BRANCHING FLOWS: DISCRETE, CONTINUOUS, AND MANIFOLD FLOW MATCHING WITH SPLITS AND DELETIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Diffusion and flow matching approaches to generative modeling have shown promise in domains where the number of elements in a state is fixed in advance (e.g. images), but require *ad hoc* solutions when, for example, the length of a response from a large language model, the number of atoms in a molecule, or the number of amino acids in a protein chain is not known *a priori*. Here we propose Branching Flows, a generative modeling framework that, like diffusion and flow matching approaches, transports a simple distribution to the data distribution. But in Branching Flows, the elements in the state evolve over a forest of binary trees, branching and dying stochastically with rates that are learned by the model. This allows the model to control, during generation, the number of elements in the sequence. We show that Branching Flows can compose with any flow matching base process on discrete sets, continuous Euclidean spaces, Riemannian manifolds, and 'multimodal' product spaces that mix these components, and we demonstrate distribution matching on small molecules and antibody sequences, and that this scales to complicated domains such as protein structures.

1 INTRODUCTION

For many applications that involve generating a sequence of elements, it isn't possible to know the required number of elements in advance. For sequences of discrete tokens this is well addressed by autoregressive (AR) models which control length by sampling stop tokens. Continuous AR models are possible (Billera et al., 2024) but underexplored, and current examples are limited in the range of problems they can address. Diffusion and flow matching methods avoid left-to-right AR structure and excel for structured grids and fixed-length sequences, but sacrifice the ability to naturally handle length variation.

One challenge not naturally addressed by fixed-length diffusion/flow models nor AR models (especially in the continuous state context) is what we call the 'unknown-length infix sampling' problem, where a segment of unknown length must be inserted into a sequence, conditioned on *a priori* known flanking regions. As a concrete example from protein design: conditioned on framework regions and some binding pose, if you wish to design the complementarity-determining regions of an antibody you must first specify how long they must be, which is absurd. There is a 'soft' version of this that arises for fixed-length diffusion/flow models, where flanking regions happen to resolve early during an inference trajectory, and the model cannot control the number of elements spanning them.

In the diffusion and flow matching context, Edit Flows addresses length variation by modeling variable-length discrete sequences via insertions and deletions (Havasi et al., 2025; Nguyen et al., 2025) and, in the purely-continuous domain, one attempt to develop variable-length diffusion models (Campbell et al., 2023) relies on an approximation related to the addition of new elements, and it is unclear how this scales to more complex examples.

2 BRANCHING FLOWS

Here we present Branching Flows, a framework for generative modeling over variable-length sequences where the elements can be continuous, manifold-valued, discrete, or combinations of these.

This is formulated in Generator Matching (Holderrieth et al., 2024), where a conditional process transports samples from a simple distribution to the data distribution. The generator of this is learned, in expectation, yielding a process whose marginals match those of the conditional process. Branching Flows augments a 'base' Markov generator on an element space with a branching and deletion process. A forest of trees is used to couple a sample from a simple distribution (with elements at the roots) with the data distribution (elements at the leaves). Like the processes used in phylogenetics (Felsenstein, 1981), the elements evolve independently along each branch but duplicate and decouple at bifurcations in the tree. This construction, composed with optional deletions, allows us to train a generative model over sequences of varying length.

In Generator Matching (Holderrieth et al., 2024; Lipman et al., 2024), conditioning on Z ensures that the conditional process terminates at a sample from the data distribution, and can be used to control the properties of the conditional paths (Tong et al., 2024). With Branching Flows, we make *extensive* use of Z, which includes: $X_1^{+\emptyset}$, which is a sample from the data distribution X_1 but augmented with "to be deleted states"; X_0, a sample from a simple prior (not necessarily the same length as X_1 but strictly shorter than $X_1^{+\emptyset}$); a forest of planar trees, T, with one tree per element of X_0; and 'anchors' on nodes of the trees, which live in the same state space as the elements of X_1. The anchors are fixed to the $X_1^{+\emptyset}$ elements on leaves, and control the conditional paths along internal branches of the trees where they can be chosen freely. For internal anchors, we use the midpoint of the two child anchors for continuous states and a masked token for discrete states. See Figure 1 (top left) and App. Sec. B.2 for details. While Z fixes the branching structure and the elements to be deleted, the branching and deletion times are sampled stochastically under the conditional process, from root

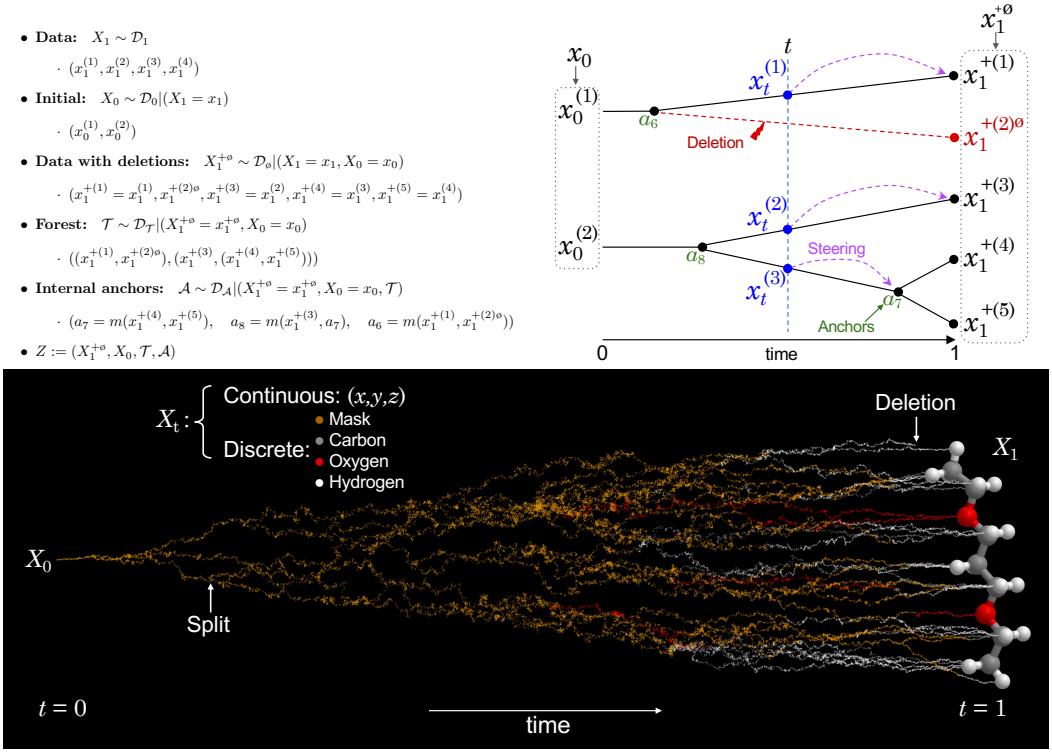

Figure 1: **Branching Flows construction.** (Top) Left outlines the sampling of Z when x_1 has 4 elements and x_0 has two elements, for a conditional bridge that will incur a single deletion (element $x_1^{+(2)\emptyset}$). $m(.,.)$ derives an anchor from two children, which could be e.g. a midpoint for continuous states and a mask token for discrete states. Right depicts the conditional sampling of $x_t|Z$, where elements can split and be deleted but only, in the conditional path, according to the pre-determined branching structure of the two trees $\mathcal{T}$ in Z. (Bottom) An illustration of an inference trajectory from a Branching Flows model where a variable-length state, with each element comprising a Euclidean position and a discrete label, is generated by a stochastic process with splits and deletions.

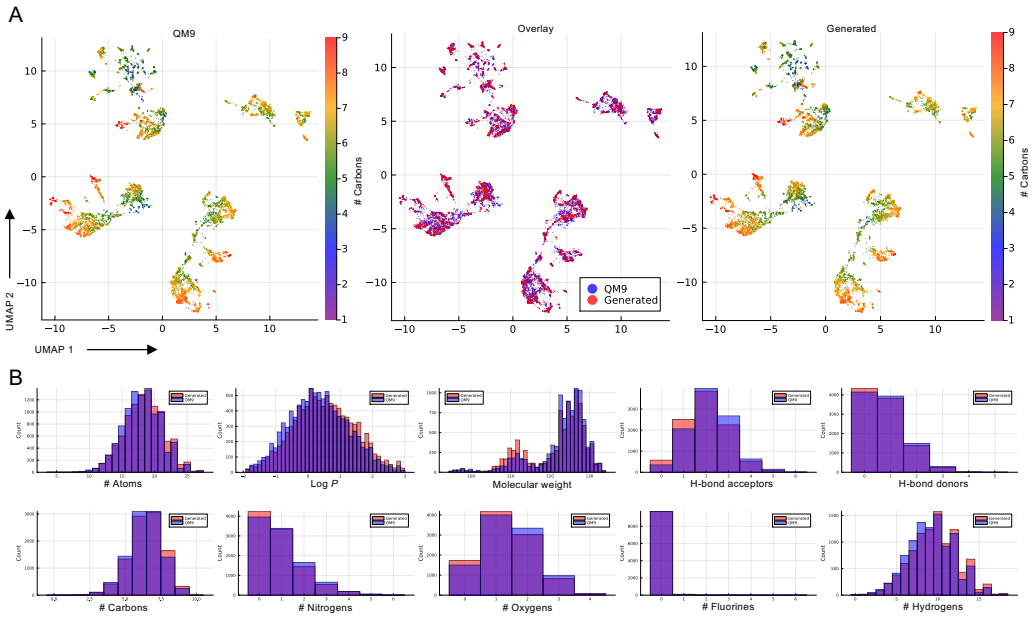

Figure 2: **QM9 data vs generated samples.** 10,000 random samples from the QM9 dataset vs 10,000 samples generated using a Branching Flows model. Panel A computes molecular fingerprints (using RDKit) and jointly embeds QM9 data and generated samples using UMAP. Left shows QM9-only embeddings, colored by the number of carbon atoms in each molecule; right shows generated samples, similarly colored, and center shows the overlap of QM9 (blue) vs Branching Flows generated samples (red), showing similar distributions. Panel B depicts property distributions from QM9 and Branching Flows generated samples, including atom counts, as well as a number of standard molecular descriptors computed with RDKit.

to tips. Each element in the state is updated using the base process generator, independently along each branch, conditioned to terminate, were it to reach $t = 1$ at the anchor associated with the node ahead of it. See Figure 1 (top right) and App. Sec. B.3. The resulting process is the composition of the base generator and a jump process governing split and deletion events. Because the elements evolve independently, conditioned on Z, along each branch, the loss decomposes into the sums of an elementwise base process loss, a split rate Poisson loss, and a deletion cross-entropy (see App. G.4). Training and sampling proceeds as in flow matching, using the conditional process to sample X_t states and training targets, with the model optimized to learn their expected generator, which is used to generate novel variable-length samples (see App. B.6).

3 Results

We consider three datasets: i) small molecules, where we jointly generate atom positions and atom labels, ii) Antibody heavy chains (discrete sequence only), and iii) protein backbones, comprising amino acid 'frames' (Jumper et al., 2021) with Euclidean positions & SO(3) orientations, and discrete amino acid sequence labels.

QM9. QM9 (Ramakrishnan et al., 2014) contains small organic molecules selected from the GDB (Generated Database) chemical space enumeration. We train Branching Flows to jointly generate atom coordinates and discrete labels; full data preprocessing, Branching Flows specification, and model/training details are in Appendix H.1. To investigate distribution matching, we generated 10k samples and compared these to 10k true QM9 samples (See App. H.1 for details). Figure 2 compares generated and data distributions for molecule fingerprints, which encode molecule properties into a fixed-length bit vector, embedded into two-dimensions by UMAP (McInnes et al., 2020), showing no apparent differences. Also shown are histograms of atom counts and molecule properties, with some subtle departures, most visible in LogP and molecular weight. We also compare 10,000 samples

Table 1: Distribution agreement ($1 - \mathrm{KS}_D$, higher is better) between properties of samples from each model vs. QM9 data. From left to right: Molecular weight, LogP, hydrogen-bond donors, hydrogen-bond acceptors, number of: carbons, hydrogens, oxygens, nitrogens, fluorines, and the total number of atoms.

Model	mw	LogP	hbd	hba	C	H	O	N	F	atoms
Branching Flows	0.9091	0.9276	0.9876	0.9259	0.9638	0.9408	0.9589	0.9672	0.9946	0.9457
TD-jumps	0.8867	0.8319	0.8726	0.8407	0.8144	0.8553	0.9316	0.8645	0.9678	0.8690

from a QM9-trained 'Transdimensional Jump' diffusion model (Campbell et al., 2023), using a Kolmogorov-Smirnov based statistic ($1 - KS_D$; see App. H.1) to quantify property agreement between samples and the data distribution (Table 1). For all metrics, Branching Flows better matches the data distribution. We explore this further with visualizations in Figure 5 (Appendix).

Antibody Sequences. We investigate a pure discrete sequence learning task: antibody amino acid sequence generation. We compare a Branching Flows model to an 'oracle-length' standard discrete flow matching model, which is trained knowing the true length, and sampled using a 'true' length from the data distribution. We also compare to a re-implementation of Edit Flows (Havasi et al., 2025). Full dataset details, Branching Flows specification, and model/training details are in Appendix H.2. Figure 6 (Appendix) shows the distributions of statistics (App. H.2) from antibody sequences generated by both the Branching Flows and oracle-length models, perplexity-over-time training dynamics, and seqUMAP (Hanke et al., 2022) representations of the $10\,000$ generated sequences. Branching Flows is comparable to the oracle length model, can learn sequence-position-specific amino acid frequencies (Figure 8 (Appendix)), and outperforms Edit Flows on property matching (Table 2 and Figure 9 (Appendix)).

Protein Structure and Sequence. Starting with ChainStorm (Oresten et al., 2025), a multimodal flow matching model that jointly generates protein backbones and amino acid sequence labels, as a base model, we finetuned two variants: 'BranchChain', where the designable residues are complete chains, and BranchSegment which designs random-length contiguous segments (see App. H.3). Figure 3 shows that BranchChain samples backbones with protein-like geometry, consistently recovered by the Boltz2 folding model when provided with a pseudo multiple sequence alignment (Oresten et al., 2025), and Figure 11 (Appendix) shows that BranchSegment is a promising solution to the 'unknown length infix sampling' problem, sampling antibody CDR3s of variable length (see Appendix H.3).

4 Discussion

We have shown that Branching Flows is a capable 'distribution-learner' of multimodal sequences that vary in length, addressing a previously-unsolved fundamental problem in generative modeling. For example it is, to our knowledge, the first method that can sample an infix of unknown length from a multimodal sequence. For now, we restrict ourselves to demonstrating that Branching Flows works in a variety of domains, rather than extensive comparisons and benchmarks on downstream tasks, as performance on benchmarks is often more sensitive to practically useful but theoretically uninteresting domain-specific adjustments to the model and training details, which is orthogonal to the purpose of this manuscript. See Appendix I for a more detailed discussion of related work (Campbell et al., 2023; Havasi et al., 2025; Nguyen et al., 2025; Schneuing et al., 2025).

Table 2: Distribution agreement ($1-\mathrm{KS}_D$, higher is better) between properties of samples from each model vs. validation sequences. From left to right: Sequence Length, Diversity (within-samples min Cosine distance), Novelty (minimum edit distance to true sequence), CDR3 length, and AbNatiV 'humanness' score. Histograms of these properties are shown in Figure 7 (Appendix).

