Entropy Coding of Unordered Data Structures

Julius Kunze [1] **Daniel Severo** [2 3] **Giulio Zani** [4] **Jan-Willem van de Meent** [4] **James Townsend** [4]

Abstract

We present shuffle coding, a general method for optimal compression of sequences of unordered objects using bits-back coding. Data structures that can be compressed using shuffle coding include multisets, graphs, hypergraphs, and others. We release an implementation that can easily be adapted to different data types and statistical models, and demonstrate that our implementation achieves state-of-the-art compression rates on a range of graph datasets including molecular data.

1. Introduction

The information stored and communicated by computer hardware, in the form of strings of bits and bytes, is inherently ordered. A string has a first and last element, and may be indexed by numbers in $\mathbb{N}$, a totally ordered set. For data like text, audio, or video, this ordering carries meaning. However, there are numerous data structures in which the 'elements' have no meaningful order. Common examples include graphs, sets, and 'map-like' datatypes such as JSON. Recent applications of machine learning to molecular data benefit from large datasets of molecules, which are graphs with vertex and edge labels representing atom and bond types (some examples are shown in Table 1 below). All of these data are necessarily stored in an ordered manner on a computer, but the order then represents *redundant information*. This work concerns optimal lossless compression of unordered data, and we seek to eliminate this redundancy.

Recent work by Severo et al. (2023a) showed how to construct an optimal lossless codec for (unordered) multisets from a codec for (ordered) vectors, by storing information in an ordering. Their method depends on the simple structure of multisets' automorphism groups, and does not extend to other unordered objects such as unlabelled graphs. In this paper we overcome this issue and develop *shuffle coding*, a method for constructing codecs for general 'unordered objects' from codecs for 'ordered objects'. Our definitions of ordered and unordered objects are based on the concept of 'combinatorial species' (Joyal, 1981; Bergeron et al., 1997), which were developed to assist with the enumeration of combinatorial structures. They include multisets, as well as all of the other unordered data structures mentioned above, and many more.

Although our method is applicable to any unordered object, we focus our experiments on unordered (usually referred to as 'unlabelled') graphs, as these are a widely used data type, and the improvements in compression rate from removing order information are large (as summarized in Table 2). We show that shuffle coding can achieve significant improvements relative to existing methods, when compressing unordered graphs under the $G(n, p)$ model of Erdős & Rényi (1960), as well as the recently proposed Pólya's urn-based model of Severo et al. (2023b). Shuffle Coding extends to graphs with vertex and edge attributes, such as the molecular and social network datasets of TUDatasets (Morris et al., 2020), which are compressed in Section 4. We release source code[1] with straightforward interfaces to enable future applications of shuffle coding with more sophisticated models and to classes of unordered objects other than graphs.

2. Background

This section gives definitions for ordered and unordered objects. Entropy coding is reviewed in Appendix B. Examples are given throughout the section for clarification.

For $n \in \mathbb{N}$, we let $[n] \coloneqq \{0, 1, \dots, n-1\}$, with $[0] = \emptyset$. The symmetric group of permutations on $[n]$, i.e. bijections from $[n]$ to $[n]$, will be denoted by $\mathcal{S}_n$. Permutations compose on the left, like functions, so for $s, t \in \mathcal{S}_n$, the product st denotes the permutation formed by performing t then s.

[1]Department of Computer Science, University College London [2]Department of Electrical and Computer Engineering, University of Toronto [3]Vector Institute for Artificial Intelligence, Toronto [4]Amsterdam Machine Learning Lab (AMLab), University of Amsterdam. Correspondence to: Julius Kunze <juliuskunze@gmail.com>, James Townsend <j.h.n.townsend@uva.nl>.

Accepted to the Neural Compression Workshop, at the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. Copyright 2023 by the author(s).

[1]Source code, data and detailed results are available at https://github.com/juliuskunze/shuffle-coding.

Table 1. Examples of molecules and their order information. The 'discount' column shows the saving achieved by shuffle coding by removing order information (see eq. 11). For each molecule $\mathbf{m}$, n is the number of atoms and $|\mathrm{Aut}(\mathbf{m})|$ is the size of the automorphism group. All values are in bits, and $\log$ denotes the binary logarithm.

Molecular structure		Permutation $\log n!$	Symmetry $\log\lvert\mathrm{Aut}(\mathbf{m})\rvert$	Discount $\log n! - \log\lvert\mathrm{Aut}(\mathbf{m})\rvert$
Nitric oxide	N=O	1.00	0.00	1.00
Water	H–O–H	2.58	1.00	1.58
Hydrogen peroxide	H–O–O–H	4.58	1.00	3.58
Ethylene	H_2C=CH_2	9.49	3.00	6.49
Boric acid	B(O–H)$_3$	12.30	2.58	9.71

Permutations are represented as follows

$$\begin{smallmatrix} 0 \leftarrow 1 \\ \searrow\ \nearrow \\ 2 \end{smallmatrix} = (2, 0, 1) \in \mathcal{S}_3. \tag{1}$$

The glyph on the left-hand side represents the permutation that maps 0 to 2, 1 to 0 and 2 to 1. This permutation can also be represented concretely by the vector $(2, 0, 1)$.

Concepts from group theory, including subgroups, cosets, actions, orbits, and stabilizers are used throughout. We provide an overview of the required background in Appendix A.

We will be compressing objects which can be 're-ordered' by applying permutations. This is formalized in the following definition:

Definition 2.1 (Permutable class[2])**.** For $n \in \mathbb{N}$, a *permutable class* of order n is a set $\mathcal{F}$, equipped with a left group action of the permutation group $\mathcal{S}_n$ on $\mathcal{F}$, which we denote with the $\cdot$ binary operator. We refer to elements of $\mathcal{F}$ as *ordered objects*.

