COSC 420: BIOLOGICALLY-INSPIRED COMPUTATION

PROJECT 3 - "HOPFIELD NETWORK"

CLARA NGUYEN

TABLE OF CONTENTS

1. SYNOPSIS

In this project, I explored how Hopfield Networks work, and go over how it works based on the experiments run on it. In most of the simulations done here, the Hopfield network consists of 50 patterns that have 100 neurons in each pattern. The goal is to get information by running the simulator multiple times. Then the data of each simulation is averaged. Most simulations are run 50 times before being averaged. In other cases (to determine different behaviour), the simulator may be run more times, upwards to absurd values.

2. DEVELOPMENT

The simulator was written in C++. The simulator was written with an object-oriented aspect in mind. As such, an entire object was created for a single run of a simulation. The object is configurable and expandable, which allows us to tweak it to get more interesting results for researching how Hopfield Networks work. The files are stored in the “src” directory. However, the makefile is in the root directory of the project. To compile, simply type “make” in the root directory of the project, and the makefile will go into the “src” directory and compile the executable itself. To run the simulator, supply the following arguments:

```
./hopfield patterns neurons simulations
```

For example, the default configuration consists of 50 patterns, 100 neurons, and 50 simulations. Therefore, the following must be passed into the program:

```
./hopfield 50 100 50
```

The program will output a valid CSV file containing information about the “p”, the “Fraction of Unstable Imprints”, and the “Stable Imprints”. Pipe the information into a file, and then you can generate a graph with an external application such as Microsoft Excel.

3. HOW IT WORKS

```
initialize data structures for keeping statistics
for i from 0 to number of runs
  generatePatterns()
  for p from 1 to 50
    imprintPatterns(p)
    testPatterns(p)
compute averages over number of runs
normalize data for basins of attraction (527 students)
write data file
```

Figure 1. *Pseudo Code for the Simulator.*

The program's structure was written with pseudo code from Figure 1 in mind. The program abides by a few mathematical equations that were given to us in a lab writeup. It sets up a number of patterns, that consist of multiple numbers of neurons. Initially, the states of the neurons are set to be either -1 or 1 randomly.

3.1. IMPRINTING PATTERNS

The imprintPatterns function from the pseudo code is where the weights of neurons are computed. It will store the weights in a 2D array of equal width and height. The size of the 2D array is determined by the number of neurons in a pattern (e.g., a pattern with 100 neurons will have a 100x100 array of weights). The equation in Figure 2 will determine the weight at any point in a specific pattern *p*. The weight of *i* and *j* only have a value if they are not the same, as the weight of a neuron to itself is 0.

$$w_{ij} = \begin{cases} \frac{1}{N}\sum_{k=1}^{p} s_i s_j \,, & i \neq j \\ 0, & i = j \end{cases}$$

Figure 2. *Equation for computing weights at i and j in a specific pattern p.*

3.2. TESTING PATTERNS FOR STABILITY

The testPatterns function from the pseudo code is where the simulator will test whether or not a pattern of neurons is stable or not. The procedure for doing this involves computing h, which requires weights of each neuron. The equation in Figure 3 shows how to compute h_i, which is used in the formula featured in Figure 4 to generate a new state. The function will do this for every neuron in the pattern. If *any* neuron does not match its new state, the pattern is considered to be unstable. Otherwise, it is stable.

$$h_i = \sum_{j=1}^{N} w_{ij} s_j$$

Figure 3. *Equation for computing* h_i.

$$s'_i = \sigma(h_i)$$

$$\sigma(h_i) = \begin{cases} -1, & h_i < 0 \\ +1, & h_i \geq 0 \end{cases}$$

Figure 4. *Equation for generating the new state for stability comparison.*

The simulator takes all of the values of h, sums them up, and uses that to tell whether or not the entire pattern is stable.

