Solving Dynamic Graph Problems with Multi-Attention Deep Reinforcement Learning

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

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

Shanika Karunasekara
University of Melbourne
Melbourne
Australia
karus@unimelb.edu.au

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

Abstract

Graph problems such as traveling salesman problem, or finding minimal Steiner trees are widely studied and used in data engineering and computer science. Typically, in real-world applications, the features of the graph tend to change over time, thus, finding a solution to the problem becomes challenging. The dynamic version of many graph problems are the key for a plethora of real-world problems in transportation, telecommunication, and social networks. In recent years, using deep learning techniques to find heuristic solutions for NP-hard graph combinatorial 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 graph problems. The dynamic nature makes NP-hard graph problems much more challenging to learn, and the existing methods fail to find reasonable solutions.

In this paper, we propose a novel architecture named Graph Temporal Attention with Reinforcement Learning (GTA-RL) to learn heuristic solutions for graph-based dynamic combinatorial optimization problems. The GTA-RL architecture consists of an encoder capable of embedding temporal features of a combinatorial problem instance and a decoder capable of dynamically focusing on the embedded features to find a solution to a given combinatorial problem instance. We then extend our architecture to learn heuristics for the real-time version of combinatorial optimization problems where all input features of a problem are not *known a priori*, but rather learned in real-time. Our experimental results against several state-of-the-art learning-based algorithms and optimal solvers demonstrate that our approach outperforms the state-of-the-art learning-based approaches in terms of effectiveness and optimal solvers in terms of efficiency on dynamic and real-time graph combinatorial optimization.

*K*eywords Graphs, Combinatorial Optimization, Spatial Databases, Reinforcement Learning

1 Introduction

Graph combinatorial problems have been studied by computer scientists for decades as these problems appear in many fields, including transportation, computer networks, social networks, and logistics planning. Most of the existing approaches focus on the static version of the problems where the input attributes do not change over time. Unfortunately, many real-world applications of these problems in transportation, telecommunication, and social networks require the dynamic version to be addressed where the input attributes change over time [1, 2, 3].

Let us consider a motivating example for a dynamic graph problem, 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. This is an NP-hard optimization problem, and can be reduced to a minimum Steiner tree [4]. 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. Many road network related graph problems [5, 6, 7] approximate this case and assume 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 [3, 8]). This common problem then becomes a dynamic graph combinatorial optimization 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 combinatorial optimization. A similar scenario can happen in telecommunication networks where we need to allocate bandwidths to communication links to maximize the overall network utilization while traffic in each link changes over time[9]. With these real-world examples, it is evident that the dynamicity of graph attributes need to be considered to make them applicable for many real-world scenarios.

Due to the NP-hard nature, even in static combinatorial optimizations, finding an exact solution for a large instance is computationally expensive [10]. On the other hand, heuristic solutions can be fast but require domain knowledge of a given problem and a significant amount of manual-engineering while designing the heuristics [11]. Finding such a heuristic in a dynamic setting can be even more challenging due to a large number of possible combinations[1]. In this work we thus seek a generalized method to find fast and efficient heuristics for dynamic graph problems without relying on hand-crafted methods.

The recent advancements in Deep Neural Networks (DNN) and Reinforcement Learning (RL) have led to finding heuristic solutions for combinatorial optimization efficiently without labor intensive hand-crafted engineering [11, 12, 13, 14, 15]. All these approaches model combinatorial optimization as a graph (complete or incomplete) 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. However, most of these works focus only on static data [13, 14, 15]. As we demonstrate in our experiments, these algorithms fail to find reasonable solutions in dynamic graph combinatorial optimization problems.

To identify the shortcomings of the mentioned approaches for dynamic settings, we categorize the existing methods into two main types: (1) Message passing neural networks with value-based reinforcement learning [11, 12], and (2) Attention-based/Recurrent Neural Networks (RNN) with policy-gradient reinforcement learning [13, 14, 15]. The former methods use a message-passing neural network to iteratively encode the graph information and use Q-learning [16] for the decision making. The latter use an attention mechanism [17] 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 suggest [13, 15], the latter approaches achieve better performance for combinatorial optimizations largely due to the fact that the policy-gradient RL algorithms perform better when the action space (i.e., the number of nodes to select in a combinatorial problem) is large/variable [18].

Despite the success of attention-based policy gradient methods in static settings, the reason for them not performing well in dynamic combinatorial optimizations is three-fold. (1) The graph/input features are embedded before making decisions/actions. Thus, changes in graph/input features due to the decision-making process are not captured, which is the case in dynamic combinatorial problems. (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. The above-mentioned attention-based models are unable to capture such temporal changes. (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 in dynamic settings.

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. 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

---

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

==dynamically pay attention to the spatio-temporal representation for the decision-making stage. 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 problems== [1, 19]. Next, 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:

- We propose a novel deep learning architecture, GTA-RL, to tackle the dynamic graph combinatorial optimization problems.
- We introduce an encoder and decoder that are capable of preserving temporal information in sequential decision making.
- We propose an iterative encoder-decoder architecture that can handle real-time information of a graph combinatorial optimization problem.
- Our extensive experimental results demonstrate that GTA-RL achieves superior performance compared to state-of-the-art learning-based algorithms in the associated combinatorial optimization domain as demonstrated on an important class of graph problems.

2   Related Work

Until recently, the success of Deep Neural Networks (DNN) was mostly limited to prediction tasks [13]. With the advancements in reinforcement learning algorithms combined with DNNs emerged Deep Reinforcement Learning (DRL) algorithms, which demonstrated great success in sequential decision making and control problems in many domains including atari games [20], traffic control [21, 22], and robot navigation [23]. In recent years, DRL has been applied to many combinatorial optimization problems to learn heuristics that require no human input [12, 14, 11, 15, 13, 24]. These works have achieved near-optimal performance outperforming hand-crafted heuristics in many combinatorial optimization 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 [20]. (2) Using Policy-gradient algorithms such as REINFORCE and Actor-Critic algorithms [18].

Value-based method for graph combinatorial optimization was first proposed by Dai *et al.*[11]. The proposed architecture uses *structured2vec* to embed the graph information and uses fitted Q-learning [25] 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.*[12] using 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.*[11].

The policy-gradient methods have demonstrated improved results over value-based based counterparts, as evident by the recent research [14, 13, 24]. The main reason is because in graph combinatorial optimization, 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 [18]. The first policy-gradient algorithm for graph combinatorial optimization was proposed by Bello *et al.*[24]. The paper uses an LSTM encoder-decoder architecture named *pointer network* which was first proposed by Vinyals *et al.*[26] 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.* [15] 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 combinatorial optimization 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. As our experiments show, GTA-RL achieves superior performance in several combinatorial problems over the architecture of Nazari *et al.* [15].

