Population Dynamics of Conway's Game of Life and Its Variants

Yoni Biel and Martin Pelikan

Abstract
The purpose of this project was to simulate and analyze the behavior of the cellular automaton (CA) known as Conway's game of life and its variants. First, the basic programming concepts necessary to implement a CA were learned by solving a series of increasingly difficult problems in mathematics and number theory. Then, the learned concepts were applied to implement Conway's game of life and several its variants. The implementation was then used to simulate several variants of the game of life in order to gain understanding of the behavior of the simulated CA variants. The results confirmed that the standard set of rules of Conway's game of life is in agreement with some of the main design goals of its author, and that different sets of rules yield a wide variety of behaviors.

1 Introduction

Cellular automata (CA) are computational models (von Neumann, 1948). A CA usually consists of a grid whose individual elements are called cells. At any point in time, each cell is in one state out of a number of states. After the cells' states are initialized, CA use specified rules to decide how the states of the cells change throughout time using discrete time steps, with one time step representing the smallest unit of time. The rules of how the cell's state changes are based on the states of the cell itself and the cells directly surrounding the cell, also known as the cell's neighbors. CA can be used for computations and simulations in various disciplines such as biology, chemistry, physics, computer science, and image processing. CA can also be used to demonstrate the principles of emergence, self-organization, and self-replication in a straightforward manner, which were among the main motivations for the development of CA (von Neumann, 1948).

The purpose of this project is to simulate and analyze the behavior of the CA known as Conway's game of life and its variants (Gardner, M. 1970). In the simulations, we will study the effects of different initialization patterns on the dynamics of the CA. In order to simulate the CA, the basic programming concepts necessary to compute and simulate a CA using the C++ programming language were learned by solving a series of increasingly difficult problems.

The following sections will describe and explain different aspects of the project. Section 2 will describe Conway's game of life and a few of its variants. Section 3 will describe experimental methodology and present the results of the simulations of the Game of Life. Section 4 will summarize and conclude the project. Programming concepts and examples of the problems solved in the initial stage of this project are detailed in Appendix A.

2 Conway's Game of Life

2.1 Description

Conway's game of life, developed by John Conway in the 1970s, is an implementation of a cellular automaton with a set of a few specific rules. Conway's original version of the game is based on the following set of rules (Gardner, M. 1970):

1. Each cell exists either in a living or dead state.
2. Living cells with fewer than two living neighbors die from loneliness.
3. Living cells with more than three living neighbors die from overcrowding.
4. Dead cells with three living neighbors become alive as a result of reproduction.
5. Otherwise, the cells' states remain the same.

He specifically developed these rules according to the following expectations (Gardner, M. 1970):

1. There shouldn't be any initial patterns that have simple proofs that the populations can grow without limit.
2. There should be initial patterns that can seemingly cause populations to grow without limit.
3. There should be simple initial patterns that cause populations to change for a large amount of time until the cells die from overcrowding, die from being too thinly spaced out, or until the cells become static or form oscillating structures.

From the computational perspective, Conway's game of life is interesting especially because it is relatively simple yet it has the power of simulating any algorithm by simulating the universal Turing machine (Adamatzky, 2002). Furthermore, as was already mentioned, the game of life demonstrates how simple rules of local interaction can lead to the emergence of complex behavior, self-organization, and self-replication (Janssen and Ostrom, 2006); these features make the game of life an interesting model for biology, social sciences, physics, and artificial intelligence.

2.2 Special Structures

The rules of the game of life give rise to various interesting patterns, including static patterns that do not change over time, patterns that periodically repeat a sequence of states, and patterns that appear to travel through the grid while retaining their basic shape. This section presents several examples of such patterns.

Still Lives:

Still lives are patterns that stay in the same shape. The formation of still lives is the main reason for the ceasing of changes in any simulation. Several still lives are

presented in figures 2.1, 2.2, 2.3, and 2.4; all of these patterns were obtained from Burgers (2009).

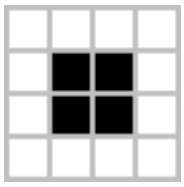

Fig. 2.1 The still life known as a block (Burgers, 2009).

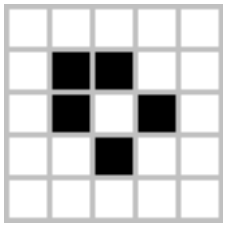

Fig. 2.2 The still life known as a boat (Burgers, 2008).

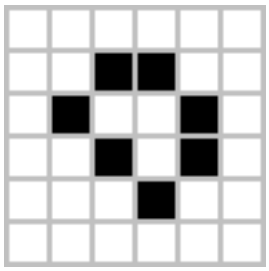

Fig. 2.3 The still life known as a loaf (Burgers, 2009).

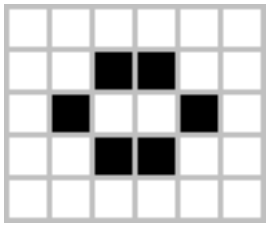

Fig. 2.4 The still life known as a beehive (Burgers, 2009).

Moving Structures:

The two most basic patterns that move while retaining their shape are *gliders*, shown in figure 2.5, and *lightweight spaceships* (LWSS), shown in figure 2.6.

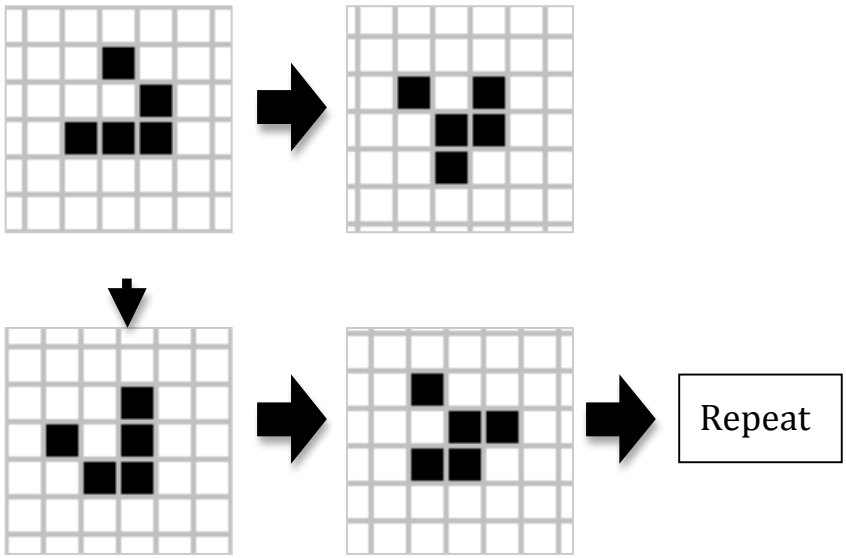

Fig. 2.5 The moving pattern known as a glider (Camargo, 2008).

