

DipGNNome: Diploid *de novo* genome assembly with geometric deep learning and beam-search

Martin Schmitz^{1,2}, Lovro Vršek², Kenji Kawaguchi¹, and Mile Šikić^{2,3}

¹ School of Computing, National University of Singapore, Singapore

² Genome Institute of Singapore, A*STAR, Singapore

³ Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia

Abstract. *De novo* genome assembly remains a central challenge in computational biology, particularly for diploid genomes where maternal and paternal haplotypes must be accurately resolved. Existing assemblers achieve impressive results through carefully designed heuristics, yet modern deep learning methods remain largely unexplored in the diploid setting.

We present **DipGNNome**, the first deep learning-based framework for diploid *de novo* genome assembly. Our approach formulates genome assembly as an edge classification and graph traversal problem, given haplotype-aware assembly graphs. We train a graph neural network (GNN) to guide contig construction as the layout phase in an Overlap-Layout-Consensus genome assembly pipeline. To enable this, we establish a novel pipeline for generating diploid graphs with ground-truth edge labels, providing the first systematic way to produce training data for machine learning models in this domain. This framework creates a foundation for applying and extending graph-based deep learning to diploid assembly. **DipGNNome** creates assemblies comparable to SotA and demonstrates the feasibility of deep learning for diploid assembly and introduces a paradigm that bridges algorithmic genomics with graph representation learning.

Our code, dataset and trained model are openly available at <https://github.com/lbcb-sci/DipGNNome>.

Keywords: *De novo* genome assembly · Graph Neural Networks · Diploid assembly

1 Introduction

The genome is encoded in sequences of four nucleotides—adenine (A), guanine (G), cytosine (C), and thymine (T), that together form deoxyribonucleic acid (DNA). DNA is organized into chromosomes, each spanning millions to hundreds of millions of nucleotides. A genome typically consists of multiple chromosomes, often present in pairs or higher ploidy levels, where homologous chromosomes are similar but not identical. Sequencing technologies make it possible to extract genomic information from these chromosomes, producing large collections of short, unordered fragments known as reads. Reconstructing the full genome from such reads is the central problem of genome assembly. Reference-based assembly methods align reads to a reference genome. While effective when a high-quality reference is available, this approach can introduce bias and obscure genuine structural variation between the target and reference genomes. In contrast, *de novo* assembly reconstructs genomes directly from reads, without relying on a reference, thereby avoiding these biases.

De novo assembly is a cornerstone of computational genomics, with applications spanning foundational research in biology, evolutionary biology, and personalized medicine. Nurk et al. (2022) [12] produced the first complete telomere-to-telomere (T2T) human genome, followed by numerous high-quality assemblies for humans and other species. These efforts combine complementary sequencing technologies, sophisticated assembly algorithms, and extensive manual curation by large consortia. The progress of recent assemblers has been driven primarily by algorithmic innovations. SotA assemblers like `hifiasm` [3] and `Verkko` [14] exploit the high accuracy of PacBio HiFi reads and the ultra-long range of Oxford Nanopore reads to generate phased, near-complete T2T assemblies. Yet, despite these achievements, current approaches remain purely algorithmic and do not incorporate modern machine learning techniques that have transformed other domains.

In this work, we present `DipGNNome`, the first deep learning-based assembler capable of reconstructing diploid genomes directly from raw assembly graphs, augmented with trio information. `DipGNNome` takes overlap graphs as input and performs the Layout step of an Overlap-Layout-Consensus pipeline. Unlike existing methods, `DipGNNome` integrates GNNs with haplotype-aware graph processing to jointly reconstruct maternal and paternal haplotypes, bridging the gap between classical algorithmic assemblers and emerging machine learning approaches.

2 Related Work

GNNome [18] marks the first deep learning-based genome assembler. It trains a GNN to classify assembly graph edges as correct or incorrect and then applies a greedy pathfinding algorithm to construct contigs from the scored edges. Similarly, Simunovic et al. [15] focus on the layout of de Bruijn graphs. However,

these approaches are limited to producing haploid assemblies.

Diploid assembly poses additional challenges. Most organisms, including humans, carry two homologous copies of each chromosome. While homozygous regions are nearly identical, heterozygous regions can harbor substantial divergence. Distinguishing sequencing errors from true variants and correctly phasing both haplotypes remains a central difficulty in diploid assembly. Recent years have seen the development of deep learning-based phasing methods such as **GAEseq** [6], **CAECseq** [5], **NeurHap** [19], and **ralphi** [1]. These approaches highlight the potential of machine learning for haplotype resolution, but they operate in a reference-based setting and do not address *de novo* assembly.

Strategies for *de novo* diploid assembly usually incorporate external information to resolve haplotypes. A prominent example is trio binning [7], which leverages parental short reads to identify haplotype-specific k -mers and guide separation during assembly, as implemented in tools such as **Verkko** [14] and **hifiasm** [3]. Our method builds on this paradigm by combining haplotype-aware assembly graphs with parental k -mer annotation and graph-based machine learning.

3 Overview

Our contributions are threefold: (1) we introduce the first publicly available pipeline for constructing machine learning-ready assembly graphs with ground-truth, enabling supervised training in the diploid setting, and share our dataset; (2) we develop an assembly algorithm that combines model predictions with a beam-search strategy to efficiently traverse long, string-like graphs with limited branching, a design that may generalize to other path-finding tasks; and (3) we show that a GNN-based assembler can achieve assembly results close to SotA methods while following a fundamentally different, learning-driven paradigm.

Figure 1 outlines the three stages of our approach: (A) **Data processing**, where HiFi reads are assembled into a unitig graph with **hifiasm**, simplified, and annotated with haplotype information using trio-derived k -mers; (B) **Synthetic training**, where simulated diploid reads generate labeled graphs for supervised GNN training via edge classification; and (C) **Genome assembly**, where real data are processed as in step (A), edges are scored by the trained GNN from step (B), and a beam-search reconstructs phased maternal and paternal haplotypes.

The remainder of the paper is organized as follows: The next three sections describe the three parts of the pipeline, (A), (B), and (C). Section 4 details the data processing pipeline, Section 5 describes GNN training, and Section 6 presents the genome assembly algorithm. Section 7 evaluates **DipGNNome** on real and simulated datasets, comparing it to the SotA. Section 8 concludes with a discussion of implications, limitations, and future directions.

