LeanSolver: Solving theorems through Large Language Models and Search

Improving Theorem Proving with Proof Assistants and Sequential Monte Carlo in Large Language Models

Avi Luciano Halevy

LeanSolver: Solving theorems through Large Language Models and Search

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

APPLIED MATHEMATICS

by

Avi Luciano Halevy
born in Vlaardingen, the Netherlands

Department of Applied Mathematics
Faculty of Electrical Engineering, Math and Computer Science
Delft University of Technology
Delft, the Netherlands

LeanSolver: Solving theorems through Large Language Models and Search

Author: Avi Luciano Halevy

Abstract

We consider a subset of simple proving exercises that are part of the Lean 4 tutorials. The exercises consist of a statement, and the task will consist of creating a proof term of the desired type through the use of tactics. Large Language Models on their own are known to be able to produce syntactically correct pieces of text but are unable to come up with semantically correct pieces of text once very similar solutions are lacking in the dataset it was trained on. This work reviews the Sequential Monte Carlo with Expectation-Maximization (SMX) algorithm for general reinforcement learning problems, and applies it to theorem proving using LLMs. This work shows that the SMX algorithm is applicable to tasks where formal verification can be performed on the output, allowing the calculated reward to steer the reasoning process. We furthermore formally prove a theorem that will be important in showing that the resampling step in SMX is necessary to mitigate sample impoverishment. The theoretical part of this thesis builds on the theory of the SMX paper by verifying the derivation of the E-step, starting from the Evidence Lower Bound (ELBO). This derivation was missing from the original SMX paper.

Thesis Committee:

Chair:	Dr. Ir. G.F. Nane, Faculty EEMCS, TU Delft
University supervisor:	Dr. S. Dumančić, Faculty EEMCS, TU Delft
Committee Member:	Dr. B.P.Ahrens, Faculty EEMCS, TU Delft

Contents

List of Figures

Chapter 1

Introduction

Consistency and critical thinking are essential parts of human cognition, which we often rely on for complex tasks. As large language models are increasingly used for tasks designed for humans, their inability to replicate these qualities becomes a significant limitation. This is most likely due to the way they are trained and the way inference is done on these models [25]. This master thesis focuses on improving both the training method and inference methods for large language models.

The current training approach, known as autoregressive language modeling [4], involves predicting the next wordpiece (called a token) in a sequence given previous ones. This thesis proposes an enhancement inspired by expert iteration and reinforcement learning, where a reward function evaluates how relevant each step is. Steps are furthermore only considered if they can be validated by a proof assistant. This makes sure the model generates logically sound and verifiable reasoning.

While prior methods such as RLHF [29] have been used to align outputs with human preferences, they do not enforce logical consistency. Alternative methods, such as neuro-symbolic reasoning in Pan et al. [30], attempt to integrate structured reasoning into neural models, but these often require separate symbolic components to which the solving gets delegated to solvers. This thesis proposes an extension of these efforts by integrating proof validation at both the training and inference stages.

For inference, large language models generate text one token at a time by sampling from a probability distribution over possible next tokens. This distribution is determined by the predictions of the model at each step, which assigns probabilities to every token in the vocabulary based on the preceding context (i.e., the text provided to the model up to that point). Sampling methods such as top-p [18] and temperature scaling [38] are then applied to guide token selection. Top-p considers tokens until their cumulative probability exceeds a threshold, and temperature adjusts the sharpness of the probability distribution to control diversity.

Existing LLM inference techniques rely on token-by-token sampling strategies, often supplemented by heuristic search methods such as Monte Carlo Tree search [33, 34, 7], other forms of tree search [23] or chain of thought [51]. However, these require computationally expensive expansions or constraining reasoning to a single linear chain, limiting their ability to explore multiple reasoning paths efficiently at the same time. Instead, this

thesis proposes using Sequential Monte Carlo, which allows efficient exploration paths, prioritizing paths based on logical validity and relevance.

This work introduces a method combining Sequential Monte Carlo (SMC) with these sampling strategies, drawing inspiration from Macfarlane et al. [27]. SMC is a probabilistic inference method that uses a set of weighted samples, called particles, to approximate a target distribution. The particles are iteratively updated through importance sampling and resampling steps to focus computational resources on high-probability regions of the distribution. The approach uses SMC to promote running different reasoning paths in parallel during inference, prioritizing paths based on validity and relevance. One domain where reasoning correctness is critical, and particularly suited to validation, is formal theorem proving. In traditional mathematics, a theorem is considered proven when its reasoning is detailed enough to convince fellow mathematicians of its validity. Formal proofs take this a step further, by working out the proof completely until it is reduced to axioms and universally accepted rules applied to these axioms. This level of rigor allows the proof to be verified by a computer using a proof assistant. To achieve this, the proof must be translated into the formal language of the proof assistant, which then checks whether the reasoning logically follows from the axioms. The proof assistant that will be used in this thesis is called the LEAN4 proof assistant, the foundation and inner workings of which will be described later on in this thesis.

To illustrate how formal theorem proving works, consider the following logical statement: If it is not the case that for all $x, p(x)$ is false, then there must exist some x for which $p(x)$ is true.

$$\neg(\forall x, \neg p(x)) \rightarrow (\exists x, p(x))$$

In a formal proof, this must be verified step-by-step using logical rules. This would be proven by contradiction, as follows: Assume the negation of what we want to prove: suppose that $\neg(\exists x, p(x))$ holds. Yet $\neg(\forall x, \neg p(x))$ is also true, since this is also something we assume in the statement.

By definition of negation and the existential quantifier, saying $\neg(\exists x, p(x))$ means that for every x, $p(x)$ must be false, i.e.

$$\neg(\exists x, p(x)) \iff \forall x, \neg p(x)$$

We only go from the left to the right statement so that direction is the only direction we have to establish. We introduce an arbitrary x and assume $p(x)$ holds. However, this leads to a contradiction since there does not exist an x for which $p(x)$ holds ($\neg(\exists x, p(x))$). Therefore, for arbitrary x we know $p(x)$ does not hold, i.e. $\forall x, \neg p(x)$.

Now, we have established $\forall x, \neg p(x)$, but out of our assumptions we have $\neg(\forall x, \neg p(x))$, thus we have derived the desired contradiction. Thus we obtain the proof of the logical statement $\neg(\forall x, \neg p(x)) \rightarrow (\exists x, p(x))$.

Recent research has explored integrating LLMs into theorem proving via heuristic search methods [33, 34, 23]. While these approaches show promising results because it does inference and training using tree search-based methods they are more computationally demanding, due to expansion during inference and backpropagation during training, as already mentioned, our proposed method mitigates this. However, concurrent work Xin et al. [52]

significantly improved state of the art, in part due to optimizations in proof search and tactic handling. Notably, they allow the model to generate proofs freely, truncating at the nearest error message. Additionally, instead of treating actions as individual tactics, they frame them as code continuations and later decompose them into specific tactics using a modified Lean REPL. We will discuss these implementation choices in more detail in the methods section and the discussion.

Besides implementing a different training and inference technique for large language models, we will formally prove Karp's theorem, which will be useful in proving that the resampling step in particle filtering is necessary to mitigate two issues: weight degeneracy, where all weight concentrates on a single particle, and progressive loss of diversity, where fewer unique particle lineages persist over multiple iterations, reducing the effectiveness of the approximation.

Beyond taking inspiration for implementing a different inference technique based on Macfarlane et al. [27], this thesis also builds on its theory, which combines Expectation-Maximization (EM) with Sequential Monte Carlo (SMC) sampling. The Expectation Maximization algorithm is an iterative optimization technique designed for problems where directly optimizing all parameters simultaneously is difficult or intractable. EM addresses this by breaking the problem into two interdependent steps: the E-step (Expectation step) and the M-step (Maximization step).

Specifically, we provide a derivation in the E-step from the Expectation Maximization from the Expected Lower Bound (ELBO). While Macfarlane et al. [27] presents an E-step formulation, their work does not give the derivation of it, leaving some details and assumptions unexplained. This thesis addresses this by presenting a complete derivation of the E-step, making the assumptions on the theory and notational simplifications more explicit.

1.1 Outline & research questions

The structure of this thesis is as follows. Chapters 2 and 3 review background information on Large Language Models and the LEAN4 proof assistant, and programming language, respectively. Then in Chapter 4 we describe Karp's theorem and its application to this thesis. In the SMX chapter (Chapter 5), we go over the theory introduced by Macfarlane et al. [27], which combines Expectation Maximization and Sequential Monte Carlo, and rederive the E-step, the derivation itself is described in Appendix B.1. Related work on improving Large Language Model reasoning through external verification is discussed in Chapter 6. The methodology of this thesis is described in Chapter 7, where we attempt to answer the following main research question followed by the results in Chapter 8. Finally, Chapter 9 draws conclusions based on the results and provides recommendations for future research.

The main research question we attempt to answer in this thesis is:

> Will a hybrid approach with an LLM and a symbolic
> solver improve in reasoning performance compared to only using the LLM?

We focus on the SMX algorithm and the proof assistant as our symbolic solver. Concretely, the research questions investigated in this thesis are:

1. How does supervised fine-tuning influence computational efficiency and accuracy in formal theorem proving using the SMX algorithm?
2. How does the number of particles in the SMX method affect accuracy on the test set and the efficiency across the problems in the test set?
3. What is the relationship between temperature, top-p, and their combined effect on efficiency and accuracy in the SMX method?

Chapter 2

Large Language Models

In this chapter we will go into detail about how standard transformer-based large language models work, focusing on inference techniques and fine-tuning methods. Before going into these specifics though, we will give an introduction into how large language models work.

There are various types of large language models, such as mixture of expert models and state space models. However, most popular models today are transformer-based. These transformer-based models are typically decoder-only and pretrained in an autoregressive manner. One of the most well-known examples is the Generative Pretrained Transformer (GPT) model. In the sections below we will look specifically at the LLaMA-3 architecture, whenever we go into detail on how large language models work internally.

The LLaMA-3 is the third iteration of improvements on the decoder-only large language models developed by Meta. The architecture improved the computational efficiency compared to the original transformer architecture. It furthermore improved the performance by modifying the three key processing stages of the transformer architecture. Before these can be discussed in detail, it is necessary to first introduce these fundamental processing stages common to decoder-only transformer models.

When a decoder-only transformer processes text, the computation can typically be broken down into the following stages:

- Tokenization: Converting text into a sequence of tokens (sub-word units)
- Positional Encoding: Adding positional information to the tokens to account for the order of words.
- Transformer Decoding: Applying the layers of the transformer decoder to generate predictions, to be processed in the next layer.

All three components can vary significantly across decoder-only transformer models. We will look into each component in detail. In the last Transformer Decoding layer the predictions will be used as logits each entry corresponding to a different wordpiece/token. This will be normalized, giving a probability distribution over the next possible tokens.

2.1 Tokenization

In this stage, the text gets broken down into pieces called tokens. An example of this is illustrated in the figure below.

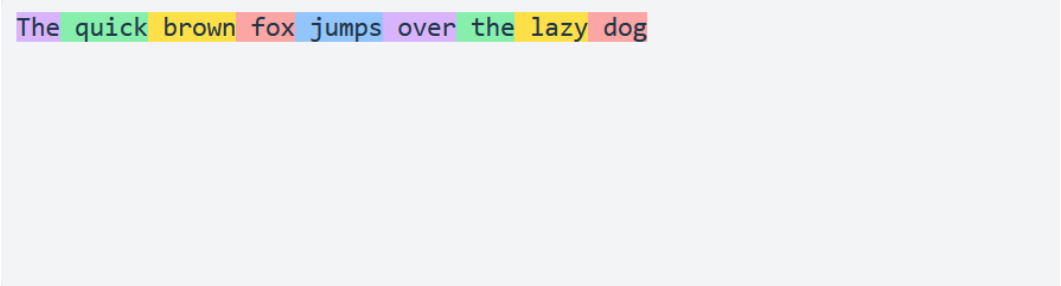

Figure 2.1: Illustration of tokenization, adapted from the Tokenizer Playground available on Hugging Face Spaces (Xenova).

Figure 2.1 shows the sentence "The quick brown fox jumps over the lazy dog" tokenized into individual tokens. Each token represents a sub-word or word unit, with different colors highlighting the separation between tokens. Tokenization breaks down the piece of text into smaller, more manageable units for the language model to process.

Each of these subwords known as tokens is mapped to an integer. In the case of the abovementioned sentence, this would be

$$[791, 4062, 14198, 39935, 35308, 927, 279, 16053, 5679]$$

Each number would then indexes into a large list of vectors that represent that token.

This collection of wordpieces is called a vocabulary. The way this collection is created in the case of llama3 is through using byte pair encoding [11], an example of byte pair encoding is given in 2.2. Byte pair encoding (bpe) is a way to better handle uncommon words more effectively. It works by iteratively merging the most frequent token pairs to create a vocabulary of subwords. In this way it is not necessary to give every word a different token. Only the most frequent ones receive a token and the rest get decomposed into subwords giving us a balance between coverage and efficiency. In the next section, we will illustrate how Byte pair encoding (bpe) works with a small example.

2.2 Basic example byte-pair encoding

To create a byte pair encoding we start by selecting a corpus of text to apply the encoding to. Let us for this example pick the sentence:

“a fox is not caught twice with the same snare."

This then gets split into individual characters including the spaces and punctuation. Each of these characters is then assigned a token ID. We then count the token pairs, also known as bigrams. For our example sentence, we get the following bigrams:

$$
\begin{array}{c}
('a',' ') \\
(' ',' f') \\
('f','o') \\
\vdots \\
('e','.')
\end{array}
$$

The most frequent 7 pairs are:

- $('e',' ')$, with frequency 3
- $(' ',' s')$, with frequency 2
- $(' ',' t')$, with frequency 2
- $('t',' ')$, with frequency 2
- $('t','h')$, with frequency 2
- $('w','i')$, with frequency 2
- $(' ',' f')$, with frequency 1

The most frequent pair is then added as another subword. In this case the pair "e " would be merged into a single token. This new subword gets added to the vocabulary, which initially consists of only individual characters, and is assigned its own token ID.

We then repeat this process: counting the frequency of token pairs (bigrams), merging the most frequent one, and updating the vocabulary with the newly created subword. This cycle continues until no more pairs can be merged or until a predetermined vocabulary size is reached.

2.3 Positional encoding

In the upcoming sections, we will see that the transformer decoder is naturally permutation-invariant, meaning it processes tokens without considering their position in the sequence. However, for natural language processing the order of tokens is critical for understanding meaning.

For example, consider the sentences

The dog bit the man

and

The man bit the dog

Despite containing the same words, the change in order changes the meaning of the sentence. To address this, the original transformer paper [50] proposed adding a small position-dependent value to each vector after the tokens are converted into embedding vectors. This value would vary depending on where the vector lies within the sequence. This allows the transformer decoder to learn to account for the ordering of the input tokens.

While the original transformer uses additive positional encoding, in the case of LLaMA-3.2 the positional encoding is multiplicative. The positional encoding used is called RoPE-embeddings, introduced in the RoFormer paper by Su et al. [40]. The idea is that the relative distance between tokens affects how their representations interact in the attention mechanism. We will explain what this means in more detail later in this section. RoPE stands for Rotary Position Embedding. This is because each embedding vector is effectively matrix-vector multiplied with a rotation matrix:

$$f_{\{q,k\}}(x_m, m) = R(\Theta, m)W_{\{q,k\}}x_m,$$

where $x_m \in \mathbb{R}^d$ is the embedding vector, where d is the number of entries of the embedding vector. And

$$R(\Theta, m) = \begin{pmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & 0 & 0 & \dots & 0 & 0 \\ \sin(m\theta_1) & \cos(m\theta_1) & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \cos(m\theta_2) & -\sin(m\theta_2) & \dots & 0 & 0 \\ 0 & 0 & \sin(m\theta_2) & \cos(m\theta_2) & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \cos(m\theta_{d/2}) & -\sin(m\theta_{d/2}) \\ 0 & 0 & 0 & 0 & \dots & \sin(m\theta_{d/2}) & \cos(m\theta_{d/2}) \end{pmatrix},$$

with $\Theta = \left\{\theta_i = 10000^{-2(i-1)/d},\ i \in [1, 2, \dots, d/2]\right\}$ [40]. Lastly $W_{\{q,k\}}$ is a weight matrix for either transforming the embedding to queries or keys. We will discuss the role of queries and keys in more detail in upcoming sections, but for now, the important point is that embedding vectors that are nearby will undergo less rotation with respect to each other compared to vectors of tokens that are farther apart.

After applying $W_{\{q,k\}}$ to project the input into query and key spaces, the resulting vectors are rotated by $R(\Theta, m)$. This rotation affects the inner product between the query and key vectors, which in turn will have an impact on how attention is computed. We will discuss the attention layer in the next section.

2.4 Transformer Decoder

The majority of the computation happens in this step. The transformer decoder is the decoder part of the transformer neural network architecture [50], which consists of the transformer encoder and the transformer decoder. The transformer architecture is schematically shown in Figure 2.2.

Aside from the number of layers most of the core building blocks of the transformer model have remained largely unchanged. In tasks like translation and question answering,

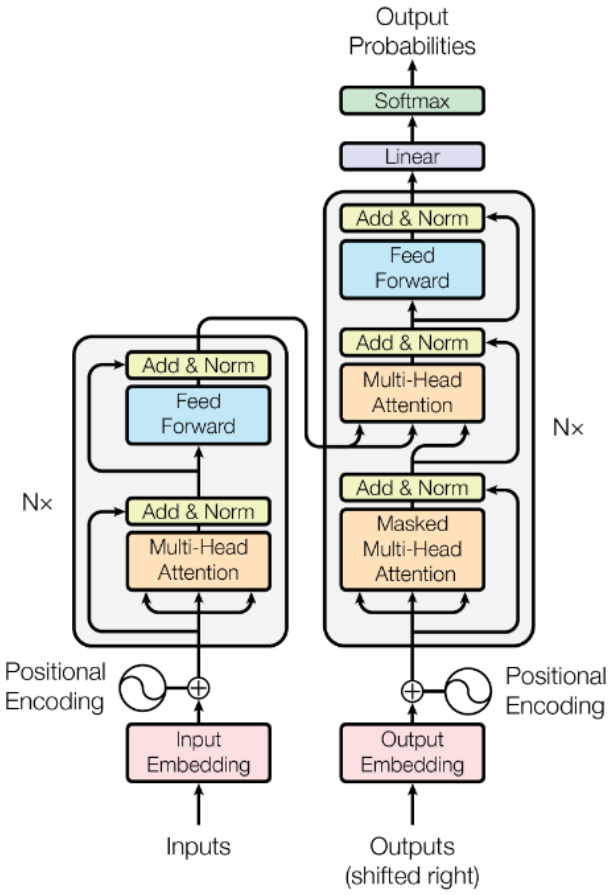

Figure 2.2: This figure shows the original transformer architecture with the encoder and decoder part. Adapted from *Attention Is All You Need* by Vaswani et al. (2023) [50].

the encoder is used to encode the input sentence or question. During inference, the decoder generates the tokens on a token-by-token basis. It produces either the answer or the translation, depending on what task the transformer was trained to perform.

The exact details of this are outside of the scope for this thesis. However, in 2018 the GPT-1 model was introduced by openAI [37]. This model relied only on the transformer decoder and thus was purely autoregressive —meaning it generates tokens one by one, with each token being conditioned on the previous ones. Interestingly, this model was able to perform tasks such as translation and question answering, and many others. These tasks were possible as long as the initial sequence properly primed the model into outputting the right response. This starting sequence is often referred to as the prompt.

Other popular decoder-only models include GPT-2 and GPT-3 [38] [5]. Additionally, all of the llama-models from Meta [48] [49] [14] are decoder-only models. The popularity of these models can be attributed to the fact that they can be trained to predict the next

token over very large text corpora without the need for any supervision. Moreover, their architecture allows for parallelized training, which we will discuss in more detail in the section on training.

Because of the aforementioned properties, the decoder-only transformer model is very amenable to scaling both data and the number of parameters/layers. This makes it a popular approach among large and well-funded companies such as OpenAI and Meta.

As mentioned before, the LLaMA-3.2 model is a decoder-only transformer model. When the embedding vectors derived from the tokens are being given as input, these vectors go through the following types of layers:

- Attention layer
- Feedforward layer
- RMSNorm

We will discuss each layer in detail, and afterward how they form the transformer decoder.

2.5 Attention layer

The LLaMA-3.2 model uses a variant of multihead attention that is more efficient than the normal multihead attention (MHA) called grouped-query attention (GQA) [1]. In Figure 2.3 the different versions of attention are displayed visually.

For single-head attention, each embedding vector x_i gets transformed by three matrices W_q, W_v, W_k into queries q_i, keys k_i, and values v_i, respectively. Let n be the number of different embedding vectors the set of these embedding vectors is denoted as $\{x_i\}_{i=1}^n$. The terms 'queries', 'keys', and 'values' are intentionally chosen. The naming reflects the idea of querying a database, where queries retrieve information based on keys. Each embedding vector produces a query, which is compared with the keys of all the other embedding vectors in the sequence derived from the text [13]. The similarity scalars $(s_j)_{j=1}^n$ are used to compute a weighted sum over the values $\sum_{j=1}^n s_j v_j$, producing the output vector o_i. This process is similar to database retrieval. This similarity scalar consists of the inner product between the queries and the keys, followed by normalization. We will discuss this in more detail when we describe the grouped-query attention.

For the first layer instead of transforming by just W_q, W_v, W_k we also apply the rotation:

$$f_{\{q,k\}}(x_m, m) = R(\Theta, m) W_{\{q,k\}} x_m.$$

Applying either W_q or W_k depending on whether we must output queries or keys.

Then for multihead attention (MHA) we repeat this but with different weight matrices for each head, obtaining o_i^h, with $h = 1 \dots H$, with H being the number of heads.

Because the queries from one embedding vector can retrieve values from any other vector, the model can learn to capture the nuances of language. However, this detailed retrieval process increases memory usage and reduces the speed of both inference and training. The way in which LLaMA-3.2 solves this problem is to make use of grouped-query attention instead of using normal multihead attention.

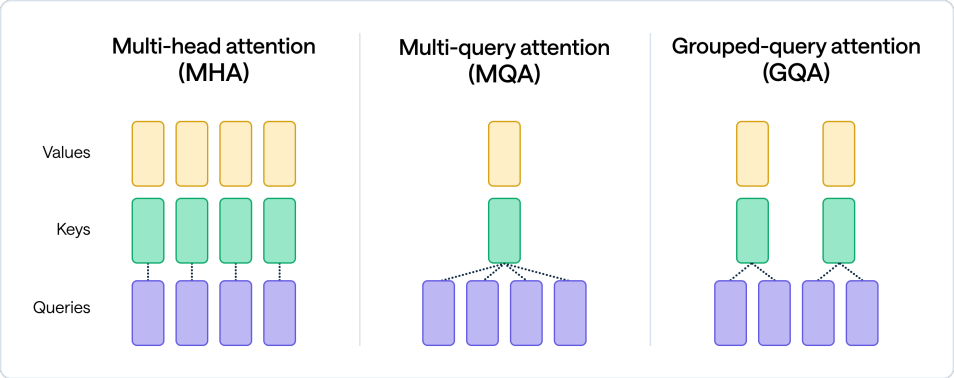

Figure 2.3: This figure shows the different attention mechanisms: multihead attention (MHA), multi-query attention (MQA), and grouped-query attention (GQA). Adapted from Friendli.ai [13].

GQA was introduced by Google [1] to control the trade-off between performance and quality. In the case of MQA, there was a small but noticeable drop in performance Ainslie et al. [1] compared to multihead attention. As we will discuss, MQA and multihead attention can be seen as a special case of GQA. Hence, we will present the mathematical formulation of GQA, and discuss how the other two types of attention can be derived from it, which reveals why it strikes a balance between computational demand and flexibility.

Let us consider the following sentence: "The dog bit the man". For simplicity, assume each word is transformed into a single embedding vector. We will ignore positional encoding for now and focus solely on just the GQA computation.

Then after mapping to embedding vectors we obtain $x_i \in \mathbb{R}^{d_{model}}$, $i = 1, \ldots, 5$, and each x_i is a d dimensional vector with d_{model} being the embedding dimension. For the sake of simplicity let us consider d_{model} being of dimension 10. In LLaMA-3 this dimension is 4096 [11].

We will start with the definitions:

- b: Batch size, this defines the number of sentences to process in parallel.
- n: Sequence length, this defines the largest sentence within the batch. Since the transformer computations require uniform length tensors, sentences shorter than this maximum length n must be extended by adding special tokens known as padding tokens. Padding makes sure each sentence matches the maximum length n. The attention mechanism masks these padding tokens to prevent them from influencing the computations (this will be explained in more detail later).
- d_{model}: Embedding dimension of the model, the input tokens get this many entries when transforming them into vectors.
- h_q: Number of query heads. For each embedding vector, we get h_q query vectors.
- h_k: Number of key/value heads. For each embedding vector, we get h_k key and value vectors.

- g: Number of groups, which is the ratio of query heads to key/value heads ($g = h_q/h_k$).

- d: head dimension, which is equal to $d = d_{model}/h_q$

Now, the embedding vectors x_i, $i = 1, \ldots, 5$ can be put into the matrix form:

$$X = \begin{pmatrix} \text{———} \mathbf{x}_1 \text{———} \\ \text{———} \mathbf{x}_2 \text{———} \\ \text{———} \mathbf{x}_3 \text{———} \\ \text{———} \mathbf{x}_4 \text{———} \\ \text{———} \mathbf{x}_5 \text{———} \end{pmatrix}$$

Here, the matrix has a shape of $(5, d_{model})$; in the general case, the shape is (b, n, d_{model}).

Then in order to obtain the query matrix, we multiply the embedding matrix X by the learned weight matrix $W_q \in \mathbb{R}^{d_{model} \times d_{model}}$. The resulting query matrix Q has shape (b, n, d_{model}) or in our case $(1, 5, d_{model})$ which can be considered (isometrically) equivalent to $(5, d_{model})$.

Now, let $d_{kv} = d \cdot h_k$. We obtain the key matrix K by multiplying the embedding matrix X with the learned weight matrix $W_k \in \mathbb{R}^{d_{model} \times d_{kv}}$. This results in the key matrix $K = XW_k$ with shape $(5, d_{kv})$ in our specific case, and in the general case (b, n, d_{kv}).

Similarly, for V, we calculate it as $V = XW_v$ where $W_v \in \mathbb{R}^{d_{model} \times d_{kv}}$, resulting in V having shape (b, n, d_{kv}) as well or in our case $(5, d_{kv})$.

We now have our keys, values, and queries in matrix form as K, V and Q. However, in order to continue the calculations, the matrices need to be reshaped. While this operation is not standard in traditional linear algebra, it is very common in deep learning. It consists of rearranging the entries of a matrix (or tensor, as it is often called in deep learning) to fit the desired shape. Suppose we have a tensor X of shape $(d_1, \ldots, d_n)$, where the total number of elements is:

$$N = d_1 \times d_2 \times \cdots \times d_n$$

A reshape operation transforms this tensor into another tensor Y with a new shape $(c_1, c_2, \ldots, c_m)$, such that:

$$c_1 \times c_2 \times \cdots \times c_m = N.$$

The reshape preserves the total number of elements and their ordering, but changes how these elements are indexed.

Formally, we can express a reshape as a bijection from the original indices to the reshaped indices. Given an element at original indices $(i_1, i_2, \ldots, i_n)$, its position in a flattened representation is computed by:

$$\text{flat\_index} = i_n + d_n i_{n-1} + d_n d_{n-1} i_{n-2} + \cdots + (d_n d_{n-1} \ldots d_2) i_1$$

Then, to get the new indices $(j_1, j_2, \ldots, j_m)$ in the reshaped tensor, we invert this flattening:

$$\begin{aligned} j_m &= \text{flat\_index} \mod c_m \\ j_{m-1} &= \left\lfloor \frac{\text{flat\_index}}{c_m} \right\rfloor \mod c_{m-1} \\ &\vdots \\ j_1 &= \left\lfloor \frac{\text{flat\_index}}{c_m c_{m-1} \ldots c_2} \right\rfloor \end{aligned}$$

We will start with reshaping Q:

$$Q_{reshaped} = reshape(Q, (b, n, h_q, d))$$

This effectively splits the d_{model} sized vectors into h_q query vectors, each of dimension d. Similarly K and V are reshaped as follows:

$$K_{reshaped} = reshape(K, (b, n, h_k, d))$$

and

$$V_{reshaped} = reshape(V, (b, n, h_k, d))$$

resulting in the shapes (b, n, h_q, d), (b, n, h_k, d) and (b, n, h_k, d) respectively.

Next, in order to calculate the attention, we need to first group the queries into g groups, each containing h_k heads. After this, we permute the dimensions to prepare for the attention calculation:

$$Q_{grouped} = reshape(Q_{reshaped}, (b, n, g, h_k, d)).permute(0, 2, 3, 1, 4)$$

giving $Q_{grouped}$ shape (b, g, h_k, n, d). We apply a similar permutation to K:

$$K_{permuted} = permute(K_{reshaped}, (0, 2, 1, 3))$$

resulting in $K_{permuted}$ having the shape (b, h_k, n, d).

We now calculate the inner product between the d-dimensional query vectors and d dimensional key vectors, by multiplying $Q_{grouped}$ and $K_{permuted}$ along the d-dimension:

$$\alpha_{b,g,h,n,\hat{n}} = \frac{\sum_{i=1}^{d} Q_{b,g,h,n,i} \cdot K_{b,h,\hat{n},i}}{\sqrt{d}}$$

Here we use $\hat{n}$ to denote the third axis in the tensor $K_{permuted}$, as the variable name n is already in use when describing the fourth axis in the tensor $Q_{grouped}$.

Notice that for each group, the attention score can be calculated in parallel. Moreover as we will see, K has less entries here than in full MHA. As a result, GQA requires less memory access during computation, which reduces the time needed perform the computation.

The entries in this attention score represent how strongly the query and key vectors are correlated. A highly positive score will result in the corresponding value vector having a

significant influence on the output for that position in the sequence. Conversely, a very negative value will result in the value vector belonging to this key vector having almost no effect on the output vector.

This will become clearer in later sections when we describe the rest of the computation. The key point here is that any sequences shorter than the longest sequence must be padded to create a valid tensor so that the attention score calculation can be performed properly. However, these artificially added padding tokens should *not* contribute to the predictions of the model. To make sure this is the case, the attention scores corresponding to padding positions are explicitly set to $-\infty$.

After this score α is calculated we apply the softmax function along the $\hat{n}$ axis:

$$A_{b,g,h,n,\hat{n}} = softmax_{\hat{n}}(\alpha_{b,g,h,n,\hat{n}})$$

The softmax function normalizes the scores along the $\hat{n}$ axis, converting them into a probability distribution [8]:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Where x is a vector of values. So, in this specific case, the softmax evaluates to:

$$A_{b,g,h,n,\hat{n}} = \frac{e^{\alpha_{b,g,h,n,\hat{n}}}}{\sum_{i=1}^{\hat{n}} e^{\alpha_{b,g,h,n,i}}}$$

These values are then used as weights for the output tensor:

$$O_{b,g,h,n,d} = \sum_{i=1}^{\hat{n}} A_{b,g,h,n,i} V_{b,h,i,d}$$

Due to the $-\infty$ values in α, the corresponding entries in A will be set to zero, because of the way the softmax is calculated. This ensures that the values associated with padding, as explained earlier, do not affect the output.

To obtain the final output we need to perform some reshaping and permutations. We start by permuting the output:

$$O_{permuted} = permute(O, (0, 3, 1, 2, 4))$$

This results in $O_{permuted}$ having shape (b, n, g, h, d). By permitting O into the shape (b, n, g, h, d) first, we place the token dimension n next to batch dimension b. This makes sure that when we flatten the dimensions g and h into a single dimension $h_q = g \cdot h_k$, each resulting h_q-dimensional vector correctly corresponds to exactly one token position. Next, to combine the groups, each containing h_k d-dimensional vectors, we reshape the output:

$$O_{combine} = reshape(O_{permuted}, (b, n, h_q, d))$$

where we use the fact that $h_q = g \cdot h_k$. Finally, we combine the h_q many d-dimensional vectors into a single vector of size d_{model}:

$$O_{flattened} = reshape(O_{combine}, (b, n, d_{model}))$$

In our simplified example, this results in $O_{flattened}$ having shape $(1, 5, d_{model})$ or more simply $(5, d_{model})$, ignoring the redundant first dimension. In the general case the shape of $O_{flattened}$ has shape (b, n, d_{model}).

To obtain MHA we set h_k to h_q, which results in $g = 1$. This eliminates the advantage of sharing keys across groups. Additionally, the tensor K becomes larger leading to increased latency due to the overhead of loading larger tensors from memory, which slows down computation.

2.6 FeedForward layer

The FeedForward layer takes the output of the multi-head attention mechanism and applies the following transformation [15]:

$$y = W_2^T(\text{SiLU}(W_1^T\mathbf{x}) \odot W_3^T\mathbf{x}),$$

where:

- $\mathbf{x} \in \mathbb{R}^{dim}$: input, this denotes the input of the FeedForward layer, usually the output of the attention layer.
- $W_1, W_3 \in \mathbb{R}^{dim \times hidden\_dim}$: learned weight matrices that usually upscale the input significantly in terms of its dimension, i.e. $hidden\_dim \gg dim$.
- $W_2 \in \mathbb{R}^{hidden\_dim \times dim}$: learned weight matrix that downscales the input back to its original size again.
- SiLU: sigmoid linear unit activation function [36]; see Figure 2.4
- $\odot$: elementwise multiplication.

The FeedForward layer enables the model to capture more non-linearities in the data by expanding each input vector to the hidden dimension, applying the non-linearity entrywise, and projecting the result back to its original size. Without the Feedforward layers, the model would in fact be unable to capture non-linear relationships in the data since the attention layer is fuly linear.

2.7 RMSNorm

This layer applies root mean square normalization (RMSNorm) to the output of the previous layer. In the case of this thesis, the previous layer is the attention layer. The RMS function is defined as:

$$RMS(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2},$$

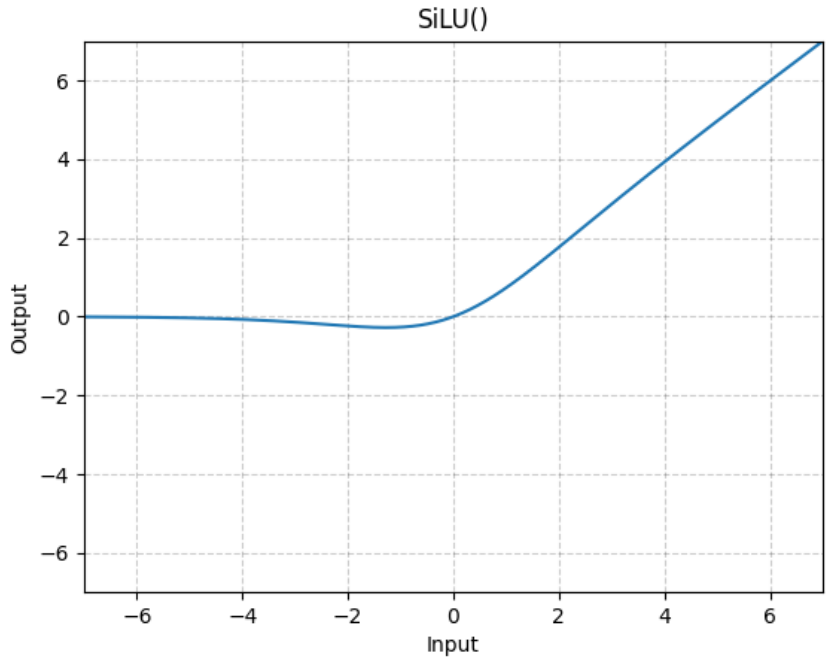

Figure 2.4: SiLU (Sigmoid Linear Unit) activation function, adapted from the official PyTorch documentation [36].

where x has shape (b, n, d), representing batch size b, sequence length n, and feature dimension d. The elements of x are squared entrywise, summed over the feature dimension, and then averaged. Then the square root is applied. More precisely:

$$RMS(x)_{b,n} = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_{b,n,i}^2},$$

and the output of the layer is

$$y = \frac{x}{RMS(x) + \varepsilon} \cdot \gamma,$$

where again, more precisely, we have:

$$y_{b,n,d} = \frac{x_{b,n,d}}{RMS(x)_{b,n} + \varepsilon} \cdot \gamma_d.$$

Here, ε makes sure no division by 0 occurs, and γ is a learnable vector with d entries. For this division to work, the $RMS(x)$ is broadcast across the feature dimension d. This means its elements are repeated for each entry in the feature dimension, and the division is applied elementwise. For each value of b (batch index), n (sequence index), and d, we use the value of $RMS(x)$ at index b, n along with the same scalar value of ε. This concept of broadcasting is also applied to the multiplication by γ. The purpose of RMSNorm is

to scale each layer according to the mean. According to the authors, [54], this improves stability during training.