Example 2.2 (Simple graphs $\mathcal{G}_n$)**.** Let $\mathcal{G}_n$ be the set of simple graphs with vertex set $[n]$. Specifically, an element $g \in \mathcal{G}_n$ is a set of 'edges', which are unordered pairs of distinct elements of $[n]$. We define the action of $\mathcal{S}_n$ on a graph by moving the endpoints of each edge in the direction of the arrows, for example

$$\begin{smallmatrix} 0 \leftarrow 1 \\ \searrow \uparrow \\ \circlearrowleft 3 \quad 2 \end{smallmatrix} \cdot \begin{smallmatrix} 0 - 1 \\ \diagdown\, | \\ 3 - 2 \end{smallmatrix} = \begin{smallmatrix} 0 - 1 \\ \times\ | \\ 3 \quad 2 \end{smallmatrix}. \tag{2}$$

Our main contribution in this paper is a general method for compressing *unordered* objects. These may be defined formally in terms of the equivalence classes, known as orbits, which comprise objects that are identical up to re-ordering (see Appendix A for background):

Definition 2.3 (Isomorphism, unordered objects)**.** For two objects f and g in a permutable class $\mathcal{F}$, we say that f is *isomorphic* to g, and write $f \simeq g$, if there exists $s \in \mathcal{S}_n$ such that $g = s \cdot f$ (i.e. if f and g are in the same orbit under the action of $\mathcal{S}_n$). Note that the relation $\simeq$ is an equivalence relation. For $f \in \mathcal{F}$ we use $\tilde{f}$ to denote the equivalence class containing f, and $\widetilde{\mathcal{F}}$ to denote the quotient set of equivalence classes. We refer to elements $\tilde{f} \in \widetilde{\mathcal{F}}$ as *unordered objects*.

For the simple graphs in Example 2.2, the generalized isomorphism in Definition 2.3 reduces to the usual notion of graph isomorphism. We can define a shorthand notation for unordered graphs, with points at the nodes instead of numbers:

$$\begin{smallmatrix} \cdot - \cdot \\ \diagdown\, | \\ \cdot - \cdot \end{smallmatrix} := \widetilde{\begin{smallmatrix} 0 - 1 \\ \diagdown\, | \\ 3 - 2 \end{smallmatrix}}. \tag{3}$$

Using this notation, the unordered simple graphs on three vertices, for example, can be written:

$$\widetilde{\mathcal{G}_3} = \left\{ \begin{smallmatrix} \cdot \quad \cdot \\ \cdot \end{smallmatrix}, \begin{smallmatrix} \cdot - \cdot \\ \cdot \end{smallmatrix}, \begin{smallmatrix} \cdot - \cdot \\ \diagdown \\ \cdot \end{smallmatrix}, \begin{smallmatrix} \cdot - \cdot \\ \diagdown\,\diagup \\ \cdot \end{smallmatrix} \right\}. \tag{4}$$

Finally, we define the subgroup of $\mathcal{S}_n$ containing the symmetries of a given object f:

Definition 2.4 (Automorphism group)**.** For an element f of a permutable class $\mathcal{F}$, we let $\mathrm{Aut}(f)$ denote the *automorphism group* of f, defined by

$$\mathrm{Aut}(f) := \{s \in \mathcal{S}_n \mid s \cdot f = f\}. \tag{5}$$

This is the stabilizer subgroup of f under the action of $\mathcal{S}_n$.

[2]This definition is very close to that of a 'combinatorial species', the main difference being that we fix a specific n. See discussion in (Yorgey, 2014).

The elements of the automorphism group of the simple graph from Example 2.2 are:

$$\mathrm{Aut}\left(\begin{array}{c} 0 - 1 \\ \diagdown\ | \\ 3 - 2 \end{array}\right) = \left\{ \begin{array}{cc} \circlearrowleft 0 & 1 \circlearrowright \\ \circlearrowleft 3 & 2 \circlearrowright \end{array}, \quad \begin{array}{cc} 0 \rightleftarrows 1 & \\ \circlearrowleft 3 & 2 \circlearrowright \end{array} \right\}. \tag{6}$$

2.1. Canonical orderings

To define a codec for unordered objects, we will introduce the notion of a 'canonical' representative of each equivalence class in $\widetilde{\mathcal{F}}$. This allows us, for example, to check whether two ordered objects are isomorphic, by mapping both to the canonical representative and comparing.

Definition 2.5 (Canonical ordering)**.** A *canonical ordering* is an operator $\bar{\cdot} : \mathcal{F} \to \mathcal{F}$, such that

1. For $f \in \mathcal{F}$, we have $\overline{f} \simeq f$.
2. For $f, g \in \mathcal{F}$, $\overline{f} = \overline{g}$ if and only if $f \simeq g$.

For graphs, the canonical orderings we use are computed using the `nauty` and `Traces` libraries (McKay & Piperno, 2014). The libraries provide a function, which we call `canon_perm`, which, given a graph g, returns a permutation s such that $s \cdot g = \overline{g}$. As well as `canon_perm`, `nauty` and `Traces` are able to compute the automorphism group of a given graph, via a function which we refer to as `aut`.[3]

While permutable objects other than graphs cannot be directly canonized by `nauty` and `Traces`, it is often possible to embed objects into graphs in such a way that the structure is preserved and the canonization remains valid (Anders & Schweitzer, 2021).

3. Codecs for unordered objects

Our main contribution in this paper is a generic codec for unordered objects, i.e. a codec respecting a given probability distribution on $\widetilde{\mathcal{F}}$. We first derive an expression for the rate that this codec should achieve, then in Section 3.1 we describe the codec itself.

To help simplify the presentation, we will use the following generalization of exchangeability from sequences of random variables to arbitrary permutable classes:

Definition 3.1 (Exchangeability)**.** For a probability distribution P defined on a permutable class $\mathcal{F}$, we say that P is *exchangeable* if isomorphic objects have equal probability under P, i.e. if

$$f \simeq g \Rightarrow P(f) = P(g). \tag{7}$$

[3] In fact, a list of generators for the group is computed, rather than the entire group, which may be very large.

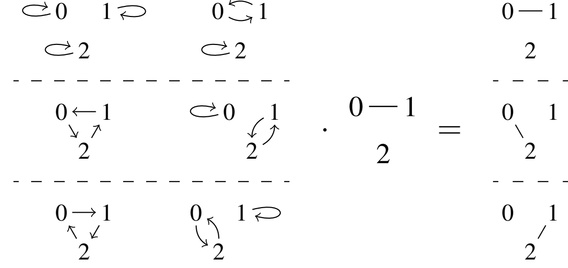

Figure 1. Visualization of Lemma 3.3. For a fixed graph g, the six elements $s \in \mathcal{S}_3$ can be partitioned according to the value of $s \cdot g$. The three sets in the partition are the left cosets of $\mathrm{Aut}(g)$.

