electronics

MDPI

Article

Intelligent Path Planning for UAV Patrolling in Dynamic Environments Based on the Transformer Architecture

Ching-Hao Yu [1], Jichiang Tsai [2,*] and Yuan-Tsun Chang [1]

[1] Department of Electrical Engineering, National Chung Hsing University, Taichung 40227, Taiwan; g111064015@mail.nchu.edu.tw (C.-H.Y.); d110064007@mail.nchu.edu.tw (Y.-T.C.)
[2] Department of Electrical Engineering and Graduate Institute of Communication Engineering, National Chung Hsing University, Taichung 40227, Taiwan
* Correspondence: jichiangt@nchu.edu.tw

Abstract: Due to its NP-Hard property, the Travelling Salesman Problem (TSP) has long been a prominent research topic in path planning. The goal is to design the algorithm with the fastest execution speed in order to find the path with the lowest travelling cost. In particular, new generative AI technology is continually emerging. The question of how to exploit algorithms from this realm to perform TSP path planning, especially in dynamic environments, is an important and interesting problem. The TSP application scenario investigated by this paper is that of an Unmanned Aerial Vehicle (UAV) that needs to patrol all specific ship-targets on the sea surface before returning to its origin. Hence, during the flight, we must consider real-time changes in wind velocity and direction, as well as the dynamic addition or removal of ship targets due to mission requirements. Specifically, we implement a Deep Reinforcement Learning (DRL) model based on the Transformer architecture, which is widely used in Generative AI, to solve the TSP path-planning problem in dynamic environments. Finally, we conduct numerous simulation experiments to compare the performance of our DRL model and the traditional heuristic algorithm, the Simulated Annealing (SA) method, in terms of operation time and path distance in solving the ordinary TSP, to verify the advantages of our model. Notably, traditional heuristic algorithms cannot be applied to dynamic environments, in which wind velocity and direction can change at any time.

Keywords: traveling salesman problem; deep reinforcement learning; generative AI; transformer architecture; simulated annealing; unmanned aerial vehicle

check for updates

Citation: Yu, C.-H.; Tsai, J.; Chang, Y.-T. Intelligent Path Planning for UAV Patrolling in Dynamic Environments Based on the Transformer Architecture. *Electronics* **2024**, *13*, 4716. https://doi.org/10.3390/electronics13234716

Academic Editors: Cecilio Angulo and Stefano Scanzio

Received: 30 September 2024
Revised: 6 November 2024
Accepted: 27 November 2024
Published: 28 November 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

In recent years, global geopolitical tensions have escalated, with frequent occurrences of wars and conflicts. Against this backdrop, when Coast Guard personnel spot a suspicious vessel within territorial waters, they typically dispatch an Unmanned Aerial Vehicle (UAV) to conduct a reconnaissance patrol. When multiple vessels are found, the current patrolling method relies on the experience-based rules of Coast Guard personnel to plan a path for the investigation of all the target locations. However, as the number of suspicious vessels increases to ten, twenty, or even fifty, this experience-based path-planning method may fail to find the optimal path. This will make it difficult to respond efficiently to such overwhelming invasion scenarios, potentially leading to security vulnerabilities.

There are numerous advanced algorithms available for devising the shortest patrolling path for the above Travelling Salesman Problem (TSP) [1], a classic combinatorial optimization problem. The goal is to find the shortest path that allows a salesperson to start from one city, visit each city exactly once, and finally return to the starting city. However, the traditional TSP assumes a static environment in which the travelling costs, and usually the distances, between cities remain constant. On the contrary, in practical applications, the TSP often occurs in dynamic environments, such as changing traffic conditions, varying weather patterns, and other unforeseen disruptions. Hence, solving the TSP in these dynamic environments is crucial not only for theoretical researches but also for practical applications, including logistics management, supply chain management, autonomous vehicle routing, and UAV patrols for security, i.e., the context of this paper.

Developing adaptive algorithms capable of rapidly reoptimizing solutions is crucial to address these dynamic challenges. Traditional methods for solving TSP, such as branch and bound [2] and dynamic programming [3], and metaheuristic methods [4,5] like genetic algorithms and simulated annealing [6,7], are often computationally expensive, making them unsuitable for real-time applications in dynamic settings. The rapid development of artificial intelligence (AI) and machine learning, especially with advances in deep learning, has opened new pathways for tackling complex optimization problems like the TSP. The Transformer architecture, initially designed for natural language processing tasks [8], has demonstrated effectiveness in handling sequential data and capturing long-range dependencies. Unlike traditional recurrent neural networks (RNNs), Transformers utilize a self-attention mechanism that enables the parallel processing of entire input sequences, making Transformers particularly well-suited to dynamic and evolving environments.

With the rise of deep learning, recent studies have increasingly applied neural networks to tackle the TSP. In 2018, Kool et al. [9] introduced an attention-based model capable of effectively learning routing tasks, highlighting the potential of neural networks in this domain. In 2019, Marcelo et al. [10] demonstrated that Graph Neural Networks (GNNs) can efficiently solve the decision variant of TSP. Using message-passing and minimal supervision, GNNs can verify sub-cost paths and estimate optimal routes within a 2% deviation, showcasing their potential in handling complex NP-complete problems. In 2019, Qiang Ma et al. [11] presented Graph Pointer Networks (GPNs), using reinforcement learning to solve TSPs, and introducing hierarchical layers and graph embeddings that improve both scalability and solution quality for constrained problems like the TSP with time windows. In 2021, Zhang et al. [12] utilized deep reinforcement learning to address dynamic TSP, demonstrating the promise of reinforcement learning in complex routing scenarios. The latest advancements bring in Graph Language Models (GraphLLMs), which provide innovative approaches to TSP and other graph-based optimization problems. Building on Transformer architectures, GraphLLMs use graph structures to capture complex relationships between nodes, enhancing performance on tasks that require the understanding of topological data. For example, Li et al. [13] in 2024 introduced models that improve accuracy and scalability on large, complex datasets by integrating graph-specific attention mechanisms, making these models well-suited for dynamic environments and offering notable improvements over traditional language models in path optimization. In the same year, Jung et al. [14] presented a lightweight CNN-Transformer model customized for TSP problem-solving, further underscoring the relevance of Transformer architecture in the field of path planning.

Our manuscript focuses on comparing the performances of the Transformer model and the Simulated Annealing algorithm in solving the ordinary TSP across multiple dynamic environments, considering changes in wind velocity and direction and dynamically adding or removing nodes so as to identify the best balance between computation time and efficiency, achieving the shortest route planning with the lowest cost. Additionally, this manuscript explores the scalability and stability of these two kinds of algorithms when dealing with TSPs with high-dimensional factors and numerous nodes. We expect that the findings of our work can provide practical references for maritime patrol and other related fields, enhancing the operational efficiency of drones and other automated systems in complex dynamic environments. The contributions of this paper are primarily reflected in the following aspects:

1. Model Architecture and Inference Process Improvement: This work modifies the DRL model architecture and inference process used to solve the ordinary TSP problem to accommodate dynamic changes, such as variations in wind speed and the addition or removal of nodes, aiming to meet the requirements of real-world scenarios.

2. Hybrid Training Methods: We propose a hybrid training approach to the improvement of convergence in challenging models, integrating different training strategies, and thereby enhancing overall performance.
3. Comparison of Generative Models and Traditional Algorithms: This paper evaluates the strengths and weaknesses of contemporary generative models (e.g., Transformers) versus traditional algorithms (e.g., Simulated Annealing), providing valuable insights for future research.
4. Scalability and Stability Evaluation: The study assesses the scalability and stability of both algorithms when handling large datasets and high-dimensional challenges, offering practical guidance for improving automated system efficiency.
5. Practical Reference Value: Our findings aim to serve as practical references for maritime patrols and related fields, enhancing the operational efficiency of drones and automated systems in dynamic environments.