2.8 Transformer block

In Figure 2.5, the LLaMA-3.2 architecture is shown with the layers we have discussed in the previous sections. It can be noticed in Figure 2.5 that the transformer block is repeated N times, where N is 32 in the case of LLaMA-3.2-8B. The only layer we have not touched on is the linear layer which is, as the name suggests, just a linear mapping that will make sure the embedding vectors will map to a tensor that is as large as the vocabulary of the model. So that after applying the softmax it can be treated as a probability distribution over the next token.

Furthermore, the $\oplus$ symbol signifies the residual connections. These are there to mitigate the vanishing gradient problem, they were first introduced in He et al. [16].

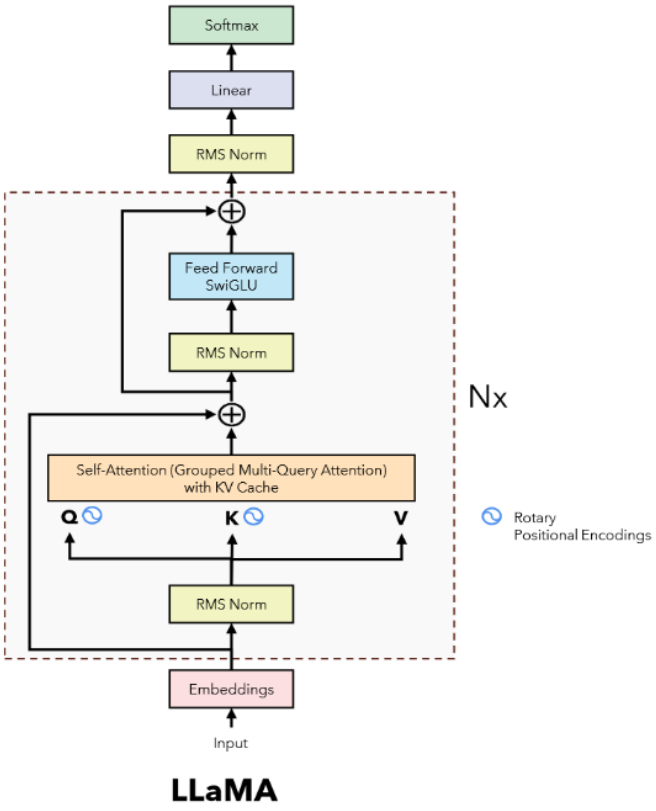

Figure 2.5: This figure shows the LLaMA-3.2 architecture. Adapted from *PyTorch LLaMA Notes* by Umar Jamil (2023) [20].

2.9 Training Large Language Models

Large language models such as Qwen [3], DeepSeek [10], Meta's models [14], Mistral [21] and the GPT series such as GPT-1, GPT-2, GPT-3 [5] are all pretrained using autoregressive next-token prediction.

As mentioned in the previous section, this approach works well because of the large amount of text data available online. Moreover, predicting the next subword in a sequence does not require generating new labels, since the labels are simply the next subword in the text.

We will illustrate autoregressive next-token prediction with a simplified example. Let us again look at the following example sentence:

The dog bit the man

We will again make a simplifying assumption that the tokens consist of individual words. Then, the input will have shape $(5, d_{model})$ or $(1, 5, d_{model})$ if we consider batches. In the general case, the shape will be (b, n, d_{model}), where we reuse the variable definitions from the previous section.

The final output of the model, denoted as $\hat{y}$, is obtained after processing the input through all layers, including the last linear layer and softmax. The notation $\hat{y}$ distinguishes the predictions of the model from y, which represents the ground truth label from a dataset this model would be trained on.

The model output $\hat{y}$ has shape (b, n, V), where V denotes the vocabulary size. This vocabulary size is the same number as the number of tokens from Section 2.1. For each batch b and each token position n, the model outputs a probability distribution over the vocabulary. Specifically, for all b and n:

$$\sum_{v=1}^{V} \hat{y}_{b,n,v} = 1$$

Let us introduce a notation for the model. Let f_θ represent the LLaMA-3.2 model, where x is the input of shape (b, n, d_{model}) and θ is a vector of trainable parameters in the model. The model outputs a tensor of shape (b, n, V). The goal of training f_θ is to maximize the probability of outputting the right tokens.

An important detail to note is that, when training the transformer model autoregressively, a causal mask is applied. This ensures that the model learns to predict each token based only on the previous tokens, not to the ones that come after. For each batch b, the causal mask is applied such that for each token position n, it prevents it from attending to positions $i > n$:

$$\alpha_{b,g,h,n,i} = -\infty$$

This makes sure that the query corresponding to the n^{th} embedding vector/token cannot retrieve the values of any embedding vectors/tokens that come afterwards. In this way we remove the risk of the transformer model "cheating" by using future tokens (including the very token it tries to predict).

The labels are generated by taking the next token $i+1$ for each token $i = 1, \ldots, n-1$. For each of these tokens, we find its index in the vocabulary and create a vector of V entries with zeros everywhere except on the desired word/subword's index. There we put a one instead. Repeating this process for each batch b and each token in the sequence n we get the desired labels y with shape (b, n, V).

The training of these models is based on gradient descent, applied to a loss function called cross-entropy loss:

$$\mathcal{L}_{b,n}(y, \hat{y}) = -\sum_{v=1}^{V} y_{b,n,v} \log(\hat{y}_{b,n,v}).$$

Since the dataset is very large, the loss is usually calculated for subsets (batches) of the dataset:

$$\mathcal{L}_s(y, \hat{y}) = -\frac{1}{n \cdot z} \sum_{b=s}^{s+z} \sum_{i=1}^{n} \sum_{v=1}^{V} y_{b,i,v} \log(\hat{y}_{b,i,v}),$$

where s denotes the s^{th} batch of size z, these batches are randomly selected during training uniformly. The uniform random selection makes sure the different parts of the dataset will inform the updates in the model approximately equally. The loss is in this case averaged across the sequence and across the subset of the batch.

The gradient descent step is as follows:

$$\theta_i = \theta_i - \eta \frac{\partial \mathcal{L}(y, f_\theta(x))}{\partial \theta_i}$$

Where η is the learning rate, which usually is some small positive number. The term $\frac{\partial \mathcal{L}(y, f_\theta(x))}{\partial \theta_i}$ points in the direction that will result in the steepest increase of the loss value. Thus, updating each θ_i in the opposite direction, scaled by η will make it move into the direction with the steepest decrease. In other words, this process "nudges" all the parameters of the model in such a way that cross-entropy loss decreases.

This results the term $\sum_{v=1}^{V} y_{b,n,v} \log(\hat{y}_{b,n,v})$ getting closer to zero for each token in the sequence and each batch, assuming that the batches are representative of the full dataset. To see why this is effective, observe that when the model predicts the next token with high probability, the loss term for that token goes to zero. Specifically, if the model outputs a probability close to 1 for the correct token, we obtain:

$$y_{b,n,v} \log(\hat{y}_{b,n,v}) \approx 1 \cdot 0 = 0$$

at the index v that represents the correct next token index. Where we used $\log(1) = 0$ in the approximate equality. Conversely, for incorrect tokens:

$$y_{b,n,v} \log(\hat{y}_{b,n,v}) \leq 0 \cdot M = 0$$

where $y_{b,n,v} = 0$ for incorrect tokens. Here, M represents a real number such that $\log(\hat{y}_{b,n,v}) \leq M$.

In this thesis and many other works a more advanced optimizer than standard gradient descent [45] [9] is used. Specifically, we use the 'AdamW' [26] optimizer, which adapts the learning rate for each parameter θ_i and penalizes larger values for θ_i. Furthermore, the learning rate is not constant; it decays linearly as training progresses [26]. Lastly, we apply the LORA finetuning method in this thesis [19]. This method, along with the other details mentioned in this paragraph, is beyond the scope of this thesis, but interested readers can refer to the references cited above for more information.

2.10 Inference methods

After training, the model can provide a probability distribution over possible output tokens given a sequence of tokens. Language model inference involves sampling from this probability distribution in an effective way. In this section we will mainly cover beam search and top-p sampling, which are two examples of inference methods. The former tries to generate text with high likelihood under the model, while the latter focuses on increasing diversity.

In Figure 2.6 the two types of sampling are compared. Beam search tends to produce more redundancy, since repetitive text is often assigned high probability. Pure sampling, which selects tokens directly from the model's probability distribution and iteratively appends them to the text, often results in incoherent output. This is because rare tokens, once sampled, cannot be undone, forcing the model to continue generating even though the text already has degraded quality. Moreover, because most text found on the internet is coherent, the input to the model will be out-of-distribution once low-quality tokens are sampled, degrading the quality of token generation afterwards. Lastly, Top-p shows little incoherence and less repetition than Beam search. As will be discussed in Section 2.10.2, top-p is similar to Pure sampling, resulting in having more diversity, but mitigates the incoherence by avoiding the sampling of rare tokens.

2.10.1 Beam search

Under the assumption that the model assigns higher probability to higher-quality text, a naive method of sampling would be to pick the highest-probability token at each step. This method is known as greedy search. However, a drawback of this method is that selecting the most likely token at each step independently may result in a lower-probability sentence than when the tokens are jointly selected to maximize the overall likelihood. Given our assumption, this can lead to a less natural sounding sentence.

Beam search solves this problem by maintaining B candidates, where B is known as the beam width, rather than only selecting the single most likely token. At each step, beam search expands all B candidates [32]. This produces B^2 possible continuations. These candidates are then pruned on their multiplicative weight, leaving B top candidate sequences. The multiplicative weight for each possible sequence (y_i, y_j), with $i, j \in \{1, \ldots, B\}$ and starting prompt x, is calculated as:

$$\mathbb{P}((y_1, y_2)|x) = \mathbb{P}(y_1|x) \cdot \mathbb{P}(y_2|x, y_1)$$

Choosing a sufficiently large beam width results in sequences with a much higher likelihood than those produced by greedy search. However, despite the high likelihood, the generated text often appears overly repetitive when read by a person, as shown in Figure 2.6. This repetition most likely occurs because repetitive text is more predictable, leading the model to assign a higher likelihood to it even though the text sounds more generic and redundant.

A way to address this issue is to place less emphasis on maximizing the likelihood of a certain text and occasionally sample less likely tokens. This comes at the cost of a lower

likelihood but results in a more varied text that is often of higher quality. The next sampling method will follow this principle.

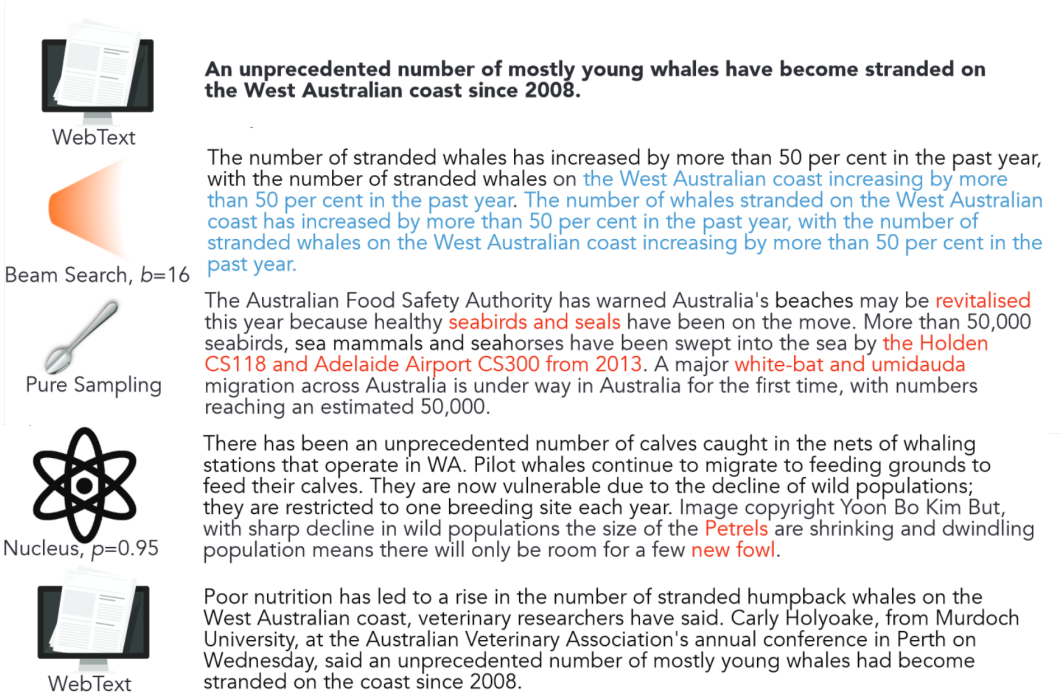

Figure 2.6: Comparison of generations produced by beam search, top-p sampling, and pure sampling. The blue highlighting represents excessive repetition, and the red highlight indicates incoherence. Adapted from Holtzman et al., *The Curious Case of Neural Text Degeneration* (2020) [18].

2.10.2 Top-p sampling

In Top-p sampling, rather than pursuing tokens that jointly optimize the likelihood, we construct a "nucleus" of high-likelihood tokens from which we sample [18].

Specifically, we create a top-p vocabulary $V^{(p)} \subset V$ as the smallest subset of the vocabulary such that [18]:

$$\sum_{x \in V^{(p)}} \mathbb{P}(x|x_{i:(i-1)}) \geq p$$

Where x represents the potential next token and $x_{i:(i-1)}$ is the sequence of tokens that forms the context. Lastly, p is the probability threshold. In practice, this subset is constructed by adding the highest probability tokens in descending order until the sum of probability exceeds the threshold p.

After constructing this set, the probabilities within it are renormalized to sum to 1, while tokens outside this set are given probability 0. On this new probability distribution, sampling takes place.

2.10.3 Temperature scaling

Temperature scaling is an inference method that adjusts token probabilities before sampling. In temperature scaling, we modify the logits y_i of the second last output layer of the Large Language Model. Temperature scaling, in the form of modifying logits before applying the softmax, was first introduced in Hinton et al. [17], where it was used to smooth softmax probabilities. Recall, the output layer of the Large Language Model is a probability distribution where each token gets assigned a probability; these probabilities are derived from the so-called logits from the model. These logits are fed through the softmax function to produce the probabilities:

$$\begin{aligned}(p_i)_{i=1}^N &= \text{Softmax}(N; y_0, \ldots, y_N) \\ &= \left(\frac{e^{y_0}}{\sum_{j=0}^N e^{y_j}}, \frac{e^{y_1}}{\sum_{j=0}^N e^{y_j}}, \ldots, \frac{e^{y_N}}{\sum_{j=0}^N e^{y_j}} \right)\end{aligned}$$

When we apply temperature scaling, we modify this softmax with a division by T:

$$p_i = \frac{e^{y_i/T}}{\sum_{j=0}^N e^{y_j/T}}.$$

A larger value for T, will make the probabilities more uniform, resulting in less probable wordpieces becoming more likely. This promotes more diverse outputs being generated by the model. A smaller value for T magnifies the differences between the logits, thus making higher likelihood wordpieces even more likely, and less likely wordpieces even less likely. This results in the model generating less diverse and more conservative outputs.

Chapter 3

Lean 4

3.1 Dependent type theory

In this thesis, we aim to have a large language model interact with a program that validates proofs, known as a proof assistant. For the proof assistant to validate a proof, it must be written in a formal language. The formal language and proof assistant we are using is called LEAN4. This language has as its foundation in dependent type theory, specifically the Calculus of Inductive Constructions (CIC). In this section, we will give an introduction to dependent type theory.

In almost every mathematics program, we start with its foundation in set theory. However, describing every object as a set misses the benefits of using types to clearly distinguish the various objects, as developed in programming languages.

In most programming languages we have variables with types, types such as integers and floating points, and operations that can only be done on variables of a certain type. To run a program written in such a language, it often needs to be compiled, though this is not always the case. In this process the compiler first checks if the types match within the program. For example, if a function expects a certain type as an argument, the type of the variable passed to it must match. If the types do not match, the compiler will throw an error and will not return the executable the computer can run to execute the desired program.

To make optimal use of this type-checking paradigm when trying to validate mathematics, the developers of LEAN4 used dependent type theory as its foundation. Analogous to how every mathematical object in set theory is a set, in dependent type theory, every mathematical object is a term with a specific type [6]. One fundamental type is the function type. For example, if a term f has a function type that takes input of type A and produces output of type B, then this is denoted as $f : A \to B$.

On these types, specific typing rules define how they can be combined and what the resulting type will be [6]. For instance, consider a term f with a function type $A \to B$, which maps terms of type A to terms of type B. Then provided a term x of type A (written as $x : A$), a typing rule dictates that applying f to x, written as $f(x)$ or fx, results in a term of type B, i.e., $f(x) : B$. If x were of type B instead of A, the LEAN4 proof assistant will throw an error, as this combination of terms would be illegal. The application of such

typing rules leads to "typing judgments" [6].

The typing rule for function application more formally can be described as:

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash x : A}{\Gamma \vdash f(x) : B},$$

this has the form of

$$\frac{\text{Premises}}{\text{Conclusion}},$$

The left part of the premise says that out of context Γ we can obtain f having type $A \to B$. The right part of the premise says under the assumptions in Γ, we can obtain x has type A. Then the conclusion says, under the assumptions in Γ we can obtain that $f(x)$ has type B.

Both the conclusion and the left and right part of the premise are type judgments. Type judgments are statements of the form:

$$\Gamma \vdash t : T,$$

asserting a term t has type T. Another important type in dependent type theory is the product type. Suppose we have two values $a : A$ and $b : B$, we can form a pair that has type $A \times B$, called the product type:

$$(a, b) : A \times B$$

In set theory this structure is constructed using sets. In LEAN4, which uses the calculus of inductive constructions (CIC) form of dependent type theory, this is defined through inductive types. We will discuss inductive types in later sections.

Everything covered so far is also captured within basic type theory. However, LEAN4 uses dependent type theory so that types can depend on other terms [6]. For instance, Buzzard et al. [6] provides the example of the type $\mathbb{R}^n$, where the type depends on another term $n : \mathbb{N}$. This is called a dependent type.

A different type that involves dependencies between terms and types is the sum type. Sum types allow a term to belong to multiple types. Suppose $t : A$, functioning as an index or tag, and $x : B(t)$. Then (t, x) is a sum type, where t is a "tag" that indicates to which type x belongs to. The notation for this type is [22]:

$$(t, x) : \sum_{t:A} B(t)$$

where t is of type A and x is of type $B(t)$. Here $B(t)$ is a type-valued function, making it a dependent type.

Up until now, we have discussed typing judgment, typing rules, and some important types, such as sum types, product types and function types. However, an essential part of dependent type theory is still missing: reduction rules from λ-calculus [6]. For example, suppose we have a term $(n \mapsto n \cdot n) : \mathbb{N} \to \mathbb{N}$. When applied to $3 : \mathbb{N}$, this term reduces to $3 \cdot 3$ by a reduction rule. This is then by definition of multiplication equal to 9. This reduction rule is known as β-reduction.

Given the versatility of the described types, along with inductive types (covered in the next section), many mathematical statements can be expressed using them. In this setup, assumptions are treated as terms of a type representing the kind of propositions they describe.

We will use an illustrative example from [6]. Let P and Q represent types corresponding to a mathematical statement. We can then associate $P \Rightarrow Q$ with the function type $P \to Q$ (formally known as the Curry-Howard isomorphism [6]). Using this propositions as types framework, we can construct a proof term of type Q by composing a function $h : P \to Q$ with a proof term $h_P : P$, giving $h\ \ h_P : Q$, using the typing rules. In this way, we can see that proving becomes a form of typing judgment [6].

3.2 Inductive types

Inductive types are another essential way to construct other types in Calculus of Inductive Constructions (CIC). Important types that are built using inductive types are $List$, $Bool$, $\mathbb{N}$,$\times$ (product type) and Σ (sum type). In fact except for the function type, the universal quantifier ($\forall$) and type universes, every type is constructed using inductive types [6]. What is especially surprising is that equality can be defined as an inductive type. This is however out of scope for this thesis. For the curious reader we recommend the following reference [2] for more information on this topic, specifically, the chapter on inductive types will hit on this specific topic.

Type universes are where the types themselves reside, we will not discuss this in detail. In LEAN4 these type universes are denoted by `Sort u` or `Type u` where u denotes the level, and `Type 0 = Sort 1` i.e. they are equivalent up to a difference in indexing. Another fact we will use is that `Sort 0` is where propositions reside and `Sort 1` is where $\mathbb{R}, \mathbb{Z}$ and $\mathbb{N}$ resides.

The name mathematical induction actually has to do with the principle of mathematical induction. The principle of mathematical induction on natural numbers follows naturally from how natural numbers are constructed using inductive types. We will discuss this in later sections. Let us first discuss how the construction looks like in the case of natural numbers:

```
inductive ℕ : Type
| zero : ℕ
| succ : ℕ → ℕ
```

An inductive type enumerates the ways in which the type that is being defined can be constructed. In the case of the natural numbers, there are two. The first constructor is $zero : \mathbb{N}$, so this can be used to define a term without any arguments. On the other hand, we have the constructor $succ : \mathbb{N} \to \mathbb{N}$, which can only be applied using a previously constructed natural number. We can obtain any number of successors of zero using the $succ$ function. The first successor can be constructed using $succ\ zero : \mathbb{N}$, which in turn is succeeded by $succ(succ\ zero) : \mathbb{N}$. Moreover, what is implicit in this definition is that each of these terms is unequal to one another, i.e. differently constructed terms of an inductive type are unequal. This results in 0 being unequal to 1 in particular [6]. In lean this principle is called the no confusion principle and is automatically generated when you define an inductive type [35].

The inductive type for $\vee$ is:

```
inductive ∨ (a b : Prop) : Prop where
  | inl : a → a ∨ b
  | inr : b → a ∨ b
```

Where there are two ways to construct the type $a \vee b$. The arguments it takes in must be of type $Prop$, which are where the propositional types reside, $Prop$ is another name for 'Sort 0'. We only want $\vee$ to be used for terms of type $Prop$ since we only want this to be used in the context of propositional logic. For more examples see Avigad et al. [2].

3.3 Inductive types and induction

Let us again have a look at the inductive type for natural numbers. For every inductive type, LEAN4 automatically generates a recursor. In the case of natural numbers, this recursor can be used to apply induction.

The purpose of a recursor is to be able to define a dependent function f from $\mathbb{N}$ to any domain,i.e.:

$$f(n : \mathbb{N}) \to motive\ \ n$$

Where $motive : \mathbb{N} \to Sort\ \ u$ [2]. In other words $motive\ \ n : Sort\ \ u$. 'Sort u' here can describe any type universe.

Suppose we want to show that when zero is added to a natural number this is equal to the number itself, i.e. $0 + n = n$. Let us describe how this would work with the recursor. The function signature of $\mathbb{N}.recOn$ is:

```
(motive : ℕ → Sort u)
→ (t : ℕ )
→ motive zero
→ ((n : ℕ ) → motive n → motive (succ n))
→ motive t
```

We want to establish a proof, so the $motive : \mathbb{N} \to Sort\ \ u$, will have $u = 0$, since we want to create a term with type proposition $Prop$. $motive$ will be defined as $((n : \mathbb{N}) \mapsto 0 + n = n)$. The second argument in recOn will specify on which variable induction is applied. In our case this is $n : \mathbb{N}$. The third and fourth argument will be the base case and induction step, $motive\ \ zero$ will after applying the typing rules and reduction be equal to:

$$0 + 0 = 0 : Prop$$

LEAN4 will in this case reduce the addition using its definition obtaining $0 = 0$, this process is called elaboration. In LEAN4 this term can be constructed using 'rfl' which uses the reflexivity principle of equality to produce a term of the type $0 = 0 : Prop$. Since 'rfl' produces a term of exactly the type $0 = 0$ we have produced the desired proof term, hence we have proven the base case. The fourth argument is the induction step where we are supposed to provide a function taking value $(n : \mathbb{N})$ a term ih of type $motive\ \ n$ (proof of $0 + n = n$, the induction hypothesis) and the function should then return $motive\ \ (succ\ \ n)$ (proof of $0 + (n + 1) = (n + 1)$).

Before continuing, let us establish that $0 + succ\ \ n = succ(0 + n)$, this follows from the reduction of the definition of addition and comparing this with the term $0 + succ\ \ n = 0 + succ\ \ n$ constructed by reflexivity. Furthermore, we know $0 + n = n$ by the induction hypothesis, thus substituting this in $succ(0 + n)$, we obtain $succ\ \ n$. In LEAN4 we can chain these calculations together using a calc statement:

```
calc 0 + succ n
    _ = succ (0 + n) := rfl
    _ = succ n       := by rw [ih]
```

Each := is followed by a proof of equality from the current line to the line before it.

The `rw [ih]` will look for a subexpression within a given type in this case $(0 + n)$ and will use the equivalence $0 + n = n$ provided by ih to substite it for n. `rw` is one of the many tactics provided by LEAN4, whenever you want to use a tactic you must first use the keyword `by`, i.e. `by rw [ih]`. We will talk about tactics in a later section. The complete code is:

```
theorem zero_add (n : Nat) : 0 + n = n :=
  Nat.recOn (motive := fun x => 0 + x = x)
   n
   (rfl)
   (fun (n : Nat) (ih : 0 + n = n) =>
    calc 0 + succ n
      _ = succ (0 + n) := rfl
      _ = succ n       := by rw [ih])
```

In LEAN4, whenever we want to specify a theorem, we first write the keyword 'theorem' and write the name of the theorem. Theorems are effectively functions that provided 0 or more arguments returns a term of the type that represents the mathematical statement you tried to prove. In this case it takes as its argument a term n of type $\mathbb{N}$.

The third arugment is the function with type:

$$(n : \mathbb{N}) \to motive\ n \to motive\ (succn)$$

Where we used a concept called currying. Using currying we can rewrite a function as:

```
(fun (n : ℕ) => (fun ih : 0+n = n => ...))
```

to

```
(fun (n : ℕ) (ih : 0+n = n) => ...)
```

Two additional ways to write `zero_add` that are all three equivalent to one another is: `theorem zero_add : (n : ℕ) 0+n = n :=` and `theorem zero_add : ∀(n : ℕ), 0+n = n :=` . The last statement makes use of a quantifier instead of a function, this quantifier in CIC is actually just a different notation for $(n : \mathbb{N}) \to (0 + n) = n$ [2].

The $recOn$ function in LEAN4 is rarely used, however when using higher level tactics such as `induction` or as we will later see `match`, recOn is used by LEAN4 under the hood. Below the theorem can be seen written fully in tactic mode:

```
theorem zero_add (n : ℕ) : 0 + n = n := by
  induction n with
  | zero => rfl
  | succ n ih =>
    calc 0 + succ n
    _ = succ (0 + n) := rfl
    _ = succ n := by rw [ih]
```

3.4 Tactics

As mentioned before, we prepend "by" to a keyword to let LEAN4 know you use a tactic. When "by" is used within the proof LEAN4 enters in tactic mode. Tactics make it more natural to write incremental proofs. After every tactic, the user will see what is still left to do to formally prove a theorem.

When in tactic mode we start with an initial goal state which specifies the desired type we want a term to have. We furthermore start with terms that are given as input variables. The goal and initial variables are separated by the symbol $\vdash$.

In the code section below a proof of a Proposition can be seen.

```
example : p ∨ q → q ∨ p := by
  /-
  p q : Prop
  ⊢ p ∨ q → q ∨ p
  -/
  sorry
  /- no goals -/
```

The `/- -/` enclosed sections are comments, here the states right after `by` and after `sorry` are shown. The tactic `sorry` can produce a term of any type and is usually used to break up a proof into smaller pieces assuming one part to be true, using that part of the proof somewhere else, and proving the part that you initially proved using `sorry`.

Tactics are small programs that build terms, the term is then applied to the goal state using type judgment and reductions which are mentioned in the previous section [6]. When the typing judgment fails LEAN4 throws an error that the tactic failed.

We will go through two example problem statements, proving $p \vee q \to q \vee p$ and proving $(\exists x, p\ x \wedge r) \to (\exists x, p\ x) \wedge r$.

```
example : p ∨ q →
    q ∨ p := by
  /-
  p q : Prop
  ⊢ p ∨ q → q ∨ p
  -/
  intro hp_or_q
  /-
  p q : Prop
  hp_or_q : p ∨ q
  ⊢ q ∨ p
  -/
  apply Or.elim
   hp_or_q
  case left =>
    /-
    p q : Prop
    hp_or_q : p ∨
   q
    ⊢ p → q ∨ p
    -/
    intro hp
    apply Or.inr
    exact hp
  case right =>
    /-
    p q : Prop
    hp_or_q : p ∨
   q
    ⊢ q → q ∨ p
    -/
    intro hq
    apply Or.inl
    exact hq
```

```
example : p ∨ q →
    q ∨ p := by
  /-
  p q : Prop
  ⊢ p ∨ q → q ∨ p
  -/
  intro hPQ
  match hPQ with
  | Or.inl hp =>
    /-
    p q : Prop
    hPQ : p ∨ q
    hp : p
    ⊢ q ∨ p
    -/
    apply Or.inr
    exact hp
  | Or.inr hq =>
    /-
    p q : Prop
    hPQ : p ∨ q
    hq : q
    ⊢ q ∨ p
    -/
    apply Or.inl
    exact hq
```

```
example : p ∨ q →
    q ∨ p := by
  /-
  p q : Prop
  ⊢ p ∨ q → q ∨ p
  -/
  intro hp_or_q
  /-p q : Prop
  hp_or_q : p ∨ q
  ⊢ q ∨ p -/
  cases' hp_or_q
   with hp hq
  /-
  case inl
  p q : Prop
  hp : p
  ⊢ q ∨ p
  case inr
  p q : Prop
  hq : q
  ⊢ q ∨ p
  -/
  case inl =>
    apply Or.inr
    /-
    p q : Prop
    hp : p
    ⊢ p
    -/
    exact hp
  case inr =>
    apply Or.inl
    /-p q : Prop
    hq : q
    ⊢ q-/
    exact hq
```

In the code above we see three similar ways of proving the first statement. Match uses, as the name suggests, pattern matching. As we have discussed $p \vee q$ is the inductive type with two constructors `Or.inl` and `Or.inr`, with `match` you have to define what operation to do for each possible constructor. However, you get to write it in an intuitive way and let the LEAN4 pattern-matching mechanism resolve what the types are of the variables written. In the example code thats `hp` and `hq`, with types `p` and `q` respectively.

Another way displayed above is using `cases'`. The `cases'` syntax will split the inductive type into the cases that could have brought about this term. Here we can choose the names of the terms as well. We then target the two subgoals that were brought about separately using `case` followed by the name of the newly created subgoal.

In the third code block we see how we can use `Or.elim` to deconstruct the $\vee$ inductive type. The keyword `apply` will make LEAN4 try to combine the goal and the term that follows the `apply` keyword. `Or.elim` has type: $Or.elim(h : a \vee b)(left : a \to c)(right : b \to c) : c$, using again currying to make implications into function arguments where a, b, c are propositions. So `Or.elim hp_or_q` will have type:

$$(Or.elim\ \ hp\_or\_q) : (p \to q) \to (q \to c) \to c$$

Now c here can still represent any proposition. But when using it in conjunction with apply, we will try to combine this term with the goal, resulting in c being changed to $q \vee p$. If alongside hp_or_q we gave left and right, we would obtain the desired term $q \vee p$. However, 'left' and 'right' are not provided so the 'apply' keyword will not throw an error for not having proven the goal, instead the missing pieces in the proof will become the next goals to prove.

In this case, the goals can be referenced with the names `case left` and `case right`, which will have the goals $p \to q \vee p$ and $q \to q \vee p$ respectively.

The last keyword worth mentioning is the `exact` keyword exact is very similar to apply however there is a restriction that there should be no goals left after the `exact` tactic. In the `Or.elim` part above we would for example have to provide the terms for `left` and `right` alongside `h`. For example, we can change

```
intro hp
apply Or.inr
exact hp
```

to `exact (fun hp : p => Or.inr hp)`, since this has type $p \to q \vee p$.

Let us now take a look at two proofs of the second problem:

```
example (p : α → Prop) : (∃ x, p x ∧ r) →  (∃ x, p x) ∧ r
  := by
  intro hExists
  apply Exists.elim (hExists)
  intro x
  intro hPxAndR
  have hxPx : ∃ x, p x := ⟨ x, hPxAndR.left⟩
  exact ⟨ hxPx, hPxAndR.right⟩
```

and

```
example (p : α → Prop) : (∃ x, p x ∧ r) →  (∃ x, p x) ∧ r
  := by
  intro ⟨ x, hPxAndR⟩
  have hxPx : ∃ x, p x := ⟨ x, hPxAndR.left⟩
  exact ⟨ hxPx, hPxAndR.right⟩
```

In the first code block, we see a new eliminator of $\exists$, this eliminator has type:

$$Exists.elim(h_1 : \exists x, p\ x)(h_2 : \forall(a : \alpha), p\ a \to b) : b$$

Now, `Exists.elim (hExists)` will result in type $(\forall(a : \alpha), p\ a \to b) \to b$. Where b still can be chosen freely. Then similarly as before 'apply' will set b to be $(\exists x, p\ x) \wedge r$, moreover h_2 has not been provided yet, hence this will be the next goal.

The next line of note is `have hxPx : ∃x, p x := ⟨ x, hPxAndR.left⟩`. The `have` keyword is a way to construct a term that can then be referred to by the name specified afterward. The name is then followed by the type of the term and the proof term is then constructed after `:=`.

The `⟨ x, hPxAndR.left⟩` is an example of using the anonymous constructor. When the type is clear from the context LEAN4 can infer how to construct a term from a certain type without having to write it out completely. Another place where it is used often is when constructing a term of type $a \wedge b$ with $a : Prop$ and $b : Prop$, here we can construct a term of that type with 'a,b', as long as LEAN4 knows what the desired type is.

In the last line we again use anonymous constructors in this case to create $(\exists x, p\ x) \wedge r$ from $hxPx : (\exists x, p\ x)$ and $hPxAndR.right : r$. The '.right' here retrieves the right part of the $\wedge$ inductive type. These last two lines could have been written into one line: `exact ⟨⟨x, hPxAndR.left⟩, hPxAndR.right⟩` since LEAN4 could have gotten that from context.