We can assume, without loss of modelling power, that unordered objects are generated by first generating an ordered object from an exchangeable distribution and then 'forgetting the order':

Lemma 3.2 (Symmetrization)**.** *For any distribution Q on a class of unordered objects $\widetilde{\mathcal{F}}$, there exists a unique exchangeable distribution P on ordered objects $\mathcal{F}$ for which*

$$Q(\tilde{f}) = \sum_{g \in \tilde{f}} P(g). \tag{8}$$

Proof. For existence, set $P(f) := Q(\tilde{f})/|\tilde{f}|$ for $f \in \mathcal{F}$, and note that $g \in \tilde{f} \Rightarrow \tilde{g} = \tilde{f}$. For uniqueness, Definition 3.1 implies that the restriction of P to any particular class must be uniform, which completely determines P. □

We will model real-world permutable objects using an exchangeable model, which will play the role of P in Equation (8). To further simplify our rate expression we will also need the following application of the orbit-stabilizer theorem (see Appendix A for more detail), which is visualized in Figure 1:

Lemma 3.3. *Given a permutable class $\mathcal{F}$, for each object $f \in \mathcal{F}$, there is a fixed bijection between the left cosets of $\mathrm{Aut}(\overline{f})$ in $\mathcal{S}_n$ and the isomorphism class $\tilde{f}$. This is induced by the function $\theta_f : \mathcal{S}_n \to \mathcal{F}$ defined by $\theta_f(s) := s \cdot \overline{f}$. This implies that*

$$|\tilde{f}| = \frac{|\mathcal{S}_n|}{|\mathrm{Aut}(f)|} = \frac{n!}{|\mathrm{Aut}(f)|}. \tag{9}$$

Proof. Follows directly from the orbit-stabilizer theorem (Theorem A.4) and the definitions of Aut, $\overline{f}$ and $\tilde{f}$. □

For any $f \in \mathcal{F}$, this allows us to express the right hand side of Equation (8) as:

$$\sum_{g \in \tilde{f}} P(g) = |\tilde{f}| P(f) = \frac{n!}{|\mathrm{Aut}(f)|} P(f) \tag{10}$$

where the first equality follows from exchangeability of P, and the second from Equation (9). Finally, from Equations (8) and (10), we have the following rate expression, which a codec on unordered objects should achieve:

$$\log \frac{1}{Q(\tilde{f})} = \underbrace{\log \frac{1}{P(f)}}_{\text{Ordered rate}} - \underbrace{\log \frac{n!}{|\mathrm{Aut}(f)|}}_{\text{Discount}}. \quad (11)$$

Note that only the $\log 1/P(f)$ term depends on the choice of model. The $\log(n!/|\mathrm{Aut}(f)|)$ term can be computed directly from the data, and is the 'discount' that we get for compressing an *unordered* object vs. compressing an ordered one. The discount is larger for objects which have a smaller automorphism group, i.e. objects which *lack symmetry*. It can be shown that almost all simple graphs have a trivial automorphism group for large enough n (Bollobás, 2001, Chapter 9), and in practice the discount is usually equal to or close to $\log n!$.

3.1. Achieving the target rate for unordered objects

In Listing B.1 we give examples of codecs for ordered strings and simple graphs which achieved the 'ordered rate'. To operationalize the negative 'discount' term, we can use the 'bits-back with ANS' method introduced by Townsend et al. (2019), the key idea being to *decode* an ordering as part of an *encode* function (see line 4 in the code below).

The value of the negative term in the rate provides a hint at how exactly to decode an ordering: the discount is equal to the logarithm of the number of cosets of $\mathrm{Aut}(\overline{f})$ in $\mathcal{S}_n$, so a uniform codec for those cosets will consume exactly that many bits. Lemma 3.3 tells us that there is a direct correspondence between the cosets of $\mathrm{Aut}(\overline{f})$ and the set $\tilde{f}$, so if we uniformly decode a choice of coset, we can reversibly map that to an ordering of f.

The following is an implementation of shuffle coding, showing, in comments, the effect of the steps on message length.

```
1  def encode(m, f):
2    f_canon = action_apply(canon_perm(f), f)
3    # Changes message length by -log n!/|Aut(f)|:
4    m, s = UniformLCoset(f_canon.aut).decode(m)
5    g = action_apply(s, f_canon)
6    # Changes message length by +log 1/P(f):
7    m = P.encode(m, g)
8    return m
9
10 def decode(m):
11   m, g = P.decode(m)
12   s_ = inv_canon_perm(g)
13   f_canon = action_unapply(s_, f)
14   m = UniformLCoset(f_canon.aut).encode(m, s_)
15   return m, f_canon
```

The encode function accepts a pair `(m, f)`, and reversibly *decodes* a random choice `g` from the isomorphism class of `f`. This is done using `UniformLCoset`, discussed in detail in Appendix E. The canonization on line 2 is necessary for the decoder to recover the chosen coset and encode it on line 12. We avoid representing equivalence classes explicitly as sets, instead using a single element of the class as a representative. Thus the encoder accepts any `f` in the isomorphism class being encoded, and the decoder then returns the canonization of `f`. Similarly, `UniformLCoset.encode` accepts any element of the coset, and `UniformLCoset.decode` returns a canonical coset element.

4. Experiments

To demonstrate the method experimentally, we first applied it to the TUDatasets graphs (Morris et al., 2020), using a very simple Erdős-Rényi $G(n, p)$ model for P. Table 2 shows a summary, highlighting the significance of the discount achieved by shuffle coding. We compressed a dataset at a time (note that for each high-level graph type there are multiple datasets in TUDatasets). To handle graphs with discrete vertex and edge attributes, we treated all attributes as independent and identically distributed (i.i.d.) within each dataset. For each dataset, the codec computes and encodes a separate empirical probability vector for vertices and edges, as well as an empirical p parameter, and the size n of each graph. For encoding these meta-data we used `Uniform` codecs. Further details can be found in Appendix G.

In Appendix F, we additionally compare our method to PnC (Bouritsas et al., 2021) and SZIP (Choi & Szpankowski, 2012), achieving state-of-the-art compression rates on a range of graph datasets. We also provide runtimes and ablations.

Table 2. For the TUDatasets, this table shows the significance of the discount term in Equation (11). With an Erdős-Rényi (ER) model, with edge probability adapted to each dataset, the percentage improvement (Discount) is the difference between treating the graph as ordered (Ordered) and using Shuffle coding to forget the order (Shuffle coding). Rates are measured in bits per edge.

Graph type	Ordered	Shuffle coding	Discount
Small molecules	2.11	1.14	46%
Bioinformatics	9.20	6.50	29%
Computer vision	6.63	4.49	32%
Social networks[4]	3.98	2.97	26%
Synthetic	5.66	2.99	47%

[4] Three of the 24 social network datasets, REDDIT-BINARY, REDDIT-MULTI-5K, REDDIT-MULTI-12K, were excluded because compression running time was too long.

