and if `anim` contains 25 cells) then the produced images will range from `bumblebee_00.ppm` to `bumblebee_24.ppm`.

Delivery

In an archive file (zip or tgz) put your code and the set of images produced by the last question and upload the archive file in the link on Moodle.