The structure of this paper is as follows: Section 2 describes the architecture of the Transformer model improved for the dynamic TSP and the Simulated Annealing algorithm proposed for solving the ordinary TSP, including a detailed explanation of the experimental setup and the data utilized. Section 3 presents the results of our experiments, comparing the performance of the Transformer model with that of Simulated Annealing across various dynamic scenarios. Section 4 analyzes the experimental results, highlighting both the strengths and limitations of our method. Finally, Section 5 summarizes the key findings of our work and introduces some potential directions for future research.

2. Our Methods

In this manuscript, we primarily investigate the performance and accuracy of the Transformer-based [15] generative model and the Simulated Annealing algorithm under four distinct scenarios. The first scenario establishes a baseline by considering a static environment devoid of any dynamic factors. The second scenario introduces wind speed and direction into the static environment. The third scenario examines a dynamic environment where nodes are added or removed, but wind factors are not considered. Finally, the fourth scenario explores a fully dynamic environment that incorporates both wind speed, direction, and node additions/removals. These four scenarios provide a comprehensive framework for comparative analysis, allowing us to evaluate the effectiveness of the methods under diverse conditions.

2.1. Improved Transformer Model Architecture

The generative model employed in this paper utilizes a Transformer architecture, as depicted in Figure 1, which was created with reference to [8,14]. The model architecture comprises three primary components: the model input, the encoder, and the decoder. We will introduce the details of Figure 1 in the subsequent subsections.

2.1.1. Model Input

In this paper, we convert node coordinates (representing the latitude and longitude of the vessels) into flight time. To achieve faster response times and fuel efficiency and ensure rapid inspection of suspicious vessels along the shortest possible path, while also enhancing the system's scalability and adaptability to larger search areas, we must utilize the great-circle distance rather than the rhumb-line calculation method for distance measurements. In doing so, we employ the haversine formula to calculate the great-circle distance. The haversine formula and its inverse are essential for calculating the distance between two points when given the latitude and longitude coordinates of the points [16,17]. The haversine formula is used to calculate the central angle between two points on a great circle based on their latitude and longitude, as defined in Equation (1):

$$hav\left(\frac{d}{r}\right) = hav(\varphi_2 - \varphi_1) + cos(\varphi_1)cos(\varphi_2)hav(\lambda_2 - \lambda_1) \tag{1}$$

$$haversine\ (\theta) = \frac{1 - cos(\theta)}{2} = sin^2\left(\frac{\theta}{2}\right) \tag{2}$$

where φ_1 and φ_2 represent the latitudes of points i and $i+1$, and λ_1 and λ_2 represent the longitudes of points i and $i+1$. The variable d denotes the distance between the two points, r is the radius of the sphere, and hav is the abbreviation for the haversine function, as shown in Equation (2). In this context, θ represents the central angle in radians. The haversine function is particularly useful for small angle calculations due to its numerical stability, which is crucial for spherical distance computations. Subsequently, the inverse haversine function can be employed to determine the distance between two geographic coordinates, as shown in Equation (3). By substituting Equation (1) into Equation (3), we obtain Equation (4). Finally, incorporating Equation (2) yields Equation (5).

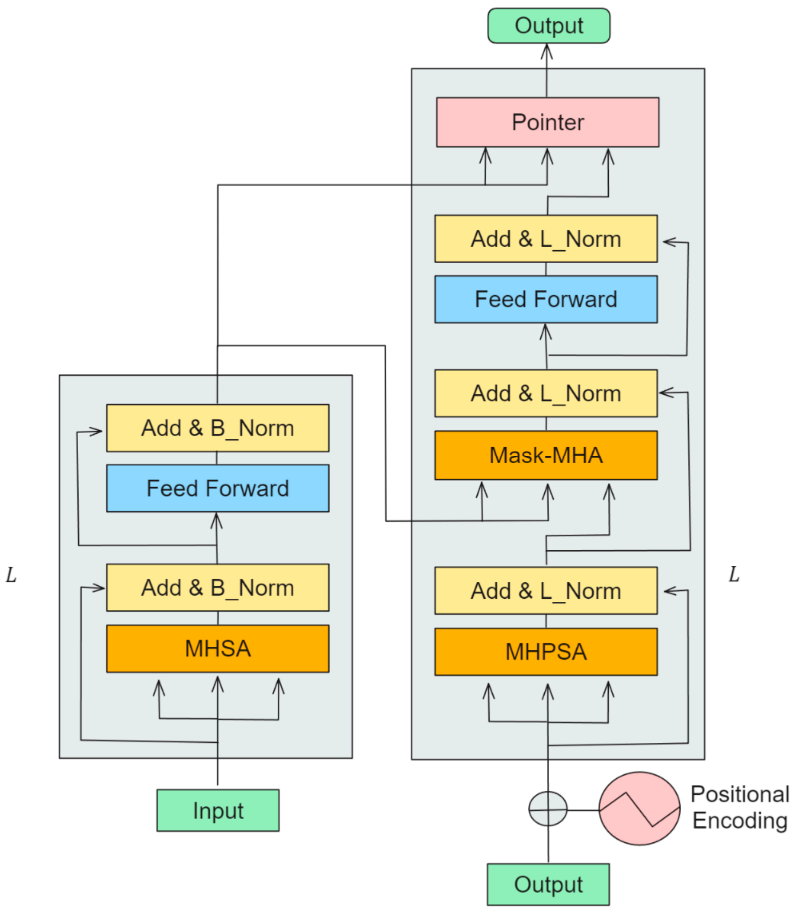


Figure 1. The Transformer model architecture diagram.

$$d = r\ archav(h) = 2r\ arcsin\left(\sqrt{h}\right) \tag{3}$$

$$d = 2r\ arcsin\left(\sqrt{hav(\varphi_2 - \varphi_1) + cos(\varphi_1)cos(\varphi_2)hav(\lambda_2 - \lambda_1)}\right) \tag{4}$$

$$d = 2r\ arcsin\left(\sqrt{sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + cos(\varphi_1)cos(\varphi_2)sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \tag{5}$$

In addition to distance, flight speed is also required to compute flight time. In aviation, speed is commonly expressed as air speed and ground speed [18]. Air speed refers to the speed of the aircraft relative to the surrounding air mass, while ground speed is the speed of the aircraft relative to the ground. Ground speed takes into account not only the speed of the aircraft but also factors such as wind speed, wind direction, and air currents. This paper utilizes the components of air speed, wind speed, and wind direction to derive ground speed, which is then employed in Equation (5) to calculate flight time.

Firstly, the two-argument arctangent function (*atan2*) is used to obtain the angle θ between the flight direction and the x-axis, as well as the angle θ_w between the wind direction and the x-axis, as shown in Equation (6). Here, x and y represent the flight direction vector, while w_x and w_y represent the wind direction vector.

$$\theta = atan2(y,\ x),\ \theta_w = atan2\left(w_y, w_x\right) \tag{6}$$

Next, the components of the flight air speed along the x-axis and y-axis are given by $v\ cos(\theta)$ and $v\ sin(\theta)$, respectively, while those of the wind speed are $w\ cos(\theta_w)$ and $w\ sin(\theta_w)$. Here, v and w represent the flight air speed and wind speed, respectively. These components are then vectorially added to obtain the ground speed vector, as shown in Equation (7). Incorporating this ground speed into Equation (5) yields Equation (8), which is subsequently utilized to calculate the flight time between two points. By applying this method across all nodes, the input for the model is generated.

$$v = \sqrt{\left(v\ cos(\theta) + w\ cos(\theta_w)\right)^2 + \left(v\ sin(\theta) + w\ sin(\theta_w)\right)^2} \tag{7}$$

$$Flight\ Time = \frac{2\gamma\ arcsin\left(\sqrt{sin^2\left(\frac{\varphi_2-\varphi_1}{2}\right) + cos(\varphi_1)cos(\varphi_2)sin^2\left(\frac{\lambda_2-\lambda_1}{2}\right)}\right)}{\sqrt{\left(v\ cos(\theta) + w\ cos(\theta_w)\right)^2 + \left(v\ sin(\theta) + w\ sin(\theta_w)\right)^2}} \tag{8}$$

2.1.2. Encoder

The Transformer model employed in this paper [9,19] comprises six encoder layers. Prior to encoding, the input data undergo embedding to extract spatial features from the flight time matrix. In this study, the embedding is implemented as a linear layer. Initially, each corresponding node x_i in the flight time matrix is multiplied by a trainable weight matrix W and then added to a bias vector b, resulting in the embedded vector x_i^{emb} for each node, as shown in Equation (9). Here, x represents the flight time matrix, i denotes the current node, and *emb* refers to the dimensional space of the embedded vector, which in this case is 128-dimensional.

$$x_i^{emb} = W^{emb}x_i + b^{emb} \tag{9}$$

Each encoder layer consists of four sub-layers, comprising a multi-head self-attention mechanism and a pointwise fully connected feedforward neural network. These sub-layers enhance the model's memory capacity and strengthen its ability to represent data. Additionally, each sub-layer is followed by a residual connection and a batch normalization layer, which help mitigate the vanishing-gradient problem and facilitate model training.

