One-shot Autoregressive Generation of Combinatorial Optimization Solutions based on the Large Language Model Architecture and Learning Algorithms

Bishad Ghimire *, Ausif Mahmood, Khaled Elleithy

Posted Date: 24 February 2025

doi: 10.20944/preprints202502.1797.v1

Keywords: Large Language Models; Transformer; Reinforcement Learning; Direct Preference Optimization; Combinatorial Optimization; Travelling Salesman Problem

 doi:10.20944/preprints202502.1797.v1

Article

One-Shot Autoregressive Generation of Combinatorial Optimization Solutions Based on the Large Language Model Architecture and Learning Algorithms

Bishad Ghimire *, Ausif Mahmood and Khaled Elleithy

Department of Computer Science and Engineering, University of Bridgeport, Bridgeport, CT 06604, USA
* Correspondence: bghimire@my.bridgeport.edu

Abstract: Large Language Models (LLMs) have immensely advanced the field of Artificial Intelligence (AI) with recent models being able to perform chain of thought reasoning and solve complex mathematical problems ranging from theorem proving to ones involving advanced calculus. The success of the LLMs derives from a combination of the Transformer architecture with its attention mechanism, the autoregressive training methodology with masked attention, and the alignment finetuning via reinforcement learning algorithms. In this research, we attempt to explore a possible solution to the fundamental NP-hard problem of combinatorial optimization, in particular the Travelling Salesman Problem (TSP), by following the LLM approach in terms of architecture and the training algorithms. Similar to the LLM design, which is trained in an autoregressive manner to predict the next token, our model is trained to predict the next-node in a TSP graph. After the model is trained on random TSP graphs with known near-optimal solutions, we fine tune the model using Direct Preference Optimization (DPO). The tour generation in a trained model is autoregressive one step generation with no need for iterative refinement. Our results are very promising and indicate that for TSP graphs, for up to 100 nodes, a reasonably small amount of training data yields solutions within a few percent of the optimal. This optimization improves if more data is used to train the model.

Keywords: Large Language Models; Transformer; reinforcement learning; Direct Preference Optimization; combinatorial optimization; Travelling Salesman Problem

1. Introduction

Large Language Models (LLMs) have demonstrated impressive knowledge comprehension skills with the introduction of ChatGPT, and the follow-up models of Gemini, Llama, Claude and recently, DeepSeek, among others. While most of these excel at Natural Language Processing (NLP) tasks such as question answering, text comprehension and summarization, more recent models have focused on improving their reasoning and mathematical understanding capabilities such as DeepSeek-R1 [1]. It has been demonstrated via bench marks that DeepSeek-R1 surpasses average humans in complex mathematical reasoning. Since LLMs are trained on enormous amounts of data, an important question is, are these becoming really intelligent, or simply memorizing all the facts that have been presented to them during training?. Since the LLMs are able to draw conclusions about complex problems using chain of reasoning process, we explore a much more difficult problem in the context of an LLM's learning capability i.e., can the LLM model learn to predict a one-step solution to the combinatorial optimization problem?. We explore this answer with respect to the learning of the solution to the Traveling Salesman problem (TSP) by the existing Transformer-based architecture of an LLM and its learning and alignment by the Reinforcement Learning (RL) algorithm.

TSP optimization problem can be described as: given n nodes and their (x,y) coordinates, determine a Hamiltonian cycle such that the total distance covered in the cycle (or tour) is lowest possible. The Hamiltonian cycle is defined as starting from a node, visiting each other node exactly once, and returning to the starting node at the end of the tour. Since there are $(n-1)!/2$ possible tours, the solution

 doi:10.20944/preprints202502.1797.v1

space to find the optimum greatly increases due to the factorial with respect to n. Because of this, and for its practical applicability, TSP optimization is an important problem in both theoretical Computer Science and industry. It is carried out in route planning for package deliveries, job scheduling, circuit layouts, airline and freight industry, among others. The optimization of TSP has been extensively studied in the last several decades both from mathematical as well as algorithmic perspectives. Since it is a provable NP-hard problem [2], only approximate solutions exist via metaheuristic evolutionary approaches.

In a metaheuristic approach, a population of candidate problem solutions is created initially (often random solutions). These are then evolved by following processes similar to behavior in nature (e.g., evolution in a Genetic Algorithm, or foraging behavior in Ant Colony Optimization). Some of the important metaheuristic approaches to solving the TSP optimization include: Genetic Algorithms [3,4], Simulated Annealing [5,6], Tabu Search [7,8], Particle Swarm Optimization [9,10], Ant Colony Optimization [11,12] and Memetic Algorithms [13,14]. Some approaches have attempted to use a hybrid of metaheuristic algorithms, e.g., [15,16]. The work in [15] used Particle Swarm Optimization, Ant Colony Optimization and three-Opt algorithms to effectively optimize the TSP, while [16] combined Simulated Annealing and Tabu Search for a specialized version of the TSP optimization.

While different metaheuristics are able to provide a near-optimal solution to TSP graphs with up to a few hundred nodes, there are still some disadvantages in their approach. First is the evolutionary nature of all metaheuristic algorithms, which requires numerous iterations to refine the solution. For larger problems, the execution time increases greatly to arrive at a better solution. Because of the inherent iterative nature of the approach, the implementation of a metaheuristic algorithm can only be partially parallelized. Even though significant research has been carried out in parallelizing different metaheuristics, e.g.,[12,17,18], the execution times for larger TSP instances can take many hours or even days of execution times. Additionally, in a metaheuristic algorithm, we need to balance the exploration of solutions in the search space with the exploitation (finding a better solution). Emphasizing the exploitation can lead to getting stuck in local optima, while promoting more exploration can lead to oscillatory behavior where solutions can get better and then worse fairly quickly.

The advancements in Artificial Intelligence via the use of Deep Learning algorithms and architectures offers a promising avenue for solving the TSP optimization in a more effective manner. We review some of important works in this respect in the next section.

2. Related Work

One of the first important work with respect to combinatorial optimization by the use of Deep Learning was presented in [19], termed as "Pointer Network". In this approach, the input data is represented as a sequence of vectors, with each vector representing the x, y coordinates of the node in the input graph. An encoding Recurrent Neural Network (RNN) converts the input sequence to a latent representation that is fed to the generating network. At each step, the generating network produces a vector that modulates a content-based attention mechanism over inputs. The output of the attention mechanism is a softmax distribution with dictionary size equal to the number of nodes in the graph. The training data consisted of pairs of TSP instances (node coordinates) and their corresponding optimal or near-optimal tours. While this approach generates the solution in one-step, it only demonstrated success for a relatively small number of nodes in the graph. For graphs with 40 nodes, an invalid tour was generated 98% of the time, i.e., it contained duplicate nodes.