Method	Length	Diversity	Novelty	CDR3 length	AbNatiV
Oracle length	0.995	0.899	0.903	0.984	0.943
Branching Flows	0.986	0.894	0.888	0.989	0.931
Edit Flows	0.989	0.713	0.770	0.983	0.769

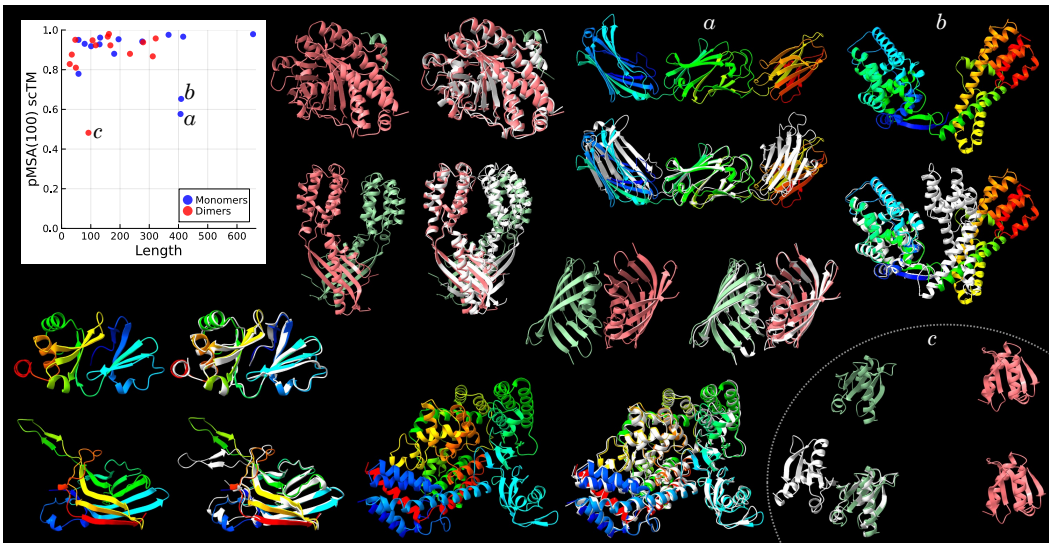

Figure 3: **Branching Flows protein samples.** Top left inset shows pseudoMSA scTM refolding scores, and selected samples are shown (rainbow colored for single chains, and per-chain for dimers). To the right or below each is shown an overlay of the sample and the refolded structure (grey). Specifically, the three worst scTMs are depicted (labelled a, b, and c on the inset and the main panel) showing that the low scTMs are driven by unpredictability due to flexible linkers, or incorrect dimerization prediction.

For the QM9 small-molecule dataset, we compare samples from our Branching Flows-trained model with those from the Transdimensional Jump Diffusion model of Campbell et al. (2023), showing favorable distribution matching for Branching Flows. We suspect this difference is attributable to two things. Firstly, in transdimensional jump diffusion, while the conditional distribution of the element spawn point is Gaussian conditioned on the corresponding X_1 element, the marginal distribution during inference is a (possibly infinite) mixture of Gaussians, which is challenging to approximate. Branching Flows avoids this by spawning new elements via it's splitting mechanism. Secondly, transdimensional jump diffusion handles discrete atom types by converting them to continuous vectors. In other contexts, this does not compare favorably to discrete character diffusion models (Sahoo et al., 2024).

For the antibody sequence modeling problem, we compare Branching Flows to Edit Flows (Havasi et al., 2025). Here the model architectures are matched between Branching Flows and Edit Flows, where the only difference is that the Edit Flows insertion head is as large as the alphabet size, whereas Branching Flows only needs one splitting rate per element. While we see favorable performance of Branching Flows, one caveat is that this is our own Edit Flows implementation since, at the time of writing, there was no reference implementation provided by the authors.

Protein structure generation, as a field, has matured to the point where models are practically used in an experimental setting. It has recently become clear that *in silico* metrics, mostly relying on protein folding models, are poor predictors of experimental success (Garcia et al., 2026). As such, attempting to create and experimentally (not *in silico*) validate a state-of-the-art protein design model is beyond the scope of the current work. For now we content ourselves in showing that i) Branching Flows models generate samples that have geometry consistent with a state-of-the-art folding model, and ii) that they have new capabilities related to length variation that, if experimentally validated, would extend the space of problems approachable with protein design.

Further work will explore the performance of Branching Flows in an array of different domains, and more systematically elucidate the design space of Branching Flows models, comparing the benefits of different components of the process (e.g. branching-only vs deletion-only vs branching & deletion), as well as hyperparameters such as hazard distributions, anchor merging strategies, and the optimal choice of base process for different kinds of problems.

REFERENCES

Giuseppina Albano and Virginia Giorno. Inference on the effect of non homogeneous inputs in Ornstein-Uhlenbeck neuronal modeling. *Mathematical Biosciences and Engineering*, 17(1):328–348, 2020. doi: 10.3934/mbe.2020018.

Lukas Billera, Anton Oresten, Aron Stålmarck, Kenta Sato, Mateusz Kaduk, and Ben Murrell. The continuous language of protein structure. *bioRxiv*, 2024. doi: 10.1101/2024.05.11.593685.

Lukas Billera, Hedwig Nora Nordlinder, and Ben Murrell. Time dependent loss reweighting for flow matching and diffusion models is theoretically justified. 2025. doi: 10.48550/arXiv.2511.16599.

Andrew Campbell, William Harvey, Christian Weilbach, Valentin De Bortoli, Tom Rainforth, and Arnaud Doucet. Trans-dimensional generative modeling via jump diffusion models. *arXiv*, 2023. doi: 10.48550/arXiv.2305.16261.

J. Dauparas, I. Anishchenko, N. Bennett, H. Bai, R. J. Ragotte, L. F. Milles, B. I. M. Wicky, A. Courbet, R. J. de Haas, N. Bethel, P. J. Y. Leung, T. F. Huddy, S. Pellock, D. Tischer, F. Chan, B. Koepnick, H. Nguyen, A. Kang, B. Sankaran, A. K. Bera, N. P. King, and D. Baker. Robust deep learning–based protein sequence design using proteinmpnn. *Science*, 378(6615):49–56, 2022. doi: 10.1126/science.add2187.

James Dunbar and Charlotte M. Deane. Anarci: antigen receptor numbering and receptor classification. *Bioinformatics*, 32(2):298–300, 2016. doi: 10.1093/bioinformatics/btv552.

Joseph Felsenstein. Evolutionary trees from dna sequences: a maximum likelihood approach. *Journal of molecular evolution*, 17(6):368–376, 1981. doi: 10.1007/BF01734359.

Mario Garcia, Sugyan M. Dixit, and Gabriel J. Rocklin. Evaluating zero-shot prediction of monomeric protein design success by alphafold, esmfold, and proteinmpnn. *Protein Science*, 35(2):e70453, 2026. doi: 10.1002/pro.70453.

Itai Gat, Tal Remez, Neta Shaul, Felix Kreuk, Ricky T. Q. Chen, Gabriel Synnaeve, Yossi Adi, and Y. Lipman. Discrete flow matching. *ArXiv*, abs/2407.15595, 2024. doi: 10.48550/arXiv.2407.15595.

Leo Hanke, Daniel J Sheward, Alec Pankow, Laura Perez Vidakovics, Vivien Karl, Changil Kim, Egon Urgard, Natalie L Smith, Juan Astorga-Wells, Simon Ekström, Jonathan M. Coquet, Gerald M. McInerney, and Ben Murrell. Multivariate mining of an alpaca immune repertoire identifies potent cross-neutralizing sars-cov-2 nanobodies. *Science advances*, 8(12):eabm0220, 2022. doi: 10.1126/sciadv.abm0220.

Marton Havasi, Brian Karrer, Itai Gat, and Ricky T. Q. Chen. Edit flows: Flow matching with edit operations. *ArXiv*, abs/2506.09018, 2025. doi: 10.48550/arXiv.2506.09018.

Peter Holderrieth, Marton Havasi, Jason Yim, Neta Shaul, Itai Gat, T. Jaakkola, Brian Karrer, Ricky T. Q. Chen, and Y. Lipman. Generator matching: Generative modeling with arbitrary markov processes. *ArXiv*, abs/2410.20587, 2024. doi: 10.48550/arXiv.2410.20587.

Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024. URL `https://kellerjordan.github.io/posts/muon/`.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021. doi: 10.1038/s41586-021-03819-2.

Aleksandr Kovaltsuk, Jinwoo Leem, Sebastian Kelm, James Snowden, Charlotte M Deane, and Konrad Krawczyk. Observed antibody space: A resource for data mining next-generation sequencing of antibody repertoires. *The Journal of Immunology*, 201(8):2502–2509, 10 2018. doi: 10.4049/jimmunol.1800708.

Yaron Lipman, Marton Havasi, Peter Holderrieth, Neta Shaul, Matt Le, Brian Karrer, Ricky T. Q. Chen, David Lopez-Paz, Heli Ben-Hamu, and Itai Gat. Flow matching guide and code. *ArXiv*, abs/2412.06264, 2024. doi: 10.48550/arXiv.2412.06264.

Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *ArXiv*, 2020. doi: 10.48550/arXiv.1802.03426.

John Nguyen, Marton Havasi, Tariq Berrada, Luke Zettlemoyer, and Ricky T. Q. Chen. Oneflow: Concurrent mixed-modal and interleaved generation with edit flows. *ArXiv*, 2025. doi: 10.48550/arXiv.2510.03506.

Anton Oresten, Kenta Sato, Aron Stålmarck, Lukas Billera, Hedwig Nora Nordlinder, Jack Collier Ryder, Mateusz Kaduk, and Ben Murrell. Spontaneous emergence of symmetry in a generative model of protein structure. *bioRxiv*, 2025. doi: 10.1101/2025.11.03.686219.

Saro Passaro, Gabriele Corso, Jeremy Wohlwend, Mateo Reveiz, Stephan Thaler, Vignesh Ram Somnath, Noah Getz, Tally Portnoi, Julien Roy, Hannes Stark, David Kwabi-Addo, Dominique Beaini, Tommi Jaakkola, and Regina Barzilay. Boltz-2: Towards accurate and efficient binding affinity prediction. *BioRxiv*, 2025. doi: 10.1101/2025.06.14.659707.

William Peebles and Saining Xie. Scalable diffusion models with transformers. *ArXiv*, 2023. doi: 10.48550/arXiv.2212.09748.

Raghunathan Ramakrishnan. Qm9pack: A python package for data-mining the qm9 dataset, 2024. URL `https://github.com/raghurama123/qm9pack`.

Raghunathan Ramakrishnan, Pavlo O Dral, Matthias Rupp, and O Anatole Von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific data*, 1(1):1–7, 2014. doi: 10.1038/sdata.2014.22.

Aubin Ramon, Montader Ali, Misha Atkinson, Alessio Saturnino, Kieran Didi, Cristina Visentin, Stefano Ricagno, Xing Xu, Matthew Greenig, and Pietro Sormanni. Assessing antibody and nanobody nativeness for hit selection and humanization with AbNatiV. *Nature Machine Intelligence*, 6:74–91, 2024. doi: 10.1038/s42256-023-00778-3.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models. *ArXiv*, 2024. doi: 10.48550/arXiv.2406.07524.

Arne Schneuing, Ilia Igashov, Adrian W. Dobbelstein, Thomas Castiglione, Michael Bronstein, and Bruno Correia. Multi-domain distribution learning for de novo drug design. *ArXiv*, abs/2508.17815, 2025. doi: 10.48550/arXiv.2508.17815.

Dmitry V Shcheblyakov, Irina A Favorskaya, Inna V Dolzhikova, Anastasia I Korobkova, Irina A Alekseeva, Ilias B Esmagambetov, Olga L Voronina, Amir I Tukhvatulin, Olga V Zubkova, Artem A Derkaev, Ekaterina I. Ryabova, Anna A. Iliukhina, Ilia D. Zorkov, Daria M. Grousova, Dmitriy A. Reshetnikov, Natalia N. Ryzhova, Ekaterina I. Ermolova, Marina S. Kunda, Ilya O. Matyuta, Konstantin M. Boyko, Vladimir O. Popov, Denis Y. Logunov, Nikolai N. Sluchanko, and Alexander L. Gintsburg. Ultra-potent rbm-specific single-domain antibody broadly neutralizes multiple sars-cov-2 variants with picomolar activity. *International Journal of Biological Macromolecules*, pp. 145386, 2025. doi: 10.1016/j.ijbiomac.2025.145386.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *ArXiv*, 2023. doi: 10.48550/arXiv.2104.09864.

Alexander Tong, Kilian Fatras, Nikolay Malkin, Guillaume Huguet, Yanlei Zhang, Jarrid Rector-Brooks, Guy Wolf, and Yoshua Bengio. Improving and generalizing flow-based generative models with minibatch optimal transport. *ArXiv*, 2024. doi: 10.48550/arXiv.2302.00482.

Oliver M Turnbull, Dino Oglic, Rebecca Croasdale-Wood, and Charlotte M Deane. p-iggen: a paired antibody generative language model. *Bioinformatics*, 40(11):btae659, 11 2024. doi: 10.1093/bioinformatics/btae659.

A Appendix

B Supplementary Methods

As Branching Flows requires some intricacy, for readability, we first include a supplementary methods description of the sort that might be found in the main text of a longer-form paper, which references formal arguments later in the appendix.

B.1 Generator Matching and Auxiliary Generator Matching

Following Generator Matching (Holderrieth et al., 2024; Lipman et al., 2024), for $t \in [0, 1]$, we seek a probability path p_t that interpolates between two fixed marginals p and q, where p is a distribution that is easy to sample from, and q is the data distribution. This goes via a conditional-path construction: Conditional on an auxiliary variable $z \in \mathcal{Z}$ with marginal distribution p_Z and a family of conditional probability measures $p_{t|Z}(\mathrm{d}x \mid z)$, the corresponding marginal path can be described by hierarchical sampling

$$Z \sim p_Z, \quad X_t|(Z = z) \sim p_{t|Z}(\mathrm{d}x \mid z) \;\Rightarrow\; X_t \sim p_t(\mathrm{d}x).$$

Choose the conditional family so that the boundary constraints $p_0 = p$ and $p_1 = q$ hold after marginalization. This can be achieved by sampling an $X_0 \sim p$ and an $X_1 \sim q$ and having the process start at X_0 and end at X_1. The goal is then to parameterize a generator $\mathcal{L}_t$ by a neural network $\mathcal{L}_t^\theta$ and train it to generate the marginal path p_t. This is done by first drawing samples from the conditional path $X_t \sim p_{t|Z}(\mathrm{d}x \mid z)$, and then training $\mathcal{L}_t^\theta$ against the generators of the conditional paths $\mathcal{L}_t^z$. Taking gradient steps under a 'conditional generator matching' loss (a Bregman divergence) encourages $\mathcal{L}_t^\theta$ to learn the expectation of the conditional generator which, via a gradient identity, generates the marginal path p_t.

By *'Auxiliary Generator Matching'* we mean: augment our process with G_t (with realisations denoted g_t) for each $t \in [0, 1]$, yielding joint conditional paths $(X_t, G_t)|(Z = z) \sim p_{t|Z}(\mathrm{d}x, \mathrm{d}g \mid z)$, but train a model that marginalizes over G_t and Z, learning a generator on X_t with marginals matching $X_t \sim p_t(\mathrm{d}x)$. This was used in Edit Flows (Havasi et al., 2025) for the discrete case, but see Appendix G.3 for our setting.

B.2 Z

At time t the state is an ordered list $X_t = (X_t^{(1)}, \ldots, X_t^{(L_t)})$ with variable length L_t, where each element $X_t^{(i)}$ is a random variable on some space $\mathcal{E}$. An observation from the data distribution is $x_1 = (x_1^{(1)}, \ldots, x_1^{(\ell_1)}) \sim q$, and a sample from the initial distribution is $x_0 = (x_0^{(1)}, \ldots, x_0^{(\ell_0)}) \sim p$, where ℓ_0 does not necessarily equal ℓ_1.

We augment our sequence at $t = 1$ by sampling and inserting 'to be deleted' states $\varnothing_1$, each also in $\mathcal{E}$, into X_1 yielding $X_1^{+\varnothing}$, such that $X_1 = \psi_\varnothing(X_1^{+\varnothing})$ where $\psi_\varnothing$ discards all 'to be deleted' states. We denote the length of $X_1^{+\varnothing}$ as $L_1^{+\varnothing}$, and require that $L_1^{+\varnothing} > L_0$ (condition 1).

To couple X_0 to $X_1^{+\varnothing}$, we sample an ordered list of labeled binary rooted plane trees, $\mathcal{T} = (\mathcal{T}_1, \ldots, \mathcal{T}_{L_0})$, where the root of each tree $\mathcal{T}_i$ is associated with the element $X_0^{(i)}$. Each element of $X_1^{+\varnothing}$ is associated with a single leaf of a single tree, such that the ordering of the leaves (since the trees are planar) matches the ordering of $X_1^{+\varnothing}$. All nodes in the tree are labeled 'surviving' except leaf nodes associated with 'to be deleted' states, which are labeled 'deleted'. The number of descendants of node n_i (including surviving and deleted) is denoted w_i.