The detailed process is as follows: The input to the multi-head self-attention sub-layer of the Lth encoder layer is $E^{(L-1)}$, the output from the preceding encoder layer. The output of the multi-head self-attention sub-layer in the Lth encoder layer, denoted as

$\hat{E}^L$, is obtained by first applying multi-head self-attention $MHSA^L$ to $E^{(L-1)}$, and this is followed by a residual connection and batch normalization BN^L [20]. The function $MHSA^L(Q, K, V)$ takes three inputs: Q (queries), K (keys), and V (values). All are equal to $E^{(L-1)}$ in the encoder, allowing the multi-head self-attention mechanism to be executed in the Lth encoder layer. The output of the multi-head self-attention sub-layer, $\hat{E}^L$, is computed using the following, Equation (10):

$$\hat{E}^L = BN^L\left(E^{(L-1)} + MHSA^L\left(E^{(L-1)}, E^{(L-1)}, E^{(L-1)}\right)\right) \tag{10}$$

When $L = 0$, the initial embedded vector E^0 serves as the input for feature embedding. Subsequently, $\hat{E}^L$ is passed through a pointwise fully connected feedforward neural network; this is followed by a residual connection and batch normalization, resulting in

Equation (11). In this process, the final output of each encoder layer encodes the n nodes, expressed as $E^L = \left\{ e_f^L, e_1^L, \ldots, e_n^L \right\}$, where f represents the starting node.

$$E^L = BN^L\left(\hat{E}^L + FF^L\left(\hat{E}^L \right) \right) \quad (11)$$

2.1.3. Decoder

The decoder employed in this paper builds upon the work presented in [14]. In addition to embedding vectors, positional encoding is incorporated to account for the sequential order of the nodes. The input to the decoder comprises previously generated nodes. For the initial node, as no previous nodes have been generated, a pre-trained optimal starting point is utilized for inference.

The decoder in this paper consists of multiple layers, each comprising six sub-layers: the Multi-Head Partial Self-Attention Layer (MHPSA), the Masked Multi-Head Attention Layer (MMHA), and the pointwise fully connected feedforward neural network. Subsequent to each sub-layer, a residual connection and layer normalization [21] are applied. This sequence is repeated three times in total. Furthermore, the final layer of the decoder incorporates a Pointer Layer. The subsequent sections provide a detailed elucidation of each sub-layer.

The first layer in the decoder is the MHPSA Layer, designed to extract information from previous steps by applying attention mechanisms to the encoder outputs of previously visited nodes. In contrast to traditional methods, this layer employs fewer reference vectors, as the nodes visited most recently are deemed more pertinent for selecting the next node than those visited earlier. Thus, utilizing fewer reference vectors enhances both performance and computational efficiency.

This layer enables the decoder to attend to relevant information from already visited nodes. It utilizes the input to the decoder at the current time step t, denoted as h_t, as the query vector, and the inputs of recently visited nodes as key and value vectors. The calculation of h_t is shown in Equation (12):

$$h_t = e_{\pi_{t-1}}^L + PE_t \quad (12)$$

where e^L represents the output of the final encoder layer, π_{t-1} represents the index of the node selected in the previous step, and PE_t denotes the positional encoding at time step t. To reduce the number of reference vectors, this layer only considers the last m visited nodes as reference vectors. If the current time step is t, the MHPSA layer uses the decoder inputs from time steps *t-m* to t as key and value vectors, denoted as $H_t = \{h_{t-m}, \ldots, h_t\} \in \mathbb{R}^{m \times d}$, where m represents the number of recently visited nodes.

The output of the MHPSA layer, $\hat{h}_t$, is obtained by applying the multi-head self-attention mechanism MH^{L+1} to the reference vectors, followed by layer normalization. The output $\hat{h}_t$ can then be used in the subsequent decoding process, as shown in Equation (13).

$$\hat{h}_t = LN\left(h_t + MH^{L+1}(h_t, H_t, H_t) \right) \quad (13)$$

The second layer in the decoder is the Masked Multi-Head Attention (MMHA) Layer. In this layer, the query vector is the output $\hat{h}_t$ from the MHPSA layer, while the key and value vectors are the encoder outputs E^L of unvisited nodes. This mechanism enables the decoder to focus on nodes that have not yet been visited, aiding in the determination of the next node to be visited.

This layer is analogous to the multi-head attention layer employed in the encoder but has an additional mask $\zeta_t \in \mathbb{R}^{n+1}$ to distinguish between visited and unvisited nodes. The mask ζ_t is a vector of length $n+1$, where 1 indicates an unvisited node, and 0 indicates a visited node. The output $\widetilde{h}_t$ of the MMHA layer is obtained by applying the modified attention mechanism MMHA (Q, K, V, ζ), as defined in Equation (14), followed by a residual connection and layer normalization, as shown in Equation (15). The *MMHA* function ensures that the attention weights for visited nodes are zero, while calculating the attention weights for unvisited nodes.

$$MMHA\ (Q, K, V, \zeta) = Softmax\left(\frac{|\ QK^T}{|\ \sqrt{d_k}} \odot \zeta\right) |\ V \tag{14}$$

$$\widetilde{h}_t = LN\left(\hat{h}_t + MMHA^{L+1}\left(\hat{h}_t, E^L, E^L, \zeta_t\right)\right) \tag{15}$$

The third layer in the decoder is the Pointwise Feedforward Neural Network Layer. The algorithm implemented in this layer is identical to that in the encoder's pointwise feed-forward neural network layer, comprising two linear layers with a ReLU activation function, followed by a residual connection and layer normalization, as shown in Equations (3)–(16).

$$\overline{h}_t = LN\left(\widetilde{h}_t + FF^{\ L+1}\left(\widetilde{h}_t\right)\right) \tag{16}$$

The final layer in the decoder is the Pointer Layer, responsible for calculating a probability distribution to determine the next unvisited node for selection. This process utilizes a single-head attention mechanism, where $\overline{h}_t$ serves as the query vector, E^L serves as both the key and value vectors, and ζ_t prevents the model from selecting previously visited nodes. The calculation is shown in Equation (17), where the SoftMax function converts raw scores into a probability distribution, ensuring that the sum of probabilities equals 1.

$$p_t = Softmax\left(tanh\left(\frac{|\ QK^T}{|\ \sqrt{d_k}}\right) \odot \zeta\right) \tag{17}$$

During training, the decoder treats p_t as a categorical distribution, sampling node indices and selecting the one with the highest probability. During inference, the model directly selects the node index with the highest probability. This process is iterated n times, generating a sequence of node indices $\pi = \{\pi_1, \ldots, \pi_n\}$, representing the final output of the model, as shown in Figure 2. However, this represents a static process, where the set of nodes remains fixed throughout the entire computation.

