Life - Part 2: Assignment Specification

CAB201 Semester 2, 2020

Last Updated: 08/10/2020

Assigment Details

Progress Due Date: 18/10/2020 - 11:59pm
Final Due Date: 25/10/2020 - 11:59pm
Weighting: 35%
Type: Individual Project
Version: 1.3 - 08/10/2020

Change Log

There is no intention to have this assignment specification change over time. However, if anything does change or additional detail is added to the specification, the table below will be updated to reflect the changes.

Version	Date	Description
1.0	21/09/2020	Initial specification
1.1	25/09/2020	Rectified validation requirements for neighbourhood command line option
1.2	02/10/2020	Fixed test-case videos and seed files
1.3	08/10/2020	Fixed some small typos and added new test cases

Overview

In Part 1 of your major project for this semester you implemented the classic, *Conway's Game of Life*. Part 2 is focussed on extending your implementation to be more generalised, scalable and versatile. As such, your solution to Part 1 is crucial to completing Part 2. This specification details the additional features that your game must implement. This assignment is designed to test your ability to:

- Modify existing code to implement new features
- Enforce good object oriented programming practices
- Handle unexpected code behaviour elegantly

Neighbourhoods

Previously, the neighbourhood was defined as the cells positioned horizontally, vertically and diagonally adjacent of any given cell. This is type of neighbourhood is also commonly referred to as a *Moore* neighbourhood. There is another type of neighbourhood that will be introduced for this assignment: the *von Neumann*. Both of these neighbourhoods will be of **variable size** and described below. Your program must have the ability to observe these neighbourhoods of a specified order.

In addition, your program must have the ability to count the centre of the neighbourhood part of the neighbourhood when requested.

Moore Neighbourhood

An n^{th} order *Moore* neighbourhood is defined as the cells that have a *Chebyshev* distance of n or less from any given cell. The *Chebyshev* distance is defined as the largest distance in a single dimension among all dimensions between two points. A *Moore* neighbourhood may be easier to understand visually. See figure 1 for a visual depiction of a Moore neighbourhoods of different sizes.

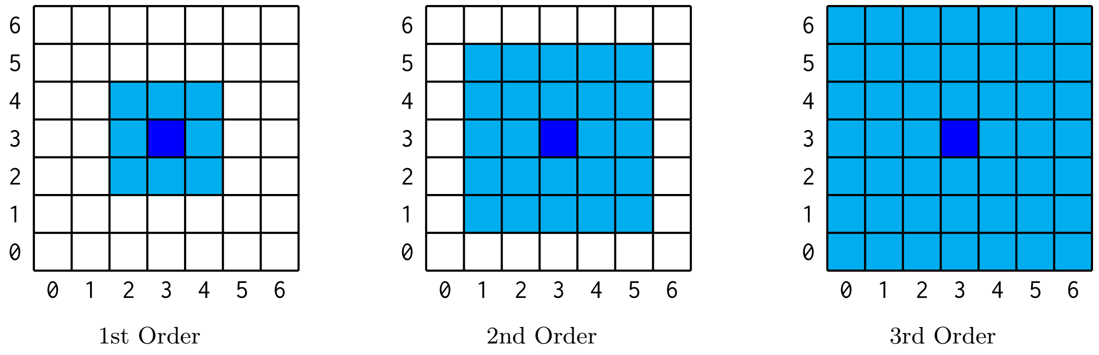


Figure 1: Visual representation of a Moore neighbourhoods of various orders in *Life*. The neighbours of the cell highlighted in blue are highlighted in light blue.

Von Neumann Neighbourhood

An n^{th} order *von Neumann* neighbourhood is defined as the cells that have a *Manhattan* distance of n or less from any given cell. The *Manhattan* distance is defined as the sum of distances in each dimension between two points. As with the *Moore* neighbourhood, the *von Neumann* neighbourhood may be easier to understand visually. See figure 2 for a visual depiction of a Moore neighbourhoods of different sizes.