5. Conclusion

We have presented shuffle coding, the first general method which achieves an optimal rate when compressing sequences of unordered objects. We have also implemented experiments which demonstrate the practical effectiveness of shuffle coding for compressing many kinds of graphs, including molecules and social network data. We look forward to future work applying the method to other forms of unordered data, and applying more sophisticated probabilistic generative models to gain improvements in compression rate.

Acknowledgements

James Townsend acknowledges funding under the project VI.Veni.212.106, financed by the Dutch Research Council (NWO). We thank Ashish Khisti for discussions and encouragement, and Heiko Zimmermann for feedback on the paper.

References

Anders, M. and Schweitzer, P. Parallel Computation of Combinatorial Symmetries. August 2021.

Bergeron, F., Labelle, G., and Leroux, P. *Combinatorial Species and Tree-like Structures*. Encyclopedia of Mathematics and Its Applications. 1997.

Besta, M. and Hoefler, T. Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations, April 2019.

Bollobás, B. *Random Graphs*. Cambridge Studies in Advanced Mathematics. Second edition, 2001.

Bouritsas, G., Loukas, A., Karalias, N., and Bronstein, M. Partition and Code: Learning how to compress graphs. In *Advances in Neural Information Processing Systems*, volume 34, pp. 18603–18619, 2021.

Chen, X., Han, X., Hu, J., Ruiz, F., and Liu, L. Order Matters: Probabilistic Modeling of Node Sequence for Graph Generation. In *Proceedings of the 38th International Conference on Machine Learning*, pp. 1630–1639, July 2021.

Choi, Y. and Szpankowski, W. Compression of Graphical Structures: Fundamental Limits, Algorithms, and Experiments. *IEEE Transactions on Information Theory*, 58(2): 620–638, February 2012.

Duda, J. Asymmetric numeral systems. *arXiv:0902.0271 [cs, math]*, May 2009.

Erdős, P. and Rényi, A. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5(1):17–60, 1960.

Holt, D. F. *Handbook of Computational Group Theory*. Discrete Mathematics and Its Applications. 2005.

Joyal, A. Une théorie combinatoire des séries formelles. *Advances in Mathematics*, 42(1):1–82, October 1981.

Kingma, F., Abbeel, P., and Ho, J. Bit-Swap: Recursive Bits-Back Coding for Lossless Compression with Hierarchical Latent Variables. In *Proceedings of the 36th International Conference on Machine Learning*, pp. 3408–3417, May 2019.

Knuth, D. E. *The Art of Computer Programming*, volume 2. 2nd ed. edition, 1981.

McKay, B. D. and Piperno, A. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, January 2014.

Morris, C., Kriege, N. M., Bause, F., Kersting, K., Mutzel, P., and Neumann, M. TUDataset: A collection of benchmark datasets for learning with graphs. In *ICML 2020 Workshop on Graph Representation Learning and beyond (GRL+ 2020)*, 2020.

Seress, Á. *Permutation Group Algorithms*. Number 152 in Cambridge Tracts in Mathematics. 2003.

Severo, D., Townsend, J., Khisti, A., Makhzani, A., and Ullrich, K. Compressing Multisets with Large Alphabets. *IEEE Journal on Selected Areas in Information Theory*, 2023a.

Severo, D., Townsend, J., Khisti, A. J., and Makhzani, A. Random Edge Coding: One-Shot Bits-Back Coding of Large Labeled Graphs. *Proceedings of the 40th International Conference on Machine Learning*, 2023b.

Sims, C. C. Computational methods in the study of permutation groups. In *Computational Problems in Abstract Algebra*, pp. 169–183. January 1970.

Steinruecken, C. *Lossless Data Compression*. PhD thesis, University of Cambridge, 2014.

Steinruecken, C. Compressing Sets and Multisets of Sequences. *IEEE Transactions on Information Theory*, 61 (3):1485–1490, March 2015.

Steinruecken, C. Compressing Combinatorial Objects. In *2016 Data Compression Conference (DCC)*, pp. 389–396, March 2016.

Townsend, J. A tutorial on the range variant of asymmetric numeral systems, January 2020.

Townsend, J. *Lossless Compression with Latent Variable Models*. PhD thesis, April 2021.

Townsend, J., Bird, T., and Barber, D. Practical lossless compression with latent variables using bits back coding. In *International Conference on Learning Representations (ICLR)*, 2019.

Varshney, L. and Goyal, V. Toward a source coding theory for sets. In *Data Compression Conference (DCC'06)*, pp. 13–22, March 2006.

Yorgey, B. *Combinatorial Species and Labelled Structures*. PhD thesis, University of Pennsylvania, 2014.

Zhu, Y., Du, Y., Wang, Y., Xu, Y., Zhang, J., Liu, Q., and Wu, S. A Survey on Deep Graph Generation: Methods and Applications. In *Proceedings of the First Learning on Graphs Conference*, pp. 47:1–47:21, December 2022.

A. Group actions, orbits and stabilizers

This appendix gives the definitions of group actions, orbits and stabilizers as well as a statement and proof of the orbit-stabilizer theorem, which we make use of in Section 3. We use the shorthand $H \leq G$ to mean that H is a subgroup of G, and for $g \in G$, we use the usual notation, $gH \coloneqq \{gh \mid h \in H\}$ and $Hg \coloneqq \{hg \mid h \in H\}$ for left and right cosets, respectively.

Definition A.1 (Group action)**.** For a set X and a group G, a *group action*, or simply *action*, is a binary operator

$$\cdot_G : G \times X \to X \tag{12}$$

which respects the structure of G in the following sense:

1. The identity element $e \in G$ is neutral, that is $e \cdot_G x = x$.
2. The operator $\cdot_G$ respects composition. That is, for $g, h \in G$,

$$g \cdot_G (h \cdot_G x) = (gh) \cdot_G x. \tag{13}$$

We will often drop the subscript G and use infix $\cdot$ alone where the action is clear from the context.

Definition A.2 (Orbit)**.** An action of a group G on a set X induces an equivalence relation $\sim_G$ on X, defined by

$$x \sim_G y \quad \text{if and only if there exists} \quad g \in G \quad \text{such that} \quad y = g \cdot x. \tag{14}$$

We refer to the equivalence classes induced by $\sim_G$ as *orbits*, and use $\mathrm{Orb}_G(x)$ to denote the orbit containing an element $x \in X$. We use X/G to denote the set of orbits, so for each $x \in X$, $\mathrm{Orb}_G(x) \in X/G$.