The next generation of policy-gradient methods uses pure attention architectures instead of RNNs inspired by the architecture proposed in Vaswani *et al.*[17]. Deudon *et al.* uses a Multi-head attention [17] for encoding the combinatorial optimization problem and a decoder with a pointing mechanism similar to Bello *et al.* but without recurrent elements. The REINFORCE algorithm [18] with a critic baseline has been used for training the network. In parallel to Deudon *et al.*, Kool *et al.* [13] proposed an improved architecture by introducing a new attention-based decoder instead of the pointing mechanism. The paper also shows that using a rollout baseline instead of the critic baseline improves the results further for combinatorial optimization problems. Kool *et al.* outperforms existing methods in

Algorithm	Decision making algorithm	Type of neural network	Static	Dynamic	Real-time
Dai *et al.*[11]	Value-based	Message-passing	✓	-	✓
Barrett *et al.*[12]	Value-based	Message-passing	✓	-	✓
Bello *et al.*[24]	Policy-gradient	RNN	✓	-	-
Vinyals *et al.*[26]	Supervised learning	RNN	✓	-	-
Nazari *et al.*[15]	Policy-gradient	RNN	✓	-	-
Deudon *et al.*[14]	Policy-gradient	Pure-attention	✓	-	-
Kool *et al.* [13]	Policy-gradient	Pure-attention	✓	-	-
Our approach (GTA-RL)	Policy-gradient	Spatio-temporal attention	✓	✓	✓

Table 1: The table shows a summary of existing methods and GTA-RL. The ✓ indicates whether a particular algorithm supports a feature.

TSP, VRP (Vehicle Routing Problem), and variants of both TSP, VRP. Peng *et al.* [27] extends this architecture to the VRP problem where changes from an agent behaviour are taken into account. However, external changes to the graph problem are not considered. A similar graph attention network approach is followed by Drori *et a.l*[28] as well. 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 combinatorial optimization problems and dynamically focusing on temporal features during the decision making process. Our experimental results show that we outperform the afore-mentioned state-of-the-art approaches for dynamic combinatorial optimization problems. We also extend our architecture to real-time combinatorial optimization.

We provide a summary of the existing methods in Table 1.

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 as 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. We denote $x_{i,t} \in \mathbb{R}^D$ 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* $f_c : x_{i,t} \times x_{j,t} \mapsto \mathbb{R}$. 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). Note that the complete graph scenario can also be considered as a simple sequence of nodes without any graph/edge properties, resulting in dynamic combinatorial optimization.

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. The C is an evaluating function that checks whether the sequence Y satisfies all the problem constraints[2].

Learning Formulation: Now that we formulated the 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 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

---

[2]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.
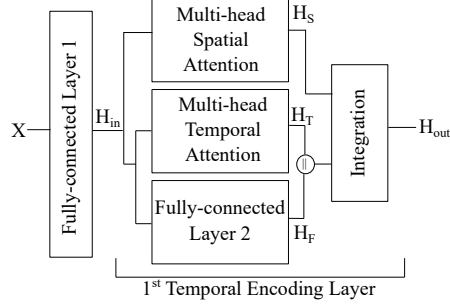
Figure 1: Temporal Encoder with one layer. The input features X will be encoded with an initial *Fully-connected Layer 1* before the first temporal encoding block. The temporal encoder's three parallel layers are shown next named *Spatial Attention, Temporal Attention, Fully-connected Layer 2*. The outputs from these layers are combined at the *Integration Layer* and the final output is provided to the next temporal encoding layer.

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:

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

Finally, our objective transforms to learn and represent the set of parameters θ in Equation 2, so that we minimize P_{obj}.

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 below.

$$\pi_\theta(Y|G) = \Pi_{t=1}^{T} \pi_\theta(y_t|G_{1:t-1}, Y_{1:t-1}) \tag{3}$$

Note that, in Equation 3, the graph instance is $G_{1:t-1}$ which indicates that only the information up to time step t is available. Equation 3 is harder to optimize than the dynamic combinatorial optimization 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 combinatorial optimization to be on par with dynamic combinatorial optimization.

Remark: The real-time combinatorial problems typically occur in transportation networks where the road network is represented as a graph. The edge attributes can be traffic loads or average speeds of a road segment. Given that we want to reach several points in the road network, first, we 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, these external traffic information could not have been accessed beforehand. This is a typical example of a real-time combinatorial problem.

4 Graph Temporal Attention with Reinforcement Learning

To find a solution to the dynamic combinatorial optimization problem defined in Equation 2, we need to find a neural network architecture to represent θ parameters and a learning algorithm to train the neural network. We propose an encoder-decoder neural network architecture named *Graph Temporal Attention (GTA)* to represent θ parameters. Then, we use a loss function computed through policy-based reinforcement learning to train θ parameters. This section provides details about these two components.

4.1 Graph Temporal Attention

A major limitation of the previously proposed attention based models [13, 14] 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, GTA network follows an encoder-decoder architecture consisting of two components, namely a *temporal encoder* and a *temporally pointing decoder*.

4.1.1 Temporal Encoder

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 $H^{(0)} \in \mathbf{R}^{T \times N \times D^h}$, at *fully connected layer 1*. D^h refers to the hidden dimension of *fully connected layer 1*. The output from *fully connected layer 1* is given to the temporal enoder. The temporal encoder contains a spatial attention, a temporal attention and a fully-connected layer in parallel. $H^{(0)}$ is given as an input to these three layers and the outputs from these layers $(H_S^{(1)}, H_T^{(1)}, H_F^{(1)})$ 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 second layer, and so on. We denote $H^{(l)}$ as the input to the lth temporal encoding layer and $H^{(l+1)}$ as the output from that layer. 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 learning heuristics for combinatorial optimization, this idea has recently emerged in other prediction tasks [29, 30, 31]. Zheng *et. al* [29] uses a neural architecture named *STAtt Block* which consists of two attention mechanisms and a fixed graph embedding for traffic prediction, while Gangopadhyay *et al.* [30], and Pareja *et al* [31] use an LSTM-based mechanism. Compared to LSTM, attention-based methods are more computationally efficient and have the ability to handle graph based information [17]. Therefore, our proposed temporal encoder, whilst motivated by *STAtt Block*, differs in several key aspects. First, in our temporal encoder we do not use the fixed graph embedding. This is because in traffic prediction tasks learning happens for a single road network topology, whereas in combinatorial optimizations 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 combinatorial problems, 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 or congestion of every node change over time. In the temporal encoder, we introduce the *fully connected layer 2* in parallel to *temporal attention layer*, to handle nodes whose attributes do not change over time.

Next, we detail each sub-element in the temporal encoder.

Spatial Attention Layer: In a given graph instance, there are dependencies between nodes, and the spatial attention layer is used to encode those dependencies.

The spatial attention layer uses the self-attention mechanism introduced by Vaswani et al.[17]. The spatial attention layer considers a list of node features at a given time step as a sequence: $H_{S,t}^{(l)} = \{h_{1,t}^{(l)}, h_{2,t}^{(l)}, ...h_{N,t}^{(l)}\} \in \mathbb{R}^{N \times D^h}$. The spatial attention uses the same standard parameters named query ($wS_q^{(l)} \in \mathbb{R}^{D^h \times D^k}$), key ($wS_q^{(l)} \in \mathbb{R}^{D^h \times D^k}$) and value ($wS_q^{(l)} \in \mathbb{R}^{D^h \times D^v}$). In this setup, we set D^k and D^v to the same dimension as D^h. However, the derivations of the paper can be applied even when these values are different.

