1 Genome Sequencing

1. From a high level, explain the problem of *genome sequencing*, and what are the given inputs and desirable outputs. What limitations prevent us from having more informative inputs?

   Genome sequencing is the problem of extracting the genome of a cell from a collection of reads. We start by taking a set of white blood cells (containing many copies of DNA), immerse them in a denaturing agent to break it down into small strands. We then read some of the first characters of them: these are called reads. Each read is roughly 100–400 bases long. The bioinformatics problem of Genome Sequencing arises from taking these reads and reconstructing the original genome.

   The inputs to the algorithm are a multiset of k-mers – defined as the (disjoint) union of all subsequences of length k in any read. The desired outputs is the genome which is most likely to have generated such an output.

   We can't get a more informative input because sequencing machines are unable to read sufficiently long sequences. For example the length of Chromosome 1 in humans is 100 million bases.

2. Define a k-mer, a *prefix* and a *suffix* of a string within this context. How are these individual components used within the Hamiltonian and de Bruijn graphs?

   A k-mer is a sequence of k bases. A prefix is the first $k-1$ bases in a k-mer. A suffix is the last $k-1$ bases in a k-mer.

3. What is a necessary condition for a graph to have a Eulerian cycle?

   Every node has both an even in-degree and an even out-degree.

4. There exists an $\mathcal{O}(n^2 2^n)$-time algorithm for computing Hamiltonian paths (where n is the number of nodes). Conversely, what is the computational complexity of the best-known algorithm for computing Eulerian cycles? Provide pseudocode for both of those algorithms.

   The complexity of finding a Hamiltonian Cycle is $\mathcal{O}(n^2 \cdot 2^n)$ and is a brute force algorithm.

```
def hamiltonian(V, E):
    """
    :param V: start node
    :type V: set[str]
    :param E: dictionary of edges E[u] = {v | (u, v) is an edge}
    :type E: dict[str, set[str]]
    :return:
    """
    cycle = [V.pop()]

    def dfs():
        if not V:
            return True
        for v in V.copy():
            if v in E[cycle[-1]]:
                V.remove(v)
                cycle.append(v)
                if dfs():
                    return True
                else:
                    V.add(v)
                    return False
```

```
    dfs()

    return cycle
```

The complexity of finding an Eulerian Cycle is $\mathcal{O}(n)$ for a graph with n edges.

```
def eulerian(V, E):
    """
    :param V: start node
    :type V: set[str]
    :param E: dictionary of edges e[u][v] = |{(u, v) | (u, v) is an edge}|
    :type E: dict[str, dict[str, int]]
    :return:
    """
    path = []
    def dfs(u):
        while E[u]:
            v = next(iter(E[u].keys()))
            E[u][v] -= 1
            if not E[u][v]:
                E[u].pop(v)
            dfs(v)
        path.append(u)

    dfs(next(iter(V)))

    return path
```

5. Build the Hamiltonian and de Bruijn graphs over the following set of k-mers: `{"ATG", "TGG", "GGC", "GCG", "CGT", "GTG", "TGC", "GCA", "CAA", "AAT"}`

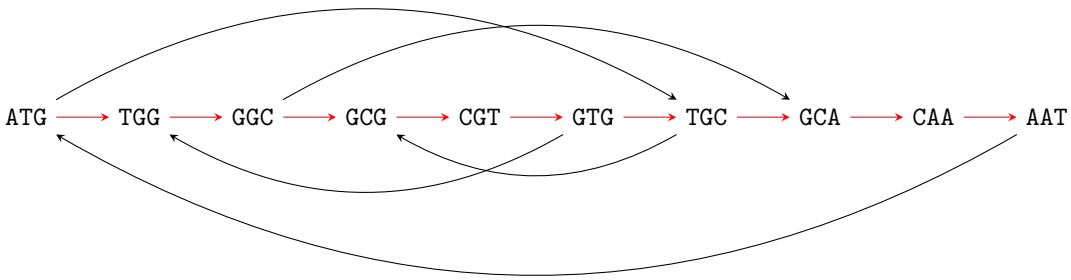

Figure 1: Hamiltonian Graph

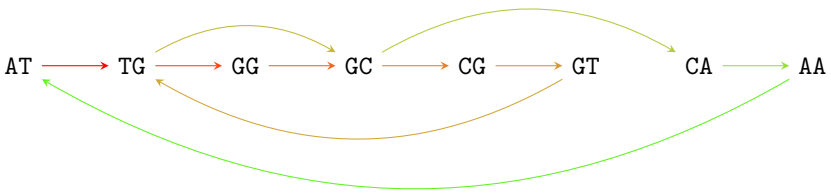

Figure 2: De Bruijn Graph

6. Explain how to sequence a genome using de Bruijn graphs constructed from *paired reads*. What is the limitation of the previous approach to sequencing that this approach tries to overcome?