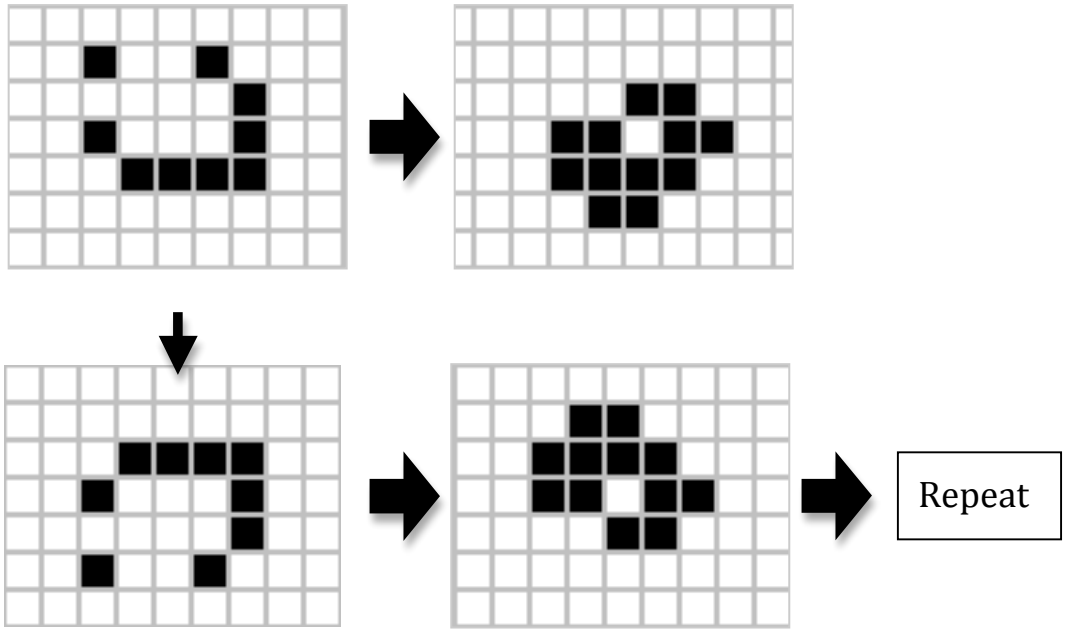

Fig. 2.6 The moving pattern known as a lightweight space ship (Camargo, 2008).

Oscillating Structures:

Oscillating structures, or oscillators, are patterns that are in a repeating loop. Two examples of oscillators are shown in figures 2.7 and 2.8.

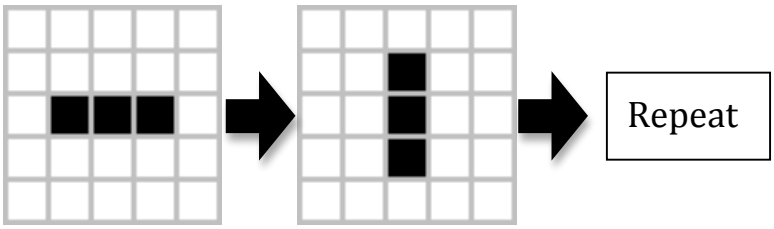

Fig. 2.7 The oscillator known as a blinker (Johnston, 2009).

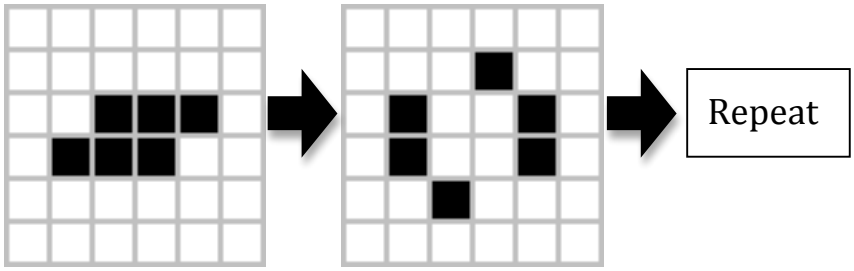

Fig. 2.8 The oscillator known as a toad (Johnston, 2009).

2.3 Variations

The rules for the game of life can be defined by the statement B3/S23. B3 states that cells are born only by having 3 neighbors, and S23 states that living cells survive only by having 2 or 3 neighbors. Similarly, any similar, two-state CA can be defined by a specific combination of numbers defining survival of living cells and birth of new cells. Some commonly studied variants of the game of life include the *high life* rule set B36/S23 created by Nathan Thompson, the *seeds* rule set B2/S created by Brian Silverman, and the *diamoeba* rule set B35678/S5678 created by Dean Hickerson (Wojtowicz, M. 2001).

3 Simulations and Results

An implementation of the game of life was developed in the C++ programming language to run the simulations according to the rule statements. The code was written in order for the user to be able to control the following aspects of the simulations:

1. The size of the grid.
2. The initial proportion of the number of living cells.
3. The maximum number of time steps for each run of the simulation.

4. How often, in terms of time steps, the grid was displayed as an output.
5. The number of runs to be executed.
6. Whether or not the grid would be displayed in the output.

The code was also designed to output the following statistics of the simulations after each time step:

1. The number of time steps that had passed.
2. The average number of living cells in the grid.
3. The average number of dead cells in the grid.
4. The average percentage of living cells in the grid.
5. The average percentage of dead cells in the grid.
6. The average percentage of cells that changed states during the specified time step.

In the simulations, it is assumed that the neighborhoods "wrap around", so the top neighbors of the cells in the top row are located in the bottom row and, similarly, the right neighbors of the right-most column are located in the left-most column. Therefore, the cells are in fact arranged on a two-dimensional torus. To initialize the states of all cells in the grid, a pseudorandom generator was used to set each cell's state to alive with the given initial proportion, which was a user specified parameter. For example, for the initial proportion 20%, about 20% of randomly chosen cells would start in the living state. The initial proportion was typically varied between 0% and 100% to gain understanding of the effects of this parameter on the dynamics of the simulation.

The entire code remained the same for each simulation except for the section containing the rules of the simulation. For each simulation, the rules were changed to agree with the rule set used in the simulation. The main goal of simulating a few variants of the game of life was to observe and analyze the dynamics of each variant, mainly focusing on the population dynamics, stable states, and differences between the rule sets.

A preliminary study of the Game of Life was conducted using this code. The results showed that world sizes below 20 by 20 cells caused the percentage of living cells to drop significantly, while world sizes greater than and equal to 20 by 20 cells had very similar effects upon the percentage of living cells. Conclusions were made that a 20 by 20 cell world would be the best size for later simulations because of its fast computational speed; so later simulations were only conducted using 20 by 20 cell worlds.

3.1 Simulation B3/S23

The first simulation studied was the original game itself. The rule statement for this simulation was B3/S23. This means that living cells survive if they have 2 or 3 neighbors and dead cells become alive if they have exactly 3 neighbors. The size of the world for each of the simulations was set to be 20 by 20 cells. The number of runs for each simulation was set to 100 runs. The number of time steps for each

simulation was set to 100 time steps per run. The results of this set of simulations were similar for most settings of the initial proportion of living cells. When the initial proportion of living cells was set to 20%, the proportion of the living cells gradually dropped to about 9% over the course of the simulation. When the initial proportion of living cells was set to 40%, the proportion of the living cells gradually dropped to about 12% after about 55 time steps, where it leveled out and only dropped to about 10% over the course of the remaining time steps. When the initial proportion of living cells was set to 60%, the proportion of the living cells immediately dropped to about 15% after 1 time step, where it dropped at a relatively constant rate to about 9% over the remaining course of the simulation. When the initial proportion was set to 80%, the proportion of living cells immediately dropped to about 0% after 1 time step, where it remained for the remainder of the simulation. Each of these simulations is shown in Figure 3.1.