First, we compute the following three projections as below.

$$qS_t^{(l)} = wS_q^{(l)} H_{S,t}^{(l)} \in \mathbb{R}^{N \times D^h}$$
$$kS_t^{(l)} = wS_k^{(l)} H_{S,t}^{(l)} \in \mathbb{R}^{N \times D^h}$$
$$vS_t^{(l)} = wS_v^{(l)} H_{S,t}^{(l)} \in \mathbb{R}^{N \times D^h}$$

Then, we compute the scaled dot product between the query and the key projections and use a *softmax layer* as in Vaswani et al.[17].

$$\alpha_t = softmax\left(\frac{(qS_t^{(l)})^T . kS_t^{(l)}}{\sqrt{D^h}}\right) \in \mathbb{R}^{N \times N} \tag{4}$$

Here, $\alpha_{t,i,j} \in \alpha_t$, represents the relevance of node j to node i at time t. In a non-complete graph, we can set $-\infty$ to non-adjacent nodes before computing the softmax value. Finally, we can output the spatial attention for each node with their relevance to others as:

$$H_{S,t}^{(l+1)} = \alpha_t . vS_t^{(l)} \in \mathbb{R}^{N \times D^h} \tag{5}$$

By stacking $H_{S,t}^{(l+1)}$ over the time axis, we get the $H_S^{(l+1)}$ as below.

$$H_S^{(l+1)} = \|_{t=1}^T H_{S,t}^{(l+1)} \in \mathbb{R}^{T \times N \times D^h} \tag{6}$$
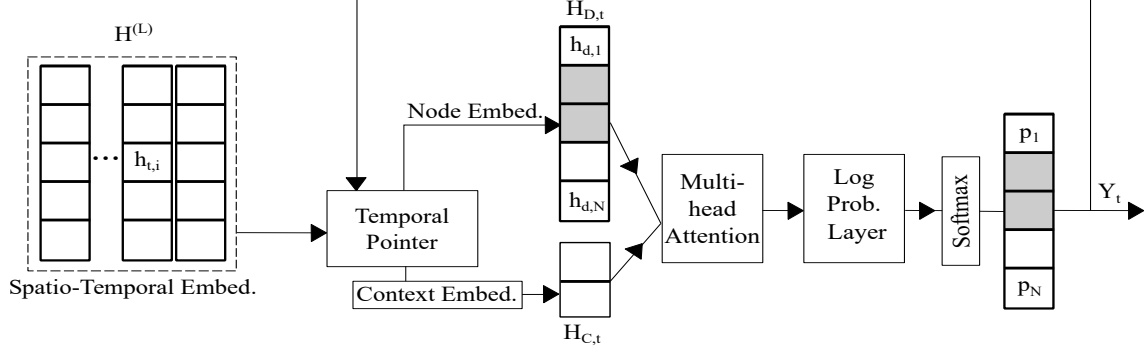
Figure 2: This figure depicts the entire decision making process of the Temporal Decoder. The temporal decoder consists of a *Temporal Pointer* which adaptively points to the part of the encoded output from the temporal encoder(*Node Embed*). The temporal decoder then generates a *Context. Embed* which represents the context of the problem, given its partial solution. These two outputs will be fused together at *Multi-head Attention Layer* and the final probabilities for each node will be computed at *Log Prob. Layer*. The node with the highest probability is selected as a part of the solution and will be added to Y_t. During the entire process, invalid nodes will be ignored.

Temporal Attention Layer with fully Connected Layer: In dynamic combinatorial problems, in addition to the dependency between nodes, there is a dependency between node features across different time steps for the same node. To encode such dependencies, the temporal attention layer is introduced.

We also note that in some dynamic combinatorial problems, apart from nodes that change over time, there can be some nodes that are not changing over time as well. For example, 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 temporal attention layer 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 layer. First, we 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 $H_F^{(l)} \in \mathbb{R}^{T,U,D^h}$ which only contains the nodes with fixed features. In the fully connected layer, we apply a linear transformation for $H_F^{(l)}$ to get $H_F^{(l+1)}$.

Then, in the temporal attention layer, 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 spatial attention layer, we consider a sequence, but now 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 $i \notin \mathbb{U}$. The temporal attention uses the same standard parameters named query ($wT_q^{(l)} \in \mathbb{R}^{D^h \times D^h}$), key ($wT_q^{(l)} \in \mathbb{R}^{D^h \times D^h}$) and value ($wT_q^{(l)} \in \mathbb{R}^{D^h \times D^h}$).

First, we compute the following three projections as below:

$$qT_i^{(l)} = wT_q^{(l)} H_{T,i}^{(l)} \in \mathbb{R}^{T \times D^h}$$
$$kT_i^{(l)} = wT_k^{(l)} H_{T,i}^{(l)} \in \mathbb{R}^{T \times D^h}$$
$$vT_i^{(l)} = wT_v^{(l)} H_{T,i}^{(l)} \in \mathbb{R}^{T \times D^h}$$

Then, we compute the scaled dot product between the query and the key projections like in the spatial attention layer.

$$\beta_i = softmax(\frac{(qT_i^{(l)})^T.kT_i^{(l)}}{\sqrt{D^h}}) \in \mathbb{R}^{T \times T} \tag{7}$$

Here, $\beta_{i,t_a,t_b} \in \beta_i$, represents the relevance of the node features at time t_b to the node features at t_a for node i. In contrast to the spatial attention, we do not use the masking and assume all time steps are connected. The temporal attention layer outputs $H_{T,i}^{(l+1)}$, for each node as below.

$$H_{T,i}^{(l+1)} = \beta_t.vS_i^{(l)} \in \mathbb{R}^{T \times D^h} \tag{8}$$

By stacking $H_{T,i}^{(l+1)}$ for nodes that are changing over time, we compute $H_T^{(l+1)}$ as below.

$$H_T^{(l+1)} = \|_{i=1,i\notin\mathbb{U}}^N H_{T,i}^{(l+1)} \tag{9}$$

Finally, the outputs from the temporal attention layer and the fully connected layer are concatenated to get the final temporal embedding representation.

$$H_{TF}^{(l+1)} = H_T^{(l+1)} \| H_F^{(l+1)} \in \mathbb{R}^{N \times T \times D^h} \tag{10}$$

Remark: The factorization of the temporal attention layer with the fully connected layer can be extended to multiple parallel layers as well. Suppose one knows how the nodes evolve over time according to some underlying distributions, then more than one temporal attention layer and more than one fully connected layer can be applied. Of course, how one may gain such knowledge on the existence of different distributions and the decision on the number of layers is beyond the scope of this paper and may be investigated in future research.

Multi-head Attention: As suggested by Vaswani et al.[17], using more than one attention head (a.k.a. multi-head attention) improves the stabilization of the learning process. In multi-head attention, D^k and D^v defined above will be equal to D^h/M, where M is the number of attention heads. The final output will contain M number of embeddings with a dimension of $\mathbb{R}^{N \times D^h/M}$ and these M embeddings can be concatenated or projected through another weight vector to get a representation of shape $\mathbb{R}^{N \times D^h}$. Both the temporal and spatial attention layers use the multi-head attention mechanism.

Integration Layer: Once we learned the spatial representation ($H_S^{(l+1)}$) and the temporal representation ($H_{TF}^{(l+1)}$), we need to learn how to 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.

$$H^{(l+1)} = \sigma(wI.(H_S^{(l+1)} \| H_{TF}^{(l+1)})) \in \mathbb{R}^{T \times N \times D^h} \tag{11}$$

Finally, $H^{(l+1)}$ 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* [29], 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 the previous and the future time variants of node i.