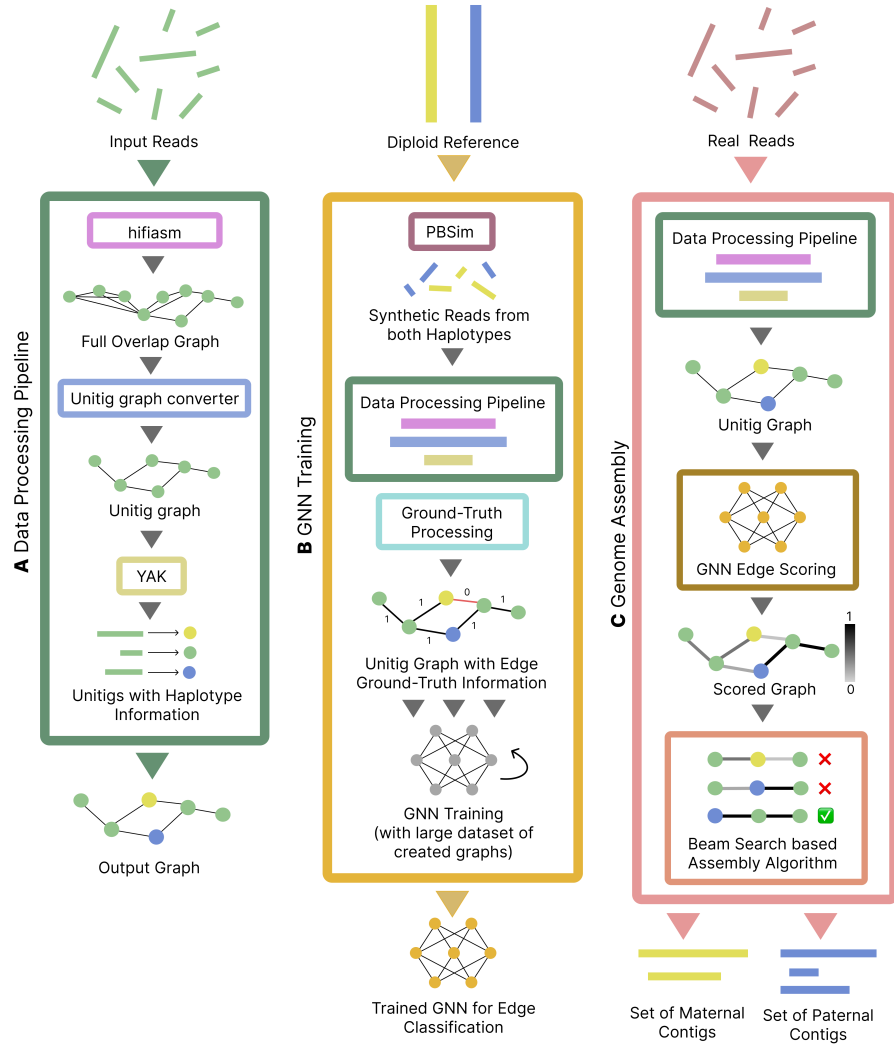


Fig. 1. Overview of the DipGNNome pipeline. **A: Data processing.** HiFi reads are assembled into a unitig graph with **hifiasm**, simplified, and annotated with haplotype information using trio-derived k -mers. **B: Synthetic training data.** Simulated diploid reads generate labeled graphs, enabling supervised training of a GNN for edge classification. **C: Genome assembly.** Real data is processed as described in **A**, edges are scored by the trained GNN from **B**, and a beam-search based algorithm reconstructs phased haplotypes.

4 Data Processing Pipeline

The following section describes how we construct graphs as summarized in Figure 1 A. Given a set of reads (real or synthetic) as input, the pipeline consists of the following steps:

4.1 Create Raw Overlap Graph (Step 1)

We start with an overlap graph, where each node represents a read in the dataset and each edge represents an overlap. Any tool that can produce such a graph is possible to be used here. For our experiments, we use `hifiasm` v0.25 [4] with default parameters.

4.2 Convert Raw Graph to Unitig Graph (Step 2)

Given the raw overlap graph, we search for transitive edges using an algorithm based on Myers’ transitive edge reduction algorithm [11]. An edge $e_t = a \rightarrow c$ is considered transitive if edges $e_1 = a \rightarrow b$ and $e_2 = b \rightarrow c$ both exist. To relax this rule, we introduce an $\epsilon = 0.12$ parameter (inspired by Raven [17]), which allows transitive edges to be removed only if the indirect path length lies between $d(1 - \epsilon)$ and $d(1 + \epsilon)$ (where d is the direct path). After the removal of transitive edges, unbranching chains of nodes are merged into so-called *unitigs*. This significantly reduces the size and complexity of the graph. The resulting graph is called a unitig graph.

4.3 Add Trio Binning to Unitigs Using yak (Step 3)

We use Illumina short reads from both parents of the genome of interest and apply `yak` [4] trio binning. `yak` identifies unique k-mers in both maternal and paternal short reads. Then it traverses the unitigs in the graph and marks unique k-mers as identified before. We leverage this counts of unique parental k-mer hits, $k_m(v)$ and $k_p(v)$. These values provide an informative representation of each unitig, capturing both the haplotype assignment (maternal vs. paternal) and the strength of the heterozygosity (the absolute count $k_m(v)$ and $k_p(v)$ may reflect how heterozygous a unitig is).

4.4 Add Node and Edge Features (Step 4)

We enrich the graph with both node and edge features. Edge features include the *overlap length* (number of base pairs shared between two reads). Node features include *read support* (the number of raw reads contributing to a given unitig, with read-level coverage computed by `hifiasm` and aggregated in unitigs as a length-weighted average) as well as *in-degree/out-degree* of the nodes.

5 GNN Training

This Section explains the GNN training pipeline as shown in Figure 1 B. Given a set of reference genomes, we create a synthetic training dataset with ground-truth and use this to train a GNN.

5.1 Sample reads

We simulate $40\times$ coverage reads ($20\times$ per haplotype) using `PBSim3` [13], sampling independently from maternal and paternal references of each chromosome. The resulting FASTQ files are merged into one, with reads annotated by position, strand, chromosome, and haplotype.

5.2 Compute Ground-Truth

Training our GNN requires labeled assembly graph edges. We first mark all edges connecting different chromosomes (translocations) or opposite strands (inversions) as false. Translocation edges are additionally saved as a separate list to train a second translocation classifier. Positional information determines whether an edge represents a valid suffix-prefix overlap. This requires assigning coordinates to unitigs.

For diploid genomes, differences in haplotype coordinates between homologous loci complicate verification. To resolve this, we translate coordinates between haplotypes using `Liftover` [16] and refine missing values by leveraging information from multiple reads within unitigs. Overlaps are then validated in both coordinate systems, and an edge is labeled correct if it is consistent in at least one haplotype. This procedure yields reliable supervision signals for robust model training. A full description of our ground-truth assigning algorithm can be found in Appendix A.

5.3 GNN Model

We train a `SymGatedGCN` model with `SymBCELoss`, both introduced by GN-Name [18], using the ground-truth edge labels described above. `SymGatedGCN` extends `GatedGCN` [2] and produces d -dimensional node and edge embeddings for the assembly graph. A jointly trained MLP classifier uses these embeddings to predict (i) whether an edge is incorrect (i.e., not part of the optimal assembly) and (ii) whether it represents a translocation. The model is optimized using binary cross-entropy loss and stochastic gradient descent.

