Cryptography Project

AST 4930: Computational Astrophysics

Mitchell Kelly

Code Architecture:

1. *Probability Matrix:* The probability matrix was created by loading War and Peace and calculating the occurrence of any combination of two letters. Two for loops nested inside one another accounted for each of the 676 letter combinations whereas another for loop ran through the entire text. Every hit resulted in an increase by one in each element of the matrix, and the entire matrix was divided by the sum of all elements to construct an accurate probability.

```
wp = np.load('wp.npz')
wp = wp['text'].tobytes()
wp = wp.decode("utf-8")

def prob_matrix(s):
    probability = np.zeros(shape = (26,26))
    d = list(string.ascii_lowercase)
    for i in range(0,len(d)):
        for j in range(0,len(d)):
            count = 0
            for k in range(0,len(wp)):
                if(s[k:k+2] == d[i] + d[j]):
                    count += 1
                probability[i][j] = count
    return probability

np.save('data.npz', prob_matrix(wp))
```

2. *Key Function:* Two lists containing each letter of the alphabet were introduced where one list would have its letters randomly shuffled and matched into a corresponding dictionary.

```
def key():
    random.shuffle(x)
    key = {alp[i]:x[i] for i in range(len(alp))}
    return key
```

3. *Decode Function:* In combination with other functions, the decode function took the phrase inputs and decoded them to prepare the letter combination for scoring

```
def decode(key,s):
    jumbled = ''
    for i in range(len(s)):
        convert = key[phrase[i]]
        jumbled += convert
    return jumbled
```

4. *Swap Function:* Two random integers from 0 to 25 which correspond to elements of the alphabet list were chosen to swap elements around in order to compare relative scores of combinations based on the probability matrix.

```
def swap(key):
    rand1 = np.random.randint(0,25)
    rand2 = np.random.randint(0,25)
    key2 = key.copy()
    s = key2[alp[rand1]]
    key2[alp[rand1]] = key2[alp[rand2]]
    key2[alp[rand2]] = s
    return key2
```

5. *Scoring Key Function:* An array of 676 unique pairs was created, so the scoring could be mapped to its specific pair. The numpy ravel function was imported to produce a 1-D array where the scores would be applied to their rightful pair. The logspace of the probability matrix was the distribution input of the function to calculate the score of each combination.

```
def scoring_key(distribution):
    pairs = []
    for i in range(len(alp)):
        for k in range(len(alp)):
            pairs.append(alp[i]+alp[k])
    scoring_key = {pairs[i]:np.ravel(distribution)[i] for i in range(len(pairs))}
    return scoring_key
```

6. *Scoring Function:* The scoring function used the scoring key to input the phrase for actual scoring operations. Its counter represented the scoring key's internal tier list of preferred combinations.

```
def scoring(phrase, scoring_key):
    counter = 0
    for i in range(len(phrase)-1):
        counter += scoring_key[phrase[i] + phrase[i+1]]
    return counter
```

7. *Correctness Percentage Function:* This function looked at the decoded phrase utilizing the aforementioned functions to determine its accuracy compared to the correct phrase. Every time a letter matched the correct phrase, a point would be added to a counter sum averaged by the number of letters in the correct sequence.

```
def correctness_percent(decoded):
    correct_phrase = 'jackandjillwentupthehilltofetchapaleofwater'
    correct_counter = 0
    for i in range(len(decoded)):
        if decoded[i] == correct_phrase[i]:
            correct_counter += 1
    return correct_counter*100/len(decoded)
```

8. *Annealing Function:* Used in the MCMC section of the code, the annealing function took the negative exponential of the absolute value of two scores which had a factor of one over T attached (the time constant).

```
def annealing(score1, score2, T):
    return (np.random.uniform() < np.exp(-np.abs((score2-score1))/T))
```

MCMC Process and Testing:

