Research Article

An Efficient Dynamic Programming Algorithm for Finding Group Steiner Trees in Temporal Graphs

Youming Ge 1, Zitong Chen 1, Weiyang Kong 1, Yubao Liu 1,2, Raymond Chi-Wing Wong 3, and Sen Zhang 1

1 School of Computer Science and Engineering, Sun Yat-Sen University, Guangzhou 510275, China
2 Guangdong Key Laboratory of Big Data Analysis and Processing, Guangzhou 510006, China
3 Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong 999077, China

Correspondence should be addressed to Yubao Liu; liuyubao@mail.sysu.edu.cn

Received 16 November 2022; Revised 28 August 2023; Accepted 31 October 2023; Published 21 November 2023

Copyright © 2023 Youming Ge et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The computation of a group Steiner tree (GST) in various types of graph networks, such as social network and transportation network, is a fundamental graph problem in graphs, with important applications. In these graphs, time is a common and necessary dimension, for example, time information in social network can be the time when a user sends a message to another user. Graphs with time information can be called temporal graphs. However, few studies have been conducted on GST in terms of temporal graphs. This study analyzes the computation of GST for temporal graphs, i.e., the computation of temporal GST (TGST), which is shown to be an NP-hard problem. We propose an efficient solution based on a dynamic programming algorithm for our problem. This study adopts new optimization techniques, including graph simplification, state pruning, and A* search, are adopted to dramatically reduce the algorithm search space. Moreover, we consider three extensions for our problem, namely the TGST with unspecified tree root, the progressive search of TGST, and the top-N search of TGST. Results of the experimental study performed on real temporal networks verify the efficiency and effectiveness of our algorithms.

1. Introduction

Temporal networks or graphs with time information have attracted considerable research attention [1–13]. Some important categories of temporal graphs include communication networks, social networks, transportation networks, and neural networks. The computation of group Steiner trees (GSTs) is a fundamental graph problem with important applications, such as the VLSI design in industrial applications, the keyword search in relational databases, and the finding of a team of experts in social networks [14–20]. However, only few studies have been conducted on GST in terms of a temporal graph.

This study analyzes the computation of GST for temporal graphs, i.e., the computation of temporal GST (TGST). Calculating GST for temporal graphs is quite useful, for example, querying the relationship between authors in DBLP throughout various time intervals, evaluating the propagation of viruses or messages in social networks, and querying routes to a group of POIs in a public transit network.

Based on the abovementioned application, we studied the computation of GSTs in temporal graphs, namely TGST. Specifically, given a weighted and labeled temporal graph G, where each vertex is associated with a set of labels, our approach searches for the minimum-weighted connected tree from G that covers all the specified given labels.

In reality, the conventional GST in a static (nontemporal) graph is a classical problem. However, the existing GST solutions cannot be suitable for our problems because the temporal information is not considered. Therefore, the dynamic programming algorithm is often used in the existing solutions. Ding et al. [18] presented a parameterized dynamic programming algorithm has been presented which takes $O(3^k \cdot n + 2^k \cdot (n \cdot \log n + m))$ time, where $k$ is the number of specified labels, and $m$ and $n$ are the number of edges and vertices of the graph, respectively. Owing to the exponential time
complexity, the parameterized algorithm is impractical even for a very small value of \( k \) (e.g., \( k = 8 \)) in large graphs. Therefore, an improved dynamic algorithm was devised in [20].

Similar to the existing methods, we present an efficient solution based on a dynamic programming algorithm for our problem. Specifically, we define a state as a connected tree in the given temporal graph. Our approach differs from those proposed in [18, 20] in that our state contains the directed edges associated with time information. Moreover, we propose three state-transition operations, namely, edge-growth, tree-merger, and path-growth. Similar to the approaches in [18, 20], the edge-growth combines a state with an edge into a new state, and tree-merger combines two states into a new state. In particular, the path-growth corresponds to a series of edge-growth and tree-merger operations, which can reduce the number of states to be generated. Consider the huge size of a temporal graph. We propose a new method called graph simplification and state pruning technique. Specifically, our main contribution is the proposed graph simplification and state pruning technique for TGST.

2.1. Temporal Graph. Let \( G = (V, E) \) be a temporal graph, where \( V \) (\( E \)) is the set of vertex (edge) of \( G \). In particular, there may be multiple directed edges between two vertices in \( G \). Each edge \( e \in E \) is defined to be a quintuple which includes a starting vertex \( u \), an end vertex \( v \), a starting time \( t_s \), an arrival time \( t_a \), and a weight value \( w \), denoted as \( e = (u, v, t_s, t_a, w) \), where \( t_s, t_a, w \) are nonnegative real numbers, and \( t_s \leq t_a \). The starting vertex, starting time, arrival time, and weight value of \( e \) are also denoted by \( (s(e), a(e), t_s(e), t_a(e), w(e)) \), namely \( s(e) = u, a(e) = v, t_s(e) = t_a(e) = t_a, \) and \( w(e) = w \). For simplicity, we sometimes write \( e = (u, v) \).

For any vertex \( u \in V \), let \( E^u(u) \) and \( E_o(u) \) denote the sets of in-edges and the set of out-edges of \( u \), respectively. Then, \( E^u(u) = \{ e | a(e) = u, e \in E \}, \) and \( E_o(u) = \{ e | s(e) = u, e \in E \} \). The in-degree and out-degree of \( u \) are equal to \( |E^u(u)| \) and \( |E_o(u)| \), respectively.

Example 1. Figure 1(a) illustrates a temporal graph containing eight vertices numbered 0–7 and 13 edges, \( e_1, e_2, e_3, \ldots, e_{13} \). Each edge is labeled with a serial number, \([t_s, t_a, w] \) and \([u] \). In this example, we set the edge weight \( w \) as the time duration, namely \( t_a - t_s \). In practice, \( w \) can be set as any value. In this figure, the edge \( e_1 = (0, 1, 1, 5, 4) \) indicates a bus route from 0 to 1 with the departure time \( t_0 = 1 \), the arrival time \( t_1 = 5 \), and the cost \( w = 4 \). Furthermore, the set of in-edges and out-edges of the vertex 1 is defined as \( E^1(1) = \{ e_1, e_7 \} \) and \( E_o(1) = \{ e_3, e_5, e_6 \} \), respectively.

Path \( P = \langle e_1, e_2, \ldots, e_k \rangle \) in \( G \) is a sequence of edges such that \( a(e_i) = s(e_{i-1}) \) and \( t_a(e_i) \leq t_s(e_{i+1}) \) where \( 1 \leq i < k \). We say that \( P \) is a path from the vertex \( s(e_1) \) to the vertex \( a(e_k) \). The starting edge and end edge of \( P \) are \( e_1 \) and \( e_k \), respectively. We denote \( t_s(e_i) \) as the starting time and \( t_a(e_i) \) as the arrival time of \( P \), calculated as \( \delta(P) = \sum_{e \in P} w(e) \).

For example, in Figure 1(a), the sequence of \( \langle e_1, e_2 \rangle \) is not a path as \( t_a(e_1) = 5 > t_s(e_2) = 3 \). The sequence \( \langle e_1, e_3 \rangle \) is a path, denoted by \( P_1 \). Then, \( \delta(P_1) = w(e_1) + w(e_2) + w(e_3) = 6 \), \( \delta(P) = 3 \), and \( \delta(P) = t_a(e_2) = 8 \).

We call requirement \( \delta(P) \leq \delta(P) \) as the path time constraint. For example, in transportation networks, each directed edge implies a bus route, and the departure time of \( e_i \) initiates after the arrival time of \( e_i \). Moreover, the edge durations may be denoted as zero for some temporal graphs such as e-mail communication and Facebook wall posting.

We can say that a path \( P \) is satisfied with the time interval \([t_0, t_f] \) if \( \delta(P) \leq t_f \) and \( \delta(P) \leq t_f \). We say \( v \) is reachable from \( u \) in \([t_0, t_f] \) if there is a path from \( u \) to \( v \) that is satisfied by \([t_0, t_f] \).

We say that a connected tree \( T \) in \( G \) is satisfied with \([t_0, t_f] \) if all paths from the root to each leaf vertex are reachable in \([t_0, t_f] \). Similarly, the weight or cost of \( T \) is defined as the sum of the weight of each edge included in \( T \), namely, \( W(T) = \sum_{e \in T} w(e) \).
2.2. Our Problem

Definition 2. (TGST). Given a temporal graph \( G = (V, E) \), each vertex \( v \in V \) is associated with a set of labels \( L_v \). Let \( \mathcal{L} = \cup_{v \in V} L_v \) be the set of all labels. Given a root \( r \in V \), a set of query labels \( L \subseteq \mathcal{L} \), and a time constraint \( [t_a, t_b] \), our approach involves the finding of a minimum weight tree, \( T \), from \( G \) that is rooted at \( r \), satisfied by \( [t_a, t_b] \), and contains all labels in \( L \). For simplicity, we assume that \( L_r = \emptyset \), as we can always exclude \( L_r \) from \( L \).

Consider Figure 1(a) as an example, where vertex 0 is defined as root \( r \), and \( L_1 = \{a, c\} \), \( L_2 = \{a\} \), \( L_3 = L_4 = \{b\} \), \( L_5 = \{c\} \), and \( L_2 = L_3 = \{d\} \). Suppose that time constraint \( [t_a, t_b] \) is set as \([0, \infty)\), and the set of query labels is \( L = \{a, b, c, d\} \). Then, we obtain an optimal solution, \( T \), as shown in Figure 1(b), the weight of which is denoted as \( \mathcal{W}(T) = \omega(e_1) + \omega(e_2) + \omega(e_4) + \omega(e_5) = 4 \).

