CS238: Homework 1

Pratyay Dutta

April 2025

1 Problem 1: Branch and Bound implementation in Python

I have implemented the branch and bound algorithm in python. The following are the source code and the corresponding outputs from the code.

```python
from typing import List

class PartialDigestSolver:
    def __init__(self, encoded_list: List[str]):
        self.L_dict = self.parse_input(encoded_list)            # Getting the counter from the
            dataset input
        self.width = max(self.L_dict)                           # Getting Max L

        # Initialisations
        self.L_dict[self.width] -= 1
        if self.L_dict[self.width] == 0:
            del self.L_dict[self.width]
        self.X = [0, self.width]
        self.solutions = []
        self.seen = set()                                       # Seen set to filter out
            symmetrical solutions

    def parse_input(self, encoded_list: List[str]) -> dict:
        """
        Parses an encoded multiset input list (e.g., ['3(2)', '5(1)']) and returns a dictionary
        representing the multiset with elements as keys and their counts as values.
        """
        L = {}
        for item in encoded_list:
            if '(' in item:
                num, count = item.strip(')').split('(')
                num = int(num)
                count = int(count)
            else:
                num = int(item)
                count = 1
            L[num] = L.get(num, 0) + count
        return L

    def delta(self, y, X):
        """
        Computes the multiset of absolute distances between a value y and all elements in set X.
        """
        return [abs(y - x) for x in X]

    def count_occurrences(self, lst):
        """
        Counts the occurrences of each element in a list and returns a frequency dictionary.
        """
        counts = {}
        for x in lst:
            counts[x] = counts.get(x, 0) + 1
        return counts

    def is_subset(self, subset, multiset):
        """
```

```
        Checks whether all elements in 'subset' (with multiplicities) are present in 'multiset'.
        """
        subset_counter = self.count_occurrences(subset)
        return all(subset_counter[x] <= multiset.get(x, 0) for x in subset_counter)

    def remove_lengths(self, L, lengths):
        """
        Decreases the count of each length in the given dictionary, removing entries when count
            reaches zero.
        """
        for length in lengths:
            L[length] -= 1
            if L[length] == 0:
                del L[length]

    def add_lengths(self, L, lengths):
        """
        Adds back lengths into the multiset dictionary by incrementing their counts.
        """
        for length in lengths:
            L[length] = L.get(length, 0) + 1

    def place(self, L, X):
        """
        Recursively attempts to reconstruct the original set of coordinates that could generate
        the input distance multiset L. It uses branch and bound to try placing the largest
        remaining distance from both sides of the interval.
        """
        if not L:
            normalized = tuple(sorted(X))
            if normalized not in self.seen and tuple(self.width - x for x in normalized) not in
                self.seen:
                self.solutions.append(sorted(X))
                self.seen.add(normalized)
            return

        y = max(L.keys())
        dy = self.delta(y, X)
        if self.is_subset(dy, L):
            self.remove_lengths(L, dy)
            X.append(y)
            self.place(L, X)
            X.pop()
            self.add_lengths(L, dy)

        y_complement = self.width - y
        dy_reflected = self.delta(y_complement, X)
        if self.is_subset(dy_reflected, L):
            self.remove_lengths(L, dy_reflected)
            X.append(y_complement)
            self.place(L, X)
            X.pop()
            self.add_lengths(L, dy_reflected)

    def solve(self):
        """
        Initiates the recursive reconstruction process and returns all distinct solutions found.
        """
        self.place(self.L_dict.copy(), self.X[:])
        return self.solutions
```

Listing 1: PartialDigestSolver Class Implementation

```
# Provided datasets
datasets = {
    "L1": ["2(1)", "3(1)", "4(1)", "5(1)", "6(1)", "7(2)", "8(2)", "10(1)"],
    "L2": ["1(9)", "2(8)", "3(7)", "4(6)", "5(5)", "6(4)", "7(3)", "8(2)", "9(1)"],
    "L3": ["1(1)", "2(1)", "3(2)", "4(1)", "5(2)", "6(1)", "7(1)", "9(2)", "10(1)", "12(1)", "
        14(1)", "15(1)"],
    "L4": ["1(16)", "2(15)", "3(14)", "4(13)", "5(12)", "6(11)", "7(10)", "8(9)", "9(8)",
           "10(9)", "11(8)", "12(7)", "13(6)", "14(5)", "15(4)", "16(3)", "17(2)", "18(1)"]
}

# Solving the datasets
for name, dataset in datasets.items():
    solver = PartialDigestSolver(dataset)
    solutions = solver.solve()
    print(f'\n----------Dataset {name}----------\n')
    print(f'Number of Distinct Solutions = {len(solutions)}')
    print(f'Solutions: {solutions}')
```