Definition A.3 (Stabilizer subgroup)**.** For an action of a group G on a set X, for each $x \in X$, the *stabilizer*

$$\mathrm{Stab}_G(x) \coloneqq \{g \in G \mid g \cdot x = x\} \tag{15}$$

forms a subgroup of G.

We make use of the orbit-stabilizer theorem in Section 3. Here we give a statement and brief proof of this well-known theorem.

Theorem A.4 (Orbit-stabilizer theorem)**.** *For an action of a finite group G on a set X, for each $x \in X$, the function $\theta_x : G \to X$ defined by*

$$\theta_x(g) \coloneqq g \cdot x \tag{16}$$

induces a bijection from the left cosets of $\mathrm{Stab}_G(x)$ to $\mathrm{Orb}_G(x)$. This implies that the orbit $\mathrm{Orb}_G(x)$ is finite and

$$|\mathrm{Orb}_G(x)| = \frac{|G|}{|\mathrm{Stab}_G(x)|}. \tag{17}$$

Proof. We show that θ_f induces a well defined function on the left-cosets of $\mathrm{Stab}_G(x)$, which we call $\tilde{\theta}_f$. Specifically, we define

$$\tilde{\theta}_f(g\,\mathrm{Stab}_G(x)) \coloneqq g \cdot x, \tag{18}$$

and show that $\tilde{\theta}_f$ is injective and surjective.

To see that $\tilde{\theta}_f$ is well defined and injective, note that

$$h \in g\,\mathrm{Stab}_G(x) \iff g^{-1}h \in \mathrm{Stab}_G(x) \tag{19}$$

$$\iff g^{-1}h \cdot x = x \tag{20}$$

$$\iff g \cdot x = h \cdot x, \tag{21}$$

using the definition of Stab_G.

For surjectivity, we have

$$y \in \mathrm{Orb}_G(x) \implies \exists g \in G \text{ s.t. } y = g \cdot x \tag{22}$$

$$\implies y = \tilde{\theta}_f(g\,\mathrm{Stab}_G(x)) \tag{23}$$

using the definition of Orb_G. □

In Appendix E.2, it will be helpful to have an explicit bijection between G and the Cartesian product $\mathrm{Orb}_G(x) \times \mathrm{Stab}_G(x)$. This requires a way of selecting a canonical element from each left coset of $\mathrm{Stab}_G(x)$ in G. This is similar to the canonical ordering of Definition 2.5:

Definition A.5 (Transversal)**.** For a group G with subgroup $H \leq G$, a *transversal* of the left cosets of H in G is a mapping $t : G \to G$ such that

1. For all $g \in G$, we have $t(g) \in gH$.
2. For all $f, g \in G$, if $f \in gH$, then $t(f) = t(g)$.

Given such a transversal, we can setup the bijection mentioned above:

Lemma A.6. *Let G be a group acting on a set X. If, for $x \in X$, we have a transversal t_x of the left cosets of $\mathrm{Stab}_G(x)$ in G, then we can form an explicit bijection between G and $\mathrm{Orb}_G(x) \times \mathrm{Stab}_G(x)$.*

Proof. For $g \in G$, let

$$o_x(g) \coloneqq g \cdot x \tag{24}$$

$$s_x(g) \coloneqq t_x(g)^{-1} g, \tag{25}$$

then $o_x \in \mathrm{Orb}_G(x)$. By condition 1 in Definition A.5, there exists $h \in \mathrm{Stab}_G(x)$ such that $t(g) = gh$, and in particular $s_x(g) = h \in \mathrm{Stab}_G(x)$. So there is a well-defined function $\phi_x(g) \coloneqq (o_x(g), s_x(g))$ with $\phi_x : G \to \mathrm{Orb}_G(x) \times \mathrm{Stab}_G(x)$.

To see that ϕ_x is injective, suppose that $\phi_x(f) = \phi_x(g)$. Then $o_x(f) = o_x(g)$, so $f \cdot x = g \cdot x$, and therefore $f \in g\,\mathrm{Stab}_G(x)$. Condition 2 in Definition A.5 implies that $t_x(f) = t_x(g)$, and since $s_x(g) = s_x(f)$ we have $t_x(f)^{-1} f = t_x(g)^{-1} g$, so $f = g$.

The orbit-stabilizer theorem implies that $|G| = |\mathrm{Orb}_G(x)||\mathrm{Stab}_G(x)|$, and therefore if ϕ_x is injective it must also be bijective. ☐

B. Codecs

We fix a set M of prefix-free binary messages, and a length function $l \colon M \to [0, \infty)$, which measures the number of physical bits required to represent values in M.

Definition B.1 (Optimal codec)**.** For a set X and a probability distribution on X with mass function P, an *optimal codec* for P is an inverse pair of functions

$$\mathsf{encode} : M \times X \to M \qquad \text{and} \qquad \mathsf{decode} : M \to M \times X \tag{26}$$

which respect P in the sense that, for any $m \in M$ and $x \in X$,

$$l(\mathsf{encode}(m, x)) \approx l(m) + \log \frac{1}{P(x)}.^{5} \tag{27}$$

We refer to $\log \frac{1}{P(x)}$ as the *rate* of the codec.

Definition B.1 captures the abstract properties of codecs based on the range variant of asymmetric numeral systems (rANS). Note that the encode function requires a pre-existing message as its first input. Therefore, at the beginning of encoding we set m equal to some fixed, short initial message m_0, with length less than 64 bits. This constant 'initial bit cost' exists for all entropy coding methods, and is amortized as we compress more data.

We will assume access to three primitive codecs provided by rANS. These are

- `Uniform(n)`, respecting a uniform distribution on $\{$`0`, `1`, . . . , `n-1`$\}$.
- `Bernoulli(p)`, respecting a Bernoulli distribution with probability `p`.

[5] This condition, with a suitable definition of $\approx$, is equivalent to rate-optimality in the usual Shannon sense, see (Townsend, 2020).

- `Categorical(ps)`, respecting a categorical distribution with probability vector `ps` on $\{0, 1, \ldots, \texttt{len(ps)-1}\}$.

Listing B.1 shows how we can use the primitive codecs to implement codecs for strings and simple graphs. The string codec respects a distribution where each character is drawn i.i.d. from a categorical with known probabilities. For simple graphs, the codec respects the Erdős-Rényi $G(n, p)$ model, where each edge's existence is decided by an independent draw from a Bernoulli with known probability parameter. We will use these codecs for ordered objects as a component of shuffle coding.