   When we are breaking long sequences of DNA up into strands, rather than just reading from one end, we read from both! This creates a "paired read" where we have two reads which are a known distance away from each other (in this simplification).

   We use these paired reads to form paired k-mers where we have two sequences of length k which have d bases between them. We can then form a De-Bruijn Graph where each node contains two prefixes (p_1, p_2) of length $k-1$ and there is an edge between the nodes (a_1, a_2), (b_1, b_2) if, and only if there is a paired k-mer (x_1, x_2) such that the prefix of x_1 is a_1, its suffix is b_1 *and* the prefix of x_2 is a_2 and its suffix is b_2.

7. Outline the key incorrect assumptions that these approaches make, and how to fix two of them.

   Incorrect assumptions:

   - Reads are error-free

     We can ameliorate this by increasing the quality of machines; for example by using machine learning methods to extract the bases from the machines rather than deterministic algorithms.

   - We know the number of occurrences of each k-mer

     In reality we get a bag of reads which we then form into k-mers: if the end of one read matches the end of another read then they will have a lot of duplicate k-mers – and we can't tell if we have duplicates because we had multiple coverage of a part of the genome or because there is truly duplication! Since $\sim 50\%$ of most genomes are duplicate (often in long sections), we cannot even assume that if we have longer reads, we will not have duplicates.

   - We have total coverage of the genome

     We can solve this in by using smaller k and longer reads. But longer reads come up against mechanical constraints; and smalleer k increases the problems of duplicates.

   - We know the distances between the reads when we are making paired k-mers

2 Clustering

1. What is the output of a typical *gene expression* experiment, and why might one with to do further processing on such a result?

   In a typical *gene expression* experiment, we will get a set of vectors representing how much each gene is expressed at the moment. This information on its own is fairly uninterpretable: we can run clustering algorithms on it to find out which species / individuals are very similar to each other.

2. Define the inputs and outputs of the *k-means clustering* algorithm, and state its complexity class.

   Inputs: a set of nodes, each of which is represented by some vector of attributes. Outputs: a set of clusters, each containing a number of nodes. The algorithm is NP-hard!

3. Outline the steps taken by *Lloyd's algorithm*, which attempts to circumvent the issue from the above. State its time complexity, and provide an informal proof of its convergence. How might we use it to find "good approximations" for the k-means clustering solution?

Lloyd's Algorithm:

(a) **Initialisation:** set the centres of the clusters to be random nodes.

(b) **Iteration:** for each centre c, set its new position to be the mean position of all nodes for which c is the closet centre.

(c) **Termination:** terminate if no node has moved to a new cluster.

Consider this from an AI perspective. We are exploring a set of states, and move only to an adjacent state if it is better than the current state. Furthermore, there are only a *discrete* and *finite* number of states ($\mathcal{O}(n^m)$ when we have n data and m clusters). Consider placing a total order on the states based on their loss. At any given step we either reach a better state, or remain in the same state (we are in a local maxima). If we remain in the same state, then we will terminate. Since we are in a finite total order, we cannot infinitely move to better states: thus the Lloyd's algorithm is guaranteed to terminate in $\mathcal{O}(n^m)$ steps in the worst case (although realistic cases will be *far* faster).