To mitigate oversmoothing, we apply `PairNorm` [23] after each GNN layer. For node representations $V = [v_1, \dots, v_n] \in \mathbb{R}^{n \times d}$, `PairNorm` centers, normalizes, and rescales features:

$$V' = s \cdot \frac{V - \mu(V)}{\text{mean}_i \|v_i - \mu(V)\|_2}, \quad (1)$$

where $\mu(V) = \frac{1}{n} \sum_{i=1}^n v_i$ and $s = 1$. This preserves relative feature distances and improves training stability.

Node features $x_i \in \mathbb{R}^{d_v}$ and edge features $z_{ij} \in \mathbb{R}^{d_e}$ are projected into d -dimensional embeddings through the GNN. The MLP classifier then combines edge and incident node embeddings to produce the two prediction scores, which are used for graph pruning and downstream diploid assembly.

6 Genome Assembly Algorithm

6.1 Problem Definition

We formulate the *de novo* genome assembly problem as a path-finding task in a directed graph $G = (V, E)$, where each node can be traversed exactly once.

Nodes (V): Each node $v \in V$ represents a unitig and is associated with the following attributes:

- $l_n(v)$: Length of the genomic sequence represented by the unitig.
- $\text{kmer}_m(v), \text{kmer}_p(v)$: Counts of unique k -mers assigned to the maternal and paternal haplotypes, respectively.

Edges (E): Each edge $e = (u, v) \in E$ represents an overlap between two unitigs and has the following attributes:

- $S_G(e)$: The model output, interpreted as the probability that the connection is incorrect. Due to the sigmoid activation in the output layer, $S_G(e) \in [0, 1]$.
- $l_e(e)$: Length of the sequence contributed by v when extending from u to v .

Note, the attributes listed here differ from those used during GNN training. Certain features used for training are no longer directly relevant, as their information is captured in the model predictions. Conversely, the assembly algorithm requires additional attributes that were not important for predicting edge quality but are essential for constructing contigs. Additionally, the graph has the following additional properties:

Complement Relationships For each node v , its complement $v \oplus 1$ represents the reverse complement sequence, where \oplus represents a bit-wise XOR (for example $0 \oplus 1 = 1$ and $1 \oplus 1 = 0$). This relationship is important throughout the algorithm. For each node v , node $v \oplus 1$ is guaranteed to exist in the graph, and for each edge (u, v) , an edge $(v \oplus 1, u \oplus 1)$ exists, which imposes a symmetry on the entire graph.

Component Decomposition The graph is decomposed into weakly connected components, with small components filtered out. Each remaining component is processed independently. This means we can assume for the algorithm that the input graph is weakly connected. Before running the assembly algorithm, we prune the graph using the model’s *translocation edge score*. Translocation edges are defined

as edges that connect nodes from different chromosomes (i.e., the source node lies on one chromosome and the target node on another). Specifically, we remove all edges with a translocation prediction $p_t > c_t$, where c_t denotes the translocation cut threshold (set to $c_t = 0.5$ in our experiments). We also prune edges with edge score $S_G(e) > 0.9$, to remove edges with very poor model predictions entirely.

6.2 Assembly Algorithm

The full genome assembly consists of two independent passes: one with haplotype $h = m$ (maternal), and one with $h = p$ (paternal). In each pass, the algorithm traverses weakly connected components of the graph and penalizes paths based on unique k-mer counts (Section 4.3) from the opposite haplotype. Specifically, nodes with $kmer_p(v) > 0$ are penalized in the maternal pass, and nodes with $kmer_m(v) > 0$ are penalized in the paternal pass.

Each weakly connected component is traversed independently and over several iterations. A path is computed, the component is reduced by the set of visited nodes, and the process is repeated until no valid path longer than a threshold can be found or the component becomes too small. One path search is initiated by sampling n random edges (u, v) in the graph. For each sampled edge, we perform two beam search runs: one forward search from v and one reverse search from $u \oplus 1$ (complement of the source). The reverse path is then flipped and modified to use complement nodes. Finally, we concatenate both paths and select the longest among the n samples. The complete process is shown as pseudocode in Appendix B.

The core of this assembly algorithm is the `AssemblyBeamSearch` function, a path-finding approach based on beam search—an algorithm that maintains a set of k candidate paths (the "beams") and iteratively extends them by considering the k best resulting paths. Our assembly algorithm includes several diploid-genome-assembly-specific adaptations to traditional beam search. The goal is to maximize an assembly heuristic, based on unitig lengths, haplotype information, and model scores. The approach is shown in detail as Pseudocode in Appendix C.

We incorporate several changes to the default beam-search. First, we **save the best intermediate beam** during the search. This is important because it may happen that only negative-scoring edges are available at some point, leading to decreasing path scores. While we do not want to prevent the agent from taking negative edges (to ensure contiguity and because the predictions of the model are not always reliable), we also want to avoid letting the search degrade indefinitely. To address this, we track the best beam seen so far and retain it, even if the current beams degrade. This way, we can output high-scoring sub-paths without requiring the beam to reach the end of a component.

We also explicitly **prevent complement pairs** from being included in a single beam. Each beam maintains a visited set that includes all its nodes and their complements; any node present in this set is excluded from further beam extensions.

Additionally, we perform **beam merging** to reduce redundancy. The directed assembly graph tends to have string-like regions where many beams overlap. When two beams reach the same node, we retain only the higher-scoring one and discard the other. This conserves memory and runtime while ensuring that the best-scoring paths are preserved. Appendix D shows the complete Pseudocode of the beam merging strategy.

To mark visited nodes and their neighbors, we call **MarkVisited**, which includes each node, its complement, and the 2-hop neighborhood of both. This prevents re-traversal of nodes and the traversal of nodes that represent the same sequence in the genome as already visited ones.

6.3 Beam Score Heuristic

The beam search algorithm evaluates each candidate edge $e = (u, v)$ using a score $S(e)$ that balances the amount of novel sequence added to the assembly, haplotype consistency, and model-predicted edge confidence. The overall score is defined as:

$$S(e) = L(v) - K(v) - M(e) \quad (2)$$

The three components are defined as follows:

Length Reward ($L(v)$): This term gives a reward for additional base pairs added to the sequence from passing this edge, such that longer contigs get higher scores than shorter ones. Let $l(v)$ be the total length of the sequence associated with node v , and $l(e)$ the overlap length between nodes u and v . Then the net new sequence added by edge e is $l(v) - l(e)$, and the reward is computed as:

$$L(e) = \alpha \cdot (l(v) - l(e)) \quad (3)$$

where α is a scaling hyperparameter.

