IE 172 Laboratory 2: The Game of Life

Dr. T.K. Ralphs

Due February 18, 2016

1 Laboratory Description and Procedures

1.1 Learning Objectives

1. Understand how to document classes and functions in Python.
2. Understand how to use the profiler.
3. Understand how to refactor an existing code.

1.2 Key Words

1. Refactoring
2. List
3. Matrix
4. Profiler

1.3 The Game of Life

The purpose of this lab is to do further analysis on the implementation of Conway's game of life that was provided to you in Lab 1 and to do a basic refactoring using more efficient data structures.

1.4 Data Structure

As discussed in Lab 1, the algorithm for the game is conceptually simple for a small and finite game board. We can store the current state of the population in a two-dimensional matrix. The challenge is how to efficiently update this data structure from one generation to the next. Recall your analysis from Lab 1. Did the given implementation run as fast as you expected?

The game board in the given implementation is stored as a dictionary. Such a data structure makes it very efficient to determine whether a given "key" is present. One of the fundamental operations in updating the board is to determine whether a given cell is on the board or not, so it might seem as though a dictionary is a good choice. By putting the list of cell tuples into a dictionary, we can easily determine whether a given cell tuple is on the board. Does this make sense? If so, what is it that makes the given implementation so slow?

Refactoring the Code *Refactoring* an existing code means re-writing parts of it to improve efficiency, readability, simplicity, etc. The idea is usually to leave the functionality of the API untouched while improving performance. Occasionally, re-factoring can mean changing the API to make the code more usable or maintainable. In the last lab, we changed the interface to the life code to make it more object-oriented. Refactoring to improve efficiency usually involves using different data structures or changing details of the internal algorithmic implementations. In this laboratory, we will consider refactoring of two kinds:

- Documentation for improved readability and usability.
- Improvements to the data structures used for storing the state of the population.

Improving usability and readability of the code involves introducing doc strings for all functions and classes.

To improve performance of the code, we might consider using a different data structure for maintaining the board and doing the updates. The current implementation uses a dictionary for storing the status of each cell. Essentially, this means that rather than being able to look directly at the status of an individual cell by using the coordinates, as you would be able to in a matrix, the status of each cell is stored in a lookup table indexed by an arbitrary key value, which can be thought of as a string. Dictionaries in Python are stored in a data structure called a *hash table* that we will learn more about later in the course. The list of cell tuples does not have an unpredictable structure, however. In fact, we can easily determine by inspection whether a given tuple represents a cell on the current board if we know the dimensions. Thus, we should be able to improve efficiency by storing the board explicitly in a way that exploits the structure of the board.

An alternative (and a slightly more straightforward) data structure to use would be a traditional matrix, which we could implement in Python as a list of lists (there are "true" C-style array and matrix data structures provided by the package `numpy`). The possible advantage of a matrix data structure is that we can directly look up the value of any element of the matrix from it directly by coordinates, whereas with a dictionary, determining the status of a given cell requires looking into a hash table, which can be slower. There are other hidden advantages of a list implementation that you may also discover.

2 Laboratory Assignments

2.1 Programming (50 points)

1. (10 points) Profile the code you were given in Lab 1 and observe that the percentage of the total running time taken by the `count_neighbors` method (and the `keys()` method within it) increases as the size of the board increases. Create a table that shows this.

2. (10 points) Change

   ```
   if (cell[0] + n[0], cell[1] + n[1]) in board.keys():
   ```

   to

   ```
   if (cell[0] + n[0], cell[1] + n[1]) in board:
   ```

   in the original `life` implementation you were given in Lab 1. Verify that the code seems to run much faster. Create a graph comparing the running time of the two versions of the code.

What is the approximate growth rate of the two running time functions (use big-O notation)? Can you explain why?

3. (10 points) Add the ability to define the color of a cell to the data structure for storing the board to your object-oriented implementation from Lab 1. There are a number of possible ways of doing this.

4. (10 points) Define rules by which the colors change and see what interesting patterns you can make by changing these rules. One example might be to start with random colors and then set the color of any cell that is newly alive to be the average of the colors of its neighbors.

2.2 Follow-up Questions (30 points)

1. (10 points) Show that if f and g are running time functions, then we have $f+g \in \Omega(\min\{f, g\})$. You can use the method of limits and assume all limits exist.

2. (10 points) Show formally that $n \lg n \in o(n^2)$.

3. (Extra credit) Describe the steps one would need to undertake to convert the data structure for storing the board to a "list of lists" rather than a dictionary.