UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 3028

Using Graph Neural Networks to Separate Haplotypes in Assembly Graphs

Filip Wolf

Zagreb, May 2022

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 3028

Using Graph Neural Networks to Separate Haplotypes in Assembly Graphs

Filip Wolf

Zagreb, May 2022

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 3028

Pristupnik:	**Filip Wolf (0036510053)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	prof. dr. sc. Mile Šikić

Zadatak: **Korištenje graf neuronskih mreža za odvajanje haplotipa u grafovima sastavljanja**

Opis zadatka:

Cilj diploidnog de novo sastavljanja genoma jest ne samo rekonstruirati genomsku sekvencu pojedinca, već i odvojiti dva haplotipa, po jedan naslijeđen od svakog roditelja. Čak i nakon godina istraživanja i brojnih pokušaja, pouzdan alat za ovu vrstu problema nije konstruiran. Međutim, s najnovijom HiFi tehnologijom sekvenciranja, jedan smo korak bliže rješenju. U ovom projektu, prvi korak je korištenje asemblera Raven za konstruiranje grafova sastavljanja iz diploidnih podataka, u kojima čvorovi predstavljaju sekvence koje pripadaju pojedinim haplotipovima, a bridovi predstavljaju preklapanju među tim očitanjima. Idući korak jest konstrukcija modela dubokog učenja za predikciju bridova koji povezuju čvorove iz različitih haplotipova. Micanje tih čvorova iz grafa sastavljanja bi problem diploidnog sastavljanja pojednostavilo na dva zasebna problema haploidnog sastavljanja genoma. Učenje treba biti napravljeno na sintetičkom skupu podataka simuliranom iz genoma bakterija i kvasaca. Evaluacija treba biti napravljena na stvarnim očitanjima bakterija i kvasaca sekvenciranih PacBio HiFi tehnologijom. Rješenje treba biti implementirano u Pythonu koristeći Pytorch ili sličnu biblioteku za duboko učenje. Kod treba biti dokumentirati koristeći komentare i razvijati prema Google Python Style Guide ako je moguće. Cijeli programski proizvod potrebno je postaviti na GitHub pod jednu od OSI odabranih licenci.

Rok za predaju rada: 27. lipnja 2022.

Contents

1. Introduction

Traditionally, the focus of *de novo* genome assembly has always been on the reconstruction of an individual's genome from its numerous broken-up fragments obtained after sequencing (Illumina). We will here, however, focus on a different application of *de novo* genome assembly: haplotype separation. Every individual's genome is composed of both a mother's and a father's genome, each contributing about half of genetic material on average. We call these gene "halves" haplotypes. By separating the two haplotypes from the original genome, we can determine which parent contributes what genes. This has a wide range of applications, from ancestry tests to finding hereditary diseases.

We will try to do this using novel algorithms from the field of *deep learning* (DL). Bioinformatics has long been dominated by algorithms that employ complex heuristics and expert knowledge to find solutions to the problems researchers face. This is however slowly changing. More and more research is being done using deep learning to solve problems in bioinformatics, foregoing the laborious process of feature engineering and extensive human intervention. First, DL was employed only for finding dense and abstract representations of genome features, but it later started to completely replace the previously mentioned algorithms. It has contributed tremendously to the field in recent years and shows no signs of stopping, the most notable achievement being the solution to the protein folding problem which previously wasn't solvable for 50 years (Jumper (2021)). Still, there remains a long way to go before DL becomes completely standard within the field. Thus, this Thesis is concerned with applying recent DL techniques in order to solve the problem of separating the two haplotypes in an existing genome. This could not only potentially improve performance and reduce the necessity for human experts, but also bring bioinformatics to a wider range of people.

1.1. Bioinformatics

Bioinformatics is an interdisciplinary field of research that has had a tremendous impact on humanity in the last few decades. Since the completion of the Human Genome Project (Lander et al. (2001), Nurk et al. (2022)), the cost of sequencing a human genome has fallen exponentially. We can now reliably sequence a human genome for less than a \$1000 (NIH (2021)), all thanks to recent advances in sequencing technology, as well as the accompanying algorithms. We will here briefly explain the general pipeline of genome sequencing.

In an ideal world, we would extract a human genome in the form of DNA from a cell, input it into a sequencer, and get a complete and accurate sequence as output which we could immediately use for further study. However, unfortunately this is not (yet) the case and it is hard to predict when this might become possible. Due to this, we have to make due with sequencers that can only output genomes in the form of thousands of fragmented reads, at maximum about 10 kilobases (kb), with shorter read sequencers sequencing reads at lengths of around 150 base pairs (bp). This process is called *shotgun sequencing*. An average genome is much longer than that, e.g. the yeast genome is around 12 Mb long (NCBI (b)) and the human genome is around 6.4 Gb long (NCBI (a)), so after sequencing, we need to assemble these short reads before going deeper into analysis.