4. Implement LLoyd's algorithm in a language of your choice, and demonstrate that it works by applying it for $k = 4$ on an easily separable set of 2D points. You may, for example, generate the points as $C + (\varepsilon_x, \varepsilon_y)$, where $\varepsilon_x, \varepsilon_y \sim U(-2, 2)$ (where U is a uniform real distribution) and $C \in \{(0,0), (0,5), (5,0), (5,5)\}$. The choice of C should then determine the resulting cluster.

```
import numpy as np


def lloyds(points, k):
    n, d = points.shape
    centres = points[np.random.choice(n, k, replace=False)]
    changed = True

    while changed:
        # broadcast
        distances = points.reshape((1, n, d))
        centres_br = centres.reshape((k, 1, d))

        # indices[i] is the index of the centre closest to the ith point
        indices = np.argmin(
            np.sum((distances - centres_br) ** 2, axis=2) ** 0.5, axis=0
        )

        # count[i] is num points s.t. the ith centre is the closest centre
        _, count = np.unique(indices, return_counts=True)

        # arr[i, j] = 1 iff centre i is the closest centre to the jth point
        arr = np.zeros((k, n))
        arr[indices, np.arange(n)] += 1

        # new_centres[i] is the mean of all points s.t.
        # the ith centre is the closest centre
        new_centres = (arr @ points) / count.reshape(k, 1)

        # check for convergence
        if np.allclose(centres, new_centres):
            changed = False

        centres = new_centres
```

```
    return centres


if __name__ == "__main__":
    np.random.seed(0)
    centres = np.array([[0, 0], [0, 5], [5, 0], [5, 5]])
    n_div_4 = 1000
    points = np.full((n_div_4, *centres.shape), centres).reshape(
        4 * n_div_4, centres.shape[1]
    ) + np.random.uniform(-2, 2, (4 * n_div_4, 2))
    lloyds(points, 4)
    # for each centre $\tilde x, \tilde y ~ N(\mu, 0.0316)$
    # [[ 5.01226398e+00 -2.55733633e-02]
    #  [-4.19110650e-02  4.96920291e+00]
    #  [-3.09700334e-02 -1.77992770e-04]
    #  [ 4.99306361e+00  4.98167004e+00]]
```

5. Explain the two high-level steps taken by the *expectation maximisation* (EM) algorithm, and then show how it relates to *soft k-means clustering* (giving particular reference to the *stiffness parameter*).

   We start off with a parameterised function which specifies the probability of a cluster generating a node.

   We now randomly initialise the clusters positions.

   - **Fit Parameters:**

     Find the hidden matrix where each element corresponds to the probability of each datapoint being in each cluster.

     This is done by calculating the probability of each cluster having generated a particular datapoint and then normalising across rows such that each datapoint has probability 1 of being generated by *some* cluster.

   - **Fit Nodes:**

     Update the position of each cluster according to the hidden matrix.

     This is done by maximising with respect to all the datapoints, but weighting each one by the probability that the datapoint is actually in that cluster.

   There are cases where it's not possible to have a "probability". In these cases, we create a proxy function (a "responsibility") which we use in leiu of the probability. Since we normalise, responsibilities need only to be positive (the same constraint on real probabilities: if we use a continuous distribution, the pdf at a particular datapoint could be greater than 1). A common responsibility is given by $e^{-\beta \cdot \text{distance}(\text{Data}_j, \text{Centre}_i)}$. The hyperparameter β is known as a "stiffness parameter". Higher β means clusters have very low responsibility for data in other clusters. This will force clusters further apart; but cause strange clusters if there are outliers. Lower β means clusters have a higher responsibility for data in other clusters: this means clusters are forced closer to each other; and outliers are less of a problem.

6. What is the time complexity of the *hierarchical clustering* algorithm when using the *minimum distance* metric between clusters?

   $\mathcal{O}(n^3)$ where there are n nodes in the graph.

7. Explain the inputs, outputs, steps and time complexity of the Markov Clustering (MCL) algorithm.

Markov Clustering is a graph clustering algorithm. The input to the algorithm is a graph $(V, E \in V \times V)$, an expansion parameter e and an inflation parameter r.

The output of the algorithm is a set of clusters. Each cluster consists of a set of nodes.

Markov Clustering Algorithm:

(a) **Initialisation:** construct the adjacency matrix from the graph. Then normalise (divide each row by the sum of values in the row) to form a valid Markov Matrix M' where $M'_{i,j}$ is the probability that a random walk would go from node i to node j.

(b) **Iteration:** Repeated Expansion and Inflation.

(c) **Expansion:** $M \coloneqq M^e$

(d) **Inflation:** $M_{i,j} \coloneqq M^r_{i,j}$; then normalise.

(e) **Termination:** Once the desired number of clusters have been found, return them

The algorithm runs for a non-constant number of steps (generally some linear function of the diameter of the graph). The time complexity of each step is $\mathcal{O}(n^3)$ where n is the number of nodes in the graph. This complexity can usually be brought down by using sparse matrix operations (since the Markov Matrices become very sparse in most cases).

The space complexity of the algorithm is $\mathcal{O}(n^2)$: this is required for storing the Markov Matrices.

Question 1 (Points to Clusters)**.** At some point, we will end up with a Markov Matrix $M_{i,j}$ where $M_{i,j}$ is the probability of i going to J and this will represent a graph which is split into clusters: how do we go from this to the clusters themselves?

Question 2 (Leiden Algorithm Refinement)**.** The lecture notes mention a "refinement stage" in the Leiden algorithm: but this is not explained. What is it?