The results in figure 3.2 lead to further inferences about the dynamics of the game. In each respective initial proportion of living cells studied, the dynamics of the proportion of cells that changed states followed a nearly identical trend as the dynamics of the proportion of living cells. Only the dynamics of the 80% initial proportion were exactly identical regarding both aspects. The only clear difference between the two aspects of the 20%, 40%, and 60% initial proportions was that at any given time step, the proportion of changed cells was about 3% lower than the proportion of living cells. It can be inferred from the 3% of the cells that didn't change but were alive that this 3% of living cells formed still lives and became static early on in the simulation. In addition, some of the cells that did change may have formed oscillating structures, while others underwent normal birth and death.

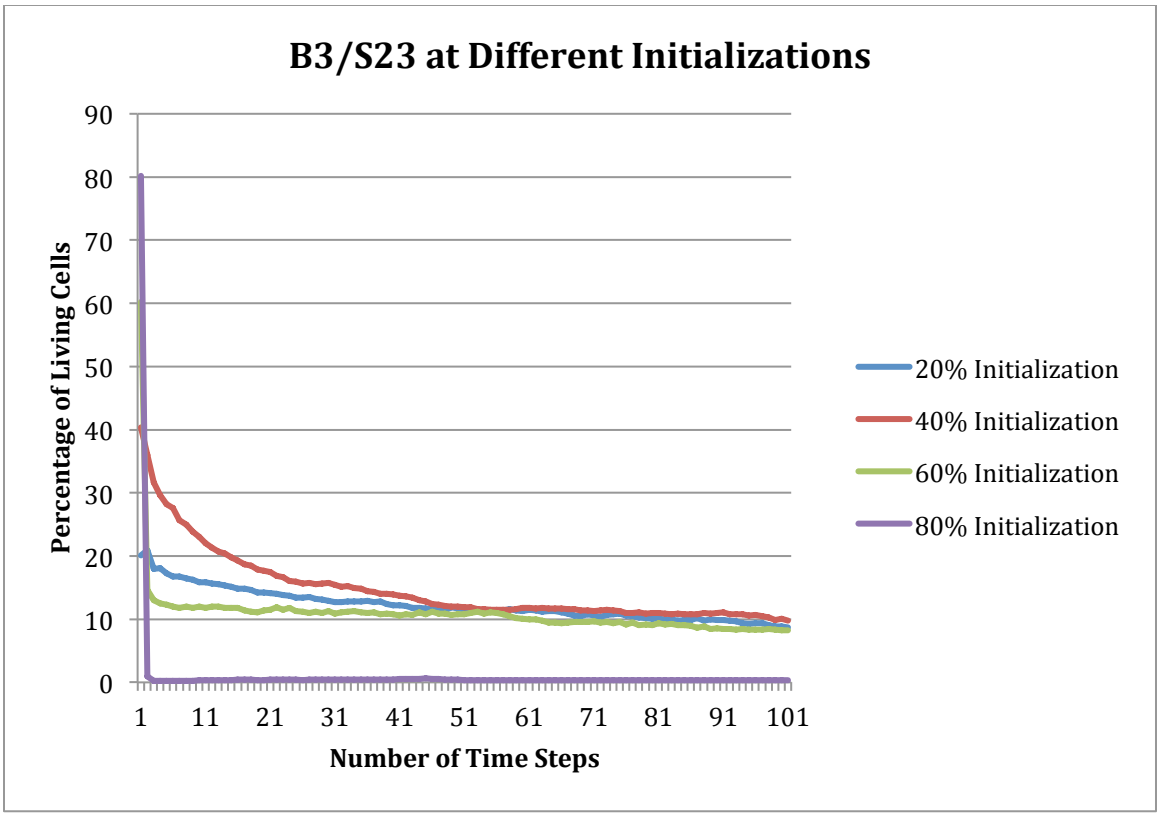

Fig. 3.1. The proportions of living cells using the rule set B3/S23 for a range of settings of the initial proportion of living cells. The initial proportions of 20%, 40%, and 60% all dropped to a proportion of about 12% after about 50 time steps, and they all gradually dropped about 2% during the last 50 time steps. The 80% initial proportion resulted in the proportion of about 0% after about 1 time step, where it stayed for the remainder of the simulation. The 80% initial proportion seemed to have resulted in an immediate death of all or most living cells by overcrowding. Each initial proportion of living cells shows a similar trend of a decreasing percentage of living cells.