To steer the conditional process along internal branches, we introduce 'anchors' $\mathcal{A}$, where each node's anchor A_i is a random variable on $\mathcal{E}$, requiring the anchor of the surviving leaves to equal the leaf-associated element of $X_1^{+\varnothing}$ (condition 2). Internal anchors $\mathcal{A}^{\mathcal{I}}$ have no such constraint, and can

e.g. be chosen to steer the process along internal branches towards the centroid of the descendant elements.

For Branching Flows, $Z \sim p_Z$ is a tuple $(X_1^{+\emptyset}, X_0, \mathcal{T}, \mathcal{A})$, which is obtained by first sampling $X_1 \sim q$, and then sampling from $p(x_1^{+\emptyset}, x_0, \mathcal{T}, \mathcal{A} \mid X_1 = x_1)$ according to a scheme that satisfies conditions 1 and 2 above, and that also admits a simple marginal $p(x_0)$. Note that, since $\mathcal{T}$ carries labels for which leaves are deleted, X_1 is recoverable from Z. To sample Z in practice, to be able to ensure that conditions 1 and 2 are met, we factorize $p(x_1^{+\emptyset}, x_0, \mathcal{T}, \mathcal{A} \mid x_1)p(x_1)$ as shown in fig. 1. See App. C for concrete schemes for sampling Z, especially $\mathcal{T}$ and $\mathcal{A}$.

B.3 Conditional Paths

We construct conditional paths where elements evolve along branches of our trees, splitting and deleting, to terminate at X_1. We need to specify i) how the elements evolve along a branch, ii) what happens when they split, iii) the per-element split rate, and iv) the per-element deletion rate.

Between split or deletion events, each element evolves independently of the other elements along the branch with which it is associated. This is governed by a 'base' generator, which can be any element-wise conditional process generator from Generator Matching, allowing drift, diffusion, and jumps, and conditioned to terminate at a specific value by $t = 1$. In Branching Flows, the anchor associated with each branch steers the base generator, which is conditioned to terminate, were it to reach $t = 1$, at the anchor a_i of the node ahead of it.

When an element is split it is replaced by two adjacent duplicates, and each duplicate is associated with a child branch. To control the rates of split events, let H_{split} be a 'split hazard' distribution supported on $[0, 1]$ with no atom at 1, with its associated hazard rate $h_{\text{split}}(t) = \frac{f_{H_{\text{split}}}(t)}{1-F_{H_{\text{split}}}(t)}$ where $f_{H_{\text{split}}}(t)$ is the density at t and $F_{H_{\text{split}}}(t)$ is the cumulative distribution. For an element associated with a specific branch, the tree (which is pre-sampled, within Z) determines the number of remaining split events $w_i - 1$ to be realized by $t = 1$. The instantaneous per-element split event rate is $(w_i - 1) \cdot h_{\text{split}}(t)$, and it holds that all remaining splits are consumed by $t = 1$ when $w_i > 1$.

For deletions, let H_{del} similarly define $h_{\text{del}}(t)$, and the deletion event rate is $h_{\text{del}}(t)$ on branches associated with deleted leaves, and 0 on branches associated with internal and surviving terminal nodes. Since the hazard rates are designed such that all splits and all deletions implied by $\mathcal{T}$ will be consumed by $t = 1$, and since the anchors a_i of all surviving terminal nodes are equal to the elements of x_1, we thus have a conditional path $x_t \sim p_{t|Z}(x_t \mid z)$ that begins at x_0 and terminates at x_1 with probability 1.

For each element to track which branch on which tree it belongs to (which is required for the conditional process to be Markov in the state), we augment the state space with a branch indicator g_t. With each element in $\tilde{\mathcal{E}}$, we have $\tilde{X}_t := (X_t, G_t) \sim p_{t|Z}(\mathrm{d}x, \mathrm{d}g|z)$ that begins at $\tilde{x}_0 := (x_0, g_0)$ and terminates at $\tilde{x}_1 := (x_1, g_1)$. See Appendix E for details of our branch tracking and split and deletion operators, and see figure 1 for a depiction of Z and the conditional path construction. Moreover, in Appendix G.3, we discuss how in a manner similar to Havasi et al. (2025) one can marginalize over G_t, treated as a latent discrete process. This reduces architectural complexity by not requiring the model to keep track of G_t.

B.4 Sampling from the Conditional Path

For training, we need samples at time t from the conditional path $\tilde{X}_t \sim p_{t|Z}$. With Branching Flows, even though the element evolution along a branch for the conditional paths is independent of the other elements given Z, because of the branching structure the distribution of the element *value* at any given time is not. This prevents us from adopting the usual strategy for flow matching which draws conditional samples for each element independently. However we can efficiently sample the waiting time until the next split or deletion event (see Corollary D.1.4 and Remark D.2.1), so if we additionally have an efficient per-element sampler for $p(\tilde{x}_v^{(k)} \mid \tilde{x}_s^{(k)}, a_k)$, where $s < v$ and a_k is the anchor associated with the node ahead of $\tilde{x}_s^{(k)}$, then we can sample from the conditional path at t by recursively sampling the waiting time until the next split or deletion event, and then sampling

the value of the element at the next split event (or at t if the next split event is after t). This is, like standard generator matching's full element-wise factorization, $\mathcal{O}(L_1^{+\emptyset})$.

For the base process in our empirical examples below, we use an Ornstein–Uhlenbeck (OU) like process with time-inhomogeneous diffusion, and with mean-reversion towards the anchors, and for discrete states we use an instance of Discrete Flow Matching (DFM) corresponding to equation 10 in Gat et al. (2024), which includes unmasking and uniform noise. See app. F for details.

B.5 Parameterization of the Generator and Loss

The Conditional Branching Flows (CBF) loss (cf. Appendix G.4), which uses the conditional process to train a model that samples from the marginal path, is the sum of three Bregman divergences: one for the base process, one against the split intensity, and one against the deletion intensity:

$$\begin{aligned} L_{\text{CBF}}(\theta) = &\mathbb{E}_{t\sim\mathcal{D}[0,1],Z\sim p_Z,(X_t,G_t)\sim p_{t|Z}(d\tilde{x}|z)} \\ &\Big[D^{\text{split}}_{t,X_t}(R_t^{Z,G_t}(X_t), R_t^{\theta}(X_t)) \\ &+ D^{\text{del}}_{t,X_t}(\rho_t^{Z,G_t}(X_t), \rho_t^{\theta}(X_t)) \\ &+ D^{\text{base}}_{t,X_t}\big(F_t^{Z,G_t,\text{base}}(X_t), F_t^{\theta,\text{base}}(X_t)\big)\Big]. \end{aligned}$$

Concretely, we use an elementwise loss where we parameterize the split and deletion rates in terms of the number of splits by $t = 1$ and the probability of deletion by $t = 1$ (i.e. "X_1-prediction"), denoting $R_{t,i}^{Z,G_t}(X_t)$ for the number of split increments ahead of the i'th component, and $\rho_{t,i}^{Z,G_t}(X_t) \in \{0,1\}$ for the i'th component of $\rho_t^{Z,G_t}(X_t)$. These are parametrized by $R_{t,i}^{\theta}(X_t) \in (0,\infty)$ and $\rho_{t,i}^{\theta}(X_t) \in (0,1)$ respectively. We write $F_{t,i}^{Z,G_t,\text{base}}(X_t)$ as the 'base' generator of a per-element conditional process along a single branch, steered to terminate (by $t = 1$) at the anchor $a(i, Z, G_t)$. This is the anchor at the end of the branch along which the i-th component of X_t is evolving (tracked by the branch indicator of the i-th component of the augmented state $\tilde{X}_t$). Our loss is then:

$$\begin{aligned} L_{\text{CBF}}(\theta) = &\mathbb{E}_{t\sim\mathcal{D}[0,1],Z\sim p_Z,(X_t,G_t)\sim p_{X_t,G_t|Z}(\mathrm{d}x,dg|z)} \\ &\Big[\sum_{i=1}^{L_t}\Big(R_{t,i}^{\theta}(X_t) - R_{t,i}^{Z,G_t}(X_t)\log R_{t,i}^{\theta}(X_t)\Big) \\ &+ \sum_{i=1}^{L_t}\Big(-[\rho_{t,i}^{Z,G_t}(X_t)\log\rho_{t,i}^{\theta}(X_t) \\ &+ (1-\rho_{t,i}^{Z,G_t}(X_t))\log(1-\rho_{t,i}^{\theta}(X_t))]\Big) \\ &+ \sum_{i=1}^{L_t}\Big(D^{\text{base}}_{t,X_t,i}(F_{t,i}^{Z,G_t,\text{base}}(X_t), F_{t,i}^{\theta,\text{base}}(X_t))\Big)\Big]. \end{aligned} \tag{1}$$

Note that only X_t (and not G_t) is input to the model, as G_t, like Z, is marginalized out in the training process per app. G.3.

B.6 Training and Sampling

Training proceeds exactly as is standard in flow matching, by stochastic gradient descent. We draw $t \sim \mathcal{D}$, $X_1 \sim q$, $Z \sim p_{Z|X_1}$, and then draw $\tilde{X}_t = (X_t, G_t) \sim p_{t|Z}(d\tilde{x}_t|z)$. From Z, X_t, and G_t, we recover $R_{t,i}^{Z,G_t}(X_t)$, $\rho_{t,i}^{Z,G_t}(X_t)$, and $F_{t,i}^{Z,G_t,\text{base}}(X_t)$, and take gradient steps on the loss in equation 1.

We sample from the marginal paths by taking small steps in time. A trained model outputs element-wise predictions $R_{t,i}^{\theta}(X_t)$, $\rho_{t,i}^{\theta}(X_t)$, and $F_{t,i}^{\theta,\text{base}}(X_t)$. For a step size Δ_t, we first take a step on the base process according to the model's learned expected generator, independently per element, just as in standard Generator Matching. For the branching and deletion part of each step, we calculate

per-element split intensities $h_{\text{split}}(t) \cdot R^{\theta}_{t,i}(X_t)$ and deletion intensities $h_{\text{del}}(t) \cdot \rho^{\theta}_{t,i}(X_t)$, and we sample which elements split or delete within the step. Elements with deletion events are removed, and elements with split events are duplicated in-place. The trees and branch indicators have been marginalized out and are not considered when sampling.

C Z SAMPLING SCHEMES

Here we outline concrete schemes for sampling $Z := (X_1^{+\emptyset}, X_0, \mathcal{T}, \mathcal{A})$, starting from X_1, that we use in our empirical examples below. Some datasets have elements that are naturally grouped (e.g. proteins, grouped by chain), and one can construct Z to respect these groupings, where it makes sense to do so, for example in the protein model described in the main text. In the case where there is only one group, all elements have the same group index. Further, if any elements are 'fixed' (e.g. in the protein example below, by a conditioning mask), then any elements with a fixed element between them in the sequence must belong to a different group.

C.1 INITIAL DISTRIBUTION: $X_0 \sim \mathcal{D}_0|(X_1 = x_1)$

To sample X_0 we need to sample the number of elements, and the state of each element. While in general the state of X_0 elements can be coupled to X_1, here we only concretely explore schemes where each X_0 element is independently sampled from a base distribution, which is a 'masked token' for all discrete state components, a standard isotropic Gaussian for all continuous components, and uniform samples on SO(3) for rotations.

To sample the X_0 length, we use:

$$\ell_0^{\text{group}} \sim 1 + \text{Poisson}(\lambda_{\text{group}}),$$

where in many cases λ_{group} is 0, meaning X_0 starts with a single element (or one per group).

C.2 DELETIONS: $X_1^{+\emptyset} \sim \mathcal{D}_\emptyset|(X_1 = x_1, X_0 = x_0)$

Sampling $X_1^{+\emptyset} \sim \mathcal{D}_\emptyset|(X_1 = x_1, X_0 = x_0)$ includes how many 'to-be-deleted' elements are introduced, where they are introduced in the sequence, and what their state is, all which can be flexibly chosen. In what we think will provide a better learning target, we consider schemes where elements of X_1 are duplicated, and their state copied. In some cases it also might sense to modify the state, such as enforcing that 'to-be-deleted' discrete states are always masked tokens.

Importantly, deletions are chosen such that $\ell_1^{\text{group}} \geq \ell_0^{\text{group}}$ for all groups. To ensure this, one scheme we use is to specify a deletion rate $d_r \geq 1$, and draw the number of to-be-deleted elements with

$$\text{del}_{\text{group}} = \text{Poisson}(\max(\ell_0^{\text{group}}, \ell_1^{\text{group}})d_r - \ell_1^{\text{group}})$$

$\text{del}_{\text{group}}$ elements are then sampled, uniformly with replacement, from the elements from the matching group of X_1, and inserted, uniformly, either side of X_1 template element, inheriting its state.

Another scheme we use, only when $\ell_0^{\text{group}} = 1$, is to take each element in X_1 and duplicate it, inserting that randomly on either side.

C.3 FOREST: $\mathcal{T} \sim \mathcal{D}_{\mathcal{T}}|(X_1^{+\emptyset} = x_1^{+\emptyset}, X_0 = x_0)$

We need ℓ_0^{group} trees in $\mathcal{T}$ (i.e. one tree per X_0 element). Since the trees must be planar, we adopt a backward coalescence scheme that starts with the elements of $X_1^{+\emptyset}$ and uniformly selects adjacent pairs that belong to the same group, coalescing them and replacing the pair with a single node. This proceeds until there are only ℓ_0 remaining elements, which become the root of each tree that is associated, maintaining the order, with the elements of X_0.

C.4 INTERNAL ANCHORS: $\mathcal{A}^{\mathcal{I}} \sim \mathcal{D}|(X_1^{+\emptyset} = x_1^{+\emptyset}, X_0 = x_0, \mathcal{T})$

Anchors control the conditional process along branches, guiding each element so that it would terminate, were t to reach 1, at the anchor. Along terminal branches, the anchors are equal to the elements of $X_1^{+\emptyset}$, but along internal branches we have freedom in how we construct them.

Consider a path along one tree from the root to a single leaf. An element drifts, along this path, towards one anchor after another, changing each time a split event occurs. If the anchors change too erratically, the process will be difficult to learn. One principle is to sample anchors that avoid, as much as possible, high-velocity changes.

Since anchors can be sampled conditioned on $X_1^{+\emptyset}$, one convenient approach is to specify an anchor merging distribution $p(a_{\text{parent}}|a_{\text{child}_1}, a_{\text{child}_2}, \mathcal{T})$ and recursively determine the anchors from leaves to root. This is depicted as $m(\cdot,\cdot)$ in figure 1. Since ordered sequences of continuous values (especially for spatial positions) often exhibit strong autocorrelation with similar values for neighboring elements but discrete values typically do not, the anchor merging schemes can differ depending on whether the state components are continuous or discrete. For our empirical investigations, we use:

- Continuous ($\mathbb{R}^n$): mean of a_{child_1} and a_{child_2}, weighted by the number of descendants of each child.
- Manifold: Geodesic interpolation between a_{child_1} and a_{child_2}, weighted by the number of descendants of each child.
- Discrete: All internal anchors are set as a mask/dummy token.

An alternative choice which we have explored, but not used in our examples below, is copying a random child (with probability optionally weighted by some function of the number of descendants of each child).

D Counting and Deletion Process Flow Matching

We first describe Counting Flows and Deletion Flows, which we will use to construct Branching Flows.

D.1 Counting Flows

D.1.1 Conditional distributions

We first introduce 'Counting Flows': flow matching over a Poisson counting process that generates samples from the space of non-negative integer 'counts' via a conditional process that repeatedly increments by 1 to terminate at an observed count z_c by $t = 1$. This will later be repurposed for Branching Flows to model the counts of "splitting events" ahead of a node at time t, but we present it here in isolation.