Listing 2: Dataset Execution Loop

```
----------Dataset L1----------

Number of Distinct Solutions = 0
Solutions: []

----------Dataset L2----------

Number of Distinct Solutions = 1
Solutions: [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]

----------Dataset L3----------

Number of Distinct Solutions = 2
Solutions: [[0, 5, 9, 12, 14, 15], [0, 1, 3, 6, 10, 15]]

----------Dataset L4----------

Number of Distinct Solutions = 1
Solutions: [[0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18]]
```

Figure 1: Resulting output on running the PDP solver on the provided datasets

2 Problem 2: Brute Force and Branch and Bound algorithm for DDP

The **Double Digest Problem (DDP)** is a classical problem in computational biology. It arises in physical mapping of DNA, where two restriction enzymes (say, A and B) cut a DNA sequence at specific locations. When each enzyme is applied separately, we obtain two sets of DNA fragment lengths, denoted by:

- d_A — fragment lengths from enzyme A.
- d_B — fragment lengths from enzyme B.

When both enzymes are applied together, the resulting set of fragments is denoted by:

- d_X — fragment lengths from the double digest (i.e., applying both A and B).

The goal is to recover the locations of the cuts made by enzymes A and B, consistent with all three sets of fragment lengths. The challenge arises because the enzymes can cut in overlapping or nested ways, leading to many possible configurations.

Problem Assumptions

- The DNA is linear and of known total length L, given by the sum of fragments in d_A, d_B, or d_X.
- Cuts by enzyme A and enzyme B do not occur at the same location.
- The fragments in each dataset are unordered and may contain repeated lengths.
- The task is to determine an ordering of the fragments in d_A and d_B such that the implied overlapping cuts explain d_X.

Algorithmic Analogy

This problem is closely related to the **Set Partitioning Problem** and **String Reconstruction Problem**. The brute-force algorithm for DDP resembles:

- Enumerating all permutations of fragments (like TSP or exhaustive sequence alignment).
- Testing each permutation pair for consistency via interval overlaps.

Brute-Force Approach

The idea of the brute-force algorithm is as follows:

1. Enumerate all possible permutations of fragment lengths in d_A and d_B.
2. For each permutation of d_A, construct hypothetical cut positions for enzyme A.
3. For each permutation of d_B, construct hypothetical cut positions for enzyme B.
4. Merge the two cut sets to simulate the double digest, and derive the resulting fragment lengths.
5. Compare the simulated double digest lengths with the given d_X. If they match (as multisets), store the cut locations.
6. Return all valid combinations of cut locations (or report no solution).

Pseudocode for Brute-Force Double Digest Solver

Algorithm 1 BruteForceDoubleDigest(d_A, d_B, d_X)

```
Let L ← ∑ d_A
Initialize an empty list Solutions
for all permutations P_A of d_A do
    Compute cuts C_A ← cumulative positions from P_A
    for all permutations P_B of d_B do
        Compute cuts C_B ← cumulative positions from P_B
        Merge cuts C ← (C_A ∪ C_B) and sort
        Compute fragments F from adjacent elements in C
        if multiset F equals multiset d_X then
            Append (C_A, C_B) to Solutions
return Solutions
```

Notes on Complexity

- The time complexity is $\mathcal{O}(n! \cdot m!)$ where $n = |d_A|$ and $m = |d_B|$.
- The bottleneck is the enumeration of permutations and the verification of each pair.
- Suitable only for small instances ($n, m \leq 8$) due to factorial blow-up.