In a follow up work on the Pointer Network, [20] improved the learning via use of Reinforcement Learning (RL). It employed a policy gradient RL approach (similar to the actor-critic algorithm). The authors used a baseline to reduce variance in the gradient estimates to train the Pointer Network in an unsupervised manner. The results obtained indicated near-optimal results for TSP graphs with up to 100 nodes. The shortcomings of the work in [20] include the generalization to TSP instances that are different than those used for training, and the degradation in performance as the problem

 doi:10.20944/preprints202502.1797.v1

size increases. Further, since the core Pointer Network architecture is based on an RNN, the training stability and scalability in applying to large problems is a significant challenge.

With the introduction of Graph Neural Networks (GNNs), the feasibility of solving TSP was explored in some research papers, e.g., [21,22].The work in [21] attempts to solve a simpler version of the TSP optimization problem known as the decision TSP. Their model is trained to function as an effective message-passing algorithm in which edges (node distance) communicate with nodes for some iterations after which the model decides whether the route with cost < C exists. The work in [22] uses Bidirectional Graph Neural Network (BGNN) for the symmetric TSP. The network learns to produce the next city to visit sequentially by imitation learning. The results on smaller TSP instances are within few percent of the optimal. It is not clear if the approach is applicable to larger TSP instances.

With the immense success of the Transformer in the NLP [26] and Vision [27,28] domains, it is natural that instead of RNN-based approaches, a Transformer may be more effective in learning the TSP landscape. Some of the notable works in this respect include [23–25]. Both of these works use RL approach in learning. The work in [24] uses a policy gradient approach for training its hierarchical RL agents. While the results are not as impressive, the focus is to be able to approximately solve large instance TSP problems. The work in [25] presented a Light Encoder and Heavy Decoder (LEHD) model for the TSP domain. The LEHD model learns to dynamically capture the relationships between nodes of varying sizes. The results presented in the paper achieve better results than existing approaches using the Deep Learning Netwoks.

The revolutionary success of Large Language Models (LLMs) in a range of problems from question answering to theorem proving, naturally begs the question: can their design be applied to solve difficult NP-hard combinatorial problems such as TSP optimization?. Some of the recent work, e.g. [29], attempts to use LLMs in a hybrid setting with respect to TSP. Their approach, termed as hyper-heuristics uses the LLM to guide and control existing optimization algorithms (e.g., local search or Ant Colony Optimization) to find better solutions. Thus LLM is not explicitly used to determine the solution, it used in guiding the evolutionary search for efficiently exploring the heuristic space.

Instead of designing a specialized Deep Learning architecture for solving combinatorial optimization problems, the focus of the work in this paper is to use the same Transformer architecture as used in LLMs, and its training policies to determine its viability in effectively solving such problems in one-shot generation style. Before providing the details of our implementation, we briefly review the Transformer architecture and its training algorithms as followed in the creation of Large Language Models in the next section.

3. Preliminaries

3.1. Transformer Architecture

The seminal work proposing the Transformer architecture [26] has led to development of LLMs. The input data to a Transformer is first converted to a sequence of vectors. In case of NLP, each word is first tokenized into one or more tokens which is a number representing the position of the word in the dictionary of unique words in the corpus. The numbers representing the input text are then converted to embedding vectors (each of size $d \times 1$) by a linear network. These embedding vectors are added with position vectors to maintain their position information with each input.

The fundamental learning mechanism in a Transformer is referred to as the "Attention". This is a simple pair-wise similarity computation in the form of an inner product on the learnt position encoded embeddings of the sequence of n input words. Each layer in the Transformer, then readjusts the position encoded embedding vectors according to the attention information. This process is repeated in many layers. At the end of the last layer, the vector corresponding to the last input is connected to a classifier to predict the next word. The size of output in the classifier is the size of the dictionary of unique words. The index of the highest output (via softmax on the output) decides the next word that would be predicted. We describe the operation of the Transformer formally in the following paragraphs.

 doi:10.20944/preprints202502.1797.v1

If $\mathbf{x}$ contains the n input vectors to a Transformer layer with embedding dimensionality $d \times 1$, then $\mathbf{x} \in \mathbb{R}^{n \times d}$, and Q, K and $V \in \mathbb{R}^{n \times d}$ learnable transformations of the input are computed as:

$$Q = \mathbf{x}W_q, \quad K = \mathbf{x}W_k, \quad V = \mathbf{x}W_v \tag{1}$$

where $W_q, W_k, W_v \in \mathbb{R}^{d \times d}$. The attention is computed as:

$$A = \text{softmax}(QK^T) \tag{2}$$

The attention contains the dot product similarity of each input token with every other token in the input sequence. For an input sequence with n tokens, $Q, K \in \mathbb{R}^{n \times d}$ and attention $A \in \mathbb{R}^{n \times n}$. Each layer in the Transformer divides the attention computation into parallel heads with each head operating on $d_k = \frac{d}{h}$ size vectors, where h is the number of heads in a layer. The output in each head is computed by further multiplying the attention A with V. The output of the i^{th} head is:

$$H_i = \text{softmax}(\frac{Q_i K_i^T}{\sqrt{d_k}})V_i = A_i V_i \tag{3}$$

where $H_i \in \mathbb{R}^{n \times d_k}$. The output in each Transformer layer L_i, is obtained by concatenating the output of all heads and transforming further by a projection matrix W^o:

$$L_i = \text{concatenate}(H_0, H_1, ..., H_{h-1})W^o \tag{4}$$

where $W^o \in \mathbb{R}^{d \times d}$ and $L_i \in \mathbb{R}^{n \times d}$. The output from each layer has the same dimensions as the input. A classification layer is added to the last layer which predicts the next token as:

$$out = \underset{i}{\text{argmax}}[softmax(classifier(L_{k-1}(L_{k-2}(...L_0(embedding(x) + PE(x))))))] \tag{5}$$

The number of outputs in the classifier is equal to the size of the dictionary. The maximum index in the softmax function applied to output decides the token that is being predicted. Skip connections and layer normalization are also used in each layer to stabilize the training of the Transformer.