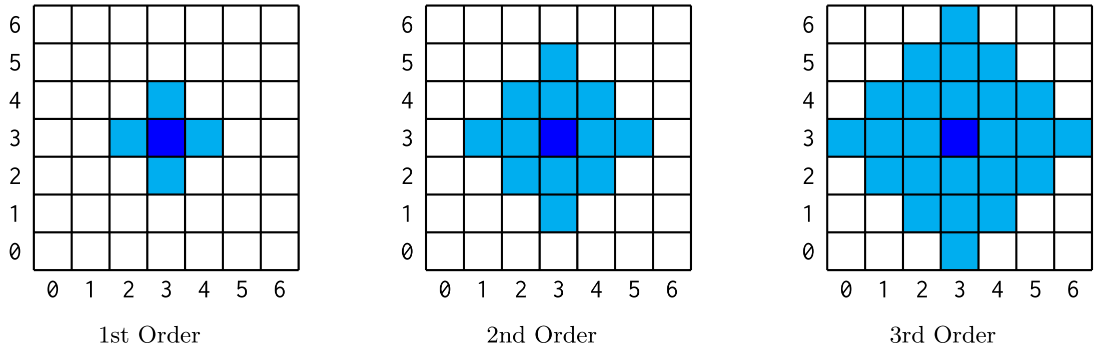


Figure 2: Visual representation of a von Neumann neighbourhoods of various orders in *Life*. The neighbours of the cell highlighted in blue are highlighted in light blue.

Custom Rules

With variable neighbourhood sizes and shapes comes the necessity of more versatile rules. Previously, the number of alive neighbours required for a cell to survive or be born was fixed. Your program must have the ability to evolve by allowing cells to survive or be born based on a variable number of alive neighbours required for either.

As before, the game will progress (or evolve) through a series of generations, with each generation depending on the one that proceeded it. But this time, the rules that determine the next generation are generalised to the following:

- Any live cell with exactly any number of live neighbours contained in the set of numbers $\mathcal{S}$ survive (stay alive) in the next generation.
- Any dead cell with exactly any number of live neighbours contained in the set of numbers $\mathcal{B}$ are born (become alive) in the next generation.
- All other cells are dead in the next generation.

The original rules of *Life* can still be respected using these generalised rules with the following set definitions:

$$\mathcal{S} = \{2, 3\}$$
$$\mathcal{B} = \{3\}$$

Steady-State Detection

During your development and testing of Part 1, you likely noticed that the game can exhibit forever repreating patterns or settle to some unchanging state. Both of these scenarios are considered as *steady-states*. Your program must be able to detect *steady-states* by comparing each generation against its predecessors.

By default your program must use the 16 most recent generations to determine whether a steady state has been reached, but this can be as high as 512 and as low as 4. When a steady state is reached, the game must be marked as complete and the user must be notified with a message after the grid display is closed.

The notification should detail whether a steady-state was reached. If a steady-state is reached, the program should report the periodicity of the steady-state. The periodicity refers to the number of generations required for the steady-state to repeat itself. If the steady-state is non-periodic (for example, in the case where all cells die) then the periodicity should be displayed as *none* or *N/A*.

File I/O

If you ever tried creating your own seed file based on the previous specification, you may have noticed that it is simple, but rather tedious. For Part 2, your program must be able to read a new version (`#version=2.0`) of seed files. Instead of having each line of the file detailing the row and column of alive cells, the new version of the seed file will allow each line to detail a *structure* of cells and whether it should be initially treated as dead or alive. This way, initial configurations can be more easily designed.

The new version of seed files have a number of structures (described below) and will be formatted according to the following rules:

- The first line of the file will always be `#version=2.0`.
- Each line of the file represents a structure that is initialy alive or dead with the following syntax:
  - The line will start with an `x` or an `o` character in parenthesis. `o` if initially alive, `x` is initially dead.

- A keyword representing the structure type will follow, followed by a colon.
- The parameters for the structures will follow, each separated by commas.
- Whitespaces are to be ignored completely when reading each line.

Below is an example of a seed file that would result in the glider pattern shown in the last specification:

```
#version=2.0
(o) cell: 3, 2
(o) cell: 2, 0
(o) cell: 2, 2
(o) cell: 1, 1
(o) cell: 1, 2
```

It is important to note that your program must be able to support **both** versions of seed files when reading. Your program should use the version number to determine how it is going to read the file.

Your program must also be able to write the final generation of the game to a seed file if an output file path is specified via the command line arguments. The seed file must be written using the `#version=2.0` specification, using only the `cell` structure.