K-mer Penalty ($K(v)$): To promote haplotype consistency during diploid assembly, we penalize edges that introduce unique k-mers associated with the *non-target* haplotype. Let $k_{\neg h}(v)$ denote the number of such unique k-mers in node v , where $\neg h$ represents the haplotype opposite to the current assembly target h . We scale this penalty by the proportion of node v that is newly added by edge e , assuming a roughly uniform distribution of k-mers along the sequence (a simplification that holds well enough in practice):

$$K(e) = \beta \cdot k_{\neg h}(v) \cdot \frac{l(v) - l(e)}{l(v)} \quad (4)$$

The hyperparameter β controls the weight of this penalty in the overall beam score.

Model Score Penalty ($M(e)$): Edges are penalized based on their model score $m(e) \in [0, 1]$. To discourage high-confidence anomalous connections strongly, we use a quadratic penalty:

$$M(e) = \gamma \cdot m(e)^2 \quad (5)$$

where γ controls the sensitivity of the penalty to prediction uncertainty.

Putting all components together, the final beam score for edge $e = (u, v)$ is:

$$S(e) = \alpha \cdot (l(v) - l(e)) - \beta \cdot k_{-h}(v) \cdot \frac{l(v) - l(e)}{l(v)} - \gamma \cdot m(e)^2 \quad (6)$$

We conducted an ablation study demonstrating that each component of the beam search scoring function is critical for optimal performance. Omitting the model score penalty reduces contiguity, removing the k-mer penalty eliminates the ability to properly phase the assembly, and excluding the length reward results in highly fragmented assemblies. The full results can be found in Appendix E.

7 Evaluation

We evaluate our methods’ capacity on various real genomes. Our model is trained on synthetic human data based on a single reference genome: I002C [8]. For evaluation, we focus exclusively on real-world data and assess assembly quality across a range of datasets consisting of a different human genome and 6 primate species genomes. All experiments are conducted at the full diploid genome level, using datasets for which high-quality reference genomes are available.

7.1 Training Dataset Overview

We construct a training dataset, consisting of graphs generated from all chromosomes of the human genome I002C, with about 7.7M nodes and 10.5M edges across training and validation splits, and an average degree of 2.7. Roughly 85% of edges are labeled as correct, while the remaining 15% correspond to false edges. A detailed breakdown of the training dataset statistics is provided in Appendix G.

7.2 Evaluation Dataset Overview

The evaluation set comprises seven genomes: The human (*H. sapiens*) reference HG002 by Nurk et al. 2022 [12], and the six T2T ape references created by Yoo et al. 2024 [21] : *P. paniscus* (bonobo), *G. gorilla* (gorilla), *P. troglodytes* (chimpanzee), *S. syndactylus* (siamang), *P. abelii* (Sumatran orangutan), and *P. pygmaeus* (Bornean orangutan).

For *P. troglodytes*, *S. syndactylus*, *P. abelii*, and *P. pygmaeus*, no parental short-read data were available. Instead, we generated unique k-mers directly from the reference genomes. To approximate read input, we duplicated each reference sequence file 30 times and treated them as read files. The resulting “synthetic” YAK files were then used in place of real-world unique k-mer files typically derived from Illumina parental data. All tested methods use these YAK files for fair comparison. The exact files and coverages used for the evaluation dataset are detailed in Appendix H

7.3 Training Setup

To allow for mini-batch training on large graphs, we partition each graph using METIS into subgraphs containing at most 40,000 nodes (which is approximately the graph size to fit the full GPU memory). Each resulting subgraph is treated as a training batch. More training details, such as the final model configuration and the hardware used, can be found in Appendix I.

7.4 Genome Assembly Results

We benchmark three approaches: a greedy variant of DipGNNome, the full DipGNNome algorithm, and `hifiasm`. The greedy variant is identical to DipGNNome except that it replaces beam-search with a purely greedy expansion strategy, always selecting the next best node (according to the same scoring metric as the beam-search-based algorithm).

Table 1 shows the results including assembly length, duplication rate (Rdup), contiguity (NG50 and NGA50), and haplotype error rates (YAK Switch Error and Hamming Error) for paternal (P) and maternal (M) haplotypes. Values are shown as P/M (paternal/maternal). Hamming and switch error rates were computed using `yak` [4], while all other metrics (length, duplication rate, NG50, NGA50) were computed using `minigraph` [9]. Lower duplication and error rates indicate better assembly quality, while higher NGA50 reflects better (correct) contiguity. NG50 and total length alone do not necessarily indicate better or worse assembly quality. However, differences between NG50 and NGA50 display misassemblies.

Across most genomes, `hifiasm` achieves the lowest Hamming error rates and is generally on par with or superior to DipGNNome across the majority of haplotypes, particularly in terms of phasing accuracy. Nonetheless, DipGNNome remains competitive, often reaching comparable NGA50 values and in some cases surpassing `hifiasm`.

Most notably, on the orangutan genomes, DipGNNome shows clear advantages: for *P. pygmaeus*, it achieves substantially higher NGA50 for both haplotypes while maintaining comparable or slightly improved Hamming error, and for *P. abelii*, it also improves contiguity (NGA50) for one haplotype despite a moderate increase

Table 1. Comparison of DipGNNome and hifiasm on different genomes. The '*' marks genomes, where the parental k-mers are created synthetically.