In the second code block we see `intro ⟨x, hPxAndR⟩`, here we use the pattern matching ability of LEAN4 again, it effectively does the same thing as:

```
intro hExists
match hExists with
| ⟨ x, hPxAndR ⟩ => ...
```

In `match` as well as with `Exists.elim` the automatically generated `recOn` is used under the hood.

Chapter 4

Karp's theorem

In this chapter, we derive Karp's theorem. Sections 4.1 and 4.2 introduce important background necessary for understanding the theorem. Then in Section 4.3 an example application of Karp's theorem is given to provide the reader with a more intuitive understanding what the assumptions mean and what the theorem implies. Section 4.4 discusses the relevance of Karp's theorem to the rest of the thesis. Section 4.5 will illustrate the differences in assumptions and the proofs themselves between Karp's theorem as presented in Tassarotti and Harper [44] and the version in this thesis. Finally, in Section 4.7 the proof for Karp's theorem is presented.

4.1 Execution time

When analysing algorithms, a key quantity of interest is the execution time, often denoted by W_n for an input of size n. Many algorithms solve a problem by recursively breaking it into smaller subproblems. This leads to a recurrence relation describing the execution time.

Let us briefly introduce Quicksort before using this as our example. Quicksort is a recursive sorting algorithm that works by selecting a pivot element, partitioning the array into elements smaller and larger than the pivot, and recursively applying the Quicksort algorithm on both the subarrays. For single single-element arrays, Quicksort returns just the array itself, ensuring the sorting algorithm will terminate. The execution time W_n, measured in the number of comparisons, of Quicksort satisfies the following recurrence relation:

$$W_n = W_U + W_{n-U-1} + C_n$$

Where U is the randomly chosen pivot, C_n is the number of comparisons needed to split the array into the two subarrays, W_U is the comparisons needed when applying Quicksort again on the left subarray (elements smaller than the pivot); and W_{n-U-1} is the number of comparisons needed to sort the right subarray (elements larger than the pivot). Then according to Tassarotti and Harper [44] this can be solved to obtain:

$$\mathbb{E}[W_n] = O(n \log n)$$

Where $O(n \log n)$ means there exists a constant $c \in \mathbb{R}$ such that the expected time $\mathbb{E}[W_n]$ is bounded by $c \cdot (n \log n)$, i.e.,

$$\mathbb{E}[W_n] \leq c \cdot (n \log n)$$

This expectation gives an average-case guarantee, but it does not quantify the likelihood of large deviations from the expected runtime.

4.2 Tail bounds

Tail bounds, such as Karp's theorem, help quantify these deviations. In Tassarotti and Harper [44], a bound for Quicksort is given:

$$\mathbb{P}(W_n > c_k n \log n) < \frac{1}{n^k}$$

Where W_n is the number of comparisons that are made when sorting a list with length n. And c_k is a scalar value depending on k such that the probability denoted on the left-hand side is smaller than $1/n^k$. This shows that the probability of exceeding $O(n \log n)$ time –and thus $O(n \log n)$ time– decreases polynomially with n.

4.3 Leader election algorithm

Let us introduce the setup of Karp's theorem and tail bounds before going into detail about its proof. We will do this by applying Karp's theorem to an example algorithm.

Suppose we have n candidates to elect into becoming a leader. Then the leader election algorithm will consist of sampling a Bernoulli for each active candidate with $p = \frac{1}{2}$. At the start all candidates are active. We terminate at the round where only 1 candidate is left. In the event no candidate is left, then that round's result gets ignored and we continue with the active candidates of the previous round. What we want to show with Karp's theorem is that the probability of the rounds going over a certain number get vanishingly small. Concretely, the authors of Tassarotti and Harper [44] were able to show:

$$\text{if } n > 1, \mathbb{P}(W_n > k \log(n) + 1 + w) \leq \left(\frac{3}{4}\right)^w$$

Where W_n is a random variable that represents the number of rounds that the leader election algorithm will take. n represents the number of candidates and w is an integer that can be chosen to be any non-negative value. k is furthermore defined as $k := \frac{1}{log(3/4)}$. In order to prove this statement we will first rewrite the statement to make Karp's theorem applicable:

$$\text{if } size(z) > 1, \mathbb{P}(W(z) > k \log(size(z)) + 1 + w) \leq \left(\frac{3}{4}\right)^w$$

Now, instead of keeping the input implicit we explicitly call the input z, which represents the indices of the still active candidates, and refer to their number as $size(z)$. In other

words, each candidate is initially assigned a unique index, and z denotes the set of indices containing all the candidates that are still active.

Let us introduce the functions a and h here. Let I be a finite set of inputs that can be given to the election algorithm. Then let $z : I$ denote an input that is contained in set I. Now let $\mathcal{D}(I)$ represent the set of all probability distributions on I. Then

$$h : (z : I) \to \mathcal{D}(I)$$

represents applying the leader election algorithm once, returning a distribution over the number of still active candidates. a represents a toll function, this represents the cost of a single iteration of an algorithm. In the case of leader election, it will be equal to 1 for every round except when a leader was already selected in which case nothing has been done so the toll would be 0. The toll function and the work $W(z)$ are related in that:

$$W(z) = a(z) + W(h(z))$$

Since W will represent the number of rounds, choosing $a(size(z))$ as the number one makes sense, since for each iteration of the algorithm, we go to another round. In Tassarotti and Harper [44] the authors defined a as:

$$a(x) = \begin{cases} 0 & \text{if } x \leq 1, \\ x - 1 & \text{if } 1 < x < 2, \\ 1 & \text{if } x \geq 2. \end{cases}$$

Since Karp's theorem defined in their paper requires a to be defined from $\mathbb{R}^{\geq 0}$ (the non-negative reals) to $\mathbb{R}$ and continuous. In this thesis, we drop the continuity assumption on a, since we do not use it anywhere. We will go into detail on the assumptions in Section 4.6. Broadly speaking, Karp's theorem can be summarized as follows: it involves two main steps.

- First, bound the expected size of the recursive subproblem by identifying a function m such that $\mathbb{E}[h(z)] \leq m(size(z))$.
- Then, determine a solution u that satisfies the deterministic recurrence relation:$u(x) \geq a(x) + u(m(x))$, for all $x : \mathbb{R}$.

Then Karp's theorem states that we get the following inequality:

$$\mathbb{P}(W(z) > u(size(z)) + wa(size(z))) \leq \left(\frac{m(size(z))}{size(z)}\right)^w$$

for all $w : \mathbb{Z}_{\geq 0}$. Intuitively $m(x)$ represents the expected size reduction after one iteration of the algorithm starting from problem of size x. For the leader election algorithm we have

$$m(x) = \begin{cases} \frac{3}{4}x & \text{if } x \geq 2, \\ 0 & \text{if } x < 2, \end{cases}$$

We derive the bound for $x \geq 2$ by recognizing that there are two mutually exclusive scenarios. The first consists of there being zero survivors after the the Bernoulli distributions have been sampled, the probability of this is:

$$P(X = 0) = (1 - p)^x = \left(\frac{1}{2}\right)^x$$

Where x are the number of candidates we started with before the Bernoulli sampling. X is a random variable representing the sum of the Bernoullis after sampling, i.e. the number of candidates after the Bernoulli sampling is done for each candidate. The second possibility is that there are one or more candidates left after the sampling has completed. The expected number of candidates in this case is:

$$\mathbb{E}[X] = x \cdot p = x \cdot \frac{1}{2}$$

Now by the law of total expectation we get:

$$\begin{aligned}
\mathbb{E}[size(h(x))] &= \mathbb{E}[X] + x \cdot \mathbb{P}(X = 0) \\
&= \frac{1}{2}x + \left(\frac{1}{2}\right)^x \\
&\leq \frac{1}{2}x + \frac{1}{4} \\
&= \frac{3}{4}x
\end{aligned}$$

Where in the second last line we used $x \geq 2$. For $x < 2$ we can only have that one candidate is left so a leader can be selected and the election is done. So the problem size can be thought of as zero.

$u(x)$ estimates the expected time for the algorithm to reduce the problem to a minimum size. In the case of the leader election algorithm this is defined as:

$$u(x) = \begin{cases} k \cdot \log(x) + 1 & \text{if } x > 1, \\ u_{min} & \text{if } x \leq 1, \end{cases}$$

u_{min} here is defined as $u(1)$ and represents the minimum value of u, which in this case is $u_{min} = 1$. The time or number or rounds estimate increases logarithmically as the input size increases linearly.

Two important terms we have not yet mentioned are sample spaces $\Omega_h(z)$ and $\Omega_W(z)$ which is where outcomes $w : \Omega_h(z)$ and $w : \Omega_W(z)$ live for distributions $W(z)$ and $h(z)$ respectively. Events such as $\{W(z) > u(size(z)) + w \cdot a(size(z))\} \subseteq \Omega_h(z)$ can also be denoted as

$$\{w : \Omega_h(z) | W(z)(w) > u(size(z)) + w \cdot a(size(z))\}$$

Moreover $\hat{h} : \mathbb{N} \times (z : I) \to \mathcal{D}(I)$, represents a distribution over input after applying h a number of times. $\hat{h}(n)(z)$ will give the distribution after applying h n times starting from $z : I$.

4.4 Relevance

Karps theorem is relevant to the rest of this thesis because it provides a probabilistic framework for analyzing recursive algorithms where the problem size shrinks in expectation after each iteration. So it can be applied to analyze the behavior of weights in Sequential Monte Carlo (Section 5.1.4), which gets executed in the SMX algorithm after each iteration of the E-step, see Algorithm 4 in Section 7.

The Sequential Monte Carlo method involves updating weights through importance weighting followed by a resampling step that samples in accordance to the weights the particles carry. Ensuring higher weight particles get copies and lower weight particles get discarded. Over successive iterations this process leads to weight concentration, where a single particle retains nearly all the weight, while the remaining particles obtain a negligible weight. The question Karp's theorem allows us to then answer is: how many iterations does it take for this concentration to occur?

We can analyze this process by drawing an analogy to the leader election algorithm. In leader election, the number of active candidates decreases at each round due to independent bernoulli trials, much like how the number of effectively active particles in SMC decreases after weighting and resampling. The leader election analysis showed showed that the number of rounds required for termination is logarithmic in the number of initial candidates, with exponentially vanishing probability of exceeding this bound.

Similarly, in SMC, the effective sample size (ESS)– which represents the number of significantly weighted particles– shrinks over iterations. The definition of ESSS depends on whether resampling is performed:

- With weight resampling, ESS is the number of particles originating from different "ancestors" that still have significant weight. When ESS drops to 1, this means all samples effectively originated from a single starting action. This process is similar to sample impoverishment, though it occurs gradually over multiple iterations rather than in a single resampling step. Sample impoverishment refers to the loss of diversity in a particle set due to resampling. When particles with small weights are eliminated and those with larger weights are duplicated, the number of particle lineages decreases. This can harm the approximation accuracy.

- Without weight resampling, ESS is simply the number of particles that retain significant weight. Here, termination of the algorithm implies that the posterior distribution estimate has collapsed to a single particle, a phenomenon known as weight degeneracy.

In either case, the problem size x (ESS) decreases over iterations, and Karp's theorem provides a tail bound on how quickly this occurs. By defining: $W(z)$ as the number of iterations until weight collapse, $h(z)$ as the transformation of particle weights after one iteration, $a(z)$ as the cost per iteration (set to 1 per step), $m(z)$ as the expected reduction in ESS after one iteration. We can then invoke Karp's theorem to derive an exponential tail bound on the number of iterations needed for weight concentration.

In particular, if the effective sample size decreases in expectation by a factor of ρ per iteration (analogous to the $\frac{3}{4}$ reduction in leader election), then the probability of the weight concentration takes significantly more that $O(\log n)$ iterations is exponentially small. This guarantees that, with high probability, SMC reaches weight collapse in logarithmic number of iterations.

Thus Karp's theorem provides a way to bound the number of SMC iterations before all particles trace back to an original ancestor. Moreover, it shows the necessity of resampling to prevent weight degeneracy when running SMC for more than $O(\log N)$ steps, with N being the number of particles.

4.5 Contribution

Compared to Tassarotti and Harper [44], we remove the continuity assumption on a, as it is not used anywhere in our proof. Moreover, we specialize Karp's theorem algorithms that take only a finite number of possible inputs I. Because I is finite, llema31 in the proof is also modified. Specifically, this lemma uses a summation instead of an integral, as in Tassarotti and Harper [44]. Additionally, the proofs are written out in detail. For instance, the different cases in the main theorem – Karp's theorem– are explicitly treated in this thesis, whereas they were left to the reader in Tassarotti and Harper [44]. Then the proving technique used in lemma lemma hD_nonDec is original, and the decision to state it as a separate lemma was not made Tassarotti and Harper [44]. However, it was done in the formal proof in the formal theorem proving language Coq written by the same authors [43].

4.6 Assumptions

Let us briefly describe each variable again for the general case:

- $z : I$: This denotes a single instance of an input, not a random variable. This type I is finite, which means it has only a finite number of terms, and it represents the type of all possible inputs.
- $W(z)$: This is a random variable that represents the distribution over work. The distribution is over the space $\Omega_W(x)$.
- $a : \mathbb{R}^{>0} \to \mathbb{R}^{>0}$: This represents a toll function
- $d : \mathbb{R}$: This variable represents the threshold. If the size of the input is smaller than d the algorithm will not execute any more recursive calls.
- $h : I \to \mathcal{D}(I)$: This represents a family of random variables, for a given $z : I$, $h(z)$ is a distribution over new problems. The distribution is over the space $\Omega_h(x)$.
- $\hat{h} : \mathbb{N} \to I \to \mathcal{D}(I)$: $\hat{h}(n)$ represents a distribution over input after applying h to it n times.

- $size : I \to \mathbb{R}^{>0}$, the size function will give a real number that represents the size of the input.
- $m : \mathbb{R} \to \mathbb{R}$, this must be a continuous function that will bound the expectation of a certain random variable. The constraints will be elaborated on in more depth after introducing the theorem.
- $u : \mathbb{R} \to \mathbb{R}$, this is the minimal solution to some recurrence relation, this will get clearer later on. It is also an upper bound on the expected work.
- $\hat{u} : \mathbb{R} \to \mathbb{R}$, this is the inverse of u which is also continuous.
- $u_{min} := u(d)$, this represents the upper bound on the expected work when the input size is d large.

In order to apply Karp's theorem certain conditions must be met on m, u and a. We will go through the assumptions in this section. On the toll function a we require that after a certain size d no more recursive calls are made and the toll should be zero we will name this assumption hCutOff. We furthermore require the toll function to be increasing, this assumption we call haIncreasing. Lastly, we need a to be strictly larger than zero whenever $x > d$, this is called hApos. The names along with the statements are:

- hCutOff:$\exists d, \forall r \leq d \to a(r) = 0$.
- haIncreasing: $\forall p \;\; q : \mathbb{R}, p \geq q \to a(p) \geq a(q)$
- hApos: $\forall x, d < x \to a(x) > 0$

As mentioned before Tassarotti and Harper [44] requires a to be continuous, but in this thesis, we can drop this assumption as we have not used this within our proof. The assumptions on m are of course that it must bound the average size of the problem after the algorithm is applied for one iteration. Moreover m must always be positive or equal to zero for all inputs $z : I$ and m evaluated on the size of z, $m(size(z))$, must be smaller than the size of z itself. We furthermore require $m(y)/y$ to be a non-decreasing function.

- hM: $\forall (z : I), \mathbb{E}[size(h(z))] \leq m(size(z)) \wedge 0 \leq m(size(z)) \leq size(z)$
- hMnonDec:$\forall y, y' : \mathbb{R}, 0 < y \to y < y' \to m(y)/y \leq m(y')/y'$

For the u we eventually select for our theorem we must have that it is monotone increasing, has an inverse $\hat{u}$ that is always positive and u must have a lower bound that is strictly smaller than $u(x)$ for all x larger than d in size. Lastly, it must satisfy a certain recurrent relation.

- UmonoInc: $\forall x \;\; y, x \leq y \to u(x) \leq u(y) \wedge \forall x \;\; y, x > d \to x < y \to u(x) < u(y)$
- U2pos:$\forall x, \hat{u}(x) > 0$, this means that $\hat{u}$ is a positive function.
- U2inv: $\forall x, u(\hat{u}(x)) = x \wedge \forall x, \hat{u}(u(x)) = x$
- UminLb: $\forall x, x > d \to u(x) > u_{min}$

- Urec: $\forall x, x > d \rightarrow u(x) \geq a(x) + u(m(x))$

For the work W we require that it must be non-negative for all $z : I$. We furthermore require that the probability $W(x) > r$ can only be as large as the sum of all the possible next states of the probability the next state is x' times the probability $W(x')$ exceeds the remaining time $r - a(size(x))$.

- hWgeq0: $\forall z, W(z) \geq 0$

- Wrec:$\forall(x : I)\forall(r : \mathbb{R}), \mathbb{P}(\{w \,|(W(x)(w)) > r\}) \leq \sum_{x' \in Img(h)} \mathbb{P}(\{w \,|h(x)(w) = x'\}) \cdot \mathbb{P}(\{w \,|(W(x)(w)) > r - a(size(x))\})$

The latter intuitively follows whenever these two events are set to be equal:

$$\{W(x) > r\} = \bigcup_{x'}(\{h(x) = x'\} \cap \{W(x') > r - a(size(x))\})$$

Then the union bound gives us $\mathbb{P}(W(x) > r) \leq \sum_{x'} \mathbb{P}(h(x) = x', W(x') > r - a(size(x)))$. Then using independence we get the desired inequality. Lastly for the work W, we have that for inputs x with $size(x) < d$ that the work is smaller $u(d)$, i.e. $W(x)(w) \leq u(d), \forall(w : \Omega_W(x)$ is smaller than $u(d)$.

- Wbelow: $\forall x \;\; w, size(x) \leq d \rightarrow W(x)(w) \leq u_{min}$

Now for $\hat{h}$ we also have a recurrent relationship. The recurrent relationship represents how the behavior of the algorithm over $i + 1$ steps can be decomposed:

- hRec: $\mathbb{P}(size(\hat{h}(i+1)(z) > d)) = \sum_{q \in \text{Img}(h(z))} \mathbb{P}(h(z) = q) \cdot \mathbb{P}(size(\hat{h}(i)(q)) > d)$

This assumption can be obtained through applying the total law of probability. We can partition the event $\{size(\hat{h}(i + 1)(z) > d\}$ on possible outcomes q of $h(z)$:

$$\mathbb{P}(size(\hat{h}(i + 1)(z)) > d) = \sum_{q} \mathbb{P}(h(z) = q) \cdot \mathbb{P}(size(\hat{h}(i + 1)(z)) > d | h(z) = q)$$

Now, if we know $h(z) = q$ then $\hat{h}(i + 1)(z) = \hat{h}(i)(q)$, so

$$\mathbb{P}(size(\hat{h}(i + 1)(z)) > d | h(z) = q) = \mathbb{P}(size(\hat{h}(i)(q)) > d)$$

Substituting this back gives us the desired assumption. The last assumption we will discuss is on $\hat{h}$ as well, and it boils down to that the algorithm will eventually stop in a finite number of iterations:

- hPFinite: $\exists k, \mathbb{P}(size(\hat{h}(k)(z)) > d) = 0$

4.7 Proof of Karp's theorem

Let us first rewrite the theorem into a more convenient form. Let K be defined as:

$$\begin{aligned} K(r,z) &= \mathbb{P}(\{w : \Omega_W(z) | W(z)(w) > r\}) \\ &= \mathbb{P}(W(z) > r) \end{aligned}$$

Where r represents the time threshold. We are concerned with the probability that the running time of the algorithm $W(z)$ exceeds r. z represents the initial state of the algorithm from which we start measuring the running time or work $W(z)$. In the case of the leader election algorithm the state contains the number of active participants or candidates remaining in the election and current round number. Then let us define D as

$$D(r,x) = \begin{cases} 1 & \text{if } r \leq u_{min} \\ 0 & \text{if } r > u_{min} \wedge x \leq d \\ 1 & \text{if } r > u_{min} \wedge r \leq u(x) \wedge x > d \\ \left(\frac{m(x)}{x}\right)^{\lceil \frac{r-u(x)}{a(x)} \rceil} \cdot \frac{x}{\hat{u}\left(r - a(x) \cdot \lceil \frac{r-u(x)}{a(x)} \rceil\right)} & \text{otherwise} \end{cases}$$

Then if we prove the following:

$$K(r,z) \leq D(r,x)$$

Karp's theorem will follow after substitution. Now, the reason why Karp's theorem follows has to do with how D is defined. Substituting r for $u(size(z)) + w \cdot a(size(z))$ gives us the desired form of Karp's theorem. The form of D is specifically chosen to make proving easier while still being able to support the substitution. Let us go through the cases of D to explain why using D we will not interfere with proving the original Karp's theorem bound. When $r \leq u_{min}$, u_{min} is the minimal possible running time of the algorithm for input z with $size(z) > d$. So in this case the algorithm cannot finish in less than u_{min} time, the probability that $W(z) > r$ is 1. Now, for inputs z' with the size $size(z') \leq d$ Wbelow states that $W(z') < u(d) = m_{min}$ so the probability of $W(z') > r$ is not necessarily 1. However, 1 is still a valid upper bound in this case.

When $r > u_{min}$ and $size(z) \leq d$, then the problem size is small enough so that the algorithm is finished before u_{min} (Encoded in Wbelow). Therefore it also finished before r which comes after u_{min}. Hence $W(z) > r$ has probability 0. Thus $D(r, size(z)) = 0$ is a valid upper bound in this case.

For the case that $r > u_{min}$, $r \leq u(size(z))$ and $size(z) > d$, we know that the problem size is larger than threshold d. So the algorithm is unlikely to complete before the expected running time $u(size(z))$, so we set $D(r, size(z)) = 1$ to represent this high probability that $W(z) > r$.

The case $size(z) > d$ and $r > u(size(z))$ represents the problem size being large and that we consider times beyond the expected running time respectively. For the sake of brevity let $x := size(z)$. $\frac{m(x)}{x}$ represents the expected reduction factor after 1 iteration.

$\lceil \frac{r-u(x)}{a(x)} \rceil$ calculates the number of additional steps beyond $u(x)$ required to reach time r. $a(x)$ is the additional time per recursive step.

$$\left(\frac{m(x)}{x}\right)^{\lceil \frac{r-u(x)}{a(x)} \rceil}$$

models the exponential decrease in the probability as the number of additional steps increases. There is also a correction term to the right of the exponentially decreasing term in the fourth case of D, how this came to be will become clear in the proof. Karp's theorem formally can be written down as:

Theorem 4.7.1 (Karp's theorem)**.** *Let I be a finite set of all possible inputs to the algorithm, let $m : \mathbb{R} \to \mathbb{R}$ a function that satisfies hM and hMnonDec, furthermore let $u : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ a function that satisfies UmonoInc U2pos U2inv UminLB and Urec. Let a be the toll function that satisfies hCutoff haIncreasing and hApos, let $r \in \mathbb{R}$ and $z \in I$. Then*

$$\mathbb{P}(W(z) > u(size(z)) + w \cdot a(size(z)) \leq \left(\frac{m(size(z))}{size(z)}\right)^w$$

for all $w \in \mathbb{Z}^{\geq 0}$

We will now establish Lemmas that will support us in proving the main theorem.

Lemma 4.7.2 (K_proto_unfold)**.** *For any $r \in \mathbb{R}$ and $z \in I$,*

$$K(r, z) \leq \sum_{i \in Img(h(z))} \mathbb{P}(h(z) = i) \cdot K(r - a(size(z)), i)$$

Proof. This proof follows trivially by unfolding the definition of K and applying Wrec to r and z. □

Lemma 4.7.3 (K0)**.** *For any $r \in \mathbb{R}$ and $z \in I$, if $r < 0$, then*

$$K(r, z) = 1.$$

Proof. First, recall the definition of K:

$$K(r, z) = \mathbb{P}\left(W(z) > r\right).$$

Given that $r < 0$, we need to show:

$$\mathbb{P}\left(W(z) > r\right) = 1.$$

We know that for any x, $W(x)) \geq 0$, by the non-negativity of W (hWgeq0). Therefore, since $r < 0$, it follows that $r < (W(z))$ for all z.

Hence we can establish that $\forall z, \forall w : \Omega_W(z), w \in \{w : \Omega_W(z) | W(z)(w) > r\}$. Thus

Next, consider the probability expression:

$$\mathbb{P}(\{w | W(z)(w) > r\}) = \mathbb{P}(\{w : \Omega_W(z)\})$$

Since the sum of the probabilities in $\Omega_W(z)$ is 1, we obtain

$$\mathbb{P}(\{w|W(z)(w) > r\}) = 1$$

Therefore, $K(r, z) = 1$. □

Lemma 4.7.4 (K_unfold). *For any $r \in \mathbb{R}$ and $x \in I$,*

$$K(r, x) \leq \mathbb{E}_{h(x)}\left[K(r - a(\mathit{size}(x)), \cdot)\right]$$

Proof. We start this proof by unfolding the definition of expectation making our goal

$$K(r, z) \leq \sum_{x' \in Img(h)} \mathbb{P}(\{w|h(z)(w) = x'\}) \cdot K(r - a(size(z)), i)$$

And the proof of this follows directly from the application of Lemma K_proto_unfold. □

We will now define three more concepts that will help us bound K.

$$K_{\text{rec}}(n, r, x) = \begin{cases} K_0(r, x) & \text{if } n = 0 \\ \mathbb{E}_{w \sim h(x)}\left[K_{\text{rec}}\left(n - 1, r - a(\text{size}(x)), w\right)\right] & \text{if } n > 0 \end{cases}$$

Where

$$K_0(r, x) = \begin{cases} 1 & \text{if } r < u_{min} \\ 0 & \text{otherwise} \end{cases}$$

We will use $K_{\text{rec}}(n, r, x)$ as a recursive definition that will bound K Let I be a set, $r \in \mathbb{R}$, and $x \in I$. Define the function Krec_sup(r, x) as follows:

$$\begin{aligned} \text{Krec\_sup}(r, x) &= \sup\left\{v : \overline{\mathbb{R}} \mid \exists n \in \mathbb{N},\ K_{\text{rec}}(n, r, x) = v\right\} \\ &= \sup_i K_{\text{rec}}(i, r, x) \end{aligned}$$

Where $\overline{\mathbb{R}}$ is the extended real numbers. We will show that K is bounded by the supremum. And we will show by induction on i that D bounds each of the $K(i, \cdot, \cdot)'s$.

Let us first establish the following lemma

Lemma 4.7.5 (Krec_non_dec). *Let I be a set of inputs, $r \in \mathbb{R}$, and $x \in I$. Suppose $\mathit{size}(x) \leq d$. For all $i \in \mathbb{N}$, the following inequality holds:*

$$K_{rec}(i, r, x) \leq K_{rec}(i + 1, r, x),$$

Proof. We will proceed using induction on i.

Base Case ($i = 0$)

We need to show that for $r \in \mathbb{R}$ and $x \in I$ we get

$$K_{\text{rec}}(0, r, x) \leq K_{\text{rec}}(1, r, x),$$

Unfolding the definition of K_{rec} here we obtain

$$\begin{aligned} K_0(r, x) &\leq K_{\text{rec}}(1, r, x) \\ &= \mathbb{E}_{h(x)}\left[K_{\text{rec}}\left(0, r - a(\text{size}(x)), \cdot\right)\right] \\ &= \sum_{x:\text{Img}(h(z))} \mathbb{P}(h(z) = x) K_0(r - a(\text{size}(x)), x) \end{aligned}$$

Now K_0 actually does not depend on x, hence we can sum out the probability and we obtain

$$K_0(r, x) \leq K_0(r - a(\text{size}(x)), x)$$

This holds trivially looking at the definition of K_0.

Inductive Step

Assume the lemma holds for some $i : \mathbb{N}$. We need to show it holds for $i + 1$.

$$K_{\text{rec}}(i + 1, r, x) \geq K_{\text{rec}}(i, r, x),$$

We will start with unfolding $K_{\text{rec}}(i + 1, r, x)$ and then applying the induction hypothesis, we then obtain

$$\begin{aligned} K_{\text{rec}}(i + 1, r, x) &= \mathbb{E}_{h(x)}\left[K_{\text{rec}}\left(i, r - a(\text{size}(x)), \cdot\right)\right] \\ &= \sum_{x:\text{Img}(h(z))} \mathbb{P}(h(z) = x) K_{\text{rec}}(i, r - a(\text{size}(x)), x) \\ &\leq \sum_{x:\text{Img}(h(z))} \mathbb{P}(h(z) = x) K_{\text{rec}}(i - 1, r - a(\text{size}(x)), x) \\ &= \mathbb{E}_{h(x)}\left[K_{\text{rec}}\left(i - 1, r - a(\text{size}(x)), \cdot\right)\right] \\ &= K_{\text{rec}}(i, r, x) \end{aligned}$$

Proving the induction step as well, completing the induction. □

Lemma 4.7.6 (K_Krec_leq_below)**.** *Let I be a set, $r \in \mathbb{R}$, and $x \in I$. Suppose size$(x) \leq d$. For all $i \in \mathbb{N}$, the following inequality holds:*

$$K(r, x) \leq K_{rec}(i, r, x),$$

Proof. We again proceed by induction on i.

Base Case ($i = 0$)

We need to show that for $r \in \mathbb{R}$ and $x \in I$ with size$(x) \leq d$,

$$K(r, x) \leq K_0(r, x).$$

Consider two cases:

Case $r < u_{min}$:

By the definition of K_0,

$$K_0(r, x) = 1.$$

We need to show:

$$\mathbb{P}\left(\{n \in \Omega_W(x) \mid r < W(x)(n)\}\right) \leq 1.$$

This is trivially true since probabilities are always less than or equal to 1.

Case $r \geq u_{min}$:

Since size$(x) \leq d$, we have $(W(x)(w)) \leq u_{min}$ for all $w \in \Omega_T(x)$ by the hypothesis Wbelow.

Therefore,

$$\Pr\left(\{w \in \Omega_W(x) \mid r < (W(x)(w))\}\right) = 0.$$

We need to show:

$$0 \leq K_0(r, x).$$

By the definition of K_0,

$$K_0(r, x) = 0 \quad \text{if } r \geq u_{min}.$$

Hence,

$$0 \leq K_0(r, x).$$

Inductive Step

Assume the lemma holds for some $i : \mathbb{N}$. We need to show it holds for $i + 1$. So the goal is to show:

$$K(r, x) \leq K_{\text{rec}}(i + 1, r, x).$$

Now with the induction hypothesis we obtain

$$K(r, x) \leq K_{\text{rec}}(i, r, x).$$

Now using transitivity and lemma Krec_non_dec we get the desired proposition.

This completes the induction.

□

Lemma 4.7.7 (KrecLtK_imp_PropPos)**.** *Let I be a set, $r \in \mathbb{R}$,$i \in \mathbb{N}$, and $x \in I$. Suppose $K_{rec}(i)(r)(x) \leq K(r)(x)$. Then have the following:*

$$\mathbb{P}(size(h(i)(z) > r) > 0,$$

Proof. We will once again proceed by induction on i.

Base Case ($i = 0$)

Suppose $K_{\text{rec}}(i)(r)(z) < K(r)(z)$ holds. Then we must show $\mathbb{P}(\{w : size(h(0)(z)(w)) > d\}) > 0$. Now by our assumption h0z we have $\forall w : \Omega_h(z), h(0)(z)(w) = z$. Hence this is equivalent to showing $size(z) > d$.

We will prove this by contradiction, suppose $size(z) \leq d$, then by lemma K_Krec_leq_below we know $K(r)(z) \leq K(0)(r)(z)$, which contradicts the assumption we made at the start. Completing the base case.

Inductive Step

Assume the lemma holds for some $i : \mathbb{N}$. Then we need to show it holds for $i + 1$. Our goal becomes:

$$K_{\text{rec}}(i + 1)(r)(z) \leq K(r)(z) \to \mathbb{P}(size(h(i)(z)) > d) > 0$$

Let us assume $K_{\text{rec}}(i + 1)(r)(z) \leq K(r)(z)$ holds. Then applying lemma K_unfold and unfolding the definition of K_{rec} we obtain:

$$\mathbb{E}_h\left[K(i)(r - a(size(z)), \cdot)\right] \leq \mathbb{E}_h\left[K(i)(r - a(size(z)), \cdot)\right]$$

unfolding the definition of expectation we obtain:

$$\sum_{y \in \text{Img}(h(z))} \mathbb{P}(\{w | h(z)(w) = y\})K(i)(r - a(size(z)))(y) \leq \sum_{y \in \text{Img}(h(z))} \mathbb{P}(\{w | h(z)(w) = y\})K(r - a(size(z)))(y)$$

Now, we have

$$\exists y, \mathbb{P}(\{w | h(z)(w) = y\}) > 0 \wedge K_{\text{rec}}(i)(r - a(size(z))(y) \leq K(r - a(size(z)))(y)$$

This we obtain by contradiction, since if the proposition would have been false the left hand side would be larger than the right hand side contradicting the $\leq$ relation. Now by the assumption hRec we have

$$\mathbb{P}(size(h(i + 1)(z) > d)) = \sum_{q \in \text{Img}(h(z))} \mathbb{P}(h(z) = q) \cdot \mathbb{P}(size(h(i)(q)) > d)$$

Now we also have $K_{rec}(i)(r)(z) \leq K(r)(z)$, using transitivity together with $K_{\text{rec}}(i+1)(r)(z) \leq K(r)(z)$ and lemma Krec_non_dec. Composing this with the induction hypothesis we obtain $\mathbb{P}(size(h(i)(y)) > d) > 0$. Hence we have

$$\mathbb{P}(size(h(i+1)(z)) > d) > \mathbb{P}(h(z) = y) \cdot \mathbb{P}(size(h(i)(y)) > d) > 0$$

Using $\mathbb{P}(\{w|h(z)(w) = y\}) > 0$ and $\mathbb{P}(size(h(i))(y) > d) > 0$. Completing the inductive step. Hence we have proven the proof statement using induction. □

We will now show that $\forall r\ \ z, K(r)(z) \leq \sup_i K(i)(r)(z)$.

Lemma 4.7.8 (K_le_supKrec)**.** *Let I be a set, $r \in \mathbb{R}$,$i \in \mathbb{N}$, and $z \in I$. Then have the following:*

$$K(r)(z) \leq \sup_i K(i)(r)(z)$$

Proof. We will prove this by contradiction, so we obtain the hypothesis:

$$\exists r\ \ x, \sup_i K(i)(r)(x) < K(r)(x)$$

So take $r\ \ x$ such that $\sup_i K(i)(r)(x) < K(r)(x)$ holds then unfolding the definition of the supremum we obtain that

$$\forall i, K(i)(r)(x) < K(r)(x)$$

Hence using lemma KrecLtK_imp_PropPos with $r\ \ x$ we obtain

$$\forall i, \mathbb{P}(size(h(i)(z) > d) > 0$$

However we have assumed with hPFinite that $\exists k, \mathbb{P}(size(h(k)(z) > d) = 0$. This leads to a contradiction, hence we have proven the lemma by contradiction. □

We will now prove that $D(r)(x)/x$ is non-decreasing for $0 < x < \hat{u}(r)$ where $\hat{u}(r)$ is the inverse as mentioned in the assumptions U2inv). This will then be used to apply lemma31 (which we will prove later on) to obtain the inequality

$$\mathbb{E}[\text{size}(h(z))]\frac{D(r - a(\text{size}(z)))\ (\min(\text{size}(z), \hat{u}(r - a(\text{size}(z)))))}{\min(\text{size}(z), \hat{u}(r - a(\text{size}(z))))}$$

We will then use this to show that the $K(i)(\cdot)(\cdot)$'s are all bounded by D.

