Reinforcement learning and the Andrews–Curtis conjecture

Justin Tan

Department of Computer Science & Technology, University of Cambridge, Cambridge CB3 0FD, UK

E-mail: jt796@cam.ac.uk

Abstract: We present results on the application of reinforcement learning to finding trivialisation paths of balanced group presentations in the context of the Andrews–Curtis conjecture. We implement the environment logic completely in Jax and train an actor–critic agent with a transformer–based neural architecture via proximal policy optimisation using curriculum learning. The resultant agent performs strongly on a suite of evaluation measures.

1 Environment formulation

To begin, we implement the environment logic completely using Jax primitives. Jax [1] is essentially a functional metaprogramming language for composable function transformations which enables us to vectorise the environment and transition logic entirely on the accelerator in a fully synchronous fashion. This elides the overhead of data communication between the CPU and accelerator and enables the transition logic to be fully `jit`–compiled, enabling significant computational speedups via operator fusion. We represent a balanced presentation $\langle x_1, \ldots, x_n \,|\, r_1, \ldots, r_n \rangle$ as a sequence of length $n|r|$, where $|r|$ is the maximum relator length, which we set to 64. Following the logic in [2], each relator is represented by a sequence of generators. We tokenise the generators (x_i, x_i^{-1}) as integers `(i,-i)`, respectively, and learn a common vector embedding of each generator, trained end–to–end with the agent model parameters. For a balanced presentation with n relators, note the action space may be divided into $2(n-1)+2n$ elementary operations acting on each relator, $(1) : r_i \mapsto r_i \cdot r_j^{-1}$, $(2) : r_i \mapsto r_i \cdot r_j$, $(3) : r_i \mapsto g r_i g^{-1}$, $(4) : r_i \mapsto g^{-1} r_i g$, $i \neq j$. This will inform our subsequent architecture choice. In Figure 1, we benchmark our Jax implementation on alongside the baseline PPO agent and codebase open–sourced in [2], using an identical architecture and hyperparameter settings. Vectorisation of the environment means that Jax is around 7 times faster with the same number of parallel environments. We experimented extensively with the reward structure; we had some success with a dense reward function which rewards the agent based on the presentation length reduction ΔL achieved per–frame,

$$R(s_t, a_t . s_{t+1}) = -\mathbf{1}[s_{t+1} \neq \mathsf{terminal}] \cdot \Delta L + \mathbf{1}[s_{t+1} = \mathsf{terminal}] \cdot M \ , \tag{1.1}$$

where $M = 10^3$ is the maximum achievable reward by the agent, awarded upon episode completion. It was difficult to find a handcrafted dense reward structure which outperforms the sparse near–binary reward structure proposed in [2] with any degree of statistical significance; a reward model based on latent space distances between learned vector embeddings of the trivial and current presentation may be better suited to this task.

2 Agent architecture

We use the PPO algorithm [3] to train our agent, and implement this in Jax. The dense network proposed in [2] provides a strong baseline but does not make use of positional information inherent in the representation of a presentation. We experimented with multiple sequence processing architectures, and converged on a transformer–based architecture. This has two important features — firstly, it preserves the permutation equivariance of the relators defining the group presentation. Secondly, it is agnostic to the sequence length and thus able to process arbitrary–length presentations. Our architecture is divided into two stages;

1. The **relator encoder** accepts a single tokenised relator, maps it to a sequence of learned embeddings, and prepends a summary token to the sequence. Next, we apply two standard self–attention transformer blocks to the relator. This stage outputs the summary token of the resultant sequence as the processed representation of the relator. This module is vectorised over the relator dimension to obtain an intermediate sequence representation of shape `(n_relators, embedding_dim)`.

2. The **presentation encoder** accepts the intermediate sequence above and applies two standard transformer blocks to the sequence of processed relators to construct an embedding of the presentation informed by the relationships between the processed relators.

This forms the shared backbone of the network. To preserve permutation equivariance, the actor head is applied to each relator embedding in the output sequence independently to produce logits specific to each relator. These are then flattened to form the distribution over all possible actions π_θ. The critic head is applied to the flattened output sequence to estimate state value. Due to parameter–sharing across the sequence, the transformer–based architecture weighs in at less than 20% of the size of the dense network ($\sim 3 \times 10^5$ versus $\sim 1.7 \times 10^6$ parameters), meaning the entire training loop may be efficiently vectorised in Jax for large–scale hyperparameter searches. The computational bottleneck here is the quadratic $\mathcal{O}(|r|^2)$ time complexity in the maximum relator length incurred by the first processing stage. We compare the performance of various agent architectures in Figure 2.