4. EXPERIMENTS (UNDERGRAD PORTION)

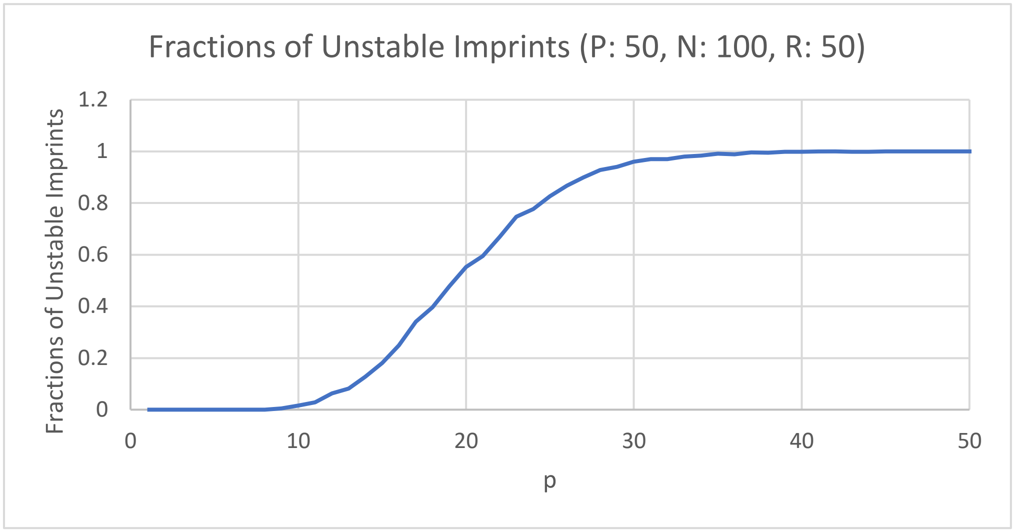
The program is configurable to allow variety in testing. We can do the following:

- Configure the number of neurons in each pattern
- Configure the number of patterns in the simulation
- Configure the number of times the test is run and averaged.

The experiments conducted put the simulator under various stress levels ranging from very lightweight to very brutal. As such, I have labelled the experiments by number, specifying the configuration used in the simulator where **P** is the number of patterns, **N** is the number of neurons in each pattern, and **R** is the number of times the program repeated the simulation.

4.1. Experiment 1 (P: 50, N: 100, R: 50)

This experiment is the default configuration of the simulator and is the one used in the Project writeup.



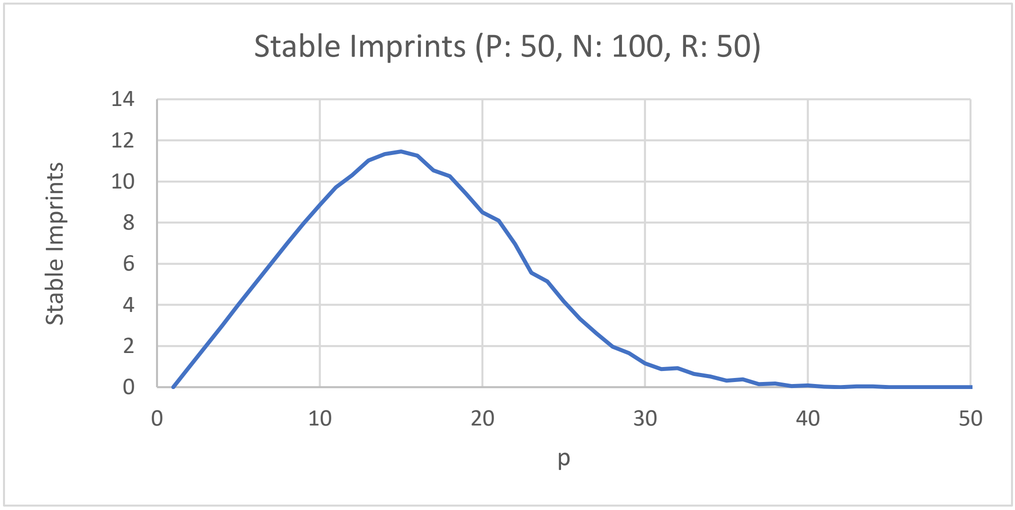
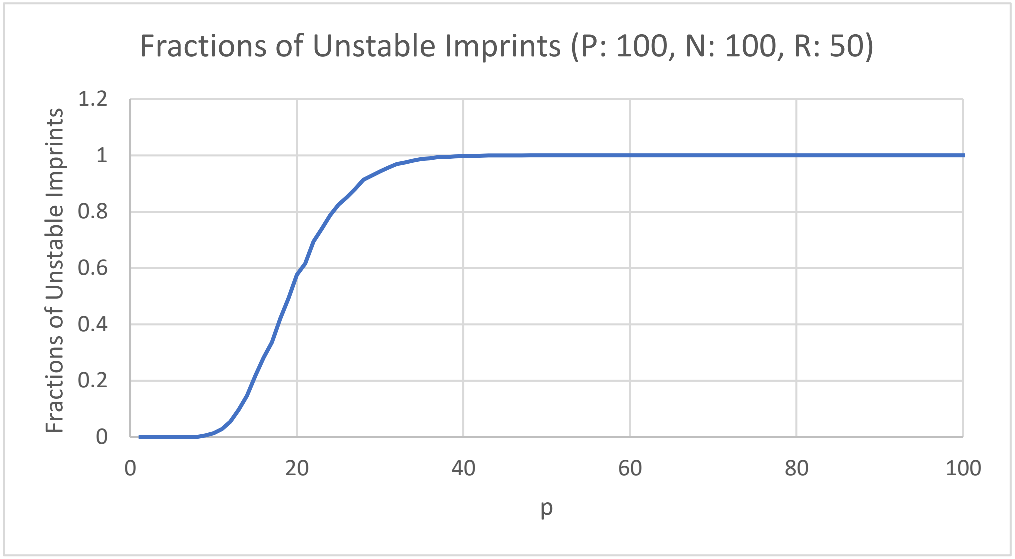


