Tensorized Ant Colony Optimization for GPU Acceleration

Anonymous Author(s)

ABSTRACT

Ant Colony Optimization (ACO) is renowned for its effectiveness in solving Traveling Salesman Problems, yet it faces substantial computational challenges in CPU-based environments, particularly with large-scale instances. In response, we introduce a Tensorized Ant Colony Optimization (TensorACO) to harness the advancements of GPU acceleration. As the core, TensorACO fully transforms ant system and ant path into tensor forms, a process we refer to as *tensorization*. For the tensorization of ant system, we propose a preprocessing method to reduce the computational overhead by calculating the probability transition matrix. In the tensorization of ant path, we propose an index mapping method to accelerate the update of pheromone matrix by replacing the mechanism of sequential path update with parallel matrix operations. Additionally, we introduce an Adaptive Independent Roulette (AdaIR) method to overcome the challenges of parallelizing ACO's selection mechanism on GPUs. Comprehensive experiments demonstrate the superior system performance of TensorACO achieving up to 1921× speedup than standard ACO. Moreover, the AdaIR method further improves TensorACO's convergence speed by 80% and solution quality by 2%. Codes will be made publicly available.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; • **Theory of computation** → **Vector / streaming algorithms**.

KEYWORDS

Tensorization, Ant Colony Optimization, GPU, Adaptive Independent Roulette, Traveling Salesman Problem

ACM Reference Format:

Anonymous Author(s). 2024. Tensorized Ant Colony Optimization for GPU Acceleration. In *Proceedings of The Genetic and Evolutionary Computation Conference 2024 (GECCO '24).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Traveling Salesman Problem (TSP), a central challenge in combinatorial optimization, is characterized by the theoretical complexity and practical relevance in routing and logistics [13, 20]. As a classic problem, large-scale TSP instances require computational resources that increase exponentially with the size of the problem [26]. This poses a significant challenge for conventional computing systems, which are often unable to handle the computational load [2].

GECCO '24, July 14–18, 2024, Melbourne, Australia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Traditional approaches to solving TSP include exact algorithms and approximate algorithms [15]. Exact algorithms are guaranteed to find the optimal solution but are characterized by being prohibitively computationally expensive and impractical for large-scale problems [2]. The limitation has catalyzed the exploration of approximate approaches. Metaheuristic algorithms excel in efficiently navigating complex search spaces to find near-optimal solutions, offering a versatile and robust approach [5].

Ant Colony Optimization (ACO) [8], a classic metaheuristic algorithm inspired by the foraging behavior of ants, has emerged as a particularly effective tool for TSP. ACO balances exploration and exploitation through a pheromone-based communication mechanism, guiding the search toward promising regions of the solution space. Despite its relative efficiency, ACO still entails significant computational overhead. The bottleneck becomes apparent with the rapid advancements in computational scale. The evolution of GPU technology offers an unprecedented opportunity to mitigate the computational challenges [21]. Leveraging the parallel processing capabilities of GPUs enables a substantial reduction in computational load and enhances the efficiency of metaheuristic algorithms.

The performance of GPU within a tensorized computational framework is recognized in processing massively parallel computations [10]. The advancement of GPU architectures and the emergence of tensorized computing provide a unique opportunity to transform traditional CPU-based ACO models. Improvements in processing speed, scalability, and overall efficiency are potential outcomes of this transformation. A paradigm shift towards a GPU-optimized framework is envisaged to enhance the performance of ACO significantly, especially in addressing complex large-scale optimization challenges.

The significant challenges of GPU-accelerated ACO are the intrinsic complexity and sequential nature of traditional computational logic [9]. The adaptation of the algorithm for parallel processing on GPU necessitates a sophisticated restructuring to ensure efficiency. Furthermore, the complex programming and deployment of GPU environments is a significant barrier to entry, which demands a comprehensive understanding of both algorithmic design and advanced computational techniques.

The JAX framework [3, 11] significantly enhances high-performance numerical computing with hardware acceleration. Building upon JAX, pioneering platforms like EvoJAX [25], evosax [14], and EvoX [12] have emerged, providing efficient tools for evolutionary computation for advanced hardware acceleration. However, a complete implementation of scalable ACO remains absent. To bridge this gap, we propose the Tensorized Ant Colony Optimization (TensorACO).

The main contributions of this paper are summarized below.

- We propose an ant system tensorization method. It integrates heuristic method values and the pheromone matrices to compute the probability transition matrix, enabling efficient preprocessing while achieving acceleration using function mapping.

- We propose an ant path tensorization method. It parallelizes the serial processing of ant path solutions in the pheromone matrix computation and reduces redundant path cost calculations through an index mapping method.
- We propose an Adaptive Independent Roulette (AdaIR) method. It improves algorithm performance while retaining the benefits of parallelization by incorporating a *learning rate* hyperparameter to dynamically adjust the tendency of ants to select cities with higher probabilities.

2 RELATED WORK

Dorigo *et al.* [8] first proposed ACO to solve the TSP. Stützle [24] introduced the Max-Min Ant System (MMAS) to improve the effectiveness of ACO. Bai *et al.* [1] first pioneered GPU acceleration for MMAS, achieving a significant 2.3× speedup, thereby marking a groundbreaking advancement in GPU computing. Pedemonte *et al.* [18] summarized the parallel ACO, detailing the various metrics, classifications, and developments. Menezes *et al.* [17] explored the trade-offs of different parallelization strategies for GPU-Based solving the various TSPs. Although these studies have achieved significant progress in GPU acceleration, they pose challenges due to insufficient acceleration efficiency and high computational complexity. In this paper, we propose a tensorized ACO framework based on the JAX platform to address these challenges.

The traditional Roulette Wheel process in ACO cannot be parallelized, Cecilia *et al.* [4] proposed the Independent Roulette (IR) method, a data-parallel approach implemented using CUDA that accelerated the path-building process in GPUs. Dawson and Stewart [7] proposed the Double-spin Roulette, achieving approximately 82× acceleration. Zhou *et al.* [28] introduced the Tiling Roulette (TR) method that reduced the generation of random numbers when applied to GPUs. Also, Lloyd and Amos [16] performed a mathematical analysis of the IR method, demonstrating the feasibility of the method. The literature above demonstrates the effectiveness of the IR method in accelerating the ACO algorithm. However, altering the probability of ants selecting cities in IR carries the risk of reducing solution quality and convergence speed. To address this challenge, We propose an AdaIR method to accelerate the convergence of the ACO algorithm.