3 Dataset and curriculum

We train our agent on a dataset of presentations randomly scrambled from the trivial set of size 2.6×10^6. The dataset is approximately equally divided into three tiers of difficulty demarcated by different uniformly distributed scramble depths $\{\mathsf{Easy} : 2 \to 7,\ \mathsf{Medium}\!: 8 \to 13,\ \mathsf{Hard}\!: 14 \to 20\}$. We use a maximum relator length of 64 to accommodate the higher scramble depths. We also randomly inject elements from the general Miller–Schupp series (4.1), parameterised by the words w, which we randomly sample with length given by half the scramble depth for each tier at a rate of 5%, 20% and 30% for each respective tier.

The trivialisation path for any presentation consists of a sequence of atomic simplification operations. To ensure our agent learns atomic operations before learning how to effectively compose them, we apply a curriculum consisting of three regimes of difficulty. The easiest regime yields batches divided between the Easy/Medium/Hard tiers in the ratios $(0.75, 0.20, 0.05)$. The corresponding ratios in the most challenging regime are $(0.30, 0.35, 0.35)$, and we linearly interpolate between each regime using an interpolant λ which increments if the current agent's weighted solve rate rises above 0.7, and decrements if it falls below 0.5. We train our agent for 8×10^8 total environment steps using a horizon length of 256 moves.

4 Agent evaluation

We evaluate our transformer–based agent trained via curriculum learning using stochastic top–k sampling from the learned policy, where we fix $k = 32$ unless otherwise stated. Curiously, both Monte Carlo tree search and beam search (of width k) using the value function as the search heuristic slightly underperform top-k sampling. In other words, the value function could not be used to reliably prune uninteresting regions of the state space for the problem at hand, suggesting that the critic trained using the sparse reward function remains suboptimal.

4.1 Generalisation on random scrambling

We generate a held–out set of 30,000 presentations randomly scrambled from the trivial set up to depth 18, drawn from the same distribution as the training set. Our agent trivialises nearly all such presentations, and we exhibit the results in Table 1 and Figure 3.

4.2 Results on special series

On the Akbulut–Kirby series $\mathsf{AK}(n) := \langle x, y \,|\, x^n = y^{n+1}, xyx = yxy\rangle$, our agent is able to trivialise up to $n = 2$ but is unable to trivialise beyond $n = 3$, and is unable to reduce the length of the presentation using elementary AC moves for $n \geq 3$. We find the following trivialisation paths for the first two members of the series, of length 5 and 13 respectively. Note we freely and cyclically reduce the relevant relator after the application of each action.

- $\mathsf{AK}(1)$: $a_7 \to a_6 \to a_7 \to a_0 \to a_0$.
- $\mathsf{AK}(2)$: $a_9 \to a_3 \to a_3 \to a_{11} \to a_9 \to a_7 \to a_0 \to a_3 \to a_1 \to a_5 \to a_0 \to a_6 \to a_0$.

The particular instance of the Miller–Schupp series we consider is defined as $\mathsf{MS}(n) := \langle x, y \,|\, x^{-1}y^n x = y^{n+1}, x^2 y = yx\rangle$. The presentations are known to be AC–trivial $\forall\, n$. Our agent successfully trivialises members of this family up to and including $n = 6$. The respective trivialisation path lengths, ordered by n, are $(5, 11, 25, 52, 111, 234)$. It is likely we would need to increase the horizon length beyond the current value of 256 to reliably trivialise elements of this series for $n \geq 7$. Figure 4 exhibits the behaviour of our agent over the evaluation rollout on a selection of special series examples.

4.3 General Miller–Schupp series

Lastly, we consider the dataset of general Miller–Schupp series of potential counterexamples generated in [2], parameterised by the word w and unobserved by the agent during training,

$$\mathsf{MS}(n, w) := \langle x, y \,|\, x^{-1}y^n x = y^{n+1}, x = w\rangle, \quad n \geq 1 \ . \tag{4.1}$$

Here w is a word in x, y with zero exponent sum on x and we consider members of this series with $n \leq 7$ and $|w| \leq 7$. The agent is able to trivialise 565/1190 such presentations, exceeding the performance of the greedy search and PPO agents presented in [2].