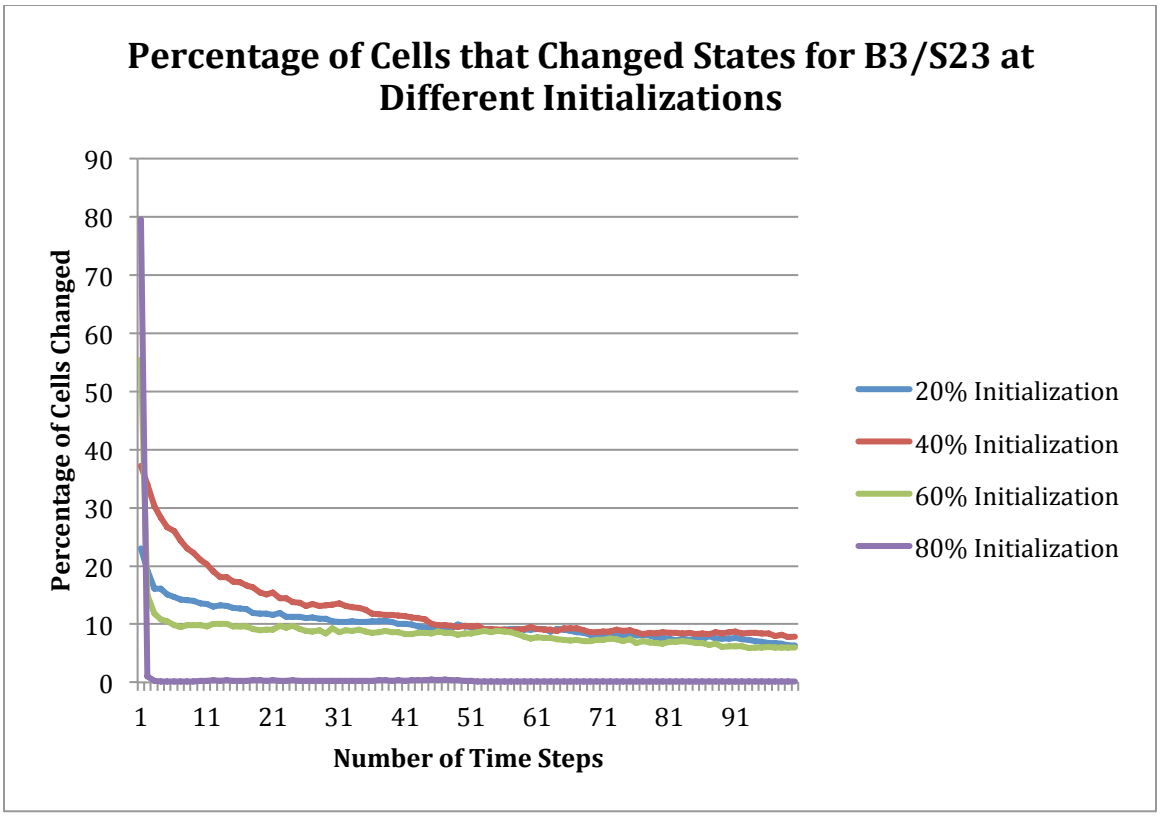

Fig. 3.2. The proportions of changing cells using the rule set B3/S23 for a range of settings of the initial proportion of living cells. The initial proportion of 20% began with about 3% more cells changing than there were initially living, continuing with a quick drop of about 6% after about 3 time steps, and resulting in a gradual drop to about 7% by the end of the simulation. The initial proportions of 40% and 60% began with about 3% fewer cells changing than there were initially living. The 40% initial proportion dropped about 17% after about 20 time steps and gradually continued to drop until it reached about 8% at the end of the simulation. The 60% initial proportion quickly dropped to about 10% after about 5 time steps and gradually continued to drop until it reach about 7% at the end of the simulation. The initial proportion of 80% immediately dropped from 80% to about 0% after about 2 time steps, where it continued until the end of the simulation. It can be inferred from each initial proportion of living cells that over time either more still lives had formed or more cells remained dead.

3.2 Simulation B23/S3

The second simulation studied was a simple variation of the game of life. The rule statement for this simulation was B23/S3. This means that living cells survive if they have exactly 3 neighbors and dead cells become alive if they have 2 or 3 neighbors. I proposed this rule set with the motivation to see how different the results could be by simply swapping the numbers from the two parts of the original rule set with each other. The size of the world for each of the simulations was set to be 20 by 20 cells. The number of runs for each simulation was set to 100 runs. The

number of time steps for each simulation was set to 100 time steps per run. The results of this set of simulations were nearly identical for most settings of the initial proportion of living cells. When the initial proportion of living cells was set to 20%, 40%, and 60%, the proportion of the living cells stabilized at about 38% after 5 time steps, as shown in Figure 3.3. When the initial proportion was set to 80%, the proportion of living cells stabilized at about 20% after about 20 time steps, as shown in Figure 3.3. The information in figure 3.4 leads to further inferences about the dynamics of the rule set. In each respective initial proportion of living cells studied, the dynamics of the proportion of cells that changed states followed a nearly identical trend as the dynamics of the proportion of living cells. The only clear differences between the two aspects of the 20%, 40%, and 60% initial proportions was that at any given time step, the proportion of changed cells was about 16% higher than the proportion of living cells, and the 80% initial proportion's percentage of changed cells was about 10% higher than the percentage of living cells at any given time step. It can be inferred from the 16% of the cells that changed and weren't alive in the 20%, 40%, and 60% initial proportions that an average of 27% of cells changed by dying and an average of 27% of cells change by being born so that the percentage of living cells would remain at 38%. It can be inferred from the 10% of the cells that changed and weren't alive in the 80% initial proportion that an average of 15% of cells changed by dying and an average of 15% of cells changed by being born so that the percentage of living cells would remain at 20%. In addition, even though the proportion of the living cell population is stable, it appears that the living cells themselves are not stable.

There is a surprising number of differences between the results of the original game and this rule set, especially from such a small change in the rules. In the original game, the populations of living cells in each initial proportion were gradually decreasing, while in this rule set, the populations of living cells in each initial proportion stabilized at certain percentages of living cell populations. At every initial proportion studied, a higher percentage of cells were living in this rule set than in the original rule set.

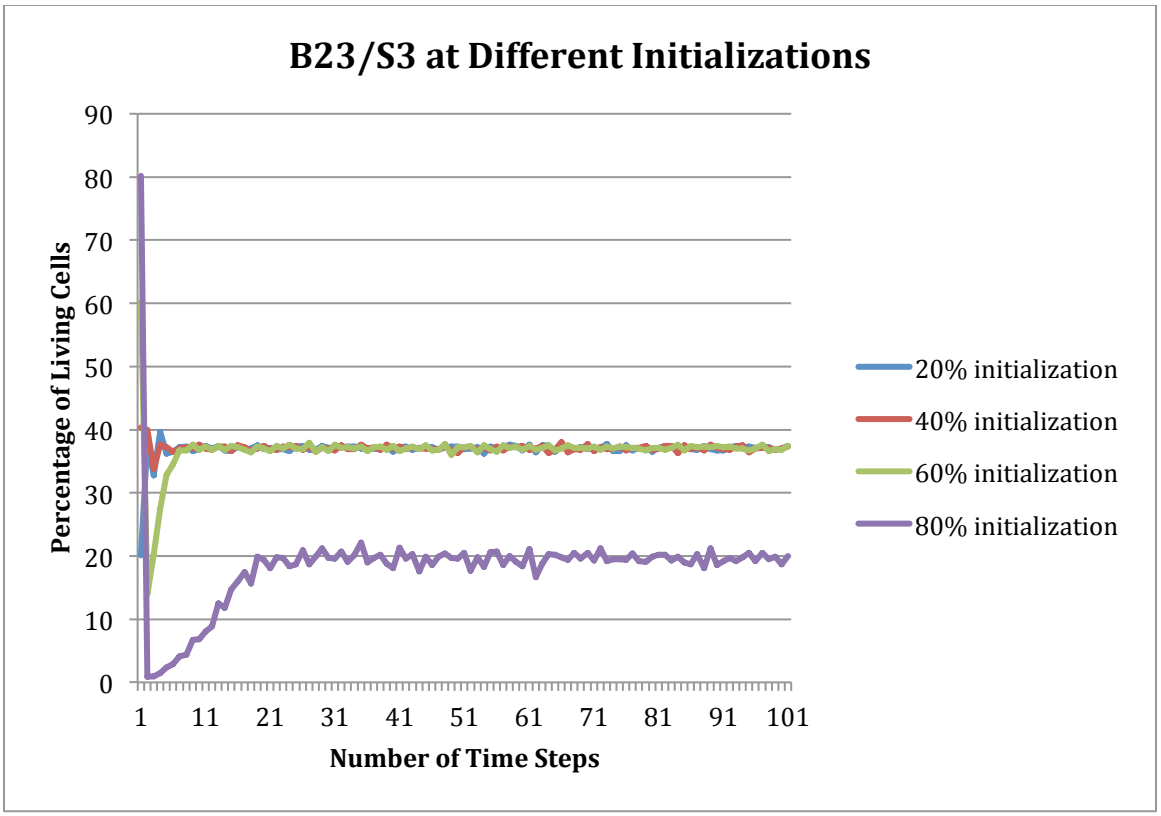

Fig. 3.3. The proportions of living cells using the rule set B23/S3 for a range of settings of the initial proportion of living cells. The initial proportions of 20%, 40%, and 60% all resulted in a stabilized proportion of about 38% after about 5 time steps with only slight fluctuations thereafter. The 80% initial proportion resulted in the proportion of about 20% after about 20 time steps with only slight fluctuations. The stabilization period of every initialization pattern shown in the graph showed no recognizable repeating pattern of its slight fluctuations in the living cell population.