Let \( \mathcal{P}(r, v) \) be the set of all paths from \( r \) to \( v \) that are satisfied with \( [t_a, t_b] \). If \( P \in \mathcal{P}(r, v) \) has the earliest arrival time among all paths in \( \mathcal{P}(r, v) \), we define arrival time \( \mathcal{A}(P) \) as the earliest arrival time for \( v \), denoted by \( \mathcal{A}(v) \). In particular, if there is no path from \( r \) to \( v \), we set \( \mathcal{A}(v) = \infty \). For example, in Figure 1(a), let \( r \) be the vertex 0, \( \mathcal{A}(7) = 9 \), and \( \mathcal{A}(6) = \infty \).

The existing GST problem in a weighted and labeled graph, in which the vertices with the same labels are in the same group, is the generation of the Steiner tree problem [21]. The existing GST problem is known to be NP-hard [14]. As additional time information is contained in a timetable graph, the existing GST problem can be viewed as a special case of our problem. Thus, we define the following theorem.

Theorem 3. Our problem is NP-hard.

2.3. Motivation for TGST. Our work is motivated by a number of applications. A TGST can identify user relationships at different time intervals in social networks. A TGST is also useful for the study of epidemiology, the paths of infectious diseases (or computer virus), when the network is about individual contacts (or a computer network). Another application is the transportation problem cited in [22]. A TGST minimizes the total cost to transport some given POIS. Some motivation applications are as follows.

Motivation 1. Consider a temporal network record when two nodes have interacted such as virus infection or e-mail forwarding. A TGST can be adopted for reconstructing the flow of epidemic propagating in temporal networks. That is, given a temporal network in which a virus has been propagating over time, we can identify the starting points of the epidemic and tell when every node got infected. In particular, each node can be associated with the types of viruses. In order to better understand the real viruses spread over time, we consider the spread of related types of viruses. When one virus spreads to a node such as the coronavirus, it is possible that this virus will not appear in the next node, but the related virus could appear such as the helicobacter pylori. If the edge’s weight is set as the affected time of viruses, a TGST can tell the propagation paths of the specified types of viruses with minimum affected time. A case study for this observation is given in Section 5.

Motivation 2. We consider an application of the DBLP graph, where each vertex corresponds to an author or a paper and each edge denotes a certain relationship (e.g., coauthor/cited by/written by) among the authors and the papers. An edge is more important if it has a smaller weight. Our solution can be used to find out the closest connections from the DBLP graph that covers the given authors within a given period. For example, from 2016 to 2018, Tom and Peter coauthored three papers that were cited by one paper written by Jack in 2019. The found connections can be highly useful and important for network analysis.

Motivation 3. A public transportation network can be modeled as a timetable graph [23], where each node represents a station, and each directed edge is associated with a timetable that records the departure (resp. arrival) time of each vehicle from a station to another. Moreover, each node is associated with some POI category keywords or labels, and each edge has a travel cost weight such as the distance and time duration. A TGST minimizes the total cost to transport some given resource from a given location to all the stations associated with the given set of POIs.
### 3. Our Solution

In summary, our solution adopts a DP paradigm similar to that proposed in [18, 20], in that we utilize the similar state representation and state-transition equation. In fact, the approach used in [20] is an improved version of that in [18] for an undirected static graph, in which some optimization techniques are proposed to reduce the search space.

To design our solution, we considered the following naive method based on an adaption version of [20]. First, we obtained a state queue, in which a set of initial states are stored for each vertex and query label. In addition, an upper bound weight of a feasible solution is stored. Second, the method is used to update the state queue and upper bound. However, studies have yet to devise a method to reduce the generated states in the search space ([20]).

\[ E = \{ (u, v, t_u, t_v, w) \in E \mid t_u < t_v \} \]

The correctness of Lemma 4 can be easily verified based on our problem definition. Lemma 4 shows that from \( G \), vertices with out-degrees equal to zero and not containing labels in \( L \) can be easily removed. For example, as shown in Figure 1(a), if \( L = \{c\} \), vertices 4, 6, and 7 can be removed. We can continue to remove vertex 3 in the remaining graph as it does not contain the labels in \( L \) and its out-degree is zero. The removal process is then halted as there are no such vertices remaining.

For any vertex \( v \in V \), we can obtain the earliest arrival time for \( v \) (i.e., \( \mathcal{A}(v) \)) by using the time-minimum spanning tree algorithm with linear time [3]. Then, we achieve the following Lemma 5.

**Lemma 5.** Given a temporal graph \( G = (V, E) \) and a root \( r \in V \), for any edge \( e \in E \), if \( t_e(s) < \mathcal{A}(s(e)) \), \( e \) can be removed from \( G \).

**Proof.** For any edge \( e = (u, v, t_u, t_v, w) \in E \), we have \( t_e(s) = t_u \) and \( s(e) = u \). Then, \( \mathcal{A}(s(e)) = \mathcal{A}(u) \) corresponds to the earliest arrival time among all paths from the root \( r \) to \( u \). As \( t_e < \mathcal{A}(u) \), the path time constraint does not hold. Therefore, edge \( e \) cannot be included in any path from \( r \) to \( v \), and it can thus be removed.

By using Lemma 5, we can continue removing edges from \( G \). For example, in Figure 1(a), let vertex 0 be the root.

Specifically, the key steps of graph simplification are as follows.

1. **Step 1:** Simplify the temporal graph according to time constraint \([t_\alpha, t_\beta]\). For each edge \( e \in E \), if \( t_e(t_\alpha) \leq t_e(s) \leq t_e(t_\beta) \) does not hold, remove the edge \( e \) from \( G \).
2. **Step 2:** Simplify the temporal graph according to the earliest arrival time
   1. Use the time-minimum spanning tree algorithm [3] to obtain the earliest arrival time \( \mathcal{A}(v) \) for each vertex \( v \in V \).
   2. For each edge \( e \in E \), if \( t_e(s) < \mathcal{A}(s(e)) \), remove the edge \( e \) from \( G \).
3. **Step 3:** Simplify the temporal graph according to the set of query labels.
   1. Traverse \( G \) and find all vertices, with zero as the out-degree and which do not contain labels in \( L \). Store these vertices in queue \( Q \).
   2. For each vertex \( v \in Q \), remove \( v \) and remove all edges in \( E_\alpha(v) \) from \( G \). Then, for each edge \( e \in E_\alpha(v) \), decrease the out-degree of the vertex \( s(e) \). If the out-degree of \( s(e) = 0 \) and \( s(e) \) does not contain any label in \( L \), add \( s(e) \) into \( Q \).
   3. \( Q \) is repeatedly checked until \( Q = \emptyset \).

After the key steps are run, there may exist isolated vertices that do not connect with other vertices; these must also be removed. Then, we obtain a simplified temporal graph denoted by \( G'(V', E') \). The main time cost for both the first and third steps is equal to the cost for traversing the

---

**Graph Simplification.** First, we introduce some important properties that are the basis of graph simplification.

**Lemma 4.** Given a temporal graph \( G = (V, E) \) and a set of query labels \( L \), for any vertex \( v \in V \), if \( L_v \cap L = \emptyset \) and \( E_v(v) = \emptyset \), \( v \) can be removed from \( G \).
3.2. State Pruning. In our algorithm, a state denoted by \((v, X, t)\) corresponds to a connected tree in \(G\) that is rooted at \(v\) and covers all labels in \(X\), where \(t\) is the earliest starting time among the out-edges of \(v\) contained in the tree. Then, let \(T(v, X, t)\) denote the tree with the minimum weight among all such trees. For example, state \((0, \{a, b, c, d\}, t(e_4))\) corresponds to the tree in Figure 1(b). As the tree has the minimum weight, it corresponds to \(T(0, \{a, b, c, d\}, t(e_4))\).

Then, the state-transition equation is formulated as follows:

\[
T(v, X, t) = \min\left\{ T_p(v, X, t), T_m(v, X, t) \right\},
\]

\[
T_p(s(e), X, t(s(e))) = \min_{e \in E, e \not\in \{s(e)\}}\{e \in E, e(e) \leq t\},
\]

\[
T_m(v, X, t) = \min_{X \subseteq V, e \in E} \{T(v, X_1, t_1) \circ T(v, X_2, t_2)\},
\]

where \(X = X_1 \cup X_2\),

\[
t = \min\{t_1, t_2\}.
\]

As shown in equation (1), \(T(v, X, t)\) is constructed from either \(T_p(v, X, t)\) or \(T_m(v, X, t)\) with the minimum weight, which is constructed using the edge-growth operation in equation or the tree-merger operation in equation (3).

Specifically, for any state \((u, X, t)\), the edge-growth operation must be used to examine each in-edge of \(u\) individually and then construct a new tree \(T_p(s(e), X, t(s(e)))\) by combining edge \(e\) and tree \(T(u, X, t)\), where \(t(s(e)) \leq t\). Note that label set \(X\) is unchanged in the edge-growth operation. However, for any two states \((v, X_1, t_1)\) and \((v, X_2, t_2)\) with the same root \(v\), if \(X_1 \cap X_2 = \emptyset\), the tree-merger operation merges both trees \(T(v, X_1, t_1)\) and \(T(v, X_2, t_2)\) into a new tree, \(T_m(v, X, t)\), where \(X = X_1 \cup X_2\) and \(t = \min\{t_1, t_2\}\). The set of labels of the expanded state then becomes larger in the tree-merger operation.