Genome	Metric	DipGNNome		hifiasm
		Greedy	Beam-Search	
<i>H. sapiens</i>	Length (Mbp)	2840.0 / 2946.1	2856.4 / 2970.3	2938.5 / 3032.3
	Rdup (%)	0.1 / 0.1	0.2 / 0.3	0.8 / 0.2
	NG50 (Mbp)	35.7 / 25.7	63.0 / 65.2	56.0 / 59.4
	NGA50 (Mbp)	33.6 / 25.7	48.4 / 49.3	49.7 / 58.6
	Switch Err (%)	1.1 / 1.2	1.1 / 1.2	0.8 / 1.0
	Hamming Err (%)	2.2 / 2.5	1.9 / 3.0	0.8 / 0.8
<i>P. paniscus</i>	Length (Mbp)	3114.5 / 2934.2	3125.5 / 2947.4	3204.4 / 3066.0
	Rdup (%)	2.0 / 0.8	1.9 / 1.0	1.2 / 1.1
	NG50 (Mbp)	52.6 / 48.5	94.0 / 70.3	100.1 / 63.0
	NGA50 (Mbp)	35.2 / 39.6	50.1 / 42.3	55.2 / 47.1
	Switch Err (%)	0.3 / 1.8	0.3 / 1.7	0.2 / 1.5
	Hamming Err (%)	1.3 / 2.3	1.7 / 3.1	0.1 / 1.3
<i>G. gorilla</i>	Length (Mbp)	3366.2 / 3267.0	3389.8 / 3281.6	3528.4 / 3351.6
	Rdup (%)	1.1 / 1.4	1.2 / 1.4	1.2 / 1.1
	NG50 (Mbp)	69.0 / 44.3	108.1 / 100.6	94.7 / 82.6
	NGA50 (Mbp)	45.7 / 33.6	56.1 / 48.9	55.5 / 48.6
	Switch Err (%)	0.3 / 0.2	0.3 / 0.2	0.2 / 0.2
	Hamming Err (%)	1.1 / 1.0	1.4 / 1.0	0.2 / 0.2
<i>P. troglodytes</i> *	Length (Mbp)	3056.0 / 2938.1	3080.7 / 2956.3	3149.0 / 3032.9
	Rdup (%)	0.1 / 0.1	0.5 / 0.2	0.1 / 0.1
	NG50 (Mbp)	72.1 / 74.8	136.8 / 136.6	126.0 / 121.9
	NGA50 (Mbp)	62.2 / 68.0	102.8 / 76.1	101.9 / 121.9
	Switch Err (%)	0.1 / 0.2	0.1 / 0.2	0.1 / 0.2
	Hamming Err (%)	0.3 / 0.4	0.7 / 1.3	0.1 / 0.1
<i>S. syndactylus</i> *	Length (Mbp)	3131.6 / 3029.6	3133.5 / 3038.1	3230.0 / 3127.6
	Rdup (%)	0.2 / 0.1	0.3 / 0.3	0.1 / 0.7
	NG50 (Mbp)	67.6 / 50.7	90.8 / 69.6	84.9 / 75.4
	NGA50 (Mbp)	55.4 / 45.8	56.1 / 58.6	76.1 / 38.7
	Switch Err (%)	0.1 / 0.2	0.1 / 0.2	0.1 / 0.2
	Hamming Err (%)	0.3 / 0.4	0.5 / 0.6	0.1 / 0.1
<i>P. abelii</i> *	Length (Mbp)	3297.8 / 3487.0	3332.7 / 3522.1	2967.5 / 3361.0
	Rdup (%)	10.5 / 12.7	11.1 / 13.6	2.3 / 7.5
	NG50 (Mbp)	55.9 / 53.7	79.8 / 80.8	40.1 / 94.2
	NGA50 (Mbp)	43.0 / 40.6	45.7 / 57.5	34.1 / 65.3
	Switch Err (%)	6.5 / 4.6	6.5 / 4.6	4.8 / 3.8
	Hamming Err (%)	6.3 / 4.4	6.7 / 4.4	5.0 / 3.8
<i>P. pygmaeus</i> *	Length (Mbp)	3166.4 / 3272.6	3176.3 / 3304.0	3092.5 / 3212.2
	Rdup (%)	5.8 / 5.8	6.2 / 6.4	2.7 / 6.5
	NG50 (Mbp)	51.8 / 52.5	108.2 / 100.9	65.8 / 85.0
	NGA50 (Mbp)	45.6 / 51.6	78.9 / 92.2	55.7 / 47.4
	Switch Err (%)	7.3 / 8.9	7.2 / 8.8	6.9 / 8.7
	Hamming Err (%)	7.7 / 9.6	7.5 / 9.0	7.8 / 10.7

in error rates. Overall, while **hifiasm** remains the stronger assembler, these results highlight the potential of machine learning-guided diploid assembly. The greedy variant consistently performs worse, with lower NGA50 and higher error rates, confirming the importance of our beam-search-based algorithm for robust path exploration.

8 Conclusion

This work presents the first successful application of deep learning to diploid *de novo* genome assembly, demonstrating that GNNs can effectively resolve complex assembly graphs. Building on ideas from **GNNome** [18], we introduce improvements to both the data processing pipeline and the assembly algorithm.

We establish a framework for generating diploid graphs with ground-truth edge correctness when read coordinates originate from distinct reference systems for the same loci. This enables the construction of training-ready diploid graphs in DGL or PyG format. In addition, we propose a beam-search-based path-finding algorithm with strategies such as beam merging, which is particularly effective for navigating string-like graph structures.

Across most genomes, **DipGNNome** achieves results comparable to **hifiasm**, and in one case (*P. pygmaeus*) surpasses the current state of the art. Orangutan genomes are known to be especially challenging due to a higher number of acrocentric chromosomes and extensive tandem segmental duplications [22], suggesting that our approach may be particularly useful in difficult assembly settings where existing methods begin to struggle.

While contiguity and switch error rates are competitive, **DipGNNome** shows higher Hamming error, likely due to the absence of a polishing stage. Unlike **hifiasm**, which integrates read-level refinement, our method operates directly on the raw graph and identifies paths in a Hamiltonian-like manner without reusing nodes or introducing additional edges. This highlights a key trade-off between structural simplicity and sequence-level accuracy.

Performance is also influenced by training data diversity. The current model is trained only on human genomes, which may limit generalization to more distant species. Nevertheless, we observe successful transfer to other primates, and the framework can be readily extended to incorporate more diverse training data.

Finally, **DipGNNome** depends on the quality of the input graph. Since it selects only among existing edges, missing connections can lead to fragmentation or incomplete assemblies. Addressing this limitation, along with improving generalization across species and sequencing technologies, represents a natural direction for future work.

Overall, these results indicate that learning-based approaches are a viable and flexible complement to existing assembly methods, with clear potential for further improvements as models, training data, and integration strategies continue to evolve.