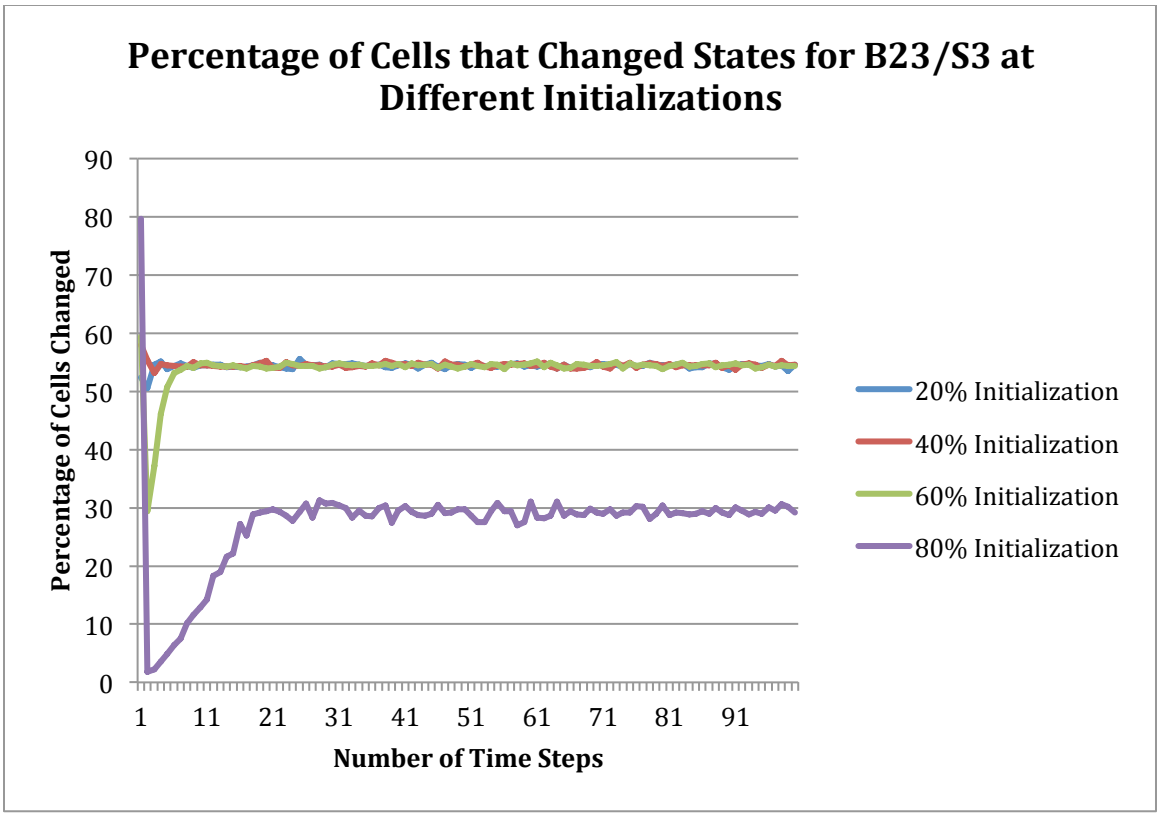

Fig. 3.4. The proportions of changing cells using the rule set B23/S3 for a range of settings of the initial proportion of living cells. The initial proportions of 20%, 40%, and 60% all resulted in a proportion of about 54% changing cells after about 5 time steps with only slight fluctuations thereafter. The 80% initial proportion resulted in the proportion of about 29% changing cells after about 20 time steps with only slight fluctuations.

3.3 Simulation B02468/S02468

The third simulation studied was another variation of the game of life, but this time, the rules were slightly more complex. The rule set for this simulation was B02468/S02468. This means that living cells survive if they have an even number of neighbors and that dead cells become alive if they have an even number of neighbors. I proposed this rule set with no other motivation than to see the results of a rule set where cells are born and survive by having an even number of neighbors including 0. The size of the world for each of the simulations was set to be 20 by 20 cells. The number of runs for each of the simulations was set to be 100 runs. The number of time steps for each simulation was set to 100 time steps per run. The initial proportion of living cells was set to 20%, 40%, 60%, and 80%. For every initial proportion of living cells studied, as shown in figure 3.5, the proportion of living cells stabilized at 50% after 1 time step with only slight fluctuations thereafter. The information in figure 3.2 leads to further inferences about the dynamics of the rule set. In each respective initial proportion of living cells studied, the dynamics of the proportion of cells that changed states followed a seemingly

identical trend as the dynamics of the proportion of living cells. The proportion of changing cells fluctuated in repeating patterns of about 12 time steps at around 50% of the total cells, just as the proportion of living cells had. It can be inferred that the proportion of changing cells is evenly split between cells dying and being born. In addition, even though the proportion of the living cell population is stable, it appears that the living cells themselves are not stable.

There are a number of significant differences between this rule set, the previous rule set, and the original rule set. In the original game, the populations of living cells in each initial proportion were gradually decreasing, and in the previous rule set, the populations of living cells in each initial proportion stabilized at certain percentages of living cell populations. On the other hand, in this rule set, the populations of living cells in every initial proportion all stabilized at the same percentage of living cell population. At every initial proportion studied, a higher percentage of cells were alive in this rule set than in the original rule set and in the previous rule set. Even though the proportions of living cells in the previous rule set were lower than the proportions of living cells in this rule set, the proportions of changing cells in the previous rule set at 20%, 40%, and 60% initial proportions were slightly higher than the proportions of changing cells in this rule set for the same initial proportions of living cells. It may be inferred from this that the cells in this rule set are less stable than in the previous rule set.

