Dynamic Graph Combinatorial Optimization with Multi-Attention Deep Reinforcement Learning

Udesh Gunarathna
pgunarathna@student.unimelb.edu.au
The University of Melbourne
Australia

Renata Borovica-Gajic
renata.borovica@unimelb.edu.au
The University of Melbourne
Australia

Shanika Karunasekera
karus@unimelb.edu.au
The University of Melbourne
Australia

Egemen Tanin
etanin@unimelb.edu.au
The University of Melbourne
Australia

ABSTRACT

Graph combinatorial optimization (CO) is a widely studied problem with use-cases stemming from many fields. Typically, in real-world applications, the features of a graph tend to change over time (e.g. traffic congestion, or travel time), thus, finding a solution to the *dynamic* graph CO problem is critical. In recent years, using deep learning techniques to find heuristic solutions for NP-hard CO problems has gained much interest as these learned heuristics can find near-optimal solutions efficiently. However, most of the existing methods for learning heuristics focus on *static* CO problems. The dynamic nature makes NP-hard CO problems much more challenging to learn, and the existing methods fail to find reasonable solutions. We propose a novel architecture named Graph Temporal Attention with Reinforcement Learning (GTA-RL) to learn heuristic solutions for dynamic versions of graph CO problems. We then extend our architecture to learn heuristics for the real-time version of CO problems where all input features of a problem are not *known a priori*, but rather learned in real-time. A detailed experimental evaluation against several state-of-the-art learning-based algorithms and optimal solvers demonstrates the efficiency and effectiveness of our approach.

CCS CONCEPTS

• **Information systems** → **Spatial-temporal systems**; • **Theory of computation** → **Reinforcement learning**; • **Mathematics of computing** → **Combinatorial optimization**.

KEYWORDS

spatial-temporal data, deep reinforcement learning, neural networks, combinatorial optimization

SIGSPATIAL '22, November 1–4, 2022, Seattle, WA, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/3557915.3560956

ACM Reference Format:
Udesh Gunarathna, Renata Borovica-Gajic, Shanika Karunasekera, and Egemen Tanin. 2022. Dynamic Graph Combinatorial Optimization with Multi-Attention Deep Reinforcement Learning. In *The 30th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '22), November 1–4, 2022, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3557915.3560956

1 INTRODUCTION

Graph combinatorial optimization (CO) problems have been studied by computer scientists for decades as these problems are relevant to many fields, including transportation, computer networks and social networks. Most of the existing approaches focus on the static version of the graph CO. Unfortunately, many real-world applications require the dynamic version to be addressed where the graph input attributes change over time [4, 30].

Let us consider a motivating example for a dynamic graph CO problem using a ride sharing scenario, where multiple customers need to be picked up from several locations in a road network, and each customer needs to be dropped off at different locations. The objective (i.e., the optimization goal) is to find an order to pick up and drop off customers providing the fastest possible travel time to all. This is an NP-hard optimization problem, and can be reduced to a minimum steiner tree [13]. In real-world road networks, the travel time between pickup and drop-off locations changes over time due to congestion levels or other external factors such as accidents. Thus, while optimizing the order of visiting customers, we need to consider the changes in the travel time between different locations; the road network graph is dynamic. Typically, traffic-related research [6] approximates this case and assumes that even though the travel times between locations are not static, changes in travel times are known beforehand (typically by relying on historical data or through traffic predictions [1, 31]). This common optimization then becomes a dynamic graph CO problem. If we assume that the future travel time estimates are not available and the future travel times between locations are only available after we reach that time, then the problem becomes real-time graph CO. A similar scenario is observed in telecommunication networks where the goal is to allocate bandwidths to communication links to maximize the overall network utilization, and traffic in each link changes over time. With these real-world examples, it is evident that the dynamicity of graph CO problems needs to be considered to make related research applicable to many real-world scenarios.

Due to the NP-hard nature, even in static COs, finding an exact solution for a large graph instance is computationally expensive. On the other hand, heuristic solutions can be fast but require domain knowledge of the problem and a significant amount of manual-engineering to design them [7]. Finding a heuristic in a dynamic setting can be even more challenging due to a large number of possible combinations[1]. Furthermore, the heuristic needs to consider the temporal dimension as well, i.e., in the dynamic TSP, a solution not only needs to find an order of nodes to visit but also needs to consider when to visit these nodes. Since the cost between two nodes can change over time in dynamic TSP, the solution should choose time steps when the cost between nodes is minimal (see visualization in Appendix A). In this work we seek a generalized method to find fast and efficient heuristics for dynamic graph COs without relying on hand-crafted methods.

Recent advancements in deep Reinforcement Learning (RL) have led to finding efficient heuristics for CO without hand-crafted engineering [2, 7, 8, 15, 18]. All these approaches model CO problem as a complete or incomplete graph [21] and formulate the decision-making process as a successive addition of nodes to the solution. By doing so, the learned heuristics can achieve near-optimal performance. All of these works however focus only on static COs [7, 8, 15], failing to find reasonable solutions for dynamic CO problems, as demonstrated in experiments.

To identify the shortcomings of the mentioned approaches for dynamic COs, existing methods can be categorized into two main types: (1) Message passing neural networks with value-based RL [2, 7], and (2) Attention-based/Recurrent Neural Networks (RNN) with policy-gradient RL [8, 9, 15, 18, 20, 27]. The former methods use a message-passing Neural Network to iteratively encode the graph information and use Q-learning [28] for the decision making. The latter methods use an attention mechanism [25] or an RNN to encode the graph instance as a sequence of nodes and use a decoder to make the decisions iteratively. As recent research suggests [15, 18], the latter approaches achieve better performance for COs largely due to the fact that the policy-gradient RL algorithms perform better when the action space is large/variable [24].

Despite the success of attention-based policy gradient methods in static COs, the reason for them not performing so well in dynamic COs is three-fold. (1) The graph/input features are embedded before making decisions/actions. Thus, changes in graph features due to the decision-making process are not captured. (2) The attention mechanism deployed is only able to encode graph node features in a single time step. In a dynamic setting, each node contains different features at different time steps and such temporal changes are not captured. (3) The decoding mechanism used by the attention-based methods focuses on the entire embedded space given by the encoder at every decision-making time step. However, in a dynamic setting, there are different encoded outputs at every time step. Thus, paying attention to the entire embedded space tends to distract the decision-making process.

We propose a novel encoder-decoder architecture named *Graph Temporal Attention (GTA)* trained with modified *Reinforcement Learning (RL)* to address the aforementioned issues and we denote *GTA-RL* as the combined overall architecture. First, we propose a multi-dimensional attention mechanism at the encoder to embed both temporal and graph (spatial) features simultaneously. Then, a fusion mechanism is introduced to learn the inter-relationship between temporal and spatial embeddings which allows to encode spatial information in multiple time steps. Such encoder layers are cascaded together. Once the set of encoder layers outputs the spatio-temporal representation of the dynamic CO, a novel decoder named *temporal pointing decoder* is used to dynamically pay attention to the spatio-temporal representation for the decision-making stage. By paying attention to specific parts of the spatio-temporal representation, the temporal pointing decoder avoids distracting the decision-making process as well as captures the changes in graph features due to the decision-making process. Finally, GTA-RL is trained through a modified policy-gradient RL algorithm. Note that, this architecture targets the scenarios where all the dynamic changes are estimated beforehand as in the standard dynamic CO [12, 30]. We introduce an iterative method by partially embedding the encoder inside the decoder to tackle the problem where the dynamic changes are not *known a priori*, thus, making GTA-RL applicable for a variety of real-time applications.

The contributions from this work are four-fold: 1. We propose a novel deep learning architecture, GTA-RL, to tackle the dynamic graph CO problems. 2. We introduce an encoder and decoder that are capable of preserving temporal information in sequential decision making. 3. We propose an iterative architecture that can handle real-time information of COs. 4. Our extensive experimental results with well known spatial problems such as TSP and Vehicle Routing Problem (VRP), demonstrate that GTA-RL achieves state-of-the-art over comparable learning-based algorithms.