Lemma 4.7.9 (hD_nonDec)**.** *Let $r \in \mathbb{R}$,$i \in \mathbb{N}$, and $x \in I$. Then for $0 < x \leq \hat{u}(r)$, $D(r)(x)/x$ is non-decreasing.*

Proof. We have $D(r)(x)/x = 0$ for $x \leq d \wedge D(r)(x)/x > 0$ for $x > d$. By unfolding the definition of $D(r)$ hence we know that for $x \in (0, d]$ that D is non-decreasing.
For the case $x \in [d, \hat{u}(r)]$ we define $S(r)(u)(k)$ as

$$S(r)(u)(k) := \{x \in [d, \hat{u}(r)]\ \ |\ \ k = \lceil (r - u(x))/a(x) \rceil\}$$

Now we can trivially show a lemma Sunique: $\forall x, \exists! k, x \in S(r)(u)(k)$. Furthermore we can trivially show lemma XLeY_imp_KLeL: $\forall x\ y, x < y \to \forall k, x \in S(r)(u)(k) \wedge \forall l, y \in S(r)(u)(k) \to k \leq l$. This can be proven by first realizing using Sunique that the $\forall k$ and $\forall l$ refer to only one unique k and l respectively, and using UmonoInc to show that for $x < y$ we would have $k < l$ whenever both k and l are nonnegative (otherwise $\lceil \cdot \rceil$ will not preserve inequality). And lastly by using U2inv we know $(r > u(x))$ and $(r > u(y))$, hence we can establish k and l are both nonnegative.

We start with the goal:

$$x \leq y \to D(r)(x)/x \leq D(r)(y)/y$$

Let us introduce the hypothesis that $x \leq y$ and unfold the definitions. Then we see the goal becoming:

$$\left(\frac{m(x)}{x}\right)^{\lceil r-u(x)/a(x)\rceil} \frac{1}{\hat{u}(r - \lceil r - u(x)/a(x)\rceil\, a(y))} < \left(\frac{m(y)}{y}\right)^{\lceil r-u(y)/a(y)\rceil} \frac{1}{\hat{u}(r - \lceil r - u(y)/a(y)\rceil\, a(y))}$$

Using lemma Sunique we can obtain k, l thus giving us

$$\left(\frac{m(x)}{x}\right)^{k} \frac{1}{\hat{u}(r - ka(x))} < \left(\frac{m(y)}{y}\right)^{l} \frac{1}{\hat{u}(r - la(y))}$$

Moving $\hat{u}$ to the other side we obtain:

$$\left(\frac{m(x)}{x}\right)^{k} < \left(\frac{m(y)}{y}\right)^{l} \frac{\hat{u}(r - ka(x))}{\hat{u}(r - la(y))} \leq \left(\frac{m(y)}{y}\right)^{l}$$

Where we used, $\hat{u}(r - ka(x)) \geq \hat{u}(r - la(y))$, by $r - ka(x) > r - la(y)$ (where we used Amono) and U2Pos to make sure division by $\hat{u}$ preserves the inequality.

We will split the proof now up into two cases

Case $\frac{m(x)}{x} \geq 1$:

$$\left(\frac{m(x)}{x}\right)^{k} \leq \left(\frac{m(y)}{y}\right)^{k} \quad \text{(by Mnondec)}$$
$$\leq \left(\frac{m(y)}{y}\right)^{l} \quad \text{(by } k \leq l\text{)}$$

Case $\frac{m(x)}{x} < 1$:

$$\begin{aligned}\left(\frac{m(x)}{x}\right)^k &\leq \left(\frac{m(x)}{x}\right)^l \quad \text{(by } k \leq l \text{ and } \frac{m(x)}{x} < 1\text{)} \\ &\leq \left(\frac{m(y)}{y}\right)^l \quad \text{(by Mnondec)}\end{aligned}$$

Thus we solve the desired goal for both cases. Therefore we have proven the lemma. □

Lemma 4.7.10 (lemma31)**.** *Let X be a random variable with its domain being of type Ω. Where Ω is a type of which there are only finitely many values that are of that type. Let x be the maximum value that is taken for all values on Ω. Let $f : \mathbb{R} \to \mathbb{R}_{>0}$ be a nonnegative function such that $f(0) = 0$. And suppose that there exists some constant c such that for $y \geq c$, $f(y) = 1$ and $f(y)/y$ is non-decreasing on $(0, c]$. We furthermore assume $y \geq 0$ and $c \geq 0$. Then we have*

$$\mathbb{E}_X[f(\cdot)] \leq \frac{\mathbb{E}[X]f(\min(x, c))}{\min(x, c)}$$

Proof. We will start by showing

$$f(y) \leq \frac{y \cdot f(\min(x, c))}{\min(x, c)}$$

We start by splitting on cases $y = 0 \vee y > 0$:

Case y=0

for $y = 0$ we know $f(y) = 0$ using this we obtain the goal

$$0 \leq 0\frac{f(\min(x, c))}{\min(x, c)}$$

simplifying this we obtain our goal.

Case $y > 0$

We will again split into two cases $x \leq c \vee x > c$:

Case $x \leq c$

Then we get the goal

$$f(y)/y \leq f(x)/x$$

By moving y to the left hand side and simplifying $\min$. This goal can then be solved by $f(x)/x$ being nondecreasing.

Case $x > c$

Then once more we will split into the cases $y \leq c \vee y > c$:
case $y \leq c \implies$ moving y to the lefthand side of the goal and simplifying $\min(x, c)$ using $x > c$ we obtain the goal

$$f(y)/y \leq f(c)/c$$

This then follows from $f(y)/y$ being nondecreasing and $y \leq c$.
case $y > c \implies$

$$\begin{aligned} f(y) &= 1 \quad (\text{by } \forall y \geq c, f(y) = 1 \text{ and } y \geq c) \\ &= f(c) \quad (\text{by } \forall y \geq c, f(c) = 1 \text{ and } c \geq c) \\ &\leq f(c) \cdot \frac{y}{c} \quad (\text{by } y/c > 1, \text{ using nonnegativity of } y \text{ and } c \text{ to preserve inequality}) \end{aligned}$$

no more goals are unsolved hence this statement is proven. Now we will move on to the main goal:

$$\mathbb{E}[f(X)] \leq \frac{\mathbb{E}[X] \cdot f(\min(x, c))}{\min(x, c)}$$

$$\begin{aligned} \mathbb{E}[f(X)] &= \sum_{y \in \text{Img}(X)} \mathbb{P}(\{w : \Omega \,|X(w) = y\}) \cdot f(y) \\ &\leq \sum_{y \in \text{Img}(X)} \mathbb{P}(\{w : \Omega \,|X(w) = y\}) \, \frac{y \cdot f(\min(x, c))}{\min(x, c)} \\ &\leq \frac{f(\min(x, c))}{\min(x, c)} \sum_{y \in \text{Img}(X)} \mathbb{P}(\{w : \Omega \,|X(w) = y\}) \, y \\ &\leq \frac{f(\min(x, c))}{\min(x, c)} \mathbb{E}[X] \end{aligned}$$

Where in the second line we used the statement proven before moving on to the main goal. □

Theorem 4.7.11 (Karp's theorem)**.** *Let $i \in \mathbb{N}$, let $m : \mathbb{R} \to \mathbb{R}$ a function that satisfies hM, $r \in \mathbb{R}$ and $x \in I$. Then*

$$K_{rec}(i)(r)(x) \leq D(r)(size(x))$$

Proof. We will prove this statement by induction on i

Base case $i = 0$:

For $i = 0$, we must show

$$K_{\text{rec}}(0)(r)(x) \leq D(r)(size(x))$$

unfolding the definition of K_{rec} we obtain

$$K_0(r)(x) \leq D(r)(size(x))$$

Let us split this into two cases $r \leq u_{min} \vee r > u_{min}$

Case $r < u_{min}$:

Simplifying both sides using $r \leq u_{min}$ we obtain

$$0 \leq 1$$

which holds trivially.

Case $r \geq u_{min}$:

simplifying using the definition of K_0 and $r \geq u_{min}$ gives us

$$0 \leq D(r)(x)$$

This holds trivially as well since D is a nonnegative function. By m and $\hat{u}$ are positive functions.

Induction step:

For the induction step let us suppose that for some $k : \mathbb{N}$ we have

$$\forall r \;\; x, K_{\text{rec}}(k)(r)(x) \leq K(r)(x)$$

Lets introduce r and x, then the goal becomes

$$K_{\text{rec}}(k+1)(r)(x) \leq D(r)(x)$$

Now we have

$$\begin{aligned}
K_{\text{rec}}(i+1)(r)(x) &= \mathbb{E}_{h(x)}[K(i)(r - a(size(x)))(\cdot)] \quad \text{(by unfolding)} \\
&\leq \mathbb{E}_{h(x)}[D_{r-a(size(x))}(\cdot)] \quad \text{(by induction hypothesis)} \\
&\leq \mathbb{E}_{h(x)}[size(\cdot)] \cdot \frac{D(r - a(size(x)))(\min(size(x), \hat{u}(r - a(size(x))))}{\min(size(x), \hat{u}(r - a(size(x))))} \\
&\leq m(size(x)) \cdot \frac{D(r - a(size(x)))(\min(size(x), \hat{u}(r - a(size(x))))}{\min(size(x), \hat{u}(r - a(size(x))))}
\end{aligned}$$

In the third line, we used lemma31 and the fourth line uses that $m : \mathbb{R} \to \mathbb{R}$ satisfies hM. Hence we obtain the inequality:

$$K_{\text{rec}}(i+1)(r)(x) \leq m(size(x)) \cdot \frac{D(r - a(size(x)))(\min(size(x), \hat{u}(r - a(size(x))))}{\min(size(x), \hat{u}(r - a(size(x))))}$$

Now let $q := \min(size(x), \hat{u}(r - a(size(x))))$ and $\hat{r} := r - a(size(x))$. Then the inequality becomes:

$$K_{\text{rec}}(i+1)(r)(x) \leq m(size(x)) \cdot \frac{D(\hat{r})(q)}{q}$$

Let us denote this inequality as hKleqMD. Let us start the proof by splitting into two cases $r \leq u_{min} \vee r > u_{min}$

case $r \leq u_{min} \implies$ For $r \leq u_{min}$, we have $D(r)(size(x)) = 1$ we furthermore know $\forall i\ r\ x, K(i)(r)(x) \leq 1$, this can be proven inductively. Hence the goal is satisfied for $r \leq u_{min}$.
case $r > u_{min} \implies$ For $r > u_{min}$, we can split it up further into $size(x) \leq d \vee (size(x) > d \wedge u(size(x)) \geq r) \vee (size(x) > d \wedge u(size(x)) < r)$

case $size(x) \leq d \implies$ in this case we have $D(r)(x) = 0$ applying hKleqMD we get the following goal:

$$m(size(x)) \cdot \frac{D(\hat{r})(q)}{q} \leq 0$$

Now, $q \leq size(x)$. Thus we have $D(\hat{r})(q)$. Now

$$\begin{aligned} \hat{r} &= r - a(size(x)) \quad \text{(by definition)} \\ &= r - 0 \quad \text{(by Alower with } size(x) \leq d\text{)} \\ &= r \end{aligned}$$

Hence we have $D(\hat{r})(q) = 0$, due to $\hat{r} = r \geq u_{min}$ and due to $q \leq size(x) \leq d$. Hence rewriting $D(\hat{r})(q) = 0$ into the goal we obtain:

$$m(size(x)) \cdot \frac{0}{q} \leq 0$$

simplifying this will give us the desired term.
case $size(x) > d \wedge u(size(x)) \geq r \implies$ simplifying $D(r)(size(x))$ we would get

$$K_{\text{rec}}(i+1)(r)(x) \leq 1$$

Now, as mentioned before, we can use induction to show $\forall i\ r\ x, K(i)(r)(x) \leq 1$. Which would solve this case when applying it for $(i+1)$, r and x.
case $size(x) > d \wedge u(size(x)) < r \implies$ This part is the hardest part of the proof we will again split it into two cases $size(x) \leq \hat{u}(r - a(size(x))) \vee size(x) > \hat{u}(r - a(size(x)))$:

Case $size(x) \leq \hat{u}(r - a(size(x)))$

Then $q = size(x)$ Now let us establish two equalities

$$\begin{aligned}
&\frac{\hat{r} - u(q)}{a(q)} \\
&= \frac{r - a(q) - u(q)}{a(q)} \\
&= \frac{r - u(q)}{a(q)} - 1
\end{aligned}$$

Let us denote this equality with hEq1. And

$$\begin{aligned}
&\hat{r} - a(q) \left(\left\lceil \frac{\hat{r} - u(q)}{a(q)} \right\rceil \right) \\
&= r - a(q) - a(q) \cdot \left(\left\lceil \frac{r - u(q)}{a(q)} \right\rceil - 1 \right) \\
&\text{(by hEq1 and } \lceil \cdot - 1 \rceil = \lceil \cdot \rceil - 1\text{)} \\
&= r - a(size(x)) \cdot \left(\left\lceil \frac{r - u(size(x))}{a(size(x))} \right\rceil \right) \quad \text{(by ring axioms)}
\end{aligned}$$

Let us denote this equality as hEq2. Now applying hKleqMD we obtain:

$$m(size(x)) \cdot \frac{D(\hat{r})(q)}{q} \leq D(r)(size(x))$$

Now simplifying D using $size(x) > d \wedge u(size(x)) < r$ and $r > u_{min}$ we obtain the expression

$$D(r)(size(x)) = \left(\frac{m(q)}{q} \right)^{\left\lceil \frac{r - u(q)}{a(q)} \right\rceil} \frac{q}{\hat{u}\left(r - a(q) \cdot \left(\left\lceil \frac{r - u(q)}{a(q)} \right\rceil \right) \right)}$$

Where we used $q = size(x)$. Let us name this expression hdEq. Now we can establish that $\hat{r} > u_{min}$, since we have that $size(x) < \hat{u}(r - a(size(x)))$. Hence, $u(size(x)) < r - a(size(x))$, by U2inv. Hence

$$\begin{aligned}
\hat{r} &= r - a(size(x)) \quad \text{(by definition)} \\
&> u(size(x)) \\
&\geq u_{min} \quad \text{(by UminLb)}
\end{aligned}$$

Now, we furthermore know that $u(size(x)) < \hat{r}$, from the inequalities we have established above. And we know $size(x) > d$. Hence we can simplify $D(\hat{r})(size(x))$ within the goal:

$$m(size(x))\frac{1}{q} \cdot \left(\frac{m(q)}{q} \right)^{\left\lceil \frac{\hat{r} - u(q)}{a(q)} \right\rceil} \frac{size(x)}{\hat{u}\left(\hat{r} - a(q) \left\lceil \frac{\hat{r} - u(q)}{a(q)} \right\rceil \right)} \leq D(r)(size(x))$$

Now,

$$
\begin{aligned}
& m(size(x))\frac{1}{q}\cdot\left(\frac{m(q)}{q}\right)^{\left\lceil\frac{\hat{r}-u(q)}{a(q)}\right\rceil}\frac{size(x)}{\hat{u}\left(\hat{r}-a(q)\left\lceil\frac{\hat{r}-u(q)}{a(q)}\right\rceil\right)} \\
=& m(q)\frac{1}{q}\cdot\left(\frac{m(q)}{q}\right)^{\left\lceil\frac{\hat{r}-u(q)}{a(q)}\right\rceil}\frac{q}{\hat{u}\left(\hat{r}-a(q)\left\lceil\frac{\hat{r}-u(q)}{a(q)}\right\rceil\right)} \\
=& m(q)\frac{1}{q}\cdot\left(\frac{m(q)}{q}\right)^{\left\lceil\frac{r-u(q)}{a(q)}\right\rceil-1}\cdot\frac{q}{\hat{u}\left(\hat{r}-a(q)\left\lceil\frac{\hat{r}-u(q)}{a(q)}\right\rceil\right)} \quad \text{(by hEq1)} \\
=& m(q)\frac{1}{q}\cdot\left(\frac{m(q)}{q}\right)^{\left\lceil\frac{r-u(q)}{a(q)}\right\rceil-1}\cdot\frac{q}{\hat{u}\left(r-a(q)\cdot\left(\left\lceil\frac{r-u(q)}{a(q)}\right\rceil\right)\right)} \quad \text{(by hEq2)} \\
=& \left(\frac{m(q)}{q}\right)^{\left\lceil\frac{r-u(q)}{a(q)}\right\rceil}\cdot\frac{q}{\hat{u}\left(r-a(q)\cdot\left(\left\lceil\frac{r-u(q)}{a(q)}\right\rceil\right)\right)} \quad \text{(by ring axioms)} \\
=& D(r)(q) \quad \text{(by hdEq)} \\
\leq& D(r)(q)
\end{aligned}
$$

Applying this to our goal we indeed prove the goal:

$$m(size(x))\cdot\frac{D(\hat{r})(q)}{q}\leq D(r)(size(x))$$

giving us the proof for this case.

Case $size(x) > \hat{u}(r - a(size(x)))$

Now let $z := size(x)$. In this case we can again simplify D using $size(x) > d, u(size(x)) < r$ and $r > u_{min}$. Giving us the equality hdEq:

$$D(r)(z)=\left(\frac{m(z)}{z}\right)^{\left\lceil\frac{r-u(z)}{a(z)}\right\rceil}\frac{z}{\hat{u}\left(r-a(z)\left\lceil\frac{z-u(z)}{a(z)}\right\rceil\right)}$$

Furthermore

$$
\begin{aligned}
u(q) &= u(\hat{u}(r-a(z)) \\
&= r-a(z) \quad \text{(by U2inv)} \\
&= \hat{r} \quad \text{(by definition of } \hat{r}) \\
&\geq \hat{r}
\end{aligned}
$$

hence we know that $D(\hat{r})(q) = 1$, since we can assume without loss of generality that $q > d$, so both for $\hat{r} \leq u_{min}$ and $\hat{r} > u_{min}$ we obtain $D(\hat{r})(q) = 1$. We can assume $q > d$,

since we otherwise obtain $D(\hat{r})(q) = 0$ after which our goal becomes trivial after applying hKleqMD. Hence our goal becomes after applying hKleqMD:

$$\frac{m(z)}{q} \leq D(r)(z)$$

Now, $r - u(z) > 0$ and

$$\begin{aligned} u(z) &> r - a(z) \\ u(z) - r &> -a(z) \\ a(z) &> r - u(z) \end{aligned}$$

Furthermore we know $a(z) > 0$, since $z > d$ and using hApos. Hence we know by dividing by $a(z)$ on both sides:

$$0 < \frac{r - u(z)}{a(z)} < 1$$

Therefore we can establish

$$\left\lceil \frac{r - u(z)}{a(z)} \right\rceil = 1$$

We can then use this to simplify the equality hdEq to obtain:

$$D(r)(z) = \frac{m(z)}{\hat{u}\,(r - a(z))}$$

Using this we can now rewrite our goal:

$$\frac{m(z)}{q} \leq \frac{m(z)}{\hat{u}\,(r - a(z))}$$

Which holds by equality using the definition of q together with $size(x) > \hat{u}(r - a(size(x)))$. Hence we have shown that the induction step holds for all cases. Proving this theorem. □

Chapter 5

SMX: Sequential Monte Carlo Planning for Expert Iteration

5.1 Background: Markov Decision Processes (MDPs)

Sequential decision-making is often formalized as a Markov Decision Process (MDP), defined as a tuple $(S, \mathcal{A}, \mathcal{T}, r, \gamma, \mu)$. Here S is the set of possible states, representing the different configurations the system can be in. $\mathcal{A}$ is the set of actions available to the decision-maker at each state. $\mathcal{T} : S \times \mathcal{A} \to \mathcal{P}(S)$ is the transition function, describing the probability distribution over the next states given a current state and action. $r : S \times \mathcal{A} \to \mathbb{R}$ is the reward function, quantifying the desirability of the state-action pairs. $\gamma \in [0, 1]$ is the discount factor, determining the importance of future rewards compared to immediate rewards. Lastly, μ is the initial state distribution, representing the likelihood of stating in each possible state.

The objective is to find a policy $\pi : S \to \mathcal{P}(\mathcal{A})$, which maps each state to a probability distribution over actions, maximizing the expected cumulative reward

$$\pi^* = \arg\max_{\pi \in \Pi} \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \right]$$

where $r_t = r(s_t, a_t)$ is the reward received at time t. In the context of this thesis, the MDP framework models reasoning steps in formal theorem proving, where states correspond to a sequence of Lean 4 tactics (predefined commands or operations used to manipulate the proof goals) and the resulting proof goals (intermediate objectives in the proof). Actions represent Lean 4 tactics that can be applied to a given state.

5.1.1 Control as Inference

The authors Macfarlane et al. [27] then suggest to connect MDPs with the SMX algorithm by reformulating the above objective, also called the reinforcement learning objective as a probabilistic inference problem. Instead of maximizing the cumulative reward directly, we introduce an “optimality variable" O_t, defined such that:

$$p(O_t = 1|\tau) \propto \exp(\gamma^t r_t)$$

where $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$ is a trajectory, representing a path through the Markov Decision Process. This allows us to frame the goal as finding the target distribution over trajectories given optimality at all timesteps, $p(\tau|O_{1:T})$. The policy optimization problem becomes [28]:

$$\log p_\pi(O = 1) = \log \int \pi(\tau) p(O = 1|\tau) d\tau$$

where $p(O = 1)$ represents the probability of acting optimally across all timesteps. Notice also that $p(O = 1|\tau)$ factors as:

$$p(O = 1|\tau) = \prod_{t=0}^{T} p(O_t|s_t, a_t) \propto \exp\left(\sum_t \gamma^t r_t\right)$$

5.1.2 Evidence Lower Bound (ELBO)

Since directly optimizing this objective is difficult, the evidence lower bound (ELBO) is used as a tractable approximation. By introducing an auxiliary distribution $q(\tau)$, the ELBO is expressed as:

$$\log p_\pi(O = 1) \geq \mathbb{E}_{q(\tau)}\left[\sum_t \gamma^t \frac{r_t}{\alpha} - \log \frac{q(\tau)}{\pi(\tau)}\right]$$

Here, $q(\tau)$ is an auxiliary distribution that simplifies optimization by allowing us to compute the ELBO through sampling, instead of directly computing the intractable integral from the previous section. Furthermore, α is a normalization constant. This lower bound is the foundation of the SMX method, it combines the Sequential Monte Carlo (SMC) sampling with Expectation-Maximization (EM) to iteratively improve the policy. SMC sampling is a probabilistic technique that approximates distributions by generating and resampling particles (samples) to represent the target distribution effectively. EM is an iterative optimization algorithm that alternates between estimating hidden variables (E-step) and optimizing model parameters (M-step) to maximize a likelihood function. The derivation of this bound can be found in the appendix B

5.1.3 Expectation maximization

In the case of the SMX algorithm, the E-step corresponds to generating auxiliary distribution q by evaluating the policy using sampled trajectories. This q will be generated in such a way as to optimize the ELBO shown above. The M-step updates the policy π to align with the auxiliary distribution q.

5.1.4 Sequential Monte Carlo

Sequential Monte Carlo (SMC) sampling is used to approximate the target distribution over trajectories, which in this context represents the probability distribution of reasoning paths that align with logical correctness and relevance. SMC maintains a set of weighted particles, where each particle represents a possible trajectory. At each step, the importance weights of the particles are updated based on how well they align with the target distribution.

The target distribution, denoted as $p(x)$, is often intractable to sample from directly. SMC solves this by using a tractable proposal distribution that is denoted by Macfarlane et al. [27] as $\beta(x_t|x_{1:t-1})$, which provides an approximation for sampling trajectories incrementally. Each trajectory is weighted according to how well it represents the target distribution relative to the proposal distribution. The set of weighted particles $\{x_t^{(n)}, w_t^{(n)}\}_{n=1}^N$ approximates the target distribution as:

$$p(x) \approx \sum_{n=1}^{N} w_t^{(n)} \delta_{x_t^{(n)}}(x)$$

where $\delta_{x_t^{(n)}}(x)$ is a Dirac delta function centered at $x_t^{(n)}$, and $w_t^{(n)}$ are the importance weights of the particles. The importance weights are updated at each step using Sequential Importance Sampling (SIS), the weights are computed as:

$$\tilde{w}_t^{(n)}(x_{1:t}) = w_{t-1}(x_{1:t-1}) \cdot \frac{p(x_t|x_{1:t-1})}{\beta(x_t|x_{1:t-1})}$$

and

$$w_t^{(n)} = \frac{\tilde{w}_t^{(n)}}{\sum_{k=1}^{N} \tilde{w}_t^{(k)}}$$

for each particle $n = 1, \ldots, N$. The normalization here makes sure the weights remain interpretable as probability, and reduces numerical instability from very large or very small weights.

To prevent the loss of diversity and maintain an accurate approximation of the target distribution, computational resources are reallocated through the resampling process. During resampling, particles with low-importance weights are discarded, while particles with high weights are duplicated. This makes sure that the particle set remains focused on regions of high probability in the target distribution while avoiding issues like sample impoverishment, which occurs when only a small subset of particles dominate the weight distribution, reducing diversity in the sampled trajectories.

5.2 SMX

SMX combines Sequential Monte Carlo (SMC) sampling with Expectation-Maximization framework for MDPs. We iteratively refine the auxiliary distribution q in the E-step, and update the policy π to align with the auxiliary distribution in the M-step.

5.2.1 KL-divergence

Shortly we will discuss the derived E-step, which will contain two new terms: the KL divergence and Q values. So before we introduce this step we will introduce the KL divergence. The KL stands for the Kullback-Leibler divergence. This is a measure of how one probability distribution q differs from another distribution π.The KL divergence between q and π

is given by:

$$\mathrm{KL}(q||\pi) = \sum_{\tau} q(\tau) \log \frac{q(\tau)}{\pi(\tau)}$$

This formula represents the expected logarithmic difference between $q(\tau)$ and $\pi(\tau)$, with the expectation taken under $q(\tau)$. A smaller KL divergence indicates that q and π are more similar, and the divergence is zero if $q(\tau) = \pi(\tau)$ for all τ.

5.2.2 Q values

The second new term that will show up is the Q value function. The Q-function, denoted as $Q^q(s, a)$, represents the expected cumulative reward when starting in state s, taking action a, and then following a policy defined by the distribution q for subsequent steps. Mathematically, it is expressed as:

$$Q^q(s, a) = \mathbb{E}_{\tau \sim q} \left[\sum_{t=0}^{T} \gamma^t r_t \,\middle|\, s_0 = s, a_0 = a \right],$$

where as stated before $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$ is a trajectory sampled according to the policy q and $\gamma \in [0, 1]$ is the discount factor, lastly r_t is the reward obtained at time-step t. Actions in the context of this thesis would consist of LEAN4 tactics. Intuitively, $Q^q(s, a)$ answers, what the expected total reward would be if we start at state s, take action a, and then follow the policy q.

5.2.3 E-step

The authors of Macfarlane et al. [27] state that the E-step optimization can be written as:

$$\max_q \mathbb{E}_{\mu(s)}[\mathbb{E}_{q(a|s)}[Q^q(s, a)]] - \alpha \mathrm{KL}(q(\cdot|s)||\pi(\cdot|s, \theta))]$$

derived from the ELBO bound. However, our derivation in Appendix B.1 shows that, starting from the ELBO bound, we arrive at:

$$\max_q \mathbb{E}_{\mu(s)} \left[\mathbb{E}_{q(a|s)} \left[\frac{Q^q(s, a)}{\alpha} \right] \right] - (T + 1) \mathbb{E}_{d_q(s)} \left[\mathbb{E}_{q(a|s)} \left[\mathrm{KL}(q(a|s)||\pi(a|s)) \right] \right].$$

Our derivation of the E-step differs in constants such as α and $(T + 1)$ and the definition of the expectation specifically, we use $d_q(s)$, the time-averaged state visitation under q, whereas Macfarlane et al. [27] use $\mu(s)$, the initial state distribution. However, since α and $(T + 1)$ do not affect the maximization itself, our bound is effectively equivalent to theirs.

It is worth noting that Macfarlane et al. [27] do not provide the full derivation of their E-step objective. Because of this, their assumptions on the MDP remain unclear. In this thesis, we present a rigorous derivation to ensure the derivation is verifiable and transparent. The differences will likely be caused by definitional choices, such as whether the MDP uses a finite horizon (with a maximum time T) or an infinite horizon (where $T = \infty$), as well as the form of the state-visitation distribution.

State visitation distributions describe the frequency with which states are visited under a given policy. There are two types of state visitation distributions. The time-averaged state visitation $d_q(s)$ reflects the proportion of time spent in each state over the trajectory, whereas the discounted state visitation emphasizes states visited earlier by weighting visits using the discount factor γ. The time-averaged state visitation is usually used for finite-horizon problems and the discounted state visitation is usually used for infinite-horizon problems.

Additionally, Macfarlane et al. [27] simplifies notation by using μ to represent both the state visitation and the initial state distribution interchangeably. While this naming choice is convenient, it can make differences in the assumptions used less visible.

Macfarlane et al. [27] then applies a common simplification, where the Q-values are fixed to represent the action-values of a fixed policy $\bar{\pi}$. For the fixed policy we use the most recent estimate of q. The authors propose using a queue of states that are visited most recently to determine $\mu_{\bar{\pi}}$. Macfarlane et al. [27] then also reformulates the problem as:

$$\max_q \mathbb{E}_{\mu_{\bar{\pi}}(s)}[\mathbb{E}_{q(a|s)}[Q^{\bar{\pi}}(s,a)]]$$
$$\text{s.t. } \mathbb{E}_{\mu_{\bar{\pi}}}[\text{KL}(q(a|s)||\pi(a|s,\theta_i)] < \varepsilon$$

However, Macfarlane et al. [27] also label $\mu_{\bar{\pi}}$ as an approximation of the *initial* state distribution. Since the queue samples states from all time steps $\mu_{\bar{\pi}}$ more closely approximates the overall state-visitation distribution $d_{\bar{\pi}}$. This matches the $\mathbb{E}_{d_q(s)}[\ldots]$ term in our E-step derivation, but it does not match our $\mathbb{E}_{\mu(s)}[\ldots]$ term, which theoretically uses only initial states. Nevertheless, using $\mu_{\bar{\pi}}$ as an approximation for both expectations is common in reinforcement learning, thus we will do this in this thesis as well.

Macfarlane et al. [27] then solved this expression using Lagrangian multipliers, obtaining:

$$q_i(a|s) \propto \pi(a|s,\theta_i) \exp\left(\frac{Q^{\bar{\pi}}(s,a) - V^{\bar{\pi}}(s)}{\eta^*}\right)$$

Here η^* is a normalizing constant that has to be solved for using a convex dual function. This is out of scope for this thesis, however.

Here, $V^{\bar{\pi}}(s)$ represents the value function, which quantifies the expected cumulative reward when starting in state s and following policy $\bar{\pi}$ for all actions up to that point. It is related to the Q-function as:

$$V^{\pi}(s) = \mathbb{E}_{a\sim\bar{\pi}(a|s)}[Q^{\bar{\pi}}(s,a)]$$

Intuitively, it represents the overall value that can be obtained by applying policy $\bar{\pi}$.

Estimating Target Distribution

As established in the above section the E-step is optimized by a target distribution of the form:

$$q_i(a|s) \propto \pi(a|s,\theta_i) \exp\left(\frac{Q^{\bar{\pi}}(s,a) - V^{\bar{\pi}}}{\eta^*}\right)$$

This is a distribution that is not tractable to sample from immediately. For the SMX algorithm, this is done using Sequential Monte Carlo sampling as we discussed earlier in this chapter. We will specifically apply Sequential Importance Sampling, to do this we need to select p and β, for which we select the following:

$$p_i(\tau_t|\tau_{1:t-1}) \propto \mathcal{T}(s_{t+1}|s_t, a_t)\pi_i(a_t|s_t, \theta_i) \exp\left(\frac{A^{\bar{\pi}}(a_t, s_t)}{\eta_i^*}\right)$$
$$\beta(\tau_t|\tau_{1:t-1}) \propto \mathcal{T}(s_{t+1}|s_t, a_t)\pi_i(a_t|s_t, \theta_i)$$

Where we define $A^{\bar{\pi}}(a_t, s_t) = Q^{\bar{\pi}}(s, a) - V^{\bar{pi}}(s)$, this is also called the advantage, since we look at what the total reward would be if the agent exclusively does a specific action a in state s, when following policy $\bar{\pi}$ compared to the total reward that would have been obtained in state s if the agent just followed policy $\bar{\pi}$.

The weight updates then become [27]:

$$w(\tau_{1:t}) \propto w(\tau_{1:t-1}) \cdot \exp\left(A^{\bar{\pi}}(a_t, s_t)/\eta_i^*\right)$$

Macfarlane et al. [27] visualized this with Figure 5.1.

In figure 5.1 we can see that we start at the starting state S_t. In our case this would be the starting proof state when we have not yet applied any tactics. We can then see N edges coming out of this, giving us N different next states. These new states are obtained by doing a_t^i, with $i = 1, \ldots, N$, actions sampled according to $\pi_\theta(a|s_t)$, i.e. $a_t^i \sim \pi_\theta(a|s_t)$, for $i = 1, \ldots, N$

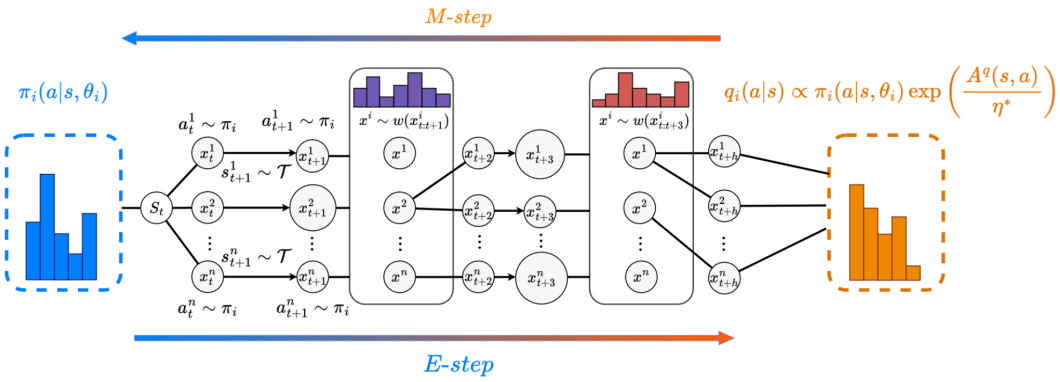

Figure 5.1: This figure visually represents the SMX search process, where multiple trajectory samples (denoted as $x^i, \ldots, x^n$) are generated in parallel using a policy π_i. During each step of the environment, particle weights are adjusted based on the advantage, as indicated by the size of the circles. The resampling stages are depicted where particles with higher weights are more likely to be prioritized for propagation, and the weights are reset to be uniform afterward. The final step involves estimating the target distribution q_i, which is constructed by the particles that are left in the final stage and their corresponding weights. This target distribution is then used to refine policy π in the M-step. Adapted from *SPO: Sequential Monte Carlo Policy Optimization* by Macfarlane et al. (2024) [27].

