Data Science and Engineering (2023) 8:368–384
https://doi.org/10.1007/s41019-023-00218-7

RESEARCH PAPER

An Efficient Keywords Search in Temporal Social Networks

Youming Ge[1] · **Zitong Chen**[1] · **Yubao Liu**[1]

Received: 18 March 2023 / Revised: 21 July 2023 / Accepted: 22 July 2023 / Published online: 9 September 2023

Abstract
With the increasing of requirements from many aspects, various queries and analyses arise focusing on social network. Time is a common and necessary dimension in various types of social networks. Social networks with time information are called temporal social networks, in which time information can be the time when a user sends message to another user. Keywords search in temporal social networks consists of finding relationships between a group users that has a set of query labels and is valid within the query time interval. It provides assistance in social network analysis, classification of social network users, community detection, etc. However, the existing methods have limitations in solving temporal social network keyword search problems. We propose a basic algorithm, the discrete timestamp algorithm, with the intention of turning the problem into a traditional keyword search on social networks. We also propose an approximative algorithm based on the discrete timestamp algorithm, but it still suffers from the traditional algorithms' low efficiency. To further improve the performance, we propose a new algorithm based on dynamic programming to solve the keyword search in temporal social network. The main idea is to extend a vertex into a solution by edge-growth operation and tree-merger operation. We also propose two powerful pruning techniques to reduce the intermediate results during the extension. Additionally, all of the algorithms we proposed are capable of handling a variety of ranking functions, and all of them can be made to conform to top-N keyword querying. The efficiency and effectiveness of the proposed algorithms are verified through extensive empirical studies.

Keywords Temporal social networks · Keyword search · Dynamic programming

1 Introduction

Social networking, such as Facebook, Twitter and Tiktok, have grown by leaps and bounds in recent years. Social network datas fully display information about individuals' social relationships, interests, and other information in full. This social network datas contain various data such as posts written by users, and comments that have been retweeted or responded to. In essence, these datas are ones that contain time information. The social network with time informations is called temporal social network (TSN). Temporal social networks have attracted the attention of researchers. Reference [1] studies the problem of connectivity in temporal social networks. Reference [2] discovers the evolution characteristic in a temporal social network. Reference [3] studies the temporal keyword search problem with temporal label in temporal social networks. Reference [4] solves the keywords search problem in temporal social networks according to 3 different timing rules.

Keywords search has been the most fundamental method of analysis in social networks. In social networks, keyword searches enable analysis of user behavior, user interactions, link evolution, opinion spreading, community search, etc [5, 6]. However, the expansion of temporal information presents new challenges for keyword search in temporal social networks. Since there may be different relationships between any two keywords in a temporal social network within the query time interval. For example, Fig. 1 is a social network. From 2000 to 2022, Tom and Mike have two different relationships. Tom, Mike and Jim are friends from 2000 to 2010. Between 2020 and 2022, Tom and Mike form a co-operative relationship through their respective companies and co-operative projects between the two companies. These query results provide a powerful aid to user analytics

✉ Youming Ge
geym@mail2.sysu.edu.cn

Zitong Chen
chenzt53@mail.sysu.edu.cn

Yubao Liu
liuyubao@mail.sysu.edu.cn

[1] Sun Yat Sen University, Guangzhou, China

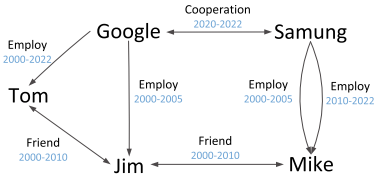


Fig. 1 Temporal social network

in social networks. Social network users can be categorized into distinct groups based on different temporal information. For instance, Tom, Jim, and Mike could all be placed in the same group from 2000 to 2010, but from 2020 to 2022, they could be split into two groups, one for Jim and the other for Tom and Mike.

For keywords searching in temporal social networks, Ref. [4] proposes an optimal path iterator-based algorithm. They suggest the best path iterator, which finds the "best" paths between any two vertices in each snapshot of time, in order to generate results efficiently.

To determine the "best" solution, there are several popular ranking factors: **Factor 1** Ranking by descending order of end time; **Factor 2** Ranking by ascending order of start time; **Factor 3** Ranking by the descending or ascending order of duration/weight. For example, if we want to know the recent work on keyword search on temporal social networks, we may use Factor 1 as the ranking; if we want to know the source of the keyword search on temporal social networks, we may use Factor 2 as the ranking; if we want to know which groups are working on the keyword search problem for a long time, we may use Factor 3 as the ranking.

However, the algorithm proposed by [4] has limitations in solving the problem of keyword search on temporal social networks. In Sect. 2, the disadvantages of the algorithm are covered in more detail.

To solve the keyword search problem on temporal social networks, we first propose a discrete timestamp algorithm. It turns the problem into an equivalent problem that can be solved by existing algorithms. In order to solve the defects in the existing method, we proposed an effective algorithm based on parameterized dynamic programming called *KS*. Similar to [7], we defined a state as a connected tree on a given temporal social networks. Different from existing dynamic programming algorithms, the state in this paper contains time information. We proposed two state operations named "edge-growth" and "tree-merger". To reduce the number of generated states significantly, we proposed two effective pruning techniques. By pruning techniques, for the same time interval and query keyword set, we only save the state with the smallest weight. Therefore, the solution has the smallest weight, that is, the keywords have the closest relationship. At the same time, our algorithms are applied to top-N problems, and the results show that our approach can effectively provide solutions to top-N problems. The efficiency of our algorithms is far better than those proposed by [4].

Our main contributions are as follows: (1) We discover a method discretizing the time interval for the keyword search on temporal social networks. (2) We propose the dynamic programming algorithm called *KS* for the problem of keyword search on temporal social networks. (3) The *KS* algorithm uses two pruning technologies to reduce the number of generated states. (4) We conducted a set of experiments based on two real temporal social networks to verify the efficiency of our algorithm. Compared with state-of-the-art algorithms, our algorithm is nearly 616 times faster.

This paper is organized as follows. In Sect. 2, the difference between the algorithm in [4] with our algorithm is discussed. Section 3 contains our problem definition. Section 4 is the discrete timestamp algorithm. Section 5 is the dynamic programming algorithm. Section 6 is the proposed pruning techniques. Our empirical study is reported in Sect. 7. The related work is discussed in Sect. 8. We conclude in Sect. 9.

2 Discussions and Analysis

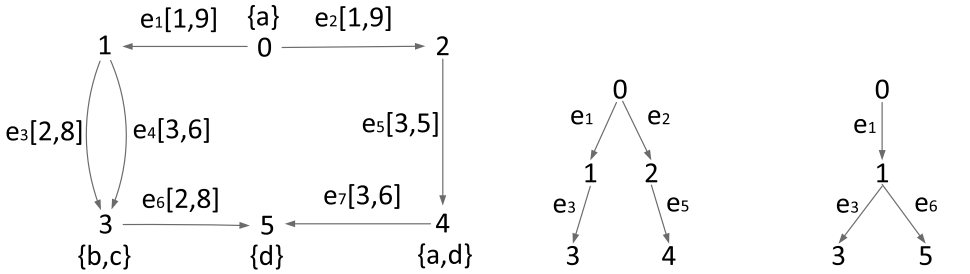
The state-of-the-art algorithm for solving keyword queries in temporal social networks is proposed in [4]. The algorithm has the three following limitations. Compared to the state-of-the-art algorithm, the limitations are three-fold as shown in Table 1.