2 RELATED WORK

Due to the success of Deep Reinforcement Learning (DRL) algorithms in sequential decision making and control problems, many domains including atari games, traffic control [10, 11], and robot navigation [32] have successfully leveraged it. With the development of neural networks such as LSTM, RNN, message-passing networks and attention networks which allowed working with graph structured data, in recent years, DRL has been applied to many graph CO problems to learn heuristics that require no human input [2, 7, 8, 15, 18] (refer to Appendix B.1 for details on how these neural networks embed graph data). These works have achieved near-optimal performance outperforming hand-crafted heuristics in many CO problems. These existing methods formulate the combinatorial graph problems as a successive addition of graph nodes (actions) to the solution. The existing methods can be broadly categorized into two types, based on the reinforcement algorithm adopted in the solutions. (1) Using Value-based methods such as Deep Q-learning [17]. (2) Using Policy-gradient algorithms such as REINFORCE and Actor-Critic algorithms [24].

Value-based method for graph CO was first proposed by Dai *et al.*[7]. The proposed architecture uses *structured2vec* to embed the graph information and uses fitted Q-learning [22] for sequential decision making. They applied the proposed solution to solve static Travelling Salesmen (TSP), Minimum Vertex Cover, and Max-Cut problems. A similar architecture is followed by Barrett *et al.*[2] using

[1]For example, Travelling Salesmen Problem with n number of nodes has $n!$ possible selections. In dynamic TSP, each node has different values at different time steps. The possible combinations would thus be $(n!)^2 = (n.n \times (n-1)(n-1)...)$.

Message Passing Neural Networks. However, in their approach, the solution is further optimized at inference time, which improves over a one-shot solution in Dai *et al.*[7].

The policy-gradient methods have demonstrated improved results over value-based based counterparts, as evident by the recent research [8, 15]. The main reason is because in graph CO, the number of nodes (the action space) can be variable from a problem instance to instance and when the action space is large, the policy-gradient algorithms tend to perform better [24]. The first policy-gradient algorithm for graph CO was proposed by Bello *et al.*[3]. The paper uses an LSTM encoder-decoder architecture named *pointer network* which was first proposed by Vinyals *et al.*[27] to solve the TSP in a supervised learning setting. However, Bello *et al.* uses policy-gradient instead of supervised learning to train the pointer network. In addition, Bello *et al.* improves the decoder by a masking scheme to mask already selected nodes. Nazari *et al.* [18] uses a different *pointer network* architecture employing an attention mechanism and RNN to solve the Vehicle Routing Problem (VRP). Their solution includes a separate dynamic encoder to embed the dynamic features of a given CO problem. Despite being able to handle dynamic changes to the problem over time, this architecture is developed mainly focusing on the internal changes due to the node selection in the decision-making process but not the changes due to external factors which are beyond the control of the decision making process (i.e. static external input). As our experiments show, GTA-RL achieves superior performance in several combinatorial problems over the architecture of Nazari *et al.* [18].

The next generation of policy-gradient methods uses pure attention architectures instead of RNNs inspired by the architecture proposed in Vaswani *et al.*[25]. Deudon *et al.* [8] uses a Multi-head attention for encoding the CO problem and a decoder with a pointing mechanism similar to Bello *et al.* but without recurrent elements. The REINFORCE algorithm [24] with a critic baseline has been used for training the network. In parallel to Deudon *et al.*, Kool *et al.* [15] proposed an improved architecture by introducing a new attention-based decoder instead of the pointing mechanism. The paper also shows that using a roll-out baseline instead of the critic baseline improves the results further for CO problems. Kool *et al.* outperforms existing methods in TSP, VRP (Vehicle Routing Problem), and variants of both TSP, VRP. Our proposed architecture GTA-RL uses an architecture akin to Kool *et al.*, and extend it with a novel encoder and decoder, which are capable of embedding temporal features of dynamic CO problems and dynamically focusing on temporal features during the decision making process. Our experimental results show that we outperform the aforementioned state-of-the-art approaches for dynamic CO problems. We also extend our architecture to real-time CO.

3 DYNAMIC GRAPH COMBINATORIAL OPTIMIZATION AS A LEARNING PROBLEM

In this section, we present a generic dynamic graph combinatorial problem formulation and discuss the approach to parameterize the problem formulation so that it can be learned.

First, we represent the dynamic graph combinatorial problem based on a graph instance G with N number of nodes. The graph features can be represented as $X = \{x_1, x_2, ..., x_N\}$, where for each node i, $x_i = \{x_{0,i}, x_{1,i}, ..., x_{T-1,i}\}$ is a vector with the dimension $\mathbb{R}^{T \times D}$. T represents the total number of time-steps and D is the number of features in one input element at a given time step. In a transportation network, features of a node can be xy coordinates, or the number of customers/goods to pick up in a TSP scenario. We denote $x_{t,i} \in \mathbb{R}^D$ as the input features of node i at time t. The cost of selecting node j at time step t, after selecting node i at the previous time step, is represented as a *cost function* $f_c : x_{t-1,i} \times x_{t,j} \mapsto \mathbb{R}$. In a complete graph, the cost function can have a value from any node i to any node j at any given time step t. In an incomplete graph, the cost function for two nodes where there is no edge can be ignored by setting a large negative value for such connections (a.k.a. masking).

Given the above details, our objective is to find an order of nodes Y with length $T_s \in (0, T]$ which satisfies the constraints of a given dynamic graph combinatorial optimization $C(Y)$, such that we minimize:

$$P_{obj}[Y|G] = \sum_{t=1}^{T_s} f_c(x(y_t, t), x(y_{t+1}, t+1)) \tag{1}$$

where $y_t \in Y$ is a selected node from G for time step t. $C(Y)$ is an evaluating function that checks whether the sequence Y satisfies all the problem constraints. For example, in dynamic TSP, the cost function, f_c, represents the distance between two nodes at a given time step t. The evaluating function, C, checks whether all nodes have been reached or not. The objective, P_{obj}, represents the total traveled distance after reaching every node.

Learning Formulation: Given a dynamic graph combinatorial problem, our objective is to find a policy, π, which will generate a sequence Y such that we minimize P_{obj} and satisfy $C(Y)$, for a given graph problem instance G. First, we can parameterize the policy as $\pi_\theta(Y|G) := Pr(Y|G)$. Next, we can factorize this as a Markov Decision Process (MDP) where input states S will be the graph instance G, plus the solution computed up to the given time step t ($S = \{G, Y_{1:t-1}\}$) and the action y_t is selected from available nodes in G. Then, we factorize $\pi_\theta(Y|G)$ using the probability chain rule as:

$$\pi_\theta(Y|G) = \Pi_{t=1}^{T} \pi_\theta(y_t|G, Y_{1:t-1}) \tag{2}$$

Our objective transforms to learn and represent the set of parameters θ in Equation 2, so that we minimize P_{obj}. In a transportation network optimization scenario, this is to find a traversing order that minimizes the total tour length/time under a dynamic setting.

Real-time Dynamic Graph Combinatorial Optimization: Here, we formulate the real-time graph combinatorial optimization problem where the input attributes of a given problem G at time step t are not available until we reach that time step. Unlike the dynamic graph combinatorial optimization, now we do not have all the information about the graph for the entire time horizon. In that case, Equation 2 can be written as; $\pi_\theta(Y|G) = \Pi_{t=1}^{T} \pi_\theta(y_t|G_{1:t-1}, Y_{1:t-1})$.

Note that, in above equation, the graph instance is $G_{1:t-1}$ which indicates that only the information up to time step t is available. This is harder to optimize than the dynamic CO version since we are acting with incomplete information. However, we later show that with a slight modification to our proposed architecture, we can solve the real-time CO to be on par with dynamic CO.

An Example of a Real-time Problem: The real-time CO typically occurs in transportation where the road network is represented as a graph. In a road network, the edge attributes can be traffic loads/average speeds of a road segment. Given that we want to reach several points in the road network, first, we need to decide on our first point to reach based on the current edge attributes. Then, we take a finite amount of time to reach that point. During that time, the edge attributes may have changed due to external traffic conditions. However, this external traffic information could not have been accessed beforehand. In this scenario, optimizing the path to visit all the points is a real-time problem.

4 GRAPH TEMPORAL ATTENTION WITH REINFORCEMENT LEARNING (GTA-RL)

To find a solution to the dynamic CO problem defined in Equation 2, we need to find a neural network architecture to represent θ parameters and a learning algorithm to train the network. We propose a novel encoder-decoder neural network architecture named *Graph Temporal Attention (GTA)* to represent θ parameters. Then, we use a loss function computed through a modified policy-based RL to train the network. The combination of these two parts is our overall architecture called *GTA-RL*. This section describes these two components and the real-time version of GTA-RL.