Listing B.1 Left: Codec for fixed-length strings implemented by applying a `Categorical` codec to each character. Right: Codec for graphs respecting an Erdős-Rényi distribution $G(n, p)$, implemented by applying the `Bernoulli` codec to each edge.

```
def String(ps, length):
  def encode(m, string):
    assert len(string) == length
    for c in reversed(string):
      m = Categorical(ps).encode(m, c)
    return m

  def decode(m):
    string = []
    for _ in range(length):
      m, c = Categorical(ps).decode(m)
      string.append(c)
    return m, str(string)
  return Codec(encode, decode)
```

```
def ErdosRenyi(n, p):
  def encode(m, g):
    assert len(g) == n
    for i in reversed(range(n)):
      for j in reversed(range(i)):
        e = g[i][j]
        m = Bernoulli(p).encode(m, e)
    return m

  def decode(m):
    g = []
    for i in range(n):
      inner = []
      for j in range(i)
        m, e = Bernoulli(p).decode(m)
        inner.append(e)
      g.append(inner)
    return (m, g)
  return Codec(encode, decode)
```

There is an implementation-dependent limit on the parameter `n` of `Uniform` and on the number of categories for `Categorical`. In the 64-bit rANS implementation which we wrote for our experiments, this limit is 2^{48}. This is not large enough to, for example, cover $\mathcal{S}_n$ for large n, and therefore permutations must be encoded and decoded sequentially, see Appendix E. For details on the implementation of the primitive rANS codecs listed above, see Duda (2009); Townsend (2021).

C. Initial bits

The increase in message length from shuffle coding is equal to the optimal rate in Equation (11). However, the decode step on line 3 of the encode function assumes that there is already some information in the message which can be decoded. At the very beginning of encoding, these 'initial bits' can be generated at random, but they are unavoidably encoded into the message, meaning that for the first object, the discount is not realised. This constant initialization overhead means that the rate, when compressing only one or a few objects, is not optimal, but tends to the optimal rate if more objects are compressed, as the overhead is amortized.

By generalizing the multiset coding method described by Severo et al. (2023a), it is possible to reduce the initial bits needed by shuffle coding from $O(\log n!)$ to $O(\log n)$. This is achieved by the encoder eagerly encoding information during the progressive decoding of the coset. We leave a detailed description of that more sophisticated method to future work and focus here on the simpler version described above. The more sophisticated method which interleaves encoding and decoding steps can be viewed as a special case of the 'bit-swap' method of Kingma et al. (2019).

D. Related work

To date, there has been a significant amount of work on compression of what we refer to as 'ordered' graphs, see Besta & Hoefler (2019) for a comprehensive survey. Compression of 'unordered' graphs, and unordered objects in general, has been less well studied, despite the significant potential benefits of removing order information (see Table 2). The work of Varshney & Goyal (2006) is the earliest we are aware of to discuss the theoretical bounds for compression of sets and multisets, which are unordered strings.

Choi & Szpankowski (2012) discuss the theoretical rate for unordered graphs (a special case of our eq. 11), and present a compression method called 'structural ZIP' (SZIP), which asymptotically achieves the rate

$$\log \frac{1}{P_{\mathrm{ER}}(g)} - n \log n + O(n), \tag{28}$$

where P_{ER} is the Erdős-Rényi $G(n,p)$ model. Compared to our method, SZIP is less flexible in the sense that it only applies to simple graphs (without vertex or edge attributes), and it is not an entropy coding method, thus the model P_{ER} cannot be changed easily. On the other hand, SZIP can achieve good rates on single graphs, whereas, because of the initial bits issue (see Appendix C), our method only achieves an optimal rate on *sequences* of objects. We discuss this issue further and provide a quantitative comparison in Section 4.

Steinruecken (2014; 2015; 2016) provides a range of specialized methods for compression of various ordered and unordered permutable objects, including multisets, permutations, combinations and compositions. Steinruecken's approach is similar to ours in that explicit probabilistic modelling is used, although different methods are devised for each kind of object rather than attempting a unifying treatment as we have done.

Our method can be viewed as a generalization of the framework for multiset compression presented in Severo et al. (2023a), which also used 'bits-back with ANS' (BB-ANS; Townsend et al., 2019; Townsend, 2021). Severo et al. (2023a) use interleaving to reduce the initial bits overhead and achieve an optimal rate when compressing a *single* multiset (which can also be applied to a sequence of multisets), whereas the method presented in this paper is optimal only for sequences of unordered objects (including sequences of multisets). However, as mentioned in Section 1, their method only works for multisets and not for more general unordered objects. Very recently, Severo et al. (2023b) proposed a codec for large ordered graphs, applying the multiset codec with a Pólya urn model to compress the graph's edge set. We use this in some of our experiments below.

There are a number of recent works on deep generative modelling of graphs (see (Zhu et al., 2022) for a survey), which could be applied to entropy coding to improve compression rates. Particularly relevant is Chen et al. (2021), who optimize an evidence lower-bound (ELBO), equivalent to an upper-bound on the rate in Equation (11), when P is not exchangeable. Finally, the 'Partition and Code' (PnC; Bouritsas et al., 2021) method uses neural networks to compress unordered graphs. We compare to PnC empirically in Table 3. PnC is also specialized to graphs, although it does employ probabilistic modelling to some extent.

E. A uniform codec for cosets of a permutation group

Shuffle coding, as described in Section 3, requires that we are able to encode and decode left cosets in $\mathcal{S}_n$ of the automorphism group of a permutable object. In this appendix we describe a codec for cosets of an *arbitrary* permutation group characterized by a list of generators. We first describe the codec, which we call `UniformLCoset`, on a high level and then in Appendices E.1 and E.2, we describe the two main components in more detail.

The target rate for a uniform coset codec is equal to the log of the number of cosets, that is

$$\log \frac{|\mathcal{S}_n|}{|H|} = \log n! - \log|H|. \tag{29}$$

This rate expression hints at an encoding method: to encode a coset we first decode a choice of element of the coset (equivalent to decoding a choice of element of H and then multiplying it by a canonical element of the coset), and then encode that chosen element using a uniform codec on $\mathcal{S}_n$. Note that if the number of cosets is small we could simply encode the index of the coset directly, but in practice this is rarely feasible.