`cell` Structure

The `cell` structure can be used to specify a cell, given it's location. The parameters that can be provided to `cell` are the following (in order):

- ☐ The row index
- ☐ The column index

`rectangle` Structure

The `rectangle` structure can be used to specify a rectangle, given the location of two of it's opposing corners. The parameters that can be provided to `rectangle` are the following (in order):

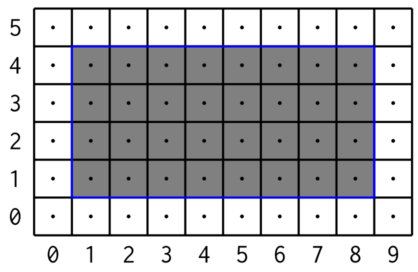
- ☐ The bottom-left row index
- ☐ The bottom-left column index
- ☐ The top-right row index
- ☐ The top-right column index

`ellipse` Structure

The `ellipse` structure can be used to specify an elipse, given the location of two of it's opposing bounding corners (you are to assume that the ellipse is bounded by a rectangle). The parameters that can be provided to `ellipse` are identical to rectangle (see above). The cells that make up an ellipse are those whose centre-points are within the elipse (or on it's border). Figure 3 contains a visual depiction of this, which may help with your understanding.

You may find the following equation useful when determining which cells are part of an ellipse structure. In the equation x_0 and y_0 represent the coordinates of the centre of the ellipse, d_x and d_y represent the width and height of the ellipse, and x and y represent the coordinates of some arbitrary point. If the equation is true, then the point (x, y) is inside the ellipse.

$$\frac{4(x - x_0)^2}{d_x^2} + \frac{4(y - y_0)^2}{d_y^2} \leq 1$$


(o) rectangle : 1, 1, 4, 8

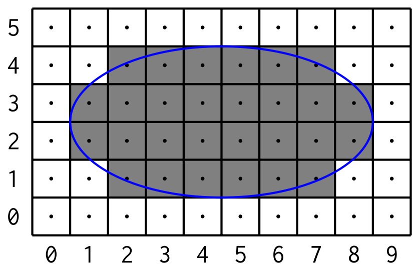

(o) ellipse : 1, 1, 4, 8

Figure 3: Example of the cells represented by `rectangle` and `ellipse` structures, each bounded by (1,1) and (4,8). Note that only cells that have their centre point within the ellipse are represented

Ghost Mode

As a "bonus feature", your program should be able to render the game of life using what will be referred to as *ghost mode*. Instead of just rendering the latest generation, ghost mode should make the program render the four most recent generations of life concurrently. The oldest generation should be rendered using the lightest shading, and the shading should increase for the more recent generations. Ghost mode is best understood visually so be sure to watch the related test cases to gain a better understanding.

If you have read into the code provided in the `Display` library, you may note that the `CellState` enum contains more states than the two that you would have used in Part 1 (`Blank` and `Full`). You should be using the the remaining states (`Dark`, `Medium` and `Light`) in tandem to implement ghost mode.

Command Line Arguments

Many of the new features introduced above will require their own command line arguments in the form of new options and parameters. Each of the new options will be described below.

Neighbourhood

The type and size of the neighbourhood to be used may be specified using the `–neighbour` option. The flag must be followed by three parameters, the first specifying the neighbourhood type, the second specifying order (size) of the neighbourhood, and the third specifying whether the centre of the neighbourhood is counted as a neighbour or not.

Defaults: The game will use a 1st order Moore neighbourhood that doesn't count the centre.

Validation: The neighbourhood type must must be one of two strings, either `"moore"` or `"vonNeumann"`, **case insensitive**. The order must be an integer between 1 and 10 (inclusive) and less than half of the smallest dimensions (rows or columns). Whether the centre is counted will be a boolean (true or false)

Usage: `--neighbour <type> <order> <centre-count>`

Survival and Birth

The number of live neighbours required for a cell to survive or be born in evolution may be specified using the `--survival` and `--birth` options respectively. These flag should be followed by an **arbitrary** number of parmeters (greater than or equal to 0).

Defaults: Either 2 or 3 live neighbours are required for a cell to survive. Exactly 3 live neighbours are required for a cell to be born.

Validation: Each parameter must be a single integer, or two integers separated by ellipses (...). Integers separated by ellipses represent all numbers within that range (inclusively). The numbers provided must be less than or equal to the number of neighbouring cells and non-negative.

Usage: `--survival <param1> <param2> <param3> ... --birth <param1> <param2> <param3> ...`

Generational Memory

The number of generations stored in memory for the detection of a steady-state may be specified using the `-memory` option. This flag should be followed by a single parameter.