4.1 Graph Temporal Attention Architecture

A major limitation of the previously proposed attention based models [8, 15] was the inability to encode the time-varying features of a problem and make a decision while input features are changing. To address these limitations, we propose a novel architecture consisting of two components, namely a *temporal encoder* and a *temporally pointing decoder* for the neural combinatorial domain.

4.1.1 Temporal Encoder. The proposed novel encoder allows to learn both spatial and temporal features of CO simultaneously, which alleviates the aforementioned limitations. Figure 1 depicts the temporal encoder architecture. First, input $X \in \mathbf{R}^{T\times N\times D}$ is transformed by a linear transformation to $H_{in}^{(0)} \in \mathbf{R}^{T\times N\times D^h}$, at *fully connected layer (FC) 1*. D^h refers to the hidden dimension of *FC layer 1*. The output from *FC layer 1* is given to the temporal encoder. The temporal encoder contains a spatial attention, a temporal attention and a FC layer in parallel. $H^{(0)}$ is given as an input to these three layers and the outputs from these layers are combined through an integration layer. For simplicity, we only show the first encoding layer in Figure 1, however, several temporal encoders can be stacked by taking the output of first temporal layer $H^{(1)}$ as the input to the next, and so on. We denote $H^{(l)}$ as the input to the l^{th} temporal encoding layer and $H^{(l+1)}$ as the output. We denote $h_{i,t}^{(l)}$ as the hidden representation of node i at time t in layer l.

Even though the embedding information along a temporal axis has not been investigated in the field of neural CO domain, this idea has recently emerged in the traffic predictions and other domains [19, 23, 33] mainly for predictive tasks in one single graph. In dynamic CO, the graph topology changes from one instance to another. Therefore, these neural networks cannot be directly applied to train a dynamic CO problem. Furthermore, decoders in these networks are not designed to work with reinforcement learning algorithms.

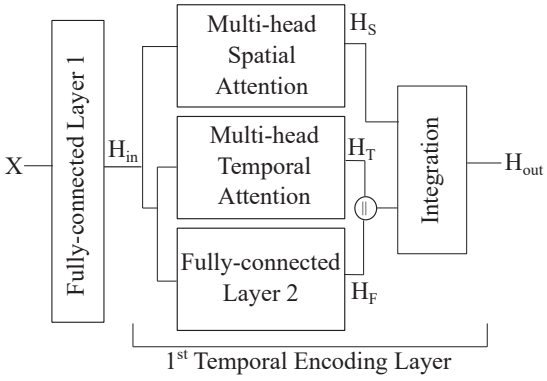


Figure 1: Temporal Encoder.

Our temporal encoder, whilst motivated by the encoder proposed in Zheng *et al.* [33] (named *STAtt Block*), differs in several key aspects. First, our temporal encoder do not use fixed graph embeddings because learning for traffic prediction only needs to handle a single road network topology, whereas in COs the input graph topology changes for every training instance. Thus, it is infeasible to learn a separate embedding for every new training instance. Second, in dynamic COs, there can be some nodes in X whose attributes do not change over time. On the contrary, in traffic prediction, attributes such as traffic load/congestion of every node change over time. In the temporal encoder, we introduce *FC layer 2* in parallel to *temporal attention layer*, to handle nodes whose attributes do not change over time.

(a) Spatial Attention Layer (SAL): In a given graph instance, there are spatial dependencies between nodes, and the SAL is used to encode those dependencies. Representing such dependencies is critical for decision making especially because the node selection fitness will depend on the surrounding nodes' information. SAL uses the multi-head self-attention mechanism (denoted as **MHA**) introduced by [25], since self-attention has been proven better at finding dependencies in variable length sequences [26]. For self-attention the same sequence is given as the input. SAL considers a list of node features at a given time step (spatial dimension): $H_{S,t}^{(l)} = \{h_{1,t}^{(l)}, h_{2,t}^{(l)}, ...h_{N,t}^{(l)}\} \in \mathbb{R}^{N\times D^h}$.

$$H_{S,t}^{(l+1)} = \mathbf{MHA}(H_{S,t}^{(l)}, H_{S,t}^{(l)}) \tag{3}$$

Here, output $H_{S,t}^{(l+1)}$ embeds the relevance of node j to node i at time t. In a non-complete graph, we set $-\infty$ to non-adjacent nodes before the softmax layer in MHA. By stacking $H_{S,t}^{(l+1)}$ over the time axis, we get the $H_S^{(l+1)}$, which represent node dependencies across all time steps for all the nodes.

$$H_S^{(l+1)} = \|_{t=1}^{T} H_{S,t}^{(l+1)} \in \mathbb{R}^{T\times N\times D^h} \tag{4}$$

(b) Temporal Attention Layer (TAL) with Fully Connected Layer: In dynamic COs, in addition to the dependency between nodes, there is a dependency between node features across different time steps for the same node. We propose a novel attention

layer fused with feed forward network to handle time dependency of dynamic COs. TAL transposes the embedding into a temporal dimension as detailed later to learn these dependencies in parallel to SAL.

Also, note that in some dynamic COs, apart from nodes that change over time, there can be some nodes that do not change over time. E.g. in *Vehicle Routing Problem* defined in Section 5.1, the features of the depot node (vehicle starting/loading node) may be fixed while customer node features change over time. Feeding such fixed nodes' information into the same TAL will reduce the effectiveness of the learning of dependencies of nodes that *are* changing over time.

To resolve this issue, we introduce a separate Fully Connected (FC) layer. First, note that there can be U number of such fixed nodes and these nodes are represented by the set $\mathbb{U}$. Then, from input $H^{(l)}$, we take $H_F^{(l)} \in \mathbb{R}^{T\times U\times D^h}$ which only contains the nodes with fixed features. In the FC layer, we apply a linear transformation for $H_F^{(l)}$ to get $H_F^{(l+1)}$.

Then, in the TAL, we select the nodes that are changing over time and represent them as $H_T^{(l)} \in \mathbb{R}^{T\times N-U\times D^h}$. Similar to the SAL, we consider a sequence, but now transposed in temporal dimension for each node i as $H_{T,i}^{(l)} = \{h_{i,1}^{(l)}, h_{i,2}^{(l)}, ...h_{i,T}^{(l)}\} \in \mathbb{R}^{T\times D^h}$, where node $i \notin \mathbb{U}$.

$$H_{T,i}^{(l+1)} = \mathbf{MHA}(H_{T,i}^{(l)}, H_{T,i}^{(l)}) \tag{5}$$

The output $H_{T,i}^{(l+1)}$ embeds the relevance of the node features at time t_b to the node features at t_a for node i. In contrast to SAL, we do not use the masking and assume all time steps are connected. By stacking $H_{T,i}^{(l+1)}$ for nodes that are changing over time, we compute $H_T^{(l+1)}$ as below.

$$H_T^{(l+1)} = \|_{i=1,i\notin\mathbb{U}}^{N} H_{T,i}^{(l+1)} \tag{6}$$

Finally, the outputs from TAL and the FC layer are concatenated to get the final temporal embedding representation; $H_{TF}^{(l+1)} = H_T^{(l+1)} \parallel H_F^{(l+1)} \in \mathbb{R}^{N\times T\times D^h}$

(c) Integration Layer: Once we learned the spatial representation ($H_S^{(l+1)}$) and the temporal representation ($H_{TF}^{(l+1)}$), we combine these two representations. First, we concatenate both $H_S^{(l+1)}$ and transposed $H_{TF}^{(l+1)}$ which results in a shape of $\mathbb{R}^{T\times N\times 2*D^h}$. Then, we use a linear transformation with a weight vector $wI \in \mathbb{R}^{D^h\times 2*D^h}$ and use a sigmoid activation layer as the output $H^{(l+1)}$. The output $H^{(l+1)}$ is given to the next layer $l+1$ in the temporal encoder. Our integration layer is much simpler compared to the layer proposed in *STAtt Block* [33], but yields similar performance. The integration layer output is designed in a way that for a given node i at time t, integration layer accounts for both the impact from surrounding nodes as well as the previous and future time variants of node i. We provide more insight on how the neural network embeds a problem instance, using TSP as an example in Appendix B.2.

4.1.2 Temporally Pointing Decoder. This subsection describes the decision making process of the decoding layer of GTA, which takes the embedded information from the encoder output as input. In the literature, a decoder with a combination of multi-head and single-head attention has traditionally yielded better results and better convergence rate over the *pointer networks* for static problems [15]. Thus, we propose an improved multi-head and single-head attention decoder to handle dynamic information.