References

1. Battistella, E., Maheshwari, A., Ekim, B., Berger, B., Popic, V.: ralphi: a deep reinforcement learning framework for haplotype assembly. In: International Conference on Research in Computational Molecular Biology. pp. 349–353. Springer (2025)
2. Bresson, X., Laurent, T.: Residual gated graph convnets. arXiv preprint arXiv:1711.07553 (2017)
3. Cheng, H., Asri, M., Lucas, J., Koren, S., Li, H.: Scalable telomere-to-telomere assembly for diploid and polyploid genomes with double graph. *Nature Methods* pp. 1–4 (2024)
4. Cheng, H., Concepcion, G.T., Feng, X., Zhang, H., Li, H.: Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. *Nature methods* **18**(2), 170–175 (2021)
5. Ke, Z., Vikalo, H.: A convolutional auto-encoder for haplotype assembly and viral quasispecies reconstruction. *Advances in Neural Information Processing Systems* **33**, 13493–13503 (2020)
6. Ke, Z., Vikalo, H.: A graph auto-encoder for haplotype assembly and viral quasispecies reconstruction. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 34, pp. 719–726 (2020)
7. Koren, S., Rhie, A., Walenz, B.P., Dilthey, A.T., Bickhart, D.M., Kingan, S.B., Hiendleder, S., Williams, J.L., Smith, T.P., Phillippy, A.M.: De novo assembly of haplotype-resolved genomes with trio binning. *Nature biotechnology* **36**(12), 1174–1182 (2018)
8. LBCB-SCI: Telomere-to-telomere diploid Indian Genome. <https://github.com/lbcb-sci/I002C> (2024), <https://github.com/lbcb-sci/I002C>, accessed: 28.12.2024
9. Li, H., Feng, X., Chu, C.: The design and construction of reference pangenome graphs with minigraph. *Genome biology* **21**, 1–19 (2020)
10. Luu, P.L., Ong, P.T., Dinh, T.P., Clark, S.J.: Benchmark study comparing liftover tools for genome conversion of epigenome sequencing data. *NAR genomics and bioinformatics* **2**(3), lqaa054 (2020)
11. Myers, E.W.: The fragment assembly string graph. *Bioinformatics* **21**(suppl_2), ii79–ii85 (2005)
12. Nurk, S., Koren, S., Rhie, A., Rautiainen, M., Bzikadze, A.V., Mikheenko, A., Vollger, M.R., Altemose, N., Uralsky, L., Gershman, A., et al.: The complete sequence of a human genome. *Science* **376**(6588), 44–53 (2022)
13. Ono, Y., Asai, K., Hamada, M.: Pbsim: Pacbio reads simulator—toward accurate genome assembly. *Bioinformatics* **29**(1), 119–121 (2013)
14. Rautiainen, M., Nurk, S., Walenz, B.P., Logsdon, G.A., Porubsky, D., Rhie, A., Eichler, E.E., Phillippy, A.M., Koren, S.: Telomere-to-telomere assembly of diploid chromosomes with verkko. *Nature Biotechnology* **41**(10), 1474–1482 (2023)
15. Šimunović, M., Šikić, M., Bankevich, A.: Gnndebugger: Gnn based error correction in de bruijn graphs. *bioRxiv* pp. 2025–05 (2025)
16. Talenti, A., Prendergast, J.: nf-lo: a scalable, containerized workflow for genome-to-genome lift over. *Genome Biology and Evolution* **13**(9), evab183 (2021)
17. Vaser, R., Šikić, M.: Time-and memory-efficient genome assembly with raven. *Nature Computational Science* **1**(5), 332–336 (2021)
18. Vrček, L., Bresson, X., Laurent, T., Schmitz, M., Kawaguchi, K., Šikić, M.: Geometric deep learning framework for de novo genome assembly. *Genome research* **35**(4), 839–849 (2025)

19. Xue, H., Rajan, V., Lin, Y.: Graph coloring via neural networks for haplotype assembly and viral quasispecies reconstruction. *Advances in Neural Information Processing Systems* **35**, 30898–30910 (2022)
20. Yang, C., Zhou, Y., Song, Y., Wu, D., Zeng, Y., Nie, L., Liu, P., Zhang, S., Chen, G., Xu, J., et al.: The complete and fully-phased diploid genome of a male han chinese. *Cell Research* **33**(10), 745–761 (2023)
21. Yoo, D., Rhie, A., Hebbar, P., Antonacci, F., Logsdon, G.A., Solar, S.J., Antipov, D., Pickett, B.D., Safonova, Y., Montinaro, F., et al.: Complete sequencing of ape genomes. *bioRxiv* pp. 2024–07 (2024)
22. Yoo, D., Rhie, A., Hebbar, P., Antonacci, F., Logsdon, G.A., Solar, S.J., Antipov, D., Pickett, B.D., Safonova, Y., Montinaro, F., et al.: Complete sequencing of ape genomes. *Nature* pp. 1–18 (2025)
23. Zhao, L., Akoglu, L.: Pairnorm: Tackling oversmoothing in gnns. In: *International Conference on Learning Representations* (2020), <https://openreview.net/forum?id=rkecl1rtwB>

A Coordinate Translation and Ground-Truth Verification

Diploid Coordinates Some reads lie within homozygous regions of the diploid genome, meaning that both the maternal and paternal references contain the same sequence at these positions. However, since the maternal and paternal genomes are not identical (due to structural variations occurring elsewhere) the coordinates of a homozygous position differ between the two haplotypes.

Consider an edge between two reads, where one read originates from one haplotype and the other from the opposite haplotype. As with any edge, we must verify whether it represents a valid overlap based on the reads’ positional information. However, because each haplotype has its own coordinate system, it is not sufficient to compare positions directly. Instead, we must convert the coordinates from one haplotype to the other’s system.

Liftover We transfer coordinates using `liftover`, specifically the Nextflow-Liftover (NF-LO) pipeline [16]. Given two reference genomes, NF-LO generates chain files that encode coordinate mappings between them. These chain files can be used with UCSC Liftover or, in our case, the Python-based `py-liftover` tool [10] to translate coordinates between haplotypes. For each read, we store both its original coordinates and its translated coordinates in the alternate haplotype’s system. If `py-liftover` fails (due to the absence of a corresponding mapping in the other haplotype) the translated coordinate is set to `None`.

During unitig construction, when multiple reads are merged, some unitigs may include reads from both haplotypes. In such cases, we refine or infer missing coordinates using the structure of the unitig. Specifically, we use the postfix lengths of reads within the unitig to compute their relative offset from the end of the unitig, which allows us to estimate a coordinate in the alternate haplotype even when `liftover` coordinates are unavailable. Figure 2 shows an example of this advanced coordinate computation.

Ground-Truth Verification When checking for edge correctness, we determine whether either the paternal or maternal coordinate system shows a valid overlap. If a correct overlap exists in at least one haplotype, we mark the edge as non-relocation (i.e., not false). For edges where we cannot determine correctness—due to missing coordinate mappings or ambiguity—we assign a separate label, `unknown`, and exclude these edges from training. Missing coordinates only occur if a unitig consists of only reads of a single haplotype and additionally converting via `liftover` to the other haplotype fails. An overlap can only not be evaluated if this is the case for both unitigs, connected over an edge, and happens very rarely.

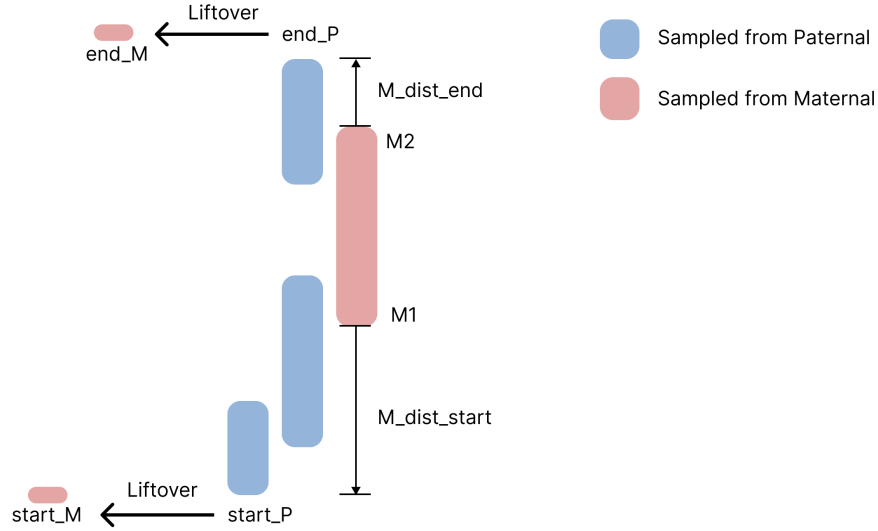


Fig. 2. Coordinate transfer. A unitig consists of multiple reads, and the paternal start and end coordinates are well defined here. Maternal coordinates can be determined either by lifting over the paternal endpoints or, if **liftover** fails, by computing them as $start_M = M1 - M_dist_start$ and $end_M = M2 + M_dist_end$.