In the field of expanding the scale of cities in TSP, almost all approaches have used parallel strategies to accelerate algorithms. Chitty *et al.* [6] proposed the PartialACO, which divides large-scale TSPs into smaller segments, facilitating computations for problem sizes up to 200,000 cities. Yelmewad *et al.* [27] introduced a parallel scheme that successfully executed on instances with 33,810 cities, achieving 60× speedup. Skinderowicz [22] proposed a method capable of finding solutions for TSPs with 18,152 nodes, with less than 1% deviation from optimal solutions, achieving the state-of-the-art quality of ACO solutions. Further building on this, Skinderowicz [23] introduced the Focused ACO (FACO) and extended its application to TSPs with up to 200,000 cities. Although the variants of ACO have been proven to be effective methods in solving large-scale TSPs, the computational consumption remains substantial. In this paper, the large-scale problems are solved by TensorACO, which achieves nearly 2000× acceleration.

3 METHOD

This section provides a detailed introduction to the proposed TensorACO. Overall, the main content of TensorACO can be divided into the following parts: the overall framework uses `state` to ensure function mapping and deterministic operations; The ant system and ant path have been tensorized to enable the algorithm to be applied on GPUs for acceleration; The AdaIR method was proposed to optimize traditional Roulette Wheel process, enables GPU acceleration while optimizing convergence speed and solution quality. Table 1 lists the common notation involved in the subsequent contents.

Table 1: Common Notation

Notation	Implication
n	The number of cities in TSP
m	The number of ants in ACO
$\mathbf{M}_\tau$	Adjacency matrix of pheromone concentration
$\mathbf{M}_p$	Adjacency matrix of transition probability
$\eta_{i,j}$	Heuristic function values for path i to j
$\mathcal{N}_i$	Feasible neighborhood solutions of city i
κ	Key of the pseudo-random number generator
$\mathbf{r}$	Generated uniform random tensor
$\odot$	Elementwise product for two tensors

3.1 Framework

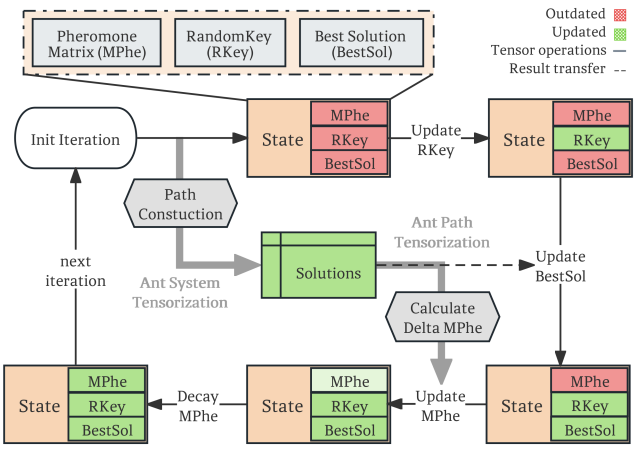


Figure 1: Iteration process of TensorACO. The framework is basically consistent with ACO. Each iteration is only related to the state, and each part of the state is continuously updated based on the results obtained from tensor operations.

The path construction process of TensorACO is in the form of function, thus the process is deterministic with all random number generation keyed. These processes are interconnected through a predicate state mechanism, consisting of three elements: the pheromone matrix, the optimal solution, and the pseudo-random number generator (PRNG) key. The detailed process is shown in Figure 1.

For the path construction process, we abstract it into a function F that takes inputs including the starting point u_0 of the ant, the transition probability matrix $\mathbf{M}_p$, and the random key κ. The output of this function is a constructed path, known as the solution tensor $\mathbf{s}$. F is an outer function that can be mapped onto a tensor.

$$F(u_0, \mathbf{M}_p, \kappa) = \mathbf{s}. \tag{1}$$

Inside the function F, we define a function f, which is the function for each ant to choose the next city under the current city. Algorithm 1 is the pseudocode for function f, which takes a triple as input: $(u, \mathbf{v}_p, \mathbf{v}_a)$. u represents the current city, $\mathbf{v}_p$ represents the tensor of visited paths, and $\mathbf{v}_a$ represents the tensor of remaining unvisited cities.

Algorithm 1 Ant Movement

Input: current state S including current city u, path array $\mathbf{v}_p$, and visited array $\mathbf{v}_a$
Output: next state $\mathbf{S}'$
$u, \mathbf{v}_p, \mathbf{v}_a \leftarrow \mathbf{S}$
$\mathbf{t}_{\text{row}} \leftarrow \mathbf{M}_{p_u} \odot \mathbf{v}_a$
$\mathbf{r} \leftarrow \text{Uniform}(\kappa, m)$
$\mathbf{r}' \leftarrow \text{AdaIR}(\mathbf{r})$
$u' \leftarrow \arg\max(\mathbf{t}_{\text{row}} \odot \mathbf{r}')$
$\mathbf{v}_p[i+1] \leftarrow u'$
$\mathbf{v}_a[u'] \leftarrow 0$
$\mathbf{S}' \leftarrow u', \mathbf{v}_p, \mathbf{v}_a$

During the path construction process of a single ant, this function is called multiple times iteratively, and a complete path solution $\mathbf{s}$ is constructed and passed back to F. The process can be abstracted as follows:

$$\begin{aligned}(u_0, \mathbf{v}_p^{(0)}, \mathbf{v}_a^{(0)}) &\xrightarrow{f} (u_1, \mathbf{v}_p^{(1)}, \mathbf{v}_a^{(1)}),\\ &\cdots\\ &\xrightarrow{f} (u_{m-1}, \mathbf{v}_p^{(m-1)}, \mathbf{v}_a^{(m-1)}).\end{aligned}$$

This approach emphasizes decomposition into small, composable functions. The function F and f are reusable, facilitating concurrent and parallel processing. The absence of shared mutable state makes the system more amenable to parallel computing, efficiently utilizing multi-core processors and distributed systems since functions can be conveniently mapped using `vmap` or `pmap`.