Figure 2 shows the overall architecture of the *temporally pointing decoder*. The temporally pointing decoder works in a sequential manner with a feedback loop. First, it takes the spatio-temporal output from the last layer (L) of the encoder ($H^{(L)}$) and the decoded solution up to the current time step t. Second, the temporal pointer outputs a node embedding representation highlighting the features of the current time step. The invalid nodes for the selection will be ignored during the computation of node embeddings. This masking is problem-dependent. For example, in dynamic TSP, already visited nodes from previous time steps are invalid nodes. Third, the temporal pointer also outputs a context embedding by considering the current decoded solution and the current time step. These two outputs will be then fused together in a multi-head attention layer. Then, a single attention layer named *Log Probability Layer* is used to find the probability of each node being in the current solution. The node with the highest probability is selected as the next node and the output will be given as a feedback to the temporal pointer for the next iteration. The individual components are detailed next.

(a) Temporal Pointer: First, we describe the motivation behind the temporally pointing decoder. In the original multi-head and single-head attention architecture [15], a set of fixed attention values are computed from the encoder output. These fixed attention values are used in every decoding time step. Even though the fixed attention works in a static problem, it is unsuitable for handling dynamic information. At every decoding time step, the node features change in dynamic CO problems. Thus, initially computed fixed attention values do not reflect the node feature changes. Also, using such a fixed attention hinders the ability of the overall architecture to handle changes to the graph instance due to the new node selections. Thus, we propose to dynamically compute these attentions by taking temporal information into account.

To dynamically compute these attentions, first, let us look at the output $H^{(L)}$ of the Temporal Encoder. The output contains temporal dimensions (a shape of $\mathbb{R}^{T\times N\times D^h}$). A naive strategy to handle the temporal information is to compute a mean value for each node over the entire time horizon as; $H_{naive} = \frac{\sum_{t=1}^{T} H^{(L)}[t,:,:]}{T} \in \mathbb{R}^{N\times D^h}$, where $H^{(L)}[t,:,:]$ represents the embedding for all the nodes at time t. Another strategy is to use only a single time-step data i.e. $H^{(L)}[0,:,:]$. The results from both strategies can then be given as an input to the multi-head and single-head attention to decode node selection. As we demonstrate in Section 6.2, use of these strategies yields similar results mainly because the resulting representation suppresses the rich information about the difference in each time step.

To alleviate this problem, we propose a way to dynamically focus on the most relevant parts of the embedded output from the encoder at every time step. We slice up the encoder embedding at every decoding time step to find $H_{D,t}$ (where $H_{D,t} = H^{(L)}[t,:,:]$) and dynamically compute attention weights using multi-head attention and update the context of CO problem (described next). In this way, node representations are updated at every time step, and we

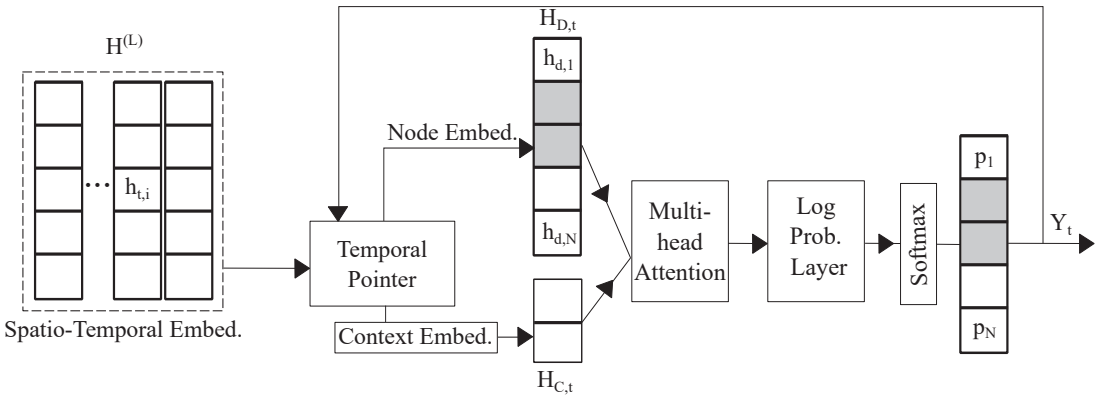


Figure 2: Decision-making process of Temporal Decoder.

compute **MHA** based on new values computed at the current time which allows to retain the current time step information without suppression. We achieve superior results using this technique as we can retain information about every time step.

(b) Context Embedding: The context embedding is used to identify the current state of a given problem (including the solution up to now) at a decoding time step. This is obviously problem-dependent. All the context embeddings are updated based on the current node representation from the encoder output. To generalize the context embedding to all the COs, a function $f_{cnxt} : \mathbb{R}^{N\times T\times N\times D^h} \mapsto \mathbb{R}^{K\times D^h+e}$ is defined. K and e are problem-dependent.

$H_{C,t} = f_{cnxt}(H(L)_t, Y_t)$

where $H_{C,t}$ is the context embedding. Section 5.1 describes the derivation of $H_{C,t}$ for different dynamic COs.

(c) Multi-head and Log Probability Layer: Once we have $H_{C,t}$ and $H_{D,t}$, we use the multi-head attention to combine the context embedding with the current graph representation.

$$H^{(F)}_{D,t} = \mathbf{MHA}(H_{C,t}, H_{D,t}) \tag{7}$$

The log probabilities for each node are computed using another weight vector, wP and we use *tanh* activation function. We clip values beyond $|C|$ similar to [3].

$$\gamma_t = tanh(H^{(F)}_{D,t}.(wP.H_{D,t})^T) \tag{8}$$

Finally, a *softmax* layer is used to compute final probabilities for each node, and the node with the highest probability is chosen as the next node and added to solution sequence Y.

4.2 Training GTA-RL

As discussed in the introduction, the policy-gradient-based methods achieve better results in COs. Thus, we use, REINFORCE, a policy-gradient-based algorithm by [24] to train GTA network. First, we define the performance measure ($J(\theta|G)$) of dynamic CO (G) using objective function P_{obj} (defined in Section 3) and using Equation 2, as $J(\theta|G) = \mathbb{E}_{Y\sim\pi_\theta}[P_{obj}(Y|G)]$. Then, we can use the *policy gradient theorem* to find the derivative of $J(\theta|G)$.

$$\nabla_\theta J(\theta|G) = \mathop{\mathbb{E}}_{Y\sim\pi_\theta}[P_{obj}(Y|G)\nabla_\theta log(\pi_\theta(Y|G))] \tag{9}$$

This derivative can be used to update parameters of GTA-RL. However, due to the high variance results in Equation 9, a baseline is used to accelerate the learning as below.

$$\nabla_\theta J(\theta|G) = \mathop{\mathbb{E}}_{Y\sim\pi_\theta}[(P_{obj}(Y|G) - b(G))\nabla_\theta log(\pi_\theta(Y|G))]$$

where $b(G)$ is the baseline function independent of Y.

Two baselines can be used: critic baseline [15] and roll-out baseline [8]. We experimented with both and found that using roll-out baseline works better than critic baseline for the dynamic CO.

4.3 GTA-RL for Real-time COs

The architecture we described so far assumes all input features of the problem instance are available beforehand. In real-time applications, all changes to the input features may not be available before we make the decisions. To handle this type of problems, we design the encoder to be iterative. At start, we only have the first time step input features, that is x_1 (defined in Section 3). As there is no temporal information we encode x_1 through the spatial encoder and send it to the decoder. The encoder also buffers the input x_1. Then, we modify the temporal pointer of the decoder to always points to the last time step of the embedded input features. This operation results in pointing to the most recent embeddings at every time step. Next, the decoder will select the first node for the solution sequence as the action. Then, we get the input at the next time step x_2. We concatenate the previous input, x_1, with the current input, x_2, and repeat the same process mentioned above. The iterative architecture follows the same approach until we find the complete solution.

In the iterative method, we cannot take the input features for the entire time horizon. However, by buffering the previous inputs, the model is able to gain knowledge about the transitions of input features over time. Since it takes some time to obtain such knowledge, the model may take more time before converging to a satisfactory solution. Compared to our original implementation, the encoder now encodes the input multiple times, thus increasing the memory costs. This is an acceptable trade-off as real-time COs are more difficult to optimize than dynamic COs.