To do this, we first need to combine shorter reads into longer sequences called *contigs*. We then look for overlaps between contigs and use that information to create a graph where each node represents a read, and each edge represents an overlap between reads. We are then tasked with finding the longest possible path on this graph to connect the individual contigs and form a complete genome.

1.2. Thesis Task

The task we are presented with here is slightly different compared to standard genome sequencing. Instead of just finding a path through the graph and assembling the genome, we instead need to remove edges in the graph that connect contigs belonging to separate parents. By doing this, we are essentially separating the two haplotypes that constitute a genome.

1.2.1. Graphs

Graphs are data structures defined with a set of nodes (vertices) V and the edges connecting them E. Formally, this can be written as follows:

$$G = (V, E)$$

where:

$$E \subseteq \{(x, y) | (x, y) \in V^2 \text{ and } x \neq y\}$$

The above is an example of a directed graph, meaning that the edges only go in one direction, which is the type of graph we will work with. This is done because edge direction can encode both suffix - prefix and prefix - suffix overlaps.

1.3. Deep Learning

1.3.1. Basics

In the last decade, deep learning has grown from a niche research area to one of the largest fields withing computer science. It is now actively employed in virtually every human endeavor, from medicine to astronomy. And it is still growing day by day on its mission to become the standard way of handling almost all data.

In deep learning, we use data processing structures called *artificial neural networks* (ANNs) to extract useful information from our data and learn to predict an outcome, such as some feature of the data or a target class. It does this by adjusting learnable *weights* defined for every neuron in our network. Due to these weights, neural networks are much denser structures when compared to previous machine learning methods and can be referred to as *universal function approximators* (Hornik et al. (1989)) due to their ability to, with large enough networks, approximate any function. This gives them unprecedented performance on previously unsolvable tasks, but due to their abstract structure, makes them somewhat difficult to interpret. Some networks, like *convolutional neural networks* (Lecun et al. (1998)), don't suffer from this problem as much and can produce some quite intuitive visualizations. On the other hand, some networks, like the ones we will use here, cannot be visually meaningfully interpreted.

To successfully explain how ANNs work, we need to introduce two concepts: a loss function and backpropagation. When use our deep learning model to learn from data, we pass it through our network and compare the output to a previously defined true value by using a *loss function*. A loss function abstracts the error of our network prediction to a single number which is then used to calculate gradients in respect to our data. These gradients are then propagated back through the network using *backpropagation*. In essence, the backpropagation algorithm tells every weight in our network how to change in order to better predict our data. If we imagine our data as a 2-dimensional function on a plane, a gradient is the information of how steeps the function is at any specified point. This steepness value tells the network weights how much they should change, while its sign specifies the direction of change. By correctly propagating these gradient values back through the network, which backpropagation does, we an successfully make our network learn from data.

1.3.2. Graph Neural Networks

While standard ANNs are great at predicting simple data with no underlying structure (or at least one that isn't known), to successfully make our network learn from assembly graphs, we will need something more refined. Yes, it is true that we can simply represent our graph in the form of a 1-dimensional vector, but we then lose precious structural information about the contig overlaps. By using networks more tailor-made for data representation on graphs, we can take advantage of the graph's underlying structure. The networks in question are called *Graph Neural Networks* (GNNs) (Scarselli et al. (2004)). Most modern graph neural networks work on the principle of *message passing* (Gilmer et al. (2017)). A node accumulates information from adjacent nodes and the edges connecting them and uses it to update its own weights. By repeating this process enough times, we can converge to a stable solution. This can be represented wit the following equation.

Let $x_v \in \mathbf{R}^{d_1}$ be the feature for node v, and $w \in \mathbf{R}^{d_2}$ be the feature for edge (u, v). The message passing paradigm defines the following node-wise and edge-wise computation at step $t + 1$:

Edge-wise: $m_e^{(t+1)} = \phi(x_v^{(t)}, x_u^{(t)}, w_e^{(t)}), (u, v, e) \in \mathcal{E}$.

Node-wise: $x_v^{(t+1)} = \psi(x_v^{(t)}, \rho(\{m_e^{(t+1)} : (u, v, e) \in \mathcal{E}\}))$.

In the above equations, ϕ is a message function defined on each edge to generate a message by combining the edge feature with the features of its incident nodes; ψ is an update function defined on each node to update the node feature by aggregating its incoming messages using the reduce function ρ.