3.2 Tensorization

In ACO, individuals in a population need to construct a sequence of paths, traversing the entire city map to compute route lengths. TensorACO utilizes the `vmap` for mapping both the route construction and state update processes. A unified tensor serves as the input for both, producing a uniformly shaped output tensor. The schematic overview of TensorACO is shown in Figure 2.

The uniform tensorized structure meets requirements for automatic computation framework construction, and reduces computational complexity by representing high-dimensional data in tensor forms, effectively utilizing capabilities for accelerating linear algebra operations. Additionally, tensorization leverages GPU memory storage, reducing computational resource requirements and enhancing the scalability of the algorithm.

3.2.1 Preprocessing. In normal ACO computations, when ants perform Roulette Wheel selection, it is necessary to calculate the selection probabilities for neighboring cities. Typically, this probability is computed using the following expression:

$$p_{i,j}^{(t)} = \frac{[\tau_{i,j}^{(t)}]^{\alpha} \times \eta_{i,j}^{\beta}}{\sum_{k \in \mathcal{N}_i} \left[[\tau_{i,k}^{(t)}]^{\alpha} \times \eta_{i,k}^{\beta} \right]}, \quad j \in \mathcal{N}_i. \tag{2}$$

However, this can be computationally expensive as it requires the hardware to calculate the power of two floating-point numbers (although they are often integers) - α and β. Moreover, in the case of TSP, the constructed graph is usually dense, resulting in a computational complexity of $O(nm^2)$, which slows down the probability transition tensor calculation process.

To address this issue, we introduce preprocessing. Considering that heuristic function values (usually using $\eta = 1/dist_{i,j}$) and pheromone concentrations do not change within an iteration, at the beginning of each iteration, we preprocess the pheromone matrix $\mathbf{M}_\tau$ to compute the probability transition matrix $\mathbf{M}_p$:

$$\mathbf{M} = \mathbf{M}_\tau^{\alpha} \times \eta^{\beta}, \tag{3}$$

$$\mathbf{M}_p = \text{RowNormalized}(\mathbf{M}_{i,j}) = \frac{\mathbf{M}_{i,j}}{\sum_{j=1}^{n} \mathbf{M}_{i,j}}. \tag{4}$$

Then, during each ant's calculation of transition probabilities, we only need to select the preprocessed probability transition tensor $\mathbf{p}_i = \left[\mathbf{M}_{p_{i,1}}, \mathbf{M}_{p_{i,2}}, \cdots, \mathbf{M}_{p_{i,n}}\right]$, from $\mathbf{M}_p$ without performing any additional computations.

3.2.2 Ant System Tensorization. According to the previous description, we utilize functional programming and treat the ant movement as a function. This function takes a tensor as input, which is generated based on the state at the end of the previous iteration. The preprocessed tensor includes the transition probability matrix $\mathbf{M}_p \in \mathbb{R}^{m \times m}$, which is composed together with the random starting point and random key. The random starting point represents the initial city for each ant, and it is generated as a tensor $\mathbf{T}_{\text{start}} = \left[v_1, v_2, \cdots, v_n\right]$. It contains randomly generated integer $v_i \in \mathbb{Z}^+ \cap [0, m)$, $\forall i = 1, 2, \ldots, n$. This tensor represents the ants' random departure cities. Additionally, the random key is a tensor $\mathbf{T}_{\text{key}} = \left[\kappa_1, \kappa_2, \cdots, \kappa_n\right]$, obtained by splitting the original random key tensor. This step ensures determinacy.

For the transition probability matrix $\mathbf{M}_p$, we reshape it into a one-dimensional tensor $\mathbf{v}_p \in \mathbb{R}^{m^2}$ and then perform a duplication operation, resulting in a tensor $\mathbf{T}_p \in \mathbb{R}^{n \times m^2}$. By merging these three tensors, we obtain a tensor $\mathbf{T}_{\text{input}} \in \mathbb{R}^{n \times (2+m^2)}$. Lastly, the tensor is reshaped along the ant dimension to complete the tensorization of the ant colony. From the perspective of the first dimension, all the necessary information for each ant is contained within this tensor.

3.2.3 Ant Path Tensorization. After generating a solution set from the ants, as illustrated in Figure 2, it is crucial to analyze the set of paths and update the pheromone matrix accordingly. Different ACO variants employ distinct workflows during this process. For