Defaults: The program should store 16 generations for detecting a steady-state.

Validation: The value must be an integer between 4 and 512 (inclusive).

Usage: `--memory <number>`

Output File

The path of the output file may be specified using the `--output` options. This flag should be followed by a single parameter.

Defaults: No output file is used.

Validation: The value must be a valid absolute or relative file path with the `.seed` file extension.

Usage: `--output <filename>`

Ghost Mode

Whether the program will render the game using ghost mode may be specified using the `--ghost` option.

Defaults: The program will **not** run in ghost mode

Validation: N/A

Usage: `--ghost`

Validation

Each of the command line arguments that can be provided to the program have the potential to modify the behaviour of the program. The program must check to see if each of the arguments is valid in accordance with the descriptions above. If the arguments for a particular category happen to be invalid, your program should revert to the defaults for that category and continue to run your program. The user should be issued a warning describing the problem and how it intends to recover.

Test Cases

To ensure your program is working as intended you should test it thoroughly. Appendix A contains a series of test cases for your program. It is imperative that you test your program using these test cases before submission as your program will be marked using similar test cases. However, that does not mean you should not limit yourself to the test cases presented here.

Each test case has a link to a video that shows the intended behaviour of each test case. Note that for test cases that involve a random seed, you will not be able to exactly replicate the video demonstration, but it will give you an expectation on the type of behaviour.

Program Flow

Your program should follow the following program flow:

1. The program displays the success or failure of interpreting the command line arguments in the console window.
2. The program displays the runtime settings based on a combination of command line arguments and defaults.
3. The program waits until the user presses the space-bar key to begin.
4. The game is run using the runtime settings. If step mode is active, the program must wait until the user presses the space-bar key before showing the next generation.
5. Once the game is complete, the program must wait until the user presses the space-bar key to revert window to it's original state.
6. The program must report information regarding the steady-state of the game.

Deliverables

For this assignment, You are required to submit a **zip** file – via Blackboard – containing the following items:

- ☐ The solution folder (containing your whole solution and it's projects).
- ☐ A brief user manual (in the form of a `README.md`).
- ☐ A self-filled CRA and statement of completeness.
- ☐ A report detailing instances object oriented programming in your project.

A submission template has been provided on Blackboard that you should download and use for your submission.

Do not submit just the `.cs` files. Do not submit just the `.sln` file. You must submit the folder which contains the `.sln` file and all folders and files contained within.

If you are unsure how to submit your assignment correctly, please seek advice from your tutor. Be sure to test by zipping your folder, moving it to a lab computer or virtual machine, and unzipping. Ensure that you can open your solution via the `.sln` file and that your solution builds and runs as expected.

Below is a description of each of the non-code items required for submission. You will be awarded points for completing each of the following items in full, make sure you complete and submit both to maximize your grade.

1. **User Manual**: Briefly explain how to build and run the *Release* version of your program using Visual Studio. Describe the location of your executable and how it should be run with command line arguments (adhering to the command line argument guidelines detailed in this specification). This needs to be formatted using Markdown formatting which you can read more about here.

2. **CRA and Statement of Completeness**: This document acts as a declaration of your work and progress on the assignment. The process of filling out this document will ensure that you full understand the task and its markable requirements. You must fill this document honestly (don't give yourself full marks unless you really believe you have aced the assignment), it exists purely for your own benefit.
3. **Report**: This document should detail how you applied object oriented programing practices in your project, and how it benefitted your solution. You will be required to provide at least one *good* example of each of the following: Encapsulation, Inheritance, Polymorphism and Exception Handling

Submission Instructions

Please follow the instruction details detailed below:

1. Test your program on the practical lab computers or on the QUT VMs to ensure that your code works as intended on a standard machine. **Code that does not compile or run will not receive any marks.**
2. Arrange your deliverables in a zip folder labelled `CAB201_2020S2_ProjectPart2_nXXXXXXXX` where the last part is replaced with your student ID. Your archive must have a `.zip` extension.
3. Submit a preliminary version of your assignment at least once before the progress submission due date. This *progress submission* does not need to be a completed version of your assignment, but you must have made some quantifiable progress. **There are grades allocated for the progress submission.**
4. Submit the final version of your assignment before the final submission due date. The code in your final submission will be the only thing marked directly.