A Jax/Torch benchmarks

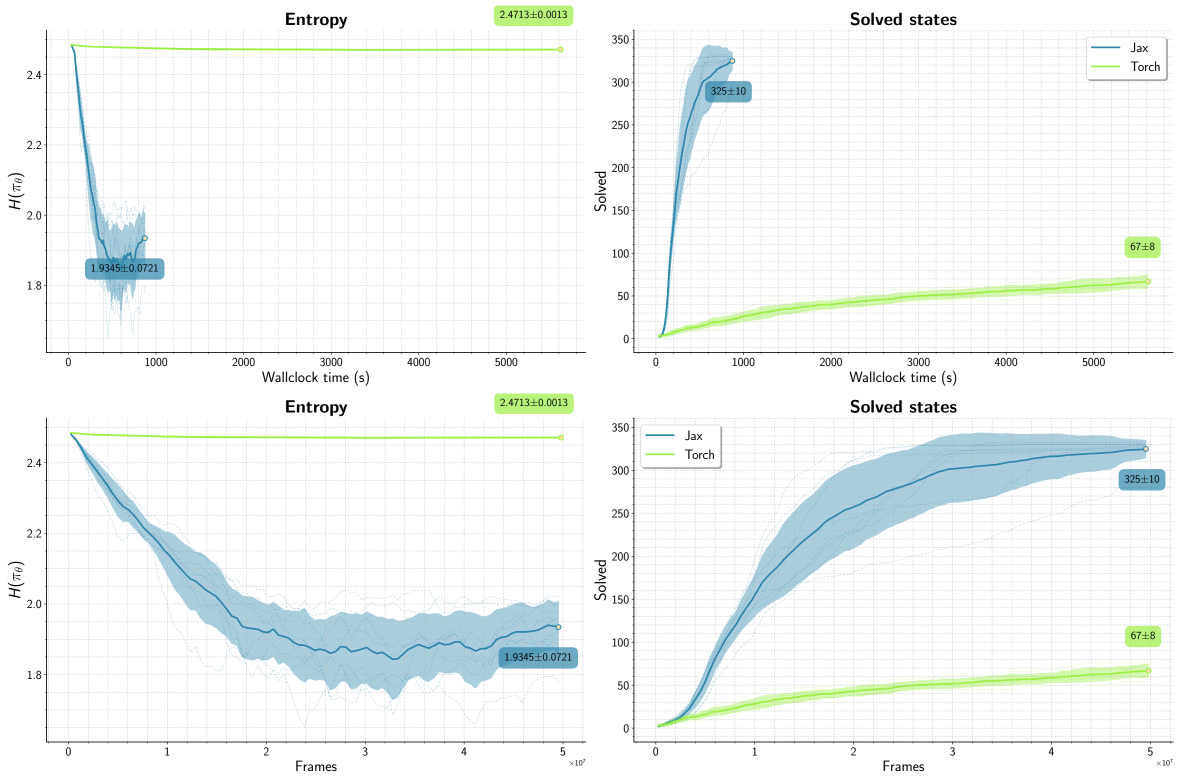

Figure 1: Benchmarking of the Jax implementation against the Torch implementation open–sourced in [2]. We follow the training procedure of the PPO agent on the dataset composed of 1190 elements of the Miller–Schupp series described in [2]. **Top row**: evolution of the entropy of the policy and number of solved states against wallclock time. **Bottom row**: the same quantities plotted against the number of frames encountered by the agent thus far. We use an identical architecture and hyperparameter settings, and train for 5×10^7 frames on a single NVIDIA RTX A5000 card. Vectorisation of the environment logic means the Jax implementation is around 7 times faster than Torch with the same number of parallel environments; with greater computational speedups possible with further environment vectorisation.