First, it has a high time complexity $O(M \times X + R \times P)$, where M is the number of vertices that contain at least one of the query keywords, R is the number of candidate results generated, P is the number of snapshots in the dataset, and $X = 2^P \times |V| \times (P + P \times \log |V|) + P^2 \times |E|$ which is the time complexity of the best path iterator. It is

Table 1 Dataset descriptions $K = 10^3$

	Algorithm in [4]	Our KS algorithm
Time complexity	$O(M \times (2^P \times \|V\| \times (P + P \times \log \|V\|) + P^2 \times \|E\|) + R \times P$	$O(\|V\| \times (\|\Gamma_v\| + \log \|V\|) + (\|E\| + \|Q\|) \times \log \|Q\| \times \max \|g_v\|)$
Solution	Not necessarily the closest	The optimal solution
Continuous time intervals	Not applicable	Applicable

Fig. 2 Running example



(a) Temporal Graph G (b) Solution (c) Optimal Solution

clear that as the valid time is divided more meticulously, that is, P becomes larger, the calculation time of the best path iterator will be longer. Our algorithm is based on dynamic programming, and the time complexity of it is $O(|V| \times (|\Gamma_v| + \log|V|) + (|E| + |Q|) \times \log|Q| \times \max|g_v|)$ where $|\Gamma_v|$ is the number of query keywords contained in v, $|Q|$ is the number of states in queue Q during the algorithm process, $|g_v|$ is the number of states with v as the root. The time complexity of our algorithm is obviously smaller than the time complexity of the approach used by [4].

Second, with this approach, in the query results, the relationship between keywords is not necessarily the closest. For example, Fig. 2 shows a temporal social network, including 6 vertices and 7 edges. Each vertex is associated with a set of labels that may contain the user's name, skills, features, and so on, and a weight representing the cost of the user. Each edge is associated with a time interval, which gives the start time and the end time of the communication among the two users, and the edge weight as the communication cost, respectively. User 3 is called b and he has a skill c, correspondingly, and vertex v_3 is marked with the keyword set $\{b, c\}$. We refer to [4] setting the vertex/edge weight to 0/1 simply. Suppose that we are to find the closest relationship among a group of users whose names are Γ_q within the given time interval $[t_\alpha, t_\beta]$. We hope to find users with $\Gamma_q = \{a, b, c, d\}$ within time $[t_\alpha, t_\beta] = [0, 9]$, and the overall weight is as low as possible. By [4], we can only find an approximate result of the above example, which is shown in Fig. 2b. The total weight observed is 4 and the valid time interval is $[2, 5]$. But our method can find the optimal solution which is shown in Fig 2c, the total weight of which is 3, and the valid time interval is $[3, 8]$. It is obvious that the keywords in the optimal solution are more closely related to each other, and has a longer valid time interval.

Third, another weakness of the best path iterator algorithm in [4] is that it can only handle the case that the time information is a set of time snapshots, and it is not applicable for continuous time intervals. However, in the definition of keyword search on temporal social networks problem, the time information could be time intervals, which involves endless time snapshots. Reference [4] assumes that there is a time unit so that time intervals can be divided into a set of time snapshots, but the time unit may not always exist. If we use a small time unit to approximate the time information and apply the approach in [4], not only the efficiency is not pleased as the time complexity is $\Omega(2^P)$, but also the effectiveness is not guaranteed.

3 Problem Definition

In this section, we introduce some definitions and notations, and our problem of keyword search on temporal social networks.

We define the temporal social networks model in a similar way as the temporal XML model used in [9]. Let $G = (V, E)$ be a temporal social networks, where $V(E)$ is the set of vertex(edge) of the temporal social networks. Each vertex $v \in V$ is associated with a set of keywords Γ_v. Let $\Gamma = \cup_{v \in V} \Gamma_v$ be the set of all keywords. The weight of v is denoted by $w(v)$.

Each edge $e \in E$ is defined to be a quintuple which includes a start vertex u, an end vertex v, a time interval $[t_s, t_e]$ in which it is *valid*, and a weight value w, denoted as $e = (u, v, t_s, t_e, w)$, where t_s, t_e and w are non-negative real numbers, and $t_s \leq t_e$. The start vertex, end vertex, start time, arrival time and weight value of e are also denoted by $s(e)$, $a(e)$, $t_s(e)$, $t_e(e)$ and $w(e)$ respectively, namely $s(e) = u$, $a(e) = v$, $t_s(e) = t_u$, $t_e(e) = t_v$ and $w(e) = w$. The valid time of e is denoted by $val(e) = [t_s(e), t_e(e)]$.

For any vertex $v \in V$, let $E_i(v)$ and $E_o(v)$ denote the set of *in-edges* and the set of *out-edges* of v, respectively. Then, $E_i(v) = \{e \in E | a(e) = v\}$, and $E_o(v) = \{e \in E | s(e) = v\}$. The *in-degree* of v is equal to $|E_i(v)|$ and the *out-degree* of v is equal to $|E_o(v)|$. The vertex may be associated with a time interval indicating when it is valid. Without loss of generality, we assume that the valid time interval for each vertex

is the whole time. Our algorithm can be easily extended to the general case.

Example 1 Figure 2a is an example of temporal social network containing 9 vertices 0, 1, 2, ⋯, 6, and 7 edges $e_1, e_2, e_3, \cdots, e_7$. Each edge is associated with a valid time $[t_s, t_e]$. The vertex/edge weight is set to 0/1. The directed edge $e_1 = (0, 1, 1, 9, 1)$ from vertex 0 to vertex 1 indicates the start time $t_s = 1$, the end time $t_e = 9$, and the cost $w = 1$. Besides, the set of in-edges and out-edges of vertex 3 are $E_i(3) = \{e_3, e_4\}$ and $E_o(3) = \{e_6\}$, respectively.

Definition 1 (Path and Continuous Path) A *path* $P = \langle e_1, e_2, \cdots, e_k \rangle$ in G, is a sequence of edges such that $a(e_i) = s(e_{i+1})$, where $1 \le i < k$. We say that P is a *continuous path* from the vertex $s(e_1)$ to the vertex $a(e_k)$ if the time interval of P is valid, that is $val(P) = \cap_{i=1}^{i=k} val(e_i) \neq \emptyset$.

The *weight* of P is defined to be $\mathcal{W}(P) = \lambda \sum_{e \in E(P)} w(e) + (1 - \lambda) \sum_{v \in V(P)} w(v)$, where, $\lambda \in [0, 1]$ is a regulating weight, and $V(P)/E(P)$ is the set of vertices/edges on P.