Academic Integrity

All submissions for this assignment will be analysed by the MoSS (Measure of Software Similarity) plagiarism detection system (`http://theory.stanford.edu/~aiken/moss/`). Serious violations of the university's policies regarding plagiarism will be forwarded to the SEF's Academic Misconduct Committee for formal prosecution. As per QUT rules, you are not permitted to copy or share solutions to individual assessment items. In serious plagiarism cases SEF's Academic Misconduct Committee prosecutes both the copier and the original author equally.

It is your responsibility to keep your solution secure. In particular, **you must not make your solution visible online via cloud-based code development platforms such as GitHub**. If you wish to use such a resource, do so only if you are certain you have a private repository that cannot be seen by anyone else. However, it is recommended to keep your work well away from both the Internet and your fellow students to avoid being prosecuted for plagiarism.

Please note that using external libraries for this project will result in academic penalty, you must only use code within the standard .NET API (3.1). You may use any class in the `System` namespace (do not use the `Microsoft` namespace).

Test Cases

Default

Video Demonstration: https://youtu.be/OlHqXVgTqVA

```
>dotnet life.dll
```

Ghost Mode

Video Demonstration: https://youtu.be/oZvycm1L2-E

```
>dotnet life.dll --ghost
```

Diamoeba

Video Demonstration: https://youtu.be/58O3fAisxss

```
>dotnet life.dll --birth 3 5...8 --survival 5...8 --periodic --dimensions 32 32
```

Glider Output

Video Demonstration: https://youtu.be/euP8rxK-njY

```
>dotnet life.dll --seed path\to\glider.seed --dimensions 8 8 --generations 20 --output
    path\to\output.seed
```

Von Neumann

Video Demonstration: https://youtu.be/DS3U3BnAun4

```
>dotnet life.dll --neighbour vonNeumann 1 false
```

Target

Video Demonstration: https://youtu.be/tlspVxp8gY8

```
>type path\to\target.dll
>dotnet life.dll --seed path\to\target.dll
```

Target 2

Video Demonstration: https://youtu.be/3yT0TdZ8cCI

```
>type path\to\target2.dll
>dotnet life.dll --seed path\to\target2.dll
```

Worker Bee Steady-state

Video Demonstration: https://youtu.be/MnlJod8GdSI

```
>dotnet life.dll --seed path\to\workerbee.seed --dimensions 15 20
```

Worker Bee No Steady-state

Video Demonstration: https://youtu.be/vaAhqiqrSqo

```
>dotnet life.dll --seed path\to\workerbee.seed --dimensions 15 20 --memory 4
```

Highlife Replicator

Video Demonstration: https://youtu.be/_9f0m7J20Og

```
>dotnet life.dll --seed path\to\highlife-replicator.seed --birth 3 6 --survival 2 3
    --dimensions 47 47 --generations 200
```

Gnarl Hanabi 1

Video Demonstration: https://youtu.be/FhkVwPvs60I

```
>dotnet life.dll --max-update 10 --dimensions 47 47 --generations 100 --birth 1
    --survival 1 --neighbour vonNeumann 1 true --seed path\to\gnarl-hanabi1.seed
```

Bugs Life

Video Demonstration: https://youtu.be/NNzy0hRVLNU

```
>dotnet life.dll --max-update 15 --dimensions 48 48 --generations 300 --periodic --birth
    34...45 --survival 34...58 --neighbour Moore 5 true
```

Bugs Life 2 (Seed)

Video Demonstration: https://youtu.be/rEu90mRq9QE

```
>dotnet life.dll --max-update 15 --dimensions 48 48 --generations 300 --periodic --birth
    34...45 --survival 34...58 --neighbour Moore 5 true --seed path\to\bugs.seed
```

Kaleidoscope 1

Video Demonstration: https://youtu.be/MGO84kDKgAk

```
>dotnet life.dll --survival 12...20 --birth 7 8 --neighbour vonNeumann 4 true --seed
    kaleidoscope1_47x47.seed --dimensions 47 47 --generations 400 --max-update 30
```

Kaleidoscope 2

Video Demonstration: https://youtu.be/t1iuRAVGj2s

```
>dotnet life.dll --survival 12...20 --birth 7 8 --neighbour vonNeumann 4 true --seed
    kaleidoscope2_47x47.seed --dimensions 47 47 --generations 2250 --memory 200
    --max-update 30
```