Physical Modeling and Simulation Project # 1

Ioannis Michaloliakos[1, a)]
University Of Florida, Physics Department

(Dated: 30 October 2018)

In this project we examine Monte Carlo Methods to decrypt a substitution cipher in the case of short phrases. We use Markov Chain Monte Carlo (MCMC) to sample in the configuration space of possible decryption keys, and we employ various techniques to improve the accuracy of our decipher. We use one, two and three letter frequencies from a reference text (War and Peace) for our score functions and we use simulated annealing to escape from local minima. We see a significant improvement in accuracy when we include word search in our reference text as part of our scoring function. We show that even for relatively short phrases (~ 50 characters) the accuracy of our results can made to be arbitrarily close to 100%.

Keywords: Decipher , Simulated Annealing, Makov Chains, Monte Carlo, MCMC

I. INTODUCTION

During the last decade MCMC algorithms are routinely used to break substitution ciphers. In the current project we examine the case of relatively small encrypted texts where the usual MCMC methods with two letter (bigram) frequencies fail. This is because the original phrase does not necessarily maximize the bigram counts of the reference texts, or of the English language in general. We start our analysis in subsection I A by describing the basic implementation of the MCMC search, and we continue in section II with the testing of various parameters of our algorithm in order to optimize our methods. Finally, in section III we optimize the execution time of our algorithm, entirely developed in python, using the profiling method of the Spyder IDE.

A. Basic algorithm of MCMC decryption

In bigram MCMC decryption we start by an educated guess of an decryption key and we propose 2 letter swaps on the key dictionary. Then, in order to improve our initial guess, we have to find a way to compare two different keys. This is done by defining a score function which is ideally maximized at the correct phrase. This maximization can be achieved by talking into account that for a random decryption key the adjacent letters would not generally appear very frequently in the English language. Thus we can try to have our score function be one that is maximized by the combination of single letters (unigrams) that have the higher chance of appearing in the English language. This unigram search would be good enough to find correctly the most frequent few letters of the english language (space " " and possibly "t","e") but will usually fail to find more than a few characters in short phrases of about 50 characters. The next simplest thing we could do is to find the combination that maximizes the two-letter (bigram) frequencies in the English language. Assuming that each letter combination is independent, the probability of a decryption key would be the product of the probabilities of the individual bigrams contained in the decrypted phrase. Therefore, ignoring (as usual in MCMC) the overall normalization, we can define the score function to be:

$$\pi_1(x) \;=\; \prod_{i\in S} r(x(s_i), x(s_{i+1})) = \prod_{\beta_1,\beta_2} r(\beta_1,\beta_2)^{f_x(\beta_1,\beta_2)}$$

where i runs over the length of the cipher text, s_i is a symbol in the cipher text, $\beta = x(s_i)$ is the tentative deciphered letter from s_i using the key x, $r(\beta_1,\beta_2)$ is the counts of $\beta_1\beta_2$ in a large reference text and $f_x(\beta_1,\beta_2)$ is the counts of $\beta_1\beta_2$ in the tentative deciphered text. Actually, since we will have to deal with generally high counts for some frequent bigrams in the reference text, we will be using the log of $\pi_1(x)$, namely:

$$\pi(x) \;=\; \log(\pi_1(x)) = \sum_{\beta_1,\beta_2} f_x(\beta_1,\beta_2)\log(r(\beta_1,\beta_2))$$