2.1 Branch and Bound Approach

The naive brute-force approach tries all permutations of d_A and d_B, which leads to factorial time complexity: $\mathcal{O}(n! \cdot m!)$. **Branch and Bound** improves this by:

- Building partial solutions incrementally (branching),
- Abandoning (pruning) branches that are inconsistent with d_X early on.

This significantly reduces the number of permutations we need to examine. The Conceptual run through for the process is as follows:

1. Enumerate all permutations of d_A.
2. For each prefix of d_A, compute cumulative cut positions.
3. Try to incrementally assign fragments from d_B in an order that, when overlapped with d_A, produces a partial d_X consistent with the known d_X multiset.
4. If at any step, the simulated combined digest diverges from what is possible from d_X, we prune that branch.
5. If a full assignment leads to a valid d_X, we record the positions.

This approach is similar in spirit to backtracking solvers for puzzles like Sudoku or the traveling salesman problem with cost thresholds.

Pseudocode

Algorithm 2 DDP_BranchAndBound(d_A, d_B, d_X)

```
Let L ← ∑ d_A
Solutions ← ∅
Let X_target ← multiset of d_X
for all permutations P_A of d_A do
    Compute cuts C_A ← cumulative cut positions from P_A
    Call Search(P_A, [], C_A, d_B, X_target, Solutions)
return Solutions
```

Algorithm 3 Search(P_A, P_B, C_A, R_B, X_{target}, *Solutions*)

if R_B is empty **then**
  Compute C_B from P_B
  $C \leftarrow C_A \cup C_B$
  Compute multiset of fragment lengths $X \leftarrow$ from sorted C
  if $X = X_{\text{target}}$ **then**
    Append (C_A, C_B) to *Solutions*
  return
for all fragment b in R_B **do**
  $P'_B \leftarrow P_B + [b]$
  Compute C'_B from P'_B
  $C' \leftarrow C_A \cup C'_B$
  Compute partial X' from sorted C'
  if X' is a prefix (or subset) of X_{target} **then**
    $R'_B \leftarrow R_B$ without b
    Search(P_A, P'_B, C_A, R'_B, X_{target}, *Solutions*)

Here, R_B is the remaining fragments of d_B that have not yet been used in P_B. It is reduced at each recursive call. Since it is a recursive call, R_B reduces every recursive call.

Time Complexity

- Brute-force: $\mathcal{O}(n! \cdot m!)$
- Branch and Bound:
  - Worst-case: still $\mathcal{O}(n! \cdot m!)$
  - Best-case: much faster due to pruning
  - Practical runtime: depends on how early and how often pruning happens

Advantages

- Significantly more efficient than brute-force for medium-size problems.
- Still guarantees exact solutions.
- Easily extendable with additional constraints (e.g., known cuts, minimum distance).

3 l-mer Frequency Distribution

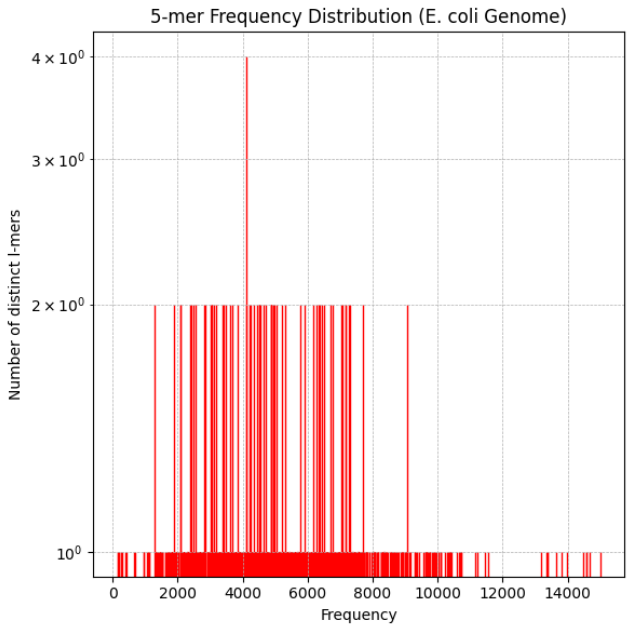
I went to the website and downloaded the genome in Fasta format for the Escherichia coli bacteria. The following steps were employed to extract and plot the frequency graphs.