Let H_{inc} be a hazard distribution with no atom at 1 supported on $[0, 1]$, implying that $S_{H_{\text{inc}}}(1) = 1 - F_{H_{\text{inc}}}(1) = 0$. Since $S_{H_{\text{inc}}}(t) = 1 - F_{H_{\text{inc}}}(t)$ is the probability of the event not occurring by time t, this implies the event occurs almost surely by $t = 1$. The hazard rate associated to H_{inc} is given by

$$h_{\text{inc}}(t) = \frac{f_{H_{\text{inc}}}(t)}{1 - F_{H_{\text{inc}}}(t)}.$$

We also note that since

$$\int_0^t h(s)ds = -\log(1 - F_{H_{\text{inc}}}(t)),$$

a hazard $h(t) \geq 0$ gives rise to a hazard distribution supported on $[0, 1]$ with no atom at 1 as long as

$$\int_0^1 h(t)dt = \infty.$$

Conditioned on an observed count z_c we construct a time-inhomogeneous CTMC on $\{0, 1, \ldots, z_c\}$ with rates

$$Q_t^{z_c}(x_t + 1|x_t) = (z_c - x_t)h_{\text{inc}}(t)I_{\{x_t < z_c\}},$$

and $Q_t^{z_c}(x_t|x_t) = -Q_t^{z_c}(x_t + 1|x_t)$, with $Q_t^{z_c}(\cdot|x_t) = 0$ otherwise.

For any t, the increment rate is thus proportional to the number of increments remaining, and the process will almost surely terminate at z_c.

With $R_t^{z_c} = z_c - x_t$ the remaining increments, and R_t^θ a neural network's prediction of the remaining increments, then

$$D_R(R_t^{z_c}, R_t^\theta) = R_t^\theta - R_t^{z_c} \log R_t^\theta$$

is a valid loss under an appropriately scaled time-dependent linear parametrization (Billera et al., 2025).

It follows that sampling single-increment independent and identically distributed unordered event times from H_{inc} corresponds to sampling from the above CTMC, and for a given t, x_t, z_c we can efficiently sample the waiting time until the next increment event.

Lemma D.1.1. *Let $T_1, T_2, \ldots, T_n \sim H$ be independent samples from the hazard distribution H, with cumulative distribution function $F_H(t)$ and hazard $h(t)$. Define*

$$X_t := \sum_{i=1}^{n} I_{\{T_i \le t\}}.$$

Conditioned on $X_t = x_t$, there are exactly $n - x_t$ remaining events. Denote $R := n - x_t$. It holds that

$$\mathbb{P}[X_{t+\delta} = x_t + 1 | X_t = x_t] = Rh(t)\delta + o(\delta).$$

Proof. Let $T_t^+ := \min\{T_i : T_i > t\}$ denote the arrival time of the next jump at time t. Condition on the specific at risk index set $A = \{i : 1 \le i \le n,\ T_i > t\}$, corresponding to $|A| = R$ remaining events. Then for any $i \in A$ we have

$$\begin{aligned} p_\delta := \mathbb{P}(t < T_i \le t + \delta \mid T_i > t) &= 1 - \frac{S(t+\delta)}{S(t)} \\ &= 1 - \frac{S(t) + \delta S'(t) + o(\delta)}{S(t)} = h(t)\delta + o(\delta), \end{aligned}$$

noting that $S'(t) = -h(t)S(t)$. By independence, the number of jumps in $(t, t+\delta]$ among the R at-risk units is $\text{Binomial}(R, p_\delta)$, and thus

$$\mathbb{P}(X_{t+\delta} = x_t + 1 \mid X_t = x_t) = Rp_\delta(1 - p_\delta)^{R-1}.$$

Since $p_\delta = h(t)\delta + o(\delta) = O(\delta)$

$$(1 - p_\delta)^{R-1} = 1 - (R-1)p_\delta + O(p_\delta^2) = 1 + o(1),$$

it follows

$$\mathbb{P}(X_{t+\delta} = x_t + 1 \mid X_t = x_t) = R(h(t)\delta + o(\delta))(1 + o(1)) = Rh(t)\delta + o(\delta).$$

□

Corollary D.1.2. *Fix $z_c \in \mathbb{N}_{\ge 0}$ and define $X_t = \sum_{i=1}^{z_c} I_{\{T_i \le t\}}$. Then it holds that X_t evolves as a CTMC with rate matrix*

$$Q_t^{z_c}(x_t + 1 | x_t) = (z_c - x_t)h(t)I_{\{x_t < z_c\}},$$

and $Q_t^{z_c}(x_t|x_t) = -Q_t^{z_c}(x_t+1|x_t)$, with $Q_t^{z_c}(\cdot|x_t) = 0$ otherwise. In other words, if the event times are distributed according to H, then counting the events that have occurred up until the current time out of z_c total events corresponds to the counting flow process conditioned on z_c, starting at zero.

D.1.2 Interarrival Times

Theorem D.1.3. *Fix $n \ge 1$, and let $T_1, \ldots, T_n \sim H$ where H is a hazard distribution having cumulative distribution function $F_H(t)$ and survival function $S_H(t) = 1 - F_H(t)$. Set*

$$X_t = \sum_{i=1}^{n} I_{\{T_i \le t\}}$$

and let $T_t^+ := \min\{T_i : T_i > t\}$. Let $V_t = (T_t^+ | X_t = x_t)$, and define $W_t = V_t - t$ as the conditional inter-arrival time for the next arrival, conditioned on x_t preceding arrivals.

Denote $R := n - x_t$ for the remaining number of events, given that x_t events already happened and that there are n events total. Then for each t, the quantile function of V_t is given by

$$F_{V_t}^{-1}(u) = F_H^{-1}(1 - S_H(t)(1-u)^{\frac{1}{R}}),$$

where $F_H^{-1}(u) := \inf\{y : F_H(y) \geq u\}$ is the quantile function for H. In particular, if $U \sim \mathrm{Unif}(0,1)$, then

$$W_t \stackrel{d}{=} F_H^{-1}\left(1 - S_H(t)(1-U)^{\frac{1}{R}}\right) - t.$$

Proof. Let $T_1, \ldots, T_R \sim H$ i.i.d. denote the $R := n - x_t$ arrival times exceeding t and fix $t \in (0,1)$. For $y \geq t$, we have

$$G_t(y) := \mathbb{P}(T_i \leq y \mid T_i > t) = \frac{F_H(y) - F_H(t)}{1 - F_H(t)} = \frac{S_H(t) - S_H(y)}{S_H(t)}.$$

By independence,

$$\mathbb{P}(V_t \geq y) = \mathbb{P}(T_i \geq y \mid T_i > t)^R = \left(\frac{S_H(y)}{S_H(t)}\right)^R.$$

The quantile function is $F_{V_t}^{-1}(u) = \inf\{y : F_{V_t}(y) \geq u\}$. It holds that

$$F_{V_t}(y) = u \iff 1 - \left(\frac{S_H(y)}{S_H(t)}\right)^R = u \iff F_H(y) = 1 - S_H(t)(1-u)^{\frac{1}{R}},$$

which allows us to conclude

$$F_{V_t}^{-1}(u) = F_H^{-1}(1 - S_H(t)(1-u)^{\frac{1}{R}}).$$

Let $U \sim \mathrm{Unif}(0,1)$. It is well known that passing U to the quantile function recovers the original distribution:

$$V_t \stackrel{d}{=} F_{V_t}^{-1}(U)$$

so since $W_t = V_t - t$, we have

$$W_t \stackrel{d}{=} F_H^{-1}\left(1 - S_H(t)(1-U)^{\frac{1}{R}}\right) - t.$$

□

Corollary D.1.4. *Fix $z_c \geq 1$ and let X_t be a counting process with hazard distribution H. Condition on $X_t = x_t$, where $x_t < z_c$. Then there are $z_c - x_t$ remaining events and the waiting time until the next event has distribution*

$$W_t \stackrel{d}{=} F_H^{-1}\left(1 - S_H(t)(1-U)^{\frac{1}{z_c - x_t}}\right)$$

where $U \sim \mathrm{Unif}(0,1)$. This allows us to efficiently sample the next waiting time given t, x_t and z_c.

D.1.3 Marginal Generator

Let $Z_c \sim p_{Z_c}(dz_c)$ and $Q_t^{z_c}$ specify a conditional on $Z_c = z_c$ rate matrix of the counting flow process,

$$Q_t^{z_c}(x_t + 1 | x_t) = h(t)(z_c - x_t) I_{\{x_t < z_c\}},$$

and $Q_t^{z_c}(x_t | x_t) = -Q_t^{z_c}(x_t + 1 | x_t)$, with $Q_t^{z_c}(\cdot | x_t) = 0$ otherwise. We will recast this in terms of a time-dependent jump kernel (see Appendix G.1). This specifies a conditional process $X_t | Z_c = z_c$.

The time-dependent jump kernel characterizing the counting process is

$$Q_t^{z_c}(dy; x_t) = h(t)(z_c - x_t)\delta_{x_t+1}(dy),$$

corresponding to jumping from x_t to $x_t + 1$ with rate $h(t)(z_c - x_t)$. It holds that all remaining events will occur by $t = 1$ almost surely. Using the notion of a time-dependent linear parametrization

(Billera et al., 2025), we can parametrize the infinitesimal generator corresponding to the jump process above by $R_t^{z_c}(x) = z_c - x$, and have a valid conditional generator matching loss be given by

$$L_{\text{cgm}}(\theta) = \mathbb{E}_{Z_c \sim p_{Z_c}, X_t \sim p_{t|Z_c}(dx|z_c)}[D(R_t^{Z_c}(X_t), R_t^\theta(X_t))],$$

where $R_t^\theta(x_t)$ is emitted by the model for each realization x_t of X_t, and D is a Bregman divergence. Note that the loss is minimized by the parameterization of the marginal generator, which coincides by linearity with the posterior averaged parameterization of the conditional generator

$$R_t(x_t) = \mathbb{E}_{Z_c \sim p_{z_c(dz_c|x_t)}}[R_t^{Z_c}(x_t)] = \mu_{z_c|x_t} - x_t,$$

setting $\mu_{z_c|x_t} := \mathbb{E}_{Z_c \sim p_{z_c(dz_c|x_t)}}[Z_c]$. The parameterization emitted by the model $F_t^\theta(x_t)$ corresponds to a time-dependent jump kernel

$$Q_t(dy|x_t) = h(t) R_t^\theta(x_t) \delta_{x_t+1}(dy).$$

The sampling procedure corresponding to the model-emitted jump kernel is

$$X_0 \sim p_0, \qquad (X_{t+\Delta t}|X_t = x_t) = \begin{cases} x_t & \text{with probability } 1 - \Delta t \cdot h(t) R_t^\theta(x_t), \\ x_t + 1 & \text{with probability } \Delta t \cdot h(t) R_t^\theta(x_t), \end{cases}$$

and when $F_t^\theta = F_t$ we have $X_t \sim p_t(dx)$ as $\Delta t \to 0$.

D.2 Deletion Flows

We also describe a CTMC in the state space $S = \{0, 1\}$, where 0 is 'present' and 1 is 'deleted'. Let H_{del} be a hazard distribution supported on $[0, 1]$ with no atom at 1, with its associated hazard rate

$$h_{\text{del}}(t) = \frac{f_{H_{\text{del}}}(t)}{1 - F_{H_{\text{del}}}(t)}$$

We represent the conditional probability of a deletion by $t = 1$ with $Z_d \in \{0, 1\}$. This determines the conditional rate $Q_t^{Z_d}(x|x_t)$, defined by:

$$Q_t^{Z_d}(1|0) = Z_d h_{\text{del}}(t) \quad \text{and} \quad Q_t^{Z_d}(0|0) = -Q_t^{Z_d}(1|0),$$

along with $Q_t^{Z_d}(\cdot|x_t) = 0$ for all other values of x_t. Conditional on $Z_d = 1$, corresponding to a deleted element, the process $X_t|(Z_d = 1)$ will terminate in deletion by $t = 1$ with probability 1. Let $\rho_t^{z_d} = z_d$ and ρ_t^θ be a neural network's prediction of the probability of deletion by time t. Then

$$D_\rho(\rho_t^{z_d}, \rho_t^\theta) = -[z_d \log(\rho_t^\theta) + (1 - z_d) \log(1 - \rho_t^\theta)]$$

is a valid, per-term, conditional generator matching loss under an appropriately scaled time-dependent linear parameterization (Billera et al., 2025).

Remark D.2.1. *The generator and the interarrival times obtained from having one or zero remaining events for counting flows will coincide for deletion flows.*

D.2.1 Branching Flows

We construct Branching Flows with split and deletion events. The hazard rate of split events is defined identically to that of the increments in Counting Flows. For splits, the waiting time until the next event will coincide with that in Theorem D.1.3, where the number of remaining split events ahead of each node corresponds to the number of remaining increments in Counting Flows. Deletions are then exactly as in Deletion Flows.

E Branch tracking, splits, and deletions

We identify a branch with the node 'ahead' of it (in the root-to-leaf, $t = 0$ to $t = 1$ direction), and $g_t^{(i)} = (\tau_t^{(i)}, b_t^{(i)})$ stores a tree index $\tau_t^{(i)}$, and a branch indicator vector $b_t^{(i)}$ comprising a possibly-empty ordered sequence of child indices $\uparrow$ (first child) or $\downarrow$ (second child) which, since the tree is

binary, read from root towards leaves identifies a single branch on the tree by identifying which of the two children are descended at each bifurcation. An element of the augmented state reads $\tilde{x}_t^{(i)} = (x_t^{(i)}, g_t^{(i)}) = (x_t^{(i)}, (\tau_t^{(i)}, b_t^{(i)}))$. With $<>$ the empty branch indicator tuple (using non-standard tuple brackets to make the tuple nesting easier to read), $(\tau_t^{(i)} = k, b_t^{(i)} =<>)$ denotes the root of the kth tree, and $(\tau_t^{(i)} = k, b_t^{(i)} =<\uparrow\uparrow\downarrow>)$ would denote the second child of the first child of the first child of the root of the kth tree.

Let $\tilde{x}_t^{(i)} = (x_t^{(i)}, (\tau_t^{(i)}, b_t^{(i)}))$. We use $\triangleright$ to denote the append operation

$$\tilde{x}_t^{(i)} \triangleright \uparrow := (x_t^{(i)}, (\tau_t^{(i)}, b_t^{(i)} \triangleright \uparrow)),$$

and analogously define $\tilde{x}_t^{(i)} \triangleright \downarrow$.

For a state $\tilde{x}_t = (\tilde{x}_t^{(1)}, \ldots, \tilde{x}_t^{(n)})$, we define a 'split operator':

$$\text{split}_i(\tilde{x}_t) := (\tilde{x}_t^{(1)}, \ldots, (\tilde{x}_t^{(i)} \triangleright \uparrow),\ (\tilde{x}_t^{(i)} \triangleright \downarrow), \ldots, \tilde{x}_t^{(n)})).$$

This duplicates, in place, the element $\tilde{x}_t^{(i)}$, sending a copy down each of the two child branches. We also define a 'deletion' operator:

$$\text{del}_i(\tilde{x}_t) := (\tilde{x}_t^{(1)}, \ldots, \tilde{x}_t^{(i-1)}, \tilde{x}_t^{(i+1)}, \ldots, \tilde{x}_t^{(n)}).$$

that deletes the element $\tilde{x}^{(i)}$ at index i.

F BASE PROCESSES

F.1 THE ORNSTEIN-UHLENBECK PROCESS WITH TIME DEPENDENT DIFFUSION COEFFICIENT

Consider the stochastic differential equation

$$dX_t = \theta(\mu - X_t)dt + \sqrt{v_t}\, dW_t, \tag{2}$$

where $\theta > 0$ and v_t is the time-dependent infinitesimal variance (so the diffusion coefficient is $\sqrt{v_t}$). We will, for some x_1, consider the time-inhomogeneous OU-bridge $X_t \mid X_1 = x_1$ where the unconditional process X_t solves

$$dX_t = \theta(x_1 - X_t)dt + \sqrt{v_t}dW_t. \tag{3}$$

A slightly more general problem is studied in Albano & Giorno (2020), specifically of the form (See their equation (2.4))

$$dX_t = \left(-\frac{X_t}{\nu} + \mu + m_t\right) dt + \sqrt{v_t}dW_t, \tag{4}$$

where explicit formulas for the transition densities (which remain Gaussian) are given, which we use to derive bridge densities.