The prediction/inference time of the real-time version is the same as the inference time of the dynamic version. This is because, during

the inference time, the neural network does not have to record gradient operation values. Therefore, despite iterations through the encoder, the inference time does not increase significantly, making it suitable for real-time applications.

5 EXPERIMENTAL SETUP

5.1 Problems

Dynamic Travelling Salesman Problem (TSP): Given n nodes/ points in the Euclidean space, and the objective is to find the order of visiting all the nodes such that the total traveled distance is minimal. In dynamic TSP, the initial node locations are assigned uniformly at random between (0,0)-(1,1) in 2d-space. Then, at every time step, the node locations are updated uniformly with maximum change of 0.1 in coordinates. In this way, the cost of traveling between two nodes is changing over time. Changing a node's coordinates leads to the changes of node features. Therefore, this scenario can be easily generalized to any problem where node features change over time such as in transport or telecommunication networks. For TSP, the context embedding H_C contains: the first selected node, the last selected node and the graph embedding computed by summing up all nodes at time t as;

$$H_{C,t} = \{h_{y_0}^{(F)} || h_{y_t}^{(L)} || H_{G,t}\}$$

where $H_{G,t} = \sum_{i=0}^{N} h_{t,i}^{(L)}$

Dynamic Vehicle Routing Problem (VRP): VRP is one of the most challenging CO problems. The conventional VRP contains a set of n nodes in Euclidean space and a vehicle with capacity c. Each node i has demand d_i of goods to satisfy and $d_i < c$ for all the nodes. Out of these n nodes, one node is called a depot, and the vehicle can visit the depot and fill goods until the load reaches the capacity. Similar to the TSP, the node coordinates are updated uniformly at random at every time step. However, in VRP, we do not change the location of the depot to demonstrate a node that is not changing over time. The objective is to find the minimum distance that the vehicle needs to travel to satisfy the demands of all the nodes. For VRP, the context embedding is the last selected node embedding, the remaining capacity r of the vehicle, and the graph embedding is;

$$H_{C,t} = \{h_{y_t}^{(L)} || r || H_{G,t}\}$$

Real-time Versions: The real-time TSP/ VRP is similar to dynamic TSP/ VRP; however, only the node locations at a time step are provided and the visited nodes order cannot be updated even if a better order is found later on.

5.2 Parameter Settings

We now detail the hyper-parameters used. Following previous works [8, 15], we test dynamic TSP and VRP with 10, 20, and 50 nodes. In GTA-RL, three layers of the temporal encoder are used, and the hidden dimension D^h is 128. We use 12800 problem instances during one epoch, and the batch is 32 instances. A total of 50 epochs is used for the training. We use a learning rate of 1e-4. The Adam optimizer [14] is used to train the network. The experiments were conducted in a machine consisting of an Intel Xeon(R) processor with 24 GB RAM and an Nvidia GRID P40 GPU. The hyper-parameters are the same for both VRP and TSP training. The code base is built on top of the following code base [2] and is available here [3].

5.3 Baselines and GTA-RL Variations

Our baselines include learning and hand-crafted heuristics, and optimal solutions computed with Integer Programming.

- **S2V-DQN:** S2V-DQN uses structure2vec for graph encoding and fitted Q-learning [7], and does not support VRP. There are two variants. The last selected can be added to the end of the tour (S2V-DQN-last) or to the middle of the tour where the resulting distance is minimum (S2V-DQN-sorted).
- **RNN-RL:** RNN-RL uses policy-gradient with two recurrent encoders named static and dynamic [18].
- **AM:** The model is limited to handle the a set of nodes [15], and we use the initial locations of nodes as an input to the model for dynamic cases. **AM-D** [20] extends this architecture to the VRP problem where changes from an agent behaviour are taken into account.
- **Gurobi:** This is the optimal solution for both dynamic TSP and VRP using the Gurobi solver. Computational times for dynamic TSP and VRP are however significantly higher than their static counterparts. Thus, we were only able to compute Gurobi for TSP10, TSP20 and VRP10. Specialized solvers such as Concorde do not support dynamic COs.
- **Nearest Neighbor (NN):** A modified version of NN is used for dynamic TSP where the next node is selected based on the distance to surrounding nodes, favouring closest nodes.
- **Farthest Insertion (F-Insert):** Similar to NN, we implement a modified version of the Farthest Insertion method to handle dynamic TSP as described in Kool *et al.*[15].
- **Min-Max Ant Colony Optimization (MM-ACO):** Ant Colony Optimization is often used as a heuristic to solve TSP problems [16]. A set of ants (agents) are deployed to find tours by selecting nodes sequentially and based on the quality of the tour, the probability of node selections is updated iteratively. Mavrovouniotis *et al.* [5] recently proposed a modified version named Min-Max Ant Colony Optimization to handle dynamic TSP. We use this version in our experiments.
- **GTA-RL-greedy** is the standard GTA-RL, where the node with the highest probability is greedily selected as the next node. **GTA-RL-bs** uses beam search for node selection. **GTA-RL-sum** uses a naive implementation of temporal decoder described in Section 4.1.2 (which sums up the encoding over the time axis). **GTA-RL-(0)** uses the first time step of the encoder output. **GTA-RL-rt** is the real-time GTA-RL.

6 EXPERIMENTAL RESULTS

First, we compared GTA-RL with the baselines for dynamic TSP and VRP w.r.t quality and generalization. Then, we analyse GTA-RL-variations to justify the architecture choices.

[2] https://github.com/wouterkool/attention-learn-to-routecode base
[3] https://github.com/udeshmg/GTA-RL