Figure 5 (Top) and 6 (Bottom). *Results of Experiment 1.*

This is an expected result, matching the results shown in the project writeup.

4.2. Experiment 2 (P: 100, N: 100, R: 50)

After Experiment 1, I decided to experiment with the values to see what happens when we have more patterns. For this experiment, we will simply double the number of patterns.



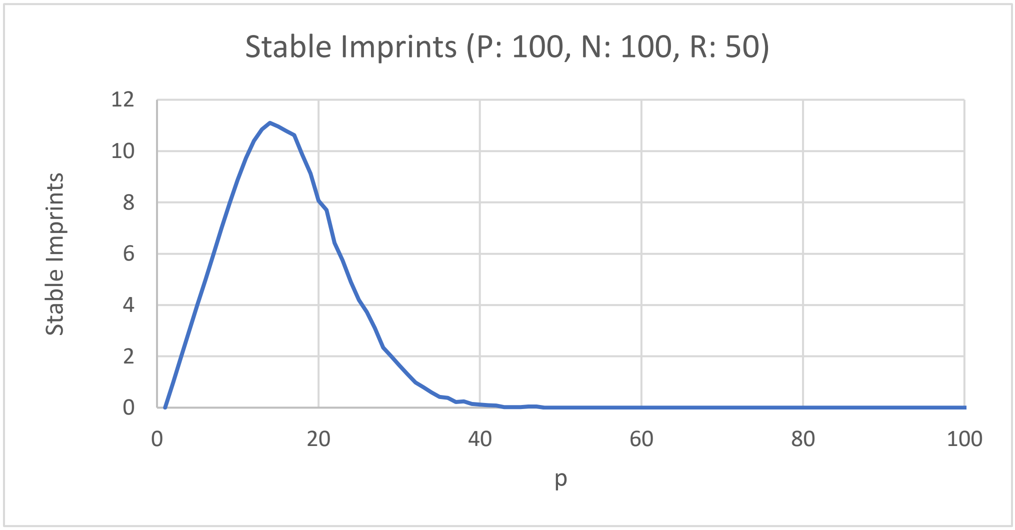
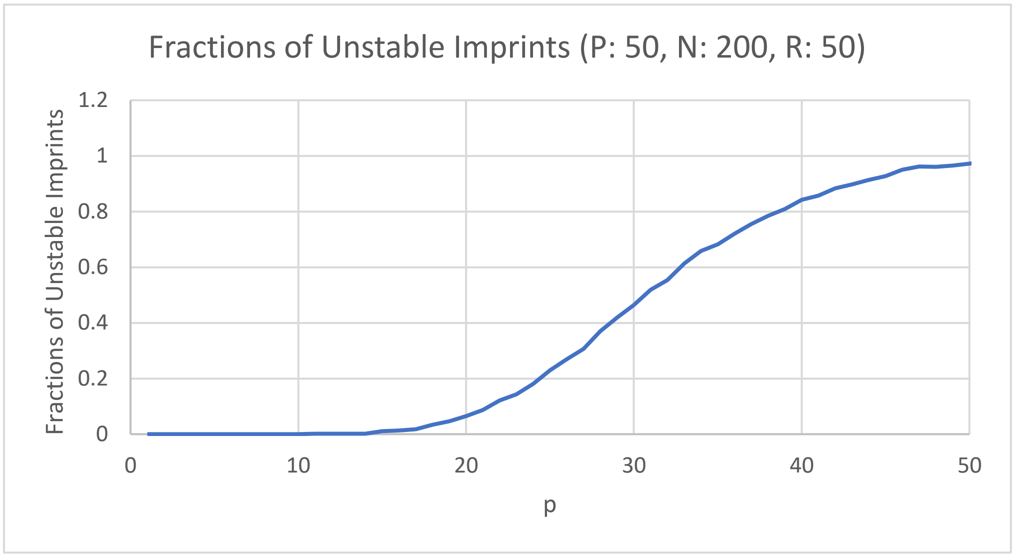