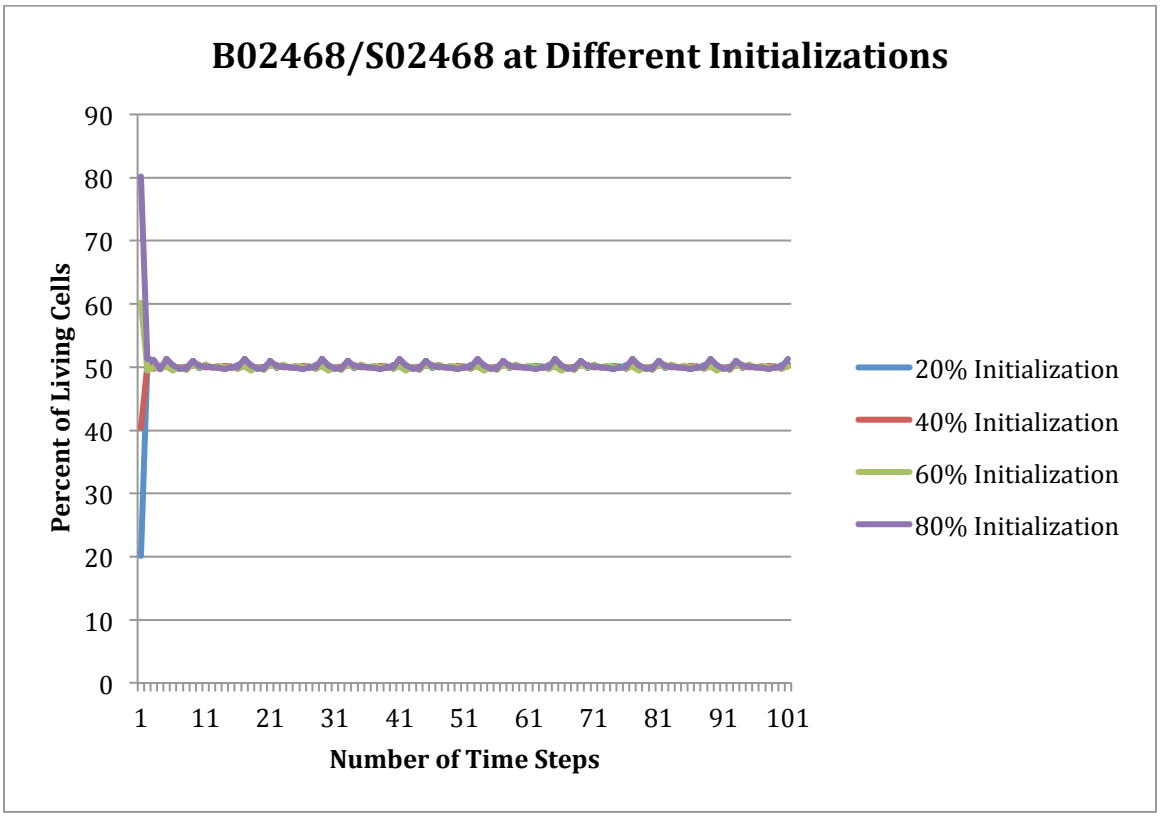

Fig. 3.5. The proportions of living cells for the rule set B02468/S02468 using a range of different initial proportions. Each initial proportion shown in the graph yielded

similar results with the proportion of living cells stabilizing at about 50%. Furthermore, for each initial proportion, the proportion of living cells started to oscillate with the period of about 10 time steps and magnitude of about 2% of the population. Different initial proportions yielded slightly different oscillating patterns.

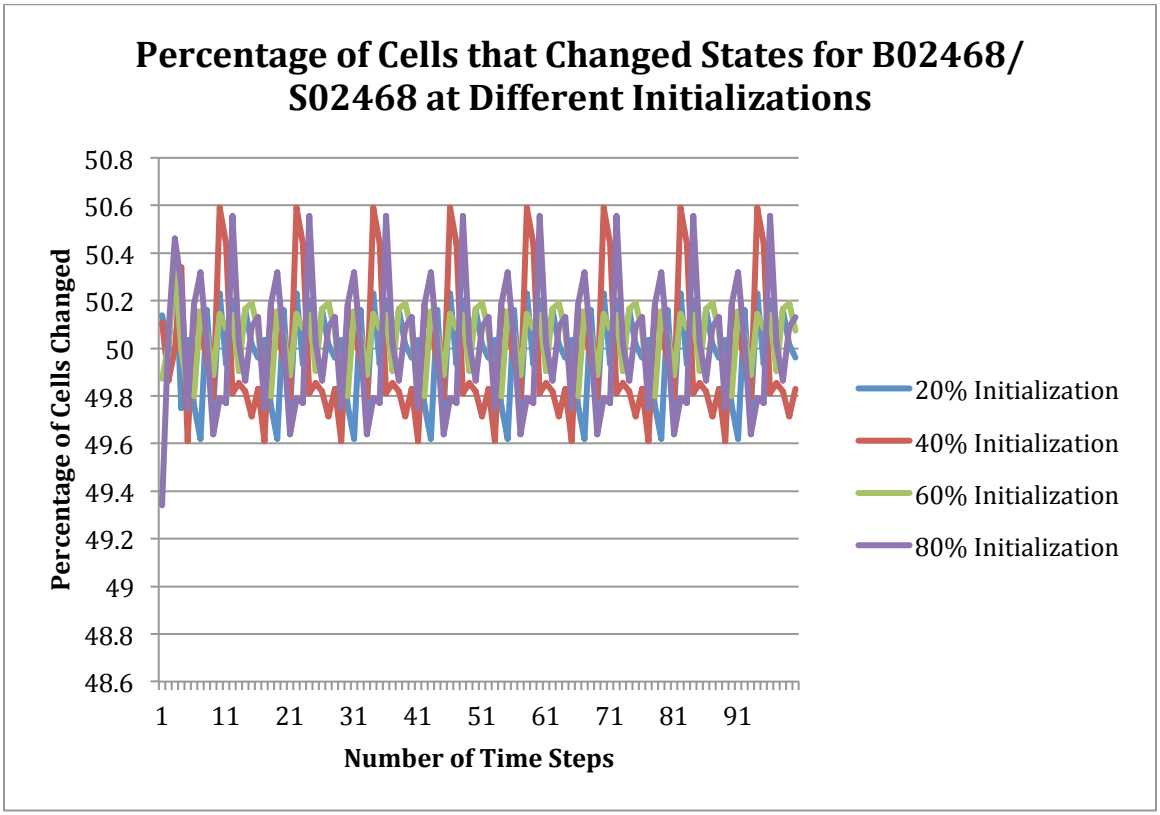

Fig. 3.6. after 12 time steps The proportions of changing cells using the rule set B23/S3 for a range of settings of the initial proportion of living cells. The proportion of changing cells for every initial proportion shown immediately stabilized at 50% with slight fluctuations of about 1%. These fluctuations have patterns that repeat after about 12 time steps.

4 Summary and Conclusion

This paper described just one of the interesting aspects of the computational model known as the cellular automaton (CA). The purpose of this experiment was to study population dynamics in the Conway's game of life and its variants, all of which are examples of CAs. Brief descriptions of Conway's game of life and its variants were provided. In order to simulate Conway's game of life and its variants, a code was implemented using the C++ programming language and the code was modified to explore variants of the basic game of life. New variants with new rules were created and put into the code. The code was designed to be able to control certain aspects of the CA, the most important and most interesting of which was the initial proportion

of living cells. Various experiments were conducted to observe the effects of different initial proportions of living cells on the dynamics of the population.

It has been concluded that simple rule sets for a cellular automaton may yield complex results. A small, simple change to one rule set may yield completely different results. In any given rule set, changes in the initialization patterns of cell populations may yield very similar dynamics, which can vary in complexity. Changes in initialization may also yield some results that are very similar, while other results are very different.

References

Adamatzky, A. (Ed.). (2002). *Collision-Based Computing*. Springer.