4.1.2 Temporally Pointing Decoder

This section describes the decision making process of the decoding layer of GTA-RL, which takes as input the embedded information from the encoder output. In literature, the decoding layers in static combinatorial optimization problems have used *pointer networks*[15] or a combination of multi-head and single-head attention [13] to select the next node in a solution. Since the combination of multi-head and single-head attention has traditionally yielded better results and better convergence rate over the *pointer networks*[13], we follow a similar multi-head and single-head attention architecture, and extend it with components that handle temporal features.

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. Then, 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 embedding representation. This masking is problem-dependent[3]. Third, the temporal pointer also outputs a context embedding by considering the current decoded solution and the current

---

[3]For example, in dynamic TSP, already visited nodes from previous time steps are invalid nodes.

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.

Temporal Pointer: First, we describe the motivation behind the temporally pointing decoder. In the original multi-head and single-head attention architecture [13], a set of fixed attention values are computed from the encoder output (where the encoder output contains a shape of $\mathbb{R}^{N \times D^h}$). These fixed attention values are used in every decoding time step. Even though the fixed attention works in a static problem, we identify that use of fixed attention is the main limitation for handling dynamic information because at every decoding time step, the node features change in dynamic combinatorial 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 (problem) 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, which results in a shape of $\mathbb{R}^{T \times N \times D^h}$. A naive way to handle the temporal axis is to compute a mean value for each node over the entire time horizon as below.

$$H_{naive} = \frac{\sum_{t=1}^{T} H^{(L)}[t, :, :]}{T} \in \mathbb{R}^{N \times D^h} \tag{12}$$

where $H^{(L)}[t, :, :]$ represents the embedding for all the nodes at time t. Another variant is to use only a single time-step data such as $H^{(L)}[0, :, :]$. Both of these can then be given as an input to the multi-head and single-head attention architecture to decode node selection as we demonstrate in our experiment in Section 6.2. However, the use of H_{naive} or $H^{(L)}[0, :, :]$ 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 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 compute the multi-head attention based on new values computed at the current time step. We achieve superior results using this technique because now we can retain information about every time step.

Context Embedding: The context embedding is used to identify the context of a given problem at a decoding time step. This is obviously problem-dependent. For the context embedding, all the context embeddings are updated based on the current node representation from the encoder output $H^{(L)}$.

To generalize the context embedding to all the combinatorial problems, we define a function named context embedding function $f_{cnxt} : \mathbb{R}^{N \times T \times N \times D^h} \mapsto \mathbb{R}^{K \times D^h + e}$. The value of K and e will be problem dependent. $H_{C,t} = f_{cnxt}(H(L)_t, Y_t)$ where $H_{C,t}$ is the context embedding. Refer to Section 5.1 for derivation of $H_{C,t}$ for different dynamic combinatorial problems.

Multi-head and Log Probability Layer: Once we have $H_{C,t}$ and $H_{D,t}$, we use the multi-head attention layer to combine the context-specific embedding with the current graph representation. Similar to the encoder, we use query $wC_q (\in \mathbb{R}^{K*D^h + e \times D^h})$, key wC_k and value wC_v weights in the attention layer. In contrast to the self-attention, $H_{C,t}$ for query and $H_{D,t}$ for both the key and value parameters is used.

$$qC_t = wC_q H_{C,t} \in \mathbb{R}^{N \times D^h}$$
$$kC_t = wC_k H_{D,t} \in \mathbb{R}^{N \times D^h}$$
$$vC_t = wC_v H_{D,t} \in \mathbb{R}^{N \times D^h}$$

Next, similar to Equation 4 and Equation 8, we can compute the attention weights and then the final embedding $H_{D,t}^{(F)}$.

$$H_{D,t}^{(F)} = softmax \frac{(qC_t^T kC_t)}{D^h} . vC_t \tag{13}$$

9

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 Bello *et al.*[24].

$$\gamma_t = tanh(H_{D,t}^{(F)}.(wP.H_{D,t})^T)$$ (14)

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.

$$P_t = softmax(\gamma_t) \in \mathbb{R}^{\mathbb{N}}$$ (15)

4.2 GTA-RL for Real-time Graph Combinatorial Problems:

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. They are only available once we take the action.

To handle this type of problems, we propose the encoder to be iterative. Next, we describe this iterative method. At the start of the problem, we only take the first time step input features, that is $x_1 \in \mathbb{R}^D$ (defined in Section 3). The input feature X_1 does not have any temporal axis as this is the node information about a single time step. First, 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 follow the same approach until we find the complete solution.

In this proposed iterative method, we do not need to take the input features for the entire time horizon. However, by buffering the previous input, 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 initially before converging to a satisfactory solution. Also, compared to our original implementation, the encoder now encodes the input multiple times, thus increasing the computational requirements. This is an acceptable trade-off as real-time combinatorial problems are more difficult to optimize compared to dynamic combinatorial problems.

4.3 Training Graph Temporal Attention with Reinforcement Learning

As we discussed in the introduction, the policy-gradient-based methods achieve better results in combinatorial optimizations. Thus, we use, REINFORCE algorithm, a policy-gradient-based algorithm [18] to train the Graph Temporal Attention network. First, we define the performance measure of dynamic combinatorial optimization (G) using objective function P_{obj} (defined in Section 3) and using Equation 2.

$$J(\theta|G) = \mathbb{E}_{Y \sim \pi_\theta} [P_{obj}(Y|G)]$$ (16)

Then, we can use the *policy gradient theorem* to find the derivative of $J(\theta|G)$.

$$\nabla_\theta J(\theta|G) = \mathbb{E}_{Y \sim \pi_\theta} [P_{obj}(Y|G)\nabla_\theta log(\pi_\theta(Y|G))]$$ (17)

This derivative can be used to update parameters of GTA-RL. However, due to the high variance results in Equation 17, a baseline is used to accelerate the learning. Then, Equation 17 can be rewritten as:

$$\nabla_\theta J(\theta|G) = \mathbb{E}_{Y \sim \pi_\theta} [(P_{obj}(Y|G) - b(G))\nabla_\theta log(\pi_\theta(Y|G))]$$ (18)

where $b(G)$ is called the baseline function, which is a function that is independent of Y.

Once we established Equation 18, we need to choose a proper baseline function. There are two baselines that have been used in this context; critic baseline [14] and rollout baseline [13]. We experimented with both these baseline functions and found that using rollout baseline works better than critic baseline for dynamic combinatorial optimization.

5 Experimental Setup

We evaluate GTA-RL on two routing-based combinatorial optimization problems named *Travelling Salesman Problem (TSP)* and *Vehicle Routing Problem (VRP)*. In this section, we first describe these two problems, then we discuss our experimental environment, and the parameters and the baselines used for our experimental evaluation.

5.1 Problems

Dynamic Travelling Salesman Problem: Similar to the static combinatorial problem, there are n number of points (nodes) in the euclidean space, and the objective is to find the order of nodes to visit to cover all the nodes such that the total traveled distance is minimal. In dynamic TSP, the node locations are changing uniformly at random in euclidean space, i.e., x,y coordinates are changing uniformly at random at every time step[4]. In this way, the cost of traveling between two nodes are changing over time in the euclidean space. This is similar to the travel time changes in a transportation network due to external traffic loads. Also, changing node's x,y coordinates means we are updating the node features. Therefore, this scenario can be easily generalized to any problem where node features change over time such as in telecommunication networks or social networks.