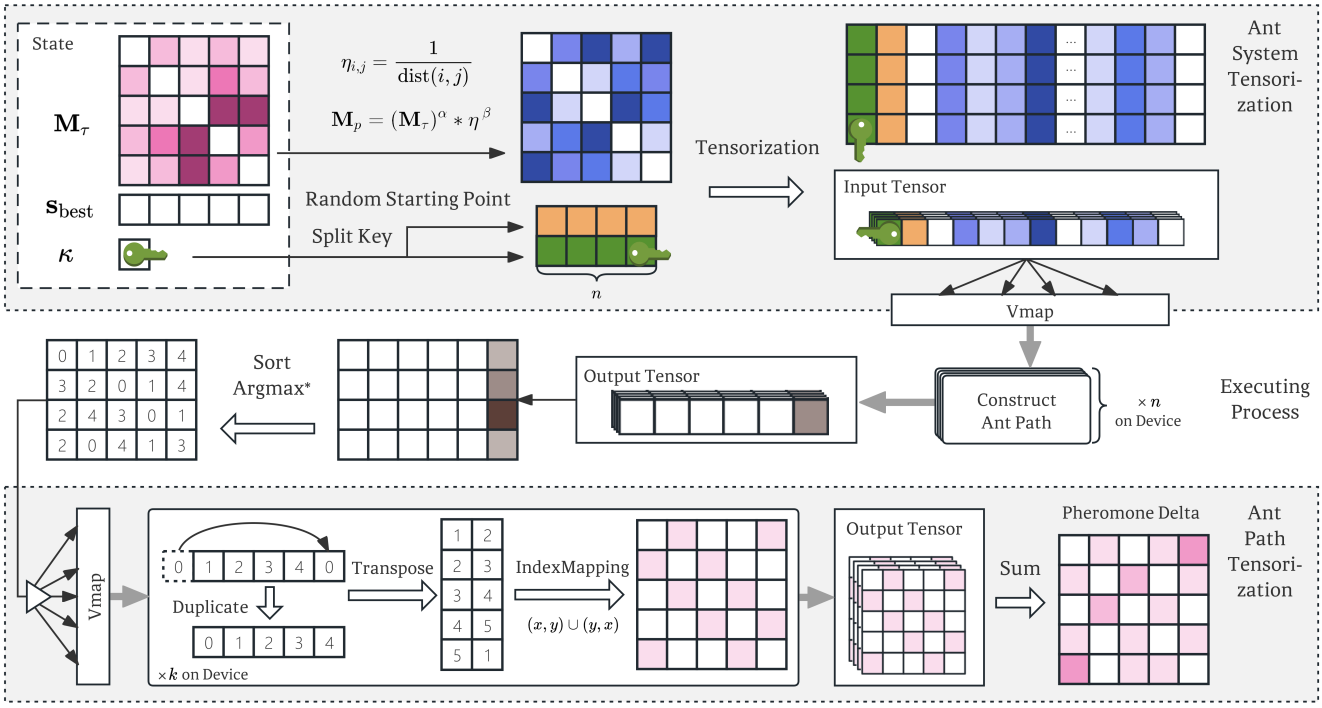


Figure 2: Schematic overview of TensorACO. The workflow comprises two main components: *ant system tensorization* and *ant path tensorization*. The matrix of squares in the figure represents the pheromone matrix or probability transition matrix, and the color depth represents the value. The key represents κ. Multiple arrows represent the mapping of function, and multiple overlay sets pointed by the arrows can be parallelized on device.

instance, some algorithms, like the Max-Min Ant System, update the pheromone matrix based solely on the optimal ant's path. In contrast, many variants consider multiple paths for updates, such as those of the top k ranked ants.

In this paper, we specifically developed a method for path tensorization, catering to variants that require referencing multiple paths for updates. Upon obtaining the solution set $\mathbf{T}_{\text{output}}$, we sort it and select the top k paths. The selected ant path solution set is described as $\mathcal{P} = \left[\mathbf{s}_1, \mathbf{s}_2, , \cdots, \mathbf{s}_l, \cdots, \mathbf{s}_k\right]$, which is the input in the function mapping process, mapped along dimension $\mathbf{s}_l$.

During the processing of the solution, we duplicate each path twice with a one-unit shift to construct the index matrix

$$\mathbf{I} = \left(\begin{bmatrix} p_1 & p_2 & \cdots & p_{m-1} & p_m \\ p_m & p_1 & p_2 & \cdots & p_{m-1} \end{bmatrix} \right)^T. \tag{5}$$

An increment matrix is then generated as follows:

$$\mathbf{A}_{i,j} = \begin{cases} \frac{1}{f(\mathbf{s}_l)}, & \exists k,\ \mathbf{e}_k\mathbf{I} = [i,j] \vee \mathbf{e}_k\mathbf{I} = [j,i] \\ 0, & \text{Otherwise} \end{cases}, \tag{6}$$

where f is the fitness function, $\{\mathbf{e}_1, \cdots, \mathbf{e}_m\}$ are the standard basis vectors in $\mathbb{R}^m$. Finally, by combining these matrices, we obtain the final pheromone increment

$$\Delta\mathbf{M}_{\tau_{i,j}} = \sum_{\mathcal{P}} \mathbf{A}_{i,j}. \tag{7}$$

This pheromone increment is used to update the pheromone matrix $\mathbf{M}_\tau$ in the state, completing the iteration process.

3.3 Adaptive Selection Mechanism

In this part, we introduce the background of the selection process in ACO and discuss the procedure and limitations of the Roulette Wheel (RW) and the Independent Roulette (IR). Next, we propose the AdaIR method, together with a theoretical analysis of it.

3.3.1 Roulette Wheel. In the application of ACO to TSP, most variant algorithms operate as follows. Each ant, acting as an independent entity, traverses every node of the graph within an iteration, essentially constructing a Hamiltonian circuit. During this process, ants face multiple decision points, namely, selecting the next city to visit from their current location. In ACO, this selection is facilitated by a probability roulette wheel mechanism. When an ant is in city u, it obtains a probability tensor $\mathbf{p}_u$ for traveling to all adjacent cities. Subsequently, by incorporating the available city tensor $\mathbf{v}_a$, a modified tensor $\mathbf{p}'_u$ is derived. This is then normalized to obtain the final probability tensor $\hat{\mathbf{p}}_u$.

Upon obtaining this probability tensor, the next city is chosen through a roulette wheel method, as described by the equation

$$v = \min\left\{k \,\middle|\, \sum_{i=1}^{k} p_i \geq r,\ k \in [1,m]\right\}, \tag{8}$$

where r is a randomly generated number in $\mathbb{R} \cap [0, 1]$.

Typically, in a standard ACO iteration, the time complexity of this process is $O(nm^2)$. Additionally, this selection method is very difficult to parallelize. Many attempts at parallelization have introduced additional computations, increased computational complexity, or relied on methods with high algorithm complexity. This has posed significant challenges to the parallelization of the ACO algorithm.

3.3.2 Independent Roulette. The IR method replaces the traditional RW process with a parallel-compatible alternative. This method introduces a random deviate, a randomly generated tensor $\mathbf{r} \in \mathbb{R}^m$. The process of selecting the next city is abstracted as:

$$v = \arg\max(\mathbf{r} \odot \mathbf{p}_u). \tag{9}$$

Intuitively, we understand that a larger probability multiplied by a random number is more likely to result in the maximum value, and vice versa. Hence, this method preserves the magnitude differences and relative positions of the original probabilities. However, this process is not a perfect replication of the original probabilities.

Lloyd, in his analysis of IR [16], concluded the following. Assuming that the probabilities in $\mathbf{p}$ are sorted in ascending order, with $p_{\max}$ being the highest probability in the original set, the modified probability by the IR process for $p_{\max}$ is $p'_{\max}$. We have

$$p'_{\max} \geq p_{\max}, \tag{10}$$

$$p'_{max} = 1 - \sum_{i=1}^{m-1} \frac{1}{(m-i)[(m+1)-i]} \frac{p_i^{(m-i)}}{\prod_{j=i+1}^{m} p_j}. \tag{11}$$

In our extended analysis, we scrutinized the convergence behavior of the ACO algorithm for different numbers of cities in TSP instances.