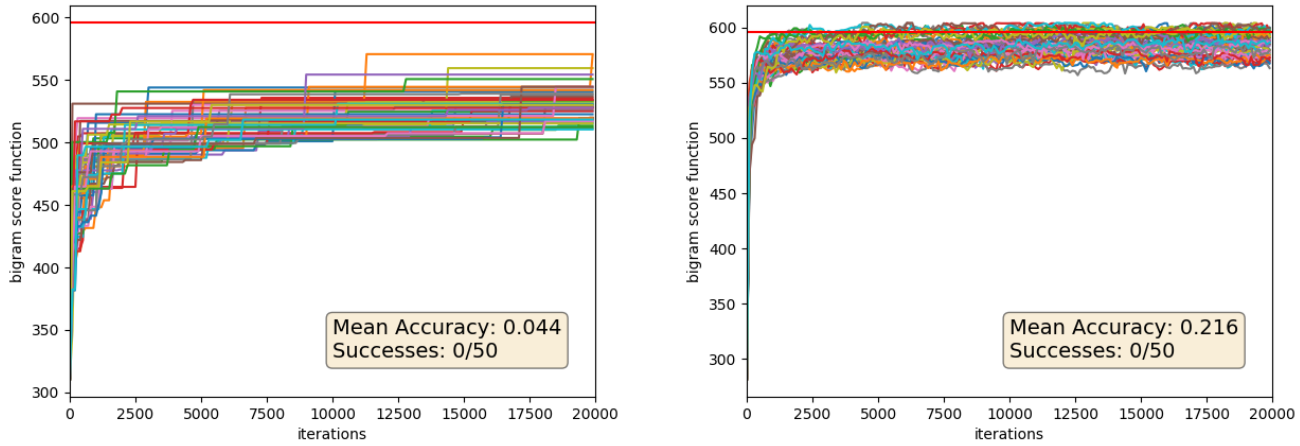
In order to deal with cases where the number of counts in the reference text is 0 for certain bigrams, we add 1 to every bigram count (but we do not do the same for counts on the deciphered text).

The algorithm procedes as follows:

1. We propose a key y from a two letter swap of our current key x
2. We accept the change if $(\pi_1(y)/\pi_1(x))^{1/T} > u$, where T is a parameter that we can think of as temperature and u is a random float betweem 0 and 1. Otherwise we reject y and keep x unchanged
3. We repeat this process either for a certain number of iterations (e.g. 10,000) or, in the case of simulated annealing, until the temperature (which falls exponentially from an initial value T_{max}) is less than a fixed value T_{min}.

In what follows we describe the methodology we followed in order to get a near 100 % accuracy for our decipher.

[a)] Electronic mail: ioannis.michalol@ufl.edu



(a) Accepting only Positive

(b) Metropolis Algorithm

FIG. 1: Metropolis Acceptance of negative changes

II. TESTING

The aim of our project is to decipher an encrypted version of the phrase

```
"the answer to life the universe and
everything is forty two"
```

Since the most common symbol in our phrase is the space (" ") we can pad our encrypted phrase at the beginning and at the end with the most common symbol in the cipher text, so essentially we start from the phrase:

```
" the answer to life the universe and
everything is forty two "
```

This helps our search since our calculations take into account more information. For even shorter phrases, where " " is not necessarily the most common symbol, we would not have had the luxury of using this trick.

We define the accuracy of our final key to be :

$$\alpha = \frac{n_C}{n_T}$$

where n_C is the number of correct distinct letters (including space " ") in the deciphered text and n_T is the total number of distinct letters in the cipher text.

We also define a successful run to be a run where our MCMC algorithm returned the correct key. We find that dispite marginal improvements, the accuracy of a bigram search is pretty low ($\sim$ 30 %), while the accuracy of trigram searches is much higher ($\sim$ 100 %).