5.2.4 M-step

Since in this thesis we were unable to also implement the M-step and instead chose for supervised fine-tuning we will not validate and derive the M-step in this thesis. This will instead be left as future work. Below we show the M-step as it was derived by Macfarlane et al. [27]:

$$\max_{\theta} \mathbb{E}_{\mu_{q_i}(s)} \left[\mathbb{E}_{q_i(a|s)} \left[\log \pi(a|s, \theta) \right] \right]$$
$$\text{s.t. } \mathbb{E}_{\mu_{q_i}(s)} \left[\mathbf{KL} \left(\pi(a|s, \theta_i), \pi(a|s, \theta) \right) \right] < \varepsilon_m$$

Chapter 6

Related Work

In this chapter, we review existing approaches related to this thesis, focusing on methods that use large language models (LLMs) for reasoning and theorem proving. We discuss how Monte Carlo Tree Search (MCTS) and other proof-guided reasoning techniques have been applied to improve the performance of LLMs in structured reasoning tasks. Additionally, we review alternative approaches like Hypertree proof search and Logic-LM, which aim to achieve faithful logical reasoning through symbolic solvers.

6.1 MCTS guided reasoning in LLMs

Monte Carlo Tree Search (MCTS) is a decision-making algorithm widely used in reinforcement learning. It combines random sampling of future actions with systematic exploration of high-reward paths to build a search tree. MCTS iteratively improves its decision-making by simulating outcomes, updating the values of visited states, and balancing exploration and exploitation. MCTS is a valuable tool for guiding reasoning processes, particularly in scenarios requiring sequential decision making, such as theorem proving.

In the work of Feng et al. [12], MCTS is proposed as a method for guiding reasoning in LLMs during both training and inference. The authors mostly look at smaller LLM's with sizes ranging from 150 Million parameters to 7 Billion parameters. Due to this smaller size the authors suggests that the model will not have a good internal critic ability–that is, the capacity to evaluate the quality of intermediate states. To address this, they recommend using separate parameters for evaluating a states and for generating an action.

For computational efficiency, the authors suggest that using a shared model architecture with different heads for the policy and the value functions. In this setup, the policy model (responsible for generating actions) and the critic model (responsible for evaluating states) share the majority of the model's parameters, differing on only the final layers, known as heads. Each head is trained for its specific purpose, policy generation or value evaluation. This would be optimal for compute efficiency.

6.2 Neural theorem proving

Theorem proving involves generating formal proofs for mathematical or logical statements through a sequence of reasoning steps, called tactics. Recent advances have explored using large language models (LLMs) to assist with or automate theorem proving. Traditional approaches rely on proof assistants, which require manually writing down the tactics by a human, while the described methods use autoregressively trained Transformer models trained on text to suggest tactics instead of letting humans come up with suggestions. The work of Tian et al. [46] is similar to Feng et al. [12], however here the models are larger and no separation is made between the value and policy network. The larger model does both. Some critical differences are that the authors try to alleviate 3 main problems

- Imperfect feedback: The authors use multiple (3) critic models
- Search efficiency: By lumping together tokens into a single action, furthermore a fast rollout policy π^{fast} is used instead to traverse the MCTS tree.
- Data scarcity: After the MCTS stage the author collects the trajectory with the highest reward from the critic models as a training example to improve the model.

In our case the solution for imperfect feedback will come from using a proof assistant instead of using multiple critic models.

6.2.1 Tree Guided Neural Theorem Proving

The model in [33] is trained using two objectives

- Proofstep objective: This is a conditional language modeling objective. Where the model is asked to generate the PROOFSTEP given the GOAL [33]
- Outcome objective: Using this objective a value function is supposed to be implemented. It is supposed to represent the likelihood a proof gets solved.

The model architecture is a GPT-2 type of model with at its largest only 774 Million parameters. Similar to what we do in this paper is they make use of an expert-like principle, where the data that is produced during training, specifically the subgoals and their proofs are added as training data for the next iteration of the model. The work of [34] has a very similar strategy but here instead of providing a large starting dataset, the authors propose a way to automatically generate and manually curate a dataset without proofs that the model will try to solve, again using expert iteration. Furthermore, LEAN was used instead of metamath as the proof assistant. Lastly, the model is trained using two different objectives

- Proofstep objective: Similar to [33] the language model is conditioned on the theorem name and the goal state.
- Proofsize objective: Unlike [33] the authors propose to have as its second objective a conditional language modeling objective where the task is generate tokens that represent a proof size estimate bucket for the current goal. So instead of representing the

likelihood a proof gets solved, it represents the number of tactics that are needed to solve the goal.

6.2.2 Hypertree proof search guided theorem proving

In the work [23] the authors develop something called hypertree proof search. This proof search progressively grows a hypergraph starting from the goal state g. The process of the proof search consists of three steps:

- Selection: The authors point out that the number of nodes in the proof hypergraph grows exponentially with the distance to the root. Making breath first search a bad candidate if we want to find complicated proofs. Since these would require deep proofs (proofs that require many steps). Hence similar to MCTS the authors use a policy model P_θ together with current estimates from the critic c_θ. The critic here represents the probability that the current goal gets solved.

- Expansion: To expand a node g, the authors use a policy P_θ to sample tactic. These are sampled token-by-token using a transformer decoder model, where the authors condition the transformer on the goal and its previously generated tokens. The authors add a hyperedge for each valid tactic t_i from the expanded node g to its children $\{g_i^0, \ldots, g_i^k\}$ which are the subgoals that result after applying tactic t_i.

- Back-propagation: For each goal g in the simulated proof tree T, the value is set to $v_T(g) = 1$ when it is solved (no more subgoals) or $v_T(g) = 0$ when the node is invalid (error state in the proof assistant). If it is neither its value is estimated by the critic model $v_T(g) = c_\theta(g)$. This can be used to give a value to all leaves in T. Then the authors use the following formula to estimate the value of the inner nodes:

$$v_T(g) = \prod_{c \in \text{children}(g,t)} v_T(c)$$

  Where t is the tactic hyperedge connecting the children with the parent, that was selected in the expansion step. These are accumulated over multiple graph traversals. And eventually used to get a total action value $Q(g, t)$. The process is detailed in Figure 6.1, see [23] for more information on the exact details of this method.

In Figure 6.1 the visit count N the total accumulated values W are mentioned, however in our short description of the method this is left out for sake of brevity. If the reader wants to have a better understanding of the interplay of N, W and the critic c_θ the reader is urged to take a look at the paper [23].

6.3 Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning

In Pan et al. [30] the authors take a very different approach, where instead of trying to let the model validate its reasoning using a proof assistant. The problem gets phrased in a formal

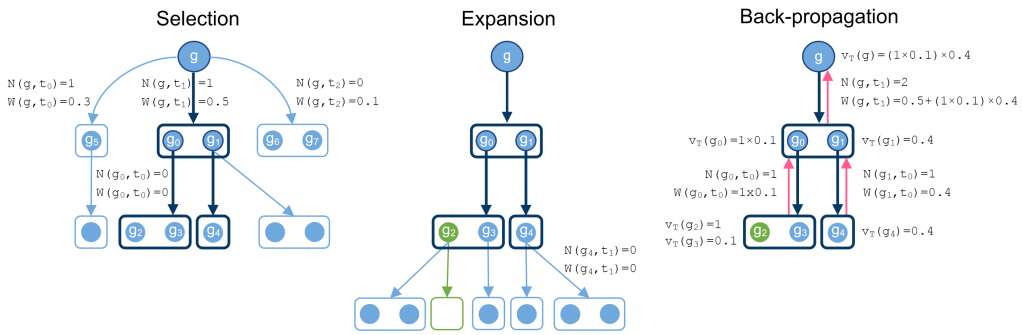

Figure 6.1: This figure is adapted from [23]. It describes finding a proof of the root theorem g. Either proving $\{g_5\}$, $\{g_0, g_1\}$ or $\{g_6, g_7\}$, will lead to a proof of g. Tactic t_1 is selected for expansion. The calculation of values can be seen in the backpropagation step, through backpropagating to the root and using the critic.

way such that it can be put inside a symbolic solver. In Figure 6.2 an overview is made of the Logic-LM model.

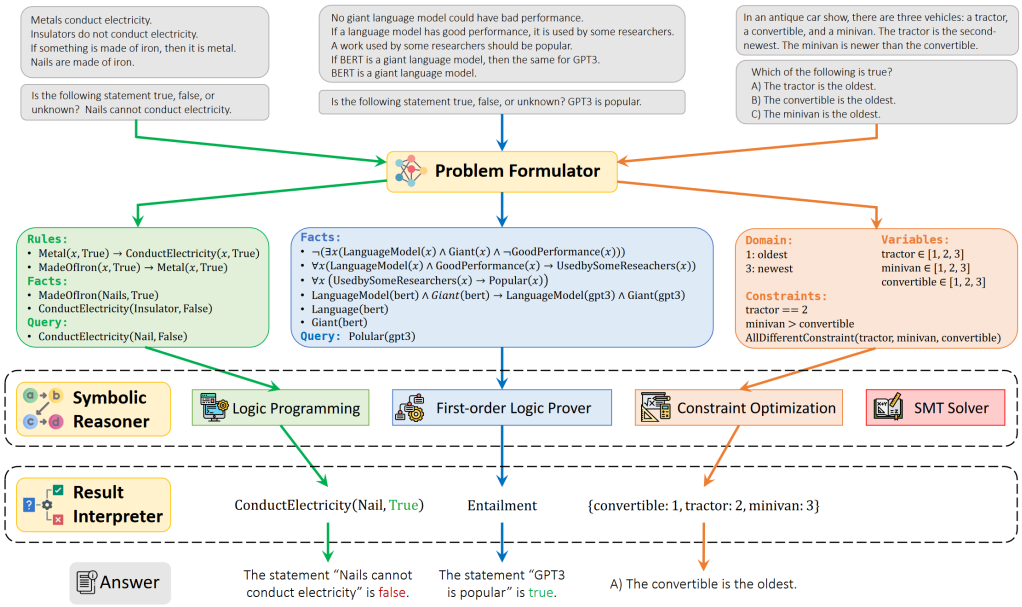

Figure 6.2: This figure is adapted from [30] The model consists of three modules. (1) The problem formulator, this essentially consists of prompting an LLM to generate a symbolic representation of the input. (2) The second module is called the Symbolic Reasoner. This effectively consists of inputting the problem into a symbolic solver. (3) The third step is executed by the Result interpreter which essentially interprets the symbolic answer.

Chapter 7

Methods

This chapter explains how Neural Theorem Proving is done with LLMs using SMX. The three main ingredients for Neural Theorem Proving are the neural network, which in our case is a Large Language Model; the inference procedure, which is the E-step in this thesis; and the improvement procedure of the neural network, which ideally would be the M-step but, in this thesis, will just be supervised fine-tuning. Section 7.1 introduces the problem setting and specifies the performance metrics for evaluation, and Section 7.2 explains the overall approach. Then, Section 7.3 explains the inference procedure that is used to solve the formal theorem-proving problem. Additionally, Section 7.4 explains the improvement step of the neural network that is used to improve the neurally parameterized policy for selecting tactics. For Neural network inference and modeling PyTorch [31] and `llama.cpp` were used.

7.1 Problem Setting

This thesis considers the problem of theorem proving, which has been cast as a Markov Decision Process (Section 5.1) with a transition model of the environment. We will go over the various components of the Markov Decision Process, how they are instantiated, and why it is instantiated in this way.

7.1.1 States

The states s_t are internal states of the LEAN4 Interface, consisting of the current LEAN4 proof script (i.e., the sequence of tactic applications written so far), along with the remaining proof goals and their associated local contexts (assumptions available for each goal). The textual representation of this is shown in Figure 7.1 in the Periwinkle Gray-colored rectangle on the left. When this information is fed to the generative model, it is transformed into this textual representation. The textual description also includes an Affirmation Prompt and examples alongside the already-mentioned textual information. This format was chosen because of the effectiveness of a similar type of prompt in Chen et al. [7] in guiding a generative model into giving better answers.

7.1.2 Actions

The actions a_t are `json` formatted strings:

```
{"Tactic":"<tactic>","Goal":"<GoalName>"}
```

The tactics are always targeted towards a particular goal that is referenced in the textual representation of the state s_t. The reason for choosing a `json` format in particular is because this structure is enforceable by `llama.cpp`, which makes parsing the output tractable.

Most commonly used LEAN4 tactics are supported, including `apply`, `exact`, `simp`, and `linarith`. These tactics allow the model to use lemmas and theorems from Mathlib to progress a proof, as well as to apply constructors of inductive types, such as `Or.inl` and `Or.inr` for disjunctions. However, tactics that require explicit case-splitting using the `| ... => ...` syntax, such as `match` and `induction`, are not supported.

This limitation was also present in Yang et al. [53], whose LEAN4 environment restricted the set of usable tactics. However, because we developed our own LEAN4 interface, we were able to extend support to the `have` tactic, allowing the model to introduce intermediate lemmas, and the `cases'` tactic, which enables explicit case handling in proofs. This extension improves the models ability to express proofs more naturally, as these tactics frequently appear in human-written Lean4 proofs. This is discussed in more detailed in Section 7.2.1, with examples illustrating how these additional tactics improve the expressiveness of the model.

The reason why actions are individual tactics instead of fragments of text is that the generative model chosen was LLaMA-3.1, which frequently outputs LEAN3 instead of LEAN4, which renders this strategy less feasible. Furthermore, when considering a Markov Decision Process, it is natural to let tactics correspond with actions since both represent steps taken towards a goal state.

Xin et al. [52] did go the route of defining actions as pieces of LEAN4 text. The authors would let the generative model continue generating new tokens until it generates the end-of-text token, which signals the generative model is finished generating text. They then truncate the outputted text to the first error that is returned by their LEAN4 interface.

The reason why their strategy would not have worked in this thesis is, in order to truncate to the first error a modification has to be made to the LEAN4 REPL as also described in Xin et al. [52]. This would be out of scope for a master's thesis, especially alongside the theoretical part of the thesis.

7.1.3 Transition Model

The transition function $\mathcal{T}$ is in the case of this thesis, a deterministic function that represents how the proof states change after applying an action/tactic. The LEAN4 Interface functions as the transition model. It maintains the state consisting of the code, and the unsolved goal names with their corresponding goal type and context.

7.1.4 Rewards

The reward function r_t uses the generative model alongside a prompt derived from the state s_t. The prompt can be seen in Figure 7.1 in the Fog (light blue) colored rectangle on the right. This prompt, similar to the prompt derived from the states s_t, was inspired by the prompt shown in Chen et al. [7]. It starts with a similar affirmation, after which a description of the bucket options is given. The bucket options are how the generative model will need to categorize the proof goals based on the estimated size of their proofs (i.e., the number of tactics necessary). The classification can be seen in Table 7.1.

Bucket	Description
A	Goals that have an infinite proof size (i.e., they are unprovable).
B	Goals with a proof size larger than 20 steps.
C	The third largest proof size among the categorized goals.
D	The fourth largest proof size among the categorized goals.
E	Goals with a proof size larger than the median.
F	Goals with a proof size approximately equal to the median.
G	Goals with a proof size smaller than the median.
H	The fourth smallest proof size among the categorized goals.
I	The third smallest proof size among the categorized goals.
J	The second smallest proof size among the categorized goals.
K	The smallest proof size among the categorized goals.

Table 7.1: Proof Size Categorization Buckets

The buckets are then transformed to rewards, with K being 10 and A being 0, and every bucket in between linearly interpolating between the two. The number of tactics is related to the buckets linearly as well:

$$b = \frac{10 \cdot (n_{tacs} - 1)}{20 - 1}$$

Where b is the bucket index, if n_{tacs} is larger than 20, we take the minimum of whatever n_{tacs} is, and 20. Furthermore, n_{tacs} can never be smaller than 1.

The output of the reward prompt conditioned generative model is also restricted to be `json` formatted:

```
{"Bucket":"<Bucket letter>"}
```

In Polu and Sutskever [33] the reward was tied to the likelihood that the generative model would be able to output the generated tactic; however, in their second paper Polu et al. [34], the authors used these buckets that estimate the difficulty of the proof goals. Using

the latter approach gave a better performance, so we opted for this approach as well. As the authors already described in their paper, the reason why we want the generative model to output buckets instead of the precise number of tactics is because for proofs it is often hard to predict how many tactics are still needed to prove a particular theorem beforehand, so making the target buckets instead of precise numerical estimates makes it easier for the model to learn since it cannot get penalized for small deviations in the predicted number of steps.

The difference between our approach and that of Polu and Sutskever [33] is that in their case instead of producing the reward by running inference on the generative model, they look at logits of all the valid bucket letters, normalize them since they do not sum to one if you ignore the probability of tokens that are not valid, and takes a weighted sum obtaining the following reward:

$$v(g) = \frac{1}{\#B} \sum_{b \in B} p_b(g) \cdot b$$

Where $\#B$ are the number of buckets, so in this thesis it would be 10; B are the bucket indices, so since we go from A to K this would go from 0 (for A) to 10 (for K); and $p_b(g)$ would for each b be the likelihood that token b is given as the continuation. The reason why in this thesis this approach is not used is that `llama.cpp` does not provide a way to obtain the logits of the next token prediction conditioned on a certain prompt. It only provides a way to quickly use inference techniques (e.g., top-p and temperature scaling) to return a certain generated text, with the option to add certain restrictions on how the output is structured.

Another approach implemented by Xin et al. [52] would have been to make the final reward binary, with 0 for unverified proofs and 1 for verified proofs. This would result in rewards becoming very sparse, so the authors introduce an intrinsic reward whenever the action introduces a new state that has not been seen before. In this thesis, we did not opt for this approach, since in the E-step with Sequential Monte Carlo we only have a limited number of particles, and we want to maximize the number of useful particles; thus, a reward purely based on exploration would not work in our case.

7.1.5 Policy function

The policy function $\pi_\theta(a_t|s_t)$, like the reward function, consists of the generative model alongside the prompt that can be seen in Figure 7.1 as mentioned already in Section 7.1.1. The generative model itself is a Large Language Model, the particular model chosen for this thesis is LlaMA-3.2-8b. This model was chosen because it was the newest and best model at the time for a reasonable inference/training cost. The 70-billion parameter version performed significantly better in the preliminary analysis; however, this would have been too computationally expensive to fine-tune.

The sampling from this policy model is done using top-p and temperature scaling, see Section 2.10. The grammar that the model can output is furthermore restricted to `json` format as described in Section 7.1.2. Lastly, to make sure the generative model actually comes up with valid tactics before committing to the action a_t it sampled, we check if a_t will produce a valid transition when it is given to the LEAN4 interface. If it does, it gets

accepted. Otherwise, 5 more samples are generated, and the first of the 5 that succeed gets accepted; and if all of them fail, the particle gets into the failure state. The sampling algorithm is summarized in Algorithm 1.

Algorithm 1 Sampling and Validation of Actions

Input: Current state s_t, policy $\pi(a|s, \theta)$, LEAN4 interface
Hyperparameters: Top-p threshold, temperature τ
Output: Valid action a_t or failure state
Sample $a_t \sim \pi(\cdot|s_t, \theta)$ using top-p sampling and temperature scaling
Restrict a_t to valid `json` format (see Section 7.1.2)
if `LEAN4_valid`(s_t, a_t) **then**
    Accept a_t
else
    Set attempts $= 5$
    for $i = 1, \ldots,$ attempts **do**
        Sample new $a_t^{(i)} \sim \pi(\cdot|s_t, \theta)$
        if `LEAN4_valid`$(s_t, a_t^{(i)})$ **then**
            Accept $a_t^{(i)}$
            Break
        end if
    end for
    if no valid action found **then**
        Assign particle to failure state
    end if
end if

7.1.6 Formal problem statement

Now that we have established all the relevant components for our problem, we can formally write down the problem statement. Let the MDP be modelled by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, r, \mu)$. Unlike the standard MDP setup, we omit the discount factor γ as we treat every reward the same alongside the proof sequences that reach a terminal solved state; this would be equivalent to taking $\gamma = 1$. Furthermore, μ is the initial state distribution, which will represent a distribution over the theorem statements we care about.

The set of states $\mathcal{S}$ contains the states specified in Section 7.1.1, and the set of actions $\mathcal{A}$ contains the actions specified in 7.1.2. Let π_θ be the policy that is parametrized by the generative model conditioned on the textual representation of the state as seen in the Periwinkle Gray-colored rectangle on the left in Figure 7.1. A general objective for theorem proving can then be expressed as:

$$\max_{\pi_\theta} \mathbb{E}_{S\sim\mu} \left[\sum_{t=1}^{T} r(s_t, a_t) \right]$$

where $S \sim \mu$ represents sampling a theorem statement S from the initial state distribution μ. The summation over t represents that we are interested in optimizing the reward over the entire proof trajectory. The trajectory terminates either when a valid proof is found (i.e., reaching a solved proof state) or when the predefined time horizon T is reached, in which case the proof attempt may be incomplete.

However, in this thesis, we approximate the general objective by training on a finite dataset D of theorem-proof pairs. This results in a supervised learning objective:

$$\max_{\theta} \mathbb{E}_{(S,P)\sim D} \left[\prod_{t=1}^{T} \pi_\theta(a_t|s_t) \right]$$

where $(S, P) \sim D$ represents problem statements S paired with their corresponding proof sequences P, drawn from the dataset D. This formulation treats D as an empirical approximation of μ, allowing us to train the policy in a data-driven manner.

7.1.7 Performance metric

For our testing, we care about compute used and about the accuracy obtained. To compute the accuracy, we use a test set containing 10 problems. For each run, the algorithm attempts to solve all 10 problems, and the accuracy is calculated as $n/10$, where n is the number of correctly solved problems. This process is repeated 4 times, resulting in 4 accuracy values. In some cases, as shown in the bar plots within the results section, the large language model server may become unresponsive, making it impossible to generate tactics. When this happens when starting a new problem, it would prevent the SMX algorithm from running. When this occurs, we exclude the affected exercise from the evaluation and compute the accuracy by dividing n by the number of remaining test exercises.

We have tested the algorithm in various configurations to pick apart what matters in terms of what increases the accuracy and the amount of compute used when solving the problems. The testing and training data samples are given in the Appendix C.3.

Alongside the accuracy estimation, we also track the number of attempts that were needed while running the algorithm. For each particle, we record at which i in Algorithm 1 we accept $a_t^{(i)}$ as a valid action. Then $i + 1$ is our number of attempts since we always start with one attempt. For each particle, this number of attempts is calculated and summed up to get the total number of attempts made during the running of the SMC algorithm.

7.2 The Approach: Interfacing with LEAN4

The approach in Polu et al. [34] and Yang et al. [53] use a LEAN proof assistant to provide feedback to the actions/tactics generated by the policy function π_θ. In both of the two works, tactics such as `cases' _ with`, `case _ =>` , and `have` are not supported. This limits the policy in generating tactics that targets the goals or sub-goals in a structured way.

Using the `cases' _ with` and `case _ =>` tactics allows the model to target specific goals or sub-goals in a way that aligns with how humans typically approach them.

This results in the generation of code that is more human-like and consistent with the patterns encountered during pre-training, resulting in the improvement of the model's performance.

Furthermore, using the `have <hypothesisName> : <type of hypothesis>` tactic allows the generative model to come up and prove intermediate proofs, also called lemmas, that will help in proving the main goal. Since human proofs often rely on this tactic, its use allows the model to generate more structured and human-like proofs. While it is possible to do everything without using lemmas, as shown in the motivating examples (Section 7.2.1), this requires deeper understanding of the proof strategy beforehand. Without lemmas, solving more complex problems becomes less feasible, since this will require even more foresight from the model.

7.2.1 Motivating Examples

In Section 7.2, we discussed how previous work [34, 53] does not support tactics like `cases' _ with`, `case _ =>` , and `have`, limiting the model's ability to generate structured proofs. To illustrate the importance of these tactics, we present two examples that show how their use enables more human-like and readable proof strategies.

Consider the two code examples below. The statement we want to prove is that $p \vee q$ implies $q \vee p$. This trivially follows from the commutativity of the "or" operation, but we will rigorously prove it in LEAN4.

In both cases, we start by introducing the term `hPorQ`: $p \vee q$. Our goal then becomes to show $q \vee p$, which we test by the second tactic `show` $q \vee p$. If that were not the case, this tactic would throw an error, which it did not. The next tactic in the two examples is where the important difference lies.

The third tactic models a standard approach in proving that is often taught in introductory proof courses [24]. If we can show that some proposition c holds when p holds (i.e., $p \to c$) and we can show that when q holds c holds as well (i.e., $q \to c$), then it follows that when p or q holds then c holds (i.e., $p \vee q \to c$). This reasoning, known as *Or elimination*, underlies both `apply Or.elim hPorQ` and `cases' hPorQ with hp hq`. The latter allows us to name the variables introduced in each sub-goal explicitly. In the first case called `inl`, the newly created term that is added to the context is named `hp`, which will have type p. In the second case, called `inr`, the newly created term is called `hq`, of type q.

In the right LEAN4 code example, we see that we write the tactics for both goals out without making use of `case _ =>` to specify which tactics apply to which sub-goal. For simple examples like the one shown below, the difference is small. However, for larger proofs, this will get significantly harder to read for humans. As a result, this style of writing proofs is more rare in pre-training data.

```
example : p ∨ q → q ∨ p := example : p ∨ q → q ∨ p
    by                            := by
  intro hPorQ                   intro hPorQ
  show q ∨ p                    show q ∨ p
  cases' hPorQ with hp hq       apply Or.elim hPorQ
  case inl =>                   intro hp
    apply Or.inr                apply Or.inr
    exact hp                    exact hp
  case inr =>                   intro hq
    apply Or.inl                apply Or.inl
    exact hq                    exact hq
```

As a second example, consider the two LEAN4 code snippets below. In this case, we want to prove the following statement:$\neg(\forall x, \neg p\ x)$ implies $\exists x, p\ x$. Where p is a function that returns a proposition, i.e., $p : (x : \alpha) \to Prop$. This too is an intuitively logical statement, as it asserts that if it is *not* the case that $\neg p\ x$ holds for all x, then there must exist an x for which $p\ x$ holds.

We begin both cases by introducing `hnxnPx`:$\neg(\forall x, \neg p\ x)$ using `intro hnxnPx`. In introductory math textbooks like Lay [24], this would be written as 'Suppose $\neg(\forall x, \neg p\ x)$'. In formal theorem proving, this introduction has to be referenced explicitly when it is used later on in the proof; hence, we assign it the name `hnxnPx`. Our goal then becomes showing $\exists x, p\ x$.

We proceed via proof by contradiction. Instead of directly proving $\exists x, p\ x$, we assume the opposite and derive a contradiction i.e.,

$$\neg(\exists x, p\ x) \to False.$$

Thus, in both cases, we introduce the assumption `hnxPx`:$\neg(\exists x, p\ x)$. The goal now is to derive a contradiction.

When doing a proof by contradiction, it is important to look for what could possibly create two contradictory statements (i.e. $c \wedge \neg c$). We observe that $x, \neg p\ x$ cannot hold for all x (`hnxnPx`); but at the same time, we see that there does not exist any x for which $p\ x$ holds (`hnxPx`). Intuitively, we can already see that both cannot be true at the same time. In this case, the best approach would be to show the opposite of `hnxnPx` to also hold, i.e., $\forall x, \neg p\ x$.

To do so, we introduce an arbitrary $x : \alpha$, using `intro x`. Then the goal becomes showing that $\neg p\ x$ holds, which is modeled in formal theorem proving as $p\ x$ leading to a contradiction (i.e. $p\ x \to False$). Thus, we assume that $p\ x$ holds, denoted by `hPx`, and we then must show a contradiction (i.e.,$False$). At this point, we have found an x for which $p\ x$ holds. However, we also have `hnxPx`:$\neg(\exists x, p\ x)$, so these two combined give us the contradiction from which anything follows, so also $False$, thus establishing the opposite of `hnxnPx` (i.e., $\forall x, \neg p\ x$).

This can then be used to prove $False$ again, as we have explicitly derived contradictory propositions: $\forall, \neg p\ x$ and `hnxPx`:$\neg(\exists x, p\ x)$. Both the upper and lower code exam-

ples follow this reasoning. However, the lower example relies on LEAN4 to infer missing steps. Instead of explicitly proving the lemma and applying it with `apply absurd hxNotPx hnxnPx`, it applies `apply absurd _ hnxnPx`, leaving what remains to be proven $\forall x, \neg(p\ x)$.

With simple examples like the two code examples below, it is hard to show the necessity of lemmas. But when problems get more complex, it is usually hard to know if the direction you are going in is the correct one. A common strategy is to establish intermediate claims that seem both reasonable and useful in proving the main goal. When multiple of such claims are needed, directly applying a lemma is often infeasible, making the trick we used in the lower code snippet much harder to implement.

These examples show why allowing the model to use tactics like `cases' _ with` and `have` improves proof structuring. Without them, the model must rely on proof strategies that are less structured and harder to follow, reducing alignment with human-written proofs. By supporting these tactics in our LEAN4 interface, we ensure that the model can generate proofs in a way that is both more readable and more aligned with the patterns found in pre-training data, potentially leading to better performance.

```
example : ¬ (∀ x, ¬ p x) → (∃ x, p x) := by
  intro hnxnPx
  apply byContradiction
  intro hnxPx
  have hxNotPx : ∀ x, ¬ (p x) := by
    intro x
    intro hPx
    exact absurd ⟨ x, hPx ⟩ hnxPx
  exact absurd hxNotPx hnxnPx


example : ¬ (∀ x, ¬ p x) → (∃ x, p x) := by
  intro hnxnPx
  apply byContradiction
  intro hnxPx
  apply absurd _ (hnxnPx : ¬∀ (x : α), ¬ p x)
  show ∀ (x : α), ¬ p x
  intro x
  intro hPx
  exact absurd ⟨ x, hPx ⟩ hnxPx
```

7.2.2 Overview LEAN4 Interaction with LLM

In Figure 7.1, we can see a schematic overview of how LEAN4 code is processed before entering the Large Language Model. The LEAN4 Interface processes the code to retrieve the current state and afterward returns a prompt used by the Large Language Model to predict the next tactic. When this tactic is successfully applied, the LEAN4 Interface updates

the state. Afterward, it retrieves the prompt that the Large Language Model will use to predict the bucket that represents how many tactics are needed to solve the remaining goals.

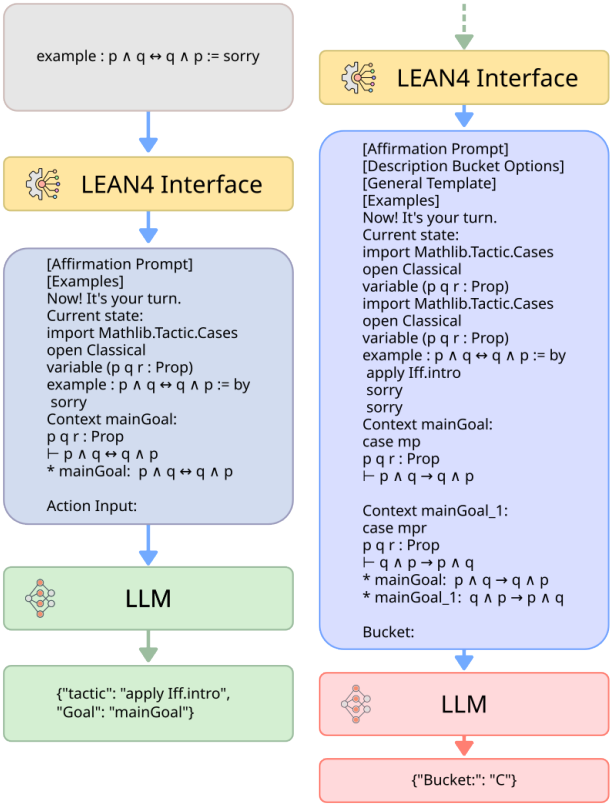

Figure 7.1: In this figure we can see how the input problem in LEAN4 is processed within the SMX algorithm.

7.3 Inference Procedure

The inference method consists of the search algorithm, in this case the E-step in SMX, and the generative model π_θ, which in this case is an LLM. The generative method samples tokens using top-p and temperature scaling, mentioned in Section 2.10. Recall from Section 5.2.3 Figure 7.2. In Figure 7.2 we again see how the E-step in SMX is applied using particle filtering. We will use this same approach in this thesis.

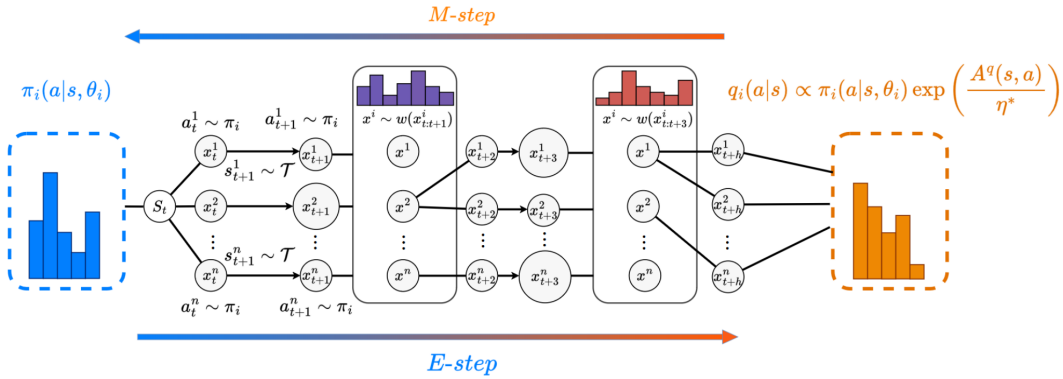

Figure 7.2: This figure visually represents the SMX search process, where multiple trajectory samples (denoted as $x^i, \ldots, x^n$) are generated in parallel using a policy π_i. During each step of the environment, particle weights are adjusted based on the advantage, as indicated by the size of the circles. The resampling stages are depicted where particles with higher weights are more likely to be prioritized for propagation, and the weights are reset to be uniform afterward. The final step involves estimating the target distribution q_i, which is constructed by the particles that are left in the final stage and their corresponding weights. This target distribution is then used to refine policy π in the M-step. Adapted from *SPO: Sequential Monte Carlo Policy Optimization* by Macfarlane et al. (2024) [27].

Each particle, representing a trajectory, gets an updated weight after an action/tactic has been applied. However, the way we update the weights differs from how it is done in Macfarlane et al. [27]. In Macfarlane et al. [27], weights are updated in the following way:

$$w(\tau_{1:t}) \propto w(\tau_{1:t-1}) \cdot \exp\left(A^{\bar{\pi}}(a_t, s_t)/\eta_i^*\right)$$

where the advantage function $A^{\bar{\pi}}$ gets approximated as $\tilde{r}_t + V^{\bar{\pi}}(s_{t+1}) - V^{\bar{\pi}}(s_t)$. Here, we introduce $\tilde{r}_t$ instead of r_t as our notation to distinguish it from the reward function defined in our MDP (Section 7.1.4). The term $\tilde{r}_t$ represents the immediate performance of the action a_t in the state s_t, this reward indicates how well that action performed on itself, not looking ahead. On the other hand, the value function $V^{\bar{\pi}}(s)$ gives the expected cumulative reward from state s, i.e.,

$$V^{\pi}(s) = \mathbb{E}_{\tau \sim \pi}\left[\sum_{t=0}^{T} \gamma^t \tilde{r}_t \mid s_0 = s\right].$$

The policy π determines the distribution of trajectories $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$ when calculating the expected cumulative reward. We took a simplified approach:

$$w(\tau_{1:t}) = w(\tau_{1:t-1}) + \hat{V}_t - \hat{V}_{t-1}.$$

Where the value function $\hat{V}_t$ is modeled by r_t. We apply supervised fine-tuning to make $r_t = r(s_t, a_t)$ accurately represent the number of tactics we still have to apply at state s_{t+1} (the state after having applied action a_t to s_t).