3.2.1. Pruning by Using the Path-Growth Operation (P1). In general, the path-growth operation merges a state with a path instead of an edge. As a path often contains multiple edges, the path-growth operation corresponds to a series of edge-growth and tree-merger operations. Then, the number of states to be generated by the path-growth operation can be reduced.

Definition 9 (State path). For any state \((v, X, t)\), let \(P(v, t) = \{P | P \in P(v) \land \delta(P) \leq t\}\). If \(P_{vf} \in P(v, t)\) has the minimum weight among all paths in \(P(v, t)\), \(P_{vf}\) can be called the state path of \((v, X, t)\).

For any state \((v, X, t)\), there are two cases. Case 1: \(|v| = P_{vf} \cap T(v, X, t)\). Case 2: \(|v| < P_{vf} \cap T(v, X, t)\). First, we consider Case 1. Let \(\psi(P_{vf})\) denote the set of labels contained in the state path \(P_{vf}\). The path-growth operation is formulated as follows.

Definition 10 (Path-growth). For any state \((v, X, t)\), if \(P_{vf} \cap T(v, X, t) = \{v\}\), the path-growth operation on \((v, X, t)\) merges \(P_{vf}\) and \(T(v, X, t)\) into a new tree, and it is defined as \(P_{vf} \circ T(v, X, t) = (r, X', \delta(P_{vf}))\), where \(X' = X \cup \psi(P_{vf})\).

Lemma 11. For any state \((v, X, t)\), if \(X \cup \psi(P_{vf}) = L\), the path-growth operation on \((v, X, t)\) is a feasible solution that has the minimum weight among all feasible solutions containing \(T(v, X, t)\).

Proof. According to Definition 10, \(P_{vf} \circ T(v, X, t) = (r, X', \delta(P_{vf}))\), where \(X' = X \cup \psi(P_{vf})\). As \(X \cup \psi(P_{vf}) = L\), \(X', \delta(P_{vf})) = (r, L, \delta(P_{vf}))\) is a tree rooted at \(r\) and contains all...
Lemma 12. For any state \((v, X, t)\), if \(X \cup \Psi(P_{st}) = L\) and \(v \in P_{st} \cap T(v, X, t)\), \(\mathcal{W}(P_{tr}, \mathcal{W}(P(v)) + \mathcal{W}(T(v, X, t))\) is an upper bound of the weight of the optimal solution.

Proof. Without loss of generality, suppose that there exist two common vertices \(u\) and \(v\) included in both \(P_{st}\) and \(T(v, X, t)\), as shown in Figure 2(a). As shown, \(P_{st}\) corresponds to the path from root to \(v\), denoted by the dashed line. \(T(v, X, t)\) corresponds to the tree rooted at \(v\) and contains vertices \(v, u, y, \) and the other vertices. In addition, the triangle is used to represent the other vertices. Suppose that \(\Psi(P_{st}) \cup X = L\). For simplicity, the labels associated with each vertex are not given in Figure 2.

Let \(G'\) denote the cycle graph in Figure 2(a), which corresponds to the merger of \(P_{st}\) and \(T(v, X, t)\). By removing the edge from \(v\) to \(u\) in \(G'\), we obtained tree \(T'\) rooted at \(v\) as shown in Figure 2(b). Evidently, \(T'\) and \(G'\) contain the same set of vertices. Then, \(T'\) contains all labels in \(L\). Thus, \(T'\) is a feasible solution. As \(T'\) has less number of edges, \(\mathcal{W}(T') \leq \mathcal{W}(G') = \mathcal{W}(P_{st}) + \mathcal{W}(T(v, X, t))\).

3.2.2. Pruning by the Bad States (P2). Given any two generated states, they may have the same root but different weights and time values. The state with a smaller time value and a larger weight value is called a bad state that can be pruned.

Lemma 13. For any two states \((v, x, t_1)\) and \((v, x, t_2)\), if \(t_1 \geq t_2\) and \(\mathcal{W}(T(v, x, t_1)) \leq \mathcal{W}(T(v, x, t_2))\), \((v, x, t_2)\) can be pruned.

Proof. Consider the edge-expansion. First, we examine \(T(v, x, t_2)\) using equation (2). For each edge \(e \in E(v)\), if \(t_2(e) \leq t_2\), we merge both \(e\) and \(T(v, x, t_2)\) into a new tree \(T_1\) as follows: \(T_1 = e \oplus T(v, x, t_2)\).

Next, we examine \(T(v, x, t_1)\) using equation (2) As \(t_2(e) \leq t_2 \leq t_1\), we merge both \(e\) and \(T(v, x, t_1)\) into a new tree \(T_2\) as follows: \(T_2 = e \oplus T(v, x, t_1)\).

Then, \(\mathcal{W}(T_1) = w(e) + \mathcal{W}(T(v, x, t_2))\) and \(\mathcal{W}(T_2) = w(e) + \mathcal{W}(T(v, x, t_1))\). In addition, as \(\mathcal{W}(T(v, x, t_1)) \leq \mathcal{W}(T(v, x, t_2))\), we have \(\mathcal{W}(T_2) \leq \mathcal{W}(T_1)\). Thus, \(T_1\) has a larger weight value. This implies that \(T_2\) does not need to be further expanded. In contrast, if \(t_2(e) > t_2\), \(T(v, x, t_2)\) does not need to be expanded further.

Similarly, for the tree-merger and path-growth operations, \(T(v, x, t_2)\) does not need to be expanded further. Thus, \(T(v, x, t_2)\) can be pruned.

3.3. DP Algorithm. Algorithm 1 describes the DP algorithm. The simplified graph \(G' (V', E')\) and upper bound \(UB\) are obtained as shown in lines 1–6. Queue \(Q\) is used to store the expanded states, which are sorted according to their weights in the ascending order. In the beginning, for each vertex \(v\) in \(V'\) and each label \(l \in L, L\), we construct an initial state \(s_0 = (v, \{l\}, \infty)\). The earliest starting time and cost value for \(s_0\) are set to be zero and zero, respectively. This is because in \(s_0\) comprises only a single vertex \(v\) and no edge. Then, we push \(s_0\) into \(Q\). Queue \(D\) is used to store the states that pop from \(Q\).

Lines 7–25 show that if \(Q\) is nonempty, the DP algorithm repeats the pop/push operation to grow/merge the states individually to obtain the optimal solution. Specifically, the algorithm pops (line 8) the top state \((v, X, t)\) that has the minimum weight among all states in \(Q\). Line 9. Otherwise, \((v, X, t)\) covers all labels in \(L\) and \(v\) is the root vertex \(r\), the algorithm returns it as an optimal solution (line 9). Otherwise, \((v, X, t)\) is pushed into \(D\) (line 10).

Next, the algorithm performs the path-growth operation (lines 11–14). If \(T(v, X, t)\) and \(P_{tr}\) contain all labels in \(L\) and only have one common vertex \(v\), then the expanded state \(s_p\) is generated. As \(s_p\) is a feasible solution according to Lemma 11, it is used to update \(Q\) and \(UB\). Otherwise, if \((v, X, t)\) and \(P_{tr}\) contain multiple common vertices, \(UB\) is updated according to Lemma 12 (lines 15 and 16).

Then, the algorithm attempts to perform the edge-growth operation for \(T(v, X, t)\) (lines 17–20). For each in-edge \(e\) of \(v\), if the arrival time of \(e\) is not larger than \(t\), the path time constraint is satisfied. Then, \(s_e\) is used to update \(Q\) and \(UB\).

Finally, the algorithm performs the tree-merger operation for \((v, X, t)\) and the states in \(D\) (lines 22–25). If there exists a state \(T(v, x', t')\) in \(D\) with the same root vertex \(v\) and \(X' \subseteq L, X\), the expanded state \(s_{me}\) is generated, which is used to update \(Q\) and \(UB\).

We update \(Q\) and \(UB\) by using the update procedure with the expanded state \((v, X, t)\) and its weight cost. In particular, if we merge \((v, X)\) and its state path \(P_{tr}\) into a new tree with a weight value equal to \(\mathcal{W}(P_{tr})\). If the weight value of the tree is larger than the current upper bound, the solution is not an optimal solution. Thus, the expanded state can be pruned and need not be pushed into \(Q\) for further expansion (line 27).

Next, if \(v\) is the root vertex \(r\) and \(X\) contains all labels in \(L\), the expanded state may be a feasible solution. Then, \(UB\) is set as the smaller value between \(UB\) and the weight value cost (line 29). Then, we push the expanded state and its cost value into \(Q\) (line 30). According to Lemma 13, in line 32, if there exists a state \((v, x, t')\) in \(D\) that is better than the expanded state \((v, X, t)\), the expanded state can be pruned. Similarly, if there exists a better state \((v, x, t')\) in \(Q\), the expanded state can also be pruned. Else, if there exists a worse state \((v, x, t')\) in \(Q\), the worse state must be removed from \(Q\). After that, the expanded state and its cost value are pushed into \(Q\) for further expansion (line 33).