Figure 7 (Top) and 8 (Bottom). *Results of Experiment 2.*

This experiment ended up showing us that no data beyond around the same number of patterns as shown in Experiment 1. This is an expected result, as p goes beyond 30-35, the fraction of unstable imprints becomes 1 and mostly stays there throughout the remainder of the simulation. As for the Stable Imprints, it goes down to 0 when p is 46 and stays there for the remainder of the simulation.

4.3. Experiment 3 (P: 50, N: 200, R: 50)

With the same mentality as in Experiment 2, I wanted to see how the simulator performs with twice as many neurons as the default setup in Experiment 1.



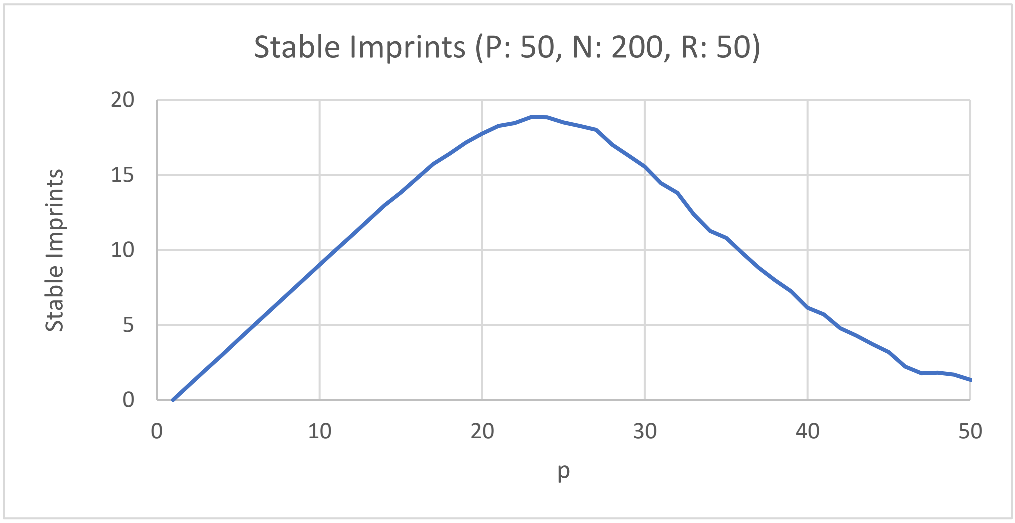
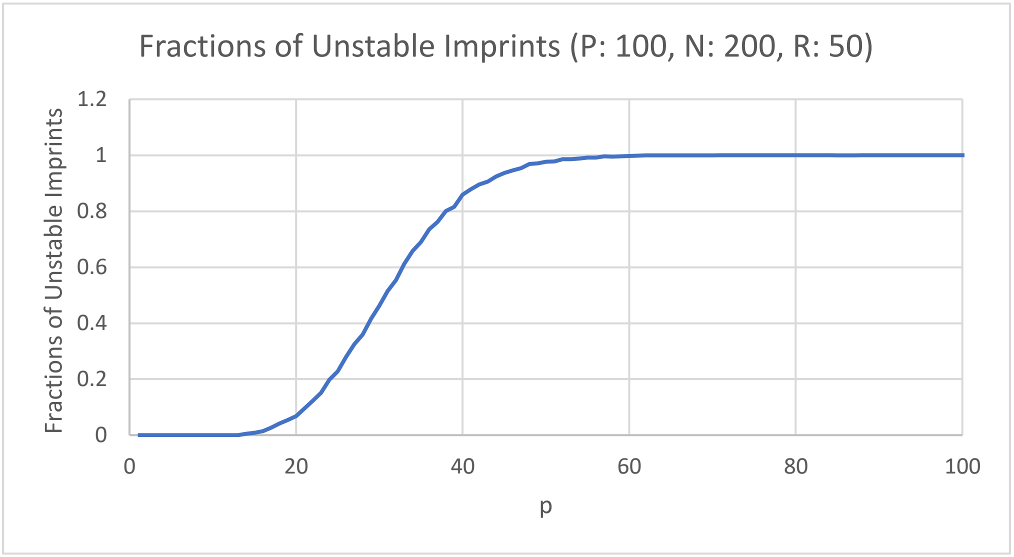