B Model comparison

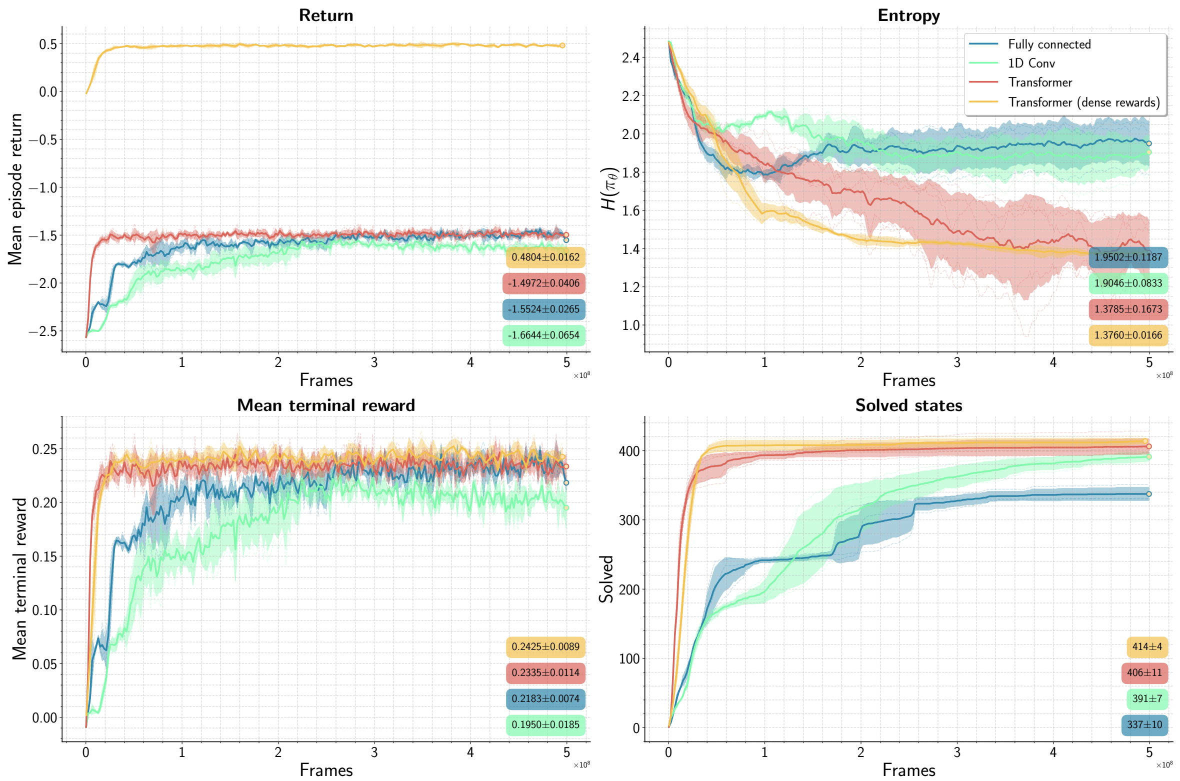

Figure 2: Trajectories of quantities of interest during optimisation on the Miller–Schupp dataset for different actor–critic architectures with identical non–architectural hyperparameters. 'Return' denotes the average return achieved for completed states over the full rollout. The **fully connected** architecture consists of independent actor–critic networks composed of 2–layer fully connected networks with 512 units per layer. The **1D Conv** architecture consists of a shared 3–layer 1D convolutional encoder with channel dimensions $(32, 64, 32)$ and kernel sizes $(32, 16, 16)$, followed by independent actor–critic heads composed of 2–layer fully connected networks with 64 units per layer. The **Transformer** architecture is outlined in Section 2, and **Transformer (dense rewards)** denotes the transformer trained with the dense reward function (1.1). We did not extensively experiment with the architectural hyperparameters for each model.

C Model evaluation

Tier	Depth	Solved	Mean est. value	Mean path length (σ)
Easy	$2 \to 7$	1.0	0.97 ± 0.07	$4\,(2)$
Medium	$8 \to 12$	0.9996	0.85 ± 0.19	$12\,(9)$
Hard	$13 \to 18$	0.9989	0.79 ± 0.18	$15\,(14)$

Table 1: Evaluation dataset statistics on a held–out dataset consisting of random scrambling from the trivial presentation, consisting of 10^4 examples per tier. We observe the agent is consistently able to recover the trivial presentation from sequences of randomly applied AC moves up to depth 18. **Mean estimated value** refers to the average of the value function evaluated for the initial presentation $V(s_0)$, **Mean path length** refers to the mean trivialisation path length, which forms a very heavy–tailed distribution.