F.2 CONDITIONAL AND MARGINAL PATHS FOR A NOISY DISCRETE PROCESS

In what follows, we use an extension of Discrete Flow Matching (DFM Gat et al. (2024)) to allow time intervals $[t_0, t_1]$. Let F_1 and F_2 be CDFs of some continuous distributions supported on $[0, 1]$, and let $\omega_u \in [0, 1)$. Then we define the schedulers

$$\kappa_1(t) = F_1(t), \qquad \kappa_2(t) = \omega_u(1 - F_1(t))F_2(t), \qquad \kappa_3(t) = 1 - \kappa_1(t) - \kappa_2(t).$$

It can be shown that our choice of schedulers enforces $\ell_\kappa := \text{argmin}_{j\in[m]} \frac{\dot{\kappa}_t^j}{\kappa_t^j} = 3$. With these schedulers, define the interpolant path

$$p_t(x|x_{t_0}, x_1) := (\kappa_t^1 - \kappa_{t_0}^1 \frac{\kappa_t^3}{\kappa_{t_0}^3})\delta_{x_1}(x) + (\kappa_t^2 - \kappa_{t_0}^2 \frac{\kappa_t^3}{\kappa_{t_0}^3})p_u(x) + \frac{\kappa_t^3}{\kappa_{t_0}^3}\delta_{x_{t_0}}(x)$$

between x_{t_0} and x_1 in the time-interval $[t_0, 1]$. This path is generated by

$$u_t(x, z|x_{t_0}, x_{t_1}) = (\dot{\kappa}_t^1 - \kappa_t^1 \frac{\dot{\kappa}_t^3}{\kappa_t^3})\delta_{x_{t_1}}(x) + (\dot{\kappa}_t^2 - \kappa_t^2 \frac{\dot{\kappa}_t^3}{\kappa_t^3})p_u(x) + \frac{\dot{\kappa}_t^3}{\kappa_t^3}\delta_z(x)$$

which follows from applying DFM Theorem 3 on the interval $[t_0, 1]$. To see this, define $\alpha_1 = \kappa_t^1 - \kappa_{t_0}^1 \frac{\kappa_t^3}{\kappa_{t_0}^3}$, $\alpha_2 = \kappa_t^2 - \kappa_{t_0}^2 \frac{\kappa_t^3}{\kappa_{t_0}^3}$ and $\alpha_3 = \frac{\kappa_t^3}{\kappa_{t_0}^3}$. It can be seen that $\ell_\alpha := \text{argmin}_{j\in[m]} \frac{\dot{\alpha}_t^j}{\alpha_t^j} = 3$, along with $\alpha_1, \alpha_2, \alpha_3 \geq 0$ and $\sum_{i=1}^3 \alpha_i = 1$. The expression for the rates follows from

$$\begin{aligned}\dot{\alpha}_1 - \alpha_1 \frac{\dot{\alpha}_3}{\alpha_3} &= (\dot{\kappa}_t^1 - \kappa_{t_0}^1 \frac{\dot{\kappa}_t^3}{\kappa_{t_0}^3}) - (\kappa_t^1 - \kappa_{t_0}^1 \frac{\kappa_t^3}{\kappa_{t_0}^3})\frac{\dot{\kappa}_t^3}{\kappa_t^3} \\ &= \dot{\kappa}_t^1 - \kappa_t^1 \frac{\dot{\kappa}_t^3}{\kappa_t^3}\end{aligned}$$

and analogously

$$\dot{\alpha}_2 - \alpha_2 \frac{\dot{\alpha}_3}{\alpha_3} = \dot{\kappa}_t^2 - \kappa_t^2 \frac{\dot{\kappa}_t^3}{\kappa_t^3}.$$

G Branching Flows in Generator Matching

G.1 The Infinitesimal Generator of Jump Processes

In line with Generator Matching (GM) (Holderrieth et al., 2024) and Flow Matching Guide (FMG) (Lipman et al., 2024), we model jumps by a time-dependent kernel $Q_t(dy; x)$ that, for each state $x \in S$, assigns a positive measure on $S \setminus \{x\}$. Its total mass

$$\lambda_t(x) = \int Q_t(dy; x)$$

gives the jump intensity or jump rate, i.e. the instantaneous hazard of leaving x, and we require $\lambda_t(x) < \infty$. When $\lambda_t(x) > 0$, normalizing Q_t yields the jump destination distribution

$$J_t(dy; x) = \frac{Q_t(dy; x)}{\lambda_t(x)},$$

which is a probability measure on $S \setminus \{x\}$. Now we see how specifying a jump kernel $Q_t(dy; x)$ describes a jump process. With the hazard/rate of $\lambda_t(x)$, we will jump to a point $y \sim J_t(dy; x)$.

For $\lambda_t(x_t) = 0$ the process is first-order constant, in the sense that the probability that a jump occurs in $[t, t + \Delta t]$ is $o(\Delta t)$. Otherwise, the sampling procedure is as follows:

$$(X_{t+\Delta t}|X_t = x_t) = \begin{cases} X_t & \text{with probability } 1 - \Delta t \lambda_t(x_t) + o(\Delta t), \\ \sim J_t(dy; x_t) & \text{with probability } \Delta t \lambda_t(x_t) + o(\Delta t). \end{cases}.$$

The infinitesimal generator $\mathcal{L}_t$ associated to a process with jump measure Q_t becomes

$$\mathcal{L}_t f(x) = \int (f(y) - f(x))Q_t(dy; x)$$

As in (Billera et al., 2025) we consider atomic time-dependent jump kernels:

$$Q_t(dy; x) = \sum_{i=1}^n \lambda_i(t, x)\delta_{\Gamma_i(t,x)}(dy),$$

specifying *jump targets* $\Gamma_i(t, x)$ and *jump rates* $\lambda_i(t, x)$. The total rate is

$$\lambda_{\text{total}}(t, x) := \int Q_t(dy; x) = \sum_{i=1}^n \lambda_i(t, x)$$

and the normalized jump distribution $J_t(dy; x)$ is

$$J_t(dy; x) = \frac{1}{\lambda_{\text{total}}(t, x)} \sum_{i=1}^n \lambda_i(t, x)\delta_{\Gamma_i(t,x)}(dy),$$

so that if $Y \sim J_t(dy|x)$, then $\mathbb{P}(Y = \Gamma_i(t, x)) = \frac{\lambda_i(t,x)}{\lambda_{\text{total}}(t,x)}$ for $i = 1, \ldots, n$.

G.2 THE MARGINAL BRANCHING INFINITESIMAL GENERATOR

Let $S := \bigsqcup_{n\geq 1} \mathcal{E}^n$. We say that $x \in \mathcal{E}^n$ has n elements. We let $h_{\text{split}}(t)$ and $h_{\text{del}}(t)$ be overall hazard multipliers as in Appendix D.1, corresponding to an overall rate multiplier for split increments and an overall rate multiplier for deletions. Suppose that splitting of the i'th element out of the n elements occurs with rate $h_{\text{split}}(t) \cdot R_{t,i}(x)$ and that deletion of the i'th element out of the n elements occurs with rate $h_{\text{del}}(t) \cdot \rho_{t,i}(x)$. Here, $R_{t,i}(x) \geq 0$ is intended to indicate the 'remaining number of splits of the i'th element' and $\rho_{t,i}(x) \in [0,1]$ is intended to indicate the 'probability of deleting the i'th element'. For $n = 1$, we require that the deletion probability is zero so as not to result in an empty sequence.

In this section, we arrive at a time- and state varying linear parametrization for the infinitesimal generator of our branching process. The notion of a linear parametrization of a generator, introduced in Holderrieth et al. (2024) and further discussed in Lipman et al. (2024), was clarified to be able to vary with both time and state in Billera et al. (2025).

For the subsequent discussion, we consider times t such that $h_{\text{split}}(t) > 0$ and $h_{\text{del}}(t) > 0$. This is justified because if the hazard rate for split events $h_{\text{split}}(t)$ is zero, there is nothing for the model to learn since in that case it is known *a priori* that the split rate $h_{\text{split}}(t) \cdot R_{t,i}(x)$ vanishes. Similarly, if the hazard rate for deletion events $h_{\text{del}}(t)$ is zero, the deletion rate $h_{\text{del}}(t) \cdot \rho_{t,i}(x) = 0$ is also known *a priori*.

In what follows, we consider $x \in \mathcal{E}^n$ having $n \geq 1$ elements. A time- and state varying linear parametrization of the branching generator is

$$\mathcal{L}_t^{\text{branch}} f(x) = \langle \mathcal{K}_{t,n}^{\text{split}} f(x), R_t(x)\rangle_{t,n}^{\text{split}} + \langle \mathcal{K}_{t,n}^{\text{del}} f(x), \rho_t(x)\rangle_{t,n}^{\text{del}} + \langle \mathcal{K}_{t,n}^{\text{base}} f(x), F_t^{\text{base}}(x)\rangle_{t,x}^{\text{base}}.$$

In the above, using an atomic time-dependent jump kernel as in (Billera et al., 2025), splits and deletions correspond to the time-dependent jump kernel

$$Q_t(dy; x) = \sum_{i=1}^{n} h_{\text{split}}(t) R_{t,i}(x) \delta_{\text{split}_i(x)}(dy) + \sum_{i=1}^{n} h_{\text{del}}(t) \rho_{t,i}(x) \delta_{\text{del}_i(x)}(dy).$$

With this time-dependent jump kernel, the infinitesimal generator for jumps becomes

$$\begin{aligned}
\mathcal{L}_t^{\text{jump}} f(x) &= \int_S (f(y) - f(x)) Q_t(dy; x) \\
&= \int_S \Bigg((f(y) - f(x)) \Bigg[h_{\text{split}}(t) \cdot \sum_{i=1}^{n} \Big(R_{t,i}(x) \delta_{\text{split}_i(x)}(dy) \Big) \\
&\qquad + h_{\text{del}}(t) \cdot \sum_{i=1}^{n} \Big(\rho_{t,i}(x) \delta_{\text{del}_i(x)}(dy) \Big) \Bigg] \Bigg) \\
&= h_{\text{split}}(t) \cdot \sum_{i=1}^{n} \Big((f(\text{split}_i(x)) - f(x)) R_{t,i}(x) \Big) \\
&\qquad + h_{\text{del}}(t) \cdot \sum_{i=1}^{n} \Big((f(\text{del}_i(x)) - f(x))\, \rho_{t,i}(x) \Big).
\end{aligned}$$

To linearly parametrize this generator, we define the inner products

$$\langle v, w\rangle_{t,n}^{\text{split}} := h_{\text{split}}(t) \langle v, w\rangle_{\mathbb{R}^n} \qquad \text{and} \qquad \langle v, w\rangle_{t,n}^{\text{del}} := h_{\text{del}}(t) \langle v, w\rangle_{\mathbb{R}^n}.$$

This allows us to linearly parametrize the jump generator with

$$\mathcal{L}_t^{\text{jump}} f(x) = \langle \mathcal{K}_{t,n}^{\text{split}} f(x), R_t(x)\rangle_{t,n}^{\text{split}} + \langle \mathcal{K}_{t,n}^{\text{del}} f(x), \rho_t(x)\rangle_{t,n}^{\text{del}},$$

where in the above, we set

$$\mathcal{K}_{t,n}^{\text{split}} f(x) = (f(\text{split}_i(x)) - f(x))_{i=1}^{n}$$

and

$$\mathcal{K}_{t,n}^{\text{del}} f(x) = (f(\text{del}_i(x)) - f(x))_{i=1}^{n}.$$

In between jumps, we assume that the process evolves according to a base generator on each branch level — this could, for example, correspond to an SDE. We assume that we have a linear parametrization

$$\mathcal{L}_t^{\text{base}} f(x) = \langle \mathcal{K}_{t,n}^{\text{base}} f(x), F_t^{\text{base}}(x) \rangle_{t,x}^{\text{base}}$$

for the base process along each branch. This results in

$$\begin{aligned}\mathcal{L}_t^{\text{branch}} f(x) &= \mathcal{L}_t^{\text{jump}} f(x) + \mathcal{L}_t^{\text{base}} f(x) \\ &= \langle \mathcal{K}_{t,n}^{\text{split}} f(x), R_t(x) \rangle_{t,n}^{\text{split}} + \langle \mathcal{K}_{t,n}^{\text{del}} f(x), \rho_t(x) \rangle_{t,n}^{\text{del}} + \langle \mathcal{K}_{t,n}^{\text{base}} f(x), F_t^{\text{base}}(x) \rangle_{t,x}^{\text{base}}.\end{aligned}$$

G.3 Conditioning on Latent Discrete Processes

In Branching Flows, the branching structure is encoded in the trees in Z, but the timing of the bifurcation events is stochastic. This means that, given the trees in Z and a state X_t, each element in X_t cannot be unambiguously assigned to a branch on a tree. We thus need to augment the state with a branch-tracking index G_t. While it is possible to additionally condition the model on the branch tracking index, this places an architectural burden on the model. To avoid this, we learn a generator on X that marginalizes over the time-dependent auxiliary process G_t, extending the framework of Edit Flows (Havasi et al., 2025) from the purely discrete X_t case to where X_t can be any process allowed under Generator Matching, including drift, diffusion, and jumps.

With joint conditional paths $(X_t, G_t)|(Z = z) \sim p_{t|Z}(\mathrm{d}x, \mathrm{d}g \mid z)$, we train a model that marginalizes over G_t and Z, learning a marginal generator $\mathcal{L}_t$ that generates a process $\widehat{X}_t$ with marginals $p_t(\mathrm{d}x)$ that generates samples from the data distribution $q(dx)$. In what follows, we will assume that the Generator Matching regularity conditions apply to the marginal process $\widehat{X}_t$ and the joint marginal (X_t, G_t) (marginalizing over Z), and the joint conditional $(X_t, G_t)|(Z = z)$.

Let $\mathcal{G}$ be discrete with $|\mathcal{G}| < \infty$ and let (X_t, G_t) be an $S \times \mathcal{G}$-valued process generated by $W_t : \mathcal{T}(S \times \mathcal{G}) \to C(S \times \mathcal{G}, \mathbb{R})$. For fixed $g \in \mathcal{G}$, define $\mathcal{L}_t^g f(x) := W_t \tilde{f}(x, g)$, where $\tilde{f}(x, g) = f(x)$.

In what follows, we define the measure $\mu_{X_t}^g(A) := p_{X_t,G_t}(A \times \{g\})$. Note that for each $E \subset \mathcal{G}$ we have $p_{X_t,G_t}(dx, E) = \sum_{g \in E} \mu_{X_t}^g(dx)$. The following lemma holds:

Lemma G.3.1.

$$\partial_t \sum_{g \in \mathcal{G}} \int_S f(x) \mu_{X_t}^g(dx) = \sum_{g \in \mathcal{G}} \int_S \mathcal{L}_t^g f(x) \mu_{X_t}^g(dx).$$

Proof. Write

$$\partial_t \sum_{g \in \mathcal{G}} \int_S f(x) \mu_{X_t}^g(dx) = \partial_t \int_S f(x) \sum_{g \in \mathcal{G}} \mu_{X_t}^g(dx)$$

where in the above we are integrating with respect to the measure $\sum_{g \in \mathcal{G}} \mu_{X_t}^g(dx)$. This becomes

$$\begin{aligned}&= \partial_t \int_S f(x) p_{X_t,G_t}(dx, \mathcal{G}) \\ &= \partial_t \int_S f(x) \int_{\mathcal{G}} p_{X_t,G_t}(dx, dg) \\ &= \partial_t \int_{S \times \mathcal{G}} f(x) p_{X_t,G_t}(dx, dg) \\ &= \partial_t \int_{S \times \mathcal{G}} \tilde{f}(x, g) p_{X_t,G_t}(dx, dg)\end{aligned}$$

and since W_t generates $p_{X_t,G_t}(dx, dg)$, by the Kolmogorov Forward Equation, it holds:

$$
\begin{aligned}
&= \int_{S\times\mathcal{G}} W_t \tilde{f}(x,g) p_{X_t,G_t}(dx,dg) \\
&= \sum_{g\in\mathcal{G}} \int_S W_t \tilde{f}(x,g) \mu^g_{X_t}(dx) \\
&= \sum_{g\in\mathcal{G}} \int_S \mathcal{L}^g_t f(x) \mu^g_{X_t}(dx).
\end{aligned}
$$

□

Theorem G.3.2. *It holds that*

$$\mathcal{L}_t f(x) := \sum_{g\in\mathcal{G}} \mathcal{L}^g_t f(x)\, p_{G_t|X_t}(g|x)$$

generates $p_t(dx)$, i.e., $\mathcal{L}_t$ solves the KFE for p_t:

$$\partial_t \int f(x) p_t(dx) = \int \mathcal{L}_t f(x) p_t(dx).$$

Moreover, assume that $F^g_t : S \to \Omega \subset V$ is a linear parametrization of $\mathcal{L}^g_t f(x)$ over a vector space V. If there is a linear $\mathcal{K} : \mathcal{T} \to \mathcal{C}(S; V)$ on the inner product space $\langle\cdot,\cdot\rangle_V$ such that

$$\mathcal{L}^g_t f(x) = \langle \mathcal{K} f(x), F^g_t(x)\rangle_V$$

for each $g \in \mathcal{G}$, then

$$F_t(x) = \mathbb{E}_{G_t\sim p_{G_t|X_t}(dg|x)}[F^{G_t}_t(x)]$$

linearly parametrizes $\mathcal{L}_t$.