A. Test 1 : Checking the Metropolis Acceptance Rule

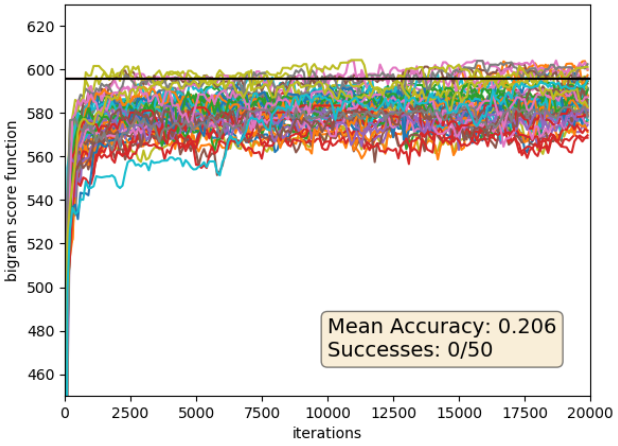
As the first test, we check the Metropolis choice of accepting proposals with lower score. We run our algorithm with and without (accepting only positive changes) the Metropolis rule 50 times with 20,000 iterations each and no annealing, starting always from the same random decryption key guess. In Table I and in Figure 1 we see that there is substantial difference between the two choices, so for all tests from now on we accept negative proposals with non-zero probability. We note that we have always unsuccessful runs, which is going to be a common theme for bigram searches. This has to do with the fact the our actual phrase does not maximize the bigram score function.

Test #1	α	# successes
Without Metropolis	0.044	0/50
With Metropolis	0.216	0/50

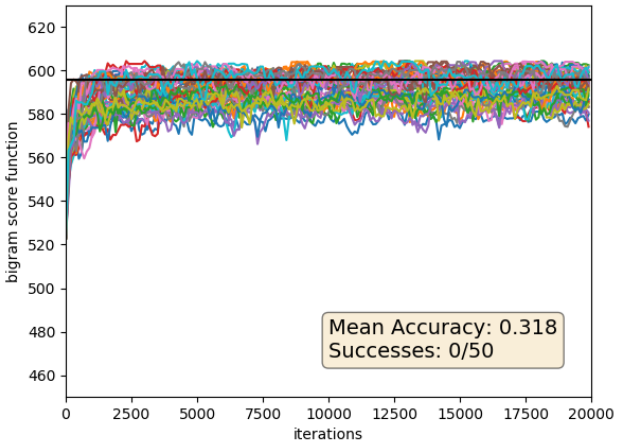
TABLE I: Accepting Only Positive Changes vs Metropolis

B. Test 2: Starting with the Unigram Guess and keeping the Best Score

We continue our tests by having our MCMC return the Key with the best score instead of the last key of the Markov Chain. Simultaneously, we start our algorithm not with a random key, but instead with the key that is



(a) Starting from a Random Key and keeping the last key



(b) Starting from a the Unigram Guess and keeping the best key

FIG. 2: Keeping the key with the best score and starting from an educated guess

favored by the one-letter frequencies. The comparison of this algorithm to an algorithm that does not implement these changes is shown in Table II and Figure 2. Again, we see significant improvement, and therefore from now on we will alway incorporate these two in our searches.

Test #2	α	# successes
w/o Best & Unigram	0.205	0/50
w/ Best & Unigram	0.317	0/50

TABLE II: Keeping the best Key and Starting from Unigram Guess

C. Test 3: Varying the number of iterations (No Annealing)

Continuing, we test the effect of varying the number of iterations in a signle run. We test our method for 5, 10, 20 and 50 thousand iterations. We have seen from the previous experiments that the score curves become flat after a few thousant iterations with little variations after that. This suggests that a few thousand iterations will be enough. Our results are summarized in Table III. Indeed, we see the gain from increased number of iterations is quit small. Since our algorithm finishes running after a couple of seconds, we decide to keep 20,000 as default.