B Full Assembly Pseudocode

ReverseComplementPath(path) replaces nodes n with their complements $n \oplus 1$ and then reverses the path. For example, the path with node IDs $(5, 10, 2)$ becomes $(2 \oplus 1, 10 \oplus 1, 5 \oplus 1) = (3, 11, 4)$.

LengthInBasePairs(path) takes a path of unitigs and computes the total length in base pairs by summing all unitig lengths and subtracting overlap lengths:

$$\text{LengthInBasePairs}(\text{path}) = \sum_{i=1}^n \text{len}(u_i) - \sum_{i=1}^{n-1} \text{ovl}(u_i, u_{i+1})$$

where $\text{path} = [u_1, u_2, \dots, u_n]$.

Input: Graph $G = (N, E)$, number of samples per round n , beam width k , minimum component length C_{\min} , minimum path length P_{\min}

Output: Set of full diploid assemblies

```

contigs  $\leftarrow \{m : [], p : []\}$ ;
foreach haplotype  $h \in \{m, p\}$  do
  components  $\leftarrow$  weakly connected components of  $G$ ;
  foreach component  $G_c \in$  components do
    while  $|G_c| \geq C_{\min}$  do
       $P \leftarrow []$ ; // Set of candidate paths
      for  $i = 1$  to  $n$  do
         $(u, u') \leftarrow$  random edge from  $E$ ;
         $(p_1, v_1, s_1) \leftarrow$  AssemblyBeamSearch( $G_c, u', k, h, \{u, u \oplus 1\}$ );
         $(p_2, v_2, s_2) \leftarrow$  AssemblyBeamSearch( $G_c, u \oplus 1, k, h, v_1$ );
         $p_2 \leftarrow$  ReverseComplementPath( $p_2$ );
         $p_{\text{full}} \leftarrow p_2 + p_1$ ;
         $v_{\text{full}} \leftarrow v_1 \cup v_2$ ;
        if LengthInBasePairs( $p_{\text{full}}$ )  $\geq P_{\min}$  then
          Append ( $p_{\text{full}}, v_{\text{full}}$ ) to  $P$ ;
        end
      end
       $(p_{\text{best}}, v_{\text{best}}) \leftarrow$  longest path in  $P$ ;
      Append  $p_{\text{best}}$  to contigs( $h$ );
      MarkVisited( $G_c, v_{\text{best}}$ );
      Remove all nodes in  $v_{\text{best}}$  and their incident edges from  $G_c$ ;
    end
  end
end
return contigs

```

Algorithm 1: Full Assembly: Complete diploid genome assembly algorithm that performs independent maternal and paternal passes through weakly connected components, using beam search with bidirectional path extension

C AssemblyBeamSearch Pseudocode

Input: Graph component $G_c = (N, E)$, starting node s , beam width k ,
haplotype $h \in \{m, p\}$, visited set V

Output: Best assembly path and visited nodes

$B_{\text{new}} \leftarrow \{([s], V, 0)\}$; // Each beam is a tuple: (path, visited set,
total score)

best_beam $\leftarrow ([s], V, 0)$ **while** $B_{\text{new}} \neq \emptyset$ **do**

```

|    $B_{\text{old}} \leftarrow B_{\text{new}};$ 
|    $B_{\text{new}} \leftarrow \emptyset;$ 
|   foreach  $(p, V, s) \in B_{\text{old}}$  do
|   |    $u \leftarrow$  last node in  $p$ ;
|   |   foreach  $\text{edge } (u, u') \in E$  do
|   |   |   if  $u' \notin V$  and  $u' \oplus 1 \notin V$  then
|   |   |   |    $p_{\text{new}} \leftarrow p + u';$ 
|   |   |   |    $V_{\text{new}} \leftarrow V \cup \{u', u' \oplus 1\};$ 
|   |   |   |    $s_{\text{new}} \leftarrow s + \text{EdgeScore}(u, u', h);$ 
|   |   |   |   Add  $(p_{\text{new}}, V_{\text{new}}, s_{\text{new}})$  to  $B_{\text{new}};$ 
|   |   |   end
|   |   end
|   end
|    $B_{\text{new}} \leftarrow \text{MergeBeams}(B_{\text{new}});$       // Filter dominated beams
|   Sort( $B_{\text{new}}$ );      // Sort beams by total score in descending order
|   Prune  $B_{\text{new}}$  to top  $k$  beams;
|   if any beam in  $B_{\text{new}}$  is better than best_beam then
|   |   best_beam  $\leftarrow$  best beam in  $B_{\text{new}};$ 
|   end
| end
return best_beam;
```

Algorithm 2: AssemblyBeamSearch: Beam search algorithm for genome assembly that maintains k candidate paths, excludes visited nodes and complements, and returns the highest-scoring path

D MergeBeams Pseudocode

Input: Set of beams B
Output: Merged set of beams B
 $B_{\text{new}} \leftarrow \emptyset$;
foreach $b \in B$ **do**
 $(p, v, s) \leftarrow b$;
 $u \leftarrow$ last element of p ;
 $\text{keep} \leftarrow \text{True}$;
 foreach $(p', v', s') \in B \setminus \{b\}$ **do**
 if $u \in p'$ **then**
 $i_u \leftarrow$ index of u in p' ;
 $p'' \leftarrow p'[0 : i_u + 1]$;
 $s'' \leftarrow$ score of p'' ;
 if $s > s''$ **then**
 replace p'' in (p', v', s') with (p, v, s) ;
 end
 else
 $\text{keep} \leftarrow \text{False}$;
 end
 break;
 end
 end
 if keep **then**
 add (p, v, s) to B_{new} ;
 end
end
return new;

Algorithm 3: MergeBeams: Merges overlapping beams by retaining only the highest-scoring path when multiple beams reach the same node

E DipGNNome Ablation Study

In this section, we compare the main results of DipGNNome with results after removing components from the beam search score heuristic.

The heuristic is:

$$S(e) = L(v) - K(v) - M(e) \quad (7)$$

Length reward L , minus K-mer penalty K , minus Model Score penalty M . We compute assemblies without either of these three components and compare with the main results.

F Parameter Configuration

Table shows the parameters used in all our experiments for the Assembly Algorithm.

In all configurations, c_b denotes the *cutting threshold* for removing edges classified as “bad edges,” while c_t denotes the *cutting threshold* for removing edges

Table 2. Ablation study of DipGNNome. The final column shows the full Beam Search model (beam width = 10).