3.2. Autoregressive Training of Transformers

One of the reasons of the success of the Transformer in the NLP domain is its autoregressive training and generation. In such training, an input sequence consisting of n tokens can be used as $(n-1)$ training pairs, by simply masking the next token to become as target of the prediction. For example, if "how are you?" is used in training in NLP, this will produce three training pairs, with "how [mask] [mask] [mask]" as the first input, with target of "are". Similarly, "how are [mask] mask]" will be used as the second training data with a target of "you". If $x = (x_0, x_1, x_2, ..., x_n)$ is the sequence of input tokens in a training data, then the target tokens $y = (y_0, y_1, ..., y_{n-1}) = (x_1, x_2, ..., x_n)$. Since the predicted output is a single token based on the input, a cross-entropy loss is used in the next token learning of a Transformer as:

$$L(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k \in D} y_{i,k} \log \hat{y}_{i,k} \tag{6}$$

where D is the dictionary size, i.e., the number of unique tokens in the corpus, and $\hat{y}_i$ is the softmax output for the input $(x_0, x_1, ..x_{i-1})$ from the classification layer of the Transformer.

3.3. Improving Trained Model Using Reinforcement Learning

Once an LLM has been extensively trained using cross-entropy in an autoregressive manner, its model is further improved by aligning it with respect to human preferences. Human feedback is used to rank a pair of responses generated from the same input (i.e., prompt), then such data is used to further train the LLM using a Reinforcement Learning (RL) algorithm. This approach is referred to as

 doi:10.20944/preprints202502.1797.v1

Reinforcement Learning with Human Feedback (RLHF). One of the popular algorithms for RLHF is the Proximal Policy Optimization (PPO) [30].

PPO, overcomes some of drawbacks of previous policy gradient based methods i.e., difficulty in learning due to sensitivity with respect to step size, and sample efficiency. PPO uses a simple concept of updates at each step such that the deviation from the previous learnt policy is relatively small. It uses a ratio between the current and old policy, and clips this ratio to a range $[1-\epsilon, 1+\epsilon]$. This causes the training to be more stable.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \tag{7}$$

$$L_{clip}(\theta) = \mathbb{E}_t[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)] \tag{8}$$

where A_t is the advantage in RL that is defined as the quality of a state and an action being taken with respect to the value of that state. Due to its simplicity, stability and sample efficiency, PPO was used for RLHF in InstructGPT (precursor to ChatGPT) [31]. PPO remains a popular choice for RLHF in LLMs. Even though some newer algorithms such as Direct Preference Optimization (DPO) and Group Relative Proximal Optimization (GRPO) [32] have been attempted in recent LLMs, e.g., Llama-3 used DPO [33], and DeepSeek V3 is trained using GRPO [34].

4. Proposed Method

4.1. TSP Graph Representation

Since LLMs that are based on the Transformer architecture are trained initially in a next token prediction mode, we can adapt the Transformer to learn next-node prediction in TSP. The challenge lies in how to effectively map the given graph structure and the solution in terms of Hamiltonian cycle to a linear sequence of input tokens. If we can effectively represent this, then the learning of the TSP solution can be carried out in an autoregressive style similar to that in an LLM Transformer. In our design, each node in the TSP graph is represented by an $e \times 1$ vector (as given in Equation (9)) where $e = numberOfnodes + 3$. The three additional numbers represent the node number and its x, y coordinates in $\mathbb{R}^2$.

$$\begin{bmatrix} \text{node num} \\ x \\ y \\ \text{dist to node 0} \\ \text{dist to node 1} \\ \vdots \\ \text{dist to node n} \end{bmatrix} \tag{9}$$

Using the above representation, each node carries the complete relationship in terms of Euclidean distances to all other nodes. Thus, the entire TSP graph to be optimized in terms of its Hamiltonian cycle can be represented as a sequence of vectors as shown below:

$$\begin{bmatrix} \text{node 0} \\ x \\ y \\ \text{dist to 0} \\ \text{dist to 1} \\ \vdots \\ \text{dist to n} \end{bmatrix} \begin{bmatrix} \text{node 1} \\ x \\ y \\ \text{dist to 0} \\ \text{dist to 1} \\ \vdots \\ \text{dist to n} \end{bmatrix} \quad \dots \quad \begin{bmatrix} \text{node n} \\ x \\ y \\ \text{dist to 0} \\ \text{dist to 1} \\ \vdots \\ \text{dist to n} \end{bmatrix} \tag{10}$$

4.2. Generation of Training Data for TSP

In supervised learning of an LLM architecture, the training data is easily available in the form of any text documents from any text source (e.g., Wikipedia articles, books, news etc..). This text data

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 24 February 2025 doi:10.20944/preprints202502.1797.v1

during the training process is randomly split into sequence length of n segments (as defined by the Transformer architecture), and is used to generate $n-1$ training pairs by masking the future tokens. For TSP optimization problem, we need to know the optimal Hamiltonian cycle for a given graph. For this purpose, we use the efficient parallel Ant Colony Optimization algorithm (ACO) for solving TSP optimization problems as proposed in [35]. We create many different random graphs, and then feed those to the above-mentioned parallel ACO algorithm to determine the near-optimal Hamiltonian cycle. To generate the training data, the sequence of vectors describing the TSP graph (Equation (10)) are appended with the node vectors belonging to the near-optimal Hamiltonian cycle as shown in Equation (11).

$$\underbrace{\begin{bmatrix}\text{node 0}\\ x\\ y\\ \text{dist to 0}\\ \text{dist to 1}\\ \vdots\\ \text{dist to n}\end{bmatrix}\begin{bmatrix}\text{node 1}\\ x\\ y\\ \text{dist to 0}\\ \text{dist to 1}\\ \vdots\\ \text{dist to n}\end{bmatrix}\cdots\begin{bmatrix}\text{node n}\\ x\\ y\\ \text{dist to 0}\\ \text{dist to 1}\\ \vdots\\ \text{dist to n}\end{bmatrix}}_{\text{TSP Input Graph}}\underbrace{\begin{bmatrix}\text{node 0}\\ x\\ y\\ \text{dist to 0}\\ \text{dist to 1}\\ \vdots\\ \text{dist to n}\end{bmatrix}\begin{bmatrix}\text{node a}\\ x\\ y\\ \text{dist to 0}\\ \text{dist to 1}\\ \vdots\\ \text{dist to n}\end{bmatrix}\cdots\begin{bmatrix}\text{node z}\\ x\\ y\\ \text{dist to 0}\\ \text{dist to 1}\\ \vdots\\ \text{dist to n}\end{bmatrix}\begin{bmatrix}\text{node 0}\\ x\\ y\\ \text{dist to 0}\\ \text{dist to 1}\\ \vdots\\ \text{dist to n}\end{bmatrix}}_{\text{Near-optimal Hamiltonian Cycle}} \tag{11}$$