The framework diagram of DP is shown in Figure 3.
3.3.1. Algorithm Example 1. Consider Figure 1(a) as an example, where \( L = \{a, b, c, d\} \) and \( T_{a_1, b_1} = [0, \infty] \). From Example 8, we can obtain the simplified temporal graph \( G' \) and UB = 5. Next, Q is initialized with six states, namely \((1, [a], \infty), 0\), \((2, [d], \infty), 0\), \((3, [a], \infty), 0\), \((4, [b], \infty), 0\), \((5, [c], \infty), 0\), and \((7, [d], \infty), 0\) as shown in Figure 4(a).

As Q is nonempty, top state \( s_1 = T(1, [a], \infty) \) is popped from Q. As vertex 1 is not the root, \( s_1 \) and its cost are pushed into D. State path of \( s_1 \) is \( P_{1,\infty} = \langle e_2, e_8, e_2 \rangle \). As \( \psi(P_{1,\infty}) \cup \{a\} \neq L \), the algorithm performs the edge-growth operation. As \( e_2 \in E_1(1) \), \( s_2 = e_2 \oplus s_1 = (0, [a], 1) \) is generated. Then, \( s_2 \) is pushed into Q. Similarly, \( s_3 = e_2 \oplus s_1 = (5, [a], 4) \) is pushed into Q. When examining all initial states, Q contains six new states, as shown in Figure 4(b). Note that D contains the six initial states in Q as shown in Figure 4(a).

Next, state \( s_4 = T(5, [a], 4) \) pops from Q, the state path of which is \( P_{5,4} = \langle e_2, e_8 \rangle \). Then, \( s_{10} = P_{5,4} \oplus s_8 = (0, [a, b, c, d], 1) \) that is pushed into Q. After some state expansions, the algorithm pops \( s_{10} \) from Q and returns it as an optimal solution.

In the following text, we illustrate that the path-growth corresponds to the combination of the edge-growth and tree-merge. Consider state \( s_8 \). According to the tree-merge operation, we obtain state \( s_{12} = e_8 \oplus s_{11} = (2, [a, b, c], 3) \). Then, by using the tree-merge operation, we obtain state \( s_{14} = s_{12} \oplus s_{13} = (2, [a, b, c, d], 3) \), where \( s_{12} = T(2, [d], \infty) \) is in D. Then, by using the edge-growth operation, state \( s_{15} = e_2 \oplus s_{14} = (0, [a, b, c, d], 1) \). Clearly, \( s_{15} \) is the same as \( s_{10} \), generated using the path-growth operation on \( s_8 \). That is, the path-growth operation corresponds to two edge-growth operations and two tree-merge operations.

3.3.2. Algorithm Complexity. Consider the algorithm time complexity in the worst case. The main cost includes the queue operation cost as well as the costs of the edge-growth and tree-merge operations. In addition, consider query label set \( L \), where \( k = \lvert L \rvert \). Then, there exist \( 2^k \) subsets for \( L \). For a state rooted at \( v \in V \), there exist at most \( \lvert E_v(v) \rvert \) different starting times. Furthermore, there exist at most \( \lvert E_v(v) \rvert \cdot 2^k \) states rooted in \( v \in V \), and queue \( Q \) contains at most \( \sum_{v \in V} \lvert E_v(v) \rvert \cdot 2^k = m \cdot 2^k \) states, where \( m = \lvert E \rvert \). Then, according to the Fibonacci Heap, the total cost for the queue operations is calculated as \( O(m \cdot 2^k \cdot \log(m \cdot 2^k)) = O(m \cdot 2^k \cdot (k + \log m)) \). In the update, we must scan all states in Q and D to check whether a state is a bad state. In lines 17–20 (i.e., edge-growth), the total number of possible \( u \in V \) is bounded by \( O(\lvert E \rvert(v)) \), where \( e = (u, v) \). The total number of comparisons in line 18 is bounded by \( O(2^k \cdot \sum_{v \in V} \alpha \cdot \beta) = O(2^k \cdot n \cdot \alpha \cdot \beta) \), where \( \alpha = \max_{v \in V} \lvert E_v(v) \rvert \), \( \beta = \max_{v \in V} \lvert E_v(v) \rvert \) and \( n = \lvert V \rvert \), respectively. In lines 22–25 (i.e., tree-merge), we must scan all states in Q and D, and the total cost is similar to that of the update procedure. Thus, the time complexity is formulated as \( O(m \cdot 2^k \cdot (k + \log m) + 2^{k+1} \cdot m^2 + 2^k \cdot n \cdot \alpha \cdot \beta) \).
Input: $G = (V, E)$, root $r$, time interval $[t_a, t_b]$, and the query label set $L$.
Output: The optimal solution.
(1) obtain $G_s = (V_s, E_s)$ by using the graph simplification on $G(V, E)$;
(2) set the upper bound $UB$ based on $G_s = (V_s, E_s)$;
(3) $Q \leftarrow \emptyset$; $D \leftarrow \emptyset$;
(4) for each $v \in V'$ and each label $l \in L_v$ do
  (5) $s_v \leftarrow (v, [l], \infty)$;
  (6) $Q$.push ($s_v$, 0);
(7) while $Q \neq \emptyset$ do
  (8) $(T(v, X, t), \text{cost}) \leftarrow Q$.pop();
  (9) if $v = r$ and $X = L$ then return $T(v, X, t)$;
  (10) $D \leftarrow D \cup \{(T(v, X, t), \text{cost})\}$;
  (11) if $X \cup \psi(P_{v,t}) = L$ and $\{v\} = P_{v,t} \cap T(v, X, t)$ then
    (12) $s_p \leftarrow P_{v,t} \oplus T(v, X, t)$;
    (13) update ($Q$, $D$, $UB$, $s_p$, $\text{cost} + W(P_{v,t})$);
    (14) continue;
  (15) if $X \cup \psi(P_{v,t}) = L$ and $v \{\} \subset P_{v,t} \cap T(v, X, t)$ then
    (16) $UB \leftarrow \min UB, \text{cost} + W(P_{v,t})$;
    (17) for each $e \in E_i(v)$ do
      (18) if $t_a(e) \leq t$ then
        (19) $s_g \leftarrow e \oplus T(v, X, t)$;
        (20) update ($Q$, $D$, $UB$, $s_g$, $\text{cost} + w(e)$);
    (21) $X \leftarrow L \setminus X$;
    (22) for each $X' \subseteq X$ do
      (23) if $(T(v, X', t'), \text{cost}') \in D$ then
        (24) $s_m \leftarrow T(v, X, t) \oplus T(v, X', t')$;
        (25) update ($Q$, $D$, $UB$, $s_m$, $\text{cost} + \text{cost}'$);
      (26) Procedure update ($Q$, $D$, $UB$, $(v, X, t)$, $\text{cost}$)
      (27) if $\text{cost} + W(P_{v,t}) > UB$ then return;
      (28) if $v = r$ and $X = L$ then
        (29) $UB \leftarrow \min(UB, \text{cost})$;
        (30) $Q$.push ($((v, X, t), \text{cost}$);
        (31) return;
      (32) if $(v, X, t)$ is a bad state then return;
      (33) $Q$.push ($((v, X, t), \text{cost})$);

Algorithm 1: DP.
3.4. Further Enhancement. To design the $A^*$-search algorithm $DP+$ for our problem, we must establish an effective lower bound for each state $(v, X, t)$ in the search space [25, 26]. According to the $A^*$ search theory [25, 26], the lower bound denotes the bound from the current state $(v, X, t)$ to its goal state $(r, L, t')$, where $t' \leq t$. If $(v, X, t)$ is removed from $(r, L, t')$, let $H$ denote the remaining tree, such that $H = (r, L, t') - (v, X, t)$. Evidently, $H$ is rooted at $r$ and covers all labels in $X = L \setminus X$. Then, we must design a lower bound for the weight of tree $H$. In the following, we propose two effective lower bounds by using the state path and relaxing the set of covered labels.

For any state $(v, X, t)$, we define the path-based lower bound as $\xi_1(v, X, t) = \mathcal{W}(P_{v,,})$, where $P_{v,}$ is the state path of $(v, X, t)$. The following lemma shows that $\xi_1(v, X, t)$ is indeed a valid lower bound.

**Lemma 14.** For any state $(v, X, t)$, $\xi_1(v, X, t) \leq \mathcal{W}(H)$. 

**Proof.** As the goal state $(r, L, t')$ denotes the union of $(v, X, t)$ and tree $H$, namely, $H = (r, L, t') - (v, X, t)$, $H$ contains the paths from $r$ to $v$. As state path $P_{v,}$ of $(v, X, t)$ is a path from $r$ to $v$, $\mathcal{W}(P_{v,}) \leq \mathcal{W}(H)$.

For any state $(v, X, t)$, tree $H$ covers all the labels in $X$. Then, we develop the label-based lower bound $\xi_2(v, X, t)$ by relaxing the constraint. Specifically, we consider state $(r, [x, t',])$, which corresponds to the tree rooted at $r$ and covering only one label $x \in X$, where $t' \leq t$. Then, we define the label-based lower bound to be the weight of the maximum weight tree over all $T(r, [x, t'])$ for $x \in X$. That is, $\xi_2(v, X, t) = \max_{x \in X} [\mathcal{W}(T(r, [x, t']))]$. The following lemma shows that $\xi_2(v, X, t)$ is also a valid lower bound. □

**Lemma 15.** For any state $(v, X, t)$, $\xi_2(v, X, t) \leq \mathcal{W}(H)$. 

**Proof.** For any state $(v, X, t)$, $T(r, [x, t'])$ corresponds to the tree with the minimum weight among all trees rooted at $r$ and covering label $x$. Consider path $P$ with the minimum weight among all paths from $r$ to vertex $v$ covering label $x$. In addition, suppose that the starting time of $P$ is $t'$, such that $\delta(P) = t'$. Then, $P$ is contained in $T(r, [x, t'])$.