Our experimental findings in Figure 7 reveal that, in fact, the variations of $p_{\max}$ before and after applying the ACO algorithm in larger city counts, based on sampled probability tensors during the algorithm's execution, are contingent on the iteration changes. Moreover, their distribution consistently lies above the $p_{\max} = p'_{\max}$ line, corroborating the initial analytical results. This implies that ants are more likely to choose paths that previously yielded better results. We interpret this feature as an increase in the algorithm's *learning rate*.

3.3.3 Adaptive Independent Roulette. Based on the above, we introduce the AdaIR, which leverages a *learning rate* hyperparameter, denoted as γ. This parameter modifies each element of the original random tensor by exponentiation with γ. The mathematical representation of this operation is as:

$$\mathbf{r}' = (r_1^\gamma, r_2^\gamma, r_3^\gamma, \ldots, r_m^\gamma), \tag{12}$$

where $\mathbf{r}$ signifies the randomly generated tensor, with its elements r_i uniformly distributed within the interval $(0, 1)$, i.e., $X \sim \mathbf{U}(0, 1)$.

Let Y represent the transformed variable after exponentiation by γ, such that:

$$Y = X^\gamma. \tag{13}$$

Utilizing the Cumulative Distribution Function's definition, we can formulate $F_Y(y)$ by integrating Equation (13):

$$F_Y(y) = P(Y \leq y) = P(X \leq y^{\frac{1}{\gamma}}). \tag{14}$$

By aligning Equation (14) with the definition of CDF, we derive the probability distribution for r_i^γ as:

$$F_Y(y) = \begin{cases} 0, & y \leq 0 \\ y^{\frac{1}{\gamma}}, & 0 < y < 1 \\ 1, & y \geq 1 \end{cases}. \tag{15}$$

The Probability Density Function (PDF) of Y can be obtained by differentiating Equation (15), resulting in:

$$f_Y(y) = \begin{cases} \frac{1}{\gamma} y^{\frac{1-\gamma}{\gamma}}, & 0 < y < 1 \\ 0, & \text{Otherwise} \end{cases}. \tag{16}$$

This formulation allows us to visualize the probability distribution before and after the adaptive adjustment.

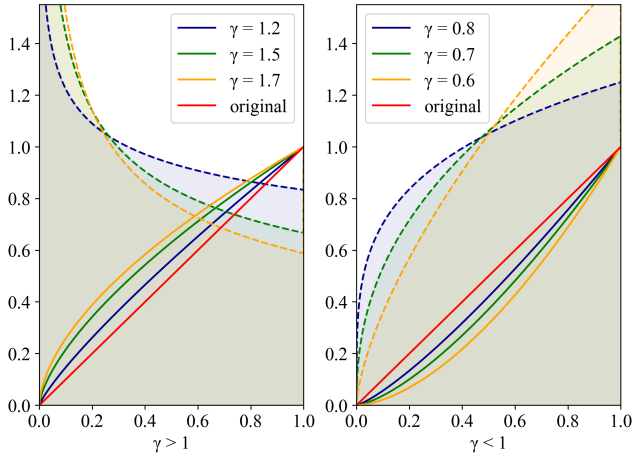


Figure 3: CDF and PDF plot of the random components r_i before (original) and after (varied γ) adaptive transformation. The solid line represents CDF, and the dashed line represents PDF. The area of the semi-transparent zone represents the probability of r_i falling in the zone.

As illustrated in Figure 3, the distribution's shape varies with γ: for $\gamma > 1$, the distribution skews towards smaller values, implying a greater degree of randomization in the adapted probability tensor. Conversely, for $\gamma < 1$, the distribution favors larger values. This modulation of γ directly influences the algorithm's *learning rate*, enhancing its adaptability and randomness in selection.

In AdaIR, we adopt the Cosine Annealing learning rate schedule as the adjustment strategy to dynamically tune the gamma hyperparameter during the iteration process, making the convergence process more rational and efficient.

4 EXPERIMENTS

In this section, we present the experiments for assessing TensorACO's performance in terms of acceleration, convergence, and the AdaIR ablation study. The benchmarked methods include standard sequential version of ACO on CPU (CPU-ACO), TensorACO on CPU (CPU-TensorACO), and TensorACO on GPU (GPU-TensorACO). All experiments were conducted on a uniform platform:

- CPU: Intel Core i7-10750H CPU @ 2.60GHz
- GPU: NVIDIA A100 Tensor Core GPU
- Version: Python 3.9 / CUDA 12.2 / JAX 0.4.23

The dataset employed in this experiment is the TSPLIB dataset [19], which is widely utilized in research.

4.1 Acceleration Performance

We compared the acceleration performance of CPU-ACO, CPU TensorACO, and GPU TensorACO under the same iteration.

4.1.1 Acceleration Across Different Population Sizes. For scaling population sizes, tests were conducted from 10^0 to 10^4, with intervals of every 0.25 power. The algorithm was executed 1000 times in 5 independently random keys, and the average time per iteration was calculated.

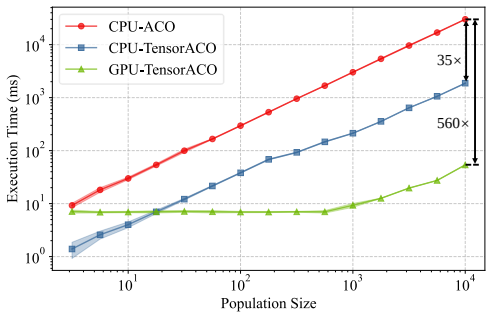


Figure 4: Runtime over scaling population size. GPU-TensorACO and CPU-TensorACO achieve over 560× and 35× speedups against CPU-ACO, respectively.

The results depicted in Figure 4 demonstrate significant acceleration for TensorACO on both CPU and GPU platforms. Compared to CPU-ACO, CPU-TensorACO achieves up to a 35× speedup, while GPU-TensorACO reaches as much as 560×. This underscores the substantial speedup achieved through the tensorization of the algorithm. CPU-TensorACO consistently outperforms CPU-ACO across various population sizes, with both showing similar trends in execution time. On GPUs, for population sizes up to approximately 10^3, the performance of GPU-TensorACO remains largely consistent, suggesting that the GPU's multiple parallel cores efficiently manage large population computations with minimal time increase.

4.1.2 Acceleration Across Different City Numbers. Six representative instances of TSPLIB (*u159, pcb442, p654, u724, pcb1173,* and *pr2392*), corresponding to small, medium, and large scales, were employed to test the acceleration effect of TensorACO. The average time per iteration was recorded, setting $m = n$ and choosing the number of ant paths $k = n/10$, with a fixed 1000 iterations conducted 5 times.