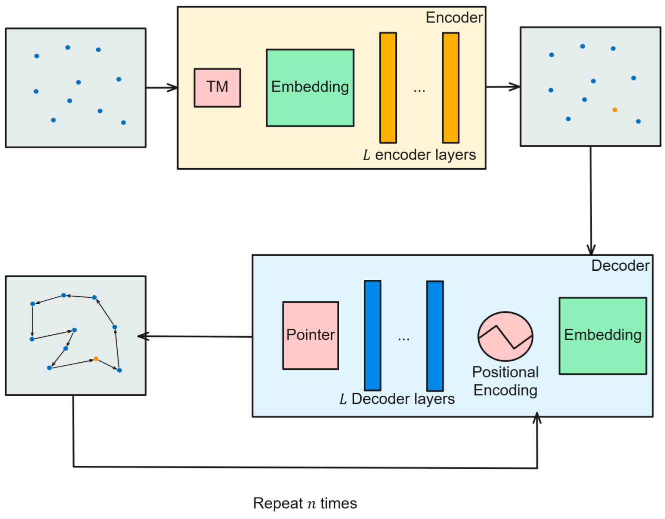


Figure 2. Inference process of the transformer model in static environment.

To accommodate dynamic environments [22], where conditions may change as the drone patrols suspicious vessels at sea, an adaptive approach is necessary. This paper focuses on utilizing drones to patrol these vessels, considering situations where new vessels may require inspection, or previously identified vessels may be removed from the list as the drone reaches each point. As shown in Figure 3, the generative model determines whether to add or remove nodes during the inference of the next node using a uniform probability distribution.

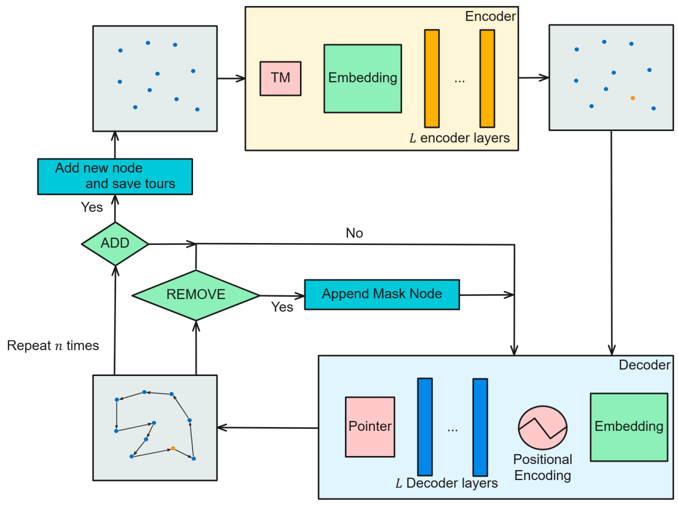


Figure 3. Inference process of the transformer model in dynamic environment.

If a node is added, random coordinates for the x-axis and y-axis are generated and appended to the node matrix, and the time matrix is subsequently recalculated. The updated matrix is then fed back into the encoder, where the newly calculated values, along with the previously visited nodes $H_t = \{h_{t-m}, \ldots, h_t\}$, serve as key and value vectors in the Multi-Head Partial Self-Attention Layer. The query vector is the input at the current time step t of the decoder, denoted as h_t. The output is then passed through the Masked Multi-Head Attention Layer, where the previously visited nodes function as the mask. This process repeats until additional nodes are added or all nodes have been inspected.

To remove a node, a random node is selected from the set of unvisited nodes, and the mask within the Masked Multi-Head Attention Layer of the decoder is adjusted to disregard this node, thereby preventing the model from considering it in subsequent computations. This approach obviates the necessity to recalculate the output of the encoder, thus reducing computational overhead and processing time.

2.2. Training Methods and Procedures

The training method employed in this paper draws inspiration from the approach outlined in [12], which utilizes reinforcement learning [23] based on a Greedy Rollout Baseline. Given the necessity to accommodate dynamic environments, a time matrix serves as the input to the model. However, directly implementing this method can result in suboptimal convergence and protracted training durations. The solution presented in [12] involves using a pre-trained model to train two separate models within the decoder: a node model and a traffic dynamics model. The outputs are then combined and fed into the decoder to mitigate issues of poor convergence and lengthy training times.

In contrast, this paper adopts a hybrid training [24,25] approach to address the convergence and training challenges, employing a single model for both training and inference. As shown in Figure 4, the training process begins with supervised learning. The dataset consists of randomly generated sets of nodes, and after applying the simulated annealing algorithm three times, the corresponding shortest paths are obtained. These randomly generated sets of nodes are then fed into the model, which produces its own predicted shortest path. By comparing the predicted path with the actual shortest path, the loss value is calculated. When the loss value is greater than or equal to 0.3, the training continues using supervised learning.

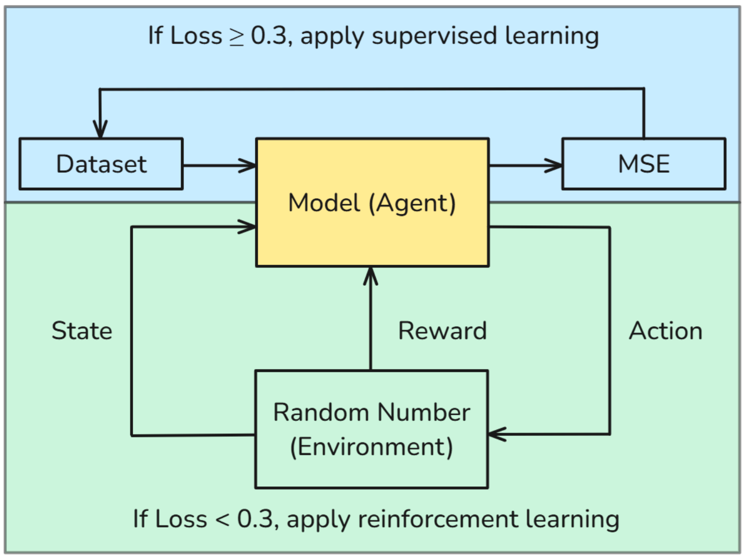

If Loss ≥ 0.3, apply supervised learning
Dataset
Model (Agent)
MSE
State
Reward
Action
Random Number (Environment)
If Loss < 0.3, apply reinforcement learning

Figure 4. Hybrid training flow chart.

However, if the loss value falls below 0.3, the training approach shifts to reinforcement learning. In this phase, random nodes are generated dynamically and treated as the model's state. Upon receiving these random nodes, the model adjusts the order of the nodes, which constitutes the action. Finally, the current path length is used as the reward, allowing the model to further optimize based on this feedback.

This method expedites inference speed and reduces the model's size, facilitating its deployment on edge devices. The hybrid training process is divided into two distinct phases: the first phase involves supervised learning, and the second phase employs reinforcement learning [26] based on the Greedy Rollout Baseline [27,28]. After a predetermined duration in the first phase, the training transitions to the second phase.

Supervised learning [29] is crucial because training the entire model by solely using reinforcement learning would necessitate a considerable amount of time for the model to discover patterns, and it may struggle to do so effectively. Therefore, this paper initiates training with supervised learning, enabling the model to converge rapidly by providing the correct answers initially.