The GNN can be though of as an extension of CNNs. A CNN takes an image's local neighborhood and extracts information from it. It does this using convolution *filters*, which take a certain amount of pixels in a neighborhood and multiply them with weights. Now, the size of this filter is predefined and cannot be changed. For instance, it can have a size of 3 x 3 or 5 x 5. If we were to create such a filter for use on graphs, we would simply designate the central weight of the filter to be the node we are currently looking at, and the surrounding weights would be its neighboring nodes. As we can see, this would limit us to graphs where nodes had a constant number of neighbors, or graphs where we could look only at a limited number of neighbors. GNNs do not have this limitation. The ψ function takes all nodes in a central node's neighborhood into account equally.

We used two different GNNs in this thesis, which we will describe in the following sections.

Graph Covolutional Networks

The first network we used was the more simple of the two, and it bases its computations on Graph Convolutional Networks (GCN) (Kipf and Welling (2016)). It aggregates information from neighboring nodes and the edges connecting them and uses them to update the central node's information. It can be defined as follows:

$$h_i^{(l+1)} = \sigma(b^{(l)} + \sum_{j \in \mathcal{N}(i)} \frac{e_{ji}}{c_{ji}} h_j^{(l)} W^{(l)})$$

Here, $\mathcal{N}(i)$ represents all the neighboring nodes of node i, e_{ji} is the scalar weight of the edge connecting nodes i and j, $c_{ji} = \sqrt{|\mathcal{N}(j)|}\sqrt{|\mathcal{N}(i)|}$, and σ is an activation function. $b^{(l)}$, $h_j^{(l)}$ and $W^{(l)}$ are the networks bias at step l, features of node j at step l and weight of the network at step l respectively. The aggregated information is used to update the features of node i at step $(l+1)$.

Graph Attention Networks

The second networks we used, and the more complex of the two, was Graph Attention Networks (GAT) (Brody et al. (2021)). The version we used is an updated version of the original GAT paper (Veličković et al. (2017)) and we did notice a slight increase in performance in the newer version. It can be defined as follows:

$$h_i^{(l+1)} = \sum_{j \in \mathcal{N}(i)} \alpha_{ij}^{(l)} W_{right}^{(l)} h_j^{(l)}$$

where:

$$\alpha_{ij}^{(l)} = \mathrm{softmax}_i(e_{ij}^{(l)})$$

$$e_{ij}^{(l)} = \vec{a}^{T^{(l)}} \mathrm{LeakyReLU}(W_{left}^{(l)} h_i + W_{right}^{(l)} h_j)$$

Here, W_{right} is one of the two network weight matrices, h is a node feature matrix, α is an attention weight and $\vec{a}$ is the attention weight vector. This attention weight is the main reason why this network performs better compared to the regular GCN.

It differs from the original GAT only in the way $e_{ij}^{(l)}$ is calculated. Instead of using a single weight matrix W, we use two separate ones in the form of W_{right} and W_{left}, giving the network more parameters and learning power.

2. Dataset

When it comes to deep learning, data can often take a central role in determining the final outcome of the project. Although carefully crafting a predictive model is important, data quality can have a large impact on the model's performance. In this thesis, in order to train our model, we artificially created a dataset based on the brewer's yeast (*Saccharomyces cerevisiae*) genome. For most experimentation purposes, this dataset was only created using the first chromosome of the genome, as using the entire genome required the creation of a much larger dataset that would have significantly slowed down the training and testing process of different models. The larger dataset was necessary due to the potentially different nature of each chromosome and the way we predict them. We will explain more about this later. Nevertheless, using only one chromosome proved effective enough, as we will see later in the results section.

To generate the dataset, we used a multi-step process. The yeast genome was stored in a simple FASTA file with the chromosomes written down in order. In the first step, we would extract the first chromosome from the FASTA. In the next step, in order to simulate a fragmented chromosome like it was obtained in the process of sequencing, we needed to generate artificial reads from the chromosome. This was done using the `seqrequester`[1] package with the `generate` option. By setting the `-nreads` flag, we chose the number of reads to generate and using the `-distribution` flag, we chose the desired distribution for the reads (here set to `pacbio-hifi`). After this was done, we ended up with a file filled with simulated reads. We used a small C++ program[2] to mutate these reads into new ones of the same size. If the original simulated reads were the mother's reads, these mutated ones could be thought of as the father's reads. We combined them into a single file by simply concatenating them, and we were now ready for the final sequencing process where we would obtain contigs and their overlaps which we could use for training our model.