Note that in Equation (10), the vectors in the "near-optimal Hamiltonian cycle" part are copies of the corresponding vectors from the TSP input graph. The parallel ACO algorithm for TSP [35] is used to generate the Hamiltonian cycle, which then is used to create the training data by copying the vectors from the TSP graph specification as described by Equation (11). In the Hamiltonian cycle in Equation (11), node 0 is the start node and node 0 is also the last node as cycle has to be completed in TSP coming back to the start node. Thus, the number of input vectors in the training data item for TSP is $2n+1$ if n is the number of nodes in the TSP graph.

4.3. Transformer Architecture

The Transformer architecture that we use for TSP optimization is shown in Figure 1. It is a Decoder only architecture as followed in many current LLM designs. There are two changes in our architecture with respect to a standard LLM Transformer. First is the difference in the Embedding layer. In our design, the input is a sequence of node vectors (as opposed to tokens) and a potential near-optimal solution vectors as indicated in Equation (11). Thus the embedding layer used is a simple linear layer with $2n+1$ inputs and d outputs, where d is the embedding dimensionality of the Transformer.

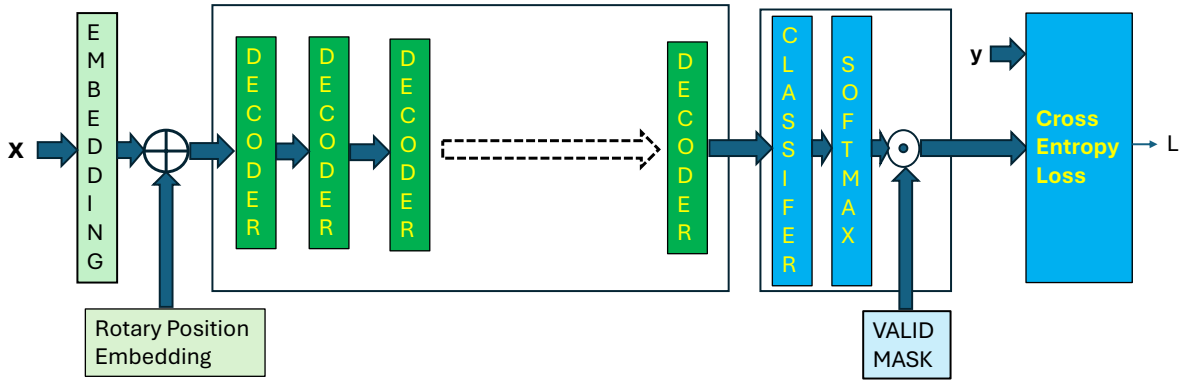


Figure 1. Transformer Architecture for TSP Graph Optimization

The second main change in our design is a Hadamard product of the "valid mask" with the output of the final softmax layer. In autoregressive generation, since the transformer predicts the next-node to visit (one at a time), the already visited nodes should be removed from the set of remaining candidate

 doi:10.20944/preprints202502.1797.v1

nodes. We accommodate this by adding a mask entry at the position of the last generated node number. Figure 2 shows the Decoder architecture. This is similar to an LLM Transformer Decoder.

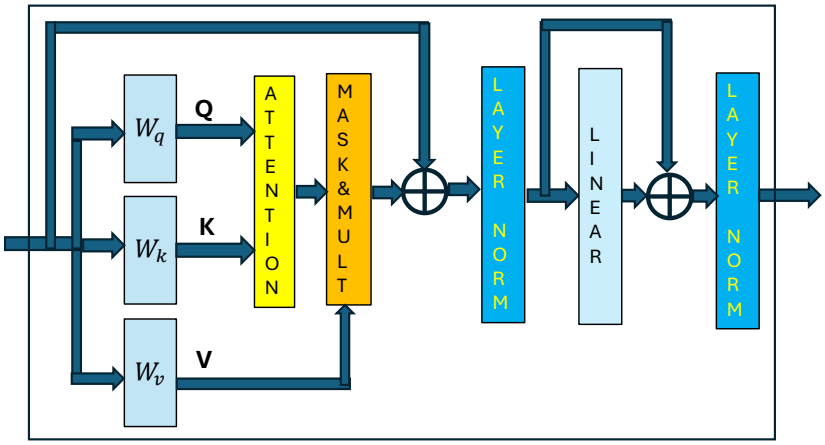


Figure 2. Architecture of Decoder Block in TSP Transformer

4.4. Training and Generation for TSP Optimization

We follow a two-step approach to training of the Transformer architecture for TSP optimization, as is followed in LLMs. The first step trains the architecture in a next-node prediction style by masking the future nodes. The training input is described by Equation (11):

$$n_0, n_1, ...n_n, n_0, \mathbf{n_a}, \mathbf{n_b}, ..., \mathbf{n_0} \tag{12}$$

The bold part in Equation (12) is the target near-optimal sequence. The bold part will be masked out one at a time during the next-node prediction training process. As an example, for the input of $n_0, n_1, ...n_n, n_0$, the target node is $\mathbf{n_a}$. Similarly, for the input node sequence of $n_0, n_1, ...n_n, n_0, n_a$, the predicted node is $\mathbf{n_b}$, and in the same manner onward. The implementation uses triangular masked attention in the target sequence part to speed up training. From a randomly generated dataset of TSP graphs and their near-optimal solutions, the Transformer architecture is trained using the cross-entropy loss, with "valid mask" to guarantee the Hamiltonian cycle it learns is valid. The predicted node $\hat{n_i}$ is determined as given in Equation (13).

$$\hat{n_i} = \underset{i}{\text{argmax}}[softmax(classifier(L_{k-1}(L_{k-2}(...L_0(embedding(x) + PE(x)))) \times Valid_{mask})] \tag{13}$$

The valid mask zeros out the output from the final softmax layer for the position where the last node number was predicted. This eliminates the possibility of a duplicate node in the predicted tour. The cross-entropy loss used in training in the first step is described in Equation (14).

$$L(\theta) = -\frac{1}{n}\sum_{i=n}^{2n+1}\sum_{k\in N} n_{i,k}\log \hat{n}_{i,k} \tag{14}$$

where n is the number of nodes in the graph, N is the set of nodes, and the subscript i indicates the position of the node in the output tour.