Figure 5 compares the execution times of CPU-ACO and GPU-TensorACO across varying city scales. Notably, for the largest city scale of 2392, CPU-ACO's runtime exceeds the tolerance limit of 10^6 milliseconds, resulting in N/A. The data reveal a rapid increase

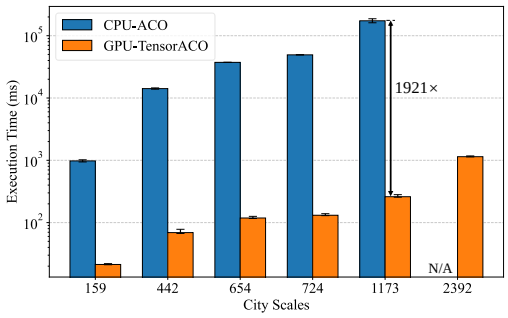


Figure 5: Runtime for different instances with varying city scales. *Note*: No data is available at the city scale of 2392 as the runtime of CPU-ACO exceeded the tolerance range.

in CPU-ACO's execution time with larger problem scales, highlighting its computational limitations on a CPU. Conversely, GPU-TensorACO consistently shows lower execution times across all scales, indicating superior computational efficiency.

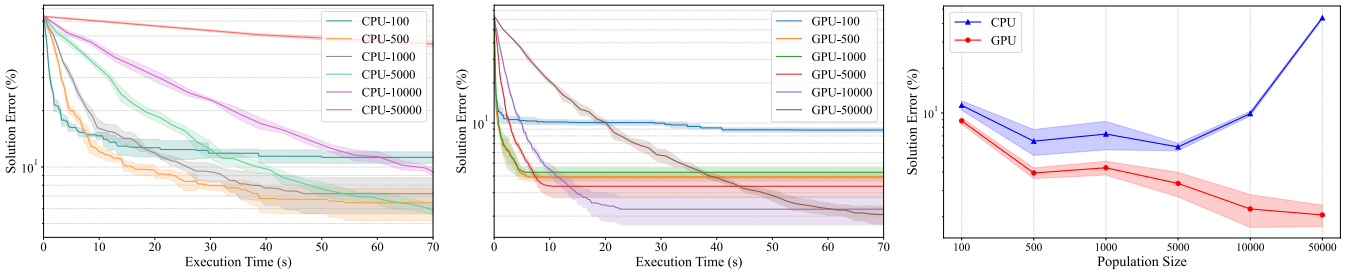
For instance, in *pcb1173*, TensorACO achieves a speedup of 1921× is achieved. This significant difference underscores the benefits of GPU acceleration in the TensorACO framework, especially for large-scale problems where computational demands are higher. The failure of CPU-ACO to complete the largest instance within the tolerance range further accentuates the importance and efficacy of GPU acceleration in advanced computational optimization tasks.

4.2 Convergence Performance

We conducted an experiment to assess the convergence performance of TensorACO. Using the same natural time limit of 70s, we compare the performance of CPU-TensorACO and GPU-TensorACO on an instance of 500 cities. To ensure fairness, each instance is run independently five times, and results were evaluated based on the solution errors, *i.e.*, percentage difference from a specific solution. We experimented with $n = 100, 500, 1000, 5000, 10000, 50000$, and recorded convergence curves for both algorithms, comparing different population sizes and analyzing the final results.

As shown in Figure 6, the results reveal the influence of colony size on TensorACO's convergence speed and accuracy on CPU and GPU platforms. Both platforms show that larger populations slow convergence but improve solution accuracy, indicating that broader searches yield more precise outcomes. This is attributed to GPUs' superior computational efficiency over CPUs at similar population scales. Figure 6 (c) compares TensorACO's performance on CPU and GPU over 70 seconds at different population sizes. At 10,000 and 50,000 populations, the performance on CPU is not optimal, likely due to convergence challenges in expansive search spaces. This suggests that while larger populations enhance performance, the limiting factor is the algorithm's computational speed.

The efficacy of GPU-accelerated TensorACO, leveraging GPU's computational prowess, confirms that larger populations can improve performance given sufficient speed. These findings emphasize the need to balance population size with computational capacity for



(a) CPU-TensorACO (b) GPU-TensorACO (c) Comparison Result

Figure 6: Solution error for CPU-TensorACO and GPU-TensorACO with varying population sizes. (a) and (b): Convergence curve over runtime; (c): Final solution quality obtained by executing the two models for 70s. On GPUs, a larger population size helps to achieve better results.

optimal algorithm efficiency. Furthermore, allocating more computational resources and increasing ant colony size are hypothesized to significantly enhance the quality of solution after convergence.

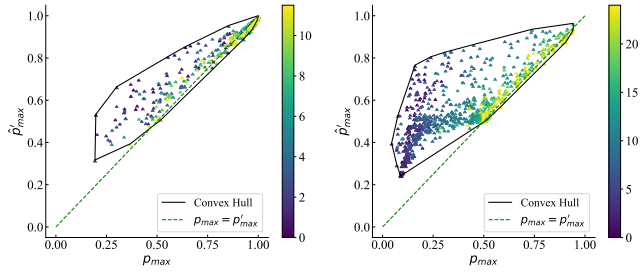
4.3 Ablation Study on AdaIR

4.3.1 Visualization. Acknowledging the large number of cities usually involved in TSP problems, we chose city counts of $m = 100$ and $m = 422$ for our study. In typical TSP scenarios, we sampled probability tensors in the path construction process. We only consider the city with the highest original probability $p_{\max}$. The AdaIR method is then applied to these tensors, with γ hyperparameter constantly set to 1.5. We performed $m \times 10,000$ Bernoulli trials on the tensors, yielding $\hat{p}'_{\max}$. Gradient color markers in our analysis represent the maximum probability values under RW probability $p_{\max}$ and the shifted probability $\hat{p}'_{\max}$ utilizing AdaIR. As Figure 7 shows, the shift of probability values correlates with different stages of the algorithm. In the initial few generations, the probability was shifted laterally to a higher level and gradually became the same as RW.