The supervised learning dataset utilized in this paper comprises 100,000 nodes generated from a uniform random distribution, with the optimal solution obtained using the Simulated Annealing algorithm. The dataset is then partitioned into three parts: 70% for the training set, 20% for the validation set, and 10% for the test set. During this phase, the loss function is calculated using the Mean Squared Error (*MSE*) [30], as shown in Equation (18). Here, $\hat{y}_i$ represents the optimal flight time from the dataset, y_i is the predicted result, and n is the dataset size.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{18}$$

In the second phase of the training process, once the *MSE* value falls below a predetermined threshold, the training transitions to reinforcement learning. The model input comprises randomly generated node coordinates. Based on the model constructed in Section 2.1, inference results are obtained, yielding a probability distribution $p_\theta(\pi|s)$, where π represents the generated solution. To train the model, the loss function is defined as $\mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(\pi|s)}[T(\pi)]$, where $T(\pi)$ denotes the cost of flight time after completing a full circuit back to the starting point. The expectation $T(\pi)$ within the probability distribution $p_\theta(\pi|s)$ is minimized using Gradient Descent. The gradient estimator in reinforcement learning is employed here to minimize the loss function $\mathcal{L}(\theta|s)$, incorporating a baseline [31] $b(s)$, as shown in Equation (19):

$$\nabla\mathcal{L}(\theta|s) = E_{p_\theta(\pi|s)}[(T(\pi) - b(s))\nabla log \mid p_\theta(\pi|s)] \tag{19}$$

The baseline $b(s)$ acts to reduce the variance in gradients, thereby improving the learning speed. In this paper, $b(s)$ is defined as the cost of the solution executed using the Greedy Rollout strategy from the current best model. Subtracting $T(\pi)$ from $b(s)$ stabilizes the training. The iteration of $b(s)$ refers to training the model over one epoch. At the conclusion of each epoch, the average flight times of both the training model and the base-line model are calculated. If the average flight time of the training model is shorter, the weights of the training model are copied to the baseline model for the next epoch, encouraging the model to generate node sequences with shorter flight times.

3. Experimental Results

The experimental results are presented in two main sections. The first section focuses on static comparisons, examining the performance of the generative Transformer-based model and the metaheuristic-based Simulated Annealing algorithm, in both a standard static environment and a static environment in which flight time is taken into account. The second section delves into dynamic scenarios. Due to the inherent limitation of the Simulated Annealing algorithm which precludes it from incorporating information from previously visited nodes, this section underscores the superior capability of the generative Transformer-based model to integrate past information and maintain accuracy.

3.1. Results for Static Environments

In the static environment, we randomly generated sets of 10, 20, and 50 nodes. The flight times obtained after running each scenario 100 times were then averaged and compared. As shown in Table 1, in terms of inference time for route planning, the Transformer model significantly outperformed the metaheuristic-based Simulated Annealing algorithm. The performance gap (GAP) between the Simulated Annealing algorithm and the Transformer model reached −97.68% in TSP 10, −98.56% in TSP 20, and −98.74% in TSP 50, clearly demonstrating the computational efficiency of the Transformer model. The calculation formula for the GAP is shown in Equation (20). As evident in the equation, the Simulated Annealing algorithm is used as the denominator. Consequently, a negative percentage signifies superior performance compared to the Simulated Annealing algorithm.

$$\text{GAP }\% = \frac{\text{Transformer } - \text{ SA}}{\text{SA}} \times 100\% \tag{20}$$

Table 1. Inference time in a static environment.

Problem	**SA**			**Transformer**			**GAP**
	Max	**Min**	**Avg**	**Max**	**Min**	**Avg**	
10 nodes	1.51100	0.09700	0.51654	0.01300	0.01100	0.01199	−97.67887%
20 nodes	5.25200	0.27300	1.38302	0.05800	0.01900	0.01990	−98.56116%
50 nodes	8.20300	6.49017	4.04517	0.08900	0.04900	0.05098	−98.73975%

However, as shown in Table 2, when comparing flight times, the Simulated Annealing algorithm exhibited an advantage in smaller-scale TSP problems. Due to the reduced number of nodes, it was easier for the algorithm to escape local optima, resulting in shorter flight times than those achieved by the Transformer model in the TSP 10 and TSP 20 scenarios. However, as the number of nodes increased, the discrepancy between the flight times of the two methods began to diminish. By TSP 50, it became evident that the flight time of the Simulated Annealing algorithm surpassed that of the Transformer model, with a gap of −1.59%.

Table 2. Flight time in a static environment.

Problem	SA			Transformer			GAP
	Max	Min	Avg	Max	Min	Avg	
10 nodes	4.68105,	2.68059	3.56941	4.79446	2.69055	3.59068	0.59594%
20 nodes	6.34680	3.65797	4.88704	5.83352	3.65797	4.90350	0.33665%
50 nodes	8.44219	6.49017	7.51615	8.43149	6.16260	7.39642	−1.59285%

3.2. Results for Static Environments Considering Dynamic Factors

In this paper, we introduce an algorithm that incorporates wind speed and direction into the time matrix (TM) utilized in our experiments. To assess the ability of these methods to account for the impact of wind in a static environment, this section compares three distinct approaches: the standard Simulated Annealing algorithm, the Simulated Annealing algorithm augmented with the time matrix, and the Transformer model with the time matrix. These approaches are compared across three different TSP scenarios to evaluate the effect of incorporating the time matrix.

As shown in Table 3, when utilizing the Simulated Annealing algorithm, considering wind speed and direction yields superior results compared to not considering them, with improvements of 0.02369%, 0.02484%, and 0.07275% in TSP 10, TSP 20, and TSP 50, respectively.

Table 3. Comparison of flight time for simulated annealing algorithm with and without time matrix.

Problem	SA	SA with TM	GAP
10 nodes	3.66641	3.66554	0.02369%
20 nodes	4.92994	4.91773	0.02484%
50 nodes	7.54127	7.53578	0.07275%

3.3. Results for Dynamic Environment Considering Node Addition and Removal

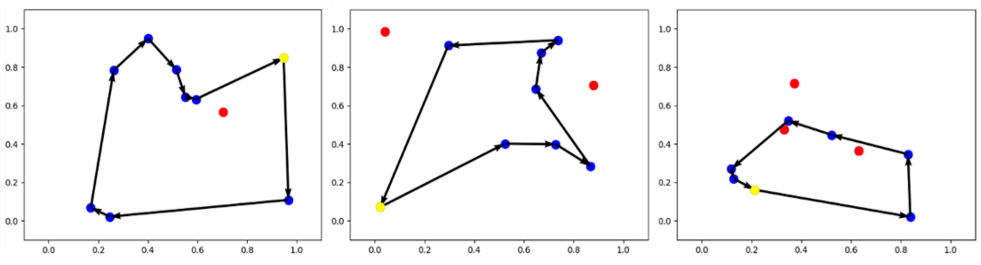
This subsection is structured into four parts: the presentation of node removal, the presentation of node addition, the presentation of concurrent node addition and removal, and the comparison of path computation times within a dynamic environment.

3.3.1. Node Removal Scenarios

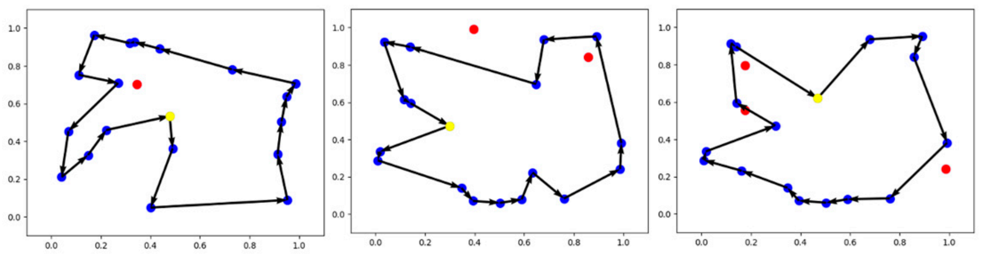
In the node removal section, as illustrated in Figure 5, the yellow point represents the starting point, the blue points represent the original nodes, and the red points indicate the nodes that were removed during the flight. In each path-planning instance, a removal probability is uniformly distributed across each time step t. A node is randomly selected for removal when the removal probability exceeds 0.8, provided it has not yet been visited. Figure 5 is divided into three parts, (a), (b), and (c), corresponding to the TSP 10, TSP 20, and TSP 50 environments, respectively. Within each environment, three scenarios are presented, showcasing the removal of one, two, and three nodes, respectively.