The second step of the training uses Reinforcement Learning (RL) to further improve the prediction capabilities for the TSP tour. We implement both the Proximal Policy Optimization (PPO), and the Direct Preference Optimization (DPO) algorithms that are popular in LLM training phase (RLHF). The DPO algorithm is more appropriate for TSP node prediction, as the preference between two generated tours can be exactly computed according to their tour cost without any need for human feedback. The DPO loss function given below in our implementation is from [36].

 doi:10.20944/preprints202502.1797.v1

$$\mathcal{L}_{DPO}(\pi_\theta; \pi_{ref}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{ref}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{ref}(y_l|x)}\right)\right] \tag{15}$$

DPO uses the model trained from the first step as the reference model (π_{ref}), and then tries to improve this via the preference data. The model that is being improved is referred to as (π_θ). The winning response i.e., $(y_w|x)$ is the preferred response, while $(y_l|x)$ is the non-preferred response. In TSP, preference data can be the near-optimal tour from training data, while the actual model output can be used as the non-preferred data in the preference pair as it is higher cost. The DPO loss function as described above helps the model improve in its learning by favoring preferred responses while also keeping the model's policy within reasonable bounds set by the reference model (i.e., model that was trained in first step).

Once the Transformer model has been trained in a two-step process as described above, it can be used to generate the near-optimal tour by feeding it the sequence of nodes in the format described by Equation (11). The target sequence component in this sequence is changed to all nodes being the zero node. Starting from the sequence of given nodes in the graph and node 0, the model predicts the first node in the tour that follows node 0. This predicted node is then appended to the input sequence after node 0, and the next-node is predicted. Valid masking is used so that a previously predicted node cannot be predicted. This autoregressive process is repeated until the complete tour is generated returning back to node 0.

5. Results

We generated TSP training data for many randomly generated graphs between 20 and 100 nodes. The random graphs are then optimized in a near-optimal manner using the parallel ACO TSP solver presented in [35]. The original graph and its near-optimal solution are then combined to generate the training data for the TSP Transformer architecture as described in the previous section. We compare the results on a test set of 10 randomly generated graphs for the following cases:

1. Using only step 1 of training where the model is trained for next-node prediction based on the cross-entropy loss only.
2. Using only step 2 of training where the model is trained for tour prediction using a DPO loss only.
3. Using both step 1 and step 2 where the model is trained using both cross-entropy loss and DPO loss.

The "% Optimal" numbers in Table 1 represent the tour cost within percentage of the known-optimal value. The test data is a set of randomly generated graphs that were not used in training the model. From Table 1, it can be seen that while both loss functions individually are able to make the model learn effectively and do a good prediction of the near-optimal tour, using both of these (as is typically followed in an LLM training) yields better results.

We also compare the percentage optimal results on the TSPLIB benchmarks for some of the graphs up to 100 nodes. Table 2 shows the results in terms of the % of the known-optimal tour cost. As it can be seen, more training data helps the model to generalize effectively and produce better results.

 doi:10.20944/preprints202502.1797.v1

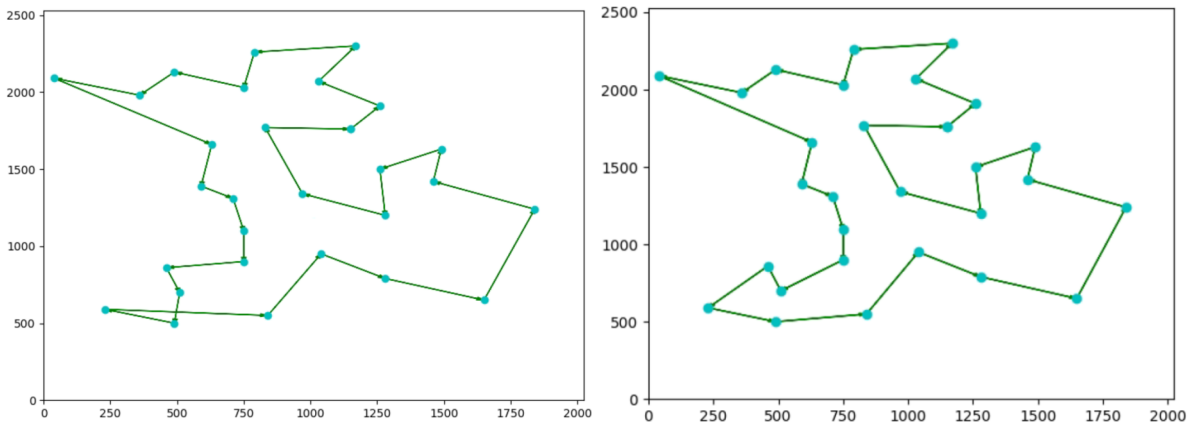
Table 1. Comparison of Average Test Accuracy Using Cross-Entropy and DPO for 10000 and 50000 Training Data for different size TSP Graphs.

Nodes	% Optimal 10000 T. Data Cross-Entropy Only	% Optimal 10000 Data DPO Only	% Optimal 10000 T. Data both Cross-Entropy and DPO	% Optimal 50000 T. Data both Cross-Entropy and DPO
20	3.4	3.5	3.3	2.9
40	8.8	8.9	8.4	6.3
60	13.6	13.9	13.4	11.8
80	17.4	17.9	17.2	15.9
100	21.8	21.3	21.6	19.6

Table 2. Comparison of Average Test Accuracy Using Cross-Entropy and DPO for 10000 and 50000 Training Data for different TSPLIB benchmarks.