Proof. For each $t \in [0,1]$, we have $p_t(dx) = \sum_{g\in\mathcal{G}} \mu^g_{X_t}(dx)$, and hence:

$$
\begin{aligned}
\partial_t \int_S f(x) p_t(dx) &= \partial_t \int_S f(x) \sum_{g\in\mathcal{G}} \mu^g_{X_t}(dx) \\
&= \partial_t \sum_{g\in\mathcal{G}} \int_S f(x) \mu^g_{X_t}(dx)
\end{aligned}
$$

and by Lemma G.3.1, it holds:

$$
\begin{aligned}
&= \sum_{g\in\mathcal{G}} \int_S \mathcal{L}^g_t f(x) \mu^g_{X_t}(dx) \\
&= \sum_{g\in\mathcal{G}} \int_S \mathcal{L}^g_t f(x) p_{G_t|X_t}(g|x) p_t(dx) \\
&= \int_S \underbrace{\left(\sum_{g\in\mathcal{G}} \mathcal{L}^g_t f(x) p_{G_t|X_t}(g|x) \right)}_{=\mathcal{L}_t f(x)} p_t(dx).
\end{aligned}
$$

Finally, by the linearity of expectations and the map $w \mapsto \langle v, w\rangle_V$ for fixed $v \in V$, we have

$$
\begin{aligned}
\mathcal{L}_t f(x) &= \mathbb{E}_{G_t\sim p_{G_t|X_t}(dg|x)}[\mathcal{L}^{G_t}_t f(x)] \\
&= \mathbb{E}_{G_t\sim p_{G_t|X_t}(dg|x)}[\langle \mathcal{K} f(x), F^{G_t}_t(x)\rangle_V] \\
&= \langle \mathcal{K} f(x), \mathbb{E}_{G_t\sim p_{G_t|X_t}(dg|x)}[F^{G_t}_t(x)]\rangle_V \\
&= \langle \mathcal{K} f(x), F_t(x)\rangle_V.
\end{aligned}
$$

□

As shown above, the process $\widehat{X}_t$ generated by $\mathcal{L}_t$ has marginals $p_t(dx)$, but we simply write our expectations over $X_t \sim p_t(dx)$ for clarity of notation.

Theorem G.3.3. *The auxiliary-process-conditioned generator matching loss and the generator matching loss coincide up to a constant in θ:*

$$\nabla_\theta \mathbb{E}_{t\sim\mathcal{D},X_t,G_t\sim p_{X_t,G_t}(dx,dg)} D(F_t^{G_t}(X_t), F_t^\theta(X_t)) = \nabla_\theta \mathbb{E}_{t\sim\mathcal{D},X_t\sim p_t(dx)}[D(F_t(X_t), F_t^\theta(X_t))].$$

Proof. Since expectations commute with the left slot of Bregman divergences under the gradient (cf. Lipman et al. (2024), Billera et al. (2025)), we obtain:

$$\begin{aligned}
&\nabla_\theta \mathbb{E}_{t\sim\mathcal{D},X_t,G_t\sim p_{X_t,G_t}(dx,dg)} D(F_t^{G_t}(X_t), F_t^\theta(X_t)) \\
&\qquad = \nabla_\theta \mathbb{E}_{t\sim\mathcal{D},X_t\sim p_t(dx)} \mathbb{E}_{G_t\sim p_{G_t|X_t}(dg|x)} D(F_t^{G_t}(X_t), F_t^\theta(X_t)) \\
&\qquad = \mathbb{E}_{t\sim\mathcal{D},X_t\sim p_t(dx)} \nabla_\theta \mathbb{E}_{G_t\sim p_{G_t|X_t}(dg|x)} D(F_t^{G_t}(X_t), F_t^\theta(X_t)) \\
&\qquad = \mathbb{E}_{t\sim\mathcal{D},X_t\sim p_t(dx)} \nabla_\theta D(\mathbb{E}_{G_t\sim p_{G_t|X_t}(dg|x)}[F_t^{G_t}(X_t)], F_t^\theta(X_t)) \\
&\qquad = \mathbb{E}_{t\sim\mathcal{D},X_t\sim p_t(dx)} \nabla_\theta D(F_t(X_t), F_t^\theta(X_t)) \\
&\qquad = \nabla_\theta \mathbb{E}_{t\sim\mathcal{D},X_t\sim p_t(dx)} D(F_t(X_t), F_t^\theta(X_t)).
\end{aligned}$$

□

Remark G.3.4. *The arguments in Holderrieth et al. (2024) allow us to include an additional latent conditioning variable $z \in \mathcal{Z}$ that is often used to steer the path towards terminating at a particular point of the data distribution. Let $\mathcal{L}_t^{z,g}$ denote the conditional generator given $z \in \mathcal{Z}$ and $g \in \mathcal{G}$. Define the z-conditioned generator*

$$\mathcal{L}_t^z f(x) := \mathbb{E}_{G_t\sim p_{G_t|X_t,Z}(dg|x,z)}[\mathcal{L}_t^{z,G_t} f(x)].$$

Then, by Theorem G.3.2 and Proposition 1 of GM,

$$\begin{aligned}
&\mathbb{E}_{Z\sim p_{Z|X_t}(dz|x),G_t\sim p_{G_t|Z,X_t}(dg|z,x)}[\mathcal{L}_t^{Z,G_t} f(x)] \\
&\qquad = \mathbb{E}_{Z\sim p_{Z|X_t}(dz|x)}[\mathbb{E}_{G_t\sim p_{G_t|Z,X_t}(dg|z,x)}[\mathcal{L}_t^{Z,G_t} f(x)]] \\
&\qquad = \mathbb{E}_{Z\sim p_{Z|X_t}(dz|x)}[\mathcal{L}_t^{Z} f(x)] \\
&\qquad = \mathcal{L}_t f(x),
\end{aligned}$$

so averaging over a fixed $z \in \mathcal{Z}$ and a discrete process $(G_t)_{0\le t\le 1}$ recovers the same marginal generator $\mathcal{L}_t$. For the loss, if we write $F_t^{z,g}$ for a linear parametrization of $\mathcal{L}_t^{z,g}$, the same linearity argument as in Theorem G.3.3 gives

$$\begin{aligned}
&\nabla_\theta \mathbb{E}_{t\sim\mathcal{D}[0,1],X_t,Z,G_t} D(F_t^{Z,G_t}(X_t), F_t^\theta(X_t)) \\
&\qquad = \mathbb{E}_{t\sim\mathcal{D},X_t}[\nabla_\theta \mathbb{E}_{Z\sim p_{Z|X_t}(dz|x),G_t\sim p_{G_t|Z,X_t}(dg|z,x)}[D(F_t^{Z,G_t}(X_t), F_t^\theta(X_t))]] \\
&\qquad = \mathbb{E}_{t\sim\mathcal{D},X_t}[\nabla_\theta D(\mathbb{E}_{Z\sim p_{Z|X_t}(dz|x),G_t\sim p_{G_t|Z,X_t}(dg|z,x)}[F_t^{Z,G_t}(X_t)], F_t^\theta(X_t))] \\
&\qquad = \mathbb{E}_{t\sim\mathcal{D},X_t} \nabla_\theta D(F_t(X_t), F_t^\theta(X_t)) \\
&\qquad = \nabla_\theta \mathbb{E}_{t\sim\mathcal{D},X_t} D(F_t(X_t), F_t^\theta(X_t)),
\end{aligned}$$

so conditioning on a fixed $z \in \mathcal{Z}$ does not change the gradient of the loss.

G.4 The Branching Flows Loss

The conditional branching flows loss is the auxiliary-process-conditioned generator matching loss associated to the linear parametrization from Appendix G.2, with conditional dynamics as described in Appendix G.3. For $x \in \bigsqcup_{n\ge 1} \mathcal{E}^n$, we specify a conditional infinitesimal generator of the form

$$\mathcal{L}_t^{z,g,\text{branch}} f(x) = \langle \mathcal{K}_{t,n}^{\text{split}} f(x), R_t^{z,g}(x)\rangle_{t,n}^{\text{split}} + \langle \mathcal{K}_{t,n}^{\text{del}} f(x), \rho_t^{z,g}(x)\rangle_{t,n}^{\text{del}} + \langle \mathcal{K}_{t,n}^{\text{base}} f(x), F_t^{z,g,\text{base}}(x)\rangle_{t,x}^{\text{base}},$$

and we similarly specify a model parametrized infinitesimal generator

$$\mathcal{L}_t^{\theta,\text{branch}} f(x) = \langle \mathcal{K}_{t,n}^{\text{split}} f(x), R_t^{\theta}(x)\rangle_{t,n}^{\text{split}} + \langle \mathcal{K}_{t,n}^{\text{del}} f(x), \rho_t^{\theta}(x)\rangle_{t,n}^{\text{del}} + \langle \mathcal{K}_{t,n}^{\text{base}} f(x), F_t^{\theta,\text{base}}(x)\rangle_{t,x}^{\text{base}}.$$

As in (Billera et al., 2025), when a generator is expressed as a sum of linear parametrizations, a valid per-term conditional generator matching loss is then the sum of Bregman divergences

$$D^{\text{split}}_{t,x}(R^{z,g}_t(x), R^{\theta}_t(x)) + D^{\text{del}}_{t,x}(\rho^{z,g}_t(x), \rho^{\theta}_t(x)) + D^{\text{base}}_{t,x}(F^{z,g,\text{base}}_t(x), F^{\theta,\text{base}}_t(x)),$$

again noting that x stands for $x \in S$. Assuming that $R^{z,g}_t(x)$ can be interpreted as the remaining number of splits ahead of each element and that $\rho^{z,g}_t(x)$ can be interpreted as deletion probabilities, we choose a Poisson-like Bregman for splits and a cross-entropy loss for deletions.

Note that these loss functions require a notion of Bregman divergence associated with convex functions that are only differentiable on a subset of their domain, as considered in (Billera et al., 2025)

If we also assume that the base Bregman divergence is separable and splits along each ahead, the conditional branching flows loss becomes:

$$\begin{aligned} L_{\text{CBF}}(\theta) = \mathbb{E}_{t\sim\mathcal{D}[0,1], Z\sim p_Z, (X_t,G_t)\sim p_{t|Z}(dx,dg|z)} & \\ \Big[\sum_{i=1}^{L_t} \Big(& R^{\theta}_{t,i}(X_t) - R^{Z,G_t}_{t,i}(X_t)\log R^{\theta}_{t,i}(X_t)\Big) \\ & + \sum_{i=1}^{L_t} \Big(-[\rho^{Z,G_t}_{t,i}(X_t)\log \rho^{\theta}_{t,i}(X_t) + (1-\rho^{Z,G_t}_{t,i}(X_t))\log(1-\rho^{\theta}_{t,i}(X_t))]\Big) \\ & + \sum_{i=1}^{L_t} \Big(D^{\text{base}}_{t,X_t,i}(F^{Z,G_t,\text{base}}_{t,i}(X_t), F^{\theta,\text{base}}_{t,i}(X_t))\Big)\Big]. \end{aligned}$$

H Empirical Details and Supplementary Results

H.1 QM9 analysis

H.1.1 Data

We use a QM9 version with 130,831 molecules (Ramakrishnan, 2024), where the atom order in the coordinate files is arranged to match the canonical SMILES string ordering, where hydrogens are (almost always) after the heavy atoms. We reasoned that the splitting mechanism of Branching Flows would not excel if there are spatially distant elements that are neighbors in the primary sequence, since a late split would then require very large flow velocities. We thus kept the primary heavy atom ordering (since neighboring SMILES atoms are often bonded, and thus spatially close), but redistributed the hydrogens such that each hydrogen is inserted into the sequence in front of its nearest heavy atom, ordered by distance when there are more than one hydrogen neighbour per heavy atom.

H.1.2 Branching Flows specification

For QM9, we constructed our Branching Flow with the following choices:

- Conditional base process:
  - Continuous: OU process flow (Appendix F.1), with a mean-reversion rate $\theta = 5$, and where the variance decays from 10 when $t = 0$ to 0.001 by $t = 1$.
  - Discrete: DFM convex interpolation (appendix F.2) of x_0, uniform noise, and x_1, with hazards $\mathcal{D}_1 = \text{Beta}(2, 2), \mathcal{D}_2 = \text{Beta}(2, 2), \omega_u = 0.2$.
- Branching hazard distribution: $\text{Beta}(1, 3/2)$
- Deletion hazard distribution: $U(0, 1)$
- To-be-deleted policy: Each element of x_1 gets duplicated and the to-be-deleted duplication is inserted (with equal probability) immediately before or after the original element.
- x_0 state distribution: $N(0, 1)$ for each continuous coordinate, and mask/dummy token for the discrete atom label.
- x_0 length distribution: 1, always.

H.1.3 Model and Training

We use a 12 layer transformer model (12 heads, each with head dimension of 64), with an embedding dimension of 384. The atom positions are encoded by Random Fourier Features (RFFs), and pairwise spatial features are also provided to each layer by a learnable, layer-specific, head-specific attention bias. Rotary Positional Encoding (RoPE, Su et al. (2023)) is used to encode the primary sequence ordering. The final six layers additively update the atom positions (and the spatial pair features are recalculated after each position update). The model outputs the endpoint-predicted (i.e. by $t = 1$) locations, logits over the endpoint-predicted discrete state, the log of the per-element expected number of future bifurcations by $t = 1$, and a deletion binary logit. The model was trained, using Muon (Jordan et al., 2024) with post-warmup learning rate of 0.005, for 500k batches (where each batch was 128 randomly-sampled QM9 molecules) and a linear learning-rate cooldown for the last 50k batches. For sampling, 10k samples from the Branching Flows QM9 model were generated with a time step schedule $t = 1 - (\cos(\pi s) + 1)/2$ for $s \in (0, 1/1000, 2/1000, \ldots, 1)$.

We study the QM9 Branching Flows model outputs by comparison to the QM9 data, comparing distributions of atom counts (total, carbon, nitrogen, oxygen, fluorine, and hydrogen), and molecular properties. Atom counts are read directly from the .xyz model output or QM9 data.