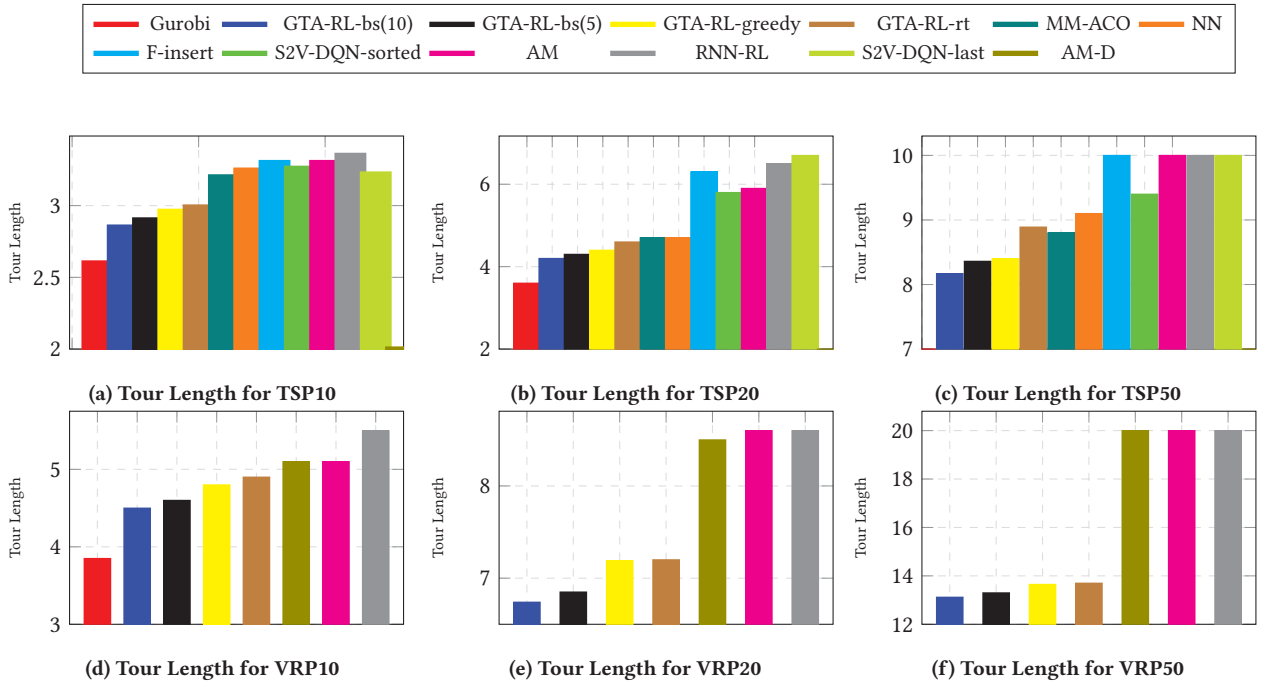


Figure 3: Tour length for dynamic TSP and VRP

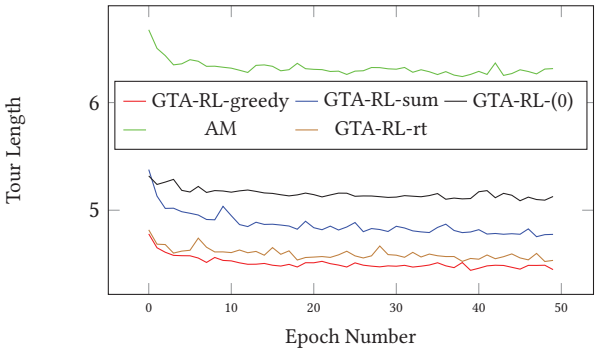


Figure 4: Tour length for validation data during the training

6.1 Dynamic TSP and VRP

Dynamic TSP (10, 20, 50): Figure 3a, 3b and 3c show the average tour length for the validation data set achieved by each algorithm with 10, 20, and 50 nodes. The learning-based methods, S2V-DQN-last, RNN-RL, AM, are not able to find satisfactory solutions, which shows that these are unsuitable for dynamic CO despite them being applicable to the static settings. S2V-DQN-sorted achieves a significantly lower tour length compared to S2V-DQN-last. This is because S2V-DQN-sorted is able to change the order of the selection process once it sees the dynamic changes. This shuffle however is impossible in a real-time setting where the selected node cannot be updated. As evident from Figure 3c, due to the increase in dynamicity of node locations in larger graphs, S2V-DQN-last, RNN-RL, AM achieve a lower performance and they are not able to find good solutions for larger graphs.

GTA-RL-greedy beats the other baseline algorithms. Employing a beam search strategy, GTA-RL-bs achieves a lower average tour length as the beam search can shuffle the final possible outcomes for further optimization. Note that the GTA-RL-greedy solution is not far from GTA-RL-bs(10) (which is the beamwidth of 10), indicating the high-quality solution even with the greedy strategy.

One could argue that GTA-RL can have an unfair advantage over S2V-DQN-last because the S2V-DQN works in a real-time fashion where future inputs are not taken into account. The rationale is justified with the results of GTA-RL-rt which considers the problem as a real-time problem. As depicted in Figure 3b, GTA-RL-rt is able to achieve a much shorter tour length compared to S2V-DQN-last. This highlights the effectiveness and flexibility of our approach.

The non-learning-based solutions, NN and MM-ACO, are able to find reasonable solutions as these baselines are designed to solve dynamic problems. However, GTA-RL-bs clearly outperforms these baselines. This is because GTA-RL is able to find a better order by avoiding sub-optimal spatial locations (i.e., avoiding time steps when visiting some nodes may incur high cost) and visiting those nodes at time steps when the cost is low (refer to Appendix A for visualization). This demonstrates that GTA-RL has been able to learn spatial-temporal features jointly.

By comparing the optimal controller Gurobi with GTA-RL-bs(10) from Figure 3a and 3b, the tour length is quite close to the optimal solution, with only 8% and 10% gap. As explained before, the complexity of dynamic COs makes it much harder to find a heuristic solution compared to the static scenario, making 10% gap acceptable.

Problem	GTA-RL trained with 50 nodes	GTA-RL trained with 20 nodes	Gap
TSP50	8.17	8.48	3.6%
VRP50	13.12	13.71	4.3%

Table 1: Tour length of Dynamic TSP50 and Dynamic VRP50 from GTA-RL trained with networks with 20 and 50 nodes.

Our objective is not to beat an optimal optimizer, but to propose a heuristic for *efficiently* solving the dynamic CO problems. This is important because Gurobi takes around 900s to solve *one* instance, while GTA-RL ran with the same computational resources only takes around *0.45s* (the inference time) with beam search to solve a batch of 100 problem instances.

Dynamic VRP (10, 20, 50): For dynamic VRP, we observe a trend similar to the trend shown in TSP results, as depicted in Figure 3d, 3e and 3f. GTA-RL-bs(10) manages to achieve the shortest tour length. All the GTA-RL variations outperform the other learning-based baselines, and the performance of GTA-RL-rt is on par with GTA-RL-greedy. The results from GTA-RL variations demonstrate that our proposed architecture can handle both dynamic TSP and dynamic VRP. We should also note that the optimal gap (of around 12%) is similar to that achieved in TSP, being slightly higher due to the increased VRP complexity.

6.2 GTA-RL Variations

We now demonstrate the impact of temporal encoder and temporal pointer. In Figure 4, all the variations of GTA-RL achieve a lower tour length compared to AM, due to temporal encoder's ability to encode the temporal information. Even though GTA-RL-greedy, GTA-RL-sum, GTA-RL-(0) all use the same encoder, they use different decoders. As we explained, GTA-RL-sum and GTA-RL-(0) do not use the temporal decoder and naively compute a fixed embedding from the encoding output. These two are outperformed by GTA-RL-greedy, which shows the benefit of the temporal decoder. GTA-RL-rt achieve a slightly higher tour length than GTA-RL-greedy because GTA-RL-rt, at its first-time step, takes the decision only based on the current time step node features. Since node features are changing uniformly at random, GTA-RL-rt cannot guarantee that the selected node will be the best given that they can change externally in the future.

6.3 Generalization

We now demonstrate that GTA-RL can generalize to larger graphs beyond those used for training. We first compute the tour length for a set of graphs with 50 nodes by GTA-RL trained originally with a set of graphs with 20 nodes. Then, for comparison, we compute the tour length for the same set of graphs originally trained for graphs with 50 nodes. These results are shown in Table 1. In both cases (dynamic TSP50 and VRP50), the tour length achieved by GTA-RL trained with 20 nodes is close to GTA-RL trained with 50 nodes with only around 3% of discrepancy. This shows that GTA-RL can generalize for much larger graphs than originally trained for. Despite scalability being an open problem in the neural CO domain, this is a highly promising result.

Algorithm	10	20	50
Brute-force	12.75	22.62	75.07*
GTA-RL-bs(10)	13.72	27.64	71.706
MM-ACO	17.42	30.82	79.40
NN	17.20	31.05	76.06

Table 2: Travel time (in minutes) in Melbourne CBD for a different number of locations to visit (10, 20 and 50). *Due to the large number of combinations, the brute-force algorithm has not been able to find the optimal solution.

6.4 Real-world Map

We use an area of Melbourne CBD to evaluate the TSP algorithms similar to the setup used in Xu *et al.* [29]. We picked a central area of CBD which consists of 98 nodes and 174 edges. We assume that there is a delivery person who needs to deliver packages to 10, 20 or 50 locations. We change the travel time between the locations over time as a percentage of the free-flow-travel time to account for the dynamic changes in the travel time. Then, we input these travel times into algorithms to find a delivery order to visit all the locations. The actual total travel time is then computed by summing up the shortest travel times between adjacent nodes in the order. We select best-performing algorithms from Section 6.1 (GTA-RL-bs(10), MM-ACO and NN). The above discussed approach provides approximate solutions over the original road graph for dynamic TSP assuming that there is a direct link between the nodes a delivery person is going to visit. To find the optimal solution in the original graph, we use a brute-force approach. Using Gurobi to find the solution is not feasible in the actual graph due to the a large number of nodes. The brute-force approach tries a different order of locations and finds the lowest total travel time. Due to the high computational time of the brute-force algorithm, we use a time limit of 1 hour to solve one instance.

Table 2 shows the travel time for algorithms tested on the Melbourne CBD graph. GTA-RL is able to find a lower travel time compared to MM-ACO or NN. This shows that GTA-RL performs better even on real-world graphs. The brute-force algorithm cannot find the optimal solution in an acceptable time, when there is a large number of locations. In contrast, GTA-RL finds the solution in less than one second. This result shows that GTA-RL can be used for real-world road graphs to find close to optimal solutions in an acceptable time window.

7 CONCLUSION

In this work, we propose an effective and efficient learning-based method to solve dynamic CO problems, which is crucial when applying CO to real-world applications. We evaluate our method by comparing it with several state-of-the-art approaches and show that our approach substantially outperforms both learning-based and hand-crafted heuristics and is on par with the optimal controller in the dynamic CO domain.