In previous works [11, 13, 14], for static combinatorial problems, training and testing were done using networks with 20, 50, and 100 nodes. However, due to the added computational complexity in dynamic problems, a similar number of possible solutions is generated by a smaller number of nodes. To illustrate, static TSP with 100 nodes contains $100!(= 9.332622e + 157)$ possible solutions and dynamic TSP with 50 nodes contains $(50!)^2(= 9.250171e + 128)$ possible solutions, i.e., yields the same complexity with half the number of nodes. Thus, in this paper, we test dynamic TSP with 10, 20, 50 nodes, and denote them as TSP10, TSP20, and TSP50, respectively. For sensitivity analysis experiments in Section 6.2, we select 20 as the default value.

We introduced a problem-dependent context embedding H_C in Section 4.1.2 under **Context Embedding** subsection. For travelling salesmen problem, H_C contains 3 elements; the first selected node, the last selected node and the graph embedding computed by summing up all nodes at time t as below.

$$\text{TSP: } H_{C,t} = \{h_{y_0}^{(L)}||h_{y_t}^{(L)}||H_{G,t}\}; H_{G,t} = \sum_{i=0}^{N} h_{t,i}^{(L)} \tag{19}$$

Vehicle Routing Problem: The vehicle routing problem is one of the most challenging combinatorial optimization problems. The conventional vehicle routing problem contains a set of n number of points in euclidean space and a vehicle with some 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 capacity. When visiting a node other than the depot, the vehicle should have enough goods to satisfy the demand of that particular node. Similar to the previous 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. Finally, the objective is to find the minimum distance the vehicle needs to travel to satisfy the demands of all the nodes. Similar to TSP, we run experiments with VRP consisting of 10, 20, and 50 nodes. These are denoted as VRP10, VRP20, and VRP50, respectively.

For the vehicle routing problem, the context embedding is the last selected node embedding plus the remaining capacity of the vehicle, and the graph embedding is the same as in TSP.

$$\text{VRP: } H_{C,t} = \{h_{y_t}^{(L)}||r||H_{G,t}\} \tag{20}$$

where r is the remaining capacity of the vehicle.

Real-time Versions: We use the same formulation in real-time TSP similar to dynamic TSP; however, only the node locations at that time step is provided which is similar to the real-world ride-sharing scenario described in the introduction where the future traffic congestion information is not available. Once a node is selected, we reach that node and the next node to visit is selected, thus in a real-time setting order of the visited nodes cannot be updated even if we found a better order of nodes in a later time step.The same principle applies to real-time VRP as well.

---

[4]The initial node locations are assigned uniformly at random between 0 and 1 in 2d-space. Then, at every time step, the node locations are updated uniformly at random with maximum change of 0.1 in coordinates.

5.2 Parameter Settings

We now detail the hyper-parameters used in our experimental setup. In GTA-RL, we use three layers of the temporal encoder, and the hidden dimension D^h is set to 128. We use 12800 problem instances during one epoch, and the batch is set to 32 instances. A total of 50 epochs is used for the training. We use a learning rate of $1e^{-4}$ for both the actor and the critic of the GTA-RL. We use Adam optimizer [32] for the training of the GTA-RL. 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. Our code base is built on top of the following code base[5] using Pytorch and can be accessed here[6].

5.3 Baselines

We use several baselines for the comparison. The baselines include learning heuristics, hand-crafted heuristics, and optimal solutions computed through Integer Programming[7].

- **S2V-DQN:** This solution was proposed by Dai *et al.* [11]. S2V-DQN uses structure2vec for graph encoding and uses fitted Q-learning [25] for the decision making, which belongs to value-based methods as described in Section 2. S2V-DQN is implemented in a way that a node in the problem instance can be selected only once. Therefore, S2V-DQN cannot be used for VRP, since VRP depot node is visited multiple times. In TSP, there are two variants of S2V-DQN depending on how the reward function is formulated. The first method is to provide a negative reward for increased distance by adding the last selected node to the end of the tour of TSP (S2V-DQN-last). The second is to provide a negative reward for increased distance by adding the last selected to the middle of the tour where the resulting distance is minimum (S2V-DQN-sorted). S2V-DQN-sorted improves the solution quality significantly, but we should point out that S2V-DQN-sorted cannot be used in a real-time setting where the tour cannot be changed once selected.

- **RNN-RL:** This solution was proposed by Nazari *et al.*[15]. RNN-RL uses two recurrent networks with REINFORCE algorithm and belongs to a class of policy-gradient base algorithms. RNN-RL contains two encoders named static and dynamic. In the original implementation coordinates of nodes were provided for the static encoder. We provided coordinates of nodes to the dynamic encoder as the coordinates are no longer static and change over time.

- **AM:** This solution was proposed by Koot *et al.*[13]. The architecture is limited to handle a fixed set of nodes, and we use the initial locations of nodes as an input to the AM model for dynamic TSP and VRP cases.

- **AM-D:** This solution is an extension of **AM**, with a dynamic encoder used to solve VRP [27].

- **Gurobi:** We implement integer programming solutions for both dynamic TSP and dynamic VRP using the Gurobi[8] python library. The integer programming approach is able to find the optimal solutions for both of the problems. Computational times for dynamic TSP and VRP are however significantly higher than their static counterparts. For instance, 900 seconds for dynamic TSP size 20 vs around 2.5 seconds for static TSP with 20 nodes. Thus, we were only able to compute gurobi optimization for TSP20 and VRP10. We should also note that specialized solvers such as concorde[9] cannot be used in our comparisons as these do not support dynamic combinatorial optimizations.

- **Nearest Neighbor(NN):** We also implement a hand-crafted heuristic named Nearest Neighbor(NN) for dynamic TSP problem. Here, the next node to visit at the next time step is selected based on the distance to surrounding nodes, favouring closest nodes first. The original NN is developed for static TSP, and we extend it to a version of NN that can handle the dynamic TSP. In extended NN, we compute the nearest neighbors for a given node at the current time step using the coordinates of the neighboring nodes at the next time step.

5.4 GTA-RL Variations

There are several variations of GTA-RL that we use for the evaluation. We use this section to name these variations formally.

---

[5]https://github.com/wouterkool/attention-learn-to-route

[6]https://github.com/udeshmg/GTA-RL

[7]Note that we are not using the algorithm proposed by Barrett *et al.*[12] in our comparison because their implementation only supports Max-cut problem, but not for TSP or VRP.
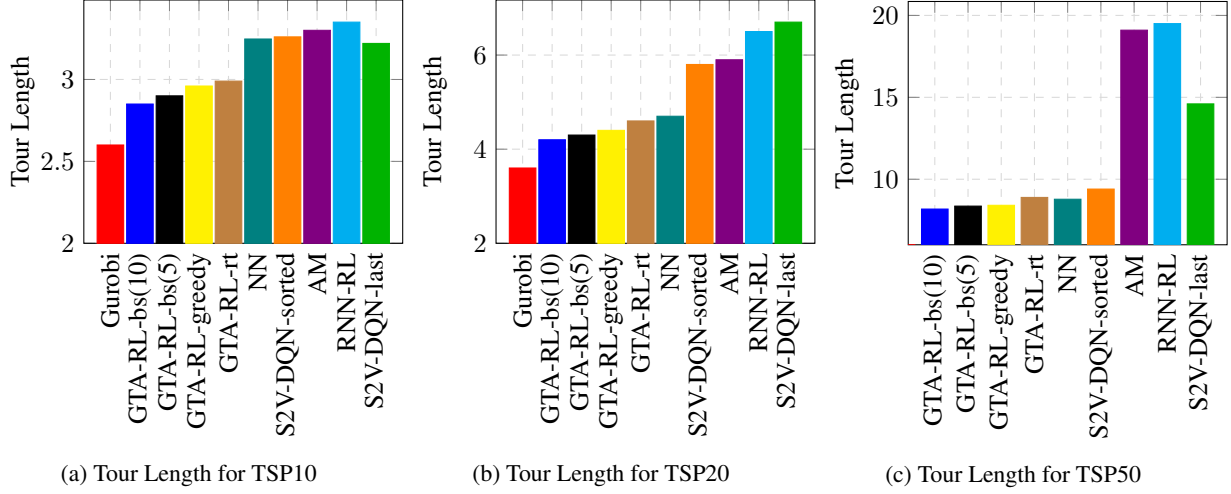
[8]https://www.gurobi.com

[9]https://www.math.uwaterloo.ca/tsp/concorde.html

(a) Tour Length for TSP10   (b) Tour Length for TSP20   (c) Tour Length for TSP50

Figure 3: Tour length for dynamic TSP

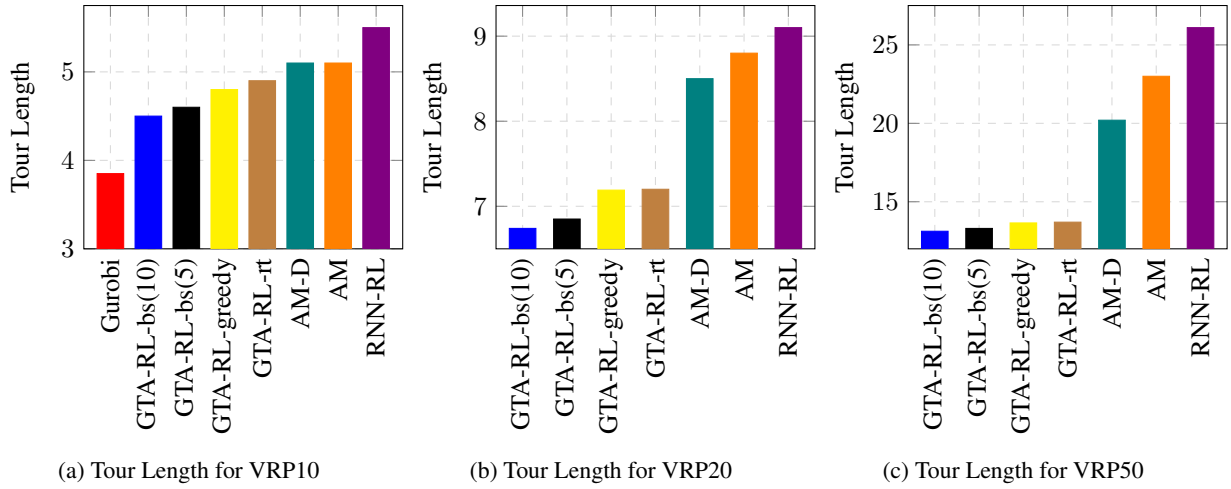(a) Tour Length for VRP10   (b) Tour Length for VRP20   (c) Tour Length for VRP50

Figure 4: Tour length for dynamic VRP

- **GTA-RL-greedy:** This is the standard GTA-RL, and it uses the greedy decoding in the decoder, where the node with the highest probability is selected as the next node.

- **GTA-RL-bs:** . Unlike GTA-RL-greedy, GTA-RL-bs uses the *beam search* technique during the decoding, where nodes with (k) number of highest probabilities are kept at every time step and decoding in the next time step is done with respect to previously-stored k values. Finally, the sequence of nodes with the highest probability according to Equation 2 is selected. We should also note that GTA-RL-bs cannot be used in a real-time setting as the order of nodes changes.

- **GTA-RL-sum** We briefly discussed a naive implementation of temporal decoder in Section 4.1.2. The method was to sum up the encoding over the time axis of the embedding given by the encoder. GTA-RL-sum uses this technique.

- **GTA-RL-(0)** Similar to GTA-RL-sum, in this algorithm we only use the first time step of the embedding given by the encoder.

- **GTA-RL-rt** This is the version of GTA-RL that is capable of handling real-time data.

6   Experimental Results

We now compare the results of GTA-RL-variations against the baselines. First, we present the results against the baselines for dynamic TSP and dynamic VRP, and then we present results for real-time TSP and real-time VRP.

Following that, we compare GTA-RL-variations to reason about our architecture. Finally, we present the solutions achieved by GTA-RL compared to the optimal solution computed by Gurobi.

6.1 Comparison for Dynamic TSP and Dynamic VRP

Dynamic TSP20: Figure 3b shows the average tour length for the validation data set achieved by each algorithm with 20 nodes (TSP20). The learning-based algorithms, S2V-DQN-last, RNN-RL, AM, are not able to find a satisfactory solution as the average tour length from these algorithms is between 5.8 to 6.3. The results demonstrate that these algorithms are unsuitable for dynamic combinatorial optimization despite them being applicable to the static settings. S2V-DQN-sorted however finds a much lower average 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, S2V-DQN-sorted can shuffle the order of nodes to reduce the tour length. This shuffle however is impossible in a real-time setting where the selected node cannot be updated. The non-learning-based solution, NN, is able to find a reasonable solution as the modified NN can handle the dynamic nature of combinatorial optimization.

GTA-RL-greedy beats all the other baseline algorithms, including hand-crafted heuristic (NN). In GTA-RL, the action selection is based on the highest probability at that time step. However, employing a beam search strategy, GTA-RL-bs based algorithm finds a lower average tour length. This reduction is expected as with beam search we select the final solution by looking at k (k is the beam width) final possible outcomes. The beamwidth of 10, GTA-RL-bs(10), achieves the lowest average tour length[10]We should also highlight that the GTA-RL-greedy solution is not far from GTA-RL-bs(10), indicating that we can produce a high-quality solution even with the greedy strategy.

One could argue that GTA-RL can have an unfair advantage over the value-based approaches such as S2V-DQN-last. This is because the S2V-DQN also 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. GTA-RL-rt considers the problem as a real-time problem and does not take future inputs while computing the solution. Thus, GTA-RL-rt and S2V-DQN-last work in the same setting. However, as depicted in Figure 3b, GTA-RL-rt is able to find a much shorter tour length compared to S2V-DQN-last. This highlights the effectiveness and flexibility of our approach.

By comparing the optimal controller, Gurobi, with the best learning-based algorithm from Figure 3b, the results show that GTA-RL-bs(10) is quite close to the optimal solution, with only 10% gap between the Gurobi optimal solution's average tour length and GTA-RL-bs(10). As explained before, we believe that the complexity of dynamic combinatorial problems makes it much harder to find a heuristic solution compared to the static scenario, making 10% gap acceptable. Also, our objective is not to beat an optimal optimizer such as Gurobi, but to propose a heuristic for *efficiently* solving the dynamic combinatorial optimization problems. For example, Gurobi optimization takes around 900s to solve one instance of TSP20 while GTA-RL ran with the same computational resources only takes around *0.45 seconds* (the inference time) with beam search and *0.39 seconds* with the greedy strategy to solve a batch of 100 problem instances. The inference time of GTA-RL and other learning baselines are similar (in between 0.3-0.5 seconds).

Dynamic TSP (10,50): Compared to Figure 3b, we observe a similar trend in Figure 3a and 3c. In Figure 3a, the learning-based algorithms (S2V-DQN-last, RNN-RL, AM) are able to find a reasonable solution, however, their performance degrades severely as we increase the number of nodes, as evident from Figure 3c. This is because when the number of nodes increases, the tour length is longer, and the possibility of node location changes is higher. Due to the increase in dynamicity of node locations, S2V-DQN-last, RNN-RL, AM are not able to find good solutions for larger graphs.

We achieve a similar optimal gap (of 8%) in dynamic TSP10, demonstrating that the number of nodes does not affect much the optimal gap. Note however that we were unable to compute the optimal solution for dynamic TSP50 using the Gurobi optimization, due to its computational complexity, which reduces its applicability to larger real-life graphs.
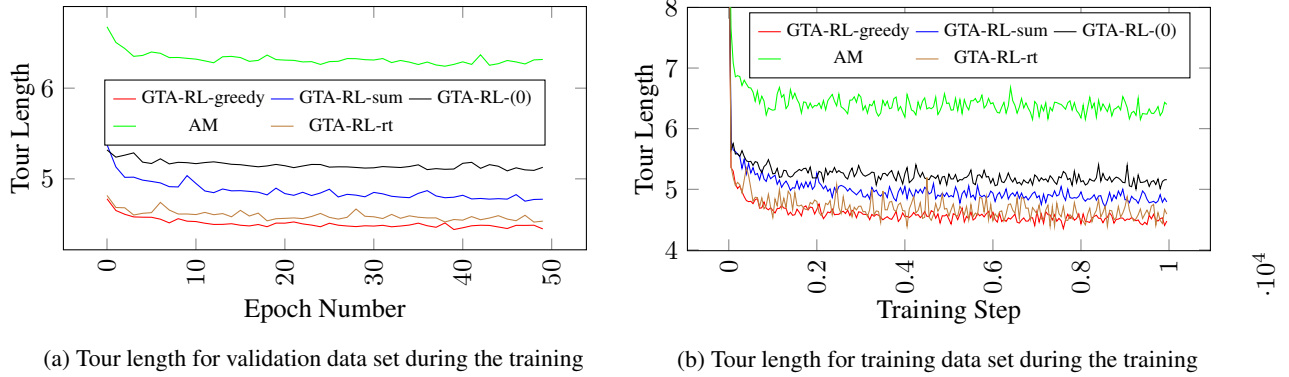
Dynamic VRP (10,20,50): For Dynamic VRP, we observe a similar trend to TSP discussed in the previous section [11]. In Figure 4a, 4b and 4c[12], GTA-RL-bs(10) finds the shortest tour length compared to the baselines and other GTA-RL variations. All the GTA-RL variations outperform the other learning-based 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 of TSP, being slightly higher due to the increased complexity of VRP problem.

---

[10]Note however that the use of beam width for real-time applications is not possible, since GTA-RL-bs(10) needs to find the whole tour before selecting the best possible tour.

[11]Note that there is a slight performance increase in AM-D compared to the AM algorithm due to the dynamic encoder.

[12]We use pre-trained weights from dynamic VRP20 while training the GTA-RL and AM-D for VRP50

(a) Tour length for validation data set during the training   (b) Tour length for training data set during the training

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

Table 2: Tour length of Dynamic TSP50 and Dynamic VRP50 from GTA-RL trained with networks with 20 and 50 nodes. Beam width of 10 is used.

6.2 GTA-RL Variations Comparison

In this section, we first demonstrate the impact of our temporal encoder and temporal pointer. Figure 5a shows the average tour length achieved by each variation of GTA-RL solutions after each training epoch. We also show the AM approach on the graph, which is somewhat similar to our approach but does not use any temporal components.

In Figure 5a, all the variations of GTA-RL achieve a lower tour length compared to the AM solution mainly because of the temporal encoder's ability to encode the temporal information. Next, even though GTA-RL-greedy, GTA-RL-sum, GTA-RL-(0) all use the same encoder, the decoder is different. 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 not able to achieve good performance compared to GTA-RL-greedy, where the temporal decoder is used, which shows the benefit of using the temporal decoder. We should also note that GTA-RL-(0) is performing somewhat better compared to GTA-RL-sum because, in GTA-RL-sum, all the encoder output is summed up in the temporal axis, which loses information about dynamic changes. However, in GTA-RL-(0), even though it uses a single time step data, GTA-RL-(0) will preserve temporal data up to a certain extent.

Next, we use both Figure 5a and Figure 5b to show how close the GTA-RL-rt and GTA-RL-greedy solutions are. In Figure 5a, compared with the other solutions, GTA-RL-rt achieves similar performance to GTA-RL-greedy. This demonstrates that even in real-time settings, our proposed method can find a high-quality solution. As shown in Figure 5b, the tour length during the training of GTA-RL-rt is also much closer to GTA-RL-greedy compared to other solutions. However, it is interesting to see that there is a variation of GTA-RL-rt that is higher than what is found by GTA-RL-greedy. This is mainly because GTA-RL-rt, at its first-time step, takes the decision only based on the current time step node locations. Since node features are changing uniformly at random, GTA-RL-rt cannot guarantee that the selected node will be the best decision given that node features can change externally in future. This randomness can cause a volatility of the tour length. Despite these, the average tour of GTA-RL-rt is able to be close to GTA-RL-greedy.

6.3 Generalization

In this section, we show how GTA-RL model can generalize to larger graphs beyond those used for training. To find such a generalization, 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 2. In both situations (in dynamic TSP50 and in dynamic 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.

15

(a) AM with a tour length of 3.85   (b) GTA-RL-greedy with a tour length of (c) Gurobi solution with a tour length 2.96
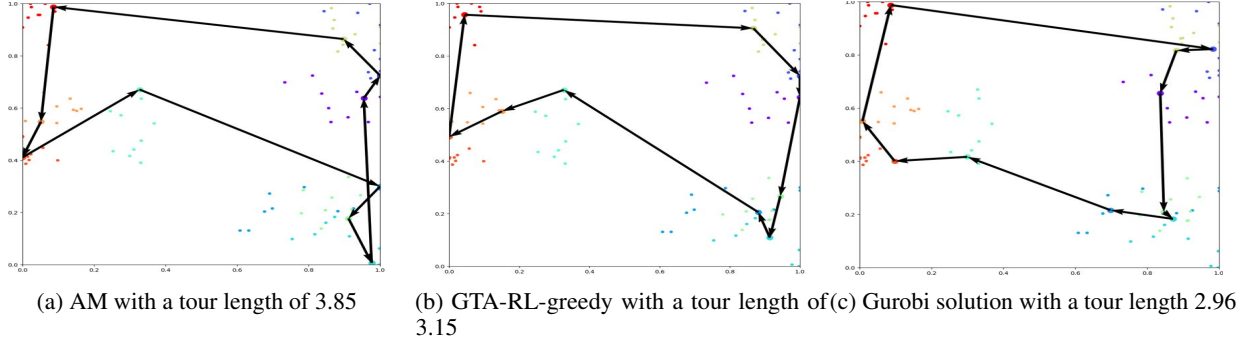3.15

Figure 6: Figure shows the order of selected nodes by AM, GTA-RL-greedy and Gurobi optimal solver. For easy visualization we use dynamic TSP10 where only 10 cities are present. The dots with the same color indicates the same city locations at different time steps. The black arrows shows the tour decision taken by each algorithm.

6.4 Visualizing the Dynamic TSP

In this experiment, we visualize the tours computed by AM, GTA-RL-greedy and Gurobi optimal solver for dynamic TSP 10 in Figure 6. The set of nodes with the same color represents the locations of the same node in different time steps. Figure 6a shows the tour output from AM is 3.86. In Figure 6a, in the bottom right corner, AM is not able to find the exact order for traversal as nodes which are further were visited before the closer ones. GTA-RL-greedy (presented in Figure 6b) is able to avoid that and select the order of nodes to achieve a shorter tour length of 3.15, which is much closer to the optimal solution found by Gurobi presented in Figure 6c.

7 Conclusion

In this work, we propose an effective and efficient learning-based method to solve dynamic graph combinatorial optimization problems, which is crucial when applying combinatorial optimization to real-world applications. Our proposed temporal encoder-decoder architecture is able solve dynamic graph problems by first, learning a representation for the given problem instance and second, dynamically focusing on different parts of the learned representation to find a solution sequence to the problem instance. We then introduce a variant of our proposed model to solve real-time combinatorial optimization where input features are known only after making a decision. We evaluate our method by comparing it against several state-of-the-art approaches, both learning-based methods, as well as hand-crafted heuristics, and against the optimal controllers, and show that our approach substantially outperforms both learning-based and hand-crafted heuristics and is on par with the optimal controller when solving famous NP-hard problems (TSP and VRP) in transportation.

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 by buffering the previous states which can be memory expensive and this can be investigated further to store and process data more efficiently. Third, this approach can be applied to many real-world applications such as flow optimization in traffic engineering, determining autonomous truck routes, bandwidth allocations for telecommunications networks and for knowledge graph reasoning in graph databases. Finally, dynamic combinatorial optimization is much harder than static combinatorial optimization; 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 graph combinatorial optimization, we hope to reduce the gap between our proposed approach and the optimal controller in future research.

References

[1] Xue Song Yan, Li Shan Kang, Zhi Hui Cai, and Hui Li. An approach to dynamic traveling salesman problem. In *International Conference on Machine Learning and Cybernetics*, pages 2418–2420, 2004.

[2] Dimitris J. Bertsimas and Garrett van Ryzin. A stochastic and dynamic vehicle routing problem in the euclidean plane. *Operations Research*, 39(4):601–615, 1991.

[3] H. Yuan, G. Li, Z. Bao, and L. Feng. An effective joint prediction model for travel demands and traffic flows. In *International Conference on Data Engineering*, pages 348–359, 2021.

[4] A. K. M. Mustafizur Rahman Khan, Oscar Correa, Egemen Tanin, Lars Kulik, and Kotagiri Ramamohanarao. Ride-sharing is about agreeing on a destination. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2017.

[5] K. F. Chu, A. Y. S. Lam, and V. O. K. Li. Dynamic lane reversal routing and scheduling for connected autonomous vehicles. In *International Smart Cities Conference*, pages 1–6, 2017.

[6] Yi-Chang Chiu, Jon Bottom, Michael Mahut, Alexander Paz, Ramachandran Balakrishna, Travis Waller, and Jim Hicks. Dynamic traffic assignment: A primer. *Dynamic Traffic Assignment: A Primer*, 2011.

[7] Yue Li, S Travis Waller, and Thanasis Ziliaskopoulos. A decomposition scheme for system optimal dynamic traffic assignment models. *Networks and Spatial Economics*, 3(4):441–455, 2003.

[8] Guanjie Zheng, Chang Liu, Hua Wei, Chacha Chen, and Zhenhui Li. Rebuilding city-wide traffic origin destination from road speed data. In *International Conference on Data Engineering*, pages 301–312, 2021.

[9] C.C.F. Fong, J.C.S. Lui, and Man Hon Wong. Quantifying complexity and performance gains of distributed caching in a wireless network environment. In *Proceedings 13th International Conference on Data Engineering*, pages 104–113, 1997.

[10] Sanjeev Arora. The approximability of np-hard problems. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, page 337–348. Association for Computing Machinery, 1998.

[11] Hanjun Dai, Elias Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[12] Thomas Barrett, William Clements, Jakob Foerster, and Alex Lvovsky. Exploratory combinatorial optimization with reinforcement learning. In *AAAI Conference on Artificial Intelligence*, pages 3243–3250, 2020.

[13] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2019.

[14] Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the tsp by policy gradient. In Willem-Jan van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 170–181, 2018.

[15] Mohammadreza Nazari, Afshin Oroojlooy, Martin Takáč, and Lawrence V. Snyder. Reinforcement learning for solving the vehicle routing problem. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 9861–9871, 2018.

[16] Christopher JCH Watkins and Peter Dayan. Technical note: Q-learning. *Machine learning*, 8:279–292, 1992.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

[18] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*, volume 135. MIT Press, 1st edition, 1998.

[19] Alex Hall, Steffen Hippler, and Martin Skutella. Multicommodity flows over time: Efficient algorithms and complexity. *Theoretical Computer Science*, 379(3):387–404, 2007.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[21] Udesh Gunarathna, Hairuo Xie, Egemen Tanin, Shanika Karunasekara, and Renata Borovica-Gajic. Real-time lane configuration with coordinated reinforcement learning. In *Machine Learning and Knowledge Discovery in Databases*, pages 291–307, Cham, 2021.

[22] Hua Wei Chach Che, Nan Xu, Guanjie Zheng, Ming Yang, Yuanhao Xiong, Kai Xu, and Zhenhui Li. Toward a thousand lights: Decentralized deep reinforcement learning for large-scale traffic signal control. *AAAI Conference on Artificial Intelligence*, 2019.

[23] Marvin Zhang, Zoe McCarthy, Chelsea Finn, Sergey Levine, and Pieter Abbeel. Learning deep neural network policies with continuous memory states. In *IEEE international conference on robotics and automation*, pages 520–527, 2016.

[24] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *arXiv preprint arXiv:1611.09940*, 2016.

[25] Martin Riedmiller. Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method. In *Machine Learning and Knowledge Discovery in Databases*, pages 317–328, 2005.

[26] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, volume 28, 2015.

[27] Bo Peng, Jiahai Wang, and Zizhen Zhang. A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems. In *Artificial Intelligence Algorithms and Applications*, pages 636–650, 2020.

[28] Iddo Drori, Anant Kharkar, William R. Sickinger, Brandon Kates, Qiang Ma, Suwen Ge, Eden Dolev, Brenda Dietrich, David P. Williamson, and Madeleine Udell. Learning to solve combinatorial optimization problems on real-world graphs in linear time. In *IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 19–24, 2020.

[29] Chuanpan Zheng, Xiaoliang Fan, Cheng Wang, and Jianzhong Qi. Gman: A graph multi-attention network for traffic prediction. In *AAAI Conference on Artificial Intelligence*, pages 1234–1241, 2020.

[30] Tryambak Gangopadhyay, Sin Yong Tan, Zhanhong Jiang, Rui Meng, and Soumik Sarkar. Spatiotemporal attention for multivariate time series prediction and interpretation. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3560–3564, 2021.

[31] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5363–5370, 2020.

[32] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *arXiv preprint arXiv:1412.6980*, 2015.