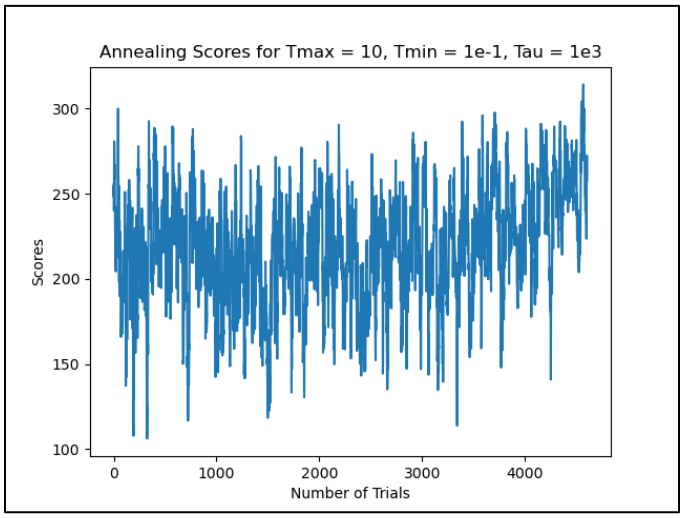
MCMC with and without annealing were developed as functions to test the correctness percentage of the decoded phrase with respect to the correct one. Only positive results were taken for the no annealing function while some negative scores were accepted in the annealing function with different rates corresponding to the tolerance allowed. Local minimums were avoided to allow for the best possible correct percentage, and the range of "temperatures" were increased by a magnitude of 100 to see the correctness percentage convergence range. The MCMC annealing function operated under a while loop that illustrated the temperature range of the "glass" needing to be cooled as

```
def MCMC_Annealing(Tmax, Tmin, tau):
    scores = []
    cipher = key()
    phrase = 'zywdynfzmbboanxjrxiaimbbxpgaxwiyrymbpgoyxal'
    scoringkey1 = scoring_key(logspace)
    t = 0
    T = Tmax
    while T>Tmin:
        t += 1
        T = Tmax * np.exp(-t/tau)
        P = scoring(decode(cipher, phrase), scoringkey1)
        swapped_cipher = swap(cipher)
        P2 = scoring(decode(swapped_cipher, phrase), scoringkey1)
        if P2>P:
            cipher = swapped_cipher
            scores.append(P2)
        elif (annealing(np.log(P), np.log(P2), T)):
            cipher = swapped_cipher
            scores.append(P2)
        else:
            scores.append(P)
    decoded = decode(cipher, phrase)
    return [decode(cipher, phrase), scores]
```

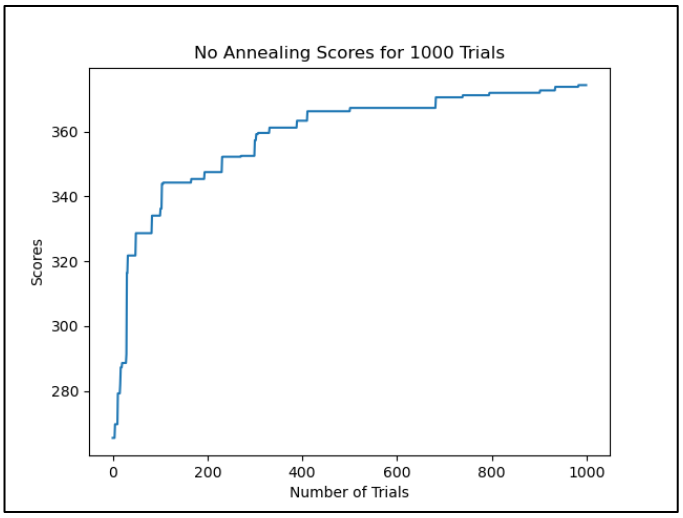
```
def MCMC_NoAnnealing(N_SIMS):
    scores = []
    cipher = key()
    phrase = 'zywdynfzmbboanxjrxiaimbbxpgaxwiyrymbpgoyxal'
    scoring_key1 = scoring_key(logspace)
    for i in range(N_SIMS):
        P = scoring(decode(cipher, phrase), scoring_key1)
        swapped_cipher = swap(cipher)
        P2 = scoring(decode(swapped_cipher, phrase), scoring_key1)
        if P2 > P:
            # keep swap
            cipher = swapped_cipher
            scores.append(P2)
        else:
            scores.append(P)
    decoded = decode(cipher, phrase)
    return decoded, scores
```

the temperature fell by one each loop. Two if statements allowed for the acceptance of good swaps and some bad ones if the choices satisfied tolerance criteria, which was the log of each possible outcome evaluated in the annealing function. The no annealing function was almost identical to the annealing function, yet the annealing if statement was eliminated to accept only positive choices of phrase decoding. Three different ranges of annealing temperature mins and maxes were analyzed for correctness with ten trials to calculate a rough estimate of accuracy based upon those parameters. The three ranges; 10 to 0.1, 100 to 0.01, and 1000 to 0.001 showed that the convergence for annealing was between 1000/100 to 0.001/0.01 because increasing the range further did not produce any significant improvement to the correctness percentage. Furthermore, increasing the number of trials in the non-annealing function showed that the number of trials did not have an impact on correctness, only altering the makeup of the results of a good or bad trial. Additionally, choosing random letters for the phrase resulted in an average of around 4 percent correctness, so it was better to use MCMC with and without annealing compared to a random draw. Finally, the highest correctness percentage, 55.8, was achieved in the largest formerly discussed temperature range, and this made sense due to the number of opportunities the decoding process had in fine-tuning its results.