3.3.2. Node Addition Scenarios

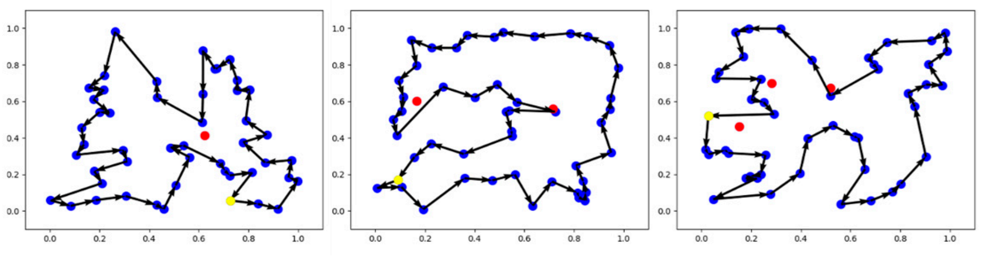
In the node addition process, at each time step t, a random value is generated. If this value exceeds 0.8, a new coordinate is randomly generated and appended to the original set of coordinates. This randomness introduces an element of unpredictability, mimicking real-world scenarios in which new tasks or targets can appear unexpectedly. Subsequently, the encoder values are recomputed, and after processing through the decoder, the next node is selected. This ensures that the model continuously adapts to the new information, maintaining an optimal path selection process. The model operates with a predefined maximum number of nodes for each TSP scenario. For instance, in TSP 10, the maximum is 10 nodes, and so forth. Therefore, the initial setup commences with only seven nodes to ensure that the addition of nodes does not violate the maximum node limit.



(**a**)



(**b**)



(**c**)

Figure 5. Dynamic nodes removal with varying numbers of nodes: (**a**) TSP 10, (**b**) TSP 20, and (**c**) TSP 50.

Figure 6 provides a visual representation. Here, the yellow node denotes the starting point, the blue nodes represent the initial coordinates, and the green nodes signify the added coordinates. The dynamic process can be observed at different time steps (t). It can be observed that at $t = 2$, a green node is generated, indicating a newly added node. At $t = 5$, a second green node is added, and at $t = 8$, a third green node is added. Finally, at $t = 10$, the path encompasses all nodes, resulting in the final route. It is noteworthy that since the third node was added later in the process, the route may not appear visually optimal on the graph. However, from the perspective of dynamic planning, this represents the shortest path at that specific point in time.

This demonstrates the model's ability to adaptively integrate new nodes while maintaining efficient path planning, which is essential for applications such as real-time surveillance or dynamic task allocation.

3.3.3. Scenarios Considering Both Node Addition and Removal

By integrating the functionalities detailed in Sections 3.3.1 and 3.3.2, we perform both node addition and removal concurrently. In contrast to the previous node addition process, the initial number of nodes is determined by generating a random value, which dictates the starting node count. Based on this initial count, the processes of adding and removing nodes are carried out.

In Figure 7, the yellow node represents the starting point, the blue nodes are the initial coordinates, the green nodes are the added coordinates, and the red nodes are the removed nodes. The figure is divided into three parts, (a), (b), and (c), corresponding to the TSP 10, TSP 20, and TSP 50 environments, respectively. Each environment illustrates the dynamic node addition and removal scenarios that occur when the random addition or removal value exceeds 0.8.

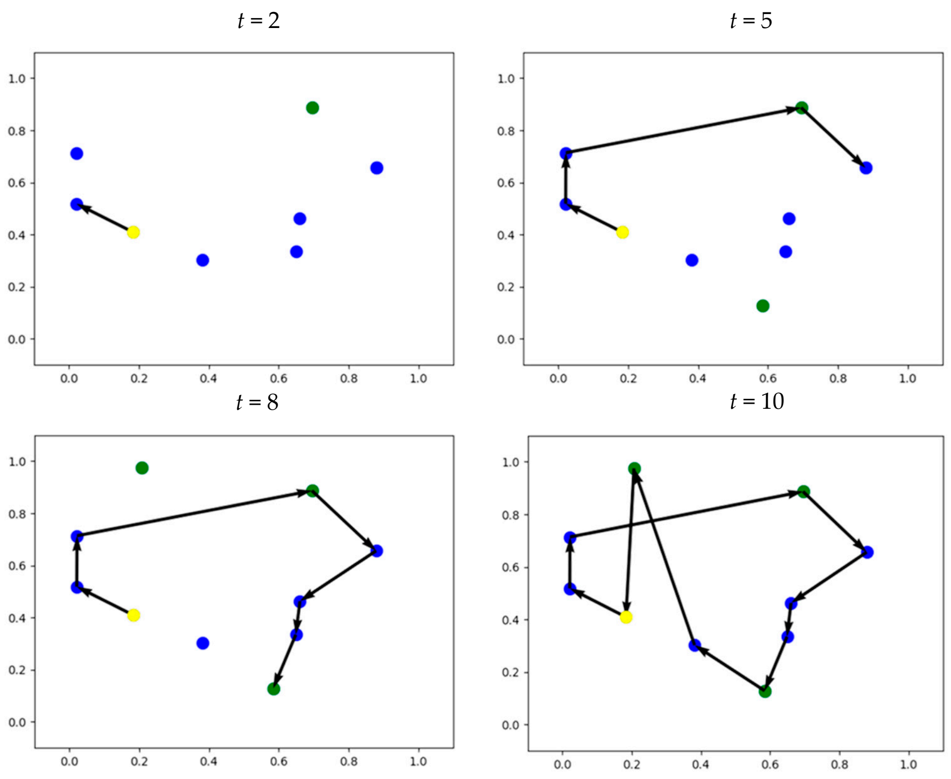


Figure 6. Node addition process.

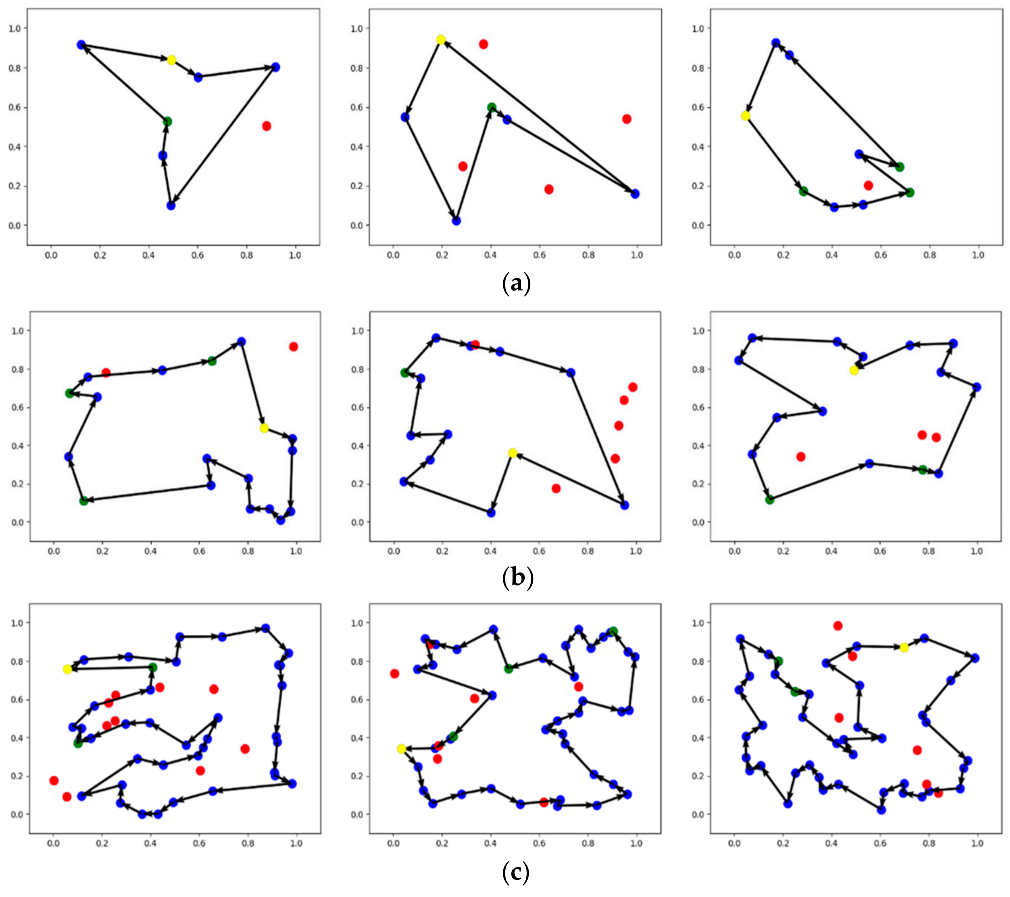


Figure 7. Dynamic node addition and removal with different node counts: (**a**) TSP 10, (**b**) TSP 20, and (**c**) TSP 50.