- Read the genome from the fasta format file by excluding headers and linebreaks.
- Find the frequency plot of l=5 l-mers in the genome.
- Generate a random string of same length as the genome used.
- Find the frequency plot of l=5 l-mers in the random string.

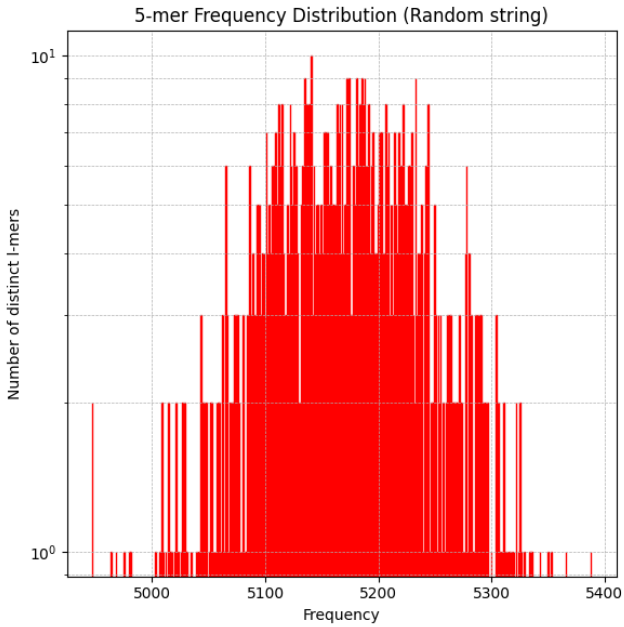
```
import matplotlib.pyplot as plt
import random

def read_fasta(filepath):
    with open(filepath, 'r') as f:
        lines = f.readlines()
    genome = ''.join(line.strip() for line in lines if not line.startswith('>'))
    return genome

def random_string(length):
    return ''.join(random.choices(['A', 'T', 'C', 'G'], k=length))

def frequency(genome, l):
    """
    Counts the frequency of each l-mer in a given genome string.

    Args:
        genome (str): DNA sequence.
        l (int): Length of l-mer.

    Returns:
        dict: Dictionary with l-mer as key and its frequency as value.
    """
    freq = {}
    for i in range(len(genome) - l + 1):
        lmer = genome[i:i + l]
        freq[lmer] = freq.get(lmer, 0) + 1
    return freq

def plot(lmer_freq, title="l-mer Frequency Distribution"):
    """
    Plots the distribution: how many distinct l-mers occur at each frequency.

    Args:
        lmer_freq (dict): Dictionary with l-mers and their counts.
        title (str): Title for the plot.
    """
    freq_dist = {}
    for count in lmer_freq.values():
        freq_dist[count] = freq_dist.get(count, 0) + 1

    x = sorted(freq_dist.keys())
    y = [freq_dist[k] for k in x]

    plt.figure(figsize=(6, 6))
    plt.bar(x, y, color='steelblue', edgecolor='red')
    plt.xlabel("Frequency")
    plt.ylabel("Number of distinct l-mers")
    plt.title(title)
    plt.yscale("log")
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)
    plt.tight_layout()
    plt.show()

# Main workflow
path = 'ecoli.fasta'
genome = read_fasta(path)
```

```
print(f'E coli Genome Length: {len(genome)}')

l = 5
lmer_counts = frequency(genome, l)
lmer_random = frequency(random_string(len(genome)), l)

plot(lmer_counts, title=f"{l}-mer Frequency Distribution (E. coli Genome)")
plot(lmer_random, title=f"{l}-mer Frequency Distribution (Random string)")
```

Listing 3: Script to compute and compare l-mer frequencies in E. coli and a random sequence



(a) E. coli Genome 5-mer Frequencies



(b) Random Genome 5-mer Frequencies

Figure 2: Comparison of 5-mer frequency distributions between the real E. coli genome and a random DNA sequence of equal length.

