Self Organizing Systems Exercise 1

Alexander Dobler 01631858
Thomas Kaufmann 01129115

November 16, 2020

1 Introduction & Problem Description

For exercise 1 of self-organizing systems we chose the task *Sequence alignment for Genetic Data (DNA Lattice) Anonymization* in which we should solve the DNA lattice anonymization described in [1] with two metaheuristic techniques from the lecture. The task is to find a pairing of DNA sequences, such that the sum of distances between two DNA sequences of a pair over all pairs is as small as possible.

More specifically we are given a set of n sequences described by strings which can have different length. In a first step we have to align these sequences, such that all sequences have the same length. This is done by introducing a *gap*-characters to increase the length of sequences. In general this process is called multiple sequence alignment (MSA) and we do not describe how this is done here but rather use a python-library as a black-box tool for this step. Now, that every sequence has the same length, we can compute the distance between two sequences as described in [1]. The last and main step is to combine this set of sequences into pairs, such that the sum of distances of two sequences of pair summed up over all pairs is minimal. Obviously this is just an application of minimal weighted matching in a complete graph, where a graph is represented by $G = (V, E)$ as follows. The set V of nodes are described by the sequences and the set E of edges is $V \times V$. For an edge $e = \{u, v\}$ its weight $w(e)$ is just the distance between the sequences u and v.

In the next sections we describe our solution approaches and main results.

2 Test Data & Preprocessing

As base set of data we chose DNA sequences from `https://www.kaggle.com/neelvasani/humandnadata` which consists of over 4000 human DNA sequences. In a next step we created multiple test-cases of different size by selecting 10-300 random samples of these sequences. Test-case sizes are always even, such that we do not have to bother about the leftover single sequence. For each of these test-cases we computed a multiple sequence alignment with the python module *Bio.SeqIO* from the package *Bio* (`https://biopython.org/wiki/SeqIO`). The last step computes the cost-matrix between pairs of sequences as described in [1]. All of the algorithms described in the next section only use the cost-matrix to compute a pairing.

These test-cases can be found in the project under the folder *data*: Sequence alignments for each test-case are stored in the files *human_data_XX.fasta* where *XX* denotes the size of the test-case. Similarly *human_data_XX.cm* stores the cost-matrices in *pickle*-format (`https://docs.python.org/3/library/pickle.html`).

3 DNALA & Exact Method

We provide two preliminary methods to solve this problem which are used to benchmark the metaheuristik techniques:

- An implementation of the DNALA algorithm as described in [1] can be found in the *algorithms* directory. We did implement the described randomness, but do not use multiple runs of the algorithm, but instead only run the algorithm for a testset once to determine its capability.
- As DNALA is only a heuristic and is not guaranteed to find an optimal solution we furthermore use a maximum weighted matching provided by the package *networkx* (`https://networkx.org/documentation/stable//index.html`) to compute an exact solution used for optimality gaps in benchmarking.

4 Genetic Algorithm

The first metaheuristic technique with which we solve the problem is a genetic algorithm. For this we use the *deap*-package for python (`https:`

//deap.readthedocs.io/en/master/) which provides a framework for creating genetic algorithms. The well-know genetic algorithm compononents are implemented as follows:

- **Solution representation**: A solution consists of $\frac{n}{2}$ pairs such that each node (sequence) appears in exactly one pair.
- **Fitness**: The fitness is just the sum of distances between pairs of points. This in fact also represents the value of a weighted matching the corresponding weighted matching.
- **Crossover**:
- **Mutation**:

Furthermore we have the option to select different population sizes and mutation rates for running the algorithm.

5 Ant Colony Optimization

At first we did not know how to solve the problem with ant colony optimization. But then we realized that a weighted matching in a complete graph is just a tour visiting every vertice where every second edge will have weight 0. This was really convenient, as ant colony optimization is well-known to perform well on the TSP-problem. Furthermore there are a lot of implementations available, which provide ACO-methods to solve the TSP-problem. For our purposes it was enough to alter the implementation provided by the *pants*-package (`https://pypi.org/project/ACO-Pants/`) in a way, such that we could solve the minimum weighted matching problem.

Our alteration can be described as follows. When solving the TSP-problem every ant starts at a specific node and tries to find a best tour. This means that at a specific point of the algorithm each ant has currently visited an acyclic path. Now this path has either even or odd length. This means when an ant can chose its next edge there are two different strategies:

- If the path until now has even length, then chose the usual strategy to select the next node to visit (pheromones in ACO with α- and β-values).

- If the path until now has odd length, then go to any not yet visited node. This resembles the fact that every other edge of the tour will have weight 0.

Of course we also have to alter the cost of a tour, in partical that every other edge has weight 0, to guide the algorithm to an optimal solution.

One might think that this construction of an algorithm to solve the minimum weighted matching problem is pretty superficial, but we will see that its performance is not too bad.

6 Results & Conclusion

References

[1] B. Malin. Protecting DNA Sequence Anonymitywith Generalization Lattices. 2004.