Figure 8 displays the probability ranges for four different γ values, depicted as a convex hull. As γ increases, the convex hull expands downwards, broadening the range of the probability of choosing other cities, and encouraging more diverse choices in the search space. Specifically, when $\gamma < 1$, the probability amplifies, favoring selection of the initially superior paths; conversely, when $\gamma > 1$, the probability diminishes, leading to a more randomized path selection. The implementation of the *learning rate* γ facilitates a more dynamic range of the IR convex hull's expansion and contraction, making the IR method more adaptable to the search process's dynamic needs and improving the balance between exploration and exploitation.

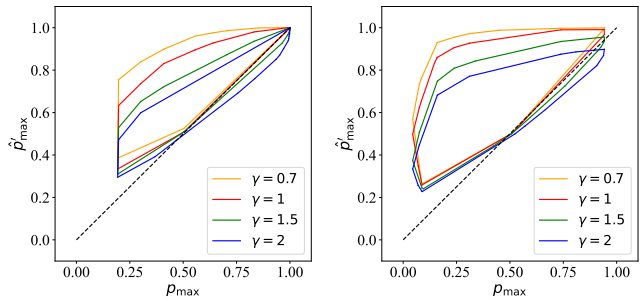
4.3.2 Acceleration. We compared RW, IR, and AdaIR methods using the same settings as described above, testing the algorithm runtime per iteration on various instances with different city numbers.

In Figure 9, the runtime performance of IR and AdaIR is compared across different city scales. AdaIR demonstrates a speedup of 2.12× to 4.62× over RW, with negligible differences from IR. The bar charts in the runtime data distinctly highlight the efficiency gains of IR and AdaIR over RW. Across all tested instances, IR methods consistently achieve over a 2× speedup compared to RW. While AdaIR is slightly slower than IR, owing to the added computational



(a) $m = 100$ (b) $m = 442$

Figure 7: Relationship between RW probability $p_{\max}$ and shifted probability $\hat{p}'_{\max}$ using AdaIR. The gradient color represents the variation of iteration. $\hat{p}_{\max}$ and $\hat{p}'_{\max}$ tend to align within fewer initial iterations.



(a) $m = 100$ (b) $m = 442$

Figure 8: Convex hull composed of probability change under different γ. With the changes of γ, ants have a directional change in their selection tendency.

load of exponential probability calculations, this minor delay is generally regarded as negligible in the context of the overall performance gains. This highlights AdaIR's feasibility as a competitive alternative to IR in terms of execution speed while offering the additional advantage of adaptive solution quality enhancements.

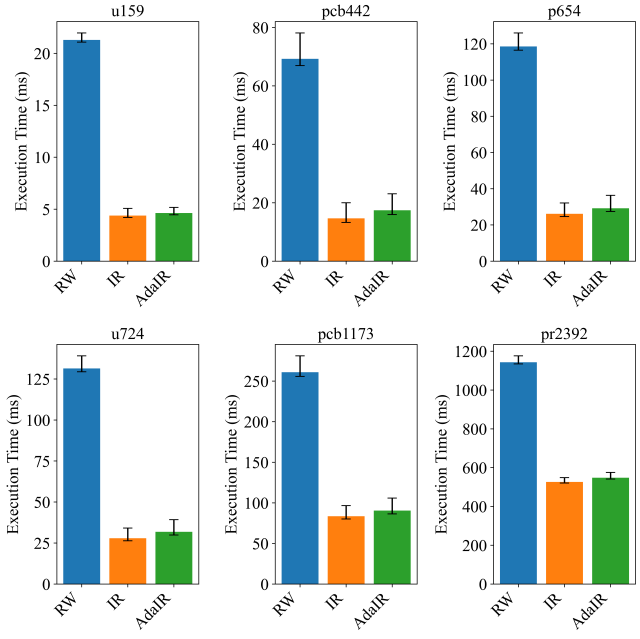


Figure 9: Runtime for RW, IR, and AdaIR on instances of varying city scales. AdaIR achieves speedup ranging from 2.12× to 4.62× against RW, with only a slight time difference from IR.

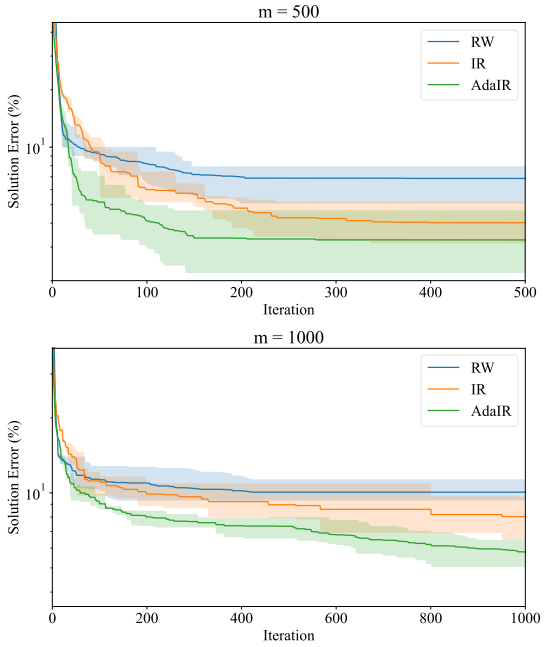


Figure 10: Convergence curves of RW, IR, and AdaIR over iteration. AdaIR outperforms RW and IR both in terms of convergence speed and solution error.

4.3.3 Convergence. We conducted an evaluation of the convergence for RW, IR, and AdaIR. This involved 1000 iterations, replicated five times, focusing on city scales of 500 and 1000. Focusing on the distinct aspects of the convergence process, we document both the range and mean quality across trials.

Figure 10 shows that for $m = 500$, AdaIR converges in 165 generations, outperforming RW, which takes 310 generations, and IR in terms of solution quality. This enhanced speed is due to AdaIR's power-law *learning rate*, which tends to favor higher probabilities in the initial iterations, resulting in faster convergence compared to IR and RW. AdaIR's initial preference for promising solutions helps accelerate movement towards optimal regions, while its adaptive strategy ensures a more diversified search in later iterations. This adaptability allows comprehensive exploration and higher convergence accuracy, reducing stagnation risk and potentially improving the likelihood of finding globally optimal solutions.