For molecule properties, since the training samples and model-generated samples are atom point clouds (ie. no bond structure is explicitly represented), we use OpenBabel to convert .xyz coordi-

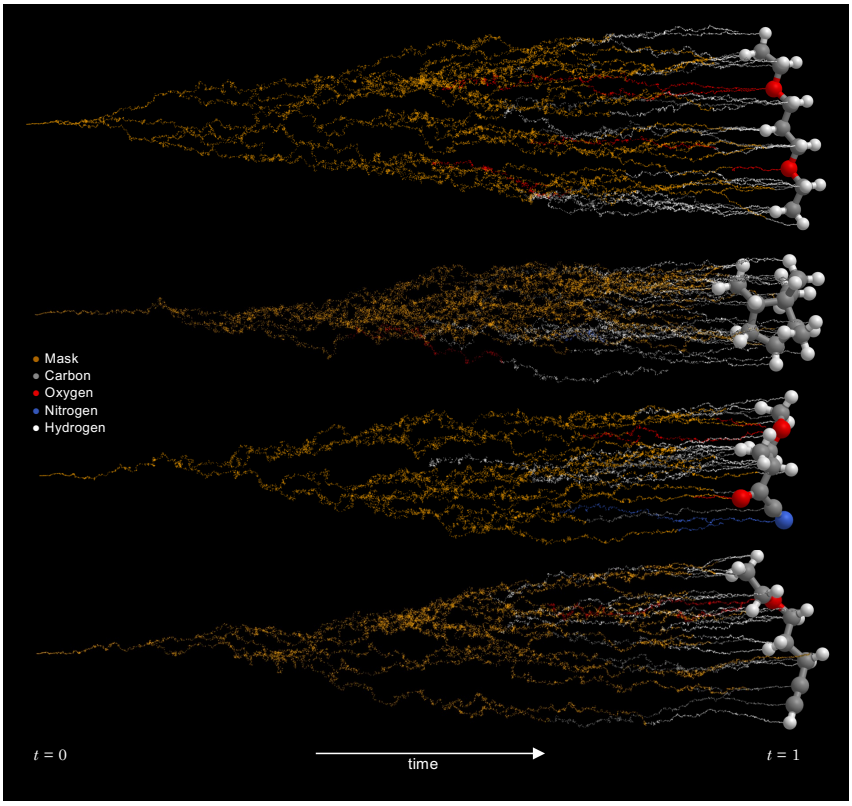

Figure 4: **QM9 Branching Flows sampling trajectories.** A visualization of inference trajectories from a QM9-trained Branching Flows model. Starting from a single x_0 element with a random position and a masked label, the process undergoes a sequence of bifurcations and deletions governed by the rates output from the model, with the discrete states switching along the trajectory as well, terminating at a generated molecule. The final sampled molecule, when $t = 1$, is depicted on the right. At every inference step from $t = 0$ to $t = 1$, we draw the current x_t state as colored points, with a small rightward displacement, which shows the branching and deletion history as x_t is transported from a single element when $t = 0$ to the final sampled molecule. The color of the trails shows the atom type which begins as 'Mask' (in orange) and switches to concrete atoms as $t \to 1$.

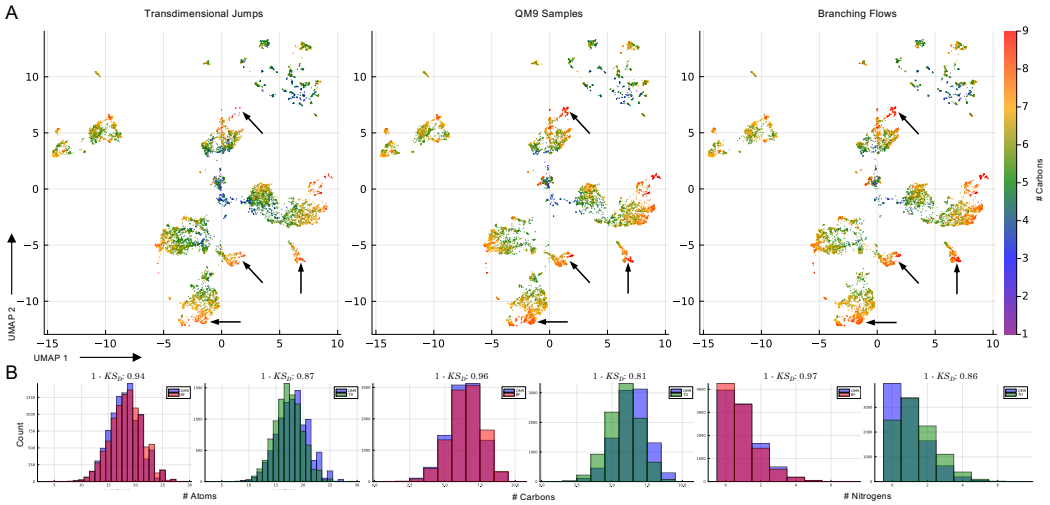

Figure 5: **Transdimensional Jump Diffusion and Branching Flows. A.** UMAP embeddings from molecular fingerprints for samples from a Transdimensional Jump Diffusion model (Campbell et al., 2023), from QM9 data, and from Branching Flows. Matched black arrows show regions there the data distribution has a high density of molecules with 8 or 9 carbons, which appears to be undersampled with Transdimensional Jumps. **B.** Histograms showing the distributions for Branching Flows (BF) vs QM9 and Transdimensional Jumps (TD) vs QM9, selected from the more extreme differences in Table 1.

nates to .sdf, which also infers bond structure and SMILES strings. The .sdf is then read into RDKit for downstream analysis, and computing molecule properties, and fingerprints. OpenBabel was run using the OpenBabel.jl Julia wrapper, and RDKit was run using the MoleculeFlow.jl Julia wrapper.

We discard any molecules that are invalid, where invalid includes molecules that RDKit returns a 'valid=false' flag, or where the molecule is not fully connected. Importantly, not all true QM9 molecules can be successfully parsed by RDKit from the .xyz files, and we apply identical filtering to both true and generated sets. After .xyz $\rightarrow$ .sdf conversion, 9882 of 10k QM9 molecules are 'valid=true' by RDKit and, of these, none were not fully connected. For Branching Flows generated molecules, 9871 of 10k are 'valid=true' and, of these, 58 were not fully connected. Thus 9882 for QM9 data and 9813 for Branching Flows generated molecules were used to compare distributions.

Molecular fingerprints encode the structural features of a molecule into a fixed-length bit vector which can be used for similarity search, clustering, etc. Here, we compute the 'RDKit fingerprints' (using MoleculeFlow.jl) and embed these in two dimensions, using Uniform Manifold Approximation (run via UMAP.jl, with default settings). This is used to visualize the distribution of true QM9 and generated samples.

Additionally, we used the model from Campbell et al. (2023) to generate 10,000 samples, and analyze them as above. Note: the UMAPs, which are computed jointly from all the samples, will differ in the analysis that includes transdimensional jump samples. This is compared to Branching Flows in table 1 and Figure 5.

To quantify distribution overlap, we use a Kolmogorov-Smirnov-like statistic between the empirical cumulative distribution functions of samples Y from each model against QM9 data. We use $1 - KS_D$, where

$$KS_D = \max_x |\hat{F}_Y(x) - \hat{F}_{QM9}(x)|$$

where $\hat{F}$ denotes the empirical CDF. This statistic is 1 when the distributions agree perfectly, and 0 when they do not overlap at all.

H.2 ANTIBODIES

H.2.1 DATA

Antibody sequence models were trained on a filtered subset Turnbull et al. (2024) of the antibody heavy chains from the Observed Antibody Space database (OAS) (Kovaltsuk et al., 2018). The filtering retained only human sequences and removed sequences with unknown residues, missing conserved cysteines and those which had framework region 1 or 4 shorter than allowed by IMGT definitions. Sequences were further filtered to remove redundancy by clustering at 95% identity. This resulted in a total of 118 million sequences, of which a random subset of 15 million were sampled and used for training. For the evaluation of the model, a similarly-filtered subset of the OAS data was used.

H.2.2 BRANCHING FLOWS SPECIFICATION

- Conditional base process:
  - Discrete: DFM convex interpolation (appendix F.2) of x_0, uniform noise, and x_1, with $\omega_u = 0.2$, and CDFs F_1 and F_2 of $\mathcal{D}_1 = \text{Beta}(1.5, 1.5)$, and $\mathcal{D}_2 = \text{Beta}(2, 2)$, respectively.
- Branching hazard distribution: $\text{Beta}(1, 2)$
- Deletion hazard distribution: $U(0, 1)$
- To-be-deleted policy: deletion rate $d_r = 1.2$
- x_0 state distribution: masked tokens.
- x_0 length distribution: uniform between 110 and 140.

H.2.3 MODEL AND TRAINING

For the antibody sequence modeling task, we used a transformer-based architecture similar to that of a diffusion transformer (Peebles & Xie, 2023), but with additional split and deletion heads. Time was embedded with RFFs, and adaptive layer normalization (adaLN-Zero) was used for time conditioning. The model had an embedding dimension of 768, 12 attention heads, and 8 layers.

The models (Branching Flows, oracle-length and Edit Flows) were trained using a batch size of 64 resulting in a total of 234 375 training steps for the 15 million sequence dataset. Muon was used with a 2000 step linear warm-up followed by a cosine decay starting at 10^{-3} and decaying to 10^{-7}.

H.2.4 ANTIBODY SEQUENCE GENERATION: METRICS AND COMPARISONS

To evaluate Branching Flows, 10 000 sequences were generated with 1 000 uniformly-spaced steps from Branching Flows, Oracle Length, and Edit Flows models and then compared to natural ones. Sequence diversity was assessed by computing the minimum pairwise cosine distance (over vectors of 3-mer counts) within each set. Sequence 'novelty' was evaluated by computing the minimum Levenshtein distance to any sequence in a hold-out validation data set (needed because otherwise the novelty of all natural sequences would be zero).

Sequence 'nativeness' was computed using AbNatiV Ramon et al. (2024). Finally, the distribution of amino acids, distribution of sequence lengths and the distribution of CDR3 lengths were also computed. The latter was computed using ANARCI Dunbar & Deane (2016).

For investigating the training dynamics of Branching Flows compared to Oracle Length and Edit Flows models the losses cannot be directly compared, so after every 500 steps during training, 5 sequences were generated (with 200 uniformly-spaced steps) and saved. The perplexity of the generated sequences was computed as evaluated by a 130 million parameter autoregressive model trained on the data from Turnbull et al. (2024).

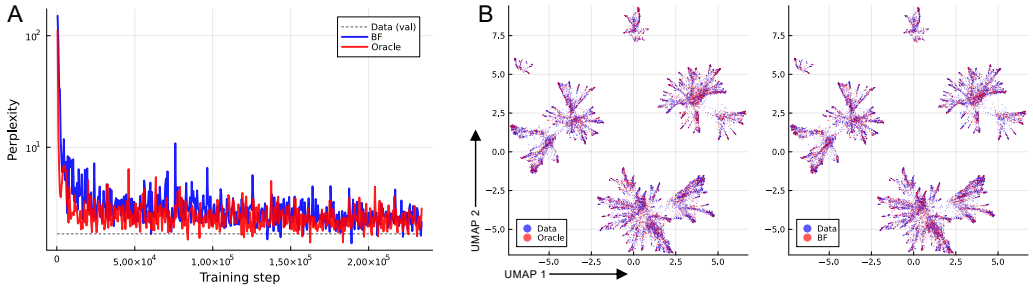

Figure 6: **Antibody generation distribution matching.** **A** The perplexity, evaluated under an autoregressive LLM, of samples generated by Branching Flows vs an oracle-length discrete flow matching model (where the length for each sample is taken from a random real sequence). This is shown over training iterations. **B** two seqUMAP (Hanke et al., 2022) plots comparing the clustering of real data with the oracle-length samples and the Branching Flows samples, respectively.

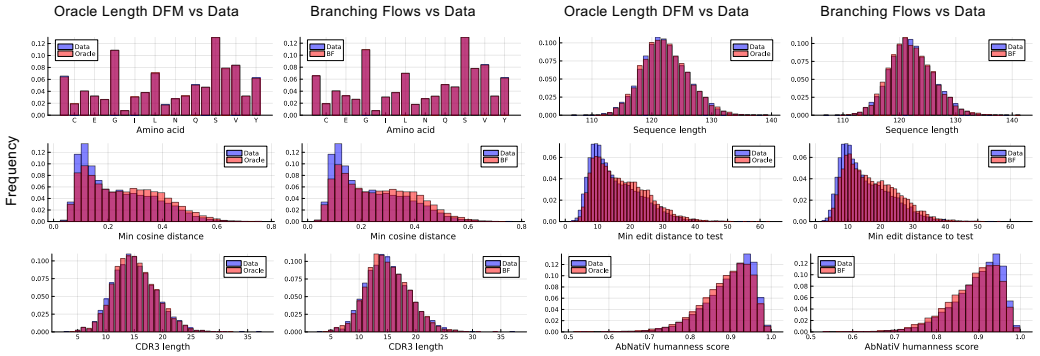

Figure 7: **Branching Flows vs Oracle Length.** Shown are property histograms for antibody sequence generation, for properties with KS agreement shown in table 2.

H.3 PROTEINS

To investigate whether Branching Flows would scale to a more complex learning problem where the state space is multimodal, we modified and finetuned a pre-trained ChainStorm model (Oresten et al., 2025) to allow the model to determine how long chains, or designable segments, should be.

Briefly, ChainStorm is a multimodal flow matching model that jointly generates protein backbones and amino acid sequence labels (but not side chain atom positions). Amino acid backbones are represented as sequences of 'frames' (Jumper et al., 2021) with Euclidean positions and SO(3) rotations, and ChainStorm uses an Invariant Point Attention (Jumper et al., 2021) transformer architecture to estimate the $t = 1$ frame positions and orientations, whose conditional/marginal paths are governed by Euclidean Brownian Motion and SO(3) rotational diffusion. Discrete Flow Matching (Gat et al., 2024) is used for amino acid sequence labels.

ChainStorm's Euclidean and SO(3) conditional processes were constructed with relatively low noise, but we reasoned that Branching Flows, especially with late-time branching events requiring higher flow velocities, might benefit from noisier processes which would have higher flow velocities generally, so we first finetuned a ChainStorm model with modified element-wise conditional processes, using

- for Euclidean positions, a time-inhomogeneous OU process (appendix F.1) with mean reversion rate of 100, and a variance of 150 at $t = 0$ that decays to 0 as $t \to 1$.
- for SO(3) rotations, a manifold-respecting version of the above OU process with identical parameters.

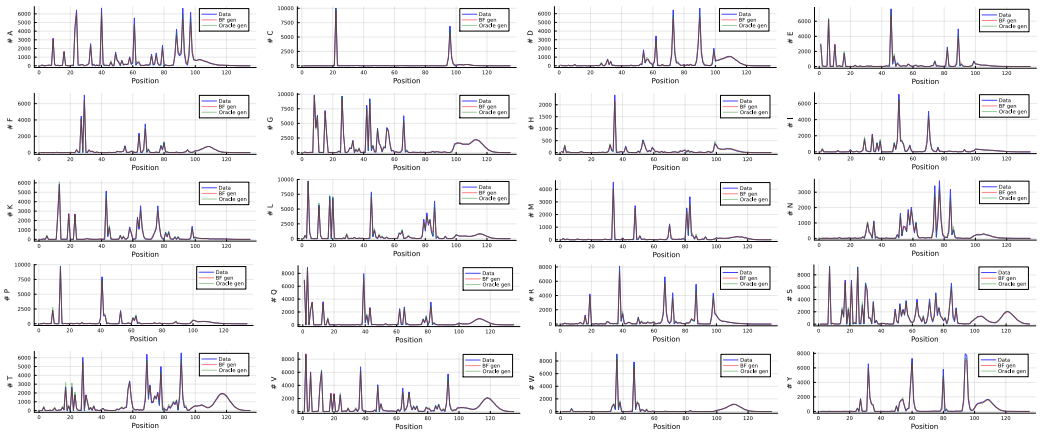

Figure 8: **Amino acid position distributions.** The count of the number of each amino acid at every position in natural data, and in samples from Branching Flows ('BF gen') and Oracle Length DFM ('Oracle gen').