D. Test 4: Finding the optimal T_{max} and T_{min} in Simulated Annealing

From now on we turn our interest to the case of simulated annealing, where the temperature T falls exponentially from an initial value T_{max}) is less than a fixed value T_{min}, namely:

Test #3	α	# successes
5,000 iters	0.285	0/50
10,000 iters	0.311	0/50
20,000 iters	0.3244	0/50
50,000 iters	0.3422	0/50

TABLE III: Varying the Number of Iterations

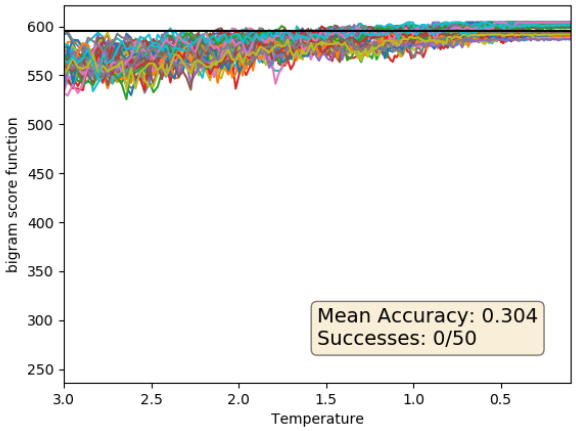
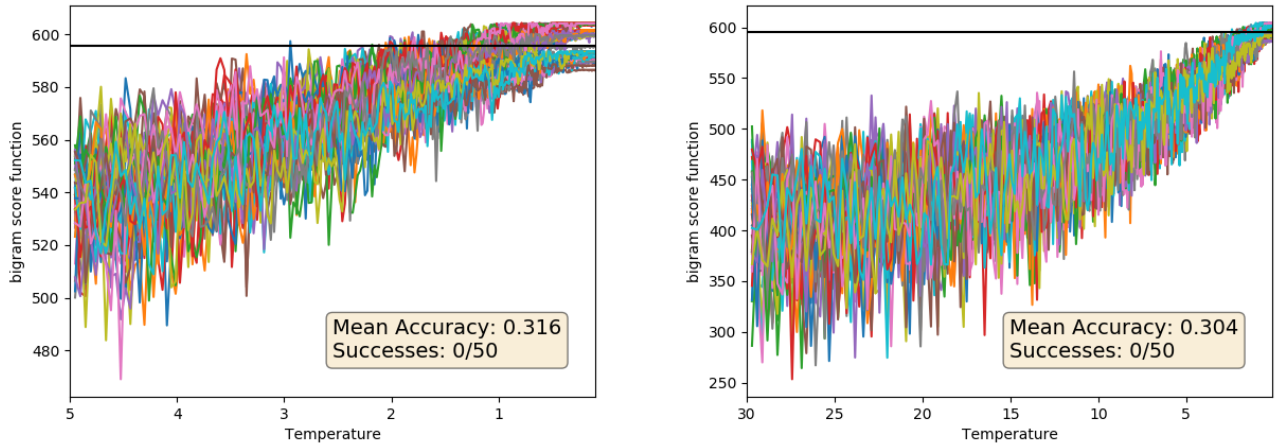


FIG. 3: Zooming in around T_{min} at Test #4

$$T = T_{max} \exp(-t/\tau)$$

where τ is the time needed for the temperature to fall by a factor of e . This temperature enters the accep-



(a) $T_{max} = 5$ (b) $T_{max} = 30$

FIG. 4: Varying the Initial Temperature

tance probability as $(\pi_1(y)/\pi_1(x))^{1/T}$ and helps us avoid getting stuck at local maxima of the score function. We first try to find the temperature maximum value of T_{min} needed for the system to reach an equilibrium. We start our chains by choosing somewhat arbitrarily $T_{max} = 30$ and $T_{min} = 0.1$. From zooming in around the region of T_{min} in Figure 3, we find that the scores do not charge significantly after $T = 0.5$. We will actually choose $T_{min} = 0.1$ from now on so that we are sure we don't stop the runs prematurely.

In Table IV and Figure 4 we see the behavior of our algorithm for two different choices of T_{max}, 30 and 5. We see that actually starting from $T = 5$ gives slightly better results. This can be explained by the large fluctuations at high frequencies which would drive the initial educated guess randomly to lower scores. Therefore, from here onward we will use $T_{max} = 5$.