The following is a concrete implementation of a left coset codec:

```
1  def UniformLCoset(grp):                          Effects on l(m):
2    def encode(m, s):
3      s_canon = coset_canon(grp, s)
4      m, t = UniformPermGrp(grp).decode(m)         −log|H|
5      u = s_canon * t
6      m = UniformS(n).encode(m, u)                 +log(n!)
```

```
    return m

  def decode(m):
    m, u = UniformS(n).decode(m)
    s_canon = coset_canon(subgrp, u)
    t = inv(s_canon) * u
    m = UniformPermGrp(grp).encode(m, t)
    return m, s_canon
  return Codec(encode, decode)
```

The codecs `UniformS` and `UniformPermGrp` are described in Appendix E.1 and Appendix E.2 respectively. `UniformS(n)` is a uniform codec over the symmetric group $\mathcal{S}_n$, and `UniformPermGrp` is a uniform codec over elements of a given permutation group, i.e., a subgroup of $\mathcal{S}_n$.

We use a *stabilizer chain*, discussed in more detail in Appendix E.2, which is a computationally convenient representation of a permutation group. A stabilizer chain allows computation of a transversal which can be used to canonize coset elements (line 3 and line 11 in the code above). Routines for constructing and working with stabilizer chains are standard in computational group theory, and are implemented in SymPy (`https://www.sympy.org/`), as well as the GAP system (`https://www.gap-system.org/`), see Holt (2005, Chapter 4) for theory and description of the algorithms. The method we use for `coset_canon` is implemented in the function `MinimalElementCosetStabChain` in the GAP system.

E.1. A uniform codec for permutations in the symmetric group

We use a method for encoding and decoding permutations based on the Fisher-Yates shuffle (Knuth, 1981, 139–140). The following is a Python implementation:

```
def UniformS(n):
  def swap(s, i, j):
    si_old = s[i]
    s[i] = s[j]
    s[j] = si_old

  def encode(m, s):
    p = list(range(n))
    p_inv = list(range(n))
    to_encode = []
    for j in reversed(range(2, n + 1)):
      i = p_inv[x[j - 1]]
      swap(p_inv, p[j - 1], x[j - 1])
      swap(p, i, j - 1)
      to_encode.append(i)

    for j, i in zip(range(2, n + 1), reversed(to_encode)):
      m = Uniform(j).encode(m)
    return m

  def decode(m):
    s = list(range(n))
    for j in reversed(range(2, n + 1)):
      m, i = Uniform(j).decode(m)
      swap(s, i, j - 1)
    return m, s
  return Codec(encode, decode)
```

The decoder closely resembles the usual Fisher-Yates sampling method, and the encoder has been carefully implemented to

exactly invert this process. Both encoder and decoder have time complexity in $O(n)$.

E.2. A uniform codec for permutations in an arbitrary permutation group

For coding permutations from an arbitrary permutation group, we use the following construction, which is a standard tool in computational group theory (see (Seress, 2003; Holt, 2005)):

Definition E.1 (Base, Stabilizer chain). Let $H \leq \mathcal{S}_n$ be a permutation group, and $B = (b_0, \ldots, b_{K-1})$ a list of elements of $[n]$. Let $H_0 \coloneqq H$, and $H_k \coloneqq \mathrm{Stab}_{H_{k-1}}(b_{k-1})$ for $k = 1, \ldots, K$. If H_K is the trivial group containing only the identity, then we say that B is a *base* for H, and the sequence of groups $H_0, \ldots, H_K$ is a *stabilizer chain* of H relative to B.

Bases and stabilizer chains are guaranteed to exist for all permutation groups, and can be efficiently computed using the Schreier-Sims algorithm (Sims, 1970). The algorithm also produces a transversal for the left cosets of each H_{k+1} in H_k for each $k = 0, \ldots, K-1$, in a form known as a Schreier tree (Holt, 2005).

If we define $O_k \coloneqq \mathrm{Orb}_{H_k}(b_k)$, for $k = 0, \ldots, K-1$, then by applying the orbit-stabilizer theorem recursively, we have $|H| = \prod_{k=0}^{K-1} |O_k|$, which gives us a decomposition of the rate that a uniform codec on H should achieve:

$$\log|H| = \sum_{k=0}^{K-1} \log|O_k|. \tag{30}$$

Furthermore, by applying Lemma A.6 recursively, using the transversals produced by Schreier-Sims, we can consruct an explicit bijection between H and the Cartesian product $\prod_{k=0}^{K-1} O_k$. We use this bijection, along with a sequence of uniform codecs on $O_0, \ldots, O_{K-1}$ for coding automorphisms at the optimal rate in Equation (30). For further details refer to the implementation.

F. Further experimental results

F.1. Rate comparison

We compared our compression rate directly to Bouritsas et al. (2021), who used a more sophisticated neural method to compress graphs (upper part of Table 3). They reported results for six of the datasets from the TUDatasets with vertex and edge attributes removed, and for two of the six they reported results which included vertex and edge attributes. Because PnC requires training, it was evaluated on a random test subset of each dataset, whereas shuffle coding was evaluated on entire datasets.

We found that for some types of graphs, such as the bioinformatics and social network graphs, performance was significantly improved by using a Pólya urn (PU) preferential attachment model introduced by Severo et al. (2023b). The urn model, for ordered graphs, treats edges as a set, and we were able to use an inner shuffle codec for sets to encode the edges, demonstrating the straightforward compositionality of shuffle coding. Differently to the original implementation, we apply shuffle coding to the list of edges, resulting in a codec for the multiset of edges. We do not use a tree data structure for the Pólya urn. We also disallow edge redraws and self-loops, leading to an improved rate, as shown in Appendix F.3. This change breaks edge-exchangeability, leading to a ‘stochastic’ codec, meaning that the code length depends on the initial message. Shuffle coding is compatible with such models. In this more general setting, the ordered log-likelihood term in the rate (eq. 11) is replaced with a variational ‘evidence lower bound’ (ELBO). The discount term is unaffected. The derivations in the main text are based on the special case of exchangeable models, where log-likelihoods are exact, for simplicity. They can be generalized with little effort and new insight.

As mentioned in Appendix D, SZIP achieves a good rate for single graphs, whereas shuffle coding is only optimal for sequences of graphs. In the lower part of Table 3, we compare the ‘net rate’, which is the increase in message length from shuffle coding the graphs, assuming some existing data is already encoded into the message. The fact that shuffle coding ‘just works’ with any statistical model for ordered graphs is a major advantage of the method, as demonstrated by the fact that we were easily able to improve on the Erdős-Rényi results by swapping in a recently proposed model.