Figure 9 (Top) and 10 (Bottom). *Results of Experiment 3.*

Interestingly enough, double the neurons means that the graphs scale horizontally in proportion to the number of neurons added in. This is close to a 1-to-1 ratio. However, as we try more patterns and neurons, we will find that the slight offset in the ratio will cause the results to scale a bit less horizontally on the graph as the number of patterns grow.

4.4. Experiment 4 (P: 100, N: 200, R: 50)

With the results from Experiment 1, 2, and 3 in mind, we can take it a step further and multiply both the patterns and the neurons and see the results.



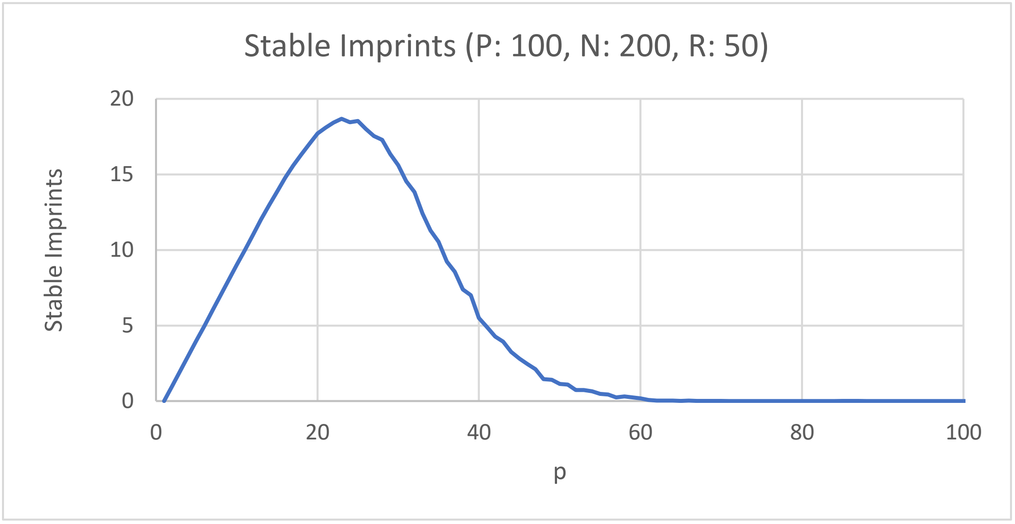
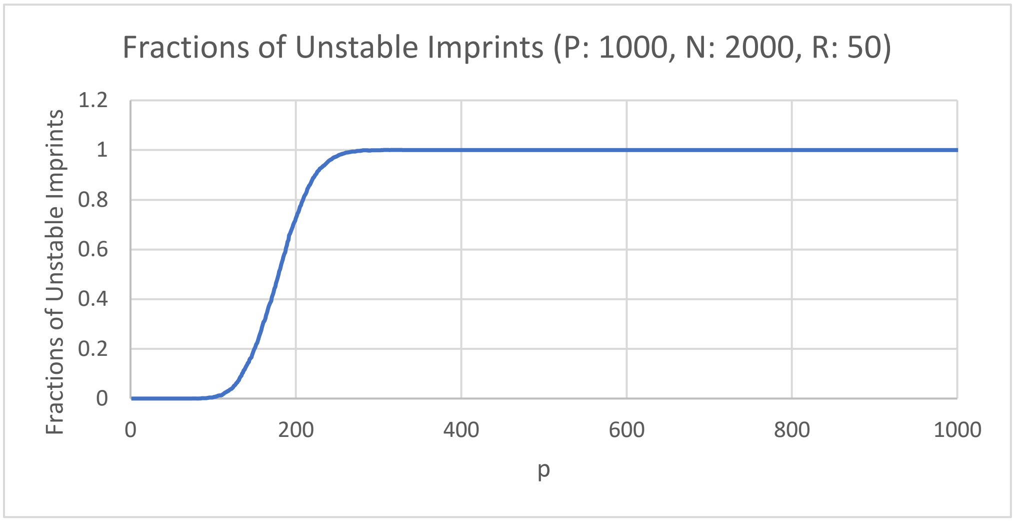


Figure 11 (Top) and 12 (Bottom). *Results of Experiment 4.*

As expected, the result looks almost the same as Experiment 1, except all of the values have nearly doubled. Though, the ratio isn't exactly 1-to-1. The peak is around 18 in Stable Imprints, meanwhile Experiment 1's peak is at around 11.6. We can further expand on this and get further data by increasing the numbers even more.

