SZÁMÍTÓGÉPES SZIMULÁCIÓK

Sejtautomaták

jegyzőkönyv

Jegyzőkönyvet készítette:
Koroknai Botond (AT5M0G)

Jegyzőkönyv leadásának időpontja:
2024.05.19

1. Feladat

A feladat során egy Conway-féle életjátékot szimuláló sejtautomatát kellett, hogy megvalósítsunk, melynek alap szabályai:

- ha n élő szomszéd van, akkor nem változik a sejt állapota
- ha n+1, akkor a sejt élő lesz
- egyébként elpusztul.

Különböző határfeltételek mellet kell, hogy vizsgáljuk:

- nyílt peremfeltétel
- periodikus peremfeltétel
- élő határ
- a peremen véletlenül sorsolt állandó állapot

1.1. Nyílt peremfeltétel

Ebben az alfejezetben bemutatom a szimulációt megvalósító program felépítésést, a különböző határfeltételeknél már csak a módosításokat fogom szemléltetni.

1. kód. Cellák frissítése nyílt határfeltétel mellett

```
void updateCell(const std::vector<std::vector<int>> &board, std::vector<std::vector<int>> &
{
    int liveNeighbors = 0;
    int rows = board.size();
    int cols = board[0].size();

    for (int i = row - 1; i <= row + 1; ++i)
    {
        for (int j = col - 1; j <= col + 1; ++j)
        {

            if (i >= 0 && i < rows && j >= 0 && j < cols && !(i == row && j == col) &&
            board[i][j] == 1)
            {
                ++liveNeighbors;
            }
        }
    }


    if (liveNeighbors == n)
    {
        newBoard[row][col] = board[row][col];
    }
    else if (liveNeighbors == n + 1)
    {
        newBoard[row][col] = 1;
    }
    else
    {
        newBoard[row][col] = 0;
    }
}
```

A függvény során végig futunk a cellákon és megszámoljuk hány élő szomszédja van, a megfelelő határfeltételeknek következtében. Ezt követően az élő szomszédok számának függvényében frrítjük, hogy élő lesz-e, vagy nem.

Egy másik függvény, az *updateBoard* segítségével, végig megyünk az összes cellán és elvégezzük állapotuk ellenőrzését.

Mivel a szimulációt n = 1- 8 értékeke között futtattam, ezért különböző méretű csoportkat adtam meg kezdeti feltételnek. A szimulációk animációi a task/nyiltperem mappában találhatók.

1.2. Élő határ

2. kód. Tábla friisítése élő határ mellet

```
void updateBoard(std::vector<std::vector<int>> &board, int n)
{
    int rows = board.size();
    int cols = board[0].size();

    std::vector<std::vector<int>> newBoard(rows, std::vector<int>(cols, 0));


    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < cols; ++j)
        {
            updateCell(board, newBoard, i, j, n);
        }
    }


    for (int i = 0; i < rows; ++i)
    {
        newBoard[i][0] = 1;
        newBoard[i][cols - 1] = 1;
    }
    for (int j = 0; j < cols; ++j)
    {
        newBoard[0][j] = 1;
        newBoard[rows - 1][j] = 1;
    }


    board = newBoard;
}
```

Annyi a különbség az előző verzióhoz képest, hogy minden tábla friisítést megelőzően a tábla határáit mindig élő cellékra állítjuk. A feladathzo tartozó animációk a task1/elo mappában vannak.

1.3. Random határfeltétel

Ez nagyon hasonlít az előző esethez, csak most az *updateBoard* függvény véleletlenszerűen 0 vagy 1 értékeket választ a tábla szélére.

1.4. Periodikus határfeltétel

3. kód. Tábla friisítése periodikus határ mellet

```
void updateCell(const std::vector<std::vector<int>> &board, std::vector<std::vector<int>>
{
    int liveNeighbors = 0;
    int rows = board.size();
    int cols = board[0].size();
```

```
for (int i = row - 1; i <= row + 1; ++i)
{
    for (int j = col - 1; j <= col + 1; ++j)
    {

        int r = (i + rows) % rows;
        int c = (j + cols) % cols;

        if (board[r][c] == 1 && !(r == row && c == col))
        {
            ++liveNeighbors;
        }
    }
}
```

A periodikus határfeltétel biztosítja, hogy a tábla szélei összekapcsolódnak. Ez a két sor végzi el:

- int r = (i + rows) % rows: Az i indexet úgy módosítja, hogy ha az index a tábla határain kívül esne, akkor ”átcsomagolódik” a másik oldalra.
- int r = (i + rows) % rows: pedig a sorok esetén

Az animációk a task1/periodikus mappában találhatóak.