As already mentioned, normally, the reward $\tilde{r}_t$ evaluates the immediate benefit of an action in a given state, while the value function aggregates future rewards to assess the desirability of the state itself. However in our setting, the function $r_t = r(s_t, a_t)$ already provides an estimate of the number of tactics that remain to be applied at s_{t+1}. This means it inherently looks ahead and acts more like a value function than a standard reward function. Thus, we omit $\hat{r}_t$ in our formulation and rely solely on $\hat{V}_t$, which is modeled using r_t.

The idea behind $A^{\bar{\pi}}$ in the weights update of Macfarlane et al. [27] is to take into account the actual reward obtained from the environment, with $\hat{r}_t$; and the increase in expected cumulative reward, with $\hat{V}_t - \hat{V}_{t-1}$. In our case, since value really just means the expected number of actions that still have to be done, we already get the desired functionality without $\hat{r}_t$. Namely, when the number of tactics would decrease as expected, the value difference would be positive, thus increasing the weight. When the number of tactics decreases more than expected, the weight would increase even more since the difference would be higher. On the other hand, if the number of tactics increases, the weight would get smaller.

In Algorithm 2, the SMC search algorithm of Macfarlane et al. [27] is shown. Furthermore, in Algorithm 3 we can see the modified SMC algorithm made in this thesis. We can see in line 1 that we start with initializing the N different particles all starting at the starting state s_t, and in line 2, we set the weights all to 1. We then iterate over h steps, where we sample from the policy π_θ in accordance to how it is described in Section 7.1.5. Then in line 7 in both algorithms, we see the weights getting updated in the two different ways. In line 8 of Algorithm 3, we see `check_cond`. This checks if one of the particles succeeded in solving all goals or if all particles are in failure state. In both these cases, the algorithm would terminate.

Furthermore, in line 12 of Algorithm 3, we apply the softmax function. This is done to ensure the weights sum up to 1, and negative weights become positive. Then in line 13, we set the weights to zero for the weights corresponding to a failure state and implicitly renormalize afterwards. Finally, in line 14, we resample the trajectories, where trajectories with higher weights are more likely to get sampled than trajectories with lower weights.

Algorithm 2 SMC q target estimation (timestep t)

Initialize $\{s_t^{(n)} = s_t\}_{n=1}^N$
Set $\{w_t^{(n)} = 1\}_{n=1}^N$
for $i \in \{t+1, \dots, t+h\}$ **do**
  $\{a_i^{(n)} \sim \pi(\cdot|s_i^{(n)}, \theta)\}_{n=1}^N$
  $\{s_{i+1}^{(n)} \sim \mathcal{T}(s_i^{(n)}, a_i^{(n)})\}_{n=1}^N$
  $\{r_i^{(n)} \sim r(s_i^{(n)}, a_i^{(n)})\}_{n=1}^N$
  $\{w_i^{(n)} = w_{i-1}^{(n)} \cdot \exp(\hat{A}(s_i^{(n)}, r_i^{(n)}, s_{i+1}^{(n)})/\eta^*)\}_{n=1}^N$
  if $(i-t) \mod p = 0$ **then**
    $\{x_{t:i}^{(n)}\}_{n=1}^N \sim \text{Mult}(N; w_i^{(1)}, ..., w_i^{(N)})$
    $\{w_i^{(n)} = 1\}_{n=1}^N$
  end if
end for
$\{a_t^{(n)}\}$ as the set of first actions of $\{x_{t:t+h}^{(n)}\}_{n=1}^N$
$\hat{q}(a|s_t) = \sum_{n=1}^N \bar{w}^{(n)} \delta_{a_t^{(n)}}(a)$

Algorithm 3 SMC q target estimation (modified)

Initialize $\{s_t^{(n)} = s_t\}_{n=1}^N$
Set $\{w_t^{(n)} = 1\}_{n=1}^N$
for $i \in \{t+1, \dots, t+h\}$ **do**
  $\{a_i^{(n)} \sim \pi(\cdot|s_i^{(n)}, \theta)\}_{n=1}^N$
  $\{s_{i+1}^{(n)} \sim \mathcal{T}(s_i^{(n)}, a_i^{(n)})\}_{n=1}^N$
  $\{r_i^{(n)} \sim r(s_i^{(n)}, a_i^{(n)})\}_{n=1}^N$
  $\{w_i^{(n)} = w_{i-1}^{(n)} + r_i^{(n)} - r_{i-1}^{(n)}\}_{n=1}^N$
  if `check_cond`$(\{s_{i+1}^{(n)}\}_{n=1}^N)$ **then**
    Break (terminate loop early)
  end if
  if $(i-t) \mod p = 0$ **then**
    $\{\tilde{w}_i\}_{n=1}^N = \text{Softmax}(N; w_i^{(1)}, ..., w_i^{(N)})$
    `set_zeros`$(\{\tilde{w}_i^{(n)}, s_{i+1}^{(n)}\})$
    $\{x_{t:i}^{(n)}\}_{n=1}^N \sim \text{Mult}(N; \tilde{w}_i^{(1)}, ..., \tilde{w}_i^{(N)})$
    $\{w_i^{(n)} = 1\}_{n=1}^N$
  end if
end for
$\{a_{t:h}^{(k)}\}$ as the set of actions of $\{x_{t:t+h}^{(n)}\}_{n=1}^N$ that solve s_t, where $k \in \{1, \dots, N\}$

7.4 Neural Network Improvement Procedure

In this section we will discuss the M-step, how it is implemented in the SMX paper, and how it could be integrated with this thesis. Furthermore, we will go into what we did in this thesis and why. Lastly, the limitations of the implemented approach will be discussed.

7.4.1 M-step of Macfarlane et al. [27]

In Section 5.2.4 we see the main objective of the M-step, from Macfarlane et al. [27] we can see that the following loss functions are brought about by it:

$$\mathcal{L}_\pi(\theta) = -\sum_{s,a\sim D} q_i(a|s) \log \pi_{\theta_i}(a|s)$$
$$\mathcal{L}_\alpha(\theta, \alpha) = \alpha \left(\epsilon_\alpha - \mathbb{E}_{s\sim p(s)}[\text{sg}\, \mathcal{D}_{KL}(\pi_{\theta_{\text{old}}} \parallel \pi_\theta)]\right)$$
$$\mathcal{L}_{KL}(\theta, \alpha) = \text{sg}[\alpha]\mathbb{E}_{s\sim p(s)}[\mathcal{D}_{KL}(\pi_{\theta_{\text{old}}} \parallel \pi_\theta)]$$

We left out the loss function to optimize η since this is not relevant to our discussion. This then results in the following pseudocode [27]:

Algorithm 4 SMX Algorithm

1: $\mathcal{B} \leftarrow \emptyset$
2: **Initialize the policy, value function, temperature, and alpha:**
3: Initialize $\theta_0, \phi_0, \eta_0, \alpha_0$
4: **for** iteration $k = 0, 1, 2, \ldots,$ num_iterations **do**
5: **Expectation Step (E-step):**
6: **for** agent $m = 1, 2, \ldots, M$ in parallel **do**
7: Initialize state s_0
8: **for** timestep $i = 1, 2, \ldots,$ rollout_length **do**
9: $\bar{a} = \{a^{(n)}\}_{n=1}^{N} \sim \hat{q}_{\text{SMC}}(s_i, \eta_k, \theta_k, \phi_k)$ ▷ Sample-based estimate of q_i
10: $a \sim \{a^{(n)}\}_{n=1}^{N}$ ▷ Sample an action to execute in the environment
11: $s_{i+1} \sim \mathcal{T}(s_i, a)$ ▷ Execute action a, observe next state s_{i+1}
12: $r_i \sim r(s_i, a)$ ▷ Observe reward r_i
13: Store $(s_i, r_i, \bar{a})$ in $\mathcal{B}$
14: **end for**
15: **end for**
16: **Maximization Step (M-step):**
17: **for** batch $b = 1, 2, \ldots,$ num_batches **do**
18: **Sample batch from replay buffer:**
19: Sample batch of size N from replay buffer $(s, r, \bar{a}) \sim \mathcal{B}$
20: **Update value function using GAE targets:**
21: $\phi_{k+1} \leftarrow \arg\min_{\phi} \mathbb{E}_{s\sim\mathcal{B}} \left[(V_{\text{GAE}}(s, \phi') - V(s, \phi))^2 \right]$
22: **Update parameterized policy using sample-based estimate of q:**
23: $\theta_{k+1}, \alpha_{k+1} \leftarrow \arg\min_{\theta,\alpha} \mathbb{E}_{s\sim\mathcal{B}} \left[\mathcal{L}_{\pi}(\theta) + \mathcal{L}_{\alpha}(\theta, \alpha) + \mathcal{L}_{KL}(\theta, \alpha) \right]$
24: **Update dual temperature:**
25: $\eta_{k+1} \leftarrow$ `update_eta`(η_k,$\mathcal{B}$)
26: $\phi' \leftarrow \text{polyak}(\phi', \phi_{k+1})$
27: **end for**
28: **end for**

In lines 6 to 13 we see the Sequential Monte Carlo loop being repeatedly applied and the resulting weighted vector of actions gets sampled and applied to the environment. The resulting rewards r_i, weighted vector of actions $\bar{a}$, and state s_i get stored as a sample for use in the M-step.

From lines 17 to 26 the actual M-step starts. We start by sampling from the dataset generated from the E-step in line 19. Then in line 21, we use the GAE value estimate (see Appendix C) to improve the parameters of the value network. In line 23 we both optimize α and θ. The purpose of optimizing α is to adjust α to become larger when the KL gets larger than ϵ and to decrease α if the KL gets smaller than ϵ, adjusting the strength of the KL loss dynamically. Furthermore, π_θ gets optimized with $\mathcal{L}_{\text{KL}}(\theta, \alpha)$ to make sure the policy does not diverge too much from the previous policy. Large differences in the policy can inadvertently remove what made the policy work before, causing large drops in performance; $\mathcal{L}_{\text{KL}}(\theta, \alpha)$ mitigates this effect. In line 25 we update η, we abstracted this

away in a function, since this is not relevant to this thesis.

Lastly in line 26, we update value function parameters ϕ', using polyak averaging. This means that the newly updated parameters ϕ_{k+1} (obtained from the value function update in line 21) are slowly incorporated into ϕ' using the update rule:

$$\phi' \leftarrow \tau\phi' + (1-\tau)\phi_{k+1}$$

Where τ is some smoothing factor close to 1. This gradual update ensures that the parameters ϕ' do not change too quickly, leading to a more stable estimation of the value function. Since the GAE method relies on the value function to compute the advantage targets, keeping ϕ' stable helps reduce variance in those estimates, improving learning stability.

7.4.2 Integrating M-step of Macfarlane et al. [27]

To integrate the M-step described in the previous section with this thesis, we need to integrate the learning of the reward function into this algorithm. We propose to reuse $\hat{q}_{\text{SMC}}$. Instead of just returning a weighted vector with possible actions, it should also return the number of tactics needed to solve the problem. Additionally, it should include an indicator that specifies whether this number is an estimate or if the problem was actually solved within the horizon h of the SMC algorithm. In line 12, we can use the actual bucket from the real number of tactics when this is returned from line 9. When the reward is stored in $\mathcal{B}$ in line 13, it should also indicate whether the reward estimate comes from the generative model. This would allow us to use only the real rewards to train the generative model.

Then for training the reward function, this could be done using supervised fine-tuning as is also done in this thesis. Furthermore, since the value function and η are not present in our thesis, we can remove lines 21, 25, and 26 in algorithm 4.

7.4.3 Practical Implementation in This Work

Due to time constraints, we focused mostly on the Sequential Monte Carlo aspect of the SMX algorithm. Instead of generating data using the model itself through the E-step, we used a dataset we constructed ourselves of exercises from the LEAN4 tutorial. We furthermore did not use the loss functions $\mathcal{L}_\pi$, $\mathcal{L}_\alpha$ and $\mathcal{L}_{KL}$ to train the policy π_θ and reward r_t, instead we used supervised learning as described in Section 2.9. In the next section, we will give the specific details of training, and in the two sections after, we discuss how the tactic data and reward data is constructed.

Training

The training is done with `pytorch`, specifically with `torchtune` [47], which has a fine-tuning script where you can customize how the data in the dataset gets formatted when entering the model. To see how the data gets structured when entering the model in this thesis, see the appendix section A.1. We trained the model for 15 epochs with the adamW loss function, and after each epoch, the model was saved so that we could test the model after each epoch. The learning rate was set to $3 \cdot 10^{-4}$ and is cosine scheduled with warmup.

We furthermore added weight decay to the optimizer with 0.01. We used QLORA as the fine-tuning method. Most of these parameters in the finetuning script (such as the optimizer, learning rate and weight decay) were kept to their default value, the formatting, epochs, and fine-tuning method were the only modified parameters in the config. The training and testing itself took place on the cloud computing service called `runpod.io` since the compute requirements are high when training such large models.

Dataset construction

The syntax of LEAN4 is very flexible, and not every part of the LEAN4 syntax can fit well into the mold of generating tactics one at a time. Especially, pattern matching with 'match with', which we discussed in section 3.4 on Tactics. To address both the computational constraints and concerns regarding the model's performance on more difficult problems, we constructed a simplified dataset containing only a subset of the syntax supported in LEAN4. To make it possible to still make use of cases, the `cases' with` syntax is supported, which we also discussed in the tactics Section 3.4 and Section 7.2.1.

The dataset is created with the exercises from the theorem proving in LEAN tutorial [2]. Each of these exercises is processed into problem statements and the tactics that would solve that problem statement. This is visualized in Figure 7.3. The human-readable form of the state is then collected after each tactic application, which takes the same form as the text in the greyish-blue rectangle (Periwinkle Gray) in Figure 7.1. Not all exercises are used for training; a part is left out to test the algorithm on. How the testing takes place was described in Section 7.1.7.

Reward collection

The reward collection is inspired by Polu et al. [34], where the reward is based on the model's estimate of the number of tactics needed, provided with the current state. Predicting the exact number of tactics required given the current state is, of course, challenging. Instead, like Polu and Sutskever [33], we let the model predict buckets from A to K, where A indicates that the proof is impossible from this point onward. K represents the smallest possible proof size, as discussed in 7.1.4.

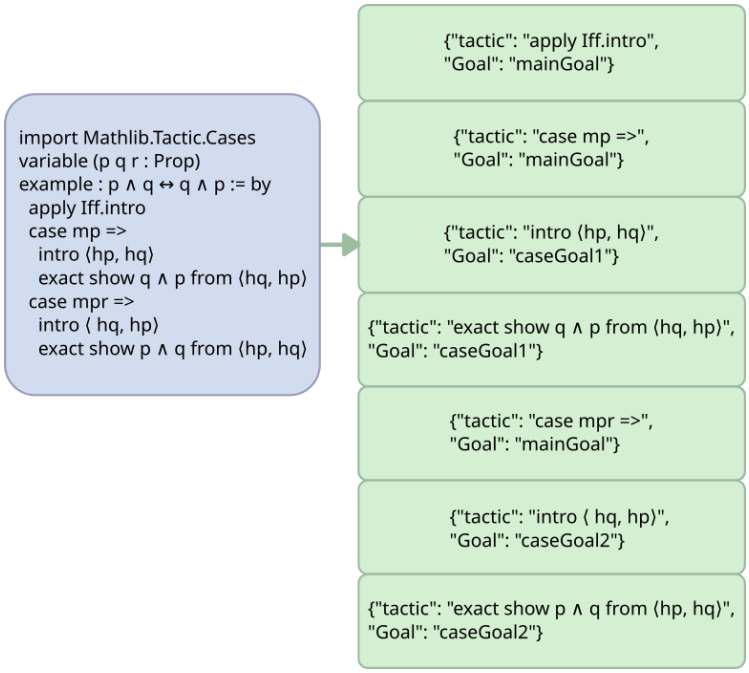

Figure 7.3: This figure shows how a solved problem is converted into individual tactics that, if given to the LEAN4 Interface in sequence, will solve the problem statement.

Chapter 8

Results

In essence, we need to evaluate three aspects. The first aspect is the evaluation of the simplified neural improvement algorithm. We saved the parameters of the generative model at multiple stages during training for each of the saved parameters we will evaluate the accuracy. Additionally, we track the number of attempts required to obtain a valid next state when applying an action during the algorithm's execution.

The second aspect is evaluating how increased computational power benefits the algorithm. Since it is closely related to Sequential Monte Carlo, we expect that increasing the number of particles will raise computational costs, while also improving the performance.

The third aspect is determining the influence of hyperparameters such as the p in top-p and the temperature τ in temperature scaling in the inference mechanism of our generative model. During an inspection of less performing runs, we have seen that these hyperparameters have a sizable influence on the algorithm's behavior. Thus, this behavior is investigated and evaluated.

These three aspects are evaluated and discussed in Sections 8.1, 8.2, and 8.3 respectively. Each section begins with the broader question we aim to answer, followed by the concrete question investigated and addressed. To fit the scope of this thesis, we made simplifications in our implementation and worked within the constraints of limited computational power. We will discuss these limitations wherever they impact our findings. All experiments are conducted on a single A40 GPU rented using `runpod.io`, the specifics for each experiment are displayed in table 8.1.

The second column in table 8.1 gives the Lora Checkpoint used in the experiment. LoRa is a training method where the updates to the model weights are stored in a small tensor. Because this tensor is small, LoRa training needs less data and fewer weight updates for the same performance increase. In Section 2.9 we briefly discuss this and reference relevant sources for more details. The third column in table 8.1 represents the Top-p parameter value in the experiment. The p in Top-p influences the diversity the generative model when generating text (see Section 2.10.2). Similarly, the fourth column gives the value for another parameter that influences the diversity of the generative model when generating text. This parameter is called the Temperature and influences the sharpness of the next token prediction probability distribution (see Section 2.10.3). The fifth column represents the number of particles N that the SMC algorithm has while running. The more particles

the more proofs are attempted in parallel, see Section 7.

Experiment	LoRa Checkpoint	Top-p	Temperature	Particles	Time
grateful-surf	6	0.9	0.8	5	6h29m27s
swept-voice	8	0.9	0.8	5	6h7m28s
rare-star	10	0.9	0.8	5	4h58m9s
lilac-snow	12	0.9	0.8	5	6h18m20s
wobbly-fog	14	0.9	0.8	5	5h22m41s
quiet-snowflake	1	0.9	0.8	8	10h37m13s
clean-sponge	1	0.9	0.8	12	15h10m4s
trim-fire	1	0.9	0.8	14	15h21m14s
zany-wildflower	1	0.9	0.8	16	19h35m30s
jumping-shadow	1	0.9	0.8	18	16h35m49s
glad-haze	14	0.9	0.8	5	2h36m51s
fast-shape	14	0.9	0.8	12	1h26m58s
light-galaxy	14	0.9	0.8	16	5h52m22s
fanciful-forest	14	0.9	0.85	5	3h21m36s
lunar-dawn	14	0.9	0.9	5	4h24m40s
worthy-moon	14	0.9	0.95	5	4h39m24s
laced-eon	14	0.9	0.85	18	4h55m9s
royal-butterfly	14	0.9	0.9	18	6h41m49s
exalted-vortex	14	0.9	0.95	18	10h29s
noble-feather	14	0.85	0.8	14	3h16m32s
vocal-capybara	14	0.9	0.8	14	4h26m
bumbling-firefly	14	0.95	0.8	14	5h20m1s
lyric-brook	14	0.8	0.95	14	3h43m47s
iconic-dawn	14	0.9	0.95	14	6h30m23s
proud-glitter	14	0.95	0.95	14	11h6m30s
quiet-aardvark	14	0.9	0.95	5	3h40m5s

Table 8.1: Experiment configurations, including LoRa checkpoint, sampling parameters, and particle count, along with the computation time required for each run.

8.1 Evaluation of the Simplified Neural Improvement Algorithm

In this section, the neural improvement algorithm is evaluated. On the small dataset containing LEAN4 exercises we apply supervised fine-tuning as discussed in Section 7.4.3. The motivation behind it is that we want to be able to answer the following question:

> Does the proposed Neural improvement algorithm
> make the search algorithm more accurate and efficient?

This research question is out of scope for this thesis. Therefore, the research question is simplified to:

> Does supervised fine-tuning influence computational
> efficiency and accuracy in formal theorem proving using the SMC algorithm?

Where instead of using the proposed neural improvement algorithm, normal supervised fine-tuning is used with a training dataset of LEAN4 problems. See Section 2.9 on training for more information on supervised fine-tuning.

8.1.1 Experimental setup

To address this research question, we tested the SMC algorithm on 10 test problems using checkpoints 6 through 14 (corresponding to the 7th and to the 15th training checkpoints). The earlier checkpoints (0 through 5) were skipped to reduce computational costs.

For each checkpoint, we tracked the number of attempts required for the SMC algorithm to solve the test problems. This allowed us to analyze whether performance trends could be attributed to failures of SMC instances.

Additionally, for 4 different checkpoints, we compared the number of attempts while running the SMC algorithm. For 6 and 14, and for 1 and 14. The difference being that for the former we take the number of particles as $N = 5$ and for the latter we take the number of particles as $N = 14$.

8.1.2 Results and Discussion

Let us start with displaying how the loss function changes over multiple gradient descent steps. In Figure 8.1 we can see the loss decreasing. Despite the model being better at predicting the next tokens on the training set, it does not perform significantly better when applying the SMC method described in the methods section. The loss function is the cross-entropy loss described as:

$$\mathcal{L}_s(y, \hat{y}) = -\frac{1}{n \cdot z} \sum_{b=s}^{s+z} \sum_{i=1}^{n} \sum_{v=1}^{V} y_{b,i,v} \log(\hat{y}_{b,i,v}),$$

where $\hat{y}$ is the prediction, y is the actual label, V is the vocabulary size (number of different tokens). z is furthermore the number of batches and $s \in \mathbb{N}$ is the s^{th} batch that was uniformly sampled from $1, \ldots, z$. See Section 2.9 for more information on this formula and training in general.

In Figure 8.2 it can be seen that the accuracy on the test problems does not improve along the different checkpoints of the training process, see Appendix C.3 for the test set problems. This is likely due to the dataset being too small, containing tactics derived from only 30 distinct problems. While Figure 8.1 shows that the training loss decreased over iterations, this reduction likely is due to overfitting. The model seems to have learned to

Figure 8.1: Model loss progresses over the training loop

output correct tactics for the training data but fails to generalize when suggesting tactics during the execution of the SMC search algorithm on the test data.

The overfitting in this context is memorization driven. In other words the model has learned to replicate training examples word for word rather than capturing generalizable patterns for generating tactics. This exacerbated by requirement for exact syntax in tactic suggestions. Even a small variation in capitalization would already produce the wrong tactic. This result implies that an iterative process, such as alternating between the E-step (executing SMC search, see Section 7.3) and the M-step (fine-tuning on newly generated data, see Section 7.4), would be more beneficial. By generating additional data through the repeated application of SMC search, the model could incrementally improve its predictions. This iterative approach also offers advantages of using self-generated data, which includes good, medium and poor-quality tactics. Having this diversity allows the model to correct when it is overly optimistic about the effectiveness of a tactic.

Figure 8.3 compares the number of attempts for the latest checkpoint and the 6^{th} checkpoint. We can see how the number of attempts looks relatively similar between the two. This confirms the suspicion that the dataset was too small to lead the model to generalize. As a result, improvements in performance or efficiency can be seen across these checkpoints.

In the bar plots of Figure 8.4 we can see that some problems have fewer bars than others. This is because in these instances the SMC algorithm never started up because the large language model server failed to start on time. Unfortunately, this will influence the accuracy scatter plots and attempts bar plots. This is even more visible in Figure 8.14.

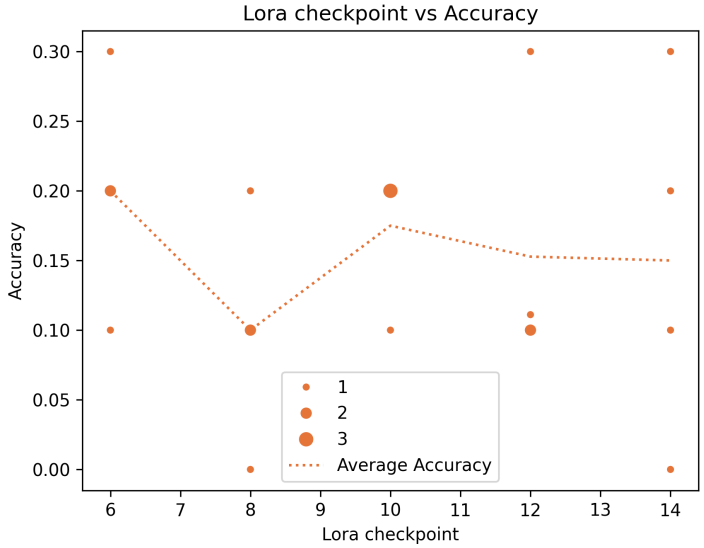

Figure 8.2: This figure displays how the accuracy changes over the various Lora checkpoints. $N = 5$ in this case, meaning, the number of particles in sequential Monte Carlo is 5. The Top-p parameter is 0.9, and the temperature is set to 0.8. The numbers 1, 2, and 3 indicate how often a specific accuracy value was observed.

8.2 Impact of Increased Computing Resources on Algorithm Performance

In this section, the impact of increasing the computing resources on the performance is assessed. The following general question is answered:

> Does the performance of the proposed search algorithm increase
> as the amount of computing resources increases for searching.

In the case of this thesis, the E-step proposed in the thesis is out of scope computationally. Therefore, we look at the inner SMC loop instead (see Algorithm 3). The research question is simplified to:

> How does the number of particles in the SMC method
> affect overall accuracy on the test set and efficiency
> across individual problems?

Here the simplification is that the amount of computing resources used is substituted for the number of particles used in the SMC algorithm. Since more particles mean more proofs are attempted in parallel, the amount of computation done increases as the number of particles increases, making this substitution a reasonable simplification.

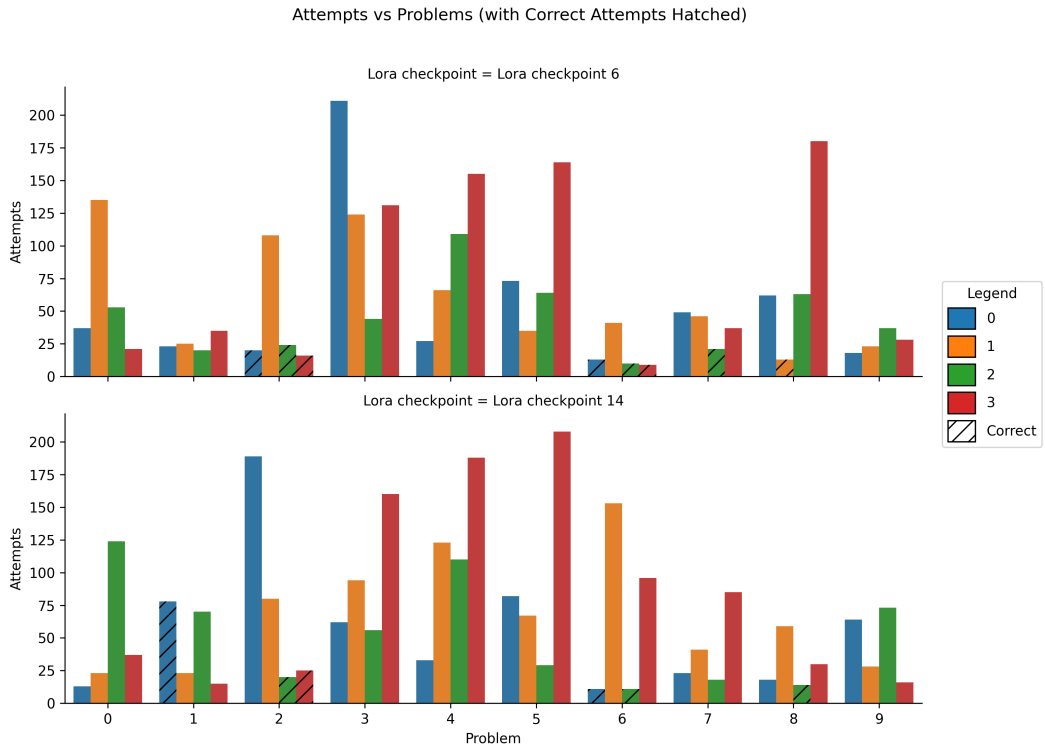

Figure 8.3: In this figure we can see the number of attempts of two different Lora checkpoints. Both have 5 different particles ($N = 5$) in the SMC algorithm and have a top-p of 0.90 and a temperature of 0.8

8.2.1 Experimental setup

To be able to answer the simplified research question, we varied the number of particles (5, 8, 12, 14, 16, and 18) to evaluate how this parameter affects accuracy and the number of attempts. We conducted these tests for two LoRA checkpoints: checkpoint 1 and checkpoint 14.

8.2.2 Results and Discussion

Figure 8.7 shows the model's performance after being trained for 15 epochs. The figure shows that the performance indeed does increase as we increase the number of particles, particularly for the final checkpoint. However, when there are 16 particles, the average accuracy decreases, as seen in the right plot of Figure 8.7. This makes it difficult to determine whether this reflects a genuine pattern or is due to the randomness inherent in the SMC algorithm. We will further explore what makes $N = 14$ and $N = 16$ different in Figure 8.6

In the bar plots in Figure 8.6 we compare two different number of particles: $N = 14$ and $N = 16$, shown in the right plot of Figure 8.7. For $N = 16$, none of the SMC instances tasked with solving problem 4 successfully started. Both $N = 14$ and $N = 16$ experience

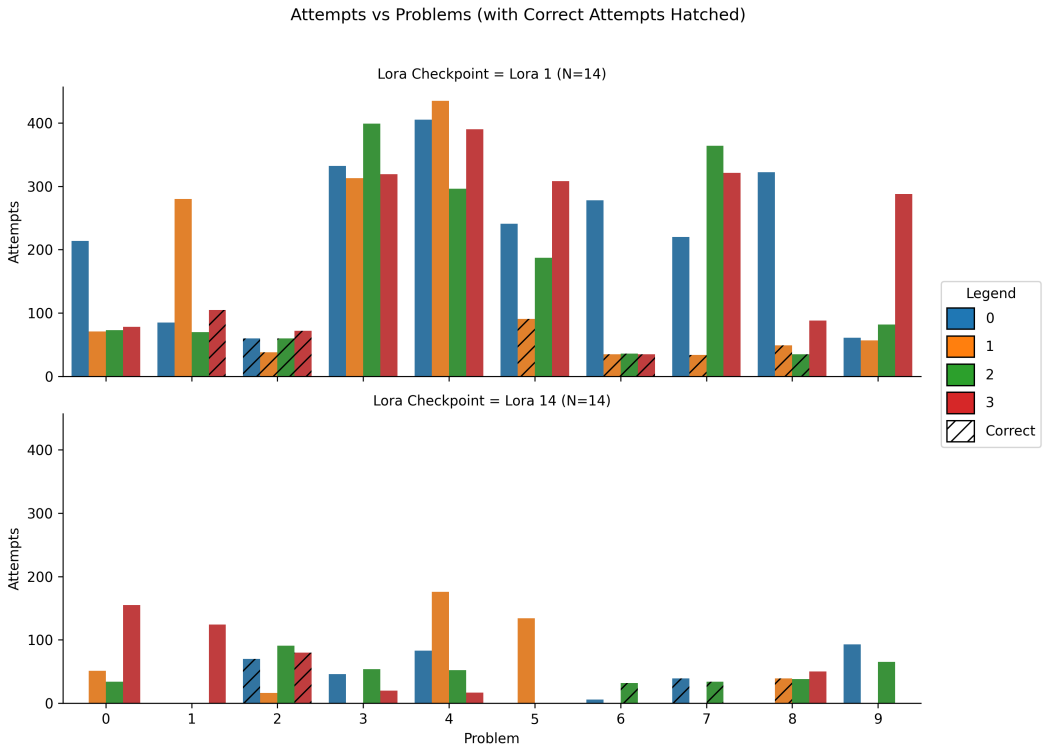

Figure 8.4: In this figure we can see the number of attempts over the different problems as bars in a bar plot the two bar plots belong running the SMC algorithm with two different checkpoints. For the number of particles we have 14 ($N = 14$), $p = 0.9$ and the temperature is 0.8

failing processes, leading to high variance in the accuracies for $N = 14$ but with a relatively high average accuracy. In contrast, for $N = 16$ we have a lower average accuracy despite solving 12 out of the total 25 successfully started exercises, compared to $N = 14$ which solves 6 of 23.

The variability in the dots representing the different average accuracies across chains has to do with failing processes. We will go into how this can be remedied in the future in more detail in the discussion section (Section 9.3). With Figures 8.8 and 8.9 we can gain insight into the relationship between the number of particles and efficiency. Despite the increase in particles, the number of attempts in the lower bar plots remains comparable. This could be explained by the resampling process: states with a higher weights are more likely to be copied. These states, coincidentally, may also align with those where the model can generate valid tactics with high likelihood. The co-occurrence of giving a higher likelihood for valid tactics combined with that state having a high weight gets higher as the number of particles increases. Conversely, when the number of particles is low, the algorithm may get stuck on a set of states where valid tactics are harder to produce, leading to a higher frequency of failed attempts.

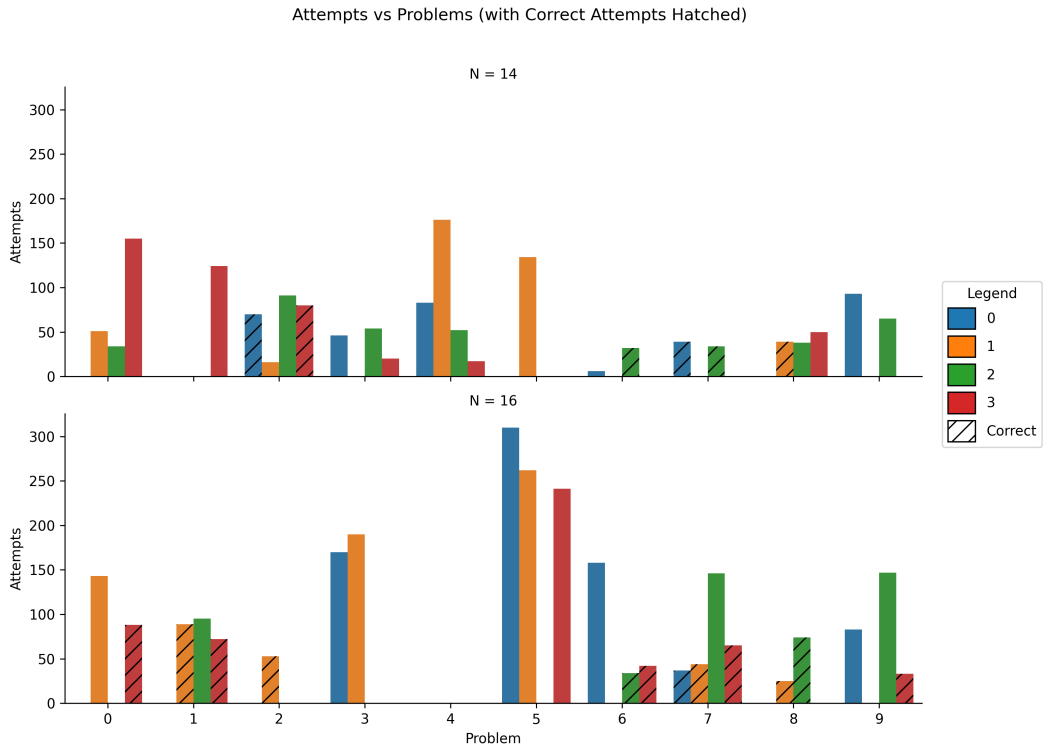

Figure 8.5: In this figure we see two bar plots, one applies SMX with 14 particles and another applies SMX with 16 particles. The temperature is 0.8 and $p = 0.9$. As our Lora checkpoint we use the 14^{th} checkpoint.