Next, we prove that $T(r, [x, t'])$ corresponds to the path $P$. Without loss of generality, assume that there exists some edge $e$ that is contained in $T(r, [x, t'])$ but not in $P$. Then, $\mathcal{W}(T(r, [x, t'])) > \mathcal{W}(P) + w(e) > \mathcal{W}(P)$. $P$ is also a tree that is rooted at $r$ and covers the label $x$. Moreover, $\mathcal{W}(P) \leq \mathcal{W}(T(r, [x, t']))$, this contradicts the definition of $T(r, [x, t'])$. Thus, the assumption does not hold.

As $T(r, [x, t'])$ corresponds to the path $P$ from $r$ to $v$ covering the label $x \in X$, the tree $H$ contains all paths from $r$ to the vertices that cover the labels in $X$. Then, $\mathcal{W}(P) \leq \mathcal{W}(H)$.

According to Lemmas 14 and 15, we have the following lemma. □

**Lemma 16.** For any state $(v, X, t)$, $\xi(v, X, t) = \max[\xi_1(v, X, t), \xi_2(v, X, t)] \leq \mathcal{W}(H)$.

3.5. $DP+$ Algorithm. The $DP+$ algorithm description is in Algorithm 1, which is similar to that of the $DP$ algorithm. The framework diagram of $DP+$ and $DP$ is similar.

The main differences between the $DP$ and $DP+$ algorithms are as follows. First, the $DP+$ algorithm adopts the sum of the weight and the lower bound as the priority, whereas the $DP$ algorithm only uses the weight as the priority. Each state is associated with an additional priority value, $lb$. Note that the states in $Q$ are sorted according to their priority values in the ascending order.

Second, in the update procedure, the $DP+$ algorithm must call the priority function (line 12) to compute the lower bound and the priority of the expanded state and then determine whether the state can be pruned or not (line 14). Note that for a state, the priority value is often not less than the cost value. Thus, the pruning becomes tighter compared with the $DP$ algorithm.

Note that in line 13, for each expanded state, the algorithm must take the maximum over priorities $lb$ and $\bar{lb}$ of its parent state to ensure consistency of the lower bound. Then, the correctness of the algorithm can be guaranteed, which is analyzed as follows.

3.5.1. Algorithm Analysis. According to the $A^*$ search theory, the developed lower bounds must satisfy the consistent property for the state-expansion operations. As the path-growth operation implies the combination of edge-growth and tree-merger operations, we only need to consider the consistent property for each operation.

For a state $(v, X, t)$, we first prove that the path-based lower bound $\xi_1(v, X, t)$ is consistent, as shown in the following Lemma 17. In particular, for the state $s = T(v, X, t)$, we can obtain a successor state $s_g = (s(e), X, t_e(e))$ by the edge-growth operation, where $e \in E(v)$ and $t_e(e) \leq t$. Similarly, by using the tree-merger operation to merge $s = T(v, X, t)$ with $s' = T(v, X, t')$, we can obtain a successor state $s_m = (v, X \cup X', \min[t, t'])$.

**Lemma 17.** For any state $(v, X, t)$, we have (1) $\xi_1(s_g) + w(e) \geq \xi_1(s)$ and (2) $\xi_1(s_m) + \mathcal{W}(s') \geq \xi_1(s)$.

**Proof.** Let $P_{v,g}$, $P_g$, and $P_m$ denote the state paths of states $s$, $s_g$, and $s_m$, respectively.

(1) Since $s_g = (s(e), X, t_e(e))$, $P_{v,g}$ corresponds to the path from root $r$ to vertex $s$. By combining $P_{v,g}$ and edge $e$, we can obtain path $P$ from $r$ to $v$, and the arrival time of $P$ is $t_e(e) \leq t$. Thus, $P \in P(v, t)$ and $\mathcal{W}(P) = \mathcal{W}(P_{v,g}) + w(e)$. As $P_{v,g}$ has the minimum weight value among all paths in $P(v, t)$, $\mathcal{W}(P) \geq \mathcal{W}(P_{v,g})$. In addition, as $\xi_1(s) = \mathcal{W}(P_{v,g})$ and $\xi_1(s_g) = \mathcal{W}(P_g)$, we have $\xi_1(s_g) + w(e) \geq \xi_1(s)$. Therefore, this case holds.

(2) As $s_m = (v, X \cup X', \min[t, t'])$ and $s = T(v, X, t)$, we have $\mathcal{W}(P_m) \geq \mathcal{W}(P_{v,g})$ according to Definition 9. Further, as $\xi_1(s_m) = \mathcal{W}(P_m)$ and $\xi_1(s) = \mathcal{W}(P_{v,g})$, we have $\xi_1(s_m) \geq \xi_1(s)$. Thus, $\xi_1(s_m) + \mathcal{W}(s') \geq \xi_1(s)$.
However, the label-based lower bound is not consistent. Therefore, the following process is used to ensure consistency of the lower bound, as done in [27].

Specifically, for the successor state $s_g$ of $s = T(v, X, t)$ that is obtained by expanding an edge $e \in \tilde{E}(v)$, we set the new label-based lower bound $\xi_s^L(s_g) = \max\{\xi_{s_g}(s_{g}), \xi_s^L(s) - \omega(e)\)}.

Similarly, for the successor state $s_m$ that is obtained by merging with $s = T(v, X, t)$ and $s' = T(v', X', t')$, we set the new label-based lower bound $\xi_s^L(s_m) = \max\{\xi_s^L(s_m), \xi_s^L(s) - \omega(s')\}$. Next, we prove that the new label-based lower bound is consistent.

**Lemma 18.** For any state $(v, X, t)$, we have (1) $\xi_s^L(s_g) + \omega(e) \geq \xi_s^L(s)$ and (2) $\xi_s^L(s_m) + \omega(s') \geq \xi_s^L(s)$.

**Proof.** (1) If $\xi_s^L(s_g) = \xi_s^L(s_{g})$, we can obtain $\xi_s^L(s_g) \geq \xi_s^L(s) - \omega(e)$. Then, $\xi_s^L(s_g) + \omega(e) \geq \xi_s^L(s)$. If $\xi_s^L(s_g) = \xi_s^L(s) - \omega(e) + \xi_s^L(s_g) = \xi_s^L(s)$. Thus, $\xi_s^L(s_g) + \omega(e) \geq \xi_s^L(s)$. (2) Similar to (1), we have $\xi_s^L(s_m) + \omega(s') \geq \xi_s^L(s)$.

The consistent property also implies that the new label-based lower bound is a valid lower bound [28]. Based on the abovementioned lemmas, the consistent property can be preserved by taking the maximum operation over the devised consistent lower bounds (i.e., line 12 in Algorithm 1). Thus, the DP+ algorithm can find the optimal solution according to the consistency-guarantee property of the $A^*$ search.

### 3.5.2 Algorithm Example 2

Consider Figure 1(a). Similar to Algorithm example 1, in this example, queue $Q$ was initialized with six states, as shown in Figure 5(a). The difference is that here, we must compute the priority values for the initial states. For example, for the state $(1, [a], \infty)$, $\xi_1(1, [a], \infty) = \omega(P_{1, a}) = 3$ and $\xi_2(1, [b], \infty) = \omega(P_2) = 3$, where $P_2 = \langle e_2, e_9, e_{13} \rangle$. Then, $lb_o = \max\{lb_1, lb_2\} = 3$.

Similarly, we have the expanded states $s_2 = e_1 \oplus s_1 = (0, [a], 1)$, $s_5 = e_9 \oplus s_4 = (2, [a], 3)$, and $s_6 = e_9 \oplus s_4 = (2, [a], 2)$, where $s_5 = T(1, [a], \infty)$ and $s_6 = T(5, [a], 4)$. As the priority values of these states are larger than that of upper bound $UB$, the states are pruned in the DP+ algorithm, as shown in Figures 5(b) and 5(c). Note that $s_2$ and $s_5$ are not pruned in the $DP$ algorithm. In addition, state $(7, [d], \infty)$ does not need to be expanded, as shown in Figure 5. This is because their priority values are the largest among the states in $Q$.  

### 4. Extension

#### 4.1. Unspecified Tree Root

Our extension problem can be formulated as follows. Given a temporal graph $G = (V, E)$, a set of query label $L \subseteq \mathcal{L}$, and a time constraint $[t_a, t_f]$. The objective of our extension problem is to find a minimum weight tree $T$ from $G$ that is satisfied with $[t_a, t_f]$ and contains all labels in $L$.

By the modification of the $DP$ algorithm, we present the algorithm called $DP-UR$ for our extension problem. The modification is given as follows.