Example 2 Consider the temporal graph in Fig. 2a, let λ be 1, the weight of each edge set to be 1. The sequence of $\langle e_1, e_3 \rangle$ is a continuous path P_1 with weight $\mathcal{W}(P_1) = w(e_1) + w(e_3) = 2$. The $val(P_1) = [2, 8]$

Definition 2 (Connected Tree) A connected tree T is a tree where each path from the root to the leaf forms a continuous path, and the valid time of T, $val(T) = \cap_{e \in E(T)} val(e)$ should not be empty, where $E(T)$ is the set of edges on T.

Similarly, the *weight* or *cost* of T is defined to be the weighted sum of the total vertex weight in T and the total edge weight in T, namely $\mathcal{W}(T) = (1 - \lambda) \sum_{v \in V(T)} w(v) + \lambda \sum_{e \in E(T)} w(e)$.

Example 3 Consider the temporal graph in Fig. 2b, let λ be 1, the weight of each vertex and edge be 1. Figure 2b is a connected tree T_1. The weight of T_1 is $\mathcal{W}(T_1) = 4$, and the valid time is $val(T_1) = [3, 5]$.

Definition 3 (Best Connected Tree Problem) Given a temporal social network $G = (V, E)$ with a set of all keywords Γ, a set of query labels $\Gamma_q \subseteq \Gamma$, temporal predicates $[t_\alpha, t_\beta]$, and ranking factors, the best connected tree problem is to find a best connected tree T from G which meets the three constraints.

1 Coverage Constraint: $\cup_{v \in V(T)} \Gamma_v \supset \Gamma_q$, i.e. T covers all keywords in Γ_q;
2 Minimum Constraint: $\nexists u \in V(T)$, $\cup_{v \in V(T), v \neq u} \Gamma_v \supset \Gamma_q$, i.e. removing any vertex of T would make T no longer contain all keywords in Γ_q;
3 Time Constraint: $val(T) \cap [t_\alpha, t_\beta]$, i.e. T satisfies the temporal predicates.

Let us take Fig. 2 as an example to illustrate our problem. For each vertex, we have $\Gamma_0 = \{a\}, \Gamma_3 = \{b, c\}, \Gamma_4 = \{a, d\}$, and $\Gamma_5 = \{d\}$. Suppose that the temporal predicate is [0, 9], and the set of query keywords is $\Gamma_q = \{a, b, c, d\}$. The ranking function is ranking by ascending order of relevance. We follow most of the existing works [4, 7, 10–14] to define relevance as the weighted result of tree size. Then, we obtain a best connected tree T in Fig. 2c whose weight $\mathcal{W}(T) = w(e_1) + w(e_3) + w(e_6) = 3$, and the valid time is [2, 8].

Theorem 1 *Our problem is NP-hard.*

Proof The existing GST problem in a weighted and labeled graph, where the vertices with the same labels are in the same group, is a generation of the Steiner tree problem [15]. It is well-known that the existing GST problem is NP-hard [16]. Since additional time information and ranking factors are included in our problem, the existing GST problem can be viewed as a special case of our problem. Thus, our problem is NP-hard. □

4 Discrete Timestamp Algorithm

In this section, we propose an algorithm which turns the problem into the keyword search problem on graph. The differences between our problem and the traditional keyword search on graph includes: (1) the edges have time intervals; (2) the query involves a time interval. If the query time interval degenerate to a single timestamp, then we can see our problem becomes solving keyword search on a graph with edges whose valid time contain the query timestamp. Therefore, we can enumerate all timestamps in the query time interval to get a solution. Though there are endless timestamps in $[t_\alpha, t_\beta]$, we don't need to enumerate all according to Lemma 1.

Lemma 1 *Let* $TS = \{t : \exists e \in E, t_s(e) = t \text{ or } t_e(e) = t\}$, *and let S be the tree of the optimal solution at timestamp* $t \in TS \cap [t_\alpha, t_\beta]$. *If* $TS \cap val(S) \neq \emptyset$, *then there exists* $t \in TS$, *S is valid at t.*

Proof According to the definition of best connected tree problem, $val(S)$ is the valid time of the optimal solution. For $TS \cap val(S) \neq \emptyset$, there exists $t \in TS$, S is valid at t.

□

Lemma 2 *Let* $TS = \{t : \exists e \in E, t_s(e) = t \text{ or } t_e(e) = t\}$, *and let S be the tree of the optimal solution at timestamp* $t \in TS \cap [t_\alpha, t_\beta]$. *If* $TS \cap val(S) = \emptyset$, *the optimal solution is valid at any time in* $[t_\alpha, t_\beta]$, *e.g.* $(t_\alpha + t_\beta)/2$.

Proof Let E_S be the set of edges in the tree S, let $Y = \{t_s(e) : e \in E_S\}, Z = \{t_e(e) : Se \in E_S\}$. Then we have $Y \subset TS$, $Z \subset TS$, besides, Y and Z are finite sets, thus, $\max Y \in TS$, $\min Z \in TS$. Let $t = \min Y \in TS$ or $t = \max Z \in TS$, For S is a connected tree, the valid time of S is $\cap_{e \in E_S} val(e) = [Y, Z]$. S is also the best connected tree, and it meet the time constraint, then the $val(S) = [Y, Z] \cap [t_\alpha, t_\beta]$. If $TS \cap val(S) = \emptyset$, $val(S) = [t_\alpha, t_\beta]$. We can have the optimal solution is valid at any time in $[t_\alpha, t_\beta]$. □

Based on Lemmas 1 and 2, we can have the discrete timestamp algorithm in Algorithm 1. We collect all the timestamps and store it in the set *TS* from line 1 to 11. Then for each timestamp in *TS*, we call existing keyword search on graph algorithms, such as the algorithm in [7] to solve the problem at line 15. Finally, we return the optimal solution among all timestamps at line 18.

Example 4 Consider Fig. 2a as an example, let λ be 1, the weight of each vertex and edge be 1, where $\Gamma_q = \{a, b, c, d\}$, $[t_\alpha, t_\beta] = [0, 9]$, and ranking function is ascending order of weight.

We collect all the timestamps and store it in $TS = 1, 2, 3, 5, 6, 8, 9$. At first, we use the first timestamp in *TS*, $t_1 = 1$. We can get a graph G_1 which has 2 edges, $e'_1 = (0, 1), e'_2 = (0, 2)$. Then, we call existing keyword search on graph G_1 at line 15. There is no solution, and we use the second timestamp in *TS*, $t_2 = 2$. A graph G_2 which has 7 edges, $e'_1 = (0, 1), e'_2 = (0, 2), e'_3 = (1, 3), e'_4 = (1, 3), e'_5 = (2, 4), e'_6 = (3, 5), e'_7 = (4, 5)$. For e'_3 and e'_4 have same vertices, e'_4 can be removed from G_2. The solution e'_1, e'_3, e'_6 is get by calling existing keyword search on graph G_2. Then we change e'_1, e'_3, e'_6 to e_1, e_3, e_6. The solution of temporal graph is get. After we use all the timestamp in *TS*, the optimal solution is get, and shown in Fig. 2c.