8.3 Influence of Hyperparameters on Inference Behavior

An inspection of low-performing SMC instances showed cases where the model repeated actions that were not useful and repetitive, see example 8.16 and example 8.17. This suggests that using a higher temperature could help prevent the generative model from getting stuck in repetitive predictions. Which is what motivates this last research question:

> How do hyperparameters influence
> the performance and efficiency of the proposed algorithm?

In this thesis, the generative model has two hyperparameters, the temperature τ and the p in top-p. So the research question is rephrased as:

> How do temperature and top-p individually affect
> efficiency and accuracy in the SMC method, and what is their combined impact?

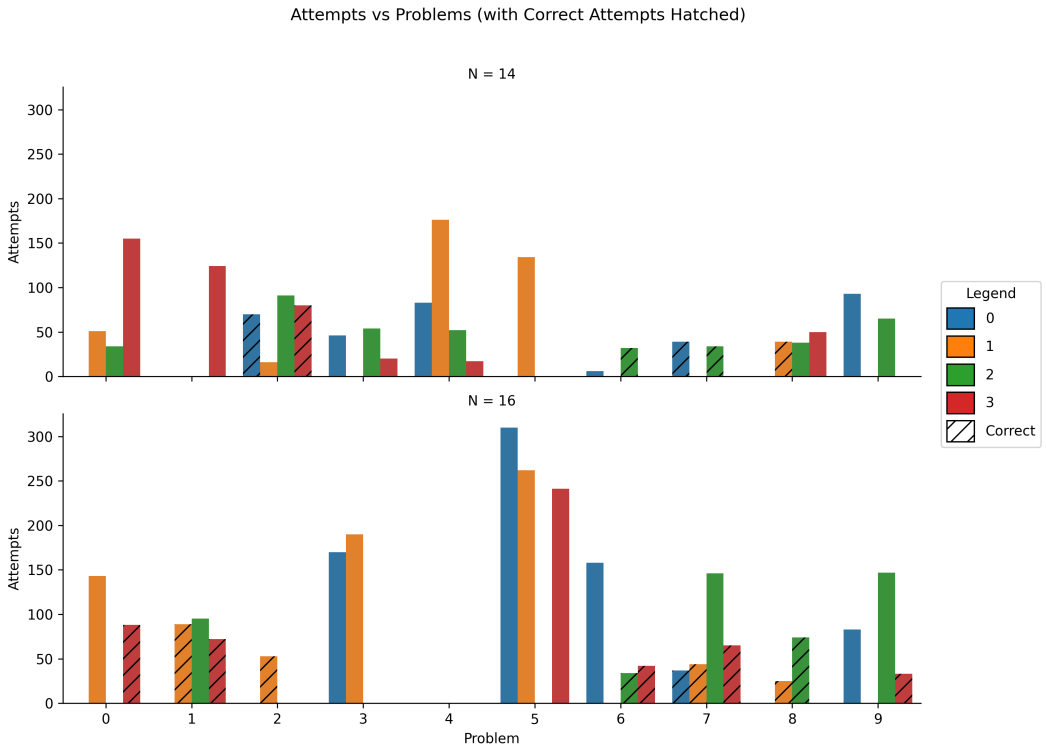

Figure 8.6: In this figure we see two bar plots, one applies SMX with 14 particles and another applies SMX with 16 particles. The temperature is 0.8 and $p = 0.9$. As our Lora checkpoint we use the 14^{th} checkpoint.

8.3.1 Experimental setup

To determine the effects of temperature and top-p on the performance of the SMX algorithm, we varied these parameters individually and in combination.

First, we evaluated how accuracy and efficiency change with temperature values of 0.80, 0.85, 0.90, and 0.95. These experiments were conducted for two different particle counts: 5 and 14.

Next, we investigated the impact of top-p by varying p values (0.80, 0.85, 0.90, 0.95). These experiments were performed under two fixed temperature conditions: 0.95 and 0.80, with the number of particles set to 14.

8.3.2 Results and discussion

Results

In Figure 8.10 we can see that increasing the temperature has more of a beneficial effect when the number of particles is higher. This aligns with the intuition that, with more parallel attempts running concurrently, selecting more diverse next possible tokens leads to greater

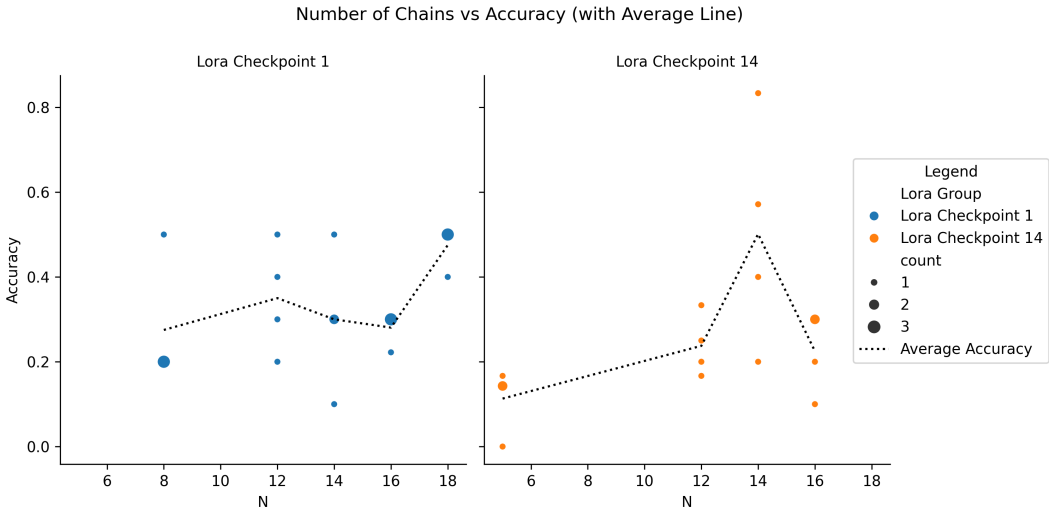

Figure 8.7: The accuracy over different numbers of particles (N) and for two different Lora checkpoints (14 and 1). The top-p parameter here is 0.9 as well and the temperature is 0.8. The numbers 1, 2, and 3 indicate how many times a specific accuracy value was observed.

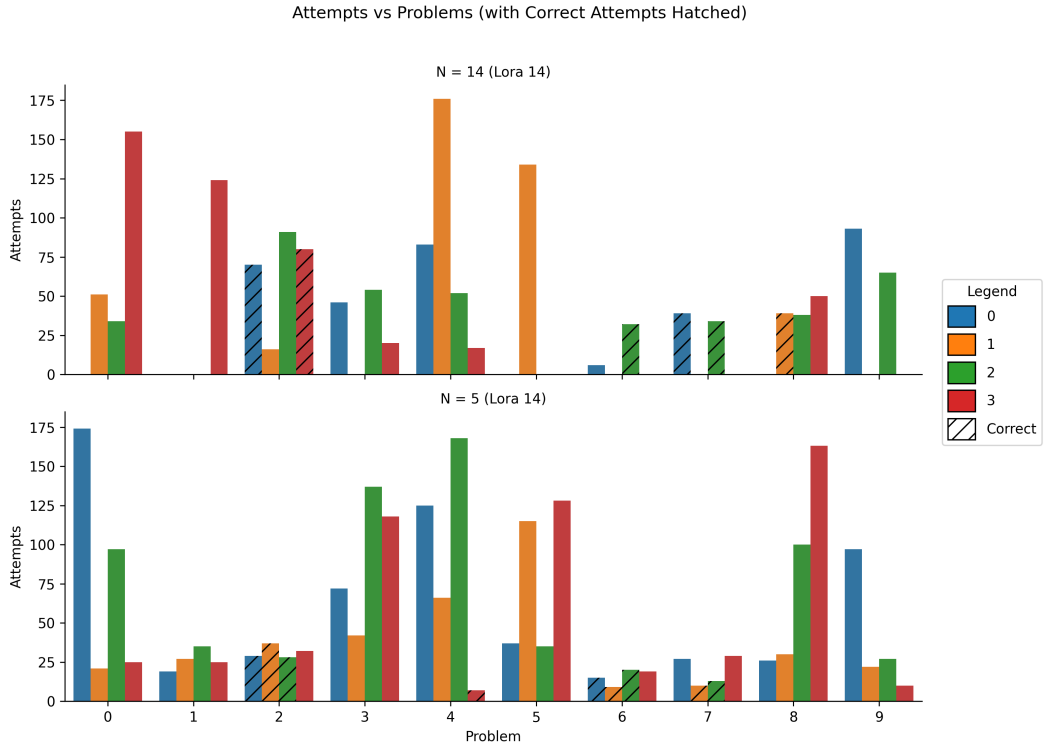

Figure 8.8: In this figure we can see two bar plots over different numbers of particles. Here $p = 0.9$ and the temperature is 0.8 and we use the latest Lora (14)

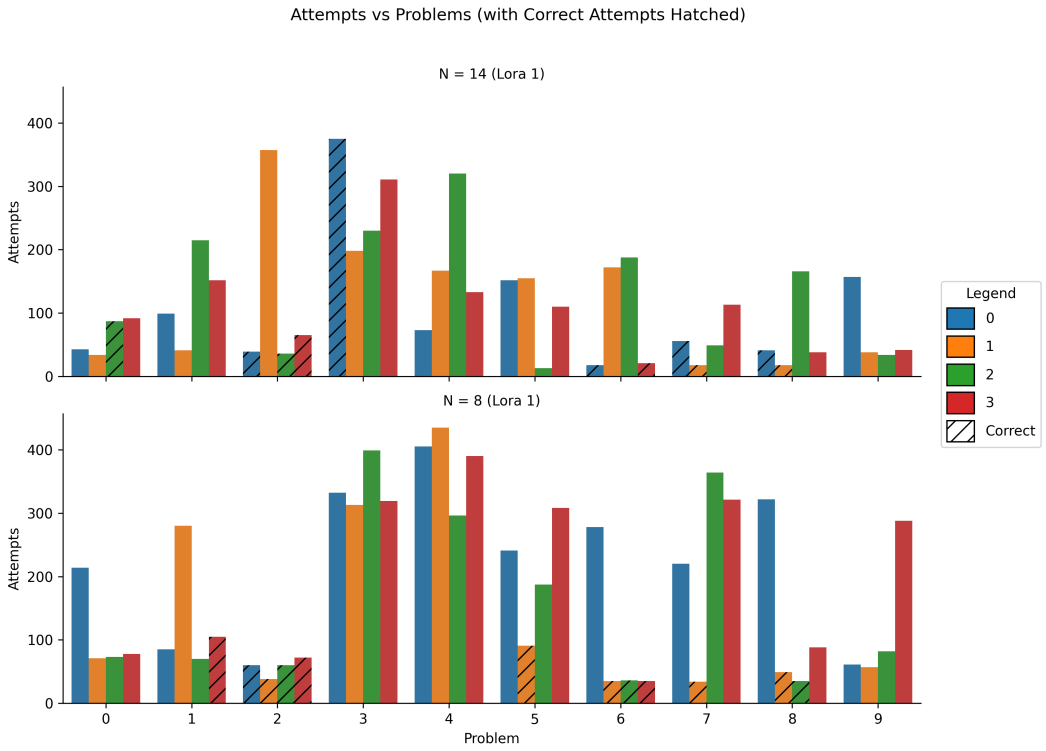

Figure 8.9: In this figure we can see two bar plots, one with 14 particles in SMX and another with 8 particles in SMX. The temperature is 0.8 and $p = 0.9$, here we use the second Lora (Lora 1).

variation in proof continuations. This diversity better leverages the parallelism by avoiding redundant or overly similar proof continuations across the parallel runs.

The plots in Figure 8.11 illustrate how accuracy changes when we increase the p in the top-p inference method for two different temperature settings. In the left plot, we observe that the accuracy goes down when p increases. The underlying reason behind this is unclear; however, given the relatively small difference, it may be the randomness inherent in the SMC algorithm, or as we will see in Figure 8.14 it may be a consequence of many failing SMX instances. In contrast, when the temperature is larger (0.95 compared to 0.8) as in the right plot, we can see that the accuracy increases significantly as p increases. This adds even more evidence to the intuition that further increasing diversity leads to better accuracy.

In Figure 8.12, we examine how the number of attempts varies across two different temperature values when using 5 number of particles. The number of attempts does not differ significantly between the two bar plots. However, for problem 5, we see that the number of attempts for one of the SMX instances is notably higher in the lower bar plot than that of any of the bar plots for problem 5 in the upper bar plot. Despite this, the differences in the two bar plots are not significant enough overall to conclude that the number of attempts

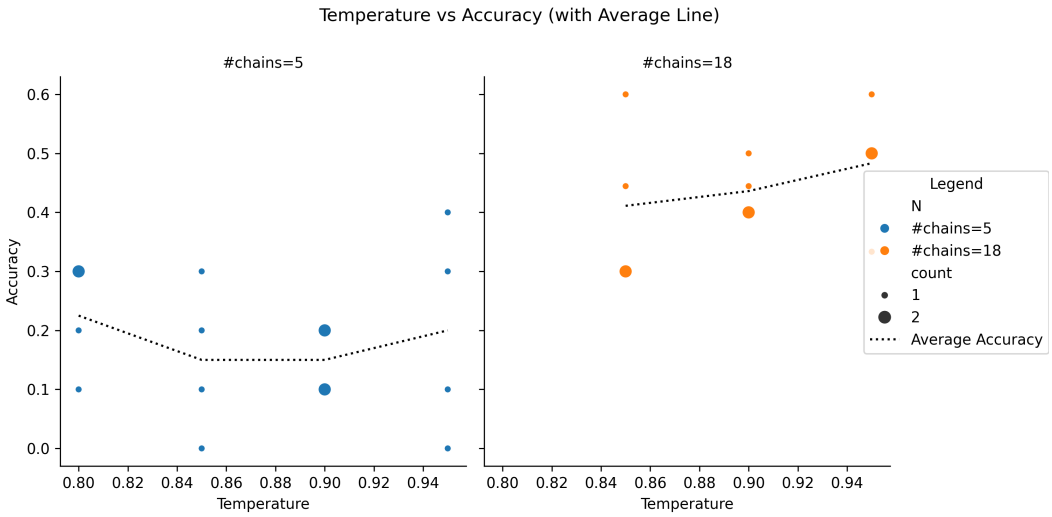

Figure 8.10: In this figure we can see the accuracy over different temperature values. Here as default values we have $p = 0.9$ and we use the latest Lora checkpoint (14). The numbers 1, 2, and 3 indicate how often a specific accuracy value was observed.

increase as the temperature increases when we have 5 particles.

In contrast to $N = 5$, for $N = 18$ in Figure 8.13, we see that in some of the problems, significantly more attempts are made in the lower bar plot belonging to the case where the temperature is 0.95 for problem 3 and 4. However despite this, problem 4 has not been solved at all in the lower bar plot whereas in the upper bar plot 2 out of 4 instances were able to correctly solve it. But for exercises 0, 1, 2 and 5 we do see that the exercises are more frequently solved. Showing that the higher temperature does benefit the success rate of the SMX algorithm, which can also be seen in Figure 8.11, but at the cost of doing more attempts.

Figure 8.14 shows for two values of top-p the number of attempts for the different problems. In the lower barplot of this figure, we see that a lot of problems have missing bars due to the SMX algorithm instances not successfully starting for these problems. This failure can partly explain why in the left plot of Figure 8.11 the accuracy is lower for the higher value of p ($p = 0.95$). The large number of failed instances reduces the reliability of the accuracy estimate for this setting.

The top bar plot in Figure 8.15 also indicates that the right plot in Figure 8.11 is unreliable in terms of its starting accuracy at $p = 0.8$. Meaning, that while the general pattern found in 8.11 is accurate, its steepness most likely is not.

Discussion

In the discussion of the second research question (Section 8.2.2), we established that reat-tempts likely stagnate because the algorithm relies on "crutches"actions that are safe but do not contribute to achieving the goal. In other words, it generates actions that avoid errors

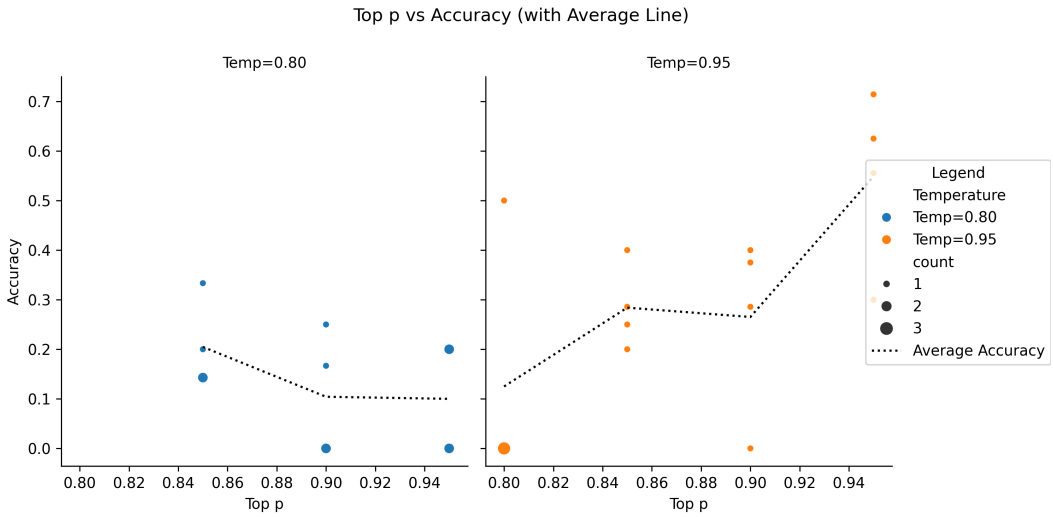

Figure 8.11: In this figure we can see the accuracy over different values of top-p. We use the latest Lora checkpoint (14) and we use as the temperature hyperparameter 0.8. The number of particles here is $N = 14$. The numbers 1, 2, and 3 indicate how many times a specific accuracy value was observed.

but fail to make progress. This explanation is supported by Figures 8.12 and 8.13. With increased temperature, less likely token continuations are explored more frequently, resulting in significantly more attempts when the number of particles is 18 compared to 5 or even 14.

Hence, despite resampling steps leading to states with fewer reattempts when the number of particles is large, higher temperatures introduce variability that counteracts this effect, leading to increased reattempts even in those states.

As was shown with example 8.16 and example 8.17 in the results section, states with high weights but low likelihood for requiring reattempts are often states where the same tactic is applied repeatedly. This observation also explains why a high temperature allows the algorithm to more easily break out these repetition loops and avoids entering them in the first place. Problems with repetition is also found with the inference method called beam search, as discussed in section 2.10, which bears similarity with our method. This provides additional evidence supporting the hypothesis that repetition is a contributing factor.

Regarding, how the combined effect of Top p and temperature influences the accuracy and efficiency, Figure 8.11 shows that with a temperature of 0.8, increasing top-p has a smaller impact on accuracy compared to a temperature of 0.95. However, this result is should be interpreted with caution, as Figure 8.14 and 8.15 show that there are a large number of failing processes, which can distort the accuracy estimation.

Nevertheless, some notable trends show up. In Figure 8.15, the number of attempts reaches to more than 500 in the when top-p is 0.8 and the temperature is 0.95. This implies that in this case, the algorithm avoids looping on repeated tactics, although it still fails to reach a proof. In contrast, the lower bar plot of the same figure 8.15 shows configurations with a top-p of 0.95 that makes use of fewer attempts but achieve relatively high accuracy,

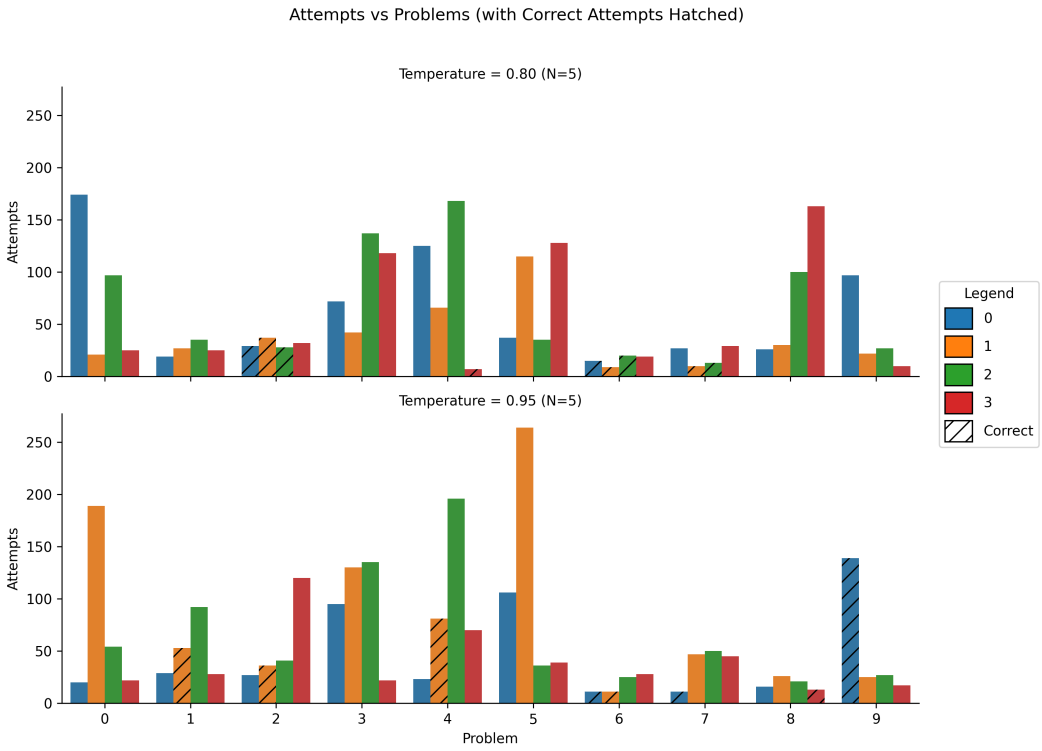

Figure 8.12: In this figure we can see two bar plots one showing the number of attempts when SMX is applied with a temperature of 0.8 and the other for when SMX is applied with a temperature of 0.95. We use for the number of particles $N = 5$ we furthermore have $p = 0.9$ and use the latest checkpoint (14).

as shown in the right plot in Figure 8.11. Furthermore, configurations with higher top-p values have minimal failed processes, which makes their accuracy estimates more reliable.

One possible explanation for this discrepancy in the number of attempts is that higher top-p values retain less likely tokens necessary for generating valid tactics. These tokens could be less likely, which are still necessary to express the right tactic. However, more research is needed to validate this hypothesis.

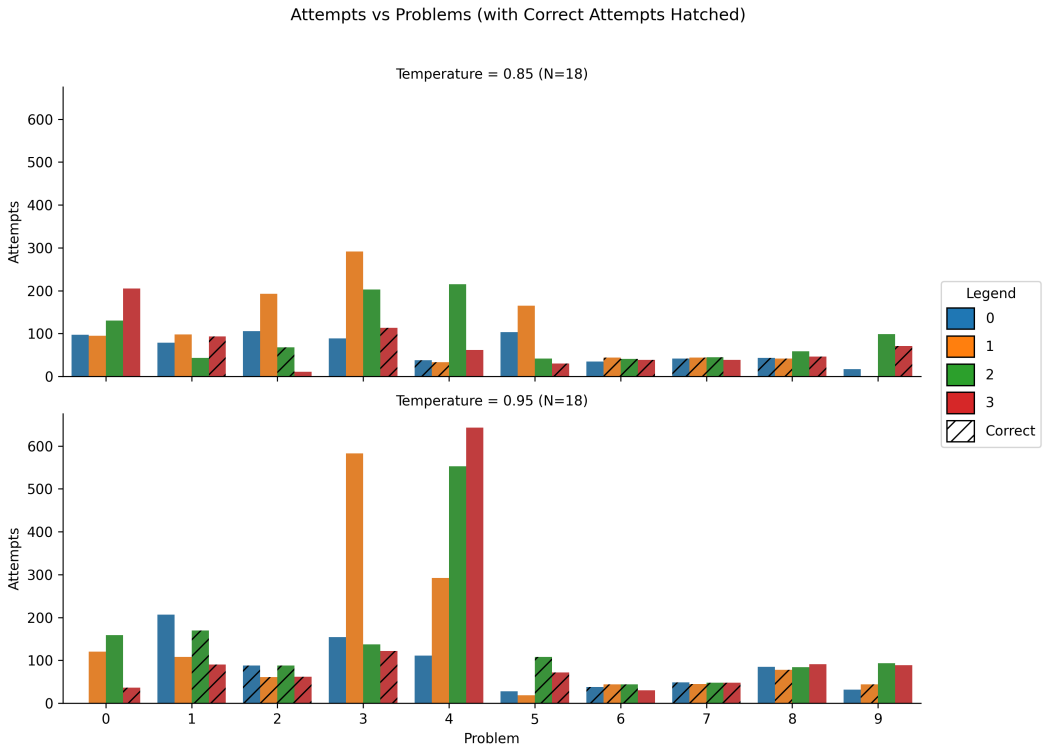

Figure 8.13: In this figure we can see a comparison of two bar plots with the number of attempts being shown for the 10 test problems. Here we have 18 for the number of particles and $p = .9$, here too we use the latest Lora checkpoint. The temperatures are 0.85 and 0.95 respectively.

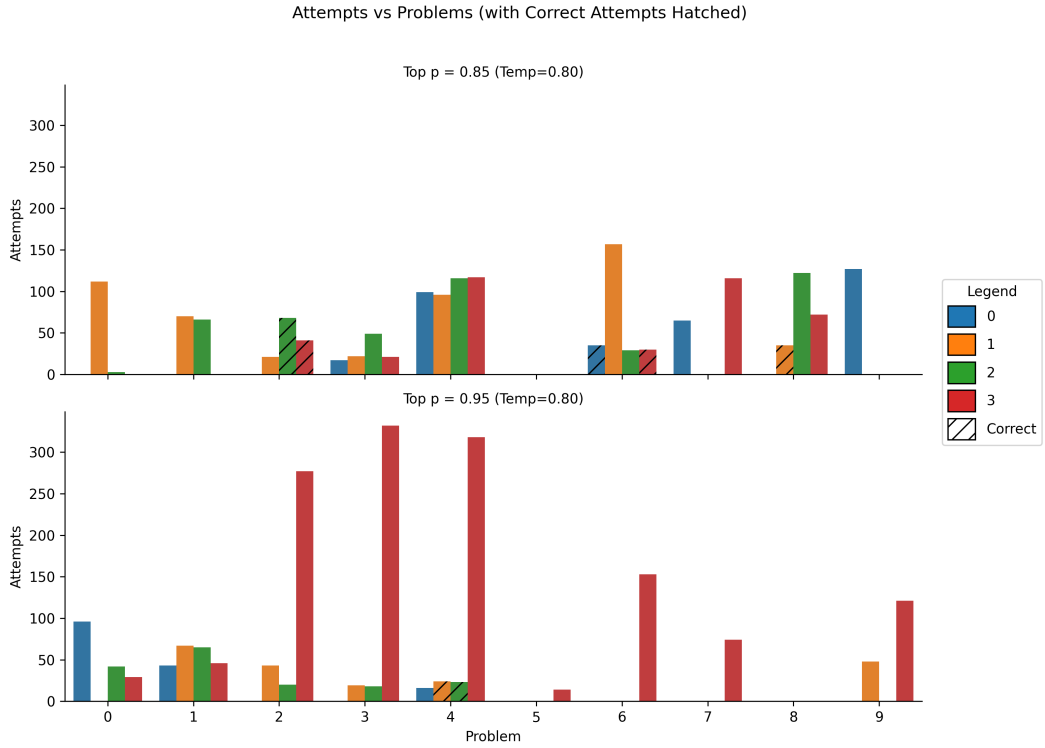

Figure 8.14: In this figure we can see two bar plots corresponding to two different SMX algorithm runs with different values for p ($p = 0.85$ and $p = 0.95$). We use the latest Lora checkpoint and a temperature of 0.8.

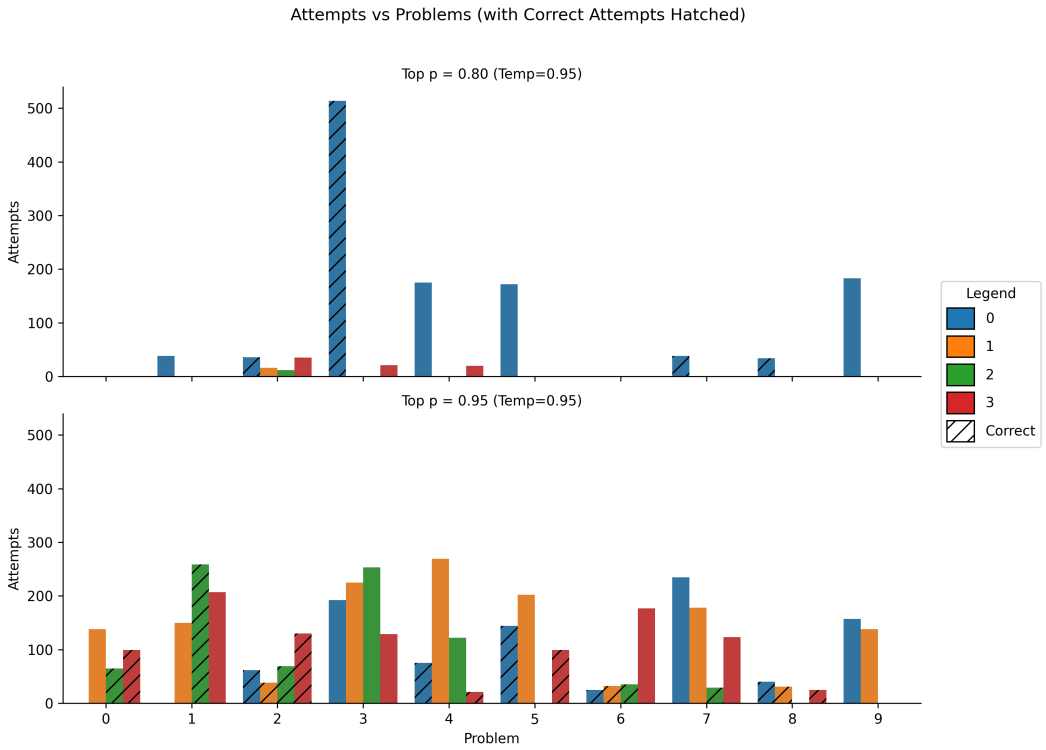

Figure 8.15: In this figure we see two bar plots corresponding to two different running the SMX algorithm across problems with two different set of hyperparameters. The top one has $p = 0.8$ and the bottom one has $p = 0.95$. The temperature is 0.95, we use the latest Lora checkpoint and we use 14 as our number of particles.

```
import Mathlib.Tactic.Cases
open Classical
variable (p q r : Prop)

example : ¬ (p ∧ q) → ¬ p ∨
    ¬ q := by
  intro h
  have hk : ¬ (¬ p) := by
    intro hp
    have hpr : ¬ r := by
      intro hpr
      skip
      skip
      have hq : ¬ q := by
        have hq : ¬ q := by
          intros hpr
          sorry
        have hq : ¬ q := by
          sorry
        sorry
      sorry
    skip
    have hq : ¬ q := by
      sorry
    sorry
  sorry
```

Figure 8.16: Proof of De Morgan's law in Lean

```
import Mathlib.Tactic.Cases
open Classical
variable (α : Type) (p q : α
    → Prop)

example : (∀ x, p x) ∨ (∀ x,
   q x) →
  ∀ x, p x ∨ q x := by
  intros x1
  cases x1
  intro x
  intros
  intros
  solve_by_elim
  intro x
  have hp : p x := by
    have hp : p x := by
      have hp : p x := by
        skip
        skip
        have hp : p x := by
          have hp : p x := by
            have hp1 : p x :=
  by
              sorry
            sorry
          sorry
        sorry
      sorry
    sorry
  intros
  have Goal : p x ∨ q x :=
   by
    sorry
  sorry
```

Figure 8.17: Example of universal and existential quantification in Lean

Chapter 9

Conclusions and Future Work

This chapter gives an overview of the project's contributions. After this overview, we will reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

9.1 Contributions

9.1.1 Better Inference with SMC for Theorem Proving

Firstly, we propose and implement an inference technique that combines Sequential Monte Carlo (SMC) sampling with standard sampling strategies (e.g. top-p and temperature scaling) to promote exploring diverse reasoning paths during inference. We specifically applied the SMC inference method to the domain of formal theorem proving in LEAN4.

Secondly, we investigate the impact of top-p and temperature scaling values on the accuracy and efficiency of the proposed SMC algorithm.

Thirdly, we analyzed the impact of scaling the number of particles in SMC, we showed that there is a trade-off between computational cost and model performance.

9.1.2 Theoretical Contributions to Expecation Maximization and SMC

Fourthly, we provide a rigorous derivation of the E-step for the Expectation Maximization framework, starting from the Evidence Lower Bound. This derivation makes the assumptions and notational simplifications clear that were not detailed in Macfarlane et al. [27].

9.2 Conclusions

This thesis investigated the application of the SMX algorithm for formal theorem proving in LEAN4. The proposed algorithm is based on a combination of the Expectation Maximization algorithm and Sequential Monte Carlo. During the E-step, tactics are being run in parallel using SMC with particles representing the different trajectories. This works because the resampling steps in the SMC algorithm make successful trajectories replicate, while filtering out those that lead nowhere.

The interface used to check the validity of LEAN4 tactics was made to additionally support `cases` and `have` tactics. Moreover, this thesis also investigated the impact of supervised fine-tuning and hyperparameter configurations on the efficiency and accuracy of formal theorem proving using the simplified SMX algorithm.

Our results indicate that supervised fine-tuning has minimal impact on computational efficiency and accuracy, likely due to limited dataset size. While the training loss decreases over iterations, this effect is primarily due to overfitting. An iterative approach that alternates the E-step and fine-tuning on newly generated data (M-step) could provide better generalization.

The number of particles in the SMX method strongly influences accuracy. Large particle counts improve performance by reducing the likelihood of getting stuck in unsuccessful search paths.

Regarding the interplay between top-p and temperature, higher top-p values appear to stabilize accuracy, but the presence of failed processes complicates the interpretation. The number of attempts increases significantly when both top-p and temperature are high, potentially preventing loops in tactic generation. This suggests that these parameters affect not only accuracy but also search behavior. More research is needed to validate this hypothesis.

Finally, our observations align with known challenges in inference strategies like beam search, where repetition is a common issue.