(i) As the tree root is unspecified, we cannot obtain the earliest arrival time for each vertex. Thus, Step 2 must be removed from the graph simplification. In addition, the path-growth operation cannot be used. This is because for any state $(v, X, t)$, we must obtain the path from a given tree root to vertex $v$. Thus, in the $DP-UR$ algorithm, we expand the states by using the edge-growth and tree-merger operations.
(ii) Our extension problem does not require that the optimal solution contains a specific root. Thus, we randomly selected a vertex \( v \) in \( V \) as the tree root to set the upper bound. Let UB be the sum of the weights of the distinct edges contained in the minimum weight paths from \( v \) to each vertex covering the labels in \( L \). As the optimal solution has the minimum weight among all trees covering all labels in \( L \), UB is not smaller than the weight of the optimal solution. Thus, we can set UB as the upper bound that can be obtained in a similar manner as the DP algorithm.

(iii) Unlike the DP algorithm, if there exists a minimum weight tree that is satisfied with the time constraint and contains all labels in \( L \), DP-UR halts operation and returns this tree as the optimal solution.

Therefore, we can say that the framework of DP-UR is similar to that of DP except for the aforementioned modification.

4.2. Progressive Search Algorithm. The progressive search algorithm works in rounds, reporting a suboptimal and feasible solution with smaller error guarantees in each round, until the last round outputs the optimal solution. In particular, we propose a new approach to construct a feasible solution for each state in each round and then produce progressively refined feasible solutions during algorithm execution.

Consider the DP+ algorithm. For each state \((v, X, t)\) that is popped from \( Q \), we can construct a feasible solution, \( T(r, L, \tilde{T}) \), as follows. (1) \( T(r, \tilde{T}) \) is obtained by performing the tree-merger operation on \( T(v, X, t) \) and its state path \( P_{v,t} \), where \( \tilde{T} = X \cup \psi(P_{v,t}) \). (2) According to the label-based lower bound, we easily obtain such a state \( T(r, [x], t') \) that covers only one label \( x \in X' (= L \setminus X) \). We repeatedly combine with \( T(r, [x], t') \) until each label in \( X' \) is included and then obtain a feasible solution \( T(r, L, \tilde{T}) \). Thus, for any state \((v, X, t)\), we can always report a feasible solution \( T(r, L, \tilde{T}) \) and its approximation ratio, namely \( \mathcal{W} (T(r, L, \tilde{T})) / \mathcal{W} (T(v, X, t)) \).

In particular, we can obtain the progressive search algorithm by inserting the following description between lines 9 and 10 of the DP+ algorithm.

\[
\begin{align*}
T(r, \tilde{T}) & \leftarrow P_{v,t} \oplus T(v, X, t); \\
\tilde{X'} & \leftarrow L \setminus \tilde{T}; \\
s & \leftarrow T(r, \tilde{X}); \\
\text{for all } x \in \tilde{X'} & \text{ do} \\
& s' \leftarrow s \oplus T(r, [x], t'); \\
& s \leftarrow s'; \\
\text{UB} & \leftarrow \min(UB, \mathcal{W}(s)); \\
\text{AR} & \leftarrow \mathcal{W}(s) / \mathcal{W}(T(v, X, t)); \\
\text{report the approximation ratio } \text{AR};
\end{align*}
\]

4.3. Top-N Search Algorithm. The top-N search algorithm aims to find the optimal GST ranked according to the weight of each tree, that is, the weight sum of the edges in each tree. In particular, we can obtain the top-N search algorithm by simply replacing line 8 of the DP+ algorithm with

\[
\text{if } v = r \text{ and } X = L \text{ then output } T(v, X, t); \\
i \leftarrow i + 1; \\
\text{terminate if } i = N;
\]

Here, \( i \) is initialized as 0. The algorithm reports approximate answers for the top-N search, where \( N > 1 \); however, the first answer, \( T_1 \), is guaranteed to be optimal. Moreover, owing to the fact that the smallest cost tree is always placed at the top of the priority queue \( Q \), we can find \( T_1, T_2, \ldots, T_N \) in the increasing order of cost, i.e., \( \mathcal{W}(T_1) \leq \mathcal{W}(T_2) \leq \ldots \leq \mathcal{W}(T_N) \). Therefore, sorting is not required. In practice, the value of UB may be smaller than...
that of $W(T_i)$, and $T_i$ may be pruned. For simplicity, we removed the upper bound UB from the search algorithm.

5. Experiment

This section presents the performance evaluation of our proposed algorithms, including the algorithm running time and memory consumption. In addition, we show the effectiveness of the algorithms according to a case study. We executed all experiments on a machine with a 3.6-GHz Intel Core i7-9700K CPU and 32 GB RAM running Ubuntu 18.04 LTS Linux OS. All algorithms were implemented in C++. We compared our proposed algorithms with the baseline algorithm that is based on the algorithm in [18].

5.1. Experimental Settings

5.1.1. Datasets. We use five real-world datasets, namely Austin, Houston, Los Angeles (LA) (https://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds), IMDB (https://www.imdb.com/interfaces/), and DBLP (https://www.cs.cmu.edu/~enron/) datasets. Austin, Houston, and LA datasets record the timetable of the public transportation networks of a major city on a weekday, which are also widely used in the path querying in temporal networks [2]. Each vertex corresponds to the station, and each edge from a vertex to another represents a bus travel from one station to another. The starting time and arrival time of the edge are the starting time and the arrival time of the route. We use the OpenStreetMap (https://www.openstreetmap.org/) to extract the POIs within 500 meters of each station. The type of each POI is used as the label associated with the station. For the IMDB temporal graph, each vertex corresponds to a person such as the principal cast or director, a title, and the edges denote the different relationships among them. Their names are used as the labels. The starting time and arrival time of the edges are set as the release year of a title or TV Series end year. We assign weights to the edges based on the weight cascade model which is similar to [3]. The details for the corresponding temporal graphs for algorithm performance are given in Table 1.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Austin</th>
<th>Houston</th>
<th>Los Angeles</th>
<th>IMDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>V</td>
<td>$</td>
<td>2.7K</td>
<td>9.1K</td>
</tr>
<tr>
<td>$</td>
<td>E</td>
<td>$</td>
<td>535.5K</td>
<td>1796.2K</td>
</tr>
<tr>
<td>Average degree</td>
<td>119.6</td>
<td>197.4</td>
<td>142.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>


5.1.2. Compared Approaches. To evaluate the effectiveness, we compare the baseline algorithm BL with our two algorithms, DP and DP+, where DP is Algorithm 1 in Section 3.3, and DP+ is Algorithm 2 in Section 3.5. There are four important pruning techniques in our algorithms. To evaluate the effectiveness of the pruning techniques, we compare UB, $P_1$, $P_2$, and $A^*$. UB means the case that we remove the state pruning $P_1$ and $P_2$ from the DP algorithm, $P_1$ means the case that we remove $P_2$ from the DP algorithm (i.e., UB + $P_1$), $P_2$ corresponds to the case UB + $P_1$ + $P_2$, and $A^*$ corresponds to the case UB + $P_1$ + $P_2$ + $A^*$. We also use algorithm BL-UR and algorithm DP-UR to evaluate the effectiveness of the extension problem which does not contain a specified tree root.

1) Baseline Algorithm. is the state-of-the-art algorithm for our problem, which is set as follows.

The basic idea of the baseline algorithm call BL is to transfer the given timetable graph into a directed weighted (static) graph and then execute the existing algorithm for the GST search on the static graph to find the optimal tree rooted at the specified vertex $r$. Specifically, BL functions as follows. First, timetable graph $G = (V, E)$ is transformed into a directed static graph $G(V, \tilde{E})$ by using the graph transformation method proposed in [3], which requires only $O(|E|)$ amount of time. In the graph transformation, the number of virtual vertices and virtual edges to be created for each nonroot vertex $v \in V$ is based on the number of distinct arrival times of each in-edge for $v$. In particular, root $r$ corresponds to one virtual vertex. Next, to solve our problem, we considered the algorithm proposed in [18], which is the best-known exact algorithm for keyword search in a relational database [17]. We processed it on $G$ to find the tree rooted at the virtual vertex of $r$, which has the minimum weight and covers all labels in $L$. After transforming, the number of vertex and edges in a directed static graph $G(V, \tilde{E})$ changes, $|V| = m + n, |E| = 2m$. Note that the algorithm in [18] possesses exponential time complexity. When the graph or query label set is relatively large, the baseline algorithm becomes impractical. Although the algorithm in [20] is more efficient, it is targeted at undirected graphs and thus cannot be applied to our problem, which comprises a directed graph. The algorithm complexity of baseline is $O(3^k \cdot n + 2^k \cdot ((m + n) \cdot \log(n + 3m)))$ time, where $k$ is the number of specified labels, and $m$ and $n$ are the number of edges and vertices of the temporal graph, respectively.

5.1.3. Comparative Experiment Setting. We vary two parameters in our experiments, namely $kn$ and $l$, where $kn$ is the number of labels in the given query label $L$ (i.e., $kn = |L|$) and $l$ is the average number of vertices covering each label in the query (i.e., the label frequency). The value of $kn$ is selected from 4, 5, 6, 7, and 8 with a default value of 4, and the value of $l$ is selected from 100, 200, 400, 600, and 800 with a default value of 400. Unless otherwise specified, when varying a parameter, the values of the other parameters are set to their default values. The time interval setting of all experiments corresponds to $[0, \infty)$. In each test, we generate 30 queries and each query contains $kn$ labels, which are randomly selected from the given set of labels, and the average results over all of them are reported. We set the query time-out value as 5 hours, i.e., 18,000 (18K) seconds. When a query time-out occurs, the query algorithm will be halted. For the root vertex in each temporal graph, we require that it can reach all other vertices of the selected
connected components. We simply scan the vertices until one such vertex is found. The details for the connected components are given in Table 1. In particular, the Enron dataset is used for the case study since the size of its connected components is relatively small.