Algorithm 1 may need to call keyword search algorithm at most $2 \times |E|$ times at line 15. If the number of calling is large, it would not be good to use Algorithm 1 in terms of efficiency, we should design more powerful algorithms.

Algorithm 1 *Discrete Timestamp*

Input: $G = (V, E)$, the query label set Γ_q, temporal predicates $[t_\alpha, t_\beta]$, and ranking function
Output: The optimal solution.

```
1:  TS ← ∅;
2:  for e ∈ E do
3:      if t_s(e) ∈ [t_α, t_β] then
4:          TS.insert(t_s(e));
5:      end if
6:      if t_e(e) ∈ [t_α, t_β] then
7:          TS.insert(t_e(e));
8:      end if
9:  end for
10: if TS is ∅ then
11:     TS.insert((t_α + t_β)/2);
12: end if
13: use S to collect the optimal solution;
14: for t ∈ TS do
15:     call any keyword search algorithm on G but ignoring edges whose valid
    time doesn't contain t;
16:     update the optimal solution S;
17: end for
18: return S with time interval val(S) ∩ [t_α, t_β];
```

We could have a trade off by sampling some timestamps in $[t_\alpha, t_\beta]$ to replace TS generated from line 1 to 11 in Algorithm 1, then we get an approximate algorithm. A simple sampling method can be the uniform sampling. Let k be a positive integer, let $\Delta t = \frac{t_\beta - t_\alpha}{k}$, then the uniform sampling timestamp set is $\{t_\alpha, t_\alpha + \Delta t, t_\alpha + 2\Delta t, \cdots, t_\beta\}$. The accuracy depends on whether there exist a sample timestamp locating on the time interval of the optimal solution. Specially, if Δ is not larger than the duration of the optimal solution, then there must be such a sample timestamp. Thus, the optimal solution can be found.

5 KS Algorithm

In a nutshell, our solution adopts a dynamic programming paradigm in which we utilizes the similar state representation and state-transition equation in [7, 17]. First, we obtain a state heap in which a set of initial states for each vertex. The states in the heap are sorted according to ranking factors. Second, for the first state in the heap, we check if the state is an optimal solution. If it is, then we return it as an answer. Third, we expand the first state by the edge-growth and tree-merger operations to generate a new state. After that, we check the next state in the heap and continue the state-expansion in the second step. In order to reduce the states that have been generated in the search space, we present two state prunings to remove the weak states and fake states from the search space.

5.1 State Operations

We first describe the some concepts as follows.

Definition 4 (State) A connected tree in G is called a state $(v, X, [t_s, t_e])$ if it is rooted at v, covers all keywords in X, and has a valid time $[t_s, t_e]$.

For example, the tree in Fig. 2c is a state $(0, \{a, b, c, d\}, [2, 8])$. Since the tree has the minimum weight, it corresponds to $T(0, \{a, b, c, d\}, [2, 8])$.

In our solution, we start a state from each vertex, and apply two operations to grow a state into another state. One is the edge-growth operation. It tries to add an edge to the state, where the adding edge should be the in-edge of the root of the state. And the starting vertex of the in-edge becomes the root of the new state. The other is the tree-merger operation. It tries to merge two states rooted at the same vertex into a larger state so that it can cover more labels.

Given a state $z = (v, X, [t_s, t_e])$, we denote the set of states by edge-growth operation on z by $S_g(z)$, and the set of states by tree-merger operation on z by $S_m(z)$. $S_g(z)$ and $S_m(z)$ are constructed by Eqs. 1 and 2, respectively.

$$S_g(z) = \bigcup_{\substack{e \in E_i(v), \\ val(e) \cap [t_s, t_e] \neq \emptyset}} \{e \oplus z\} \tag{1}$$

$$S_m(z) = \bigcup_{\substack{z' = (v, X', [t'_s, t'_e]) \in g_v, \\ X' \not\subset X, [t'_s, t'_e] \cap [t_s, t_e] \neq \emptyset}} \{z' \oplus z\} \tag{2}$$

where g_v is the group of states rooted at v.

The edge-growth operation needs to examine each in-edge e of v one by one, each state $e \oplus z = (s(e), X \cup \Gamma_{s(e)} \cap \Gamma_q, val(e) \cap [t_s, t_e])$ would be generated.

In the tree-merger operation, we consider another state $z' = (v, X', [t'_s, t'_e])$ which is also rooted at v. We want to merge the two states into a larger one $z' \oplus z = (v, X' \cup X, [t'_s, t'_e] \cap [t_s, t_e])$. In order to make the merged state valid, $[t'_s, t'_e] \cap [t_s, t_e] \neq \emptyset$ is required. It is easy to see that there is no need to merge z with z' if $X' \subset X$, i.e. the covered label set by z' is the subset of the one of z.

5.2 KS Algorithm

We are ready to introduce the *KS* Algorithm to solve the problem.

Connected trees may share some components, to save some computations, we start from a leaf and grow it into a connected tree as describing in Eqs. 1–2.

In the initialization, we turn each vertex v into a state with v as the root, if $X_v = \Gamma_v \cap \Gamma_q$ is non-empty. The time is set to be the query time $[t_\alpha, t_\beta]$, and the initial state $(v, X_v, t_\alpha, t_\beta)$ has weight equal to the weight of v. Then we push each state into a heap Q. Q sorts the states in Q based on ranking factors.

While Q is not empty, we do the extension from a state with the current best ranking to a possible solution. The state z popped from Q is the one with the best ranking currently. If the current best state has already covered the query label set L, then we can return z as the best solution. otherwise, we should extend z by applying the two state operations.

The description of *KS* algorithm without pruning is given in Algorithm 2. Line 1 to 5 is the initialization. We start checking the states while there exists valid states in Q at line 11. The state z popped from Q at line 12 is the one with the best ranking. If z has covered all query labels, we return z as the optimal solution at line 13. Otherwise, we call the two state operations at line 26 and 27.

Algorithm 2 *KS*

Input: $G = (V, E)$, the query label set Γ_q, temporal predicates $[t_\alpha, t_\beta]$, and ranking function
Output: The optimal solution.

```
1:  Q ← ∅;
2:  for each v ∈ V do
3:      X_v = Γ_v ∩ Γ_q
4:      if X_v ≠ ∅ then
5:          Q.push(((v, X_v, [t_α, t_β]), w(v)));
6:      end if
7:  end for
8:  if Use Pruning then
9:      g_v ← ∅ ∀v ∈ V;
10: end if
11: while Q ≠ ∅ do
12:     (z = (v, X, [t_s, t_e]), W(z)) ← Q.pop();
13:     if X = L then return z;
14:     end if
15:     if Use Pruning then
16:         isWeak ← false;
17:         for z′ ∈ g_v do
18:             if z′ state dominates z then
19:                 isWeak ← true;
20:                 break;
21:             end if
22:         end for
23:         if isWeak then continue;
24:         end if
25:     end if
26:     Edge-growth;
27:     Tree-growth;
28: end while
29: return null;
```

The edge-growth operation and tree-merger operation in Algorithm 3 follow Eqs. 1 and 2, respectively. Note that in the tree-merger operation, the merge is symmetric for z and z', i.e. $z \oplus z'$ is equal to $z' \oplus z$, therefore, we can restrict the cost of z' should not larger than z when doing merge to avoid duplicated computation. Thus, we only use states that have been popped before for merging at line 9. During and after the two operations, we should push the new states into Q at line 5 and 12.