Biological Significance. The 5-mer frequency distribution in the *E. coli* genome exhibits a long-tailed pattern, where a few k-mers appear very frequently, while many others occur rarely. This reflects the presence of biologically meaningful structures such as promoters, codons, repetitive elements, and conserved motifs that are reused throughout the genome. In contrast, the distribution in the random DNA sequence is narrow and symmetric, consistent with purely stochastic generation, where each k-mer appears with approximately equal likelihood. This stark difference highlights how evolutionary and functional constraints shape genomic content, leading to non-random usage of short nucleotide sequences in real biological systems.

4 Problem 4: Randomized k-Means Clustering Using Weighted Distances and Softmax Sampling

1. Standard k-Means Clustering Algorithm

Goal: Partition n data points $\{x_1, \ldots, x_n\}$ into k clusters by minimizing the within-cluster sum of squared Euclidean distances.

Algorithm 4 Standard k-Means Clustering

1: Initialize k centroids $\{c_1, \ldots, c_k\}$ (e.g., randomly selected points).
2: **repeat**
3: **for** each data point x_i **do**
4: Assign x_i to the closest cluster:

$$\text{Cluster}(x_i) \leftarrow \arg\min_j \|x_i - c_j\|^2$$

5: **for** each cluster j **do**
6: Update centroid c_j as the mean of assigned points.
7: **until** cluster assignments do not change

2. Randomized k-Means with Softmax Sampling

To introduce stochasticity, we replace the hard cluster assignment step with a **probabilistic sampling** step using a *softmax over distances*.

2.1 Distance Metric: Weighted Combination

For each data point x_i and cluster centroid c_j, define a hybrid distance:

$$d_{ij} = \alpha \cdot \|x_i - c_j\|_2^2 + (1 - \alpha) \cdot \left(1 - \frac{x_i \cdot c_j}{\|x_i\| \|c_j\|}\right)$$

where:

- $\alpha \in [0, 1]$ is the weight controlling the trade-off between Euclidean and cosine distance,
- The cosine distance term is $1 - \cos(\theta_{ij})$.

2.2 Softmax Probability Distribution

Define the probability of assigning x_i to cluster j as:

$$P(z_i = j \mid x_i) = \frac{\exp(-\beta d_{ij})}{\sum_{l=1}^{k} \exp(-\beta d_{il})}$$

where $\beta > 0$ is a temperature parameter that controls the sharpness of the distribution.

2.3 Cluster Assignment via Sampling

Instead of assigning to the nearest centroid, we **sample** a cluster index z_i from the softmax distribution:

$$z_i \sim \text{Categorical}\left(P(z_i = 1), \ldots, P(z_i = k)\right)$$

2.4 Algorithm: Randomized k-Means

Algorithm 5 Randomized k-Means Clustering with Softmax Sampling

Input: Data points $\{x_1, \ldots, x_n\}$, number of clusters k, weight α, temperature β
Initialize k centroids $\{c_1, \ldots, c_k\}$ randomly
repeat
  for each data point x_i **do**
    for each cluster j **do**
      Compute distance:

$$d_{ij} \leftarrow \alpha \cdot \|x_i - c_j\|^2 + (1 - \alpha) \cdot \left(1 - \frac{x_i \cdot c_j}{\|x_i\| \|c_j\|}\right)$$

    Compute probabilities using softmax:

$$P_j \leftarrow \frac{\exp(-\beta d_{ij})}{\sum_{l=1}^{k} \exp(-\beta d_{il})}$$

    Sample cluster z_i using $P = (P_1, \ldots, P_k)$
  for each cluster j **do**
    Update centroid c_j as mean of points assigned to cluster j
until cluster assignments stabilize

3. Discussion

This algorithm introduces randomness into the assignment step, which can help:

- Explore alternative clusterings and escape local minima,
- Model uncertainty in cluster assignment,
- Smooth transitions in soft clustering applications.

The parameters α and β offer flexible control over the balance between distance metrics and assignment randomness.

5 Problem 5: Exploration-Decaying Gibbs Sampler for Motif Finding

1. Overview

We propose a modified Gibbs sampler for motif finding that balances exploration and exploitation through a decaying exploration constant $\varepsilon \in (0, 1]$. Unlike the standard Gibbs sampler, which updates one sequence per iteration, our method dynamically adjusts the number of updated sequences using a decaying function of ε.