	Metric	No Len.	Pen.	No k-mer	Pen.	No Model Scores	Beam Search
<i>H. sapiens</i>	Length (mB)	1546.3 / 1631.1	2959.7 / 3028.3	2999.3 / 3095.5		2856.4 / 2970.3	
	Rdup (%)	0.1 / 0.2	1.2 / 1.5	1.4 / 1.2		0.2 / 0.3	
	NG50 (mB)	0.2 / 0.2	52.9 / 45.2	78.4 / 76.2		63.0 / 65.2	
	NGA50 (mB)	0.2 / 0.2	38.9 / 37.0	47.0 / 46.4		48.4 / 49.3	
	Switch (%)	1.0 / 0.8	0.9 / 1.5	1.0 / 1.2		1.1 / 1.2	
	Hamm (%)	0.9 / 0.7	21.1 / 34.3	1.7 / 2.6		1.9 / 3.0	
<i>P. paniscus</i>	Length (mB)	2057.6 / 1813.7	3404.1 / 3253.8	3175.4 / 2993.1		3162.5 / 2980.8	
	Rdup (%)	5.6 / 0.3	8.7 / 7.6	2.4 / 1.5		2.3 / 1.3	
	NG50 (mB)	0.9 / 0.4	127.2 / 94.9	77.8 / 64.0		109.0 / 64.9	
	NGA50 (mB)	0.8 / 0.4	56.9 / 58.8	45.6 / 39.6		57.4 / 50.0	
	Switch Err (%)	0.5 / 1.8	0.9 / 0.9	0.3 / 1.6		0.3 / 1.6	
	Hamming Err (%)	1.6 / 1.6	27.0 / 22.2	1.9 / 2.9		2.3 / 3.6	
<i>G. gorilla</i>	Length (mB)	1856.6 / 1854.3	3727.8 / 3763.6	3649.4 / 3533.7		3485.9 / 3411.4	
	Rdup (%)	0.5 / 1.1	7.2 / 8.4	2.1 / 2.4		1.7 / 1.8	
	NG50 (mB)	0.3 / 0.2	116.6 / 105.0	116.6 / 116.1		108.7 / 78.6	
	NGA50 (mB)	0.1 / 0.0	60.0 / 63.8	60.0 / 56.0		60.0 / 56.1	
	Switch Err (%)	0.2 / 0.1	0.2 / 0.2	0.3 / 0.2		0.3 / 0.2	
	Hamming Err (%)	0.3 / 0.2	30.3 / 27.2	1.8 / 1.3		1.6 / 1.2	

classified as “translocation edges.” Higher values of c_b or c_t make the pruning step more selective, keeping more edges in the graph.

G Training Dataset Overview

This dataset represents complete genome graphs, created out of reads of all chromosomes of the respective organisms. Table 4 summarizes its key statistics. We train on graphs from the human genome I002C only.

The training portion contains 6.86 million nodes connected by 9.38 million edges, while the validation set comprises 0.85 million nodes and 1.17 million edges. The combined dataset has 7.71 million nodes and 10.54 million edges, with an average degree of about 2.74, indicating slightly sparser connectivity than the chromosome-level dataset (3.02).

Across both splits, 84.56% of the edges are labeled as correct, while the remaining 15.46% are false edges. Among the false edges, relocation edges are the most frequent (9.51% of all edges), followed by inversion edges (4.03%), translocation edges (1.91%), and a negligible fraction of unknown edges (0.01%).

H Real Data Details

We use the diploid T2T assembly of HG002 [20]. We include Gorilla, Bonobo, Siamang, Chimpanzee, and Orangutan as described in [21]. Coverage is computed from genome size and read depth.

Table 3. Beam search algorithm configuration and heuristic parameters.

Assembly Algorithm Parameters	
Beam width (k)	5
Samples (n)	100
Min. contig length (P_{\min})	100k
Min. component length (C_{\min})	25
α	0.0001
β	3
γ	25
c_b	> 0.9
c_t	> 0.5

Table 4. Comprehensive Dataset Statistics (Edge Type Breakdown) — Full Genome Dataset

Metric	Training Validation		Combined
Graph Structure			
Total Nodes	6,855,076	851,970	7,707,046
Total Edges	9,375,111	1,168,891	10,544,002
Average Degree	2.735	2.744	2.735
Edge Classes			
Correct Edges	8,102,855	1,098,974	9,201,829 (84.56%)
Total False Edges	1,272,256	69,917	1,628,805 (15.46%)
False Edge Distribution			
Relocation Edges	890,937	111,089	1,002,026 (9.51%)
Unknown Edges	424	53	477 (0.01%)
Inversion Edges	377,113	47,557	424,670 (4.03%)
Translocation Edges	178,994	22,638	201,632 (1.91%)

I Training Configuration

The final model configuration uses 8 GNN layers and an embedding size of 512 for both node and edge features. A dropout rate of 0.3 is applied to reduce overfitting, and Pair Normalization is used throughout the network to enhance training stability. The model is optimized using the Adam optimizer with a learning rate of 0.0001.

Training is conducted on a single GPU (NVIDIA A100-PCIE-40GB), using mini-batch gradient descent and early stopping based on the validation loss. All models are implemented in PyTorch.

Table 5. Read datasets for human and ape genomes

Species	File	Depth
HG002	m64004_210224_230828	4.77×
	m64014_210227_165255	5.05×
	m64015e_210223_010616	5.32×
	m64015e_210224_100310	5.09×
	Total	20.23×
Gorilla	m54329U_210319_174352	4.32×
	m54329U_211102_230231	6.03×
	m54329U_211107_082940	5.15×
	m64076_230112_193924	4.26×
	Total	19.76×
Bonobo	m54329U_210509_045858	1.70×
	m54329U_210510_223954	2.12×
	m54329U_210512_043859	1.92×
	m54329U_211104_082916	6.88×
	m64076_210509_004533	2.17×
	m64076_210511_192234	1.91×
	m64076_210802_234415	4.36×
	Total	21.06×
Siamang	m54329U_210828_003431	4.30×
	m54329U_210829_112929	5.33×
	m54329U_210901_104742	4.92×
	m54329U_210902_233521	4.11×
	m54329U_210904_103226	1.58×
	Total	20.24×
Chimpanzee	m54329U_220226_122930	5.95×
	m54329U_220304_132403	6.54×
	m64076_210810_005444	4.27×
	m64076_210813_021703	4.02×
	Total	20.78×
Bornean	m54329U_210506_061718	3.96×
Orangutan	m54329U_220303_022849	6.30×
	m64076_210430_224715	3.91×
	m64076_210505_002126	3.81×
	m64076_210506_062052	3.71×
	Total	21.69×
Sumatran	m64076_210219_011735	4.17×
Orangutan	m54329U_210228_013345	3.16×
	m54329U_210404_020346	2.72×
	m54329U_220227_232630	5.76×
	m54329U_220313_173323	6.77×
	Total	22.58×