Algorithm 3 *Two state operations (No pruning)*

```
1: Operation Edge-growth:
2: for each e ∈ E_i(v) do
3:     if val(e) ∩ [t_s, t_e] ≠ ∅ then
4:         s_g ← e ⊕ z
5:         Q.push((s_g, W(s_g)))
6:     end if
7: end for
8: Operation Tree-merger:
9: for each (z' = (v, X', [t'_s, t'_e]), W(z')) has been popped do
10:    if X' ⊄ X and [t_s, t_e] ∩ [t'_s, t'_e] ≠ ∅ then
11:        s_m ← z ⊕ z'
12:        Q.push((s_m, W(s_m)))
13:    end if
14: end for
```

5.3 Top-*N* Query

To support top-*N* keyword search on temporal graph, we can change the stop condition at line 13 of Algorithm 2 to "if $X = L$, then add the solution into top-*N* result; if *N* results found, then return otherwise continue". Here, we "continue" because we need to find more solutions, and there is no need to do state operations for a solution.

6 State Pruning

In the two operations, there will generate a lot of states, which make the number of states becomes larger and larger. It affects both the efficiency and the memory. It would be helpful if we have techniques to identify the weak and fake states. In this section, we propose two state pruning techniques for weak states and fake states.

6.1 Weak State

Definition 5 (State Dominated and Weak State) Given two states $z_1 = (v, X_1, [t_{s_1}, t_{e_1}])$ and $z_2 = (v, X_2, [t_{s_2}, t_{e_2}])$, if $X_1 \subset X_2, [t_{s_1}, t_{e_1}] \subset [t_{s_2}, t_{e_2}]$ and $\mathcal{W}(z_1) \geq \mathcal{W}(z_2)$, then we say that z_1 is a weak state dominated by z_2. A state is called a weak state iff it is state dominated by another state.

The *KS* algorithm extends each leaf vertex into a possible solution by a bottom to top manner. That is, a state is extended either by merging two states at the same root, or by adding the in-neighbour of the root as the new root. By the bottom to top extension, we discover the following lemma to filter out the weak state from the search space.

Lemma 3 *A weak state can be pruned if we apply the bottom to top extension.*

Proof We use the notation in the weak state definition, and prove that the weak state z_1 can be pruned by z_2.

We use mathematical reduction to prove this claim: for any optimal solution which is expanded from a weak state z_1 after n operations, we can get an optimal solution be expanded from z_2 within n operations, where z_1 is state dominated by z_2.

If an optimal solution can be extended from z_1 after 0 operation, that is, z_1 itself is optimal. Note that z_2 is no worse than z_1 on all sides, thus z_2 is also optimal. So the claim holds.

Assume the claim holds for $n = k$, we prove that it can hold for $n = k + 1$. The first operation on z_1 can be either edge-growth operation or tree-merger operation.

Case 1: the first operation is the edge-growth operation. Let e be the in edge that extends z_1. As z_2 is also rooted at v as z_1, z_2 can also be extended by adding e. It is easy to verify that $e \oplus z_1$ is stated dominated by $e \oplus z_2$. An optimal solution can be extend from $e \oplus z_1$ by k operations, according to the assumption, we can get an optimal solution extended from $e \oplus z_2$ within k operations, thus, this optimal solution can be extended from z_2 within $k + 1$ operations.

Case 2: the first operation is the tree-merger operation. Let z' be the state that merges with z_1, and $z_3 = z' \oplus z_1$. If z' is z_2, then it is clear that the optimal solution can be extended from z_2 with $k + 1$ operations too. Otherwise, let $z_4 = z' \oplus z_2$, it is easy to verify that z_3 is stated dominated by z_4. An optimal solution can be extended from z_3 with k operations, according to our assumption, we can get an optimal solution extended from z_4 within k operations,

thus, the optimal solution can be extended from z_2 within $k+1$ operations.

In above, we can conclude that a weak state can be pruned if we apply the bottom to top extension. □

Specially, when we do the edge-growth operation $e \oplus z$, if $X \subset \Gamma_{s(e)}$, then we can immediately see that $e \oplus z$ will be a weak state dominated by the state at vertex $s(e)$. See line 3 in Algorithm 4.

6.2 Fake State

Definition 6 (Fake State) A state is fake if it can not be part of any optimal solutions.

By the definition, we can see that it is safe to prune these fake states, because they cannot be extended into an optimal solution. To identify the fake state, we propose the following lemma.

Lemma 4 *Let v be the root of an optimal solution z, let $c = |\Gamma_v \cap \Gamma_q|$ be the number of query labels contained in v, and $c < |\Gamma_q|$, let l be the number of leaves of z. Then we have: $l + c \leq |\Gamma_q|$.*

Proof Let the leaves be $\{u_1, u_2, \cdots, u_l\}$. Let x_i be the number of query labels which are covered by u_i and not covered by v or other leaves, then we have $\sum_{i=1}^{i=l} x_i + c \leq |\Gamma_q|$. If $x_i = 0$, then removing u_i and the associated edge from z can give a better solution, contradictory to z is optimal. Thus, $x_i \geq 1$. So we have: $|\Gamma_q| \geq \sum_{i=1}^{i=l} x_i + c \geq \sum_{i=1}^{i=l} 1 + c = l + c$. □

Lemma 4 can give us the following corollary to identify a fake state directly by simply considering the number of labels covered by the root and the number of leaves. See line 5 and 14 in Algorithm 4.

Corollary 1 *Let v be the root of a state z, let X be the set of query labels covered by z, let $c = |\Gamma_v \cap \Gamma_q|$ be the number of query labels contained in v, let l be the number of leaves(except v) of z. If $l + c > |X|$, then z is a fake state and can be pruned.*

Proof Similarly, let the leaves be $\{u_1, u_2, \cdots, u_l\}$, and x_i be the number of query labels which are covered by u_i and not covered by v or other leaves, and we have $\sum_{i=1}^{i=l} x_i + c \leq |X|$. If $l + c > |X|$, then $\sum_{i=1}^{i=l} x_i + c \leq |X| < l + c$. So, $\sum_{i=1}^{i=l} x_i < l$. Thus, there must be a $x_i = 0$ where $i \geq 1$. Then, in order to get a better solution, at least one leaf u_i and the associated edge need to remove from z. Thus, z is a fake state and can be pruned. □

The correctness of Corollary 1 is clear since z is not the optimal state in terms of query label set X, which implies z could not be part of the optimal solution.

6.3 KS with Pruning

Now we are ready to introduce the *KS* algorithm with the pruning techniques applied in Algorithm 2.