We found that generating 5000 - 10 000 reads gave satisfactory results in combination with the Raven sequencer (Vaser and Šikić (2021)). This is due to the fact that Raven's `filter` option reduces the number of edges in the graph if a smaller value for the option

[1]https://github.com/marbl/seqrequester

[2]Courtesy of R. Vaser

is specified. While setting the option to 0.99, and using the previously mentioned number of reads, Raven will generate an appropriate number of nodes and edges for training. Setting it to anything lower would create datasets too small for training. Unfortunately, this very low filter value also meant that the overlaps between our contigs were also a minimum of 99%, making them less of a candidate for usage as edge features.

After the Raven sequencer is done with the sequencing process, it outputs two files: a GFA file and CSV file. They both contain information about the contigs and their overlaps suitable in a format for generating a graph. They also contained different information in each, so to fully utilize all the graph information we have available, we needed to use both files. From the GFA file, we extract information about which contig belongs to the mutated reads and which one to the original simulated ones, as well as contig overlap length information. We extract them to a separate file for easier analyzing. Now, as we previously mentioned, our task is to separate the contigs into the father's and the mother's genome. In other words, if we are presented with a graph where nodes represent contigs and edges their overlaps, we need to remove edges between contigs that don't belong to the same parent. So for each edge, we specify if it's "correct" (i.e. it connects two contigs belonging to the same parent) or "incorrect" (it connect two contigs belonging to different parents). We now have a dataset where each edge has a label, as well as a feature in the form of overlap length. We can now use this as input to our model to generate a dataset fit for training.

3. Implementation

3.1. Technology Stack

The main functionality for this Thesis was achieved using two Python libraries PyTorch and DGL. PyTorch[1] probably needs no introduction. It is currently the most popular deep learning library used by millions due to its simplicity and versatility. It is free and open source and maintained by Facebook's AI Research lab. In this project, it was used for its *tensor* data structure, its implementation of nonlinearities such as the ReLU and ELU, loss functions such as cross entropy loss and others. It is also the main underlying architecture for all of our deep learning models which DGL is built on.

The Deep Graph Library[2] (DGL) (Wang et al. (2019)) is an free and open source deep learning library primarily aimed at graph neural networks. It is maintained by a diverse team of contributors mostly from Amazon Web Services. Here, we used it for the creation of graphs from the yeast dataset that were used for learning ans its implementation of all the GNN layers we used. It is built upon multiple available deep learning libraries, but we used the version implemented on PyTorch.

Aside from this, we also used a few other libraries to help us in some tasks. We will here shortly list them and describe them.

- NumPy[3] - useful for its large arrays
- Pandas[4] - for its CSV saving and loading capabilities
- Scikit-learn[5] - for its performance metrics such as F1 score
- TensorBoard[6] - used for visualizing our training process

[1]https://pytorch.org/
[2]https://www.dgl.ai/
[3]https://numpy.org/
[4]https://pandas.pydata.org/
[5]https://scikit-learn.org/stable/
[6]https://www.tensorflow.org/tensorboard

3.2. Structure

The project contained three main files as well as some helper scripts. In the following sections, we will describe these.

3.2.1. DGLDataset

The first thing to do was generate a dataset from the data obtained after sequencing. For this, we used DGLs `dgl.data.DGLDataset` class, which we inherit. It has multiple purposes, after generating a dataset we can save it and load it by calling the `save` and `load` functions. We can also check if there already is a previously saved dataset using the `has_cache` function. We can also get an instance of the dataset by calling the `__getitem__` function, as well as its length using the `__len__` function. But by far its most important purpose is generating the dataset we will use for training. We do this by loading the previously generated node and edge features and using them to generate a graph. We also encode label information, edge features, and node features. Node features are generated by taking the in and out degrees of every node.

3.2.2. Models

The models we used for training are defined by inheriting the `torch.nn.Module` class. In each model, we define the layers of the network, as well as the `forward` function.

3.2.3. Main

All of the previously described functions are connected in the `main` function. First, we define the function for logging parameters such as loss and accuracy for visualization using TensorBoard. Then, we define the dataset by calling the `DGLDataset` module and, if it previously hasn't been generated, create the dataset. We then define the model we will use for training with all its parameters and layer sizes. We define the optimizer, and finally, start the training process.

4. Conclusion

Bibliography

S. Brody, U. Alon, and E. Yahav. How attentive are graph attention networks? 2021. doi: 10.48550/ARXIV.2105.14491. URL `https://arxiv.org/abs/2105.14491`.

J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1263–1272. PMLR, 06–11 Aug 2017. URL `https://proceedings.mlr.press/v70/gilmer17a.html`.