Burgers, B. (Creator). (2009). *Game of life: block* [PNG Image], Retrieved July 6, 2012, from http://en.wikipedia.org/wiki/File:Game_of_life_block_with_border.svg

Burgers, B. (Creator). (2009). *Game of life: beehive* [PNG Image], Retrieved July 6, 2012, from http://en.wikipedia.org/wiki/File:Game_of_life_beehive.svg

Burgers, B. (Creator). (2009). *Game of life: loaf* [PNG Image], Retrieved July 6, 2012, from http://en.wikipedia.org/wiki/File:Game_of_life_loaf.svg

Burgers, B. (Creator). (2008). *Game of life: boat* [PNG Image], Retrieved July 6, 2012, from http://en.wikipedia.org/wiki/File:Game_of_life_boat.svg

Camargo, R. S. (Creator). (2008). *Game of life animated glider* [GIF Image], Retrieved July 8, 2012, from http://en.wikipedia.org/wiki/File:Game_of_life_animated_glider.gif

Camargo, R. S. (Creator). (2008). *Game of life animated LWSS* [GIF Image], Retrieved July 8, 2012, from http://en.wikipedia.org/wiki/File:Game_of_life_animated_LWSS.gif

Gardner, M. (October 1970) *Mathematical Games - The fantastic combinations of John Conway's new solitaire game "life"* http://web.archive.org/web/20090603015231/http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm

Hardnett, C. R. (2011) *Programming Like a Pro For Teens.* Boston MA: Course Technology PTR.

Horsley, S. (1772). The Sieve of Eratosthenes. Being an Account of His Method of Finding All the Prime Numbers, by the Rev. Samuel Horsley, F. R. S. *Philosophical Transactions, 62,* 327-347.

Hughes, C. et al. (2012) *Project Euler – Problems*. Retrieved from http://projecteuler.net/problems

Janssen, M. A., and Ostrom, E. (2006). Empirically based, agent-based models. *Ecology and Society* 11(2): 37. Retrieved from http://www.ecologyandsociety.org/vol11/iss2/art37/

Koenig, A., & Moo, B. E. (2000) *Accelerated C++: Practical Programming by Example.* Boston, MA: Addison-Wesley.

Lipa, C. (n.d.) *Cornell Math Explorers' Club – Chaos and Fractals – Conway's Game of Life* http://www.math.cornell.edu/~lipa/mec/lesson6.html

von Neumann, J. (1948, September). Pauling, L. The General and Logical Theory of Automata. *Cerebral Mechanisms in Behavior.* Symposium conducted at the meeting of the Hixon Symposium, Pasadena, CA.

Wojtowicz, M. (September 15, 2001) *Cellular Automata Rules Lexicon – Family: Life* http://www.mirekw.com/ca/rullex_life.html

Appendix A: Basics of C++ Programming Language

A.1 Programming Concepts

In the short amount of time we had to learn computer programming, we learned several different programming concepts. Here is a list of some of the concepts that we learned:

- Data Types
  Data types describe the different data structures and the operations performed using these data structures (Koenig and Moo, 2000, p. 6). They store different kinds of information depending on the exact type. Some examples of data types are int, double, bool, char, and vector. Type int represents positive or negative whole numbers. Type double represents positive and negative floating-point numbers. Type bool represents Boolean variables that are either true or false, which can be represented by 1 or 0, respectively. Type char represents single alphanumeric or symbols characters. (Hardnett, 2011, pp. 67-68) A container type stores a collection of values. (Koenig and Moo, 2000, pp. 42)
- Conditions
  Condition statements allow the program to execute differently depending on a specified condition. One type of condition statement is the if-statement. If the condition in the if-statement is true, the contents of the if-statement are executed. If the condition in the if-statement is false, the contents of the else-statement are executed (Hardnett, 2011, pp. 32-33).
- Functions
  A function allows the program to be divided into modules, with each module having its own purpose. Each of these modules or functions can be called by other parts of the program with specified inputs, and each function can produce one or more outputs. Each function has a unique name. One of the key functions in C++ is named *main* and is the first function executed in a C++ program. (Koenig and Moo, 2000, 2).
  - Passing Parameters
    Parameters of a main function can be passed to a separate function in two ways: by reference and by value. Passing by reference allows a function to change the value of the parameter with the effect in place even after the function is complete. On the other hand, passing by value creates a copy of the input parameter, and any changes to the input variable would cease to exist once the function completes.
- Loops

A loop is a structure that contains a conditional statement and a body statement. The loop executes the body of the loop repeatedly while the condition is satisfied. The three types of loops studied are while-loops, do-while-loops, and for-loops. For loops are the most complex because they allow for an initialization, a condition, and an incremental statement all in the structure of the loop. (Hardnett, 2011, pp. 84-85, 118-119)

- Generating Random Numbers
  Random Numbers can be generated by first initializing the random numbers by inputting a number into the statement, srand(). The random numbers will then be generated. To access the numbers, the statement, drand(), must be used. (Hardnett, 2011, pp. 157-158)
- Logical Operators
  Logical operators are used in conditional statements to test for more than one condition. The Boolean true and false statements usually indicate if the entire conditional statement has been satisfied. Some examples of logical operators are the logical-or and logical-and operators. The logical-or operator yields true if either of the conditions in the conditional statement have been satisfied, and it is represented by the || symbol. The logical-and operator yields true if both of the conditions in the conditional statement have been satisfied. (Koenig and Moo, 2000, pp. 24-26)
- Vector Container Type
  The vector is a container type that stores a sequence of values of a given type. It can grow to accommodate additional values and allows each value to be accessed easily. (Koenig and Moo, 2000, pp. 41)

A.2 Programming Examples

In order to reinforce and solidify our knowledge of the basic programming concepts, we solved a series of increasingly difficult mathematical problems on the website, Project Euler (Hughes, 2001). Each problem that we were given to solve focused on a different concept.

A.2.1 Problem 1

Problem 1 of Project Euler required us to create a program that adds all the natural numbers below one thousand that are multiples of 3 or 5 (Hughes, 2001, Retrieved from http://projecteuler.net/problem=1). To solve, a for loop contained iterations from 1 to 1000. Each iteration of the loop checked whether or not each number was divisible by 3 or 5. If it was divisible, that value was added to a variable whose initial value was 0. If it wasn't divisible, then the program continued on to the next iteration.

```
1 #include <iostream>
2 using namespace std;
```

```
int main ()
{
  int i;
  int n = 1000;
  int sum = 0;
  for (i=0; i<1000; i++)
    if ((i%3==0) || (i%5==0))
      sum+=i;
  cout << sum << " is the sum of all multiples of
                  3 or 5 below 1000." << endl;
  return 0;
}
```

Fig. A.1. A solution to problem 1 of Project Euler using the programming language C++.

The concepts used in this problem included a for-loop, using a conditional statement with two conditions using the "||" logical-or operator, and printing out a string and a value.

A.2.2 Problem 2

Problem 2 of Project Euler required us to create a program that finds the sum of the even-valued Fibonacci numbers that do not exceed 4,000,000 (Hughes, 2001, Retrieved from http://projecteuler.net/problem=2). In the main function, a variable was initialized whose value would be defined by the user. A new function was created to act upon that user-defined variable. The function found the Fibonacci numbers and added them together. Three different variables represented the three consequent Fibonacci numbers. First, the values of the first two variables were added together and their sum was assigned to the third variable. Then, the value of the first number was changed to the value of the second number and then the value of the second number was changed to the value of the third variable. These formulas were put into a loop continuing while the third variable was less than a fourth variable defined in the main function, which was 4,000,000. In the loop, a conditional statement checked if the third variable was even and below 4,000,000 and if so, that value was added to another variable whose original value was 0, and then after the loop ended, the value of that fifth variable was returned to be the result of the function.

```
#include <iostream>
using namespace std;
int fib(int n)
{
  int sum, f1, f2, f;
  sum = 0;
  f1 = 0;
```

```
  f2 = 1;
  while (f<=n)
     {
        f = f1 + f2;
        f1 = f2;
        f2 = f;
        if ((f%2==0) && (f<=n))
          {
            sum+=f;
          }
      }
  return sum;
}
int main ()
{
  int n = 4000000;
  cout << "The sum of the even-valued fibonacci
           numbers whose values don't exceed
           4000000 is " << fib(n) << endl;
  return 0;
}
```

Fig. A.2. A solution to problem 2 of Project Euler using the programming language C++.