When checking whether a state z is weak or not, we need to check whether there exists a state that can state dominated z. Note that z has the best ranking in Q currently, none of states in Q can state dominate z, only the states that were popped from Q before may state dominate z. Thus, for efficiency checking weak states, we can group the popped states by the root v, and use a list g_v to store them at line 19 in Algorithm 4. The process of weak state checking is from line 16 to 23 in Algorithm 2. Another benefit for using g_v is that g_v contains all the states that need to perform tree-merger operation for a popped state rooted at v, see line 11 in Algorithm 4.

The complexity of checking a weak state is much higher than the complexity of checking a fake state. Therefore, we apply the weak state checking only when a state is popped out, and we apply the fake state checking immediately when the state is generated.

Algorithm 4 *Two state operations*

```
1:  Operation Edge-growth:
2:  for each e ∈ E_i(v) do
3:      if val(e) ∩ [t_s, t_e] ≠ ∅ and X ⊄ Γ_{s(e)}; then
4:          s_g ← e ⊕ z;
5:          if s_g is not a fake state then
6:              Q.push((s_g, W(s_g)));
7:          end if
8:      end if
9:  end for
10: Operation Tree-merger:
11: for each (z' = (v, X', [t'_s, t'_e]), W(z')) ∈ g_v do
12:     if X' ⊄ X and [t_s, t_e] ∩ [t'_s, t'_e] ≠ ∅ then
13:         s_m ← z ⊕ z';
14:         if s_m is not a fake state then
15:             Q.push((s_m, W(s_m)));
16:         end if
17:     end if
18: end for
19: g_v ← g_v ∪ {z};
```

Algorithm Example Consider Fig. 2a as an example, where $\Gamma_q = \{a, b, c, d\}, [t_\alpha, t_\beta] = [0, 9]$, and ranking function is ascending order of weight.

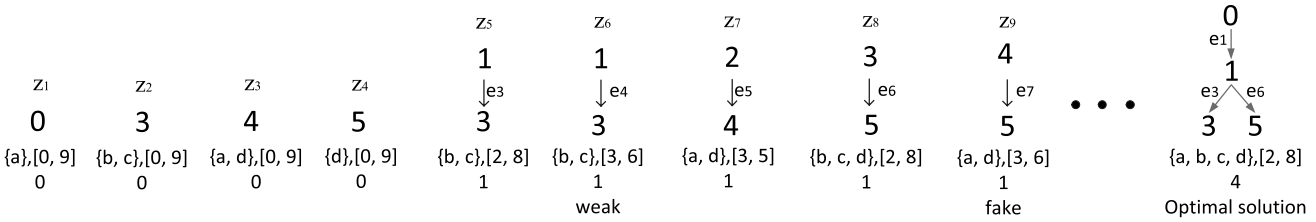


Fig. 3 Example for *KS* Algorithm

Q is initialized with four states: $(z_1 = (0, \{a\}, [0, 9]), 0)$, $(z_2 = (3, \{b, c\}, [0, 9]), 0)$, $(z_3 = (4, \{a, d\}, [0, 9]), 0)$ and $(z_4 = (5, \{d\}, [0, 9]), 0)$ as shown in Fig. 3. The list g_v for each vertex is set to be empty set.

Q is non-empty, the top state $z_1 = (0, \{a\}, [0, 9])$ is popped from Q. Since $\{a\} \neq \Gamma_q$, z_1 is not a solution, Then we check whether z_1 is weak by comparing with states in g_0. Since g_0 is empty currently, none state dominates z_1. $E_i(0) = \emptyset$ and $g_0 = \emptyset$, we no need to do edge-growth operation and tree-merger operation. Next, the top state $z_2 = (3, \{b, c\}, [0, 9])$ is popped from Q. z_2 is also not a solution, Next, the algorithm performs the edge-growth operation. Since $e_3 \in E_i(3)$, $z_5 = e_3 \oplus z_2 = (1, \{b, c\}, [2, 8])$ is generated. Next, we check whether z_5 fake or not. Since the root of z_5 is vertex 1, covers 0 query label, and the number of leaves in z_5 is 1, thus, z_5 is not fake. Then, $(z_5, w(z_5) = 1)$ is pushed into Q. Similarly, we get $z_6 = e_4 \oplus z_2 = (1, \{b, c\}, [3, 6])$, and it is not a fake state, so we push z_6 into Q. Next, we start the tree-merger operation, but g_1 is empty, so no merge is needed, and we push z_2 into g_1. Similar, we pop z_3, and z_4 one by one from Q, and $z_7 = (2, \{a, d\}, [3, 5])$, $z_8 = (3, \{b, c, d\}, [2, 8])$, and $z_9 = (4, \{a, d\}, [3, 6])$ are generated respectively.

Example of weak state: We pop z_6 from Q, and find that z_6 is dominated by z_5. z_5 and z_6 have the same root vertex 1, and cover the same labels($\{b, c\}$), but $[3, 6] \subset [2, 8]$, thus, z_6 is weak state.

Example of fake state: We pop z_4 from Q, and find that z_4 is not weak, because it is not state dominated by states in g_4. We do tree-growth and push $z_8 = z_4 \oplus e_6 = (0, \{b, c, d\}, [2, 8])$ into Q with $W(z_8) = 1$. $z_9 = z_4 \oplus e_7 = (4, \{a, d\}, [3, 6])$, note that z_9 is rooted at vertex 4(cover 2 query labels) and has 1 leaves(vertex 5), but the number of query labels covered by z_9 is $2 < 2 + 1$, thus, z_9 is a fake state. Keep on going with the algorithm, we can finally get the optimal solution as shown in Fig. 3.

Algorithm Complexity We first analyze the time complexity of the two state operations in Algorithm 4.

Operation Edge-growth: The main cost inside the for loop comes from line 6, with time complexity $O(\log |Q|)$. So the overall time complexity is $O(|E_i(v)| \times \log |Q|)$.

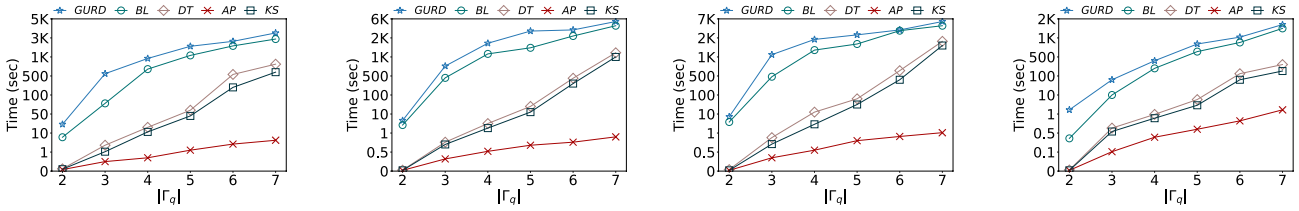
Operation Tree-merger: The main cost inside the for loop comes from line 15 with time complexity $O(\log |Q|)$. So the overall time complexity is $O(|g_v| \times \log |Q|)$.