Test #4	α	# successes
$T_{max} = 5$	0.315	0/50
$T_{max} = 30$	0.304	0/50

TABLE IV: Varying T_{max}

E. Test 5: Varying the Annealing Rate

In this section we try different choices of the annealing rate τ. Our different tries are listed in Table V and depicted in Figure 5. We see them as we increase τ, the accuracy increases substantially, and thus we keep the maximum τ with reasonable execution time, which is $\tau = 10^5$.

F. Test 6: Trigram Score Functions

In all previous searches we have zero successful runs. This suggests the need to go beyond bigrams, namely to score functions of (at least) trigrams. So we incorporate a trigram search into our code and define our trigram score function to be:

$$\pi_{tr}(x) = \sum_{\beta_1,\beta_2,\beta_3} f_x(\beta_1,\beta_2,\beta_3)\log(r(\beta_1,\beta_2,\beta_3))$$

We know perform a MCMC search with $T_{max} = 5$, $T_{min} = 0.1$, $\tau = 10^5$ first starting from the unigram educated guess that we have described above and then combining the unigram guess with a 20,000 iteration bigram search before we feed it into our main trigram MCMC. Our results are summarized in Table VI and depicted in Figure 6.

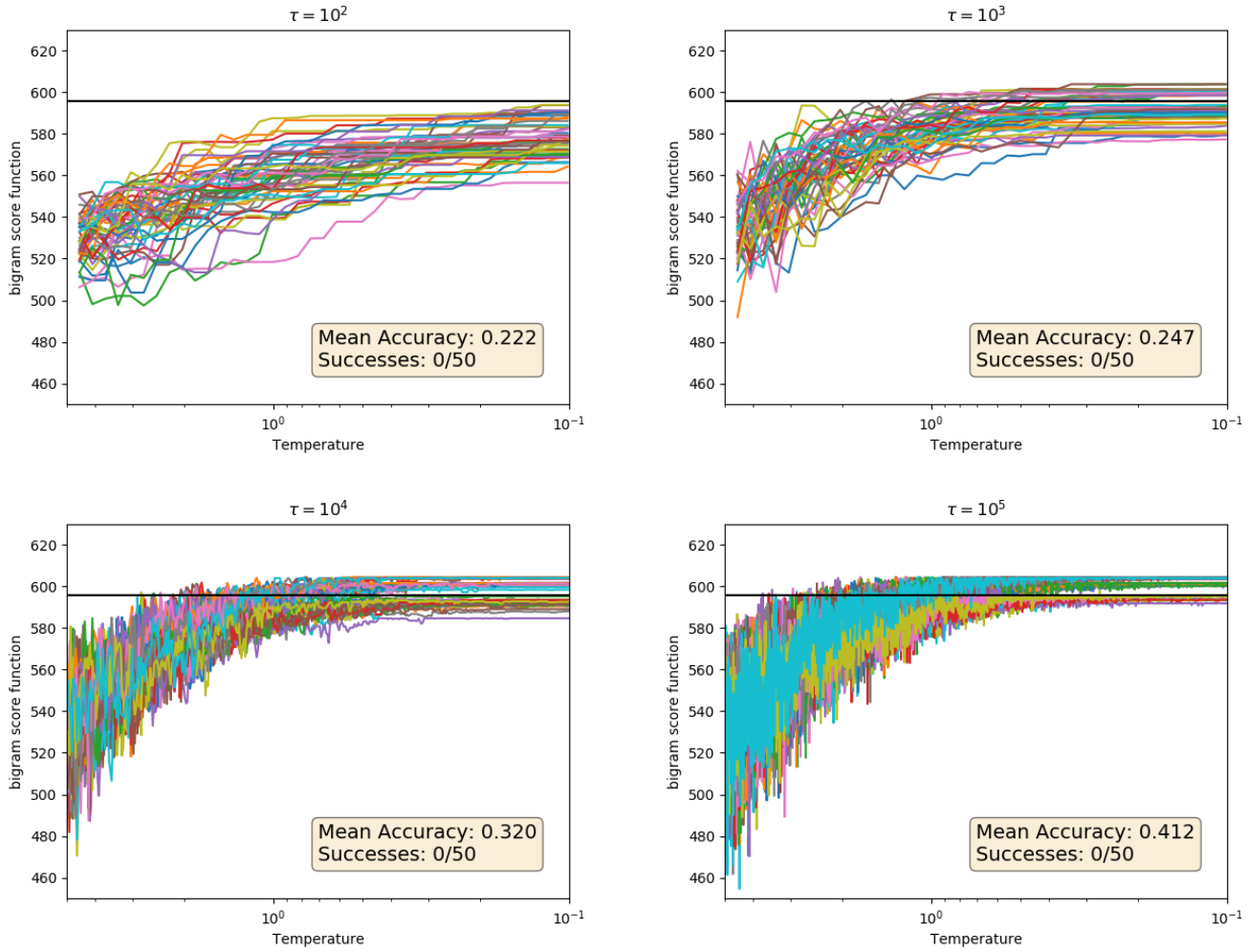


FIG. 5: Decipher Runs with Different Annealing Rates τ

Test #5	α	# successes
$\tau = 10^2$	0.222	0/50
$\tau = 10^3$	0.247	0/50
$\tau = 10^4$	0.320	0/50
$\tau = 10^5$	0.412	0/50

TABLE V: Varying the Annealing Rate τ