4.5. Experiment 5 (P:1000, N: 2000, R: 50)

To confirm the theory that the data scales almost linearly with the number of patterns and neurons, we can take the number of patterns and neurons a step further by multiplying them by 10.



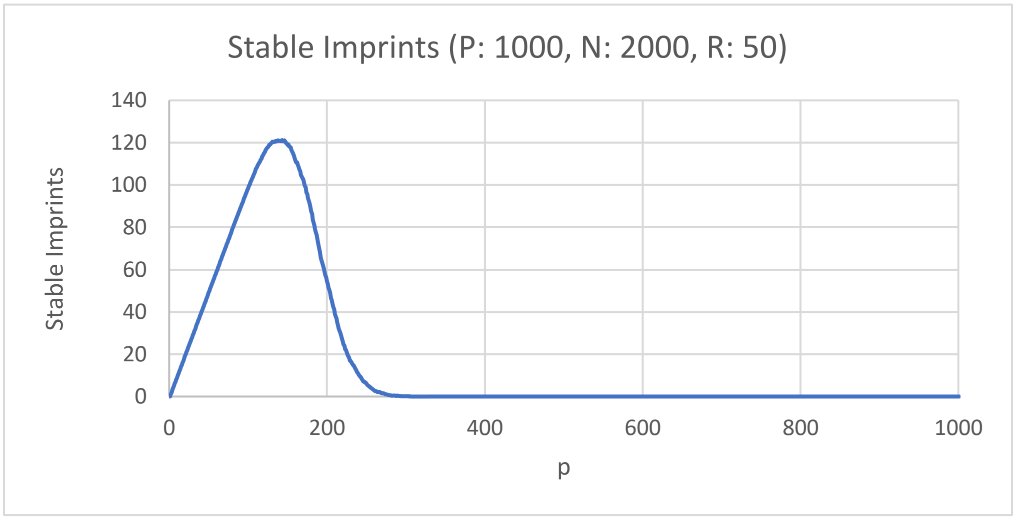


Figure 13 (Top) and 14 (Bottom). *Results of Experiment 5.*

This test was not an easy one to run. When single-threaded and compiled under “-O3”, it took around 5 hours to finish. I rewrite the simulator with a parallel programming library known as OpenMPI in order to make the test run much faster and span multiple machines. With OpenMPI, it took under 45 minutes.

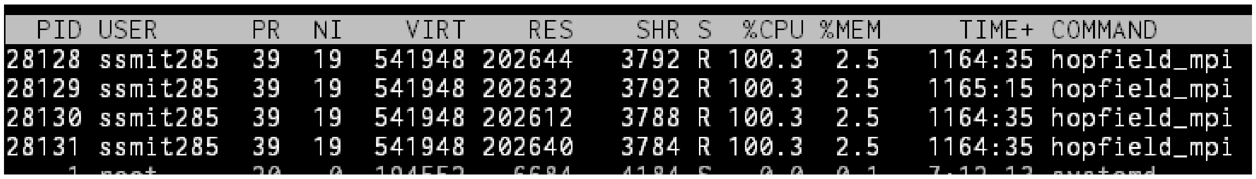
4.6. Experiment 6 (P: 5000, N: 10000, R: 50)

4.6.1. The Problem

This brutal test put the simulation to the limit in an effort of showing how accurate the data we are receiving is. The system requirements to even run this test required around 1.5865 GB of memory per machine (79.325 GB of RAM total), so I resorted to my OpenMPI implementation that I made specifically for Experiment 5. It allowed me split the job up into all of the CPU cores on the machine. However, that would only give me a 4-8x speed boost in computation. How can we speed this up? It turns out, we can take this level of multithreading to an entirely new level by gathering other machines in a cluster to help perform the computation.

4.6.2. The Solution

Nuke Hydra.