2. Core Idea

Let:

- t be the current iteration number,
- ε_t be the exploration fraction at iteration t,
- $\gamma \in (0, 1)$ be the exponential decay rate,
- n be the total number of sequences.

Then:

$$\varepsilon_t = \varepsilon_0 \cdot \gamma^t$$

At each iteration, a random subset of $\lceil \varepsilon_t \cdot n \rceil$ sequences will have their motif starting positions updated using the Gibbs sampling strategy.

3. Sampling Strategy

1. Select one sequence s_i at random.
2. Construct the profile matrix P from all sequences except s_i.
3. Randomly choose a subset $\mathcal{S}_t \subseteq \{1, \ldots, n\}$ such that $|\mathcal{S}_t| = \lceil \varepsilon_t \cdot n \rceil$.
4. For each sequence $s_j \in \mathcal{S}_t$:
   - For each possible motif position a_j in s_j, compute the probability:

     $$\Pr(a_j \mid P) = \prod_{k=1}^{\ell} P_k(s_j[a_j + k - 1])$$

     where $P_k(b)$ is the probability of base b at position k in the motif.
   - Sample a new starting position a_j from the distribution $\Pr(a_j \mid P)$.
5. Update motif locations accordingly and recompute the motif score.

4. Stopping Condition

Repeat the above process until the motif score converges (i.e., no significant improvement over a fixed number of iterations).

5. Discussion

The decaying-ε Gibbs sampler algorithm exhibits a strong conceptual similarity to *simulated annealing*, a classical optimization method inspired by the annealing process in metallurgy. Both algorithms balance exploration and exploitation through a gradually decaying control parameter, which governs the behavior of the search process.

In simulated annealing, the algorithm begins with a high temperature T, which allows it to accept worse solutions with higher probability, enabling the search to move freely across the solution space. As the temperature decreases over time, the algorithm becomes more conservative, accepting only improvements or small local moves. This transition from global exploration to local refinement helps the algorithm escape local optima and converge toward a global minimum.

In the decaying-ε Gibbs sampler, the exploration constant $\varepsilon \in (0, 1]$ plays a similar role. Initially, $\varepsilon = 1$, meaning that in each iteration, the algorithm updates motif starting positions for all sequences based on Gibbs sampling. This corresponds to a highly exploratory phase, where the algorithm performs global updates and rapidly traverses the search space.

As the algorithm proceeds, ε decays exponentially:

$$\varepsilon_t = \varepsilon_0 \cdot \gamma^t \quad \text{with } \gamma \in (0, 1)$$

Eventually, ε_t becomes small enough that only a single sequence is updated per iteration, effectively reducing the algorithm to the classical single-sequence Gibbs sampler. This late-stage behavior emphasizes local refinement and stability, allowing the sampler to fine-tune its solution.

Summary of Similarities:

- Both algorithms begin with global, exploratory updates and transition toward local, refined updates.
- Both use a decaying parameter (T in simulated annealing, ε in the Gibbs sampler) to regulate the scope and randomness of updates.
- Both aim to avoid local minima early and improve convergence quality in later stages.

Thus, the decaying-ε Gibbs sampler can be interpreted as a motif-finding analog of simulated annealing, where the annealing is realized not by accepting worse solutions with some probability, but by controlling the number of sequences updated per iteration using a decaying exploration constant.

6. Advantages and Interpretation

Table 1: Comparison of Decaying-ε Gibbs Sampler vs. Vanilla Gibbs Sampler

Advantages of Decaying-ε Gibbs	**Disadvantages of Decaying-ε Gibbs**
Improved exploration: Updates multiple sequences early on, avoiding poor local optima.	**High variance:** Simultaneously updating many sequences can destabilize the profile, especially early on.
Faster convergence: Can jump more rapidly to promising motif configurations when signal is strong.	**Overshooting risk:** Aggressive updates may miss near-optimal solutions if exploration is not decayed smoothly.
Annealing behavior: Gradually transitions from stochastic to stable, like simulated annealing.	**Increased per-iteration cost:** Requires more probability computations and sampling steps per iteration.
Parallelizability: Updates to multiple sequences can be computed independently.	**Parameter sensitivity:** Requires careful tuning of ε_0, decay rate γ, and convergence thresholds.