Tests #6-#7	α	# successes
unigram start	0.831	0/50
bigram start	0.842	0/50
bigram start $\oplus$ word search	0.973	48/50
trigram start $\oplus$ word search	0.902	41/50

TABLE VI: Trigram searches with different starting procedures and w/ or w/o Word Search in our reference text.

We note that the accuracy almost doubles and we consistently end up with decrypted phrases very close to the correct original phrase, like:

```
" the answer to fice the universe
    and everything is corty two "
```

G. Test 7: Word Search Functions

By carefully analyzing the behavior of of our trigram score function we find that the correct phrase still does not maximize the log product trigram frequency, but it is instead in the Top 3. This suggests that by incorporating a search for true English words in our algorithm, we can actually maximize the score function at the correct phrase, by essentially assigning a score of infinity at the correct phrase. So we edit our algorithm so that if it reaches a phrase that consists entirely of English words, it returns that particular key. Our results are summarized in Table VI and in Figure 7. We see that if we start from a bigram search and feed our result into a

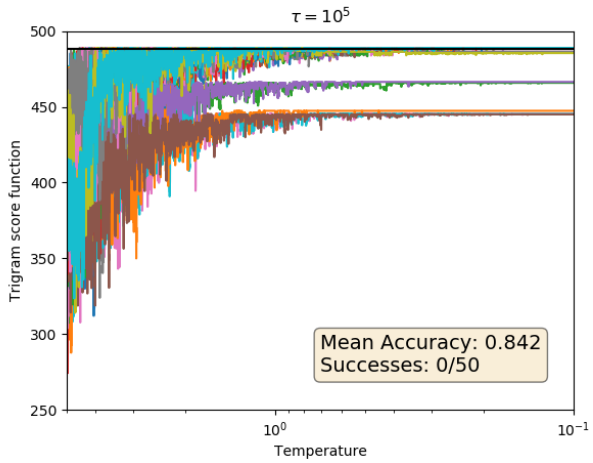


FIG. 6: Score evolution for a trigram score function

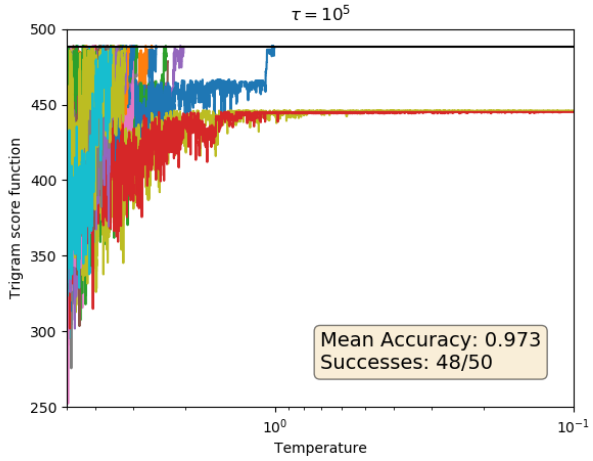


FIG. 7: Incorporating a word search in our reference text for every promising swap in the Key dictionary