There are several interesting directions of future work. First, scalability is an open problem in the neural CO domain and Section 6.3 provides promising results towards this direction and can be investigated further. Second, our real-time encoder encodes at

every time step, thus can be memory expensive and this can be investigated further to store and process data more efficiently. Finally, dynamic CO is much harder than static CO; thus, proposing learning heuristics that are close to the optimal is much more challenging. As this is a first step in learning heuristics for dynamic CO, we hope to reduce the optimal gap using local search in our future research.

REFERENCES

[1] Abdullah Aldwyish, Egemen Tanin, Hairuo Xie, Shanika Karunasekera, and Kotagiri Ramamohanarao. 2021. Effective Traffic Forecasting with Multi-Resolution Learning. In *17th International Symposium on Spatial and Temporal Databases*. 44–53.

[2] Thomas Barrett, William Clements, Jakob Foerster, and Alex Lvovsky. 2020. Exploratory Combinatorial Optimization with Reinforcement Learning. In *AAAI Conference on Artificial Intelligence*. 3243–3250.

[3] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. 2016. Neural Combinatorial Optimization with Reinforcement Learning. In *arXiv preprint arXiv:1611.09940*. arXiv:1611.09940 [cs.AI]

[4] Dimitris J. Bertsimas and Garrett van Ryzin. 1991. A Stochastic and Dynamic Vehicle Routing Problem in the Euclidean Plane. *Operations Research* 39, 4 (1991), 601–615.

[5] Sudipta Chowdhury, Mohammad Marufuzzaman, Huseyin Tunc, Linkan Bian, and William Bullington. 2018. A Modified Ant Colony Optimization Algorithm to Solve a Dynamic Traveling Salesman Problem: A Case Study with Drones for Wildlife Surveillance. *Journal of Computational Design and Engineering* 6, 3 (10 2018), 368–386.

[6] K. F. Chu, A. Y. S. Lam, and V. O. K. Li. 2017. Dynamic Lane Reversal Routing and Scheduling for Connected Autonomous Vehicles. In *International Smart Cities Conference*. 1–6.

[7] Hanjun Dai, Elias Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning Combinatorial Optimization Algorithms over Graphs. In *Advances in Neural Information Processing Systems*, Vol. 30. 6351–6361.

[8] Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. 2018. Learning Heuristics for the TSP by Policy Gradient. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*. 170–181.

[9] Iddo Drori, Anant Kharkar, William R. Sickinger, et al. 2020. Learning to Solve Combinatorial Optimization Problems on Real-World Graphs in Linear Time. In *IEEE International Conference on Machine Learning and Applications (ICMLA)*. 19–24.

[10] Udesh Gunarathna, Shanika Karunasekera, Renata Borovica-Gajic, and Egemen Tanin. 2022. Real-Time Intelligent Autonomous Intersection Management Using Reinforcement Learning. In *IEEE Intelligent Vehicle Symposium*. 135–144.

[11] Udesh Gunarathna, Hairuo Xie, Egemen Tanin, Shanika Karunasekera, and Renata Borovica-Gajic. 2021. Real-Time Lane Configuration with Coordinated Reinforcement Learning. In *Machine Learning and Knowledge Discovery in Databases*. Cham, 291–307.

[12] Alex Hall, Steffen Hippler, and Martin Skutella. 2007. Multicommodity Flows Over Time: Efficient Algorithms and Complexity. *Theoretical Computer Science* 379, 3 (2007), 387–404.

[13] A. K. M. Mustafizur Rahman Khan, Oscar Correa, Egemen Tanin, Lars Kulik, and Kotagiri Ramamohanarao. 2017. Ride-Sharing is About Agreeing on a Destination. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*.

[14] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *arXiv preprint arXiv:1412.6980*. arXiv:1412.6980 [cs.LG]

[15] Wouter Kool, Herke van Hoof, and Max Welling. 2019. Attention, Learn to Solve Routing Problems!. In *International Conference on Learning Representations*.

[16] Michalis Mavrovouniotis, Felipe M. Müller, and Shengxiang Yang. 2017. Ant Colony Optimization With Local Search for Dynamic Traveling Salesman Problems. *IEEE Transactions on Cybernetics* 47, 7 (2017), 1743–1756.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. 2015. Human-level Control Through Deep Reinforcement Learning. *Nature* 518 (2015), 529–533.

[18] Mohammadreza Nazari, Afshin Oroojlooy, et al. 2018. Reinforcement Learning for Solving the Vehicle Routing Problem. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. 9861–9871.

[19] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. 2020. EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 04 (Apr. 2020), 5363–5370.

[20] Bo Peng, Jiahai Wang, and Zizhen Zhang. 2020. A Deep Reinforcement Learning Algorithm Using Dynamic Attention Model for Vehicle Routing Problems. In *Artificial Intelligence Algorithms and Applications*. 636–650.

[21] Yun Peng, Byron Choi, and Jianliang Xu. 2021. Graph Learning for Combinatorial Optimization: A Survey of State-of-the-Art. *Data Science and Engineering* 6, 2 (2021), 119–141.

[22] Martin Riedmiller. 2005. Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method. In *Machine Learning and Knowledge Discovery in Databases*. 317–328.

[23] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 2020. DySAT: Deep Neural Representation Learning on Dynamic Graphs via Self-Attention Networks. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 519–527.

[24] Richard S. Sutton and Andrew G. Barto. 1998. *Introduction to Reinforcement Learning* (1st ed.). Vol. 135. MIT Press.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, et al. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*.

[26] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph Attention Networks. *arXiv preprint arXiv:1710.10903* (2017).

[27] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer Networks. In *Advances in Neural Information Processing Systems*, Vol. 28. 2692–2700.

[28] Christopher JCH Watkins and Peter Dayan. 1992. Technical Note: Q-learning. *Machine learning* 8 (1992), 279–292.

[29] Xiaolong Xu, Hao Yuan, Peter Matthew, Jeffrey Ray, Ovidiu Bagdasar, and Marcello Trovati. 2020. GORTS: Genetic Algorithm Based on One-by-one Revision of Two Sides for Dynamic Travelling Salesman Problems. *Soft Computing* 24, 10 (2020), 7197–7210.

[30] Xue Song Yan, Li Shan Kang, Zhi Hui Cai, and Hui Li. 2004. An Approach to Dynamic Traveling Salesman Problem. In *International Conference on Machine Learning and Cybernetics*. 2418–2420.

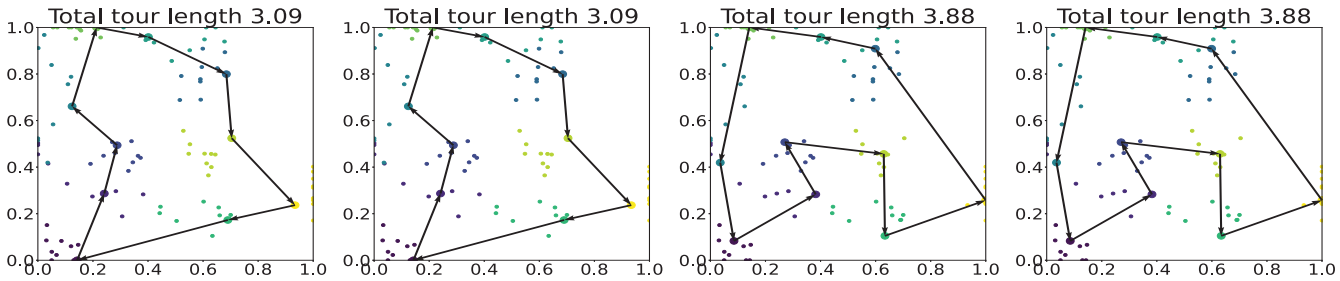
[31] Haitao Yuan, Guoliang Li, Zhifeng Bao, and Ling Feng. 2021. An Effective Joint Prediction Model for Travel Demands and Traffic Flows. In *International Conference on Data Engineering*. 348–359.

[32] Marvin Zhang, Zoe McCarthy, Chelsea Finn, et al. 2016. Learning Deep Neural Network Policies with Continuous Memory States. In *IEEE international conference on robotics and automation*. 520–527.

[33] Chuanpan Zheng, Xiaoliang Fan, Cheng Wang, and Jianzhong Qi. 2020. GMAN: A Graph Multi-Attention Network for Traffic Prediction. In *AAAI Conference on Artificial Intelligence*. 1234–1241.