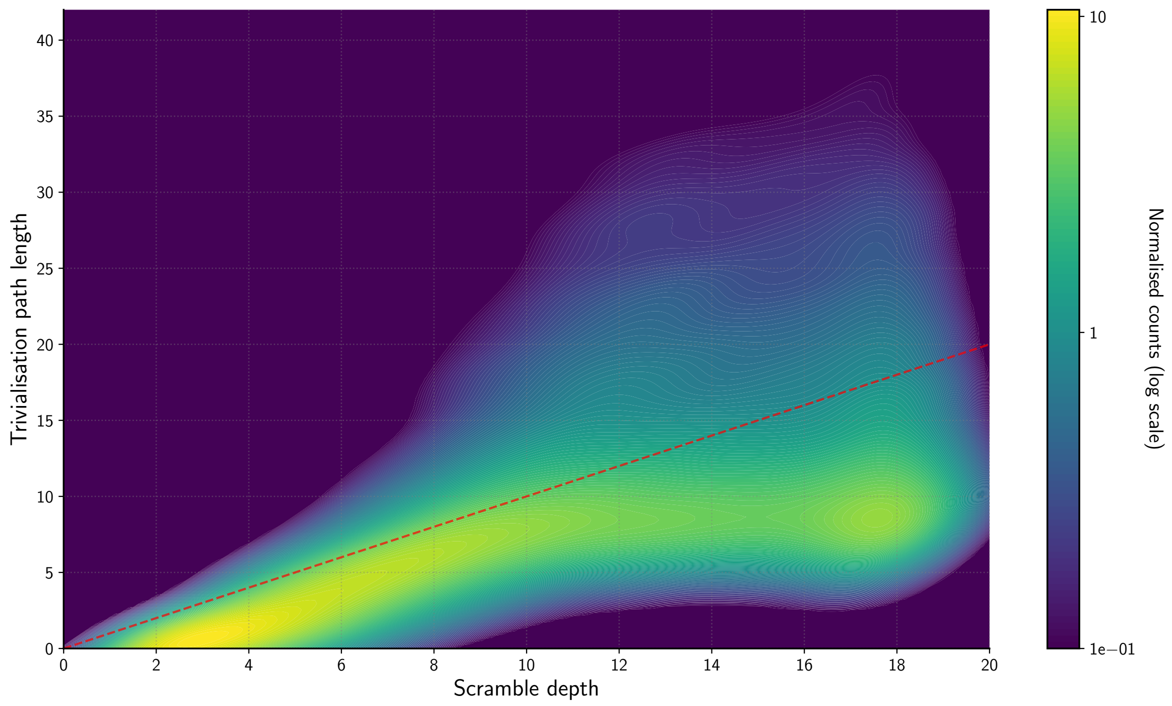

Figure 3: Heatmap of scramble depth versus length of trivialisation path on the evaluation dataset. The trained agent tends to find trivialisation paths shorter than the scramble depth, but takes substantially more steps than necessary on a heavy tail of ‘difficult’ presentations. This indicates it finds reasonable heuristics which are more efficient than the inverse of the scrambling process, but break down on the heavy tail.

D Agent behaviour on special series

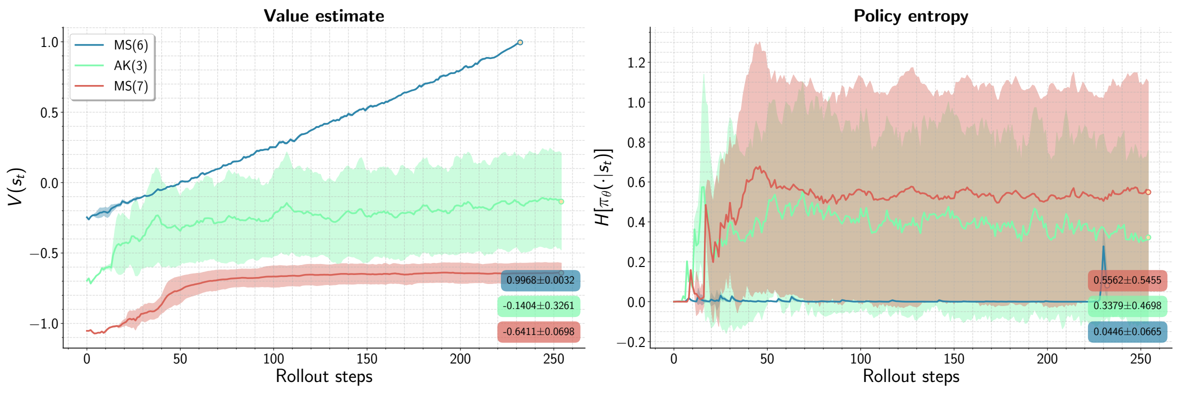

Figure 4: Behaviour of the agent evaluated on the special series instances AK(3) (unsolved), MS(6) (solved in 234 steps) and MS(7) (unsolved). The shaded area indicates the $\pm 2\sigma$ confidence region for a stochastic policy sampled 32 times. **Left**: value estimate $V(s_t)$ of the current state in the evaluation rollout by the critic network. **Right**: entropy of the policy π_θ of the current state in evaluation rollout. For all solved instances the value estimate of the current state increases monotonically to 1 over the rollout, and the policy entropy is negligible, in stark contrast to the near random–walk behaviour of unsolved instances.

References

[1] J. Bradbury, R. Frostig, *et al.*, "JAX: composable transformations of Python+NumPy programs." 2018. http://github.com/jax-ml/jax.

[2] A. Shehper, A. M. Medina-Mardones, L. Fagan, B. Lewandowski, A. Gruen, Y. Qiu, P. Kucharski, Z. Wang, S. Gukov, *et al.*, "What makes math problems hard for reinforcement learning: a case study," *arXiv preprint arXiv:2408.15332* (2024) .

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347* (2017) .