9.3 Discussion/Reflection

As can be seen in the results section of this thesis some of the experiments failed to start. This is most likely caused by the large language model server `llama.cpp` taking more time to load than was given as startup time. Furthermore, in some cases the server disconnected randomly, while the SMX instance did start successfully. So choosing a different large language model server such as `SGlang` [56] would have been better. Moreover, the output of the LLM is required to be parsable as a `json`, as shown in Appendix A.1, the way large language models are prompted matters significantly, and with `SGlang` structured generation would have been possible, which would make the output of the large language model faster, more reliable and more accurate. Additionally in the current setup we had to load in the Large Language Model with different LoRA checkpoints in a cumbersome manner where the LoRA adapter had to be fused with the base model then the server had to be shut down to reload the model with a different LoRA checkpoint. In `SGlang` a list of different LoRA adapters can be given so that the model does not have to be reloaded from disk every time a different LoRA should be used.

Letting the LLM predict the code on a tactic-by-tactic basis complicated the interface significantly, and took a significant amount of time to develop. What Xin et al. [52] did is let the large language model continue writing until it produces the end token. This is then truncated up to the first error. So instead of going tactic-by-tactic, we would go from correct code fragment to correct code fragment. In addition to adapting the interface that was created by Xin et al. [52], the pretrained model the authors released alongside would also have been a significant improvement. Unfortunately, this work was made concurrently

with this thesis, when Xin et al. [52] was released the LEAN4 interface and fine-tuning code were already written.

Lastly, In the conclusion section, we can see that the fine-tuning had little to no effect. This is likely because of the size of the dataset. If instead of the tactic-by-tactic approach, we took the correct code fragment to correct code fragment approach we could have used a much larger dataset to train on since we would have had less stringent requirements on what syntax would have been allowed.

9.4 Future work

This thesis opens up several interesting directions for future research:

9.4.1 Implementing the M-Step

The most straightforward extension of this work would be to implement the M-step described in Section 5.2.4:

$$\max_{\theta} \mathbb{E}_{\mu_{q_i}(s)} \left[\mathbb{E}_{q_i(a|s)} \left[\log \pi(a|s, \theta) \right] \right]$$
$$\text{s.t. } \mathbb{E}_{\mu_{q_i}(s)} \left[\mathrm{KL} \left(\pi(a|s, \theta_i), \pi(a|s, \theta) \right) \right] < \varepsilon_m$$

This would require training the model to predict the Value of the current and previous state, in order to accumulate these values in the weights generated through the Sequential Monte Carlo search algorithm. A possible approach would be to use Generalized Advantage Estimation (GAE) [39] as was also recommended in the SMX paper Macfarlane et al. [28].

9.4.2 Generalizing Reasoning Validation

While the focus of this thesis is on formal theorem proving, the methodology combining EM with SMC could be generalized to reasoning tasks more broadly. Approaches like Chain of Thought (CoT) reasoning [51] have shown that explicitly breaking down problems into intermediate reasoning steps improves the performance of large language models on complex tasks. CoT involves generating sequences of intermediate "thoughts" that guide the model toward the solution, similar to how humans reason step by step. Recent work, such as Team [45] improves open source developments of Chain of Thought models, leveraging these models to fine-tune on top off could significantly improve performance.

A suggested extension would be to treat "thoughts" in a chain-of-thought process as actions, similar to how LEAN4 tactics were treated in this thesis. Each action could then be evaluated by a Process-Reward Model (PRM), which assigns rewards based on the correctness of a certain step. The PRM itself can be trained using methods described in Zhang et al. [55]. However, instead of relying on LLM-as-judge approaches to verify the reward estimation, the verification would be done formally in the LEAN4 language, in the same way as was introduced in this thesis, but with the modifications that were given in the discussion section. One challenge with this approach is that the large language model has to not only solve the problem but also generate the corresponding LEAN4 problem statement,

which may or may not be correct. This would require supervision during training to ensure correctness.

9.4.3 Fine-Tuning for Specialized Applications

As shown in this thesis, a single model can be used for multiple functions (e.g., policy model, reward model). However, in the proposed improvements given in this section, we propose that the model would take on the role of the policy model, value function, and PRM, in which case the model would benefit from an improved fine-tuning technique like the method called Transformer^2 in Sun et al. [41], that works especially well in a multi-task scenario such as this.

Bibliography

[1] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023. URL https://arxiv.org/abs/2305.13245.

[2] Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. Theorem proving in lean 4. https://lean-lang.org/theorem_proving_in_lean4/, 2025. Accessed: 2025-01-16.

[3] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report, 2023. URL https://arxiv.org/abs/2309.16609.

[4] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:11371155, March 2003. ISSN 1532-4435.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

[6] Kevin Buzzard, Johan Commelin, and Patrick Massot. Lean perfectoid spaces. https://leanprover-community.github.io/lean-perfectoid-spaces/type_theory.html, 2025. Accessed: 2025-01-16.

[7] Guoxin Chen, Minpeng Liao, Chengxi Li, and Kai Fan. Alphamath almost zero: Process supervision without process, 2024. URL https://arxiv.org/abs/2405.03553.

[8] PyTorch Contributors. torch.nn.softmax. https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html, 2025. Accessed: 2025-01-16.

[9] Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, Jiarui Yuan, Huayu Chen, Kaiyan Zhang, Xingtai Lv, Shuo Wang, Yuan Yao, Hao Peng, Yu Cheng, Zhiyuan Liu, Maosong Sun, Bowen Zhou, and Ning Ding. Process reinforcement through implicit rewards. https://curvy-check-498.notion.site/Process-Reinforcement-through-Implicit-Rewards-15f4fcb9c42180f1b498cc9b2eaf896f, 2025. Notion Blog.

[10] DeepSeek-AI, :, Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao, Zhewen Hao, Ying He, Wenjie Hu, Panpan Huang, Erhang Li, Guowei Li, Jiashi Li, Yao Li, Y. K. Li, Wenfeng Liang, Fangyun Lin, A. X. Liu, Bo Liu, Wen Liu, Xiaodong Liu, Xin Liu, Yiyuan Liu, Haoyu Lu, Shanghao Lu, Fuli Luo, Shirong Ma, Xiaotao Nie, Tian Pei, Yishi Piao, Junjie Qiu, Hui Qu, Tongzheng Ren, Zehui Ren, Chong Ruan, Zhangli Sha, Zhihong Shao, Junxiao Song, Xuecheng Su, Jingxiang Sun, Yaofeng Sun, Minghui Tang, Bingxuan Wang, Peiyi Wang, Shiyu Wang, Yaohui Wang, Yongji Wang, Tong Wu, Y. Wu, Xin Xie, Zhenda Xie, Ziwei Xie, Yiliang Xiong, Hanwei Xu, R. X. Xu, Yanhong Xu, Dejian Yang, Yuxiang You, Shuiping Yu, Xingkai Yu, B. Zhang, Haowei Zhang, Lecong Zhang, Liyue Zhang, Mingchuan Zhang, Minghua Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, Qihao Zhu, and Yuheng Zou. Deepseek llm: Scaling open-source language models with longtermism, 2024. URL https://arxiv.org/abs/2401.02954.

[11] Hugging Face. Llama 3 - hugging face transformers documentation, 2024. URL https://huggingface.co/docs/transformers/en/model_doc/llama3. Accessed: 2025-03-13.

[12] Xidong Feng, Ziyu Wan, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training, 2024.

[13] Friendli.ai. Gqa vs. mha: Understanding grouped-query attention in llama-2. https://friendli.ai/blog/gqa-vs-mha, 2023. Accessed: October 11, 2024.

[14] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yi-

wen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy,

Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

[15] Simrat Hanspal. Decoding llama3: Part 6 - feed forward network. https://hasgeek.com/simrathanspal/the-llama3-guide/sub/decoding-llama3-part-6-feed-forward-network-NSnKTtJvbYSqm7vFL8996f, 2024. Accessed: 2025-01-16.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL https://arxiv.org/abs/1512.03385.

[17] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015. URL https://arxiv.org/abs/1503.02531.

[18] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration, 2020. URL https://arxiv.org/abs/1904.09751.

[19] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

[20] Umar Jamil. Pytorch llama notes, 2023. URL https://github.com/hkproj/pytorch-llama-notes. Accessed: October 14, 2024.

[21] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

[22] Thomas Kehrenberg. Basic building blocks of dependent type theory. https://www.lesswrong.com/posts/ccbsYSpTcTqXwukH8/basic-building-blocks-of-dependent-type-theory, 2022. Accessed: 2025-01-16.

[23] Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving, 2022.

[24] S.R. Lay. *Analysis: With an Introduction to Proof*. Always learning. Pearson, 2014. ISBN 9780321904416. URL https://books.google.nl/books?id=08G5NAEACAAJ.

[25] Runze Liu, Junqi Gao, Jian Zhao, Kaiyan Zhang, Xiu Li, Biqing Qi, Wanli Ouyang, and Bowen Zhou. Can 1b llm surpass 405b llm? rethinking compute-optimal test-time scaling, 2025. URL https://arxiv.org/abs/2502.06703.

[26] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

[27] Matthew V Macfarlane, Edan Toledo, Donal Byrne, Paul Duckworth, and Alexandre Laterre. Spo: Sequential monte carlo policy optimisation, 2024. URL https://arxiv.org/abs/2402.07963.

[28] Matthew V Macfarlane, Edan Toledo, Donal Byrne, Siddarth Singh, Paul Duckworth, and Alexandre Laterre. Smx: Sequential monte carlo planning for expert iteration, 2024.

[29] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022. URL https://arxiv.org/abs/2203.02155.

[30] Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning, 2023.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL https://arxiv.org/abs/1912.01703.

[32] Matt Payne. What is beam search? explaining the beam search algorithm, September 2021. URL https://www.width.ai/post/what-is-beam-search. Accessed: 2025-01-14.

[33] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020. URL https://arxiv.org/abs/2009.03393.

[34] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning, 2022. URL https://arxiv.org/abs/2202.01344.

[35] Xena Project. No confusion over no_confusion. https://xenaproject.wordpress.com/2018/03/24/no-confusion-over-no_confusion/, 2018. Accessed: 2025-01-16.

[36] PyTorch Contributors. torch.nn.silu. https://pytorch.org/docs/stable/generated/torch.nn.SiLU.html, 2024. Accessed: 2024-10-14.

[37] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[38] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI*, 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf. Accessed: 2024-11-15.

[39] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018. URL https://arxiv.org/abs/1506.02438.

[40] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

[41] Qi Sun, Edoardo Cetin, and Yujin Tang. Transformer^2: Self-adaptive llms, 2025. URL https://arxiv.org/abs/2501.06252.

[42] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

[43] James Tassarotti. coq-probrec: Coq development for "verified tail bounds for randomized programs", 2018. URL https://github.com/jtassarotti/coq-probrec. GitHub repository, accessed: March 17, 2025.

[44] Joseph Tassarotti and Robert Harper. Verified tail bounds for randomized programs. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, pages 560–578, Cham, 2018. Springer International Publishing. ISBN 978-3-319-94821-8.

[45] NovaSky Team. Sky-t1: Train your own o1 preview model within $450, 2025. URL https://novasky-ai.github.io/posts/sky-t1. Accessed: 2025-01-09.

[46] Ye Tian, Baolin Peng, Linfeng Song, Lifeng Jin, Dian Yu, Haitao Mi, and Dong Yu. Toward self-improvement of llms via imagination, searching, and criticizing, 2024.

[47] torchtune maintainers and contributors. torchtune: Pytorch's finetuning library, April 2024. URL https//github.com/pytorch/torchtune.

[48] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023. URL https://arxiv.org/abs/2302.13971.

[49] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

[51] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

[52] Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search, 2024. URL https://arxiv.org/abs/2408.08152.

[53] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS)*, 2023.

[54] Biao Zhang and Rico Sennrich. Root mean square layer normalization, 2019. URL https://arxiv.org/abs/1910.07467.

[55] Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. The lessons of developing process reward models in mathematical reasoning, 2025. URL https://arxiv.org/abs/2501.07301.

[56] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Appendix A

Implementation details

A.1 Interface

In figure A.1 we can see how the input gets transformed from LEAN4 code to REPL output using the LEAN4 REPL which in turn gets processed by the LEAN4 interface to more human/LLM readable form. We can see in the human readable form that we give the LEAN4

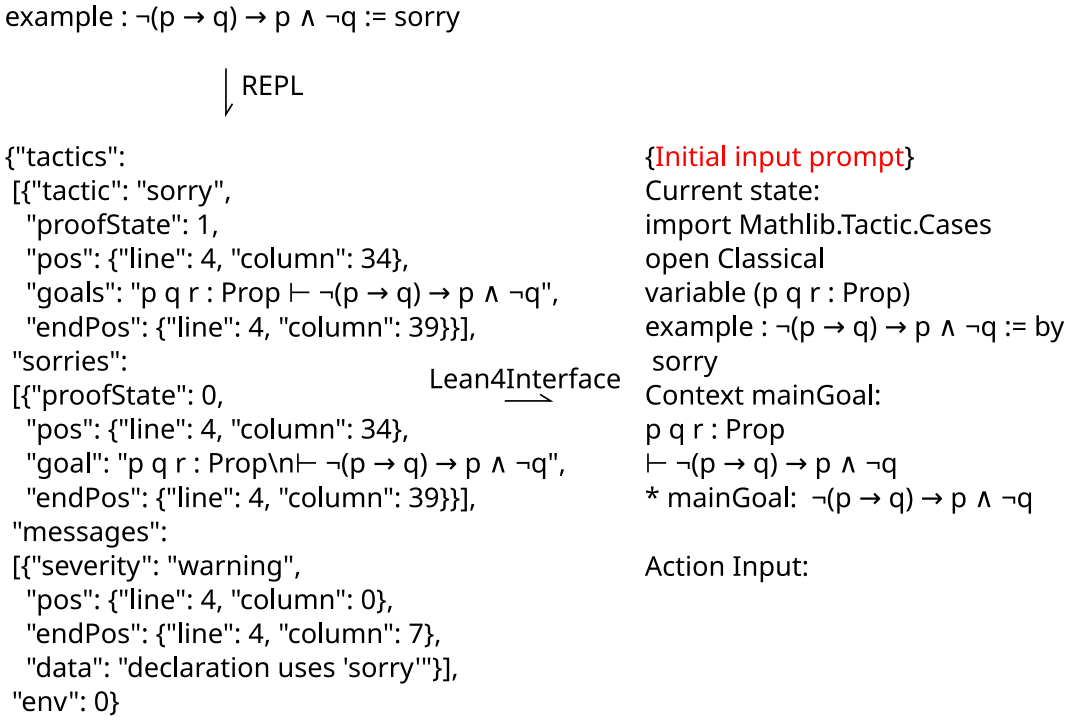

Figure A.1: In this figure we can see how the code gets processed into human readable form

code that contains the tactics that are applied up to this point. We furthermore see the that the context is provided for the goal called `mainGoal`. If there are multiple sub-goals, then for all of these sub-goals we display their context. This context contains the relevant vari-

ables that have been declared up to this point and the goal itself. We furthermore enlist the goals with their name and the type of the goal. If the large language model wants to use a tactic on one of the goals, it has to output the target goal and the tactic in json format. An example of this can be seen below:

```
{"tactic": "intro hnPtoQ", "Goal": "mainGoal"}
```

Appendix B

Proofs

In this chapter, we provide the derivations from Macfarlane et al. [27]. Since the original paper does not include these derivations, our results may differ in some cases. These differences will be discussed in the main text.

B.1 Derivation E-step

Let us recall from the main text that

$$p(O = 1|\tau) \propto \exp\left(\sum_t \gamma^t r_t/\alpha\right)$$

from the inference-based formulation, thus

$$p(O = 1|\tau) = \frac{\exp\left(\frac{1}{\alpha}\sum_t \gamma^t r_t\right)}{Z(\alpha)}$$

where $Z(\alpha)$ is a normalization constant independent of τ. By introducing an auxiliary distribution $q(\tau)$ and applying Jensen's inequality we obtain,

$$\log p_\pi(O = 1) \geq \mathbb{E}_{q(\tau)}\left[\log \frac{p(O = 1|\tau)\pi(\tau)}{q(\tau)}\right]$$

Then substituting $p(O = 1|\tau) = \frac{\exp\left(\frac{1}{\alpha}\sum_t \gamma^t r_t\right)}{Z(\alpha)}$, we get

$$\log p_\pi(O = 1) \geq \mathbb{E}_{q(\tau)}\left[\frac{1}{\alpha}\sum_t \gamma^t r_t + \log \pi(\tau) - \log q(\tau) - \log(Z(\alpha))\right]$$

Since $\log(Z(\alpha))$ is constant with respect to q, it can be dropped from the optimization perspective. Hence

$$\log p_\pi(O = 1) \geq \mathbb{E}_{q(\tau)}\left[\frac{\sum_t \gamma^t r_t}{\alpha} - \log \frac{q(\tau)}{\pi(\tau)}\right]$$

Now we furthermore assume q and π can be factorized as:

$$\pi(\tau) = \mu(s_0)\Pi_{t=0}^{T}\pi(a_t|s_t)\mathcal{T}(s_{t+1}|s_t, a_t)$$

$$q(\tau) = \mu(s_0)\Pi_{t=0}^{T}q(a_t|s_t)\mathcal{T}(s_{t+1}|s_t, a_t)$$

The transitions $\tau(s_{t+1}|s_t, a_t)$ and initial distribution $\mu(s_0)$ appear in both $\pi(\tau)$ and $q(\tau)$. Thus, in the ratio $\frac{q(\tau)}{\pi(\tau)}$, these terms cancel out:

$$\frac{q(\tau)}{\pi(\tau)} = \frac{\Pi_t q(a_t|s_t)}{\Pi_t \pi(a_t|s_t)}$$

Hence:

$$\log \frac{q(\tau)}{\pi(\tau)} = \sum_t \log \frac{q(a_t|s_t)}{\pi(a_t|s_t)}$$

Then substituting this back into the inequality:

$$\log p_\pi(O=1) \geq \mathbb{E}_{q(\tau)}\left[\frac{1}{\alpha}\sum_t \gamma^t r_t - \sum_t \log \frac{q(a_t|s_t)}{\pi(a_t|s_t)}\right]$$

Our next steps will be to establish

$$\log p_\pi(O=1) \geq \sum_s \mu(s) \sum_a q(a|s)\left[\frac{Q^q(s,a)}{\alpha}\right] - (T+1)\sum_s d_q(s)\sum_a q(a|s) KL(q(a|s)||\pi(a|s))$$

where

$$d_q(s) = \frac{1}{T+1}\sum_{t=0}^{T} q(s_t = s)$$

which is also called the discounted state visitation distribution under q,

$$q(s_t = s) = \sum_{s_0, a_0, \ldots, s_{t-1}, a_{t-1}} \mu(s_0)\Pi_{t'=0}^{t-1} q(a_{t'}|s_{t'})\mathcal{T}(s_{t'+1}|s_{t'}, a_{t'})\mathcal{T}(s_t|s_{t-1}, a_{t-1})$$

and

$$q(s_t, a_t) = \sum_{s_0, a_0, \ldots, s_{t-1}, a_{t-1}} \mu(s_0)\Pi_{i=0}^{t-1} q(a_i|s_i)\mathcal{T}(s_{i+1}|s_i, a_i) q(a_t|s_t)\mathcal{T}(s_t|s_{t-1}, a_{t-1})$$

Furthermore, to proceed with the following steps it is important to clarify how $\mathbb{E}_{q(\tau)}[\cdot]$ gets computed for $g(s_t, a_t)$, $\mathbb{E}_{q(\tau)}[g(s_t, a_t)]$ becomes

$$\begin{aligned}
\mathbb{E}_{q(\tau)}[g(s_t, a_t)] &= \sum_{s_0, a_0, \ldots, s_T, a_T} g(s_t, a_t) q(s_0, a_0, \ldots, s_T, a_T) \\
&= \sum_{s_t, a_t} g(s_t, a_t) q(s_t, a_t) \\
&=: \mathbb{E}_{s_t, a_t}[g(s_t, a_t)]
\end{aligned}$$

In the second equality we use the definition of $q(s_t, a_t)$, and in the last line we defined $\mathbb{E}_{s_t,a_t}[\cdot]$. Let us recall another definition $Q^q(s, a)$,

$$Q^q(s,a) = \mathbb{E}_{\tau\sim q}\left[\sum_{t=0}^{T} \gamma^t r_t \,|s_0 = s, a_0 = a\right]$$

as also mentioned in the main text the Q-function $Q^q(s, a)$ represents the expected cumulative reward when starting in state s, taking action a, and then following a policy defined by the distribution q for subsequent steps [42]. $\tau = (s_0, a_0, \ldots, s_T, a_T)$ again is a trajectory sampled according to q. $\gamma \in [0, 1]$ is the discount factor, determining the relative importance of future rewards. r_t is the reward obtained at time-step t. We can rewrite $\mathbb{E}_{q(\tau)}\left[\sum_{t=0}^{T} \frac{\gamma^t r_t}{\alpha}\right]$ into incorporating $Q^q(s_0, a_0)$:

$$\begin{aligned}
\mathbb{E}_{q(\tau)}\left[\frac{\sum_{t=0}^{T}\gamma^t r_t}{\alpha}\right] &= \sum_{s_0,a_0} q(s_t, a_t)\cdot\left(\frac{\sum_{t=0}^{T}\gamma^t r_t}{\alpha}\right) \\
&= \mathbb{E}_{s_0,a_0}\left[\mathbb{E}_{\tau\sim q}\left[\sum_{t=0}^{T}\frac{\gamma^t r_t}{\alpha}\,\middle|\, s_0 = s, a_0 = a\right]\right] \\
&= \frac{1}{\alpha}\mathbb{E}_{s_0,a_0}[Q^q(s_0, a_0)]
\end{aligned}$$

We used the total law of expectation from the first to the second line and the definition of the Q-function from the second to the third line. Then, let us recall one last definition: the KL divergence. The KL divergence is defined as:

$$\mathrm{KL}(q(a_t|s_t)||\pi(a_t|s_t)) = \sum_{a_t} q(a_t|s_t)\log\frac{q(a_t|s_t)}{\pi(a_t|s_t)}$$

The Kullback-Leibler divergence is a measure of how one probability distribution q differs from another distribution π, as was already discussed in the main text. We furthermore want

to rewrite the term: $\sum_t \mathbb{E}_{s_t,a_t}\left[\log \frac{q(a_t|s_t)}{\pi(a_t|s_t)}\right]$ to one that contains the KL divergence term.

$$
\begin{aligned}
&\sum_t \mathbb{E}_{s_t,a_t}\left[\log \frac{q(a_t|s_t)}{\pi(a_t|s_t)}\right] \\
&= \sum_t \sum_{s_t,a_t} \log \frac{q(a_t|s_t)}{\pi(a_t|s_t)} \cdot q(s_t,a_t) \\
&= \sum_t \sum_{s_t,a_t} \log \frac{q(a_t|s_t)}{\pi(a_t|s_t)} q(s_t) q(a_t|s_t) \\
&= \sum_t \sum_{s_t} q(s_t) \mathrm{KL}(q(a_t|s_t)||\pi(a_t|s_t)) \\
&= \sum_t \sum_{s} q(s_t = s) \mathrm{KL}(q(a|s)||\pi(a|s)) \\
&= \sum_s \mathrm{KL}(q(a|s)||\pi(a|s)) \left(\sum_t q(s_t = s)\right) \\
&= \sum_s \mathrm{KL}(q(a|s)||\pi(a|s)) \cdot (T+1) d_q(s)
\end{aligned}
$$

Now, we want to rewrite the expectation $\mathbb{E}[Q^p(s_0,a_0)]/\alpha$, using $q(s,a) = q(s)q(s|a)$:

$$
\begin{aligned}
&\mathbb{E}_{s_0,a_0}[Q^q(s_0,a_0)]/\alpha \\
&= \sum_{s_0,a_0} Q^q(s_0,a_0) q(s_0) q(a_0|s_0) \\
&= \sum_{s_0} q(s_0) \sum_a q(a|s_0) Q^q(s_0,a)
\end{aligned}
$$

In the last step, we removed the subscript under a since the action that is taken under distribution q depends only on the current state s_0, not on the time-step t under which the action a was taken. Substituting this into the inequality we obtain:

$$\log p_\pi(O=1) \geq \sum_s \mu(s) \sum_a q(a|s) \left[\frac{Q^q(s,a)}{\alpha}\right] - (T+1) \sum_s d_q(s) \sum_a q(a|s) \mathrm{KL}(q(a|s)||\pi(a|s))$$

Here we use $\mu(s) = q(s_0 = s)$, since μ is the starting state distribution, as we recall from the main text. Then, if we write this in terms of expectations we obtain the desired objective function:

$$\log p_\pi(O=1) \geq \mathbb{E}_{\mu(s)}\left[\mathbb{E}_{q(a|s)}\left[\frac{Q^q(s,a)}{\alpha}\right]\right] - (T+1)\mathbb{E}_{d_q(s)}\left[\mathbb{E}_{q(a|s)}\left[\mathrm{KL}(q(a|s)||\pi(a|s))\right]\right]$$

This objective is similar to the one that is shown as the E-step by Macfarlane et al. [27]. However, since the derivation of this bound is not shown in the paper we cannot determine if the difference results of a different derivation or if their bound is incorrect.

Appendix C

Training

C.1 GAE calculation

Below we can see how GAE gets calculated.

Algorithm 5 Value Function Update via GAE (corresponding to line 21)

Require: Value network $V(s;\phi)$, replay buffer $\mathcal{B}$ containing rollouts, discount factor γ, GAE parameter λ, learning rate β.

function COMPUTEGAEANDTARGETS
  Initialize an empty list ValueTargets.
  for each trajectory τ in $\mathcal{B}$ **do**
    $\tau = \{(s_0, a_0, r_0), (s_1, a_1, r_1), \ldots, (s_T, a_T, r_T)\}$.
    Let $\hat{A}_T = 0$. ▷ Advantage at the terminal is zero
    for $t = T-1$ **down to** 0 **do** ▷ Compute 1-step TD error (a.k.a. δ_t)

$$\delta_t \;=\; r_t + \gamma\,(1 - \text{done}_t)\,V\big(s_{t+1};\phi\big) \;-\; V\big(s_t;\phi\big),$$

▷ GAE update (backward pass)

$$\hat{A}_t \;=\; \delta_t \;+\; \gamma\,\lambda\,(1 - \text{done}_t)\,\hat{A}_{t+1}.$$

    end for
    ▷ Now form the target for each state s_t
    for $t = 0$ **to** T **do**

$$V_{\text{target}}(s_t) \;=\; V\big(s_t;\phi\big) \;+\; \hat{A}_t.$$

      ValueTargets.append$\big(s_t,\; V_{\text{target}}(s_t)\big)$
    end for
  end for
  return ValueTargets
end function

function FITVALUENETWORK(ValueTargets)

$$\text{MSE}(\phi) \;=\; \frac{1}{2} \sum_{(s,\, v_{\text{tgt}}) \,\in\, \text{ValueTargets}} \Big[v_{\text{tgt}} \;-\; V(s;\phi)\Big]^2.$$

  $\phi \;\leftarrow\; \arg\min_\phi \text{MSE}(\phi)$ ▷ In practice, do SGD/Adam steps
  return ϕ
end function

procedure UPDATEVALUEFUNCTION ▷ Corresponds to line 21 in the SPO pseudocode
  ValueTargets = COMPUTEGAEANDTARGETS
  $\phi \leftarrow$ FITVALUENETWORK(ValueTargets)
  return ϕ
end procedure

C.2 Policy update

Algorithm 6 Policy Update with Full Loss (Line 22–23, SPO Algorithm)

Require:

π_{θ_0}: Old (fixed) policy from the previous iteration
π_θ: New policy to be updated (parameters θ)
α: Dual variable enforcing $\mathrm{KL}(\pi_{\theta_0} \| \pi_\theta) \leq \varepsilon$
$\mathcal{B}$: Replay buffer containing (s, a) samples (or advantage weights)
ε: KL threshold
POLICYOPT, ALPHAOPT: Optimizers (e.g. Adam) for θ and α

procedure UPDATEPARAMETERIZEDPOLICY($\pi_\theta, \alpha, \mathcal{B}, \pi_{\theta_0}$)
  batch $\leftarrow$ $\mathcal{B}$.sample(batch_size) ▷ e.g. (s_i, a_i, w_i) for each sample
  $\mathrm{KL} \leftarrow 0, \quad \mathrm{PolicyLoss} \leftarrow 0$
  for each sample i in batch **do**
    $\pi_{\theta_0}(\cdot|s_i) = \textsc{ForwardPass}(\pi_{\theta_0}, s_i)$ ▷ old policy (no gradient)
    $\pi_\theta(\cdot|s_i) = \textsc{ForwardPass}(\pi_\theta, s_i)$ ▷ new policy (learnable)
    $\mathrm{KL}_i = D_{\mathrm{KL}}\big(\pi_{\theta_0}(\cdot|s_i) \,\|\, \pi_\theta(\cdot|s_i)\big)$
      ▷ Compute the "main" policy improvement term, e.g. $-w_i \log \pi_\theta(a_i|s_i)$:
    $\mathrm{PolicyLoss}_i = -w_i \log \pi_\theta(a_i \mid s_i)$
    $\mathrm{KL} \mathrel{+}= \mathrm{KL}_i$
    $\mathrm{PolicyLoss} \mathrel{+}= \mathrm{PolicyLoss}_i$
  end for
  $\mathrm{KL} = \mathrm{KL}/\mathrm{batch\_size}$
  $\mathrm{PolicyLoss} = \mathrm{PolicyLoss}/\mathrm{batch\_size}$
    ▷ Construct $\mathcal{L}_\alpha$ and $\mathcal{L}_{\mathrm{KL}}$:
  $\mathcal{L}_\alpha(\theta, \alpha) = \alpha \left[\varepsilon - \mathrm{KL}\right]$ ▷ α adjusts based on how KL compares to ε
  $\mathcal{L}_{\mathrm{KL}}(\theta, \alpha) = \mathrm{sg}[\alpha]\ \mathrm{KL}$ ▷ penalizes θ for large KL, with α viewed as constant
    ▷ Combine the "main" policy improvement loss with the KL terms:
  $\mathrm{TotalLoss} = \mathrm{PolicyLoss} + \mathcal{L}_\alpha(\theta, \alpha) + \mathcal{L}_{\mathrm{KL}}(\theta, \alpha)$.
  ZEROGRAD(POLICYOPT, ALPHAOPT)
  BACKWARD(TotalLoss) ▷ Compute gradients w.r.t. both θ and α
  STEP(POLICYOPT, ALPHAOPT) ▷ Update θ and α simultaneously
  return $(\theta, \alpha, \mathrm{KL}, \mathrm{PolicyLoss})$
end procedure

C.3 Test set

```
example : p ∨ (q ∧ r) ↔ (p ∨ q) ∧ (p ∨ r) := by
  apply Iff.intro
  case mp =>
    intro hPorQandR
    cases' hPorQandR with hp hQandR
    case inl =>
      have hPorQ : p ∨ q := by
        exact Or.inl hp
      have hPorR : p ∨ r := by
        exact Or.inl hp
      exact ⟨ hPorQ, hPorR ⟩
    case inr =>
      let ⟨ hq, hr ⟩ := hQandR
      exact ⟨ Or.inr hq, Or.inr hr ⟩
  case mpr =>
    intro ⟨ hPorQ, hPorR ⟩
    cases' hPorQ with hp hq
    case inl =>
      apply Or.inl
      exact hp
    case inr =>
      cases' hPorR with hp hr
      case inl =>
        apply Or.inl
        exact hp
      case inr =>
        apply Or.inr
        exact ⟨ hq, hr ⟩
```

```
example : ((p ∨ q) → r) ↔ (p → r) ∧ (q → r) := by
  apply Iff.intro
  case mp =>
    intro hPorQtoR
    constructor
    case left=>
      exact show p → r from (fun hp : p => hPorQtoR (Or.inl
    hp))
    case right=>
      intro hq
      exact show r from (hPorQtoR (Or.inr hq))
  case mpr =>
    intro ⟨ hPtoR, hQtoR ⟩
    intro hPorQ
    cases' hPorQ with hp hq
    case inl =>
      exact hPtoR hp
    case inr =>
      exact hQtoR hq



example : ¬ p → (p → q) := by
  intro hnp
  intro hp
  exact show q from absurd hp hnp



example : ¬ (p ∧ q) → ¬ p ∨ ¬ q := by
  intro hnPQ
  cases' (em p) with hp hnP
  case inl =>
    cases' (em q) with hq hnQ
    case inl =>
      exact False.elim (hnPQ ⟨ hp, hq ⟩ )
    case inr =>
      apply Or.inr
      exact hnQ
  case inr =>
    apply Or.inl
    exact hnP
```

```
example : ¬ (p → q) → p ∧ ¬ q := by
  intro hnPtoQ
  cases' (em p) with hp hnP
  case inl =>
    have hnQ : ¬ q := by
      intro hq
      exact show False from hnPtoQ (fun hp : p => hq)
    exact ⟨ hp, hnQ ⟩
  case inr =>
    cases' (em q) with hq hnQ
    case inl =>
      have hPtoQ : p → q := by
        intro hp
        exact hq
      contradiction
    case inr =>
      have hPtoQ : p → q := by
        intro hp
        exact absurd hp hnP
      exact absurd hPtoQ hnPtoQ
```

```
import Mathlib.Tactic.Cases
variable (α : Type) (p q : α → Prop)
variable (r : Prop)

open Classical
example : (∀ x, p x) ∨ (∀ x, q x) → ∀ x, p x ∨ q x := by
  intro hxpx_qx
  intro x
  cases' hxpx_qx with hxPx hxQx
  case inl =>
    apply Or.inl
    exact hxPx x
  case inr =>
    apply Or.inr
    exact hxQx x
```

```
example : (¬ ∃ x, p x) ↔ (∀ x, ¬ p x) := by
  apply Iff.intro
  case mp =>
    intro hnxPx
    intro x
    cases' (em (p x)) with hPx hnPx
    case inl =>
      exact False.elim (hnxPx ⟨ x, hPx ⟩  )
    case inr =>
      exact hnPx
  case mpr =>
    intro hxnPx
    intro ⟨ x, hpx ⟩
    exact (hxnPx x) hpx
```

```
example : (∀ x, p x → r) ↔ (∃ x, p x) → r := by
  apply Iff.intro
  case mp =>
    intro hxpx_to_r
    intro ⟨ x, hpx ⟩
    exact show r from (hxpx_to_r x) hpx
  case mpr =>
    intro hxpx_to_r
    intro x
    intro hpx
    exact show r from hxpx_to_r ⟨ x, hpx ⟩
```

```
example : (∀ x, p x) ↔ ¬ (∃ x, ¬ p x) := by
  apply Iff.intro
  case mp =>
    intro hxPx
    intro ⟨ x, hnPx ⟩
    exact absurd (hxPx x) hnPx
  case mpr =>
    intro hnXnPx
    intro x
    cases' (em (p x)) with hPx hnPx
    case inl =>
      exact hPx
    case inr =>
      have hxNpx : ∃ x, ¬ p x := by
        exists x
      exact absurd hxNpx hnXnPx

example (a : α) : (∃ x, r → p x) ↔ (r → ∃ x, p x) := by
  apply Iff.intro
  case mp =>
    intro ⟨ x, hr_to_px ⟩
    intro hr
    exact ⟨ x, show p x from hr_to_px hr ⟩
  case mpr =>
    intro hrToxPx
    cases' (em r) with hr hnR
    case inl =>
      let ⟨ x, hpx ⟩ := hrToxPx hr
      exists x
      intro hr
      exact hpx
    case inr =>
      exists a
      intro hr
      exact absurd hr hnR
```