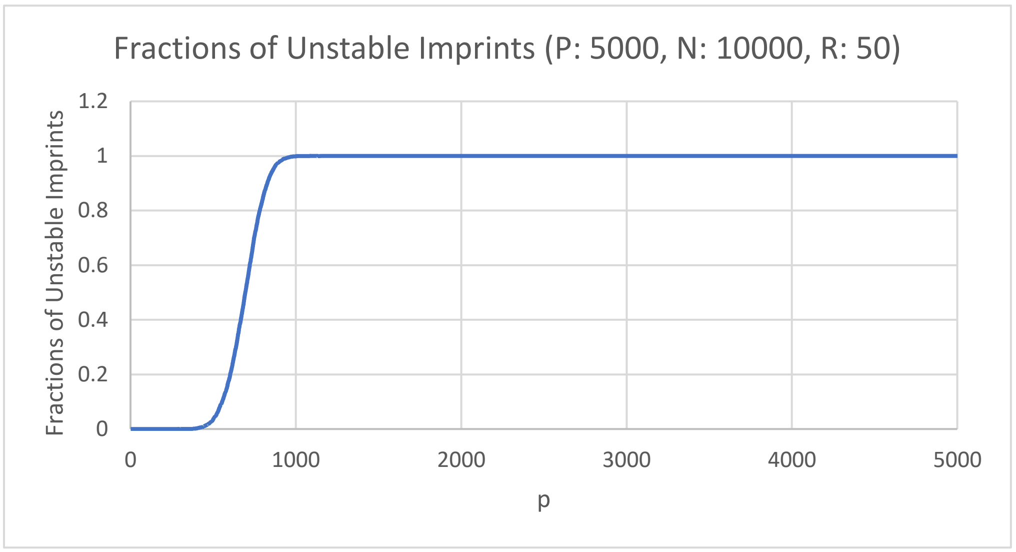
```
  PID USER       PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
28128 ssmit285   39  19  541948 202644   3792 R 100.3  2.5   1164:35 hopfield_mpi
28129 ssmit285   39  19  541948 202632   3792 R 100.3  2.5   1165:15 hopfield_mpi
28130 ssmit285   39  19  541948 202612   3788 R 100.3  2.5   1164:35 hopfield_mpi
28131 ssmit285   39  19  541948 202640   3784 R 100.3  2.5   1164:35 hopfield_mpi
```

Figure 15. *Output from "top", showing the simulator running 4 threads on one machine.*

In a desperate need of computational power, I resorted to using all 31 of the Hydra and Tesla computers and unleashing 4 threads on each. This totaled to be 248 threads all computing values for the simulation. This was to account for if any of the machines shut down mid-simulation. 4 of the Tesla machines shut off mid-simulation. However, I only needed 50 of them machines to complete the simulation so I was good.

On a single core of a single machine, the process would have taken around half a month took less than a day to compute.