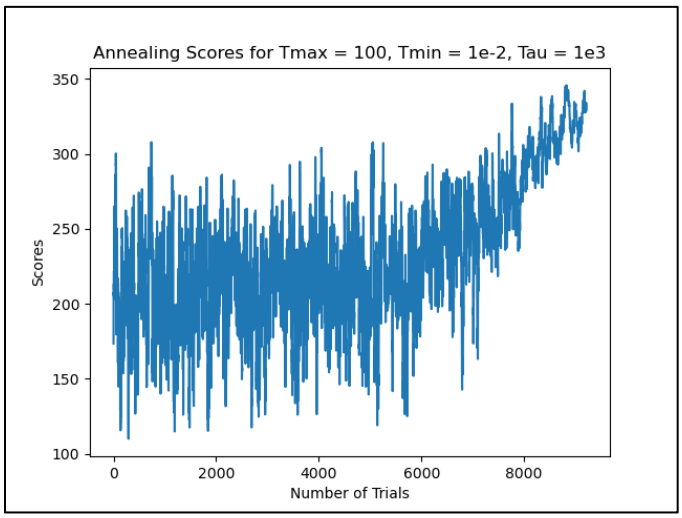
Diagnostic Plots:



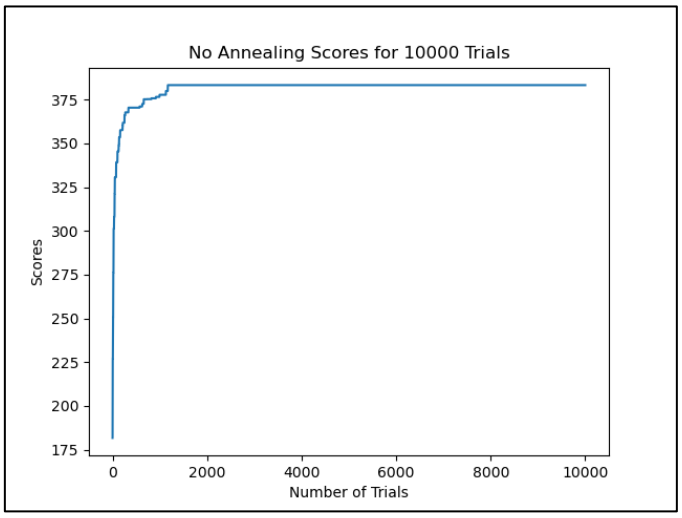
Correctness Percentage: 6.743



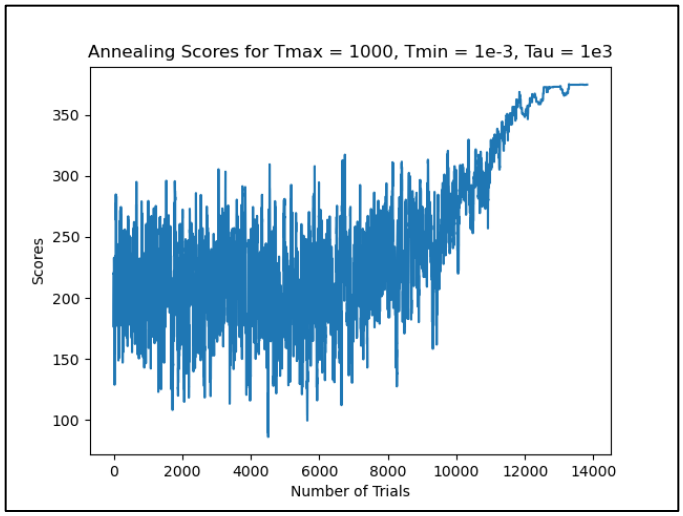
Correctness Percentage: 10.93



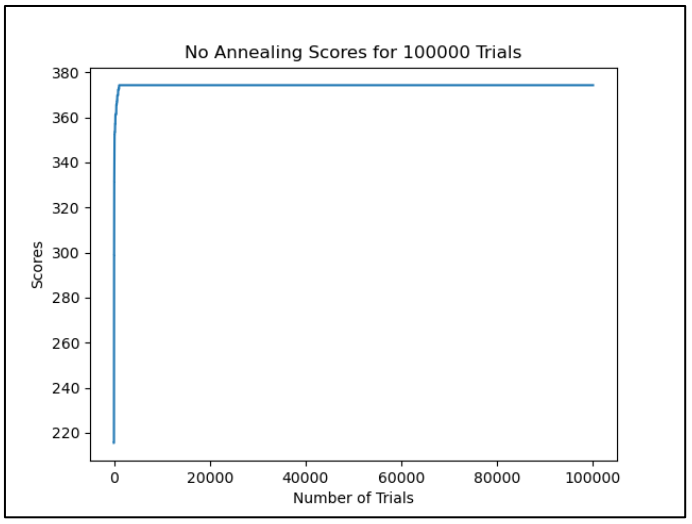
Correctness Percentage: 16.04



Correctness Percentage: 7.907



Correctness Percentage: 14.88



Correctness Percentage: 10.23