Figure 10 reveals that AdaIR's adaptive mechanism not only expedites initial convergence but also ensures steady progress toward high-quality solutions. This blend of rapid early convergence and sustained solution excellence, fostered by an adaptive, diverse search approach, underscores AdaIR's effectiveness in complex optimization challenges.

5 CONCLUSION

This study presents the development and evaluation of a Tensorized Ant Colony Optimization (TensorACO) for addressing large-scale Traveling Salesman Problems (TSP). The proposed TensorACO algorithm demonstrates its efficacy through substantial computational speedups and scalability, making it a viable solution for complex, large-scale TSP instances.

The tensorization method, combined with the GPU's parallel processing capabilities, addresses the inherent computational bottlenecks of traditional ACO. As indicated by the experimental results, TensorACO achieves an acceleration of approximately 2000 times compared to the standard ACO. Moreover, the Adaptive Independent Roulette (AdaIR) method introduced in this study further enhances the convergence speed of the algorithm, without sacrificing the solution qualities. The adaptive nature of this mechanism enables the algorithm to dynamically adjust its search strategy, striking a balance between exploration and exploitation.

Future work could focus on extending the tensorized framework and further exploring AdaIR. The potential of TensorACO in various other challenging optimization problems is also worth investigating.

REFERENCES

[1] Hongtao Bai, Dantong OuYang, Ximing Li, Lili He, and Haihong Yu. 2009. MAX-MIN ant system on GPU with CUDA. In *2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC)*. IEEE, 801–804.

[2] Ali Balma, Mehdi Mrad, and Talel Ladhari. 2023. Tight lower bounds for the Traveling Salesman Problem with draft limits. *Computers & Operations Research* 154 (2023), 106196.

[3] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[4] José M Cecilia, José M García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. 2013. Enhancing data parallelism for ant colony optimization on GPUs. *J. Parallel and Distrib. Comput.* 73, 1 (2013), 42–51.

[5] Agung Chandra, Christine Natalia, and Aulia Naro. 2021. Comparative Solutions of Exact and Approximate Methods for Traveling Salesman Problem. *Lámpsakos*

(revista descontinuada) 25 (2021).

[6] Darren M Chitty. 2018. Applying ACO to large scale TSP instances. In *Advances in Computational Intelligence Systems: Contributions Presented at the 17th UK Workshop on Computational Intelligence*. Springer, 104–118.

[7] Laurence Dawson and Iain Stewart. 2013. Improving Ant Colony Optimization performance on the GPU using CUDA. In *2013 IEEE Congress on Evolutionary Computation*. 1901–1908.

[8] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. 1996. Ant system: optimization by a colony of cooperating agents. *IEEE Trans. Syst. Man Cybern. Part B* 26, 1 (1996), 29–41. https://doi.org/10.1109/3477.484436

[9] Ivars Dzalbs and Tatiana Kalganova. 2020. Accelerating supply chains with Ant Colony Optimization across a range of hardware solutions. *Computers & Industrial Engineering* 147 (2020), 106610.

[10] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. 2023. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 804–817.

[11] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4, 9 (2018).

[12] Beichen Huang, Ran Cheng, Zhuozhao Li, Yaochu Jin, and Kay Chen Tan. 2023. EvoX: A Distributed GPU-accelerated Framework for Scalable Evolutionary Computation. arXiv:2301.12457 [cs.NE]

[13] Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi. 1995. The traveling salesman problem. *Handbooks in Operations Research and Management Science* 7 (1995), 225–330.

[14] Robert Tjarko Lange. 2023. evosax: JAX-Based Evolution Strategies. In *Proceedings of the Companion Conference on Genetic and Evolutionary Computation* (Lisbon, Portugal) *(GECCO '23 Companion)*. Association for Computing Machinery, New York, NY, USA, 659–662.

[15] Gilbert Laporte. 1992. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research* 59, 2 (1992), 231–247. https://doi.org/10.1016/0377-2217(92)90138-Y

[16] Huw Lloyd and Martyn Amos. 2017. Analysis of independent roulette selection in parallel ant colony optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. 19–26.

[17] Breno A. M. Menezes, Herbert Kuchen, Hugo A. Amorim Neto, and Fernando B. de Lima Neto. 2019. Parallelization strategies for GPU-based Ant Colony Optimization solving the Traveling Salesman Problem. In *2019 IEEE Congress on Evolutionary Computation (CEC)*. 3094–3101.

[18] Martín Pedemonte, Sergio Nesmachnow, and Héctor Cancela. 2011. A survey on parallel ant colony optimization. *Applied Soft Computing* 11, 8 (2011), 5181–5197.

[19] Gerhard Reinelt. 1991. TSPLIB-A traveling salesman problem library. *ORSA Journal on Computing* 3, 4 (1991), 376–384.

[20] Gerhard Reinelt. 1994. *The Traveling Salesman, Computational Solutions for TSP Applications*. Lecture Notes in Computer Science, Vol. 840. Springer. https://doi.org/10.1007/3-540-48661-5

[21] Tal Ridnik, Hussam Lawen, Asaf Noy, Emanuel Ben Baruch, Gilad Sharir, and Itamar Friedman. 2021. Tresnet: High performance gpu-dedicated architecture. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 1400–1409.

[22] Rafał Skinderowicz. 2020. Implementing a GPU-based parallel MAX–MIN Ant System. *Future Generation Computer Systems* 106 (2020), 277–295.

[23] Rafał Skinderowicz. 2022. Improving Ant Colony Optimization efficiency for solving large TSP instances. *Applied Soft Computing* 120 (2022), 108653.

[24] Thomas Stützle and Holger H Hoos. 2000. MAX–MIN ant system. *Future Generation Computer Systems* 16, 8 (2000), 889–914.

[25] Yujin Tang, Yingtao Tian, and David Ha. 2022. EvoJAX: hardware-accelerated neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 308–311.

[26] Bladimir Toaza and Domokos Esztergár-Kiss. 2023. A review of metaheuristic algorithms for solving TSP-based scheduling optimization problems. *Applied Soft Computing* (2023), 110908.

[27] Pramod Yelmewad, Aniket Kumar, and Basavaraj Talawar. 2019. MMAS on GPU for large TSP instances. In *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 1–6.

[28] Yi Zhou, Fazhi He, and Yimin Qiu. 2017. Dynamic strategy based parallel ant colony optimization on GPUs for TSPs. *Science China Information Sciences* 60 (2017), 1–3.