4.6.3. The Experiment



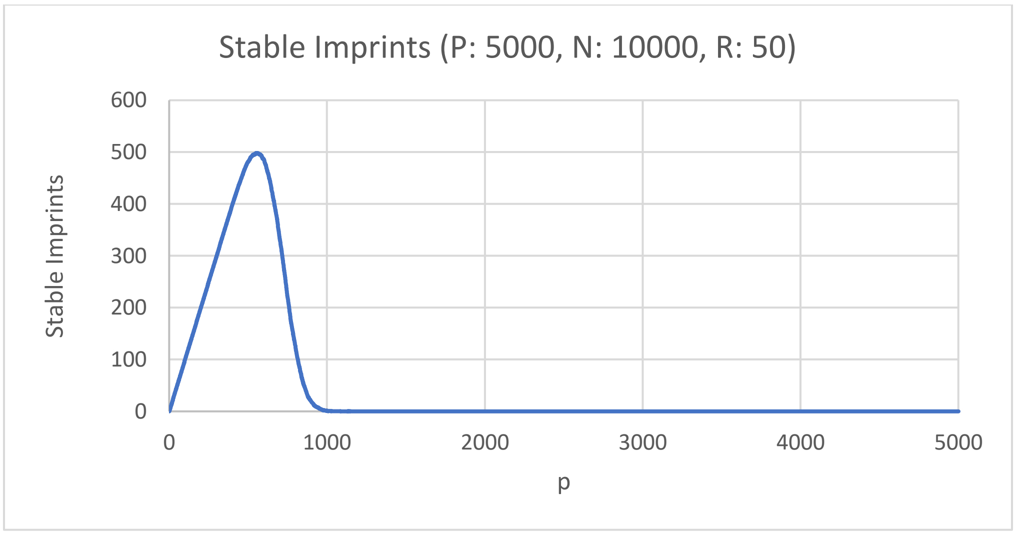


Figure 16 (Top) and 17 (Bottom). *Results of Experiment 6.*

The graph slope is getting more steep as we increment the number of patterns. It is really becoming apparent as we keep multiplying the neurons and patterns. The results of this experiment, despite the amount of time it took to compute the values, is unsurprising. Though the data generated is extremely precise, as we iterated through 100x the original guidelines stated for the Hopfield Network.

5. Observations (Undergrad Portion)

5.1. General

After a certain number of simulations, the fraction of unstable imprints will *always* go to 1. The only times that it doesn't is if you cut the simulation off at an earlier p value. Otherwise, it will eventually go to 1. The opposite is true for the number of Stable Imprints. It will eventually go down to 0, no matter the parameters passed into the simulator.

5.2. The theory for brutal testing

For the first part of the assignment, a lot of data was discovered following the experiments. The main purpose of running the brutal experiments was simple... I theorized that the precision in the graphs would increase as there were more patterns and neurons. This ended up being the case, however, it didn't scale as an exact 1-to-1 ratio. More importantly, with this data, we can also conclude statistics for increasing any of the parameters passed into the simulator.

5.2.1. Increasing Neuron Count

As the neural size increases, more patterns are in the system are stable. So if we wanted more stable patterns, we can simply increase the number of neurons for each pattern.

5.2.2. Increasing Pattern Count

Pattern count is simply the p variable. This value doesn't affect the neural network after the number of stable imprinted statements hits 0. If the number of imprinted statements at the pattern count is not 0, the simulation is cut off early. This is shown in Experiment 3, where the simulation is cut off before the values have stabilized.

5.2.3. Increasing Repetitions

The simulator repeats the simulations (by default) 50 times, then averages their values to generate the final CSV file for an experiment. Increasing this value will result in more accurate information that is being written to the CSV file, as there will be more data to average out.

5.3. Taking it one step further

There is one challenge left. I wanted to find out the ratio of increasing the patterns to make the graphs look 1-to-1 with increasing the number of neurons. As we saw in experiments 4, 5 and 6, the pattern count should be lower by a certain value in order for the graph to look the same as in experiment 1.

5.3.1. The Magic Number

We can compute this ratio by just finding common values or by simply guessing. In the end, the value was computed via the following:

$$p_r = \frac{1001}{41 * (\frac{5000}{50})} = 0.2441463 \ldots$$

Figure 18. *Equation to compute ratio to ensure graph consistency.*

Because all of the graphs incremented in the same way, we can grab the respective p values and compute the ratio through that. In experiment 1, the fraction of unstable imprints at p = 41 was 0.99048. Meanwhile, in experiment 6, the same value was found at p = 1001. So multiplying the p value from experiment 1 by the difference in neurons, then dividing 1001 by that number will give us the ratio.

5.3.2. Experiment 7 (P: 1221, N: 10000, R: 50)

To find out if our ratio was correct, let's run one last experiment and see. First, we need to compute the number of patterns to have in the simulation.

$$P = 5000 * p_r = 5000 * \frac{1001}{41 * \left(\frac{5000}{50}\right)} = 5000 * 0.2441463 \ldots = 1220.7317 \ldots$$

Figure 19. *Equation to compute ratio to ensure graph consistency.*

Therefore, there are 1221 patterns in this simulation. Okay, ready.

<Coming soon!>

Figure 20 (Top) and 21 (Bottom). *Results of Experiment 7.*

As expected, the graphs cut off right at the very end of the simulation. And it looks like a 1-to-1 expansion of experiment 1.

5.3.3. Why bother?

This is to allow for program efficiency. Look back at the graphs for experiments 4, 5 and 6. You should notice that, beyond a certain point, it's just a flat value. This is wasted computational time. However, with the ratio we posted, we are now able to predict when the neural network will constantly be unstable. With that knowledge, we can cut the simulation off early, and guarantee that we are not losing any data. This is also why Experiment 7 was based off of Experiment 6. The values passed in are smaller, so it ended up taking less memory and was much faster to compute, as a result.

6. Experiments (Graduate Portion)

7. Observations (Graduate Portion)