TSPLIB benchmark	Number of Nodes	% Optimal 10000 T. Data Cross-Entropy	% Optimal 10000 T. Data DPO	% Optimal 10000 T. Data Cross-Entropy and DPO	% Optimal 50000 T. Data Cross-Entropy and DPO
Bays29	29	3.5	3.7	3.3	2.5
Berlin52	52	4.5	4.8	4.2	3.7
Eil76	76	7.8	8.1	7.6	5.6
KroA100	100	15.6	15.9	15.4	9.8
KroB100	100	17.1	17.7	17.7	10.1
KroC100	100	16.3	16.4	16.9	10.9
KroD100	100	19.8	19.9	18.8	12.9
KroE100	100	18.8	18.2	18.7	11.9

Figures 3–5 show the actual tour generated by the Trained TSP Transformer using both training steps of cross-entropy and DPO with 50000 training data. For smaller graphs, the results are closer to the optimal than compared to relatively larger node graphs. However, as more training data is used, the tour generation gets closer to the optimal cost.

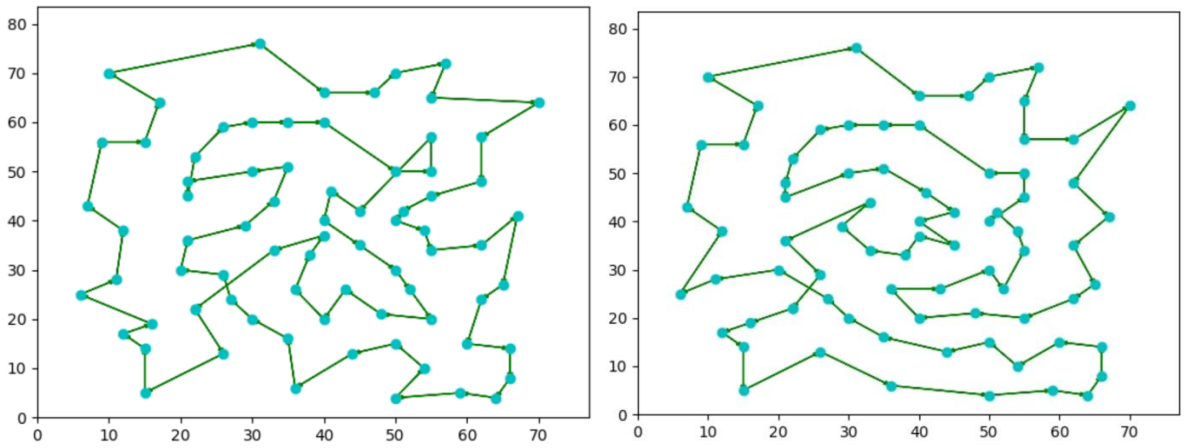


(**a**) After Cross-Entropy Loss Training - Step 1

(**b**) After DPO Loss Training - Step 2

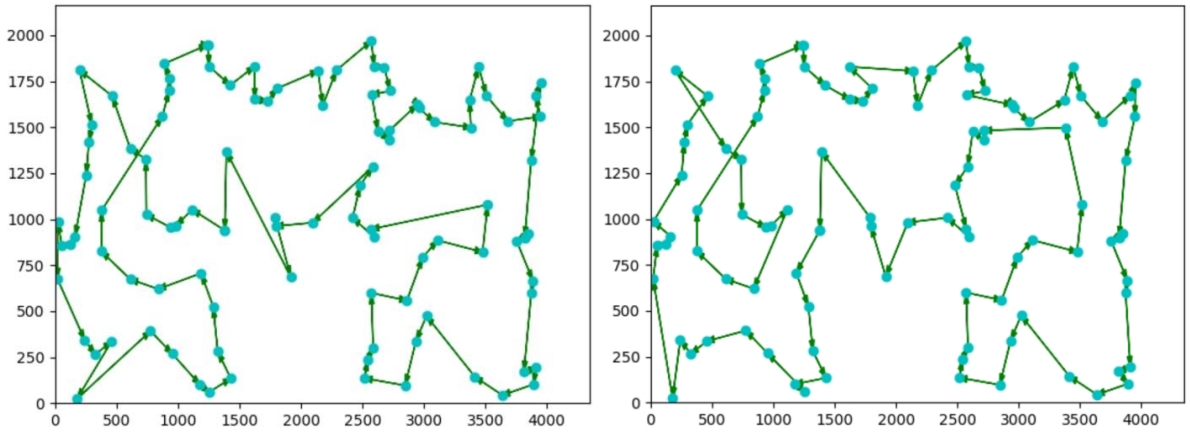
Figure 3. Optimized tour generated by the TSP Transformer for Bays29 TSPLIB benchmark (Tour cost = 9336 after step 1, 9252 after step 2, Optimal = 9076)

 doi:10.20944/preprints202502.1797.v1



(**a**) After Cross-Entropy Loss Training - Step 1 (**b**) After DPO Loss Training - Step 2

Figure 4. Optimized tour generated by the TSP Transformer for Eil76 TSPLIB benchmark (Tour cost = 590 after step 1, 584 after step 2, Optimal = 553)



(**a**) After Cross-Entropy Loss Training - Step 1 (**b**) After DPO Loss Training - Step 2

Figure 5. Optimized tour generated by the TSP Transformer for KroA100 TSPLIB benchmark (Tour cost = 23418 after step 1, 23292 after step 2, Optimal = 21282)

We also attempted to use the PPO algorithm in step 2 of the training. This did not produce an improvement over step 1 of the training process. However, the DPO algorithm is more effective and we used it in our results.

6. Discussion

From the results in the previous section, it is clear that the Transformer architecture is able to learn a near-optimal one-shot solution to the TSP optimization problem. The quality of the results improves as more training data is employed as indicated in Tables 1 and 2. The first step in training (via the cross-entropy loss) learns the conditional distribution of nodes based on the given graph and the partial tour. If $T(x)$ is the target data distribution and $Y(x)$ is model's predicted distribution, we can mathematically see the learning via the cross-entropy loss as:

$$\mathrm{H}(T, Y) = -\sum_{x} T(x) \log Y(x) \tag{16}$$

$$\mathrm{H}(T, Y) = -\sum_{x} T(x)[\log T(x) + \log Y(x) - \log T(x)] \tag{17}$$

$$\mathrm{H}(T, Y) = -\sum_{x} T(x)\left[\log T(x) + \log \frac{Y(x)}{T(x)}\right] \tag{18}$$

 doi:10.20944/preprints202502.1797.v1

$$\mathrm{H}(T,Y) = -\sum_{x} T(x) \log T(x) - \sum_{x} T(x) \log \frac{Y(x)}{T(x)} \tag{19}$$

$$\mathrm{H}(T,Y) = H(T) + D_{KL}(T||Y) \tag{20}$$

Since the target data distribution $T(x)$ is fixed, as can be seen from Equation (20), the model learns to reduce the KL divergence between the learnt distribution and the target data distribution as it minimizes the cross-entropy loss. This can be empirically observed from the results in previous section, i.e., as more data is used, the model's generalization improves. If we train the TSP Transformer on 100,000 training data for 76 node randomly generated graphs, and then test it on the Eil76 benchmark, the tour cost improves to 578 as opposed to 584 which was obtained with 50,000 training data as was shown in Figure 4. We show the improved generated tour for Eil76 in Figure 6 and also show how, for the majority of the tour, the learnt model is following the optimal path which it is learning via random graphs and their optimized solutions during training.