3.3.4. Inference Times for Scenarios Considering Node Addition and Removal

Due to the inherent characteristics of metaheuristic-based algorithms such as Simulated Annealing, direct comparison is not feasible in this context. Consequently, we focus exclusively on the inference time of the generative Transformer-based model in the dynamic environment in which nodes are added and removed. Table 4 presents the inference time comparison for dynamic node addition and removal in TSP 50, specifically contrasting the inference time for scenarios where three and ten nodes were removed, as well as where three and ten nodes were added.

Table 4. Comparison of inference times for dynamic node addition and removal in tsp 50.

Nodes	**Type**	**Inference Time**
3	Removal	0.04521
	Addition	0.04796
10	Removal	0.04205
	Addition	0.05308

3.4. Results for Dynamic Environment Considering Wind Velocity and Direction Variations

Unlike scenarios with dynamic nodes, the number of nodes remains constant when considering wind speed and direction, enabling a direct comparison of the impact of dynamic wind conditions. In this experiment, the direction of the wind changes every five nodes. The Simulated Annealing algorithm calculates the path based on the current wind speed and direction, as detailed in Section 3.1 (pertaining to static environments). After each wind direction change, the inter-node travel times are recorded, and the total flight time for the path generated by Simulated Annealing is computed. This time is then compared to the flight time of the Transformer model, which inherently accounts for the dynamic changes in wind speed and direction.

Table 5 presents the experimental results. It is evident that for TSP 10, the metaheuristic Simulated Annealing algorithm holds a marginal advantage, with a difference of 0.00106%. However, as the number of nodes increases, specifically in TSP 20, the flight time difference shifts to −0.28220%, indicating that the Transformer model, designed for dynamic environments, surpasses the Simulated Annealing algorithm. By the time the experiment reaches TSP 50, the difference has grown to −3.97117%, with the Transformer model maintaining its lead.

Table 5. Comparison of wind speed and direction considerations in a dynamic environment.

Problem	**SA**			**Transformer**			**GAP**
	Max	**Min**	**Avg**	**Max**	**Min**	**Avg**	
10 nodes	5.59516	3.33512	4.56291	5.60852	3.33963	4.56779	0.00106%
20 nodes	7.50982	5.18627	6.39710	7.55819	5.18627	6.37905	−0.28220%
50 nodes	10.8184	8.30015	9.59209	10.5211	8.20854	9.21117	−3.97117%

4. Discussions

In static environments, the metaheuristic Simulated Annealing algorithm exhibits remarkable performance and efficiency when applied to small-scale Travelling Salesman Problems (TSPs). However, its computational demands and accuracy notably diminish as the problem scale expands. In contrast, our proposed model, while initially underperforming compared to Simulated Annealing in small-scale TSPs (e.g., a 0.59594% accuracy gap in TSP 10), demonstrates a progressive improvement in accuracy as the number of nodes increases. This trend culminates in a significant outperformance of Simulated Annealing in larger TSPs (e.g., a 1.59285% accuracy advantage in TSP 50).

Regarding inference time, the Transformer model, grounded in generative neural networks, showcases superior capabilities in path planning, surpassing traditional methods across both small- and large-scale problems. Notably, in TSP 10, TSP 20, and TSP 50, the Transformer model's inference speed is, respectively, 64.21317%, 80.81983%, and 89.99539% faster than the Simulated Annealing algorithm.

From the experimental results mentioned above, in small-scale Traveling Salesman Problem (TSP) scenarios, the Transformer model performs less effectively than Simulated Annealing (SA) because the Transformer requires a large amount of training data to learn effective patterns. With fewer nodes in small-scale problems, the model is more prone to overfitting, and its complex structure and high computational resource demands make it unnecessarily cumbersome for smaller problems. In contrast, SA is a lightweight heuristic algorithm that can efficiently solve small-scale problems and avoid becoming stuck in local optima.

However, as the problem size increases, the advantages of the Transformer become more evident. Its highly parallel architecture and self-attention mechanism enables it to process large amounts of data and capture global information effectively. Additionally, by leveraging deep learning, the Transformer can learn the underlying structure of the problem, allowing it to quickly find high-quality solutions in large-scale TSPs. On the other hand, SA's step-by-step random search becomes significantly less efficient and less capable of producing high-quality solutions in larger problems. Therefore, the Transformer is better suited for solving large-scale problems, while SA tends to perform better in small-scale scenarios.

To factor in the influence of wind speed and direction on the drone's flight path, a time matrix was incorporated, adjusting the travel time cost for each node and enabling the model to adapt its trajectory based on prevailing wind conditions. Experimental results confirm that employing a time matrix effectively minimizes the predicted flight time and empowers the model to comprehensively account for diverse factors within dynamic environments.

In dynamic environments, the states and parameters of a system are continuously evolving, which significantly increases the complexity and difficulty of computations. For example, in path-planning problems, the addition or removal of nodes and changes in environmental conditions (such as wind speed and direction) can invalidate previously computed optimal paths, necessitating recalculations. This leads to more complex and time-consuming computations, especially in scenarios with multiple nodes, in which the state space and possible solution grow exponentially.

Furthermore, dynamic environments introduce uncertainty, requiring models to handle current conditions and possess predictive capabilities to account for future changes. Traditional static algorithms like Simulated Annealing struggle in dynamic situations because their step-by-step optimization process requires recalculating large portions of the solution whenever a change occurs, resulting in slower inference speeds.

In contrast, generative models based on the Transformer architecture exhibit greater adaptability. The multi-head self-attention mechanism enables the model to process relationships between multiple nodes simultaneously. Masking techniques allow the model to quickly exclude or add nodes without completely recomputing the encoding layer. When dealing with dynamic changes, Transformers can adjust their internal weights more flexibly, enhancing inference efficiency. These characteristics enable Transformers to better accommodate variations in dynamic environments, giving them a significant advantage over traditional algorithms.

In experiments involving the dynamic addition or removal of nodes, the computational limitations of Simulated Annealing preclude a direct comparison with our model. For node removal, the Masked Multi-Head Self-Attention Layer was leveraged to directly mask the nodes slated for removal, obviating the need to recompute the encoder layer and thereby enhancing the model's inference efficiency. Conversely, node addition necessitates recalculation of the encoder layer, resulting in slower inference speeds, as evidenced in Table 4.

In experiments incorporating wind speed and direction, while Simulated Annealing maintains a performance edge in TSP 10, this advantage is markedly reduced. In TSP 20, the superiority of Simulated Annealing observed in static environments is no longer evident, with the Transformer model achieving a 0.28220% flight time advantage in dynamic environments. This performance gap widens further to 3.97117% in TSP 50. These findings underscore that in dynamic environments, the Transformer architecture underpinning the generative model substantially outperforms traditional algorithms.

5. Conclusions

In this manuscript, we have compared the performance of our Transformer-based generative model with that of the Simulated Annealing algorithm across various static and dynamic TSP scenarios, focusing on the analysis of efficiency and inference time. The experimental results demonstrate that in static environments and small-scale TSPs, the traditional SA algorithm exhibits superior efficiency compared to the proposed Transformer model, but at the cost of longer inference times. In contrast, for static and large-scale TSPs, the Transformer model outperforms the traditional algorithm in terms of both efficiency and inference time. Furthermore, we have modified the inference process to enable the Transformer model to adapt to dynamic environments, including the dynamic addition and removal of nodes and the incorporation of varying wind velocity and direction. The experimental results show that the model can account for and integrate past information to optimally infer the succeeding path in the dynamic environments, which cannot be easily achieved by the traditional SA algorithms.