5.2. Experimental Results and Analyses

5.2.1. Effect of \( \text{kn} \) on Running Time. As shown in Figure 6, the running time is increased with the increase of \( \text{kn} \). This is because the increase of \( \text{kn} \) will result in the increase of algorithm search space. The running time of our algorithm \( \text{DP} \) (\( \text{DP}^+ \)) is obviously smaller than that of \( \text{BL} \). In particular, as shown in Figure 6(a), for \( \text{kn} = 8 \), the baseline algorithm \( \text{BL} \) algorithm requires 3,707 seconds, but our proposed algorithms \( \text{DP} \) and \( \text{DP}^+ \) only take 0.42 and 0.24 seconds, respectively. Our algorithm \( \text{DP}^+ \) is more than 15,446 times faster than \( \text{BL} \). This is because the proposed graph simplification can result in a smaller graph for our algorithm. Meanwhile, more states can be pruned by the proposed optimization techniques. Besides, the lower bounds based on \( A^* \) strategy can further reduce the number of states to be generated, and the running time of \( \text{DP}^+ \) is obviously smaller than \( \text{DP} \). As shown in Figure 6(b), if \( \text{kn} = 5 \), the baseline algorithm \( \text{BL} \) takes more than 3,636 seconds, and both \( \text{DP} \) and \( \text{DP}^+ \) take 0.43 and 0.3 seconds, respectively. When \( \text{kn} = 6, 7, 8 \), the baseline algorithm \( \text{BL} \) is time-out and cannot be finished in the specified 5 hours, i.e., 18K seconds. However, for the case of \( \text{kn} = 8 \), the \( \text{DP} \) and \( \text{DP}^+ \) only require 2.8 and 1.43 seconds, respectively. As shown in Figure 6(c), if \( \text{kn} = 6, 7, 8 \), the baseline algorithm \( \text{BL} \) is time-out. Both \( \text{DP} \) and \( \text{DP}^+ \) take less than 10 seconds. As shown in Figure 6(d), if \( \text{kn} = 5, 6, 7, 8 \), the baseline algorithm \( \text{BL} \) is time-out. For \( \text{kn} = 8 \), \( \text{DP} \) and \( \text{DP}^+ \) take about 771.8 seconds and 134.2 seconds, respectively.

As we discussed above, \( \text{kn} \) has an important impact on the algorithm time complexity of \( \text{BL} \), \( \text{DP} \), and \( \text{DP}^+ \). As shown in Figure 6, the time of the algorithm increases rapidly as \( \text{kn} \) increases. For \( \text{BL} \), it transformed the temporal graph into a directed static graph first. The number of vertex and edge in the directed static graph is much larger than that in the temporal graph. Therefore, the time increase of \( \text{BL} \) is significantly faster than that of \( \text{DP} \) and \( \text{DP}^+ \). However, \( \text{BL} \) does not contain any graph simplification and state pruning techniques. So, \( \text{BL} \) is much slower than \( \text{DP} \) and \( \text{DP}^+ \). For \( \text{DP} \), it adopts the proposed graph simplification and state pruning techniques. Therefore, \( \text{DP} \) is much faster than \( \text{BL} \). For \( \text{DP}^+ \), it uses \( A^* \) search strategy to further up \( \text{DP} \). So, \( \text{DP}^+ \) can find the optimal solution faster.

5.2.2. Effect of \( \text{lf} \) on Running Time. As shown in Figure 7, the running time is decreased with the increase of \( \text{lf} \). With the increase of \( \text{lf} \), there are more vertices containing the query labels. Then, it is faster for the algorithm to achieve the optimal solution. In general, the running time of \( \text{DP} \) (\( \text{DP}^+ \)) is obviously smaller than \( \text{BL} \). This is because we can obtain a simplified graph by the graph simplification and at the same time the proposed optimization techniques can reduce the number of states in the algorithm. In particular, for \( \text{lf} = 100 \), in Figure 7(d), \( \text{DP}^+ \) takes 0.66 seconds, and \( \text{BL} \) needs more than 3,116 seconds. \( \text{DP}^+ \) is 4,721 times more faster than \( \text{BL} \). As shown in other figures, \( \text{DP} \) and \( \text{DP}^+ \) can obtain the optimal solution in 1 second in most cases. Since the \( A^* \) search strategy can further reduce the number of states, the running time of \( \text{DP}^+ \) is smaller than that of \( \text{DP} \).

In order to get the optimal solution, each state needs to do the edge-growth operation and tree-merger operation. As \( \text{lf} \) grows, so does the number of initialized states. Then, more states can be merged. However, it is faster for the algorithm to achieve the optimal solution, as shown in Figure 7. For \( \text{BL} \), with the increase of \( \text{lf} \), the time of \( \text{BL} \) increases. Figure 7 illustrates how the time of \( \text{BL} \) increases as \( \text{lf} \) increases. For \( \text{DP} \) and \( \text{DP}^+ \), they adopt the proposed graph simplification and state pruning techniques. Therefore, \( \text{DP} \) and \( \text{DP}^+ \) are faster than \( \text{BL} \). For \( \text{DP}^+ \), it has more pruning techniques than \( \text{DP} \). Therefore, \( \text{DP}^+ \) is faster than \( \text{DP} \) and \( \text{BL} \).

5.2.3. Effect of Pruning Techniques on Running Time. There are four main pruning techniques in the \( \text{DP} \) and \( \text{DP}^+ \) algorithms, namely the upper bound \( \text{UB} \), the state pruning methods \( P_1 \) and \( P_2 \), and the \( A^* \) search. In Figure 8, \( \text{UB} \) means the case that we remove the state pruning \( P_1 \) and \( P_2 \) from the \( \text{DP} \) algorithm. \( P_1 \) means the case that we remove \( P_2 \) from the \( \text{DP} \) algorithm (i.e., \( \text{UB} + P_1 \)). \( P_2 \) corresponds to the case \( \text{UB} + P_1 + P_2 \), and \( A^* \) corresponds to the case \( \text{UB} + P_1 + P_2 + A^* \). The running time can be dramatically reduced since lots of states can be pruned by the proposed techniques. As shown in Figure 8(d), the running time is relatively smaller on IMDB compared with other datasets. This is because the upper bound is more efficient since the vertices containing the query labels are nearer to the root vertex. Besides, the height of the optimal solution found in IMDB is smaller than in other datasets.

5.2.4. Effect of Graph Simplification. As shown in Figure 9, the gap between the given temporal graphs and their simplified graphs is obvious. This is because lots of edges and vertices can be removed by the graph filter. For Austin, after graph simplification, the vertices are reduced by 4.2% and the edges are reduced by 55.5%. For Houston and Los Angeles, the vertices are reduced by 7.8% and 60%, and the edges are reduced by 73.1% and 27.6%, respectively. Since the connected components of Austin and Houston are more denser, more edges and fewer vertices can be removed, compared with LA and IMDB.

5.2.5. Effect of \( \text{kn} \) and \( \text{lf} \) on Memory Consumption. We conducted the experiments on the connected components of Austin, and the results are shown in Figure 10. In general, the memory consumption is increased with the increase of \( \text{kn} \). This is because there are more states that need to be stored as the \( \text{kn} \) becomes larger. The maximization consumption is about 48.3 MB (47.7 MB) for \( \text{DP} \) (\( \text{DP}^+ \)) as \( \text{kn} = 8 \). The memory consumption for \( \text{BL} \) is often larger than
Figure 6: Effect of $kn$ on running time. (a) Austin. (b) Houston. (c) LA. (d) IMDB.

Figure 7: Effect of $lf$ on running time. (a) Austin. (b) Houston. (c) LA. (d) IMDB.

Figure 8: Effect of pruning methods on running time. (a) Austin. (b) Houston. (c) LA. (d) IMDB.

Figure 9: Effect of graph simplification. (a) Number of vertices. (b) Number of edges.
1,500 MB. In general, our algorithm memory is at least 30 times smaller than BL. On the other hand, the memory consumption is decreased with the increase of If. This is because, with the increase of If, there are fewer states that need to be stored in the algorithms. Since the number of states in our algorithm is smaller, the memory consumption of $DP (DP^+)$ is at least 14 times smaller than BL.

As we discussed above, kn has an important impact on the algorithm memory consumption of BL, DP, and DP+. As shown in Figure 10, the memory consumption of the algorithm increases rapidly as kn increases. For BL, it has the highest number of states for finding the optimal solution. Therefore, the memory consumption of BL is significantly larger than that of DP and DP+. For DP and DP+, they adopt the proposed graph simplification and state pruning techniques. Therefore, DP and DP+ need less memory consumption than BL. For DP+, it uses $A^*$ search strategy to reduce the number of states. So, DP+ needs less memory consumption than DP+.

As If grows, so does the number of initialized states. Then, two states merge into a new state will be earlier. However, it is faster for the algorithm to achieve the optimal solution. Therefore, DP and DP+ need less memory consumption. For DP+, it has more pruning techniques than DP. Therefore, DP+ needs less memory consumption.