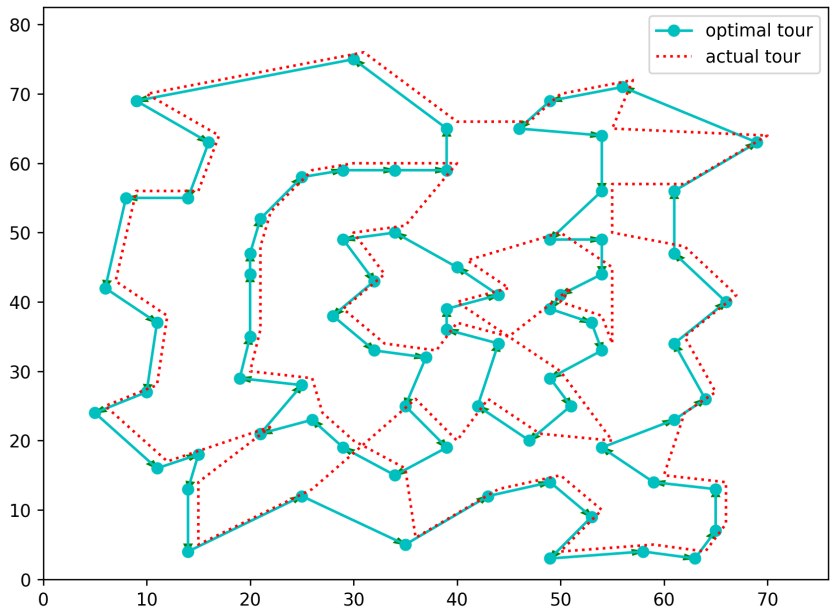


Figure 6. Actual Generated Tour vs Optimal Tour for Eil76 Benchmark for TSP Transformer Trained with 100,000 data. Actual Tour Cost = 578, Optimal Tour Cost = 553.

The second step of the training focuses on the entire tour. Using the Reinforcement Learning DPO algorithm, the quality of the generated tour (the entire tour) is optimized. Thus, the two-step training process in our implementation further improves the quality of optimization.

7. Conclusions

Our objective in this research was to explore the effective learning of LLM architecture and algorithms in the combinatorial optimization domain. We wanted to answer the fundamental question: can the LLM designs learn to solve the NP-hard combinatorial optimization problems and provide a one-shot solution to the problem? To answer this, we adapt the ubiquitous LLM Transformer to the TSP optimization problem. The changes required in the LLM Transformer architecture are only at the input data level to the Transformer and an extra masking at the output to guarantee a valid Hamiltonian cycle in its generation.

The input data uses a sequence of vectors with each vector representing the node in the TSP graph and its relationship to other nodes. The rest of the Transformer architecture is kept the same as a canonical LLM Transformer including its training regime in terms of next token generation via a cross-entropy loss, followed by Direct Preference Optimization as followed in most LLMs.

Our results on both randomly generated graphs and TSPLIB benchmarks are very encouraging. With relatively small training data (<= 50000, where as LLMs are trained on billions of data), we are

 doi:10.20944/preprints202502.1797.v1

able to demonstrate the effective one-shot generation of the TSP optimization. The tour generated by the trained architecture has a cost < few % the optimal cost for TSP graphs with up to 100 nodes. Note that this is a one-shot solution without any iterative refinement.

Our future work involves learning the effective training of the architecture in terms of data quality, i.e., using more effective training data by examining in which cases it can be improved in its generalization. In addition, we explore other constrained combinatorial optimization problems.

Author Contributions: Conceptualization, B.G., A.M. and K.E.; methodology, software, validation, B.G. and A.M.; analysis, resources, data curation, visualization, B.G.; writing—original draft preparation, B.G. and A.M.; writing—review and editing, B.G., A.M. and K.E. All authors have read and agreed to the published version of the manuscript.

Funding: The authors received no financial support for this research.