Finally, it is important to note that the current model assigns equal weights to all nodes and does not account for their distinct levels of importance. Particularly, in scenarios where prioritizing specific nodes for patrol is necessary, the proposed model falls short. Therefore, future research could focus on addressing this limitation by enabling the Transformer model to consider the importance of different nodes, perhaps by using priority-based ranking to enhance patrol efficiency.

Author Contributions: Conceptualization, J.T.; Methodology, C.-H.Y. and J.T.; Validation, C.-H.Y.; Formal analysis, Y.-T.C.; Resources, J.T.; Writing—original draft, C.-H.Y.; Writing—review & editing, J.T. and Y.-T.C.; Project administration, J.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data are contained within the article.

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

References

1. Sultana, N.; Chan, J.; Sarwar, T.; Qin, A.K. Learning to optimise general TSP instances. *Int. J. Mach. Learn. Cybern.* **2022**, *13*, 2213–2228. [CrossRef]
2. Zarpellon, G.; Jo, J.; Lodi, A.; Bengio, Y. Parameterizing branch-and-bound search trees to learn branching policies. *Proc. Aaai Conf. Artif. Intell.* **2021**, *35*, 3931–3939. [CrossRef]
3. Vaswani, A. Attention is all you need. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 1706–03762.
4. Wang, D.; Gao, N.; Liu, D.; Li, J.; Lewis, F.L. Recent progress in reinforcement learning and adaptive dynamic programming for advanced control applications. *IEEE/CAA J. Autom. Sin.* **2023**, *11*, 18–36. [CrossRef]
5. Rahman, A.; Sokkalingam, R.; Othman, M.; Biswas, K.; Abdullah, L.; Kadir, E.A. Nature-inspired metaheuristic techniques for combinatorial optimization problems: Overview and recent advances. *Mathematics* **2021**, *9*, 2633. [CrossRef]
6. Aranha, C.; Villalón, C.L.C.; Campelo, F.; Dorigo, M.; Ruiz, R.; Sevaux, M.; Sörensen, K.; Stützle, T. Metaphor-based metaheuristics, a call for action: The elephant in the room. *Swarm Intell.* **2022**, *16*, 1–6. [CrossRef]
7. Guilmeau, T.; Chouzenoux, E.; Elvira, V. Simulated annealing: A review and a new scheme. In Proceedings of the 2021 IEEE statistical signal processing workshop (SSP), Rio de Janeiro, Brazil, 11–14 July 2021.
8. Rajwar, K.; Deep, K.; Das, S. An exhaustive review of the metaheuristic algorithms for search and optimization: Taxonomy, applications, and open challenges. *Artif. Intell. Rev.* **2023**, *56*, 13187–13257. [CrossRef]
9. Kool, W.; Van Hoof, H.; Welling, M. Attention, Learn to Solve Routing Problems! *arXiv* **2018**, arXiv:1803.08475.

10. 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. *Proc. AAAI Conf. Artif. Intell.* **2019**, *33*, 4731–4738.
11. Ma, Q.; Ge, S.; He, D.; Thaker, D.; Drori, I. Combinatorial Optimization by Graph Pointer Networks and Hierarchical Reinforcement Learning. *arXiv* **2019**, arXiv:1911.04936.
12. Zhang, Z.; Liu, H.; Zhou, M.; Wang, J. Solving dynamic traveling salesman problems with deep reinforcement learning. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *34*, 2119–2132. [CrossRef] [PubMed]
13. Li, Y.; Wang, P.; Zhu, X.; Chen, A.; Jiang, H.; Cai, D.; Chan, V.W.K.; Li, J. GLBench: A Comprehensive Benchmark for Graph with Large Language Models. *arXiv* **2024**, arXiv:2407.07457.
14. Jung, M.; Lee, J.; Kim, J. A lightweight CNN-transformer model for learning traveling salesman problems. *Appl. Intell.* **2024**, *54*, 7982–7993. [CrossRef]
15. Yuan, W.; Chen, J.; Chen, S.; Feng, D.; Hu, Z.; Li, P.; Zhao, W. Transformer in reinforcement learning for decision-making: A survey. *Front. Inf. Technol. Electron. Eng.* **2024**, *25*, 763–790. [CrossRef]
16. Kent, J.T. Directional distributions and the half-angle principle. In *Robust and Multivariate Statistical Methods: Festschrift in Honor of David E. Tyler*; Springer International Publishing: Cham, Switzerlands, 2022.
17. Azdy, R.A.; Darnis, F. Use of haversine formula in finding distance between temporary shelter and waste end processing sites. *J. Phys. Conf. Ser.* **2020**, *1500*, 012104. [CrossRef]
18. Zhang, Q.; Xu, Y.; Wang, X.; Yu, Z.; Deng, T. Real-time wind field estimation and pitot tube calibration using an extended Kalman filter. *Mathematics* **2021**, *9*, 646. [CrossRef]
19. Song, X.; Li, M.; Xie, W.; Mao, Y. A Reinforcement Learning-driven Iterated Greedy Algorithm for Traveling Salesman Problem. In Proceedings of the 2023 26th International Conference on Computer Supported Cooperative Work in Design (CSCWD), Rio de Janeiro, Brazil, 24–26 May 2023.
20. Lian, X.; Liu, J. Revisit batch normalization: New understanding and refinement via composition optimization. In Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics, Naha, Japan, 16–18 April 2019.
21. Xu, J.; Sun, X.; Zhang, Z.; Zhao, G.; Lin, J. Understanding and improving layer normalization. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 1911–07013.
22. De Lima Filho, G.M.; Passaro, A.; Delfino, G.M.; De Santana, L.; Monsuur, H. Time-critical maritime UAV mission planning using a neural network: An operational view. *IEEE Access* **2022**, *10*, 111749–111758. [CrossRef]
23. Hy, T.S.; Tran, C.D. Graph Attention-based Deep Reinforcement Learning for solving the Chinese Postman Problem with Load-dependent costs. *arXiv* **2023**, arXiv:2310.15516.
24. Al-Betar, M.A.; Awadallah, M.A.; Abu Doush, I.; Alomari, O.A.; Abasi, A.K.; Makhadmeh, S.N.; Alyasseri, Z.A.A. Boosting the training of neural networks through hybrid metaheuristics. *Clust. Comput.* **2023**, *26*, 1821–1843. [CrossRef]
25. Ansari, A.; Ahmad, I.S.; Abu Bakar, A.; Yaakub, M.R. A hybrid metaheuristic method in training artificial neural network for bankruptcy prediction. *IEEE Access* **2020**, *8*, 176640–176650. [CrossRef]
26. Yan, C.; Xiang, X.; Wang, C. Towards real-time path planning through deep reinforcement learning for a UAV in dynamic environments. *J. Intell. Robot. Syst.* **2020**, *98*, 297–309. [CrossRef]
27. Lee, D.H.; Ahn, J. Multi-start team orienteering problem for UAS mission re-planning with data-efficient deep reinforcement learning. *Appl. Intell.* **2024**, *54*, 4467–4489. [CrossRef]
28. Dong, S.; Wang, P.; Abbas, K. A survey on deep learning and its applications. *Comput. Sci. Rev.* **2021**, *40*, 100379. [CrossRef]
29. Bao, J.; Yang, Y.; Wang, Y.; Yang, X.; Du, Z. Path Planning for Cellular-connected UAV using Heuristic Algorithm and Reinforcement Learning. In Proceedings of the 2023 25th International Conference on Advanced Communication Technology (ICACT), Pyeongchang-gun, Republic of Korea, 19–22 February 2023.
30. Hodson, T.O. Root mean square error (RMSE) or mean absolute error (MAE): When to use them or not. *Geosci. Model Dev. Discuss.* **2022**, *15*, 5481–5487. [CrossRef]
31. Ladosz, P.; Weng, L.; Kim, M.; Oh, H. Exploration in deep reinforcement learning: A survey. *Inf. Fusion* **2022**, *85*, 1–22. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.