(1) Running Time for Unspecified Root. Since the problem extension does not contain a specified tree root, in the baseline algorithm $BL-UR$, we adopt the graph transformation method in [4] to transfer the temporal graph into the directed weighted graph. The results on the Austin datasets are shown in Figure 11. In general, the running time of $DP-UR$ is smaller than that of $BL-UR$. This is because the graph simplification can reduce the sizes of the temporal graph and the proposed pruning techniques can reduce the number of states in the algorithm. Similar to the reasons mentioned above, the algorithm running time is increased with the increase of $kn$ and is decreased with the increase of If.

(2) Memory Consumption for Unspecified Root. The results on the Austin datasets are shown in Figure 12. In general, the memory consumption of $DP-UR$ is smaller than that of $BL-UR$ since there are fewer states in $DP-UR$. Similar to the reasons mentioned above, the algorithm memory is increased with the increase of $kn$ and is decreased with the increase of If.

(3) Effect of $N$ on Running Time. We test top-$N$ search algorithm to find the optimal $N$ solutions of all datasets. We vary $N$ from 1, 10, . . . to 50 while fixing $k$ and $f$ at the default values of DP+ and report our results in Figure 13. From Figure 13, as $N$ increases, the running time of DP+ increases.

(4) Progressive Performance Testing. In this experiment, we test how well the reported feasible solutions are progressively improved during algorithm execution. We randomly select a query with $k = 4, f = 400, N = 1$ on Austin and report the approximation ratio $AR$ in Figure 14. For each algorithm, we can see that the approximation ratio $AR$ decreases with increasing running time. As can be seen, the UB algorithm results in a 4-approximation solution within 14.8 seconds and obtains the optimal solution taking around 23.6 seconds. The $A^*$ algorithm results in a 3-approximation solution within 0.14 seconds and obtains the optimal solution taking around 0.17 seconds. These results demonstrate that the $A^*$ algorithm exhibits excellent progressive performance in practice, which further confirms our theoretical findings.

5.2.6. Case Study. The case study is on the DBLP datasets. Each vertex corresponds to an author, or a paper, and the edges denote the different relationships among them such as the coauthor relationship. Their names are used as the labels. The starting time and arrival time of the edges are set as the paper’s publication time. We assign weights to the edges based on the weight cascade model which is similar to [3]. In detail, the search labels correspond to the author names, namely Jennifer Widom, Jiawei Han, Jian Pei, and Philip S. Yu. The time interval setting is (1990, 2021). The connected tree is given in Figure 15, which shows the relationship between the specified authors and their papers during the specified time. By the found connected tree, we can know the most influenced paper “Clustering Association Rules” written by Jennifer Widom in 1997 is related to the other authors. This study is cited by the paper “Mining Frequent Patterns without Candidate Generation” by Jiawei Han and Jian Pei in 2000 and the paper “Mining Large Itemsets for Association Rules” by Philip S. Yu in 1998. Both of them are also the most influenced papers of the related authors written in the given time.

6. Related Work

The GST problem has been found useful in different applications and has attracted much research interest in recent years. The studies most closely related to our problem include the computation of a minimum spanning tree (MST) in a temporal graph [3, 9, 29], progressive GST searching in a static graph [20, 30–32], keyword searching in relational databases [18, 33], and reconstruction of an epidemic over time [34–41].

In this study, we considered searching for a GST in a temporal graph; this concept differs from that of the MST search [3, 9, 29]. In addition, our solution is a type of exact algorithm, unlike the approximation algorithm [3], and it differs from the exact solution discussed in [9], which is based on the integer programming algorithm. In [18], the problem of keyword search in a relational database was formulated as a GST problem in a directed graph. In [29], a time-varying neural network for solving the time-varying minimum spanning tree problem with constraints is proposed, which is different. In addition, in [17], the authors showed that the algorithm in [18] is the best-known exact algorithm for the keyword-search problem. However, the algorithm is impractical for a relatively large graph or label set. To overcome this shortcoming, some optimization techniques were proposed in the progressive algorithm [20]
Figure 10: Effect on memory consumption. (a) Effect of kn. (b) Effect of If.

Figure 11: Running time for unspecified root. (a) Effect of kn. (b) Effect of If.

Figure 12: Memory consumption for unspecified root. (a) Effect of kn. (b) Effect of If.
to improve the search of GST in a static graph. However, these optimization techniques are targeted at an undirected graph and are therefore not suitable for our problem. In [30], the GST problem with node and edge weights was considered and an approximate algorithm is proposed using a DP approach. In [33], a query relaxation algorithm was
proposed that solves the small or even empty result sets when performing query operations in temporal graph databases. However, the efficient management of the time information is key for our solution, which has not been studied in previous algorithms \cite{18, 20, 30, 33, 42, 43}. Moreover, these studies have not considered the case of a GST with a specified tree root.

The path querying with various settings in temporal networks was studied in \cite{2, 4, 8, 11, 12, 44–47}. Unlike these studies, our problem focuses on the search for GST in temporal graphs. As discussed earlier, our problem can also be used in the path route applications. In addition, our problem seeks to find the connected minimum-weighted tree in temporal graphs, which is unlike the subgraph mining in temporal networks \cite{7, 10, 13, 48–50}. In general, our algorithm considers the search based on labels/keywords in a large graph, a method that is similar to the keyword search in large graphs \cite{51} and the keyword-aware route search \cite{52}. However, the graphs in \cite{51, 52} do not contain temporal information.

The rumors or infection spread application in social network was studied in \cite{5, 6, 53–55}. The authors of \cite{5, 6} are based on the Steiner tree in a temporal graph. The problem in \cite{5} was to determine multiple Steiner trees in which the roots are not unspecified and each vertex is activated. In \cite{6}, the authors focused on searching for a single Steiner tree in which some vertices may be not activated. Our proposed problem can also be used in the application of rumors or infection spreading. However, we considered the labels associated with the vertices and focused on the GST search with a specified tree root in a temporal graph. Moreover, the solution in \cite{5, 6} is a type of approximation algorithm, which differs from our exact solution. The authors in \cite{53–55} did not consider the effect of time in social networks on the problem of rumor propagation, which differs from ours.

The efficient web APIs recommendation was studied in \cite{34–41}. These works model the web APIs recommendation problem as a group Steiner tree search problem. The authors in \cite{34} build an APIs correlation graph and explore a data-driven APIs recommendation approach named WAR to assist developers in finding compatible APIs. Qi et al. \cite{35} improved the model proposed in \cite{34} by introducing a weighting mechanism that improves recommendation accuracy. The authors in \cite{36} incorporate a multiagent technique with MGST to produce potentially compatible APIs compositions. The authors in \cite{37} first model the compatibility-aware web API composition allocation problem into a minimal group Steiner tree search problem and then use the determinantal point processes technique to diversify the recommended several web API compositions. The authors in \cite{38} devise an efficient web APIs recommendation approach with privacy preservation by incorporating the LSH technique with the MGST searching algorithm. The authors in \cite{39} propose a keywords-driven web API group recommendation technology for sustainable software creation, which can output multiple desirable groups of web API lists instead of only one. The authors in \cite{40} propose a collaborative filtering API recommendation model. The core idea of it is to achieve collaborative filtering by mining binary-API topics as an embedding layer between mashups and APIs.

The authors in \cite{41} first construct a web APIs correlation graph and then propose a correlation graph-based approach for personalized and compatible web APIs recommendation in mobile APP development. However, these studies have not considered the time information in the temporal graph which is important for our problem.

Although several related studies have been conducted in this aspect, such as the incremental computation of dynamic graphs \cite{56}, processing of growing temporal graphs \cite{57}, temporal data management \cite{58}, trace spatial-temporal graph evolution \cite{59}, maximal D-truss search in dynamic directed graphs \cite{60}, and querying connected components in temporal graphs \cite{61, 62}, these studies did not consider the TGST search problem.

7. Conclusion

In this paper, we propose an efficient dynamic programming algorithm called \textit{DP} for the computation of GST in temporal graphs, i.e., TGST. We adopted some new optimization techniques, including graph simplification and state pruning, to reduce the algorithm search space by a significant amount. Moreover, we designed the \textit{A}$^*$-search algorithm called \textit{DP}$^*$ that can further speed up the algorithm search. We also propose an algorithm for our extension problem, namely the TGST with unspecified root, the progressive search of TGST, and the top-$N$ search of TGST. We conducted a series of experiments on real temporal networks, and the results verified the efficiency and effectiveness of our algorithms. In the future, we plan to consider further optimization techniques for our problem with different settings such as a distributed application environment. As we can see, the \textit{A}$^*$ strategy has a very obvious effect on the \textit{DP}$^*$ algorithm, and in future work, we can continue to search for other \textit{A}$^*$ strategy. In the future, this work can be further combined with practical problems to propose specific algorithm-pruning strategies for specific practical situations.

Data Availability

The Austin, Houston, Los Angeles (LA), IMDB and DBLP data used to support the findings of this study are included within the article.

Ethical Approval

This study did not involve any human participants or animals. The results were obtained through simulation and tested several times to achieve a final value.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Authors’ Contributions

Youming Ge designed the algorithm, wrote the main manuscript, and prepared all the figures. Youming Ge, Weiang Kong, and Sen Zhang mainly wrote the algorithm code and tested the experiment in this manuscript. Zitong
Chen, Yubao Liu, and Raymond Chi-Wing Wong provided guidance on the algorithm design and proposed revisions to the writing of the manuscript. All authors reviewed the manuscript. Yubao Liu contributed equally to the study.

Acknowledgments

This study was supported by the National Nature Science Foundation of China (61572537 and U1501252).

References