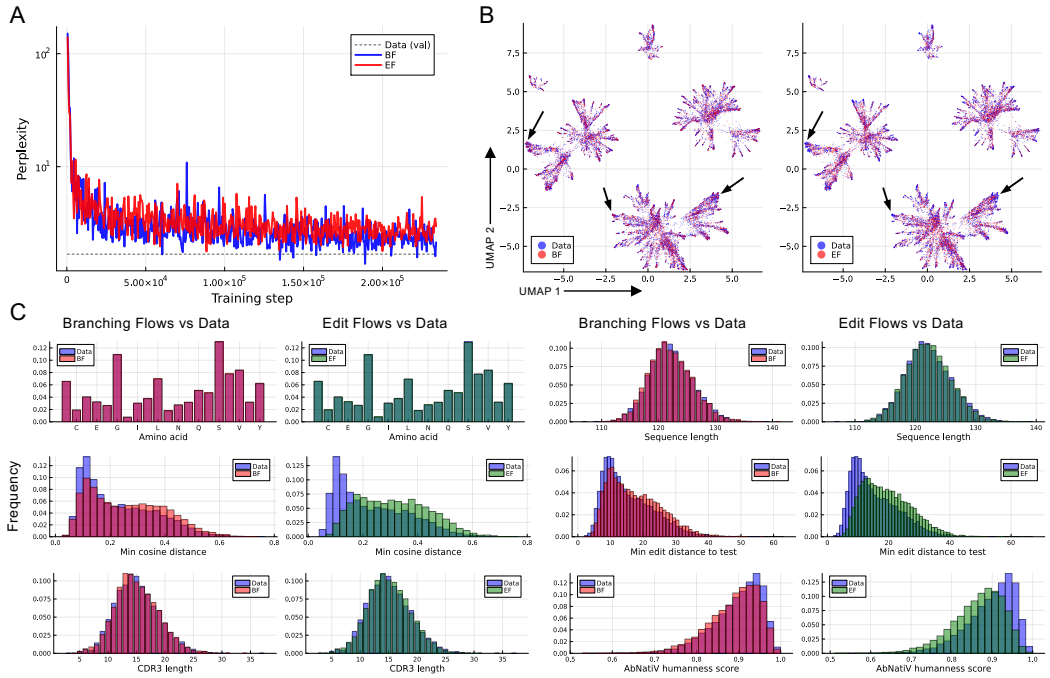

Figure 9: **Branching Flows vs Edit Flows.** **A** The perplexity, evaluated under an autoregressive LLM, of samples generated by Branching Flows (BF) vs Edit Flows (EF), which is a discrete-only variable length flow matching strategy. This is shown over training iterations. **B** two seqUMAP (Hanke et al., 2022) plots comparing Branching Flows and Edit Flows (respectively) each against real sequences. Arrows point to regions, at the extremities of the UMAP-embedded shapes, where Edit Flows density appears to be reduced. **C** Comparison of several distributions of the generated sequences with sequences from the validation data set.

- for discrete states, a DFM convex interpolation (appendix F.2) with a $\mathcal{D}_1 = \text{Beta}(3.0, 1.5), \mathcal{D}_2 = \text{Beta}(2, 2), \omega_u = 0.2$.

All Branching Flows protein models were then finetuned atop this model, which exhibited much higher variance sampling paths than the original ChainStorm.

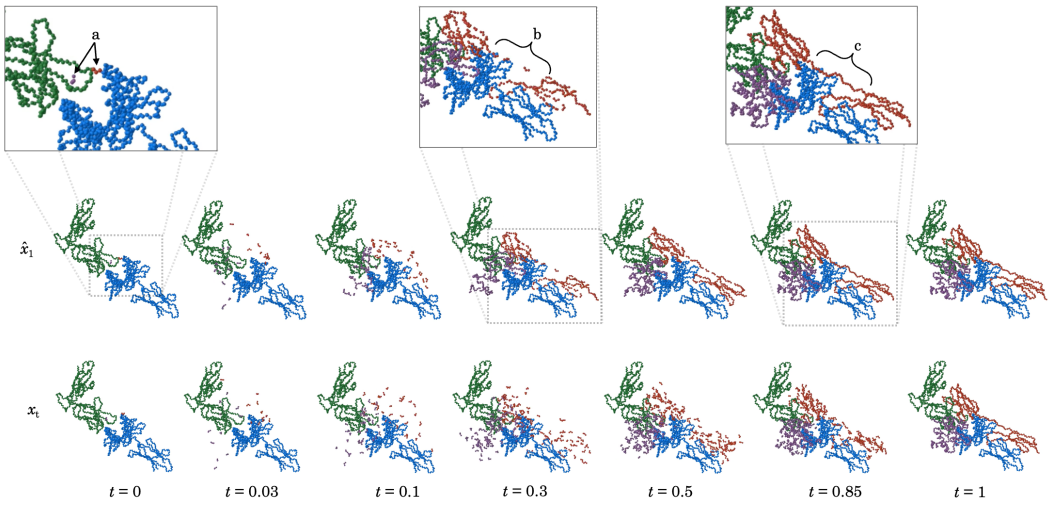

Figure 10: **Branching Flows protein sampling trajectories.** Shown are snapshots from a sampling trajectory from BF-ChainStorm, with both the current state x_t and the model's prediction of the end state, $\hat{x_1}$. The trajectories are colored by chain, and each backbone residue is shown as three spheres $(N, C\alpha, C)$. Here, two chains were fixed, and two chains were designable, with each designable chain starting from a single residue (arrow a in the first inset). The chains grow over time, and in this example the orange chain converges on two separate domains. Interestingly, during the intermediate flow state ($t = 0.3$, bracket b in the second inset), as the two domains resolve in $\hat{x_1}$, they are not connected to each other, and the model uses split events to build a complete linker between them by $t = 1$.

We then modified the ChainStorm architecture to construct 'BF-ChainStorm' to allow for i) element branching and deletion, and ii) conditioning on fixed protein components (in general any set of residues, but here chains or contiguous segments of chains) that are then prevented from changing during training or inference. Conditioning is done by including a 'conditioning mask' embedding layer whose outputs get summed into the initial activations, and by preventing the per-layer frame updates for fixed residues. For element branching and deletions, we reasoned that, since we are finetuning into an existing model, we didn't want to only extract contributions from the final output layer (since this is used for rotations, positions, and discrete states), so we concatenate the output activations from the last three layers. This is summed into an embedding of t and that sum is passed into a SwiGLU-like layer that collapses to a single dimension. This is done separately for branching and deletion, and the outputs are interpreted as 'logits' for loss (during training) and rates (during inference). Finally, since the state dimension now changes during inference, we replaced ChainStorm's self-conditioning with recycling, which requires no architectural changes but instead requires additional model passes within each inference step.

One subtlety related to Branching Flows is that, originally, ChainStorm conditioned on residue indices which provide, during training, the model with information about when parts of backbone are missing (i.e. unresolved in the crystal or cryo-EM structures) since a gap in the residue numbering will be present. To generate backbones without 'chain breaks' (usually the desired behavior), at inference time the generation is conditioned on residue numbers with no gaps. However, since BF-ChainStorm's conditional/marginal paths change by insertions and deletions with t, including residue numbers no longer makes sense, as the number of residues for any intervening segment (which, in BF-ChainStorm, must be stochastically generated during inference) would be pre-determined by the residue numbering. So instead of residue indices, we use fixed-increment positional index. However, since the training data has many chain breaks, the model would then stochastically generate structures with chain breaks. So we additionally introduced a 'break mask' encoder (also summed into the starting activations) that encodes, on a per-chain basis, whether the 'designable' portion of the chain includes any chain breaks. When used during inference, this reduces the number of chain breaks but doesn't remove them entirely, possibly because the model is getting conflicting signals between the conditioner and the emerging structure (e.g. large unstruc-

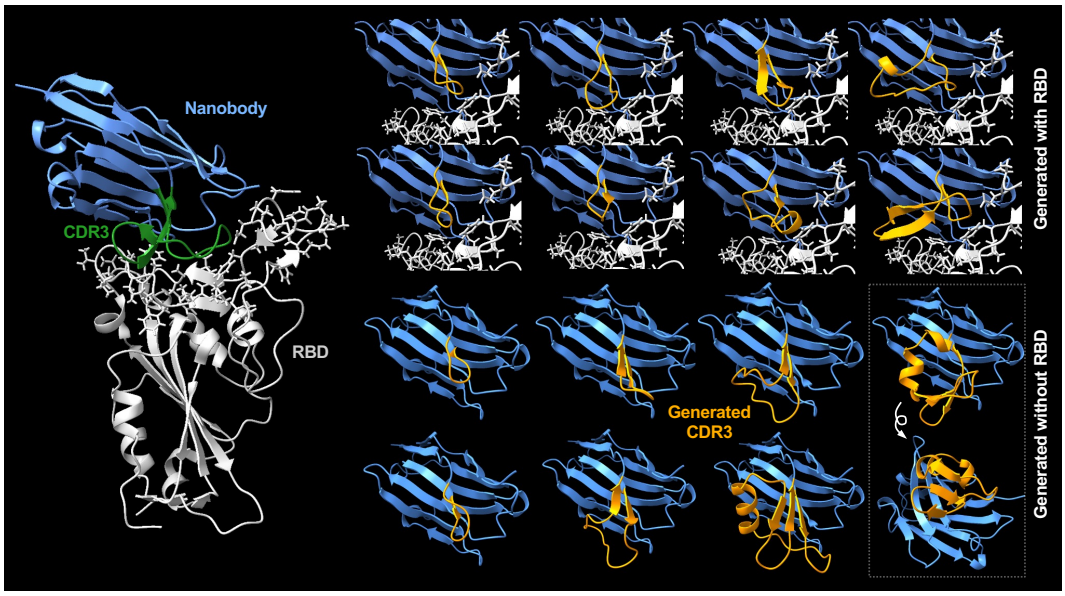

Figure 11: **Branching Flows protein 'infix' samples.** Depicted is the template PDB (9IQP, Shcheblyakov et al. (2025)) where only the CDR3 (green in the template) was designable, and the rest of the structure was fixed and conditioned upon. The top two rows of samples show generated CDR3s when they are generated in the context of the binding partner, and the bottom two rows show generations in the absence of the binding partner.

tured loops are often not resolved in protein structures, so if the model is generating one that might overwhelm the chain-break conditioner). Chain breaks are easy to detect using bond distances, and we exclude them from refolding analysis because our folding pipeline requires intact chains.

When sampling trees T for protein models, when the group (described above) is the chain index, the requirement that separate chains never occur on the same tree allows the number of sampled chains in a structure to be specified at inference time (but each of their lengths stochastically sampled by the model). Similarly, any residues (even within a single chain) separated by residues fixed in the conditioning (relevant for BranchSegment) may not coalesce when sampling trees. Thus no single tree spans two designable segments.

For BranchChain, we initialize each chain to have $1 + \text{Poisson}(20)$ x_0 elements, each drawn from the ChainStorm base element distribution, and for BranchSegment each designable segment has a single x_0 element. For sampling 'to-be-deleted' elements, we use $d_r = 1.2$ for BranchChain and $dr = 1.1$ for BranchSegment (reasoning that the more strongly informative conditioning information for BranchSegment might require less exploratory flow paths). Samples were generated with 400 steps: $t = 1 - (\cos(\pi\tau) + 1)/2$ for $\tau \in (0, 1/400, 2/400, ..., 1)$.

For illustration purposes, we also finetuned a BranchChain variant with only a single x_0 element per chain. Figure 10 shows steps along the sampling trajectory for a sample where two chains (blue and green) were fixed, and two chains were sampled. The orange chain, best seen through the lens of the model-predicted end state x_1^{hat}, highlights a key Branching Flows feature: at first, the designed chain converges upon two separate domains, with insufficient residues in between them for them to be linked. This is then progressively filled in by insertions, until a complete linker connects them as a single chain.

To investigate whether BranchChain can generate samples with reasonable protein-like geometry, we sampled, without conditioning, 20 monomers and 20 dimers, discarding samples with chain breaks (one monomer and five dimers) and samples too short for meaningful evaluation (< 25 residues in the longest chain: five monomers, and 1 dimer). Using the 'pseudoMSA refolding' strategy (Oresten et al., 2025), we generated a pseudoMSA of 100 sequences for each design with ProteinMPNN (Dauparas et al., 2022), and evaluate the 'self-consistency TM scores' of a single refold, via Boltz2 (Passaro et al., 2025) from this pseudoMSA.

Figure 3 shows the scTM scores, most very close to 1, showing that our designs have geometry consistent with Boltz2. We also depict selected examples, for monomers and dimers, of BranchChain generated proteins and their refolds, specifically showing the three cases with low scTMs. For the two poorly refolded monomers, this is because the generated structure had one or more flexible linkers, and the individual domains were well refolded. For the poorly refolded dimer, this was because BranchChain generated two disconnected chains, and Boltz2 placed these adjacent. Overall, BranchChain generates samples with a variety of protein chain lengths, with plausible backbone geometry. Further, as can be seen in many of the samples in Figure 3, the propensity of the ChainStorm base model to sample homodimers is preserved. This is interesting given that ChainStorm's single attention head that governs symmetric oligomer sampling is primarily driven by chain lengths which are fixed in ChainStorm, but now stochastically generated in BranchChain.

For BranchSegment we do not attempt quantitative evaluations, but we demonstrate that it is a promising solution to the 'unknown length infix sampling' problem. Taking a protein complex (PDB: 9IQP, Shcheblyakov et al. (2025)) of a nanobody and the SARS-CoV-2 receptor binding domain (RBD), we mask only the 'Complementarity Determining Region 3' (CDR3) as designable, and we generate 20 samples from the RBD-bound complex, and 20 samples from the unbound nanobody alone. The sampled CDR3s had a broad length distribution (7 to 29 residues for bound, 7 to 40 unbound). For the bound samples, even though the side chain atom positions from the RBD were not provided to BranchSegment (it only sees backbone frames and amino acid labels) the designed CDR3s had no atomic clashes with the RBD side chain rotamers for 15 out of 20 of the generated CDR3s. Some of the unbound CDR3s appear structurally plausible, but are extremely unlikely to occur naturally, which is not surprising since neither the base model nor BranchSegment have any antibody or nanobody-specific training.

I Related Work

I.1 Edit Flows

Havasi et al. (2025) describe Edit Flows, a method for varying-dimensionality generation of discrete sequences. Edit Flows works by parameterizing the rates of *edit operations* (insert, delete and substitute). During training, an *auxiliary alignment process* is introduced. This entails adjoining to the alphabet a null token ε such that the sequence length of X_0 is the same as the sequence length of X_1. One denotes this new space by $\mathcal{X} \times \mathcal{Z}$. Then, transitions in this augmented space are interpreted as either insertions (swapping ε to a token in the alphabet), deletions (swapping a token in the alphabet for ε) or a substitutions (swapping two tokens in the alphabet). The model only sees t and $\mathcal{X}$, and the marginal rates of a CTMC on $\mathcal{X}$ are learnt by training against a CTMC on $\mathcal{X} \times \mathcal{Z}$. Edit Flow's main contribution is proving, for the discrete case, that you can marginalize over this time-dependent latent process. In Branching Flows we sample a non-time-dependent latent that fully specifies how elements in X_0 will map, via splits or deletions, to elements in X_1, allowing us to extend beyond discrete states, and affording flexibility in how the sample paths are constructed. Furthermore, to extend Edit Flows to a continuous setting one would have to parametrize and learn a flexible distribution over insertions into a continuous space, which is often hard in practice. Later, One Flow (Nguyen et al., 2025) uses the approach of Edit Flows to allow the interleaving of images and text by inserting 'image tokens'.

I.2 Transdimensional jumps

Transdimensional jump diffusion (Campbell et al., 2023) models continuous-valued variable-cardinality states. Between jumps, a base SDE evolves the continuous coordinates, while at jump times, the dimension changes by one. In the forward direction (data noising — diffusion models adopt the opposite convention to flow matching), one component is deleted at a specified rate, chosen such that the forward process will terminate at a single surviving component.

The time reversal of the forward process is again a jump diffusion process, but where elements are inserted into the state, and a neural network parametrizes the insertion state and post-insertion distribution of the inserted element. One challenge is that, unlike Branching Flows's 'split' mechanism, the insertion distribution does not have a simple form, and any tractable approximation used in practice (e.g. a Gaussian) can cause a mismatch between the conditional and marginal process.

I.3 DRUGFLOW

In Schneuing et al. (2025) a trans-dimensional model for small molecule generation is presented. It works by introducing a virtual atom type whose coordinates are set to the center of mass of the ligand and which have no bonds. During training the training data is augmented by adding $n_{\text{virt}} \sim U(\{0, 1, \ldots, N_{\text{max}}\})$ virtual nodes, to allow the model to learn to set some atom categories to the virtual category. The model is thus deletion-only and requires specifying a priori N_{max} which leads to a fixed upper bound on generated molecule size. This contrasts to Branching Flows, which can generate arbitrary-length objects and allows both insertions and deletions. Furthermore, elements in the virtual category are retained in the flow and evolve under it (although their bond types are set to 'None'), which forces model computation to always be done on the maximum number of elements for the specific molecule being generated.