The total single-thread running time for all of our experiments was 30 hours on a MacBook Pro 2018 with a 2.7GHz Intel Core i7 CPU. Our implementation has not yet been optimized for speed and we are optimistic that the running time of shuffle coding can be significantly improved. One thing that will not be easy to speed up is canonical ordering, since for this we use the `nauty` and `Traces` libraries, which have already been heavily optimized. Fortunately, those calls are currently

Table 3. Comparison between shuffle coding, with Erdős-Rényi (ER) and our Pólya urn (PU) models, and the best results obtained by PnC (Bouritsas et al., 2021) and SZIP (Choi & Szpankowski, 2012) for each dataset. Each SZIP comparison is on a single graph, and thus for shuffle coding we report the *net* compression rate, that is the additional cost of compressing that graph assuming there is already some compressed data to append to. All measurements are in bits per edge.

	Shuffle coding		
Dataset	ER	PU	PnC
Small molecules			
MUTAG	**1.88**	2.66	2.45±0.02
MUTAG (with attributes)	**4.20**	4.97	4.45
PTC_MR	**2.00**	2.53	2.97±0.14
PTC_MR (with attributes)	**4.88**	5.40	6.49±0.54
ZINC_full	**1.82**	2.63	1.99
Bioinformatics			
PROTEINS	3.68	**3.50**	3.51±0.23
Social networks			
IMDB-BINARY	2.06	1.50	**0.54**
IMDB-MULTI	1.52	1.14	**0.38**

	Shuffle coding (net)		
Dataset	ER	PU	SZIP
Airports (USAir97)	5.09	**2.90**	3.81
Protein interaction (YeastS)	6.84	**5.70**	7.05
Collaboration (geom)	8.30	**4.41**	5.28
Collaboration (Erdos)	7.00	**4.37**	5.08
Genetic interaction (homo)	8.22	**6.77**	8.49
Internet (as)	8.37	**4.47**	5.75

only 10 percent of the overall time, and we believe there is significant scope for optimization of the rest. We report the time required for canonical ordering for each dataset in the next subsection.

We have published data used to evaluate SZIP (Choi & Szpankowski, 2012) with the authors' consent. This and more detailed results are available in the code repository `https://github.com/entropy-coding/shuffle-coding`.

F.2. Runtimes

We show runtimes of our experiments in Table 4. These runtimes include time needed for gathering dataset statistics and parameter coding. The results show that for our implementation, only a small fraction of runtime is spent on finding automorphism groups and canonical orderings with `nauty`.

F.3. Model ablations

We present results of additional ablation experiments on the PnC datasets in Table 5. We do an ablation that uses a uniform distribution for vertex and edge attributes with an Erdős-Rényi model (unif. ER). There is a clear advantage to coding maximum-likelihood categorical parameters (ER), justifying it as the approach used throughout this paper. We also show the rates obtained by the original method proposed in Severo et al. (2023b) (PU redr.), demonstrating a clear rate advantage of our approach disallowing edge redraws and self-loops (PU) in the model.

Unlike PnC, we do not rely on compute-intense learning or hyper-parameter tuning, and instead rely on very simple empirical statistics. We would expect to see improved rates for shuffle coding when combined with neural models.

Table 4. Total runtimes, in seconds, for `nauty` calls determining the canonical ordering and generators of the automorphism group of all graphs, and encoding and decoding with shuffle coding on the Erdős-Rényi (ER) and Pólya urn (PU) models, for all previously reported TU and SZIP datasets.

		ER		PU	
Dataset	nauty	encode	decode	encode	decode
TU by type					
Small molecules	172	1211	1160	–	–
Bioinformatics	≤ 1	24	19	–	–
Computer vision	≤ 1	18	16	–	–
Social networks	94	38241	35987	–	–
Synthetic	1	19	17	–	–
Small molecules					
MUTAG	≤ 1	≤ 1	≤ 1	≤ 1	≤ 1
MUTAG (with attributes)	≤ 1	≤ 1	≤ 1	≤ 1	≤ 1
PTC_MR	≤ 1	≤ 1	≤ 1	≤ 1	≤ 1
PTC_MR (with attributes)	≤ 1	≤ 1	≤ 1	≤ 1	≤ 1
ZINC_full	4	35	35	74	80
Bioinformatics					
PROTEINS	≤ 1	1	1	2	2
Social networks					
IMDB-BINARY	≤ 1	2	2	3	3
IMDB-MULTI	≤ 1	2	2	4	4
SZIP					
Airports (USAir97)	≤ 1	≤ 1	≤ 1	≤ 1	≤ 1
Protein interaction (YeastS)	≤ 1	3	4	7	10
Collaboration (geom)	12	7268	6527	5882	5868
Collaboration (Erdos)	111	747	748	762	766
Genetic interaction (homo)	18	216	252	331	275
Internet (as)	4987	38670	34507	37156	38432

G. Parameter coding details

All bit rates reported for our experiments include model parameters. Once per dataset, we code the following lists of natural numbers by coding both the list length and the bit count $\lceil \log m \rceil$ of the maximum element m with a codec respecting a discretized log-uniform distribution, as well as each element of the list using the same log-uniform codec over [0, 2^m]:

- A list resulting from sorting the graphs' numbers of vertices, and applying run-length coding, encoding run lengths and differences between consecutive numbers of vertices.
- For datasets with vertex attributes: a list of all vertex attribute counts within a dataset.
- For datasets with edge attributes: a list of all edge attribute counts within a dataset.
- For Erdős-Rényi models: a list consisting of the following two numbers: the total number of edges in all graphs, and the number of vertex pairs that do not share an edge.

Coding these empirical count parameters allows coding the data according to maximum likelihood categorical distributions. For Pólya urn models, we additionally code the edge count for each graph using a uniform codec over $[0, \frac{1}{2}n(n-1)]$, exploiting the fact that the vertex count n is already coded as described above. For each dataset, we use a single bit to code whether or not self-loops are present and adapt the codec accordingly.

Table 5. Model ablations compared to PnC.

Dataset	Shuffle coding				PnC
	unif. ER	ER	PU	PU redr.	
Small molecules					
MUTAG	–	**1.88**	2.66	2.81	2.45±0.02
MUTAG (with attributes)	6.37	**4.20**	4.97	5.13	4.45
PTC_MR	–	**2.00**	2.53	2.74	2.97±0.14
PTC_MR (with attributes)	8.04	**4.88**	5.40	5.61	6.49±0.54
ZINC_full	–	**1.82**	2.63	2.75	1.99
Bioinformatics					
PROTEINS	–	3.68	**3.50**	3.62	3.51±0.23
Social networks					
IMDB-BINARY	–	2.06	1.50	2.36	**0.54**
IMDB-MULTI	–	1.52	1.14	2.17	**0.38**