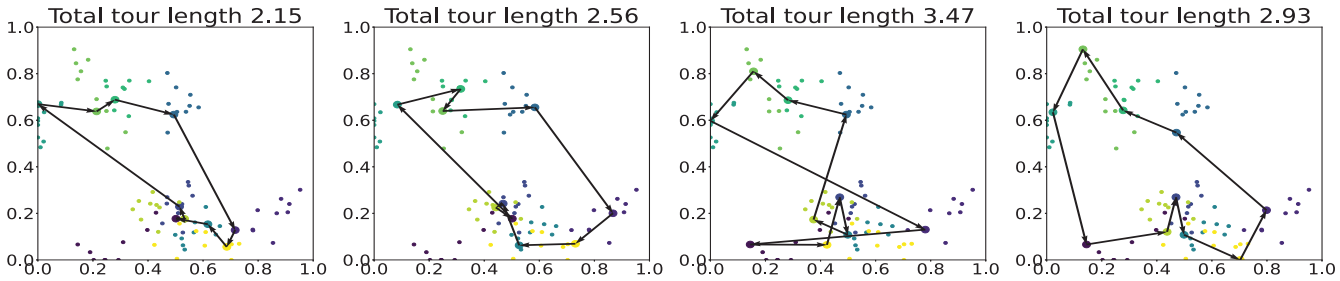
A VISUALIZING THE DYNAMIC TSP

The visualization in Figure 5 shows two scenarios of dynamic TSP with 10 nodes. The two scenarios are randomly generated as described below. In each sub-figure in Figure 5, there are 10 nodes that are assigned uniformly at random in the euclidean space. Then, at every time step, the node locations (XY coordinates) are updated uniformly at random for each node with a maximum change of 0.1. In this way, the travelling cost between nodes changes over time. There are 10 different colours to represent each of the 10 nodes. The dots in the same colour refer to different positions in the same node at different time steps. A solution for dynamic TSP needs to visit each node only once in a time step i.e. only one dot from the same colour needs to be selected in a solution. The objective is to visit all the nodes (exactly once) with a minimum tour length. We visualize the tours computed by Gurobi (optimal solver), GTA-RL (our proposed solution), NN and MM-ACO (refer to Section 5.3 for the definitions of these baselines) for dynamic TSP 10 in. are The first scenario is a relatively easier scenario compared to the second scenario. In the first scenario, GTA-RL has achieved the optimal results like the Gurobi solver. NN and MM-ACO have yielded a longer route compared to Gurobi and GTA-RL. In the second scenario, GTA-RL finds the closest tour length to the optimal solution compared to MM-ACO and NN. The interesting thing we can observe is that when GTA-RL was selecting nodes, GTA-RL has selected time steps when the node locations are close to each other which reduces the tour length by avoiding far away node locations. The same behaviour (where far away nodes have been avoided) is observed with the optimal solver, as seen in both Figure



(a) Gurobi (Optimal Solution) (b) GTA-RL-bs (c) Nearest Neighbor (d) MM-ACO

(i) Scenario One



(e) Gurobi (Optimal Solution) (f) GTA-RL-bs (g) Nearest Neighbor (h) MM-ACO

(ii) Scenario Two

Figure 5: Figure shows the order of selected nodes by Gurobi, GTA-RL, NN and MM-ACO for dynamic TSP with 10 nodes. The dots with the same color indicates the same city locations at different time steps. The algorithms select to visit a node in one time step which has been highlighted by a dot larger than other nodes in the same color. X, Y axes represent Cartesian coordinates.

5e and Figure 5f. This shows that GTA-RL has learned a heuristic similar to the behaviour of the optimal solver. However, NN in particular in the second scenario does not perform well because simply selecting the nearest neighbour is not sufficient when there are multiple locations in the same region as observed in the bottom of Figure 5g. MM-ACO achieves a lower tour length compared to NN by avoiding such drawbacks, however, it also fails to avoid distant nodes. This shows that GTA-RL has learned a better heuristic than NN and MM-ACO. GTA-RL has been able to understand both spatial and temporal features of the problem.

Advantages of GTA-RL over Other Heuristics: The objective of neural combinatorial optimization is to find a heuristic for CO, without manual feature engineering. GTA-RL provides such a heuristic. As we have shown in our experiments, GTA-RL performs better than other heuristics. Other heuristics are usually designed by manual feature engineering and domain experts specific to a problem. These may not explore all the important features while designing the heuristic and especially designing such a heuristic in a dynamic setting is even more challenging. GTA-RL can learn these type of features automatically using a large number of examples during the training. Thus, GTA-RL can even iterate through different features automatically and pick the best features to solve the given problem. This is different from using a solution such as NN that only explores the nearest nodes as a manual feature as evident from our visualizations above. Furthermore, GTA-RL does not use a problem-specific architecture but a general architecture which allows GTA-RL to learn more than one combinatorial problem. In other words, GTA-RL can perform sequential decision making for multiple problems.

B DEEP LEARNING AND REINFORCEMENT LEARNING IN COMBINATORIAL OPTIMIZATION

We first provide a general overview of Neural Combinatorial Optimization domain, then provide more insights into how GTA-RL solves dynamic TSP.

B.1 Deep Learning over Graphs

A core factor in the success of deep learning compared to other machine learning techniques is the ability of deep neural networks to represent a problem/set of features using a low dimensional vector (called embedding). This type of embedding is able to capture all the necessary information about a problem. Such learned embedding then can be used to make predictions or other decision-making. This is also known as *representation learning*. However, these types of deep neural networks were used often with structured data with fixed sizes. Embedding graphs is more challenging because of the graphs' unstructured nature, the variable number of

nodes in graphs and the difficulty of embedding the spatial features. With new neural network architectures, researchers were able to learn representations for unstructured data overcoming above mentioned challenges. Example neural networks include LSTM-based neural networks, RNN-based neural networks, attention networks and message passing neural networks. These architectures can embed the variable-length nodes in graphs while taking the adjacency matrix into account. Thus, these neural networks were able to compute a vector representation of graphs, which has shown impressive results in several graph-based problems.

B.2 GTA-RL for Combinatorial Optimization

We provide insights into why GTA-RL is successful in the dynamic CO domain, such as in solving TSP problems.

Embedding the Temporal Graph: The first step in GTA-RL is encoding the graph along with the dynamic information using the temporal encoder. The temporal encoder is able to learn a good representation of vectors to preserve the dynamic and spatial features of the given problem instance. This is because the temporal encoder contains two attention mechanisms, one for spatial data and one for temporal data. The spatial attention layer considers each node (for a given time step) in the graph as a sequence and computes the inter-dependency between these nodes using self-attention. During the self-attention, computations dependency between each node with each other node is considered. This can be thought of as a message-passing between nodes. Thus the output from the spatial attention layer contains a representation (a vector) for each node which includes the information about neighbouring nodes as well. Thus, this process avoids manual feature engineering and allows the neural network to learn a representation for the graph. Similarly, in the temporal dimension, using the same mechanism, the neural network finds a representation for each node considering its previous and future changes. Once we have learned both representations, we combine these two representations using the integration layer. The output is a representation for each node at every time step. This representation for each node at a given time step will contain information about the neighbouring nodes as well as information about the previous and future changes of that node. At the end of this process, we encode the entire unstructured graph instance into a vector representation and this vector representation contains all information about the problem and we use it for the decision making process. For better understanding of our method, we focus on using GTA-RL in solving TSP problems, where all the node locations and their changes are represented so that we can use the representation to find a tour to minimize the travel distance.

Decoding the Solution: Once the temporal encoder encodes the problem, the next step is to sequentially decode the solution using the encoder's output. In GTA-RL, the decoder computes the context embedding which is used to aid the decoding process. When GTA-RL is used for solving in TSP, the context embedding uses the graph embedding, the embedding of the first node selected on the tour and the embedding of the last node selected up to now. This information is critical for selecting the next node compared to information about nodes in the middle of the sequence decoded up to now. Thus, using context embedding rather helps the decoder to focus on important information from the output of the temporal encoder. Further, GTA-RL can use a temporal pointer to focus on specific parts of the temporal encoder output to handle dynamic information.

We use a modified version of REINFORCE algorithm to train the GTA-RL in solving TSP. The REINFORCE algorithm learns the best tour in an iterative manner. For each training iteration, the total tour length is computed for completed tours. The total length of tours is used as the training cost in the GTA-RL network for training. At every iteration, the GTA-RL network's parameters are adjusted to result in lower tour lengths. Thus, we adjust the parameters of both the temporal encoder and decoder to reduce the tour length. Thus the vector embedding learned by both the temporal encoder and temporal decoder are optimized to reduce the tour length.