trigram search we end up with an accuracy of **97.3 %** which can improved even more by increasing the annealing rate, obviously at the cost of execution speed. We conclude therefore that the combination of trigram and word searches is generally enough to decrypt phrases of $\sim$ 50 characters, but one could also try 4-letter (tetragram) frequencies to converge more quickly to the desired result.

III. PROFILING OF OUR CODE AND HIPERGATOR

In order to make sure that the searches that there no part of our algorithm that slows down unnecessarily the execution speed we use the profiling method of the Spyder Anaconda IDE. Essentially, the original version of our code computed inefficiently the score function by looping over all possible bi/tri-grams, instead of looping only over

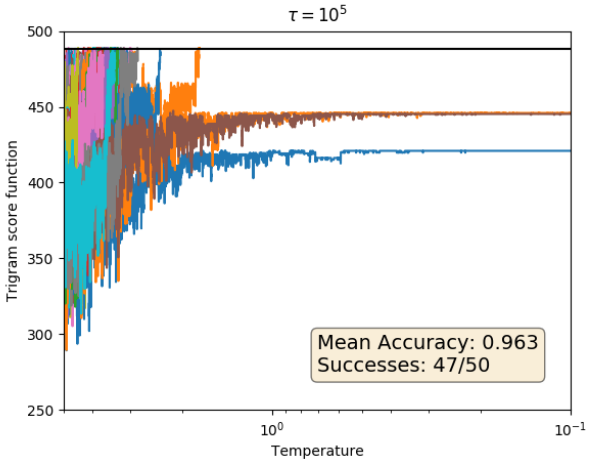


FIG. 8: Recreating the results of Test #7, using the HiPerGator cluster.

the bi/tri-grams with non-zero counts in our tentatively deciphered text. After changing this part of our code the execution time for trigrams improved by at least a factor of 500. Additionally using the hashed ordered dictionaries and count/Counter methods and objects of Python our code has more clarity and runs faster. I also tried to follow the PEP8 syntax recommendations for additional readability.

1. SLURM submission script on HiPerGator

As part of the project, we also create a submission script to the HiPerGator Computing Cluster of UF. We essentially reproduce the results of Test 7 but with 50,000 iterations for the original bigram part of the algorithm (Figure 8). In the last page of the report, we provide the SLURM script that was used to generate the job that produced Figure 8.

ACKNOWLEDGMENTS

I would like to thank the University of Florida High-Performance Computing Center for computational resources and technical support. I would also like to thank Ammar Jahin and Suzanne Rosenzweig for fruitful discussions regarding the project.

```
#!/bin/sh

#SBATCH --job-name=serial_job_test # Job name
#SBATCH --mail-type=ALL   # Mail events
#SBATCH --mail-user=ioannis.michalol@ufl.edu   # Where to send mail
###SBATCH --qos=phz5155                 # class allocation
#SBATCH --qos=phz5155-b                 # burst mode
#SBATCH --ntasks=1   # Run on a single CPU
#SBATCH --mem=2gb   # Memory limit
#SBATCH --time=00:40:00   # Time: hrs:min:sec
#SBATCH --output=serial_test_%j.out   # Output and error log

pwd; hostname; date

module load python3

echo "Running Test 7 of my Project 1 on a single CPU core"

python3 /ufrc/phz5155/ioannis.michalol/michaloliakos_project1.py

date
```