Then, we show the time complexity in Algorithm 2. We first analyze the time complexity of initialization. Line 3 takes $O(|\Gamma_v|)$ since we can use hashing collision detection for each label, and use a bit vector with length $|\Gamma_q|$ to represent the result of intersection. Line 5 takes $O(\log |V|)$ for the pushing in Q. Thus, the initialization(line 1 to 5) takes $O(|V| \times (|\Gamma_v| + \log |V|))$. Next, we analyze the main cost of lines in the while loop. Line 12 takes $O(\log |Q|)$. The time complexity for checking whether a state is weak or not is $O(|g_v|)$. The number of iterations in this while is $\sum_{v \in V} |g_v|$. Then, the time complexity for the while loop is $O(\sum_{v \in V} |g_v| \times (\log |Q| + |g_v| + |E_i(v)| \times \log |Q| + |g_v| \times \log |Q|)) = O(\sum_{v \in V} |g_v| \times (|E_{i(v)}| + |g_v|) \times \log |Q|) = O((|E| + |Q|) \times \log |Q| \times \max |g_v|)$. Finally, the complexity of Algorithm 2 is $O(|V| \times (|\Gamma_v| + \log |V|) + (|E| + |Q|) \times \log |Q| \times \max |g_v|)$.

The complexity of g_v is $O(2^{|\Gamma_q|} \times |TS|)$. The key point is that the states in g_v have no dominated relationship. For a state with X and time interval $[t_s, t_e]$, only the one with the smallest weight would be in g_v, so $|g_v|$ is no more than the number of subset $X \subset \Gamma_q$ times the number of intervals. There are $|TS|$ timestamps, so the number of intervals is $O(|TS|^2)$. However, we can pick at most $|TS|$ time intervals, in which we cannot find two intervals that have a subset-superset relationship.

Table 2 Dataset descriptions $K = 10^3$

Dataset	$\|V\|$	$\|E\|$	Avg degree	Tcime instants
IMDB	145 K	397 K	2.7	17.3 K
SNAP	256 K	420 K	1.6	20.0 K
Wiki-Fr	2,210 K	4,412 K	2.0	21.3 K
DBLP	3,812 K	4,021 K	1.1	4.5 K



(a) IMDB (b) SNAP (c) Wiki-Fr (d) DBLP

Fig. 4 Effect of $|\Gamma_q|$ on running time

7 Experiment

In this section, we evaluate the performance of our proposed algorithms on two real datasets.

Environment We run all experiments on a machine with a 3.6Ghz Intel Core i7-9700K CPU and 32 GB RAM running Ubuntu 18.04 LTS Linux OS. All algorithms were implemented in C++.

Algorithms In the experiments, we compare 2 baselines, namely *BL* and *GURD*, with our 3 algorithms, namely *DT*, *AP*, and *KS*.

BL [4]. *BL* is the state-of-the-art algorithm for solving keyword query in temporal graph.

GURD [3]. *GURD* is to find a group of keywords in temporal graph which does not consider the ranking factors, time constraint, and minimum constraint in our problem. For *BL* is the unique algorithm for solving keyword query in temporal graph, *GURD* is added as baseline. When *GURD* process a query, the ranking factors, and minimum constraint is ignored.

DT is Algorithm 1, and it call a dynamic programming algorithm on keyword search problem in [7]. *AP* is approximation algorithm. *KS* is Algorithm 2 with pruning.

Datasets We use four real-world datasets, namely IMDB,[1] SNAP,[2] Wiki-Fr,[3] and DBLP[4] datasets. These datasets are widely used in related studies of temporal graph [3, 4, 18]. The IMDB dataset is a temporal network containing IMDb's premier title and name entertainment datasets. 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 start time of the edges are set as the release year of a title or TV Series end year. We randomly generate the end time for each edge in the following way: we set the default probability that any two edges have at least one common time instant as 70%, and also vary this probability in evaluation. The SNAP dataset is a temporal network representing Wikipedia users editing each other's Talk page in which each vertex represents a user, and each edge represents the interaction of two users. The Wiki-Fr is a temporal graph for French articles in Wikipedia. For SNAP and Wiki-Fr, users' names are used as labels. The start time of each edge is the time when the two users interact. We set the end time with the same method of IMDB. For the DBLP temporal graph, each vertex corresponds to an author, and the edges denote the relationships among them such as the co-author relationship, etc. Their names and their papers' names are used as the labels. The start time of an edge is set as the paper's publication time. The publication time includes the year, and month of publication. If there is no month in the data, then we assume that it was published on January. The end time of an edge is set as now(202106) which is the same as [4]. Both datasets assume the unit weight on edges and no weight on nodes as [4]. The details for the temporal graphs are given in Table 2.

The characteristics of these four datasets are different. First, graph structures are different. In the IMDB dataset, there are more edges between two vertices because there are frequent connections between the workers involved. In the DBLP dataset, a directed path from a vertex to every other vertices exists, and there are citation paths. For SNAP and Wiki-Fr datasets, the graph structures are more general. Thus given a set of query keyword, in IMDB dataset will spend more time on edge processing, and the solution can be found for sure in DBLP dataset. Second, temporal characteristics are different. In the DBLP dataset, each edge has a longer valid time. Because citation relationship does not disappear once it occurs, the edge of the citation lasts the longest. Thus, The valid time of any two edges is more likely to intersect in DBLP.

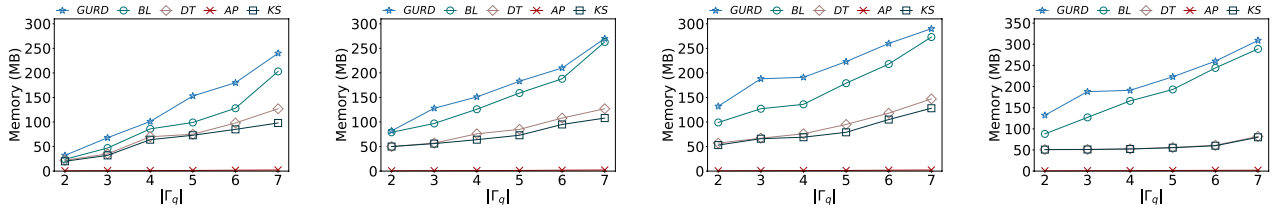
Settings We vary two parameters in our experiments, namely $|\Gamma_q|$ and f, where $|\Gamma_q|$ is the number of labels in the given query label set Γ_q and f is the average number of vertices covering each label in the query (i.e., the label frequency). $|\Gamma_q|$ is in $\{2, 3, 4, 5, 6, 7\}$ with default 4, f is in $\{100, 200, 300, 400, 500\}$ with default 300 [7]. The time

[1] http://www.imdb.com/interfaces/.

[2] http://snap.stanford.edu.

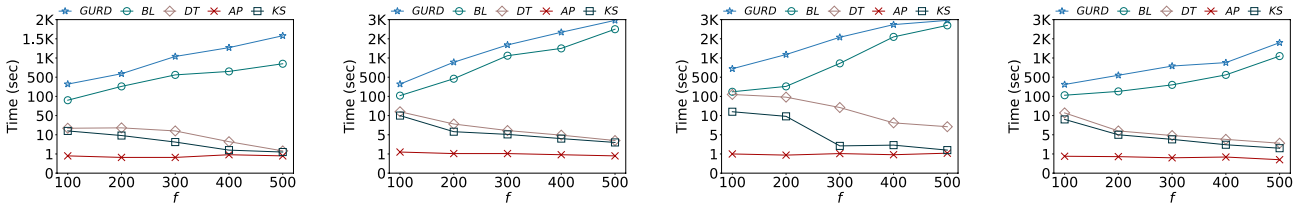
[3] http://www.konect.cc/.