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. URL `https://www.sciencedirect.com/science/article/pii/0893608089900208`.

Illumina. Assembling novel genomes. URL `https://www.illumina.com/techniques/sequencing/dna-sequencing/whole-genome-sequencing/de-novo-sequencing.html`.

E. R. P. A. e. a. Jumper, J. Highly accurate protein structure prediction with alphafold. *Nature*, 2021. doi: 10.1038/s41586-021-03819-2. URL `https://doi.org/10.1038/s41586-021-03819-2`.

T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. 2016. doi: 10.48550/ARXIV.1609.02907. URL `https://arxiv.org/abs/1609.02907`.

E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, and W. e. a. FitzHugh. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001. doi: 10.1038/35057062.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

NCBI. Homo sapiens, a. URL https://www.ncbi.nlm.nih.gov/genome/?term=Homo+sapiens.

NCBI. Saccharomyces cerevisiae, b. URL https://www.ncbi.nlm.nih.gov/genome/?term=Saccharomyces%20cerevisiae[Organism]&cmd=DetailsSearch#:~:text=The%20Saccharomyces%20cerevisiae%20genome%20is,Mb%2C%20organized%20in%2016%20chromosomes.

NIH. The cost of sequencing a human genome, 2021. URL https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost.

S. Nurk, S. Koren, A. Rhie, M. Rautiainen, A. V. Bzikadze, A. Mikheenko, M. R. Vollger, N. Altemose, L. Uralsky, A. Gershman, S. Aganezov, S. J. Hoyt, M. Diekhans, G. A. Logsdon, M. Alonge, S. E. Antonarakis, M. Borchers, G. G. Bouffard, S. Y. Brooks, G. V. Caldas, N.-C. Chen, H. Cheng, C.-S. Chin, W. Chow, L. G. de Lima, P. C. Dishuck, R. Durbin, T. Dvorkina, I. T. Fiddes, G. Formenti, R. S. Fulton, A. Fungtammasan, E. Garrison, P. G. S. Grady, T. A. Graves-Lindsay, I. M. Hall, N. F. Hansen, G. A. Hartley, M. Haukness, K. Howe, M. W. Hunkapiller, C. Jain, M. Jain, E. D. Jarvis, P. Kerpedjiev, M. Kirsche, M. Kolmogorov, J. Korlach, M. Kremitzki, H. Li, V. V. Maduro, T. Marschall, A. M. McCartney, J. McDaniel, D. E. Miller, J. C. Mullikin, E. W. Myers, N. D. Olson, B. Paten, P. Peluso, P. A. Pevzner, D. Porubsky, T. Potapova, E. I. Rogaev, J. A. Rosenfeld, S. L. Salzberg, V. A. Schneider, F. J. Sedlazeck, K. Shafin, C. J. Shew, A. Shumate, Y. Sims, A. F. A. Smit, D. C. Soto, I. Sović, J. M. Storer, A. Streets, B. A. Sullivan, F. Thibaud-Nissen, J. Torrance, J. Wagner, B. P. Walenz, A. Wenger, J. M. D. Wood, C. Xiao, S. M. Yan, A. C. Young, S. Zarate, U. Surti, R. C. McCoy, M. Y. Dennis, I. A. Alexandrov, J. L. Gerton, R. J. O'Neill, W. Timp, J. M. Zook, M. C. Schatz, E. E. Eichler, K. H. Miga, and A. M. Phillippy. The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022. doi: 10.1126/science.abj6987. URL https://www.science.org/doi/abs/10.1126/science.abj6987.

F. Scarselli, A. C. Tsoi, M. Gori, and M. Hagenbuchner. Graphical-based learning environments for pattern recognition. In A. Fred, T. M. Caelli, R. P. W. Duin, A. C. Campilho, and D. de Ridder, editors, *Structural, Syntactic, and Statistical Pattern Recognition*, pages 42–56, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-27868-9.

R. Vaser and M. Šikić. Raven: a de novo genome assembler for long reads. *bioRxiv*, 2021. doi: 10.1101/2020.08.07.242461. URL https://www.biorxiv.org/content/early/2021/02/22/2020.08.07.242461.

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. 2017. doi: 10.48550/ARXIV.1710.10903. URL `https://arxiv.org/abs/1710.10903`.

M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. 2019. doi: 10.48550/ARXIV.1909.01315. URL `https://arxiv.org/abs/1909.01315`.

Using Graph Neural Networks to Separate Haplotypes in Assembly Graphs

Abstract

Keywords:

Sažetak

Ključne riječi: