algorithms

MDPI

Article

DLMinTC+: A Deep Learning Based Algorithm for Minimum Timeline Cover on Temporal Graphs

Giorgio Lazzarinetti [1,*], Riccardo Dondi [2], Sara Manzoni [1] and Italo Zoppis [1]

[1] Università degli Studi Milano-Bicocca-Dipartimento di Informatica Sistemistica e Comunicazione, 20126 Milano, Italy; sara.manzoni@unimib.it (S.M.); italo.zoppis@unimib.it (I.Z.)
[2] Università degli Studi di Bergamo-Dipartimento di Lettere, Filosofia e Comunicazione, 24129 Bergamo, Italy; riccardo.dondi@unibg.it
* Correspondence: g.lazzarinetti@campus.unimib.it

Abstract: Combinatorial optimization on temporal graphs is critical for summarizing dynamic networks in various fields, including transportation, social networks, and biology. Among these problems, the Minimum Timeline Cover (MinTCover) problem, aimed at identifying minimal activity intervals for representing temporal interactions, remains underexplored in the context of advanced machine learning techniques. Existing heuristic and approximate methods, while effective in certain scenarios, struggle with capturing complex temporal dependencies and scalability in dense, large-scale networks. Addressing this gap, this paper introduces DLMinTC+, a novel deep learning-based algorithm for solving the MinTCover problem. The proposed method integrates Graph Neural Networks for structural embedding, Transformer-based temporal encoding, and Pointer Networks for activity interval selection, coupled with an iterative adjustment algorithm to ensure valid solutions. Key contributions include (i) demonstrating the efficacy of deep learning for temporal combinatorial optimization, achieving superior accuracy and efficiency over state-of-the-art heuristics, and (ii) advancing the analysis of temporal knowledge graphs by incorporating robust, time-sensitive embeddings. Extensive evaluations on synthetic and real-world datasets highlight DLMinTC+'s ability to achieve significant coverage size reduction while maintaining generalization, offering a scalable and precise solution for complex temporal networks.

Keywords: deep learning; minimum timeline cover; combinatorial optimization; temporal graph

Academic Editors: Mohammad Shokouhifar and Frank Werner

Received: 20 December 2024
Revised: 11 February 2025
Accepted: 13 February 2025
Published: 17 February 2025

Citation: Lazzarinetti, G.; Dondi, R.; Manzoni, S.; Zoppis, I. DLMinTC+: A Deep Learning Based Algorithm for Minimum Timeline Cover on Temporal Graphs. *Algorithms* **2025**, *18*, 113. https://doi.org/10.3390/a18020113

1. Introduction

Combinatorial optimization (CO) on temporal graphs is a rapidly evolving area of research focusing on extending classical combinatorial problems to dynamic settings [1,2]. Traditional approaches often handle these problems in static contexts, where the attributes of the graph remain unchanged [3]. In contrast, real-world applications, such as transportation, social networks, biological and telecommunication networks, require consideration of evolving features over time, adding substantial complexity to the optimization process [4,5].

Among these problems, the Minimum Timeline Cover (MinTCover) has recently gained the attention of researchers, since it represents a particular case of network summarization tasks that involves identifying latent activity intervals for all entities, producing an activity timeline that encompasses the entire network [6–11]. The MinTCover problem has been introduced with the goal of identifying crucial time intervals that elucidate significant network events, thus overcoming the main limitations of other temporal network summarization solutions, which, while effective, can be complex and difficult to interpret [12–17].

To simplify, research has moved towards the use of activity time intervals to represent interactions between entities [6].

As an example, consider how we can identify emerging skills required by companies through the analysis of job posting graphs. Job postings can be represented as graphs where nodes correspond to skills, and edges connect skills that co-occur within the same posting. By aggregating job postings over time for a specific position, we can construct a temporal graph of skills. For instance, the launch of ChatGPT in November 2022 rapidly transformed the workflows of data scientists in machine learning projects, prompting companies to seek professionals with expertise in Generative AI. Figure 1 illustrates this process: on the left, a co-occurrence graph depicts skills as vertices, with edges representing their co-appearance in job postings over time. On the right, a temporal network model visualizes these interactions, showing how timelines of (entity, time-interval) pairs can provide rich insights into significant events. These timelines, highlighted in blue, capture pivotal transitions, such as the initial focus on traditional machine learning skills and the subsequent demand for Generative AI expertise, underscoring the central role of skills in this context. This approach to mapping event-driven timelines is fundamental to addressing the MinTCover problem.

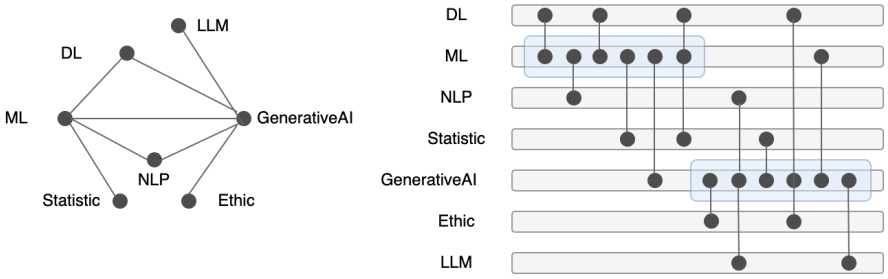

Figure 1. Illustration of a co-occurrence graph (**left**) and a temporal network framework (**right**), highlighting the relationship between activity timelines and dynamic interactions. The co-occurrence graph shows static connections between entities (black circles) based on interactions, while the temporal network visualizes evolving interactions over time. Activity timelines (blue) demonstrate the concept of identifying minimal intervals that summarize significant temporal interactions, which is central to the MinTCover problem. In the toy example shown, the analysis of skills coming from job postings shows the initial interest of companies in traditional machine learning replaced by the subsequent demand of Generative AI expertise.

Despite the interpretability and effectiveness of this problem, studies on the computational and approximation complexity highlight that MinTCover is NP-hard [6–9].

For this problem, some approximate and heuristic-based approaches have been proposed [6,10,11,18] which have different performances based on the characteristics of network considered (e.g., density, number of timestamps, number of nodes). However, no deep learning (DL)-based approaches have been proposed in the literature, despite the proven effectiveness of these solutions in dealing with vertex cover problems. This is also due to the complexity of modeling information on a temporal knowledge graph in a representation learning fashion, since algorithms need to take into account both structural and temporal patterns.

Hence, in this paper, we present a DL-based algorithm for solving the MinTCover problem with two main goals: (i) to design a solution which can compute better results with respect to approximate or heuristic based solutions, thus providing evidence of the effectiveness of DL-based approaches for CO over temporal graphs; (ii) to explore the potential of DL algorithms for the representation and embedding of temporal knowledge graphs, aiming to provide novel insights into how advanced DL techniques can capture and analyze temporal dynamics within knowledge graphs, thereby underscoring the power and

versatility of these approaches in integrating evolutionary and time-sensitive information. We, thus, propose an approach based on the combination of Graph Neural Networks (GNNs), Transformer, and Pointer Networks (PNs), enabling a nuanced representation of both the structural and temporal dynamics of nodes in temporal graphs, to build an initial solution to the problem. However, since DL by itself is not enough to grant that the solution is valid, the proposed approach also leverages a novel algorithm for adjusting the computed solution and providing theoretical guarantees on the results.

The rest of the paper is organized as follows: Section 2 formulates the MinTCover problem, defining key concepts and the theoretical background necessary for understanding our approach. Section 3 reviews related works, including heuristic and approximate methods as well as recent DL approaches for CO on temporal graphs. In Section 4, we introduce the DLMINTC+ methodology, detailing the DL components of our model and the iterative adjustment algorithm. In Appendix A, we provide the mathematical details of the DL algorithms adopted. Section 5 presents an extensive experimental evaluation, comparing DLMINTC+ to baseline approaches across synthetic and real-world datasets, with a focus on coverage precision and computational efficiency. Finally, Section 6 concludes the paper, summarizing our findings and discussing potential directions for future research.

2. Problem Formulation

Let $G = (V, E)$ be a temporal graph, where V represents the set of nodes and E denotes the temporal edges. Each edge is characterized by a triple $(u, v, t) \in E$, with $(u, v) \in V$ and $t \in T$ indicating the timestamp at which the interaction between u and v occurs. We focus on undirected and unweighted temporal graphs.

For any vertex $u \in V$, we define $E(u) = \{(u, v, t) \in E\}$ as the set of temporal edges incident to u. Similarly, $N(u, t) = \{v \mid (u, v, t) \in E\}$ represents the set of neighbors of u at time t, while $T(u) = \{t \mid (u, v, t) \in E\}$ captures the set of timestamps where u participates in an edge.

The degree of a vertex u at timestamp t is defined as $degree((u, t)) = |N(u, t)|$, which quantifies the number of temporal edges involving u at time t. We define the *density* of a temporal graph as the ratio of existing edges to the maximal possible number of edges, expressed as $d = \frac{2|E|}{|V|(|V|-1)|T|}$. A graph is regarded as sparse if the number of edges satisfies $|E| = O(|V| + |T|)$.

Given two values s_u and e_u, where $s_u \leq e_u$, we define the activity interval of vertex u as $I_u = [s_u, e_u]$. The set of such intervals for all vertices, denoted as $\mathcal{T} = \{I_u\}_{u \in V}$, forms an activity timeline of G. The *span* of an interval I_u is given by $\delta(I_u) = e_u - s_u$. We say that a timeline $\mathcal{T} = \{I_u\}_{u \in V}$ covers a temporal graph $G = (V, E)$ if for every edge $(u, v, t) \in E$, the timestamp t belongs to either I_u or I_v.

A naive timeline, defined as $I_u = [\min T(u), \max T(u)]$, provides full coverage but may include unnecessarily extended intervals. The objective is thus to determine a timeline minimizing the sum-span, expressed as $S(\mathcal{T}) = \sum_{u \in V} \delta(I_u)$.

Problem 1 (MINTCOVER$_+$)**.** *Given a temporal graph $G = (V, E)$, we identify a timeline $\mathcal{T} = \{I_u\}_{u \in V}$ that covers G while minimizing the sum-span $S(\mathcal{T})$.*

With the above definition, we can state our goal, which requires us to define the problem as follows. Given a temporal graph $G = (V, E)$, where V is the set of vertices and E is the set of temporal edges, we find a parametric function f that maps the input temporal graph G to the set of activity intervals $\mathcal{T} = \{I_u\}_{u \in V}$, denoted as $f(G; \theta) = \mathcal{T}$, where θ represents the parameters of the red function.

Suppose each vertex u can be represented at each timestamp t where it is active as a feature vector P_u^t through a parametric function g. Let us suppose these representations

capture both the temporal and structural properties of the graph relevant to determine the activity intervals, considering the fact that if an interval I_v for node $v \in V$ is part of the activity timeline it influences, the interval I_u for node u should be part of the activity timeline if u and v are adjacent in some timestamp. Thus, for each node, we have a sequence of representations for each timestamp. Let us call these sequences $\mathcal{P} = \{P_u^t\}_{u \in V, t \in T}$ and $g(G; \theta_1) = \mathcal{P}$, the parametric function that maps the input graph G to its representation $\mathcal{P}$, with θ_1 parameters of the function. For each of these sequences, we want to model the probability that a certain node u is active at time t, resulting in an interval I_u that belongs to an activity timeline $\mathcal{T} = \{I_u\}_{u \in V}$. Thus, we need to define another parametric function $f(\mathcal{P}; \theta_2) = \mathcal{T}$, which predicts the activity interval $[s_u, e_u]$ of each node u, using the learned parameters θ_2.

We reformulate the problem as follows.

Problem 2 (Minimum Timeline Cover with Least Squares)**.** *Given a temporal graph $G = (V, E)$, where $|V| = n$ and E is a set of temporal edges (u, v, t), consider a training pair $(\mathcal{P}, \hat{\mathcal{T}})$, where $\mathcal{P} = \{P_u^t\}_{u \in V, t \in T}$ is a set of node embeddings for each node u at each timestamp t generated by the function $g(G; \theta_1) = \mathcal{P}$, and $\hat{\mathcal{T}} = \{\hat{I}_u\}_{u \in V} = [\hat{s_u}, \hat{e_u}]_{u \in V}$ is the set of observed activity intervals for each node u, we define a function*

$$f(g(G; \theta_1); \theta_2) = \mathcal{T} \tag{1}$$

for which we seek to find the parameters θ_1 and θ_2 that minimize the sum of squared differences between the observed activity intervals and the predicted ones $\mathcal{T} = \{I_u\}_{u \in V} = [s_u, e_u]_{u \in V}$, across all nodes and timestamps:

$$(\theta_1^*, \theta_2^*) = \underset{\theta_1, \theta_2}{\arg\min} \frac{1}{n} \sum_{u \in V} \left((s_u - \hat{s_u})^2 + (e_u - \hat{e_u})^2 \right) \tag{2}$$

subject to the coverage constraint

$$\forall (u, v, t) \in E, \quad t \in I_u \text{ or } t \in I_v.$$

By formulating the MinTCover problem as a learning task, we aim to learn a mapping from temporal graphs to activity timelines that firstly effectively builds temporal node representation and then efficiently covers all interactions with a minimal total active duration. This approach allows us to leverage DL techniques to approximate optimal solutions.

3. Related Works

To address the challenges of the MinTCover problem on temporal graphs, this study explores two complementary areas of research that align with our objectives: heuristic-based methods and DL for CO. These focus areas are motivated by the dual nature of the problem. First, heuristic approaches have been widely studied for CO tasks, offering practical solutions with computational efficiency. However, these methods often struggle with capturing the intricate temporal and structural dynamics of complex, large-scale temporal graphs, leading to suboptimal solutions in terms of accuracy and coverage size. Second, DL has demonstrated significant potential in addressing these limitations in static cases by leveraging its ability to learn from data, model dependencies, and generalize across diverse instances; however, the application of DL approaches to dynamic cases is still little explored, and studies mainly focus on using DL to build suitable temporal representation in various ways. Therefore, our objectives are to (i) evaluate and extend heuristic methods for their strengths in computational efficiency and scalability, and (ii) develop and benchmark

a DL-based approach that can achieve superior precision and adaptability in solving the MinTCover problem.

3.1. Approximate Algorithm and Heuristics

In [6], Rozenstein et al. introduce an iterative method called INNER to solve MinTCover by first addressing the COALESCE subproblem. This involves defining key time points for each vertex, called inner points, around which activity intervals are constructed to cover all interactions. The authors provide a 2-approximate solution to COALESCE in linear time by formulating an ILP model, relaxing its integer constraints, and iteratively refining the inner points until convergence. They do not establish an approximation factor for MinTCover but can compute a feasible solution efficiently.

Dondi et al. [10,18] propose 2PHASES, an $O(T \log n)$ approximation algorithm. They construct a union graph G_u by aggregating timestamps of temporal edges. The first phase applies a 2-approximate Minimum Vertex Cover (MVC) algorithm [19] to select activity intervals. The second phase, based on a SetCover approximation [20], uses randomized rounding to determine a Minimum Non-Consecutive Timeline Cover, later transformed into a MinTCover solution.

Lazzarinetti et al. [11] introduce FASTMINTC+, a heuristic method extending an MVC approach. It consists of an initialization phase that selects high-degree vertices for coverage and an iterative refinement phase where low-loss nodes are removed, and new nodes are selected using the Best from Multiple Selection (BMS) heuristic. The process iterates until a minimal timeline with a reduced sum-span is obtained. Their benchmark comparisons show that FASTMINTC+ is computationally efficient, consistently outperforming INNER in execution time and proving superior for dense graphs, while 2PHASES, despite its theoretical guarantees, struggles with scalability in large instances.

3.2. Deep Learning Approaches for Combinatorial Optimization

Beyond heuristic and approximate methods, DL approaches have been explored for solving NP-hard problems, yielding promising results [21–23]. Most research focuses on CO problems over static graphs [24], where the challenge is capturing the combinatorial structure of the network through spatial features. However, incorporating the temporal dimension significantly increases complexity, making existing DL-based methods ineffective in providing reasonable solutions. This limitation arises because representation learning techniques designed for static graphs struggle to handle temporal dynamics properly [25].

For this reason, in recent years, temporal graph embedding has gained significant attention. Various approaches have been developed, and they can be categorized into several distinct research paradigms.

- Event-Based Temporal Graph Embedding: These methods treat node interactions as timestamped events to capture precise temporal dependencies, like in *DyRep* [26] and *TGAT* [27]. They provide high temporal resolution but are computationally demanding, especially for high-frequency events.
- Snapshot-Based Temporal Learning: Temporal graphs are divided into snapshots, with conventional GNNs applied to each one. Models like *EvolveGCN* [28] and *DCRNN* [29] work well for gradual changes but require significant memory and may miss intermediate dynamics.
- Self-Supervised Learning and Pre-Training: These methods use automatically generated pseudo-labels for training, reducing the need for labeled data. Examples include *TREND* [30] and *T-GCL* [31]. They are adaptable to low-data scenarios but can be sensitive to negative sampling strategies.

- Hybrid Models: Combining components like temporal attention and convolutions, hybrid models capture complex interactions. *TMac* [32] and the *Temporal Graph Collaborative Transformer* [33] are flexible but can suffer from high complexity and training time.
- Inductive Learning and Scalability: Focusing on scalability, models like *GraphSAGE* [34] generate embeddings for new nodes and scale well to large graphs. However, they may lose some global context and require extensive hyperparameter tuning.
- Application-Specific Models: Tailored for tasks like link prediction and anomaly detection, these models, such as *JODIE* [35] and *DySAT* [36], perform well for specific applications but have limited generalizability and require domain-specific adjustments.

The approaches demonstrate the diversity and complexity of temporal graph embedding methods, each addressing specific challenges in dynamic network representation.

4. Deep Learning-Based Minimum Timeline Cover

To solve Problem 2 in a data-driven fashion, we propose integrating DL techniques to derive actionable insights from temporal graph data, aiming to optimize decisions dynamically over time. To compute the function as in Equation (1), we can use a pointer mechanism as described in [37]. The pointer mechanism enables the model to *point* directly to elements within the input sequence, rather than generating new output tokens. This is achieved by learning a probability distribution over the input positions, effectively guiding the network to select certain elements based on their relevance to the task. As a result, PNs can dynamically adapt their outputs based on the input structure, making them highly effective for problems where the order and relationships between elements are essential, such as in graph-based CO tasks. Thus, the key goal is to transform the temporal graph's data into a format that a PN can effectively utilize to make informed decisions. This involves embedding the graph data into a numerical form that captures both the structural and temporal dynamics, and then using these embeddings to drive a PN decision-making process. For background on the DL models adopted, please refer to Appendix A.

Among the possible solutions for temporal graph embeddings, we argue that the best choice for solving MinTCover in a data-driven fashion is represented by models that can scale to instances of different sizes and complexities. Moreover, considering that there exists a trade-off between the amount of processed information and scalability, we believe that, to find an activity timeline, it is better to focus on the temporal distribution of nodes within the entire span, with respect to building a timestamped node representation considering the entire network topology at a given point in time.

According to these assumptions, we designed a DL approach enforced with an iterative procedure to solve the MinTCover problem in a data-driven fashion. The procedure is broken down into the following stages, better highlighted in Figure 2:

1. Node Representation: Firstly, each node is represented by computing the degree of that node in the temporal graph, injecting information on the topology of the network at each timestamp (bottom-left part of Figure 2).
2. Graph Embedding: The process begins with the embedding of the temporal graph, where node features are encoded at each timestamp. This captures the topology of each graph over time, mainly considering local features (bottom-right part of Figure 2).
3. Temporal Aggregation: Nodes' sequences through time are defined and processed to transform the representation of each node in each timestamp considering the entire sequence (top-left part of Figure 2).
4. Pointer Mechanism: A pointer mechanism uses these embeddings across time, producing an output that encapsulates the temporal evolution of node features (top-right part of Figure 2).

5. Iterative Adjustment: The output of the pointer mechanism includes the interval to be assigned to each node; however, they may not be a valid cover. Thus, an iterative algorithm is proposed to adjust the intervals to grant that the output is a coverage (top part of Figure 2).

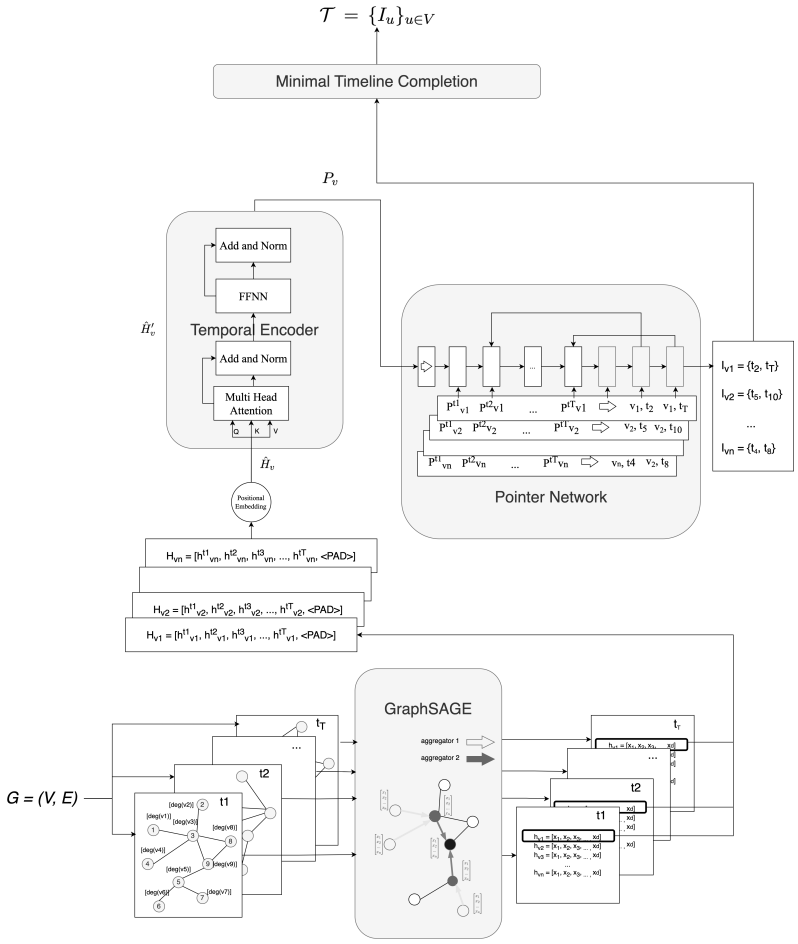

Figure 2. Schema of the DL-based method for the MinTCover problem. The input of the model is represented by a graph $G = (V, E)$ in the bottom-left part of the image. Information then flows following the direction of the arrows. The first stage is node representation. The input graph is split into temporal snapshots (one for each timestamp t_i, where each snapshot always contains all the nodes, but only contains the edges (u, v, t) such that $t = t_i$). For each node v_i in each snapshot graph, its degree is computed as a feature, which in the picture is reported as $[deg(v_i)]$. All these snapshots of the graph G are then passed to the graph embedding block, which uses them to train the parameters of a GraphSAGE model (i.e., to learn how to aggregate information to build a more nuanced representation of each node). This provide a representation for each node in each timestamp $h_{v_i} = [x_1, x_2, \ldots, x_d] \quad \forall v_i \in V, \forall t \in T$. All these representations are then combined in a unique vector for each node $H_{v_i} = [h^{t_1}_{v_1}, h^{t_2}_{v_2}, \ldots, h^{t_n}_{v_n}] \quad \forall v_i \in V$. Note that $< PAD >$ is a symbol used to pad the sequence to its maximum length. Then, these vectors (sequences of embeddings produced for each node and timestamp, for which—from now on—for simplicity, we use v omitting the subscript, even if they are computed for each $v_i \in V$) are used to train the parameters of the positional encoding which produces vectors $\hat{H}_v$ and the Temporal Encoder Transformer, which perform temporal aggregation, first producing $\hat{H}'_v$ and then producing P_v as output. P_v is actually a set of vectors $P_{v_i} = [P^{t_i}_{v_i} \quad \forall t \in T] \quad \forall v_i \in V$. These vectors are then used to train the PN. The input sequence of the PN is a sequence of representations for for each node through time, which represents the evolution of that node through the timestamps. These are used to compute a pair of timestamps $I_{v_i} = [s_{v_i}, e_{v_i}]$ for each of these sequences, which represents the start and end of each timeline. This timeline produced may not be valid; thus, it is then completed with the iterative algorithm.

This integrated approach leverages the strengths of GNNs, Transformers and PNs. By transforming the graph data with GNNs and Transformer into a state representation that reflects both their structural properties and temporal changes, the PN can select the starting and ending points of each node sequence which correspond to the interval. Finally, an iterative procedure can optimize this interval, providing a valid coverage. In the following, we will refer to this approach as DLMINTC+.

4.1. Generation of Temporal Node Features

To build the activity timeline in a data-driven fashion, firstly, we need to represent each node of the temporal graphs as a sequence of embeddings. To derive this sequence $\mathcal{P} = \{P_u^t\}_{u \in V, t \in T}$ from the temporal graph, we use a two-phase process involving GraphSAGE to obtain node embeddings for each timestamp and a Transformer to aggregate these embeddings over time. Suppose we have a temporal graph represented by a sequence of graphs $\{G_t\}_{t=1}^{T}$, where each graph $G_t = (V, E_t)$ has the same set of nodes V but a varying set of edges E_t that changes over time. For each node $v \in V$ and each timestamp t, we compute the local degree of node v at timestamp t as follows:

$$x_v^t = \text{degree}((v, t)) \tag{3}$$

4.2. Application of GraphSAGE

For each timestamp t, we apply a GraphSAGE model to obtain node embeddings as described in Appendix. GraphSAGE aggregates information from neighboring nodes to compute the embedding of node v at timestamp t:

$$h_v^{(t,k)} = \text{ReLU}\left(W_k \cdot \frac{\sum_{u \in N(v) \cup \{v\}} h_u^{(t,k-1)}}{|N(v)| + 1} \right). \tag{4}$$

where $h_v^{(t,0)} = [x_v^0, x_v^1, \ldots, x_v^T]$ is the array of node degrees at each timestamp t, $h_v^{(t,k)}$ is the embedding of node v at layer k and timestamp t, $N(v)$ is the set of neighboring nodes of v, and W_k are the trainable weights of layer k. In this case, we use the ReLU function as the activation function and the mean operator as the AGGREGATE operator, including the node itself in the computation.

The final embedding of node v at timestamp t is given by $h_v^{(t,K)}$, where K is the number of GraphSAGE layers.

Once we have obtained the node embeddings for each timestamp t, to actually use a PN to reason over the sequences of these embeddings over time, we need to build this sequence for each node.

4.3. Temporal Encoding with Transformers

For each node v, we construct a sequence of temporal embeddings:

$$H_v = [h_v^{(1,K)}, h_v^{(2,K)}, \ldots, h_v^{(T,K)}] \tag{5}$$

where $H_v \in \mathbb{R}^{T \times d}$ and d is the dimension of the embedding. This sequence, however, is composed of representations of nodes built considering only static information. No temporal processing has been performed until now. To allow the PN to effectively make informed decisions, this sequence must be reprocessed to effectively embed the temporal evolution of the network in each node representation at each timestamp. Thus, we apply a Transformer with positional encoding to the sequence H_v to obtain the final embedding of node v, as described in Appendix A.3.

Since the embeddings H_v do not inherently capture the temporal structure of the sequence, we first add temporal positional encodings to each timestamp to encode the order information. The positional encoding for a given timestamp t is defined as in Equation (A11). We then add the positional encoding to each input embedding to obtain the temporally encoded sequence:

$$\hat{h}_v^{(t,K)} = h_v^{(t,K)} + PE(t), \quad \forall t \in [1, \ldots, T]. \tag{6}$$

The resulting sequence is denoted as

$$\hat{H}_v = [\hat{h}_v^{(1,K)}, \hat{h}_v^{(2,K)}, \ldots, \hat{h}_v^{(T,K)}] \in \mathbb{R}^{T \times d}. \tag{7}$$

We then apply a Transformer layer to the temporally encoded sequence $\hat{H}_v$ to capture temporal dependencies between different timestamps. Each Transformer layer consists of a Multi-Head Self-Attention mechanism followed by a feedforward neural network (FFNN), for which details are provided in Appendix A.1. The self-attention mechanism computes three matrices— Query, Key, and Value—using learned linear transformations:

$$Q = \hat{H}_v W_Q, \quad K = \hat{H}_v W_K, \quad V = \hat{H}_v W_V, \tag{8}$$

where $W_Q, W_K, W_V \in \mathbb{R}^{d \times d}$ are the learned weight matrices, and $Q, K, V \in \mathbb{R}^{T \times d}$.

These are used to compute the scaled dot-product attention as in Equation (A9). Finally, instead of a single set of Q, K, V matrices, the Transformer utilizes multiple attention heads to capture diverse temporal patterns as in Equation (A10).

The output of the multi-head attention layer is passed through an FFNN applied independently to each timestamp and each sub-layer (self-attention and FFNN) is followed by a residual connection and a layer normalization step. The output of the multi-head attention layer is

$$\hat{H}'_v = \text{LayerNorm}(\hat{H}_v + \text{MultiHead}(Q, K, V)), \tag{9}$$

and the final output of the Transformer layer is

$$P_v = \text{LayerNorm}(\hat{H}'_v + FFNN(\hat{H}'_v)). \tag{10}$$

The resulting sequence $P_v = [P_v^{(1)}, P_v^{(2)}, \ldots, P_v^{(T)}]$ captures the temporal relationships between the timestamps while preserving the dimensionality of the input embeddings. Thus, each element $P_v^{(t)} \in \mathbb{R}^d$ is now conditioned on the entire sequence, enabling temporal dependencies to be modeled explicitly.

4.4. Application of the Pointer Network

Once we have constructed the sequence of node embeddings for each node v, represented as $P_v = [P_v^{(1)}, P_v^{(2)}, \ldots, P_v^{(T)}]$, the next step is to apply the PN to determine the optimal activity intervals for each node based on these embeddings.

The primary goal of the PN in the context of the MinTCover problem is to select, for each node in the temporal graph, the most effective activity interval that covers all relevant interactions within the shortest possible duration. The Pointer Network achieves this by processing sequences of embeddings that encapsulate both the structural and temporal dynamics of the graph, thus identifying the optimal start and end points of activity for each node.

The PN processes the sequence of embeddings P_v by applying a softmax pointer layer that computes a probability distribution over each timestamp in the sequence P_v. This

distribution is indicative of the likelihood that a particular timestamp is the starting or ending point of the optimal interval for node v.

Suppose the output from the Transformer for node v is a sequence of vectors $P_v = [P_v^{(1)}, P_v^{(2)}, \ldots, P_v^{(T)}]$. The PN then applies an attention function a to compute the probabilities:

$$a(P(t)_v) = \text{softmax}(W_a P(t)_v) \tag{11}$$

where W_a are the learnable parameters of the network, and $P(t)_v$ is the Transformer output at time t.

The final output of the PN is a set of intervals $\{s_v, e_v\}$ for each node v, where s_v and e_v are selected based on the highest probabilities from the softmax outputs. These intervals have to be chosen to minimize the overall activity span while ensuring all necessary interactions are covered.

4.4.1. Custom Loss Function

In order to properly train the network to also include the logic and the constraints of the MinTCover problem, we decided to use a custom loss to balance covering, minimizing sum-span and granting the integrity of the solution. The loss function, as described in Appendix A.1, quantifies the difference between the model's predictions and the actual target values, serving as a guide for the model to improve during training. By minimizing the loss, the model can learn the patterns in the data more accurately.

The proposed loss function aims to optimize the temporal coverage problem by incorporating three main components: (1) coverage loss, (2) span alignment loss, and (3) a penalty term. Let $G_i = (V_i, E_i)$ denote the i-th graph in a batch of graphs, where V_i is the set of nodes and E_i is the set of temporal edges. Here, a batch refers to a subset of data samples processed simultaneously during training in gradient descent algorithms to efficiently estimate gradients and update model parameters. Each edge $(u, v, t) \in E_i$ represents an interaction between nodes u and v at timestamp t.

Coverage Loss

The *coverage loss* aims to ensure that each temporal edge is adequately covered by the predicted activity intervals of its associated nodes. For an edge (u, v, t), let s_u and e_u denote the start and end times of the predicted interval for node u, respectively, and similarly s_v, e_v for node v. The goal is to penalize the model when neither u nor v is active at timestamp t.

Mathematically, we define the following indicator functions:

$$\delta_u(t) = \mathbf{1}_{\{s_u \le t \le e_u\}}, \quad \delta_v(t) = \mathbf{1}_{\{s_v \le t \le e_v\}} \tag{12}$$

which indicate whether timestamp t is covered by the interval of nodes u and v, respectively.

The total coverage loss for the graph G_i is given by summing the coverage loss for each edge over all edges:

$$\text{coverage\_loss}_{G_i} = \sum_{(u,v,t) \in E_i} \mathbf{1}_{\{\delta_u(t)=0 \,\&\, \delta_v(t)=0\}} \tag{13}$$

This loss attempts to ensure that most edges are covered by at least one of the associated nodes.

The direct use of indicator functions makes the coverage loss non-differentiable at the boundary points where $s_u = t$ or $e_u = t$. However, in the current implementation, the indicator function is implicitly relaxed using a ReLU. This smooth approximation enables the computation of gradients for the coverage loss component with respect to the

start and end times. Thus, the resulting coverage loss is differentiable almost everywhere, except at points where the smooth approximations switch values (e.g., at $s_u = t$).

Span Loss

The *span loss* aims to align the predicted start and end times with the observed activity intervals for each node. Let $\hat{s_u}$ and $\hat{e_u}$ denote the observed start and end times of node u. The goal is to minimize the difference between the predicted and observed intervals.

The span loss is defined as the mean squared error (MSE) between the predicted and observed intervals for each node ($|V| = n$):

$$\text{span\_loss}_{G_i} = \frac{1}{n} \sum_{u \in V_i} \left((s_u - \hat{s_u})^2 + (e_u - \hat{e_u})^2 \right) \tag{14}$$

This loss ensures that the model learns to predict intervals that closely match the observed ones, reducing the discrepancy between the predicted activity spans and the actual activity periods.

Each term in the summation of the span loss is a quadratic function of the start and end times s_u and e_u. Quadratic functions are differentiable everywhere with continuous first and second derivatives. Therefore, the span loss is fully differentiable and provides smooth gradients for optimizing the alignment between the predicted and true intervals.

Penalty Term

The *penalty term* is introduced to enforce the constraint that the start time s_u should always be less than or equal to the end time e_u for each node u. This ensures that the predicted intervals are valid.

The penalty term is defined as

$$\text{penalty}_{G_i} = \sum_{u \in V_i} \text{ReLU}(s_u - e_u)^2 \tag{15}$$

The squared penalty grows as s_u approaches or exceeds e_u, thereby strongly discouraging invalid intervals.

The ReLU function itself is piecewise linear and non-differentiable at $x = 0$. However, the squared ReLU term $\text{ReLU}(s_u - e_u)^2$ is differentiable everywhere except at the point $s_u = e_u$, where the gradient changes abruptly. Despite this, the penalty term contributes useful gradients in almost all regions of the parameter space and ensures that the optimization process is well behaved except at a few boundary points.

Total Loss for a Single Graph

The overall loss for a single graph G_i can be computed as a weighted combination of the coverage loss, span loss, and the penalty term. To stabilize the gradients and avoid issues with small loss values, we apply a logarithmic transformation with a small constant ϵ to each element of the loss. The logarithmic transformation is differentiable for all positive values of $loss_{G_i}$ and provides smooth gradients. The inclusion of ϵ ensures that the argument inside the logarithm is strictly positive, thereby making the entire transformation differentiable everywhere.

The final loss is given by

$$\text{Loss}_{G_i} = \alpha \cdot \log(\text{coverage\_loss}_{G_i}) + \beta \cdot \log(\text{span\_loss}_{G_i}) + \gamma \cdot \log(\text{penalty}_{G_i}) \tag{16}$$

The final loss function is a weighted sum of the adjusted logarithm of the coverage loss, span loss, and penalty term, all of which are differentiable. Since the weighted sum

of differentiable functions is itself differentiable, the entire loss function is differentiable, enabling the use of gradient-based optimization techniques.

Hyperparameter Sensitivity

The choice of weights α, β, γ is critical for balancing the contributions of each component. For instance, setting a higher value of α would prioritize edge coverage, potentially leading to larger intervals, while increasing β would focus more on accurately aligning intervals with the observed ones. Similarly, γ controls the strictness of the $s_u \leq e_u$ constraint, ensuring valid interval predictions.

4.5. Iterative Algorithm

Even though properly trained, the activity intervals $\mathcal{T} = \{I_u\}_{u \in V}$ produced by the DL approach may not be neither minimal nor valid activity intervals for MinTCover. To grant that the output is a valid set of activity intervals, we suggest adjusting these intervals minimally to ensure that all temporal edges are covered. We thus introduce an algorithm designed to achieve this by minimally extending node intervals where necessary.

4.5.1. Algorithm Description

The algorithm takes the initial intervals from the DL model and iteratively adjusts them to cover all temporal edges. For each uncovered edge, it decides which node's interval to adjust based on the minimal increase in the total sum-span $S(\mathcal{T})$.

Given a temporal graph $G = (V, E)$, an activity interval $I_u = [s_u, e_u]$, the total sum-span of all activity intervals $S(\mathcal{T}) = \sum_{u \in V}(e_u - s_u)$ and a set of uncovered edges $U = \{(u, v, t) \in E \mid t \notin I_u \wedge t \notin I_v\}$, the algorithm performs the following steps:

1. Initialization: Copy the initial intervals $\mathcal{T}$ and initialize the uncovered edges in set U.
2. Edge Checking: For each edge $(u, v, t) \in E$, check if $t \in I_u$ or $t \in I_v$. If not, add the edge to U.
3. Interval Adjustment: For each edge $(u, v, t) \in U$, calculate the minimal increase required to adjust I_u and I_v to include t.
4. Update Intervals: Adjust the interval of the node with the minimal increase to include t.
5. Iteration: Repeat steps 3 and 4 until all edges in U are covered.
6. Output: Return the adjusted intervals $\mathcal{T}$ that now cover all edges in E.

The algorithm is detailed in Algorithm 1.

4.5.2. Complexity Analysis

The efficiency of the algorithm is crucial, particularly when dealing with large-scale temporal networks. To analyze the algorithm's complexity, let us define the following parameters: $n = |V|$, $m = |E|$, and $t = |T|$, where n is the number of nodes, m is the number of edges, and t is the number of timestamps in the network. We can break down the time complexity as follows.

The algorithm begins by copying intervals and initializing a set U, which involves operations on each node in the graph. Consequently, this step has a time complexity of $O(n)$. Then, an edge-checking loop is performed. This loop iterates over all edges in the graph to perform a constant-time check on each. Since the loop covers every edge exactly once, its time complexity is $O(m)$. An interval adjustment loop follows (Lines 7–26). In this step, the algorithm iterates over the edges stored in the set U. In the worst case, U could contain all the edges in the graph, i.e., $|U| = m$. Each iteration involves a series of constant-time operations, making the overall time complexity of this loop $O(m)$.

The total time complexity of the algorithm is dominated by the edge-checking and interval adjustment loops, both of which scale linearly with the number of edges. Thus, the final time complexity is $O(m+n)$.

Algorithm 1: Minimal timeline completion.

Input: V, E, $\mathcal{T} = \{I_u = [s_u, e_u]\}_{u \in V}$
Output: Adjusted intervals $\mathcal{T}$ covering all edges in E
// Initialize Variables
$intervals \leftarrow \mathcal{T}$
$U \leftarrow \emptyset$
for $(u, v, t) \in E$ **do**
  if $t \notin I_u$ ***and*** $t \notin I_v$ **then**
    $U \leftarrow U \cup \{(u, v, t)\}$
for $(u, v, t) \in U$ **do**
  // Compute Increase$_u$
  if $t < s_u$ **then**
    Increase$_u \leftarrow s_u - t$
  else
    if $t > e_u$ **then**
      Increase$_u \leftarrow t - e_u$
  else
    Increase$_u \leftarrow 0$
  // Compute Increase$_v$
  if $t < s_v$ **then**
    Increase$_v \leftarrow s_v - t$
  else
    if $t > e_v$ **then**
      Increase$_v \leftarrow t - e_v$
  else
    Increase$_v \leftarrow 0$
  // Update $I_u \leftarrow [s_u, e_u]$ in *intervals*
  if $Increase_u \leq Increase_v$ **then**
    if $t < s_u$ **then**
      $s_u \leftarrow t$
    else
      if $t > e_u$ **then**
        $e_u \leftarrow t$
  //Update $I_v \leftarrow [s_v, e_v]$ in *intervals*
  else
    if $t < s_v$ **then**
      $s_v \leftarrow t$
    else
      if $t > e_v$ **then**
        $e_v \leftarrow t$
return *intervals*

4.5.3. Minimal Timeline Completion Validity

According to Theorem 1, we can show that the output computed by Algorithm 1 is a valid solution (i.e., covers all temporal edges) and compute the solution with a locally

minimal increase (i.e., reducing any intervals among those increased results in a solution which is not valid).

Theorem 1. *Given any initial set of activity intervals $\mathcal{T} = I_u = [s_u, e_u]u \in V$ and a temporal graph $G = (V, E)$, the Minimal Timeline Completion algorithm produces adjusted intervals $\mathcal{T}' = I'_u = [s'_u, e'_u]u \in V$ that meet the following criteria:*

1. *They cover all temporal edges in E, i.e., for every $(u, v, t) \in E$, $t \in I'_u$ or $t \in I'_v$.*
2. *They are locally minimal with respect to sum-span increase, that is, no final interval can be reduced without leaving at least one temporal edge uncovered. Moreover, each incremental expansion chosen by the algorithm was the minimal possible increase at the time it was made.*

The proof of Theorem 1 can be carried out by showing first of all that the algorithm produces intervals that cover all edges, and then showing that the increased sum-span produced is minimal, i.e., any further reduction in the increased intervals would result in uncovered edges.

Proof. Part 1: The algorithm produces intervals that cover all edges. The algorithm starts from an initial set of intervals $\mathcal{T}$ and identifies all uncovered edges:

$$U = \{(u, v, t) \in E \mid t \notin I_u \wedge t \notin I_v\}.$$

For each edge $(u, v, t) \in U$, the algorithm determines the minimal necessary increase to adjust either I_u or I_v so that t falls within it. One of these intervals is then extended just enough to cover t. As a result, this edge is no longer uncovered.

Since the algorithm processes each uncovered edge in U in this manner, every edge initially in U will be covered by at least one updated interval after these adjustments. Thus, when no uncovered edges remain, every $(u, v, t) \in E$ is covered by I'_u or I'_v. Hence, the final set of intervals $\mathcal{T}'$ covers all temporal edges.

Part 2: Local minimality of the total sum-span increase. Consider an uncovered edge $(u, v, t) \in U$. To cover it, the algorithm calculates the increases required to include t in I_u or I_v:

$$\delta_u = \begin{cases} s_u - t, & \text{if } t < s_u \\ t - e_u, & \text{if } t > e_u \\ 0, & \text{if } t \in [s_u, e_u] \end{cases} \quad \text{and} \quad \delta_v = \begin{cases} s_v - t, & \text{if } t < s_v \\ t - e_v, & \text{if } t > e_v \\ 0, & \text{if } t \in [s_v, e_v]. \end{cases}$$

The algorithm picks the endpoint with the minimal increase $\delta = \min(\delta_u, \delta_v)$ and adjusts its interval accordingly. This choice ensures that for each uncovered edge, the increase in sum-span is as small as possible at the moment of adjustment.

Now consider the final solution $\mathcal{T}'$ after all uncovered edges are addressed. Suppose we attempt to reduce the sum-span of this solution by shrinking one of the intervals $I'_u = [s'_u, e'_u]$. Any attempt to decrease e'_u or increase s'_u may exclude a timestamp that was previously uncovered and is now covered only by I'_u. Because the algorithm always adjusted intervals minimally, there is no "slack" that can be removed without losing coverage of at least one temporal edge.

This implies the solution is locally minimal: there is no unilateral reduction in the updated intervals that can decrease the total sum-span without uncovering an edge. Furthermore, at each individual step of covering an edge, the chosen increment was minimal. Hence, for each uncovered edge at the time it was covered, no smaller incremental adjustment could have achieved coverage.

Part 3: No further reductions are possible without losing coverage. Since each interval extension was chosen to be the minimal required to cover a newly uncovered edge, removing or shrinking any part of these extensions would exclude at least one such edge. Thus, any attempt to reduce the total sum-span after the algorithm completes would necessarily leave some edge uncovered. Therefore, the intervals in $\mathcal{T}'$ cannot be reduced without losing coverage, confirming the local minimality of the solution. □

5. Experimental Evaluation

In this section, we provide the results of our experiments on the DLMINTC+ methodology proposed. Given the benchmark results provided in [11], which show that on average FASTMINTC+ is the best heuristic among those proposed in the literature, we evaluate our model against FASTMINTC+ on the same datasets, i.e., *Dataset1* composed of synthetic sparse instances of different sizes, *Dataset2* composed of synthetic dense instances of different sizes, and *Dataset3* composed of real-world instances from the DIMACS dataset [38].

We recall that for a more granular analysis, instances of the synthetic datasets are categorized as small (up to 50 vertices and 20 timestamps), medium (up to 500 vertices and 500 timestamps), and hard (up to 10,000 vertices and 5000 timestamps). The final goal is understanding the potential of DL-based approaches for temporal CO problems with respect to heuristic approaches on both sparse and dense instances.

5.1. Experimental Settings

Experiments are carried out on a MacBook Pro (2017) with MacOS, using a 16 GB RAM and 4 cores of an i7-3, 1 GHz CPU.

For the FASTMINTC+ algorithm, we set the parameter $k = 50$ for the BMS heuristic (k represents the number of iterations performed by the BMS heuristic), and we execute the exchange step for 2000 iterations and the overall algorithm 5 times, each with a different shuffling of the edges as suggested by the authors.

As far as DLMINTC+'s hyperparameters are concerned, the network has been trained for 50 epochs, with a batch size $B = 128$. To optimize the custom loss function throughout the entire network, we used Adam [39] with an initial learning rate $\alpha = 10^{-3}$ (please refer to Appendix A.1 for further details on gradient descent and Adam as optimization strategies for loss computation). To optimize the performance of the model, we performed hyperparameter tuning using a grid search strategy on a validation set composed of 1000 random graphs. Table 1 shows the parameters tuned and selected from the tested values. Training took approximately 9 h to complete the 50 epochs, with an average consumption of 12.37 GB of memory during training. To minimize memory overhead in the GraphSAGE and Transformer modules, we use sparse matrix representations, which inherently benefit from sparse connectivity in large temporal graphs.

Table 1. Final hyperparameter values used in the experiments along with the ranges tested during tuning. The selected values were optimized based on validation loss and stability. Note that for loss weights, the tested values are all possible combinations of the three values listed.

Hyperparameter	Selected Value	Tested Values
Learning Rate (α)	10^{-3}	$[10^{-4}, 10^{-3}, 10^{-2}]$
Batch Size (B)	128	$[32, 64, 128, 256]$
GraphSAGE Layers (K)	3	$[2, 3, 4]$
Embedding Dimension (d)	32	$[16, 32, 64]$
Loss Weights (α, β, γ)	$0.2, 0.5, 0.3$	$\{0.2, 0.3, 0.5\}$

To train the neural model, we crafted a synthetic dataset composed of randomly generated graphs of different sizes, densities and spans. Consider that while synthetic

datasets offer flexibility and control, they also may not capture all structural and temporal properties of real-world networks or they may omit complexities, such as noise or irregular patterns, that are present in real-world graphs, thus introducing bias. To mitigate these possible drawbacks, we generated synthetic graphs of varying sizes and densities to mimic real-world scenarios. Moreover, we tested the results both on synthetic and real-world graphs, to be sure that the results obtained on synthetic data are comparable to those obtained on real-world graphs. The final training dataset is composed of 5000 graphs for training and 1000 graphs for validation with a random number of nodes between 10 and 10,000, a random number of edges between 0.1 times the number of nodes and 10 times the number of nodes, and a random number of timestamps between 1 and 30,000. This represents the maximum value allowed for the sequence length. This means that by itself, the network is not able to process temporal graphs with more than 30,000 timestamps. However, since in many cases the number of timestamps is larger and since the final output produced is not granted to be a valid coverage, in the case of temporal graphs with more than 30,000 timestamps, we split the temporal graph into sub-temporal graphs with a maximum span of 30,000, process each sub-graph independently, and merge the produced intervals, which are then passed to the final iterative algorithms. This graph partitioning technique not only allows us to extend the approach to graphs of any size, but it also reduces the memory requirements of the DL-based approach. Since computing exact solutions for such graphs is infeasible and no suitable dataset with graphs and corresponding ground truth exists in the literature, we constructed the observed values for training using state-of-the-art heuristics. Specifically, we employed both FastMinTC+ and Inner, the best-performing heuristics across different network configurations (FastMinTC+ generally excels on higher-density instances, while Inner is often superior for lower-density instances). For each graph, we generated solutions using both heuristics, selecting the one yielding the best results in terms of sum-span. Although it is not possible to train the network on optimal solutions, this approach enables the model to learn to approximate the best solutions regardless of the network's characteristics, thus taking the best of both the state-of-the-art heuristics. Provided that the training is executed correctly, the resulting model should, barring errors, produce coverage results as close as possible to those achieved by the best-performing state-of-the-art heuristics.

To ensure that, by the end of the training phase, the network had effectively learned to construct meaningful coverage, we tested its ability to replicate the coverage generated by the heuristics. We conducted this evaluation with the understanding that the output produced by the network might neither be minimal nor entirely valid, aiming instead to refine the result further using the proposed iterative algorithm. Thus, we measured the performance using the classical metrics for classification problems: *accuracy* which measures how often the model correctly predicts the outcome, *precision* which measures how often the model correctly predicts the positive class, and *recall* which measures how often the model correctly identifies positive instances from all the actual positive samples in the dataset. In this case, however, since the output of the network is an interval for each node, we convert the interval to a sequence of 0 and 1 for each timestamp for each node, which indicates weather the node is active in that timestamp or not, and we measure the problem as a binary classification problem. For this evaluation, we used a synthetically generated test dataset of 500 graphs similar to the training and validation dataset. Counting each node for each timestamp in this dataset results in a test case of more than 500×10^6 nodes. The results are reported in Table 2. Considering the size of the support, the training results, with an overall accuracy of 72.13%, while not extremely high, represent strong performances given the scale of the test cases. These results also indicate both the

stability and sufficient generalization of the model, highlighting its robustness across a substantial dataset.

Table 2. Neural model evaluation metrics (accuracy, precision and recall) computed over the synthetically generated dataset composed of a total of 500 graphs. Support represents the total number of samples over which these metrics have been computed, i.e., the total number of nodes

Accuracy	Precision	Recall	Support
72.13 %	75.01%	69.22%	$>500 \times 10^6$

5.2. Experimental Results

The comprehensive results for *Dataset1*, consisting of sparse graphs, and for *Dataset2*, consisting of dense graphs, in terms of sum-span and execution time are reported in Figure 3. DLMINTC+ outperforms FASTMINTC+ in terms of sum-span on both sparse and dense instances, despite it taking a longer execution time. Indeed, on *Dataset1*, it achieves an average reduction of 9.3% in coverage size compared to FASTMINTC+, demonstrating the model's better ability to learn temporal patterns that reduce the overall interval count, despite the 29% increase in execution time compared to FASTMINTC+, due to the added computational complexity required by the DL model. Similarly, on *Dataset2*, DLMINTC+ continues to outperform FASTMINTC+ in terms of coverage size, with an average reduction of 7.5%. This highlights the model's effectiveness in capturing complex structures and interactions also in dense temporal graphs. However, the execution time shows a more noticeable increase, averaging 38% due to the greater complexity of the graphs, which requires more iterations and calculations to produce the activity intervals.

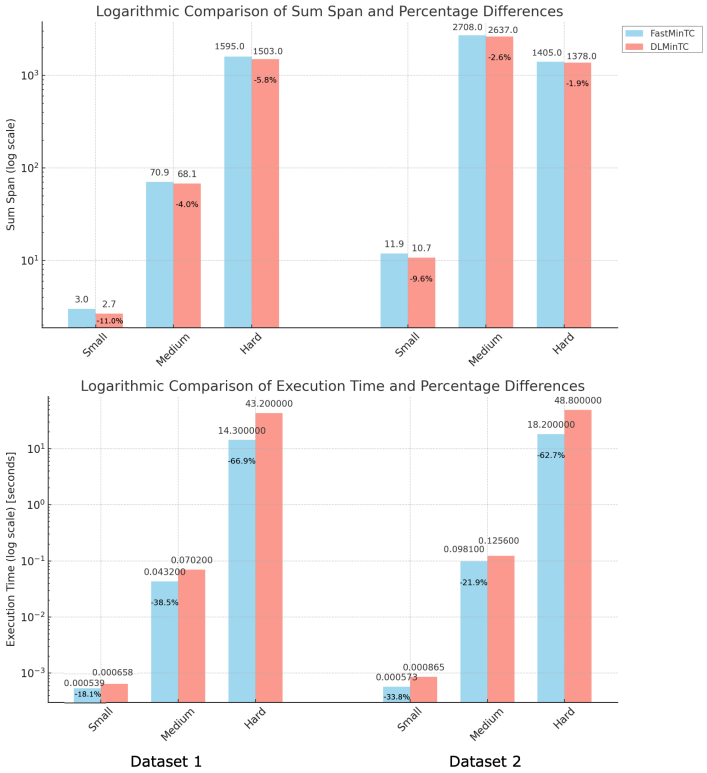

Figure 3. Comparison of sum-span (**top**) and execution time (**bottom**) by dataset and size. In the top figure, we can see the improvement in terms of percentage reduction of sum-span introduced by DLMINTC+ over FASTMINTC+; in the bottom figure, we can see the percentage increment in execution time of DLMINTC+ expressed as percentage reduction in FASTMINTC+.

As we can see in Table 3, the increase in execution time comes with a corresponding increase in memory consumption on both *Dataset1* and *Dataset2*. This is due to the complexity of the trained neural model, which, even if it does not require memory to compute backpropagation at the inference time, needs memory to load the entire graph.

Table 3. Experimentalresults on Dataset1 and Dataset2 in terms of RAM consumption. The results show the large amount of memory required by the DL-based solution to produce results, against the limited amount of memory required by the baseline heuristic.

Instances	Algorithm	Avg RAM Usage on Dataset1	Avg RAM Usage on Dataset1
Small Instances	FastMinTC+	0.2 MB	0.6 MB
	DLMinTC+	0.4 GB	1.8 GB
Medium Instances	FastMinTC+	10.3 MB	25.8 MB
	DLMinTC+	1.2 GB	3.1 GB
Hard Instances	FastMinTC+	24.2 MB	44.7 MB
	DLMinTC+	3.1 GB	7.2 GB

Experiments conducted on publicly available graphs corroborate the findings observed in the synthetic test instances too. Detailed results for these real-world instances are presented in Tables 4–6. For each graph considered, Table 4 provides the results in terms of sum-span, Table 5 provides the results in terms of execution time, and Table 6 provides the results in terms of memory consumption.

In this scenario, DLMINTC+ shows variable behavior depending on the graph structure. Figure 4 shows the details of the comparison with respect to sum-span percentage reduction and execution time percentage increment. The model achieves an average reduction in sum-span of 13.4% compared to FASTMINTC+, with some cases reaching a reduction around 50% and others with reductions limited to around 2–3%. This clearly depends on the size and complexity of the network analyzed and on the behavior of FASTMINTC+ too. Indeed, if we better analyze Figure 4, we can see that the highest improvement is obtained in cases of graphs with smaller densities, which in general correspond to cases of graphs with a larger number of timestamps. This means that the proposed approach is better at scaling with larger instances with complex temporal structures. The variability in sum-span reduction is also reflected in the execution time, with an increase of around 50% on average. This is because the baseline heuristic processes simpler graphs more efficiently, whereas the DL-based approach incurs additional computational overhead even for less complex structures.

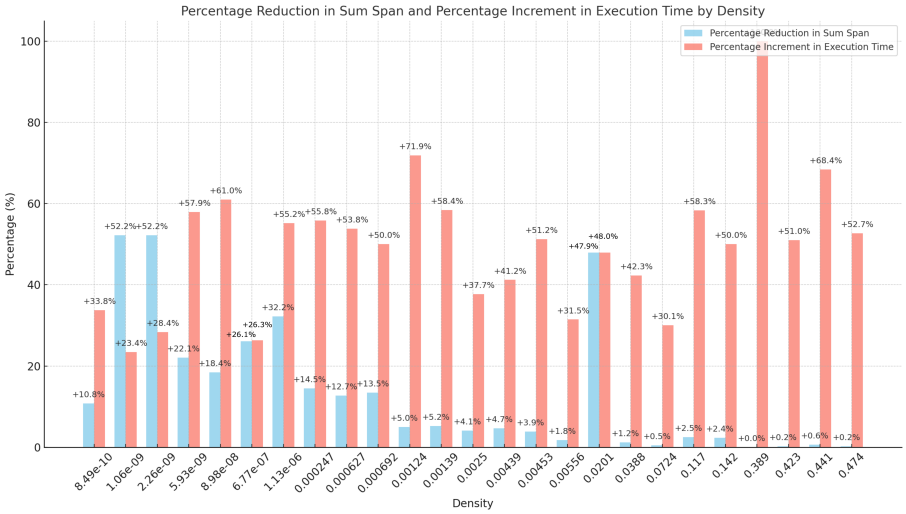

Figure 4. This graph illustrates the percentage reduction in sum-span and percentage increment in the execution time of DLMINTC+ compared to FASTMINTC+ for a variety of graph densities on the real-world DIMACS instances. Overall, the proposed approach produces better results in cases of less dense graphs, which in general correspond to cases with a larger number of timestamps. This proves the ability of the proposed approach to scale to larger instances with respect to the baseline heuristic.

Table 4. Experimentalresults on real-world publicly available graphs from Dataset3. For each considered graph, we report information on the size of the network and the results in terms of the sum-span $S(\mathcal{T})$ of DLMINTC+ and FASTMINTC+.

Graph	|E|	|V|	|T|	D	DLMINTC+ $S(\mathcal{T})$	FASTMINTC+ $S(\mathcal{T})$
aves-sparrow-social	516	52	1	3.89×10^{-1}	1.00×10^{1}	1.00×10^{1}
aves-wildbird-network	11,900	202	5	1.17×10^{-1}	4.66×10^{2}	4.78×10^{2}
copresence-InVS13	394,247	95	20,128	4.39×10^{-3}	1.42×10^{6}	1.49×10^{6}
copresence-InVS15	1,283,194	219	21,535	2.50×10^{-3}	3.49×10^{6}	3.64×10^{6}
copresence-LH10	150,126	73	12,604	4.53×10^{-3}	3.72×10^{5}	3.87×10^{5}
copresence-LyonSchool	6,594,492	242	3123	7.24×10^{-2}	6.61×10^{5}	6.64×10^{5}
copresence-SFHH	1,417,485	403	3148	5.56×10^{-3}	8.76×10^{5}	8.92×10^{5}
copresence-Thiers13	18,613,039	328	8937	3.88×10^{-2}	2.42×10^{6}	2.45×10^{6}
email-dnc	39,264	1892	19,382	1.13×10^{-6}	3.96×10^{5}	5.84×10^{5}
fb-wosn-friends	1,269,502	63,731	736,674	8.49×10^{-10}	8.77×10^{8}	9.83×10^{8}
ia-contacts-dublin	415,912	10,972	76943	8.98×10^{-8}	7.83×10^{5}	9.60×10^{5}
ia-contacts-hypertext2009	20,818	113	5245	6.27×10^{-4}	3.43×10^{5}	3.93×10^{5}
ia-digg-reply	87,627	30,398	83,942	2.26×10^{-9}	8.41×10^{7}	1.76×10^{8}
ia-hospital-ward-proximity	32,424	75	9452	1.24×10^{-3}	3.02×10^{5}	3.18×10^{5}
ia-primary-school-proximity	125,773	242	3099	1.39×10^{-3}	5.97×10^{5}	6.30×10^{5}
ia-prosper-loans	3,394,979	89,269	1258	6.77×10^{-7}	7.39×10^{5}	1.00×10^{6}
ia-retweet-pol	61,157	18,470	60,500	5.93×10^{-9}	9.43×10^{6}	1.21×10^{7}
insecta-ant-colony1	111,578	113	40	4.41×10^{-1}	3.23×10^{3}	3.25×10^{3}
insecta-ant-colony3	241,280	160	40	4.74×10^{-1}	4.76×10^{3}	4.77×10^{3}
insecta-ant-colony5	194,317	152	40	4.23×10^{-1}	4.13×10^{3}	4.14×10^{3}
mammalia-raccoon-proximity	1997	24	51	1.42×10^{-1}	7.43×10^{2}	7.61×10^{2}
rec-amz-Baby	915,446	596,316	4868	1.06×10^{-9}	1.39×10^{6}	2.91×10^{6}
reptilia-tortoise-network-bsv	554	136	3	2.01×10^{-2}	5.00×10^{1}	9.60×10^{1}
reptilia-tortoise-network-fi	1713	787	8	6.92×10^{-4}	2.70×10^{2}	3.12×10^{2}
SFHH-conf-sensor	70,261	403	3508	2.47×10^{-4}	7.07×10^{5}	8.27×10^{5}

Table 5. Experimental results on real-world publicly available graphs from Dataset3. For each considered graph, we report information on the size of the network and the results in terms of execution time of DLMINTC+ and FASTMINTC+.

Graph	|E|	|V|	|T|	D	DLMINTC+ Elapsed	FASTMINTC+ Elapsed
aves-sparrow-social	516	52	1	3.89×10^{-1}	0.03	0.00
aves-wildbird-network	11,900	202	5	1.17×10^{-1}	0.12	0.05
copresence-InVS13	394,247	95	20,128	4.39×10^{-3}	1.82	1.07
copresence-InVS15	1,283,194	219	21,535	2.50×10^{-3}	5.57	3.47
copresence-LH10	150,126	73	12,604	4.53×10^{-3}	0.80	0.39
copresence-LyonSchool	6,594,492	242	3123	7.24×10^{-2}	22.77	15.92
copresence-SFHH	1,417,485	403	3148	5.56×10^{-3}	4.64	3.18
copresence-Thiers13	18,613,039	328	8937	3.88×10^{-2}	75.71	43.69
email-dnc	39,264	1892	19,382	1.13×10^{-6}	0.29	0.13
fb-wosn-friends	1,269,502	63,731	736,674	8.49×10^{-10}	10.86	7.19
ia-contacts-dublin	415,912	10,972	76,943	8.98×10^{-8}	2.87	1.12
ia-contacts-hypertext2009	20,818	113	5245	6.27×10^{-4}	0.13	0.06
ia-digg-reply	87,627	30,398	83,942	2.26×10^{-9}	2.08	1.49
ia-hospital-ward-proximity	32,424	75	9452	1.24×10^{-3}	0.32	0.09
ia-primary-school-proximity	125,773	242	3099	1.39×10^{-3}	0.77	0.32
ia-prosper-loans	3,394,979	89,269	1258	6.77×10^{-7}	32.18	23.71
ia-retweet-pol	61,157	18,470	60,500	5.93×10^{-9}	1.83	0.77
insecta-ant-colony1	111,578	113	40	4.41×10^{-1}	0.98	0.31
insecta-ant-colony3	241,280	160	40	4.74×10^{-1}	1.29	0.61
insecta-ant-colony5	194,317	152	40	4.23×10^{-1}	1.02	0.50
mammalia-raccoon-proximity	1997	24	51	1.42×10^{-1}	0.02	0.01
rec-amz-Baby	915,446	596,316	4868	1.06×10^{-9}	16.36	12.53
reptilia-tortoise-network-bsv	554	136	3	2.01×10^{-2}	1.23	0.64
reptilia-tortoise-network-fi	1713	787	8	6.92×10^{-4}	0.02	0.01
SFHH-conf-sensor	70,261	403	3508	2.47×10^{-4}	0.43	0.19

Thus, overall, DLMINTC+ demonstrates superior performance in terms of sum-span compared to FASTMINTC+ on all graphs. Specifically, in the analysis of 25 real-world instances, DLMINTC+ achieves a better sum-span in all cases. Notably, DLMINTC+ not only outperforms FASTMINTC+ on all instances but, comparing results presented in [11], also the INNER and 2PHASES algorithms on all instances. Note that the improvements achieved can be translated into tangible benefits in various real-life scenarios. As an example, consider the case of telecommunication networks. In temporal graphs representing communication events, a reduction in the sum-span by 10% means a significant reduction in the time and resources needed to monitor and maintain network activity logs. For large networks with millions of interactions, this can reduce storage requirements by gigabytes, ensuring faster query responses and lower costs. In the case of transportation scheduling, instead, a 10% improvement in the coverage size of dynamic transportation networks (e.g., bus or train schedules), translates to fewer overlapping intervals when monitoring vehicle activity. This could result in optimized resource allocation (e.g., fewer active vehicles needed during certain time intervals) and improved passenger wait times.

The results also confirm the performance in terms of execution time and of memory consumption of the FASTMINTC+, which always achieves a lower execution time and a lower memory consumption compared to DLMINTC+, even though the order of magnitude of the execution time is comparable and memory consumption is still limited, making it a suitable solution in real-world scenarios.

Table 6. Experimental results on real-world publicly available graphs from Dataset3. For each considered graph, we report information on the memory consumption of DLMinTC+ and FastMinTC+. Note that for DLMinTC+, memory is expressed in GB, while it is expressed in MB forFastMinTC+.

Graph	|E|	|V|	|T|	D	DLMinTC+ RAM (GB)	FastMinTC+ RAM (MB)
aves-sparrow-social	516	52	1	3.90×10^{-1}	0.3	0.1
aves-wildbird-network	11,900	202	5	1.17×10^{-1}	0.7	0.4
copresence-InVS13	394,247	95	20,128	4.39×10^{-3}	7.8	24.8
copresence-InVS15	1,283,194	219	21,535	2.50×10^{-3}	8.2	26.6
copresence-LH10	150,126	73	12,604	4.53×10^{-3}	5.5	20.5
copresence-LyonSchool	6,594,492	242	3123	7.24×10^{-2}	4.6	16.4
copresence-SFHH	1,417,485	403	3148	5.56×10^{-3}	3.8	16.7
copresence-Thiers13	18,613,039	328	8937	3.88×10^{-2}	8.4	19.3
email-dnc	39,264	1892	19,382	1.13×10^{-6}	6.8	18.7
fb-wosn-friends	1,269,502	63,731	736,674	8.49×10^{-10}	12.7	45.2
ia-contacts-dublin	415,912	10,972	76,943	8.98×10^{-8}	11.4	41.6
ia-contacts-hypertext2009	20,818	113	5245	6.27×10^{-4}	5.5	15.8
ia-digg-reply	87,627	30,398	83,942	2.26×10^{-9}	10.8	38.0
ia-hospital-ward-proximity	32,424	75	9452	1.24×10^{-3}	5.3	13.4
ia-primary-school-proximity	125,773	242	3099	1.39×10^{-3}	4.5	13.4
ia-prosper-loans	3,394,979	89,269	1258	6.77×10^{-7}	2.8	12.5
ia-retweet-pol	61,157	18,470	60,500	5.93×10^{-9}	11.3	39.8
insecta-ant-colony1	111,578	113	40	4.41×10^{-1}	1.5	6.1
insecta-ant-colony3	241,280	160	40	4.74×10^{-1}	1.8	8.5
insecta-ant-colony5	194,317	152	40	4.23×10^{-1}	1.5	7.3
mammalia-raccoon-proximity	1997	24	51	1.42×10^{-1}	0.6	0.4
rec-amz-Baby	915,446	596,316	4868	1.06×10^{-9}	8.7	23.5
reptilia-tortoise-network-bsv	554	136	3	2.01×10^{-2}	0.4	0.1
reptilia-tortoise-network-fi	1713	787	8	6.92×10^{-4}	0.8	0.3
SFHH-conf-sensor	70,261	403	3508	2.47×10^{-4}	4.5	13.8

6. Conclusions

This work introduces DLMinTC+, a DL-based algorithm designed to address the MinTCover problem on temporal graphs. Our approach centers on two primary goals: first, to develop a model capable of outperforming traditional heuristic and approximate methods, demonstrating the effectiveness of DL in solving CO problems on dynamic networks; second, to advance the field of temporal knowledge graph analysis by showcasing how DL techniques can capture both temporal and structural nuances, providing robust embeddings that enable meaningful analysis of complex temporal interactions.

DLMinTC+ solves the MinTCover problem in a data-driven fashion, combining GraphSAGE for node embedding, Transformer-based temporal encoding to capture sequential dynamics, and a pointer mechanism to determine optimal start and end points for activity intervals. This architecture is designed to model both structural and temporal features of temporal graphs effectively. To train such network, we define a custom loss function to balance covering, minimizing sum-span and granting integrity of the solution. Despite this, since the output of DL models is non-deterministic, to grant that the computed solution is valid, we also provide a novel iterative algorithm to build a valid final timeline, providing theoretical guarantees on the validity and minimality of the solution.

We test our methodology on synthetic and real instances. The results of our experiments underscore the stability and generalization ability of the model, particularly on dense, highly connected datasets, where it achieves significant coverage efficiency. While DLMinTC+ incurs slightly higher computational time, this trade-off is offset by the model's enhanced coverage precision, making it a viable approach for applications prioritizing solution accuracy.

Looking ahead, several directions for future research can further advance this work: while DLMINTC+ achieves notable improvements in coverage size, the computational cost for large graphs remains a challenge. Future research could explore distributed or parallel processing strategies, more efficient graph partitioning techniques, and hardware-accelerated solutions to enhance computational efficiency. Moreover, although the proposed method achieves strong performance, further work could focus on improving the interpretability of the model's outputs. Techniques such as attention visualization or feature importance analysis can help users better understand the decision-making process and ensure trust in critical applications.

Even though this study paves the way for further exploration of DL methodologies in CO on temporal graphs, suggesting exciting possibilities for scalable and adaptive solutions in dynamic network scenarios, these avenues for future research aim to enhance the scalability, generalizability, and practical utility of DL approaches for temporal CO, paving the way for broader adoption in dynamic network analysis.

Author Contributions: Conceptualization, G.L., R.D., S.M., and I.Z.; methodology, G.L. and I.Z.; software, G.L.; validation, R.D., S.M., and I.Z.; formal analysis, G.L. and I.Z; investigation, G.L.; resources, G.L., R.D., and I.Z.; data curation, G.L. and R.D.; writing—original draft preparation, G.L.; writing—review and editing, R.D., S.M., and I.Z.; supervision, S.M. and I.Z; project administration, S.M. and I.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data are contained within this article.

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

Appendix A. Deep Learning Background

Appendix A.1. Neural Networks

The following is drawn primarily from [40]. Neural networks (NNs) consist of multiple layers of interconnected units, or *neurons*, which transform input data through a series of learned weights and non-linear activation functions. Formally, for a neuron j, the output is given by

$$a_j = f\left(\sum_i w_{ji} x_i + b_j\right)$$

where w_{ji} represents the *weight* of the connection between neuron i and neuron j, b_j is a *bias* term, x_i are the input values, and f is the *activation function*, typically a non-linear function. The activation function introduces non-linearity into the model, allowing it to learn complex patterns and relationships in the data. Some of the most commonly used functions are the sigmoid, the tangent hyperbolic, and the rectified linear unit (ReLU) functions, where $\mathrm{ReLU}(x) = \max(0, x)$.

NNs can be made up of three types of layers: the *input layer*, the *hidden layers*, and the *output layer*. The input layer receives the raw data that the network will process, while the hidden layers, which may consist of many neurons, perform intermediate calculations and transformations. The output layer then provides the final prediction or result of the network. These layers are densely interconnected, meaning each neuron in a given layer is connected to every neuron in the next layer. In the simplest form, information flows in one direction only, from the input layer through the hidden layers to the output layer. In this case, the NN is called a feedforward neural network (FFNN).

Formally, we can express the FFNN as follows. Given an input $x \in \mathbb{R}^N$ and a hidden linear model $h \in \mathbb{R}^D$, the output $o \in \mathbb{R}^M$ of the FFNN is

$$\begin{aligned} h &= g(Wx + b) \\ o &= f(Vh + c) \end{aligned} \tag{A1}$$

where $W \in \mathbb{R}^{N \times D}$ and $V \in \mathbb{R}^{D \times M}$ are learnable weight matrices for input-to-hidden and hidden-to-output connections, respectively, $b \in \mathbb{R}^D$ and $c \in \mathbb{R}^M$ are learnable bias parameters, and g and f are non-linear activation functions, used to convert the output of the FFNN to a conditional probability distribution over the input.

The training of a neural network involves learning the optimal values for the weight matrices W and V, as well as the bias parameters b and c, so that the network can accurately map inputs to the desired outputs. This learning process is guided by a *loss function*, which quantifies the difference between the predicted outputs and the actual target values.

During training, the network iteratively adjusts its weights to minimize the loss function. This process is performed using an optimization algorithm such as *gradient descent*. Gradient descent updates the parameters by moving in the direction of the negative gradient of the loss function L with respect to the weights. Formally, the update rule for a parameter θ at iteration t is given by

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t), \tag{A2}$$

where θ_t is the value of the parameter at iteration t, η is the learning rate, which controls the step size of the updates, and $\nabla_\theta L(\theta_t)$ is the gradient of the loss function with respect to θ at iteration t.

The gradients $\nabla_\theta L$ are computed using *backpropagation*, a technique that applies the chain rule of calculus to efficiently calculate the gradients layer by layer. For a single weight w_{ij} connecting neurons i and j, the chain rule can be expressed as

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial a_j} \cdot \frac{\partial a_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}, \tag{A3}$$

where a_j is the activation of neuron j and z_j is the weighted sum of inputs to neuron j.

While gradient descent is effective, it can suffer from issues such as slow convergence and poor handling of non-convex loss landscapes. To address these challenges, *Adam* was introduced as an enhanced optimization algorithm. Adam combines two key ideas: it maintains an exponentially decaying average of past gradients (also known as momentum), which helps smooth out updates and accelerate convergence in directions with consistent gradients; it adjusts the learning rate for each parameter individually, based on the second moment of past gradients, ensuring stable updates even in the presence of noisy gradients.

The loss function, therefore, plays a crucial role in this process. By providing a measure of how well the network's predictions match the target outputs, it serves as a guide for the optimization algorithm. Common loss functions include mean squared error (MSE) for regression tasks and cross-entropy for classification tasks. The choice of loss function directly impacts the network's ability to learn effectively and generalize to new data, as it defines the objective that the network strives to achieve during training.

Through this iterative process of weight adjustment, the neural network learns to represent complex patterns in the data, ultimately improving its performance in making predictions.

Appendix A.2. Graph Neural Networks

The following is drawn primarily from [41]. Graph Neural Networks (GNNs) are frameworks specifically designed to handle graph data, with the key objective of learning representations of nodes (or entire graphs) that reflect both the structure of the graph and any feature information associated with nodes.

GNNs are based on the *neural message passing* framework, according to which each node aggregates information from its neighbors and updates its own representation iteratively. The process ensures that after k iterations, a node's representation contains information from its k-hop neighborhood.

The message passing process can be formalized as follows:

$$h_u^{(k+1)} = \text{UPDATE}^{(k)}\left(h_u^{(k)}, \text{AGGREGATE}^{(k)}\left(\{h_v^{(k)} \mid v \in N(u)\}\right)\right) \tag{A4}$$

where$h_u^{(k)}$ represents the embedding of node u at the k-th iteration, and $N(u)$ is the neighborhood of node u. The *AGGREGATE* function aggregates the embeddings of node u's neighbors, while the *UPDATE* function combines the aggregated message with the current embedding of node u to produce the updated embedding.

A common approach in GNNs is to define simple *AGGREGATE* and *UPDATE* functions, like the sum operator for the embeddings of neighboring nodes, and a non-linear transformation, such as an FFNN for the update.

In this case, the aggregation and update steps for the k-th layer of a GNN can be written as

$$h_u^{(k+1)} = g\left(W^{(k)}\left(\sum_{v \in N(u)} h_v^{(k)}\right)\right) \tag{A5}$$

where g is a non-linear activation function (e.g., ReLU), and $W^{(k)}$ is a learnable weight matrix at the k-th layer.

A widely used GNN variant is the Graph Convolutional Network (GCN), which extends the concept of convolution from Euclidean spaces (such as images) to graph-structured data. The GCN aggregates information from neighboring nodes using a normalized adjacency matrix to stabilize training and ensure that information from high-degree nodes does not dominate the aggregation process.

The GCN update rule is given by

$$h_u^{(k+1)} = g\left(\sum_{v \in N(u) \cup \{u\}} \frac{h_v^{(k)}}{\sqrt{d_u d_v}} W^{(k)}\right) \tag{A6}$$

where d_u and d_v are the degrees of nodes u and v, respectively. The sum is taken over both the neighbors of u and u itself (self-loop). The adjacency matrix is normalized symmetrically to prevent the aggregation from being dominated by nodes with high degrees.

GCNs have proven effective for node classification tasks, particularly in semi-supervised settings, where only a portion of the node labels are available for training.

To address some of the limitations of GCNs, mainly their inability to generalize to unseen nodes, the Graph SAmple and AGgregate (GraphSAGE) framework was introduced [34]. Instead of aggregating information from all neighbors, GraphSAGE samples a fixed-size set of neighbors for each node during the aggregation process. Additionally, GraphSAGE introduces the idea of concatenating the node's own embedding with the aggregated neighborhood information, which helps preserve node-specific features during the message-passing process.

The GraphSAGE update rule is defined as

$$h_u^{(k+1)} = g\left(W^{(k)}\left(h_u^{(k)} \,\|\, \sum_{v \in N(u)} h_v^{(k)}\right)\right) \tag{A7}$$

where $\|$ denotes vector concatenation. Thanks to this sampling process, GraphSAGE enables inductive learning, allowing to generate embeddings for unseen nodes during inference and enhances scalability by limiting the computational and memory requirements typically associated with GNNs on large graphs. This approach also helps mitigate the issue of *over-smoothing*, where after several iterations, node embeddings can become too similar to one another.

Appendix A.3. Attention and Transformer

The attention mechanism [42] was introduced to enhance encoder–decoder models for machine translation by dynamically weighting input elements instead of treating them equally. Traditional models generate a latent representation from an input sequence and use it to produce an output sequence. However, they lack a mechanism to prioritize relevant parts of the input.

Attention introduces a *context vector*, computed at each decoding step to focus on the most relevant input elements. This involves calculating alignment scores $e_{t,i}$ using a function $a(.)$, applying softmax normalization to obtain weights $\alpha_{t,i}$, and computing c_t as a weighted sum of encoder hidden states:

$$\begin{aligned} e_{t,i} &= a(s_{t-1}, h_i) \\ \alpha_{t,i} &= softmax(e_{t,i}) \\ c_t &= \sum_{i=1}^{T} \alpha_{t,i} h_i \end{aligned} \tag{A8}$$

This mechanism has been generalized [43] for broader sequence-to-sequence tasks using *queries*, *keys*, and *values*. Here, queries correspond to previous decoder outputs, while keys and values derive from encoded inputs. The goal is to identify which keys are most relevant to a query and assign alignment scores accordingly. The context vector is computed as

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{A9}$$

where d_k is the dimension of queries and keys. This formulation enables efficient computation across multiple queries, keys, and values simultaneously, significantly improving performance in tasks like machine translation and beyond.

Based on the attention mechanism, the *Transformer* architecture was introduced [43], extending it significantly to handle long-range dependencies and process sequences in a parallelizable manner, making it much faster and more efficient than previous models. The Transformer uses a stack of encoders and decoders, each composed of self-attention layers and FFNN, to transform the input sequence into the output sequence without relying on any recurrent connections.

Indeed, in the Transformer, both the *encoder* and *decoder* are composed of a stack of identical layers (typically 6). Each layer in the encoder has two primary sub-layers:

- A *multi-head self-attention mechanism*, which allows the model to focus on different parts of the input sequence simultaneously and can be expressed as:

$$\begin{aligned} MultiHead(Q,K,V) &= Concat(head_1,\ldots,head_h)W^O \\ head_i &= Attention(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{A10}$$

Here, W_i^Q, W_i^K, W_i^V are learned projection matrices, and W^O is the final output projection matrix.

- A *position-wise fully connected FFNN* that applies non-linear transformations independently to each position in the sequence, thus taking care of the position of tokens in the input sequence. The positional encoding for position *pos* and dimension *i* is given by:

$$\begin{aligned} \mathrm{PE}(pos, 2i) &= \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \\ \mathrm{PE}(pos, 2i+1) &= \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \end{aligned} \tag{A11}$$

This sinusoidal function enables the model to learn position-sensitive patterns and can generalize to sequences longer than those encountered during training.

The decoder, in addition to these two sub-layers, also contains an additional sub-layer that performs multi-head attention over the encoder's output. This third layer is essential as it enables the decoder to attend to the relevant portions of the input sequence.

Both the encoder and decoder use *residual connections* around each of the sub-layers, followed by layer normalization to ensure stable learning. Each sub-layer can be expressed as $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$ where x is the input to the sub-layer, and *Sublayer(x)* represents the function applied by the sub-layer (either self-attention or feedforward).

References

1. Holme, P.; Saramäki, J. Temporal Networks. *arXiv* **2011**. https://doi.org/10.48550/arXiv.1108.1780.
2. Michail, O. An Introduction to Temporal Graphs: An Algorithmic Perspective. *Internet Math.* **2016**, *12*, 239–280. https://doi.org/10.1080/15427951.2016.1177801.
3. Peng, Y.; Choi, B.; Xu, J. Graph Learning for Combinatorial Optimization: A Survey of State-of-the-Art. *Data Sci. Eng.* **2021**, *6*, 119–141. https://doi.org/10.1007/S41019-021-00155-3.
4. Hosseinzadeh, M.M.; Cannataro, M.; Guzzi, P.H.; Dondi, R. Temporal networks in biology and medicine: A survey on models, algorithms, and tools. *Netw. Model. Anal. Health Inform. Bioinform.* **2023**, *12*, 10. https://doi.org/10.1007/S13721-022-00406-X.
5. Yu, J.; Xiao, B.; Cui, Y. A Review of Temporal Network Analysis and Applications. In *3D Imaging—Multidimensional Signal Processing and Deep Learning*; Patnaik, S., Kountchev, R., Tai, Y., Kountcheva, R., Eds.; Springer: Singapore, 2023; pp. 1–10.
6. Rozenshtein, P.; Tatti, N.; Gionis, A. The network-untangling problem: From interactions to activity timelines. *Data Min. Knowl. Discov.* **2021**, *35*, 213–247. https://doi.org/10.1007/S10618-020-00717-5.
7. Froese, V.; Kunz, P.; Zschoche, P. Disentangling the Computational Complexity of Network Untangling. *arXiv* **2022**. https://doi.org/10.48550/ARXIV.2204.02668.
8. Dondi, R.; Lafond, M. An FPT Algorithm for Temporal Graph Untangling. In Proceedings of the 18th International Symposium on Parameterized and Exact Computation, IPEC 2023, Amsterdam, The Netherlands, 6–8 September 2023; Misra, N., Wahlström, M., Eds.; LIPIcs; Schloss Dagstuhl—Leibniz-Zentrum für Informatik: Wadern, Germany, 2023; Volume 285, pp. 12:1–12:16. https://doi.org/10.4230/LIPICS.IPEC.2023.12.
9. Dondi, R. Untangling temporal graphs of bounded degree. *Theor. Comput. Sci.* **2023**, *969*, 114040. https://doi.org/10.1016/J.TCS.2023.114040.
10. Dondi, R.; Popa, A. Exact and approximation algorithms for covering timeline in temporal graphs. In *Annals of Operations Research*; Springer: Berlin/Heidelberg, Germany, 2024. https://doi.org/10.1007/s10479-024-05993-8.

11. Lazzarinetti, G.; Manzoni, S.; Zoppis, I.; Dondi, R. FastMinTC+: A Fast and Effective Heuristic for Minimum Timeline Cover on Temporal Networks. In Proceedings of the 31st International Symposium on Temporal Representation and Reasoning, TIME 2024, Montpellier, France, 28–30 October 2024; Sala, P., Sioutis, M., Wang, F., Eds.; LIPIcs; Schloss Dagstuhl—Leibniz-Zentrum für Informatik: Wadern, Germany, 2024; Volume 318, pp. 20:1–20:18. https://doi.org/10.4230/LIPICS.TIME.2024.20.
12. Liu, Y.; Safavi, T.; Dighe, A.; Koutra, D. Graph Summarization Methods and Applications: A Survey. *ACM Comput. Surv.* **2018**, *51*, 62:1–62:34. https://doi.org/10.1145/3186727.
13. Paranjape, A.; Benson, A.R.; Leskovec, J. Motifs in Temporal Networks. In Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, Cambridge , UK, 6–10 February 2016.
14. Hulovatyy, Y.; Chen, H.; Milenkovic, T. Exploring the structure and function of temporal networks with dynamic graphlets. *Bioinformatics* **2016**, *32*, 2402. https://doi.org/10.1093/BIOINFORMATICS/BTW310.
15. Shah, N.; Koutra, D.; Zou, T.; Gallagher, B.; Faloutsos, C. TimeCrunch: Interpretable Dynamic Graph Summarization. In Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, Australia, 10–13 August 2015; Cao, L., Zhang, C., Joachims, T., Webb, G.I., Margineantu, D.D., Williams, G., Eds.; ACM: New York, NY, USA, 2015; pp. 1055–1064. https://doi.org/10.1145/2783258.2783321.
16. Wackersreuther, B.; Wackersreuther, P.; Oswald, A.; Böhm, C.; Borgwardt, K.M. Frequent subgraph discovery in dynamic networks. In Proceedings of the Eighth Workshop on Mining and Learning with Graphs, MLG'10, Washington, DC, USA, 24–25 July 2010; Brefeld, U., Getoor, L., Macskassy, S.A., Eds.; ACM: New York, NY, USA, 2010; pp.155–162. https://doi.org/10.1145/1830252.1830272.
17. Pietiläinen, A.K.; Diot, C. Dissemination in opportunistic social networks: The role of temporal communities. In Proceedings of the Thirteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc'12, Hilton Head, SC, USA,. 11–14 June 2012; p. 165–174. https://doi.org/10.1145/2248371.2248396.
18. Dondi, R.; Popa, A. Timeline Cover in Temporal Graphs: Exact and Approximation Algorithms. In Proceedings of the Combinatorial Algorithms—34th International Workshop, IWOCA 2023, Tainan, Taiwan, 7–10 June 2023; Hsieh, S., Hung, L., Lee, C., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2023; Volume 13889, pp. 173–184. https://doi.org/10.1007/978-3-031-34347-6_15.
19. Karakostas, G. A better approximation ratio for the vertex cover problem. *ACM Trans. Algorithms* **2009**, *5*, 41:1–41:8. https://doi.org/10.1145/1597036.1597045.
20. Hochbaum, D.S. Approximation Algorithms for the Set Covering and Vertex Cover Problems. *SIAM J. Comput.* **1982**, *11*, 555–556. https://doi.org/10.1137/0211045.
21. Langedal, K.; Langguth, J.; Manne, F.; Schroeder, D.T. Efficient Minimum Weight Vertex Cover Heuristics Using Graph Neural Networks. In Proceedings of the 20th International Symposium on Experimental Algorithms, SEA 2022, Heidelberg, Germany, 25–27 July 2022; Schulz, C., Uçar, B., Eds.; LIPIcs; Schloss Dagstuhl—Leibniz-Zentrum für Informatik: Wadern, Germany, 2022; Volume 233, pp. 12:1–12:17. https://doi.org/10.4230/LIPICS.SEA.2022.12.
22. Khalil, E.B.; Dai, H.; Zhang, Y.; Dilkina, B.; Song, L. Learning Combinatorial Optimization Algorithms over Graphs. In Proceedings of the Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, Long Beach, CA, USA, 4–9 December 2017; Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R., Eds.; Neural Information Processing Systems Foundation, Inc. (NeurIPS): La Jolla, CA, USA, 2017; pp. 6348–6358.
23. Bello, I.; Pham, H.; Le, Q.V.; Norouzi, M.; Bengio, S. Neural Combinatorial Optimization with Reinforcement Learning. In Proceedings of the 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, 24–26 April 2017.
24. Vesselinova, N.; Steinert, R.; Perez-Ramirez, D.F.; Boman, M. Learning Combinatorial Optimization on Graphs: A Survey with Applications to Networking. *IEEE Access* **2020**, *8*, 120388–120416. https://doi.org/10.1109/ACCESS.2020.3004964.
25. Gunarathna, U.; Borovica-Gajic, R.; Karunasekera, S.; Tanin, E. Solving Dynamic Graph Problems with Multi-Attention Deep Reinforcement Learning. *arXiv* **2022**. http://arxiv.org/abs/2201.04895.
26. Trivedi, R.; Farajtabar, M.; Biswal, P.; Zha, H. Dyrep: Learning representations over dynamic graphs. In Proceedings of the International Conference on Learning Representations, New Orleans, LA, USA, 6–9 May 2019.
27. Xu, D.; Ruan, C.; Korpeoglu, E.; Kumar, S.; Achan, K. Inductive representation learning on temporal graphs. In Proceedings of the International Conference on Learning Representations, Addis Ababa, Ethiopia, 30 April 2020.
28. Pareja, A.; Domeniconi, G.; Chen, J.; Ma, T.; Suzumura, T.; Kanezashi, H.; Kaler, T.; Leiserson, C.E.; Bader, D.A. EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020.
29. Li, Y.; Yu, R.; Shahabi, C.; Liu, Y. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. In Proceedings of the International Conference on Learning Representations, Vancouver, BC, Canada, 30 April–3 May 2018.
30. Chen, Y.; Zhang, J.; Qin, Z. TREND: Temporal regularization for dynamic networks. In Proceedings of the 28th ACM International Conference on Information and Knowledge Management, Beijing, China, 3–7 November 2019.

31. Rossi, E.; Chamberlain, B.P.; Eynard, D.; Rowbottom, J.; Bronstein, M. Temporal Graph Networks for Deep Learning on Dynamic Graphs. In Proceedings of the International Conference on Machine Learning, Online, 13–18 July 2020.
32. Zhang, S.; Zhang, X.; Nie, F.; Zhang, J. TMac: A Multi-Aspect Collaborative Neural Network for Dynamic Graph Learning. In Proceedings of the 2020 SIAM International Conference on Data Mining, Cincinnati, OH, USA, 7–9 May 2020.
33. Wang, C.; Yao, H.; Zhao, S.; Wang, Z.; Wu, F.; Li, Z.; Wang, C.; Jiang, X. Temporal Graph Collaborative Transformer for Dynamic Graph Learning. In Proceedings of the 2021 SIAM International Conference on Data Mining, Virtual, 29 April–1May 2021.
34. Hamilton, W.; Ying, Z.; Leskovec, J. Inductive representation learning on large graphs. In Proceedings of the Advances in Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017.
35. Kumar, S.; Zhang, X.; Leskovec, J. Predicting dynamic embeddings for link prediction in evolving networks. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Anchorage, AK, USA, 4–8 August 2019.
36. Sankar, A.; Wu, Y.; Gouda, Z.H.; Qi, Y.; Jure, L. DySAT: Dynamic Self-Attention Network for Temporal Graphs. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Virtual, 6–10 July 2020.
37. Vinyals, O.; Fortunato, M.; Jaitly, N. Pointer Networks. In Proceedings of the Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, Montreal, QC, Canada, 7–12 December 2015; Cortes, C., Lawrence, N.D., Lee, D.D., Sugiyama, M., Garnett, R., Eds.; Neural Information Processing Systems Foundation, Inc. (NeurIPS): La Jolla, CA, USA, 2015; pp. 2692–2700.
38. Rossi, R.A.; Ahmed, N.K. The Network Data Repository with Interactive Graph Analytics and Visualization. In Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, Austin, TX, USA, 25–30 January 2015; Bonet, B., Koenig, S., Eds.; AAAI Press: Palo Alto, CA, USA, 2015; pp. 4292–4293. https://doi.org/10.1609/AAAI.V29I1.9277.
39. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. In Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, 7–9 May 2015; Conference Track Proceedings; Bengio, Y., LeCun, Y., Eds.; DBLP: San Diego, CA, USA, 2015.
40. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016. http://www.deeplearningbook.org (accessed on 12 January 2024).
41. Hamilton, W.L. *Graph Representation Learning*; Synthesis Lectures on Artificial Intelligence and Machine Learning; Morgan & Claypool Publishers: San Rafael, CA, USA, 2020. https://doi.org/10.2200/S01045ED1V01Y202009AIM046.
42. Bahdanau, D.; Cho, K.; Bengio, Y. Neural Machine Translation by Jointly Learning to Align and Translate. In Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, 7–9 May 2015; Conference Track Proceedings; Bengio, Y.; LeCun, Y., Eds.; DBLP: San Diego, CA, USA, 2015.
43. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention is All you Need. In Proceedings of the Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, Long Beach, CA, USA; 4–9 December 2017; Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R., Eds.; Neural Information Processing Systems Foundation, Inc. (NeurIPS): La Jolla, CA, USA, 2017; pp. 5998–6008.