[4] http://dblp.dagstuhl.de/xml/.



(a) IMDB (b) SNAP (c) Wiki-Fr (d) DBLP

Fig. 5 Effect of $|\Gamma_q|$ on memory consumption



(a) IMDB (b) SNAP (c) Wiki-Fr (d) DBLP

Fig. 6 Effect of f on running time

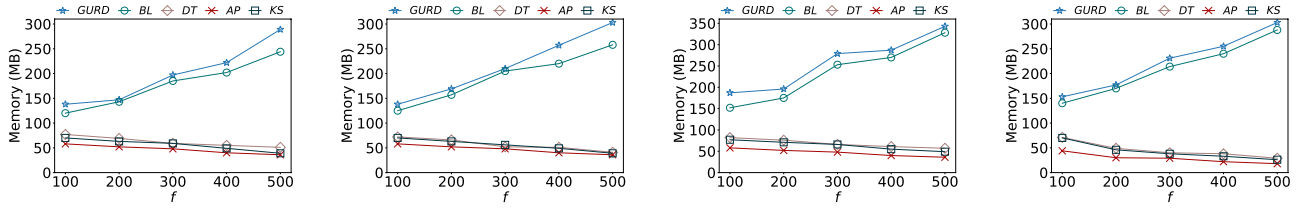
interval setting of all experiments corresponds to $[0, \infty]$, in which case the queries are most computationally challenging, since we need to consider every time in the temporal graph. The default ranking factor is ascending weight on both datasets. In each test, we generate 100 queries with $|\Gamma_q|$ labels randomly, and report the average performance. Algorithm 1 calls Algorithm 2 to search the query keywords on the temporal graph. For the approximate algorithm, k is set to 100. All algorithms perform similarly on the four datasets, we usually take SNAP as examples for the sake of space.

Effect of $|\Gamma_q|$ on running time As shown in Fig. 4, the running time increases with the increase of $|\Gamma_q|$. This is because the increase of $|\Gamma_q|$ will result in the increase of algorithm search space. The running time of our algorithm *KS* and *AP* is obviously smaller than that of *BL*. In particular, as shown in Fig. 4b, for $|\Gamma_q| = 4$, the baseline algorithm *BL* requires 1,217 s, and *GURD* requires 1,538 s, but our proposed algorithms *KS* and *AP* only take 3.7 and 0.8 s, respectively. Our algorithm *KS* is 329 times faster than *BL*. With the increase of $|\Gamma_q|$, *KS* need to generate more state for finding the optimal solution, and *KS* cost more time to find the optimal solution.

For *BL*, the best paths between two vertices with query labels in each snapshot must be found. As the number of given query labels increases, so does the number of optimal paths. As a result, the time of *BL* increases. Figure 4 illustrates how the time of *BL* increases as $|\Gamma_q|$ increases. As we discuss in Sect. 2, the length of time instants has an effect on *BL*. The length of time instants becomes larger, the time of *BL* will be larger. For example, DBLP has a time instants of 4.5K, which is very large for *BL*. However, length of time instants has little effect on on *KS*. So, *KS* is faster than *BL*. For *DT*, with the increase of $|\Gamma_q|$, the dynamic programming algorithm on keyword search problem need more time. And the length of time instants have an impact on the efficiency of discrete timestamp algorithm. However, the length of time instants does not have much effect on *KS*. Therefore, *KS* is much faster than *DT*. For the approximate algorithm does not consider the time information on the edge, *AP* only needs to calculate 100 queries without time information, so it is faster than *KS*. Although *GURD* ignores ranking factors and minimum constraint in our problem, *GURD* has no pruning technique and is therefore slower than *BL* and *KS*.

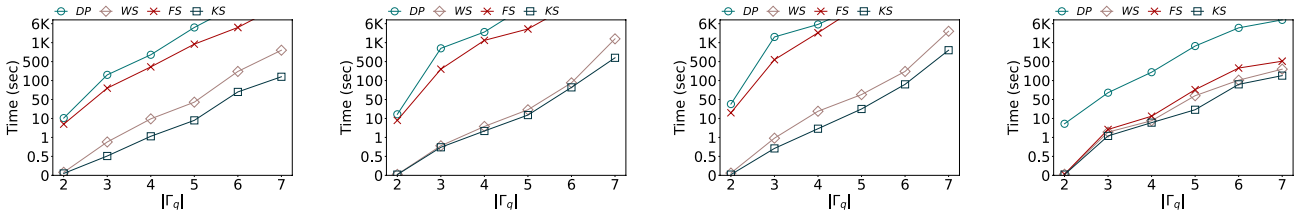
Effect of $|\Gamma_q|$ on memory consumption We conduct the experiments on the four datasets and the results are shown in Fig. 5. In general, the memory consumption increases with the increase of $|\Gamma_q|$. This is because that there are more states need to be stored as the $|\Gamma_q|$ becomes larger. As shown in Fig. 5, the memory consumption of *KS* (*AP*) is less than 150MB in most case. The maximal consumption is about 133MB for *KS* as $|\Gamma_q| = 7$ in Fig. 5c, while the memory consumption of *BL* is 274MB.

For *BL*, with the number of given query labels increases, so does the number of optimal paths. As a result, the memory of *BL* increases. Figure 5 illustrates how the memory of *BL* increases as $|\Gamma_q|$ increases. However, *KS* does not need to find all the best paths. So, *KS* needs less memory than *BL*.



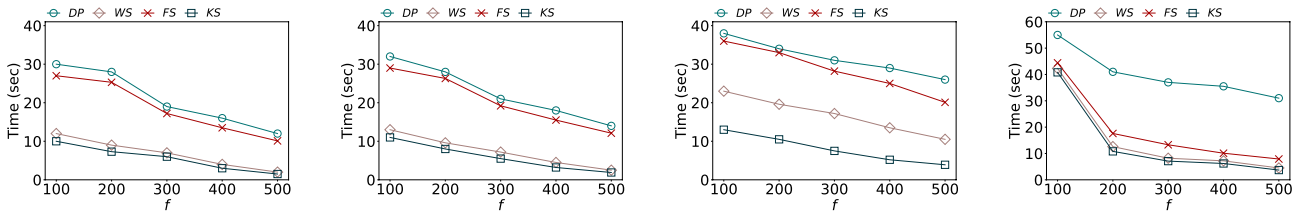
(a) IMDB (b) SNAP (c) Wiki-Fr (d) DBLP

Fig. 7 Effect of f on memory consumption



(a) IMDB (b) SNAP (c) Wiki-Fr (d) DBLP

Fig. 8 Effect of state pruning varying $|\Gamma_q|$