Data Availability Statement: Source code of our work is publicly accessible via our GitHub repository: https://github.com/bishadghimire/TSPLLM.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Zuo, Y.; Qu, S.; Li, Y.; Chen, Z.; Zhu, X.; Hua, E.; Zhang, K.; Ding, N.; Zhou, B. MedXpertQA: Benchmarking Expert-Level Medical Reasoning and Understanding. *arXiv preprint arXiv:2501.18362* **2025**.
2. Karp, Richard M.: Reducibility among Combinatorial Problems *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations*, 1972, Springer US, pp. 85–103.
3. Larranaga, P.; Kuijpers, C. M. H.; Murga, R. H.; Inza, I.; Dizdarevic, S. Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artif. Intell. Rev.* **1999**, *13*, 129–170.
4. Razali, N. M.; Geraghty, J. Genetic Algorithm Performance with Different Selection Strategies in Solving TSP. In *Proceedings of the World Congress on Engineering*; International Association of Engineers Hong Kong, China, 2011; Vol. 2, No. 1, pp. 1–6.
5. Ezugwu, A. E.-S.; Adewumi, A. O.; Frîncu, M. E. Simulated Annealing Based Symbiotic Organisms Search Optimization Algorithm for Traveling Salesman Problem. *Expert Syst. Appl.* **2017**, *77*, 189–210.
6. Meer, K. Simulated Annealing versus Metropolis for a TSP Instance. *Inf. Process. Lett.* **2007**, *104*(6), 216–219.
7. Fiechter, C.-N. A Parallel Tabu Search Algorithm for Large Traveling Salesman Problems. *Discret. Appl. Math.* **1994**, *51*(3), 243–267.
8. Brandão, J. A Tabu Search Algorithm for the Open Vehicle Routing Problem. *Eur. J. Oper. Res.* **2004**, *157*(3), 552–564.
9. Wang, K.-P.; Huang, L.; Zhou, C.-G.; Pang, W. Particle Swarm Optimization for Traveling Salesman Problem. In *Proceedings of the 2003 International Conference on Machine Learning and Cybernetics*; IEEE, 2003; Vol. 3, pp. 1583–1585.
10. Lu, C.; Wang, Q. X. Particle Swarm Optimization-Based Algorithms for TSP and Generalized TSP. *Inf. Process. Lett.* **2007**, *103*(5), 169–176.
11. Yang, J.; Shi, X.; Marchese, M.; Liang, Y. An Ant Colony Optimization Method for Generalized TSP Problem. *Prog. Nat. Sci.* **2008**, *18*(11), 1417–1422.
12. Ghimire, B.; Cohen, D.; Mahmood, A. Parallel Cooperating Ant Colonies with Improved Periodic Exchange Strategies. *Proc. High Perform. Comput. Symp.* **2014**, 1–6.
13. Merz, P.; Freisleben, B. Memetic Algorithms for the Traveling Salesman Problem. *Complex Syst.* **2001**, *13*(4), 297–346.
14. Gutin, G.; Karapetyan, D. A Memetic Algorithm for the Generalized Traveling Salesman Problem. *Nat. Comput.* **2010**, *9*, 47–60.
15. Mahi, M.; Baykan, Ö. K.; Kodaz, H. A New Hybrid Method Based on Particle Swarm Optimization, Ant Colony Optimization and 3-opt Algorithms for Traveling Salesman Problem. *Appl. Soft Comput.* **2015**, *30*, 484–490.
16. Küçükoğlu, İ.; Dewil, R.; Cattrysse, D. Hybrid Simulated Annealing and Tabu Search Method for the Electric Travelling Salesman Problem with Time Windows and Mixed Charging Rates. *Expert Syst. Appl.* **2019**, *134*, 279–303.

17. Stützle, T. Parallelization Strategies for Ant Colony Optimization. In *International Conference on Parallel Problem Solving from Nature*; Springer, 1998; pp. 722–731.
18. Cantu-Paz, E.; Goldberg, D. E. Efficient Parallel Genetic Algorithms: Theory and Practice. *Comput. Methods Appl. Mech. Eng.* **2000**, *186*(2–4), 221–238.
19. Vinyals, O.; Fortunato, M.; Jaitly, N. Pointer Networks. *Adv. Neural Inf. Process. Syst.* **2015**, *28*.
20. Bello, I.; Pham, H.; Le, Q. V.; Norouzi, M.; Bengio, S. Neural Combinatorial Optimization with Reinforcement Learning. *arXiv preprint arXiv:1611.09940* **2016**.
21. Prates, M.; Avelar, P. H. C.; Lemos, H.; Lamb, L. C.; Vardi, M. Y. Learning to Solve NP-Complete Problems: A Graph Neural Network for Decision TSP. In *Proceedings of the AAAI Conference on Artificial Intelligence*; 2019; Vol. 33, No. 1, pp. 4731–4738.
22. Hu, Y.; Zhang, Z.; Yao, Y.; Huyan, X.; Zhou, X.; Lee, W. S. A Bidirectional Graph Neural Network for Traveling Salesman Problems on Arbitrary Symmetric Graphs. *Eng. Appl. Artif. Intell.* **2021**, *97*, 104061.
23. Bresson, X.; Laurent, T. The Transformer Network for the Traveling Salesman Problem. *arXiv preprint arXiv:2103.03012* **2021**.
24. Pan, X.; Jin, Y.; Ding, Y.; Feng, M.; Zhao, L.; Song, L.; Bian, J. H-TSP: Hierarchically Solving the Large-Scale Traveling Salesman Problem. In *Proceedings of the AAAI Conference on Artificial Intelligence*; 2023; Vol. 37, No. 8, pp. 9345–9353.
25. Luo, F.; Lin, X.; Liu, F.; Zhang, Q.; Wang, Z. Neural Combinatorial Optimization with Heavy Decoder: Toward Large Scale Generalization. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 8845–8864.
26. Vaswani, A. Attention Is All You Need. *Adv. Neural Inf. Process. Syst.* **2017**.
27. Dosovitskiy, A.; Beyer, L.; Kolesnikov, A.; Weissenborn, D.; Zhai, X.; Unterthiner, T.; Dehghani, M.; Minderer, M.; Heigold, G.; Gelly, S.; et al. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. *arXiv preprint arXiv:2010.11929* **2020**.
28. Mahmood, K.; Mahmood, R.; Van Dijk, M. On the Robustness of Vision Transformers to Adversarial Examples. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*; 2021; pp. 7838–7847.
29. Ye, H.; Wang, J.; Cao, Z.; Berto, F.; Hua, C.; Kim, H.; Park, J.; Song, G. Large Language Models as Hyper-Heuristics for Combinatorial Optimization. *arXiv preprint arXiv:2402.01145* **2024**.
30. Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. *Proximal policy optimization algorithms*; arXiv:1707.06347, 2017.
31. Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; et al. *Training language models to follow instructions with human feedback*; Adv. Neural Inf. Process. Syst. **2022**, *35*, 27730–27744.
32. Shao, Z.; Wang, P.; Zhu, Q.; Xu, R.; Song, J.; Bi, X.; Zhang, H.; Zhang, M.; Li, Y. K.; Wu, Y.; et al. *Deepseekmath: Pushing the limits of mathematical reasoning in open language models*; arXiv:2402.03300, 2024.
33. Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Yang, A.; Fan, A.; et al. *The llama 3 herd of models*; arXiv:2407.21783, 2024.
34. Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; et al. *Deepseek-v3 technical report*; arXiv:2412.19437, 2024.
35. Ghimire, B.; Cohen, D.; Mahmood, A. *Parallel cooperating ant colonies with improved periodic exchange strategies*; Proceedings of the High Performance Computing Symposium, 2014; pp 1–6.
36. Rafailov, R.; Sharma, A.; Mitchell, E.; Manning, C. D.; Ermon, S.; Finn, C. *Direct preference optimization: Your language model is secretly a reward model*; Adv. Neural Inf. Process. Syst. **2024**, *36*.