The concepts used in this problem included passing a parameter by copying, using a while loop, using a function separate from the main function, using a conditional statement with two conditions using the "&&" logical-and operator, and printing out a value.

B.3 Problem 5

Problem 5 of Project Euler required us to create a program that finds the smallest number divisible by each number from 1 to 20 (Hughes, 2001, Retrieved from http://projecteuler.net/problem=5). To solve this problem, a function computed and returned the greatest common factor of two numbers whose values were defined in the main function. In the main function, two variables were initialized. Then, a loop started when one variable was equal to 2 and continued until this variable's value was 20. In the loop, the least common multiple of two variables was computed, and the second variable was reassigned as the value of the least common multiple. At the end, the value of the second variable was printed out.

```
#include <iostream>

using namespace std;

int gcf(int i, int j)
{
```

```
   if (j==0)
     return i;
   else
     return gcf(j, i%j);
}

int main()
{
  int i, k = 1;
  for (i=2; i<20; i++)
    {
      k = (i*k)/(gcf(k, i));
    }
  cout << k << endl;
  return 0;
}
```

Fig. A.3. A solution to problem 5 of Project Euler using the programming language C++.

The concepts used in this problem included passing parameters by copying, using a-for loop, and using a function separate from the main function.

B.4 Problem 6

Problem 6 of Project Euler required us to create a program that finds the difference between the sum of the squares of the first 100 natural numbers and the square of their sum (Hughes, 2001, Retrieved from http://projecteuler.net/problem=6). The solution of the problem is as follows. In the main function, a for loop was used to go through all natural numbers from 1 to 100, and in each iteration, the square of the number would be added to the sum of squares. Then, another for loop was made to go through all natural numbers from 1 to 100 and add each number to the number in the previous iteration in order to find the sum of the numbers from 1 to 100. Then the sum was multiplied by itself to find the square of the sum. Then the sum of the squares was subtracted from the square of the sums, and the difference was printed out.

```
#include <iostream>

using namespace std;

int main ()
{
  int i, n, sum_of_sq = 0;
  int sq_of_sum = 0;
  int j = 0;
```

```
  int k = 1;
  for (i=1; i<=100; i++)
    {
      k = i*i;
      sum_of_sq+=k;
    }
  for (n=1; n<=100; n++)
    {
      j+=n;
    }
  sq_of_sum = j*j;

  cout << sq_of_sum << " - " << sum_of_sq <<
    " = " << sq_of_sum-sum_of_sq << endl;
  return 0;
}
```

Fig. A.4. A solution to problem 6 of Project Euler using the programming language C++.

The most critical concept used in this problem was the for-loop.

B.5 Problem 7

Problem 7 of Project Euler required us to create a program that finds the 10,001st prime number (Hughes, 2001, Retrieved from http://projecteuler.net/problem=7). The following solution to the problem was based on the algorithm known as the Sieve of Eratosthenes (Horsley, 1772). A function with the void data type as output was created. In the function, a vector container type over bool basic data type was created to store a Boolean flag for each number whose length would be determined by the user in the main function. The element of the vector with index *i* determined whether *i* was a prime. Initially, the flag for 0 and 1 was set to false, and all remaining flags were set to true. Then the nested loop was used to check the list of numbers in ascending order, one at a time, and for each number with the flag equal to true, flags for all multiples of this number (except for the number itself) were set to false (these could no longer be primes). Another for-loop was created to check every number in the list once the first loop concluded, count the number of numbers that were marked as true (all of these were primes), and print out the 10,001st number marked as true. In the main function, the user defined the size of the list, and then the void function was brought to use that number as the size of the list. The code assumed that the user would define a sufficiently large number to discover the actual 10,001st prime.

```
#include <iostream>
#include <vector>

using namespace std;

```

```
 6  void sieve(int n)
 7  {
 8    int i, j, count = 0;
 9    vector<bool> is_prime(n+1);
10    is_prime[0] = false;
11    is_prime[1] = false;
12
13    for (i=2; i<=n; i++)
14      is_prime[i] = true;
15
16    for (i=2; i<=n; i++)
17      if (is_prime[i])
18        for (j=2*i; j<=n; j+=i)
19          is_prime[j] = false;
20
21    for (i=0; i<=n; i++)
22      if (is_prime[i])
23        {
24        count++;
25        if (count == 10001)
26          {
27          cout << i << " " << count << endl;
28          }
29        }
30  }
31
32  int main()
33  {
34    int n;
35    cin >> n;
36    sieve(n);
37    return 0;
38  }
```

Fig. A.5. A solution to problem 7 of Project Euler using the programming language C++.

B.6 Problem 9

Problem 9 of Project Euler required us to create a program that finds the Pythagorean triplet whose sum is 1000 (Hughes, 2002, Retrieved from http://projecteuler.net/problem=9). A Pythagorean triplet is a set of three numbers in which the sum of the squares of the first two numbers is equal to the square of the third number. The solution to the problem is as follows. In the main function, a nested loop was created that checked the values of the variables so that the target Pythagorean triplet would be found. In the loop, a conditional statement made sure that the largest variable in the triplet was still larger than the other two variables while the sum of the variables was still below 1000. Then another conditional

statement within the previous condition checked if the triplet was a Pythagorean triplet, and then printed the value if it actually was a Pythagorean triplet.

```
#include <iostream>

using namespace std;

int main ()
{
  int a, b, c;
  for (a=1; a<=998; a++)
    {
      for (b=a+1; a+b<1000; b++)
        {
          int c = 1000 - a - b;
          if ((a*a) + (b*b) == (c*c))
            cout << a << " * " << b << " * "
              << c << " = " << a*b*c<< endl;
        }
    }
  return 0;
}
```

Fig. A.6. A solution to problem 9 of Project Euler using the programming language C++.

Acknowledgments

I would like to give an enormous thanks to Dr. Martin Pelikan of the Missouri Estimation of Distribution Algorithms Laboratory (MEDAL) in the Computer Science department at University of Missouri – St. Louis for all of the mentoring he performed and the help he gave to us to make this project successful. A very short period of two to three weeks was given to him to teach us the basic programming concepts and skills needed to create a code for the Game of Life, and he was successful in teaching us all of those concepts within that time period. He put in much effort and patience to help us write the code, making many corrections to our work along the way. He has been guiding us through the writing of this research paper and been informing and advising us on many of our mistakes in the paper, even with his very busy schedule. Dr. Pelikan has definitely made the Students And Teachers As Research Scientists (STARS) program the successful experience it was designed to be. I would also like to thank Mr. Michael Hope for advising us on how to correctly format and write this paper. Martin Pelikan was supported by the National Science Foundation under grants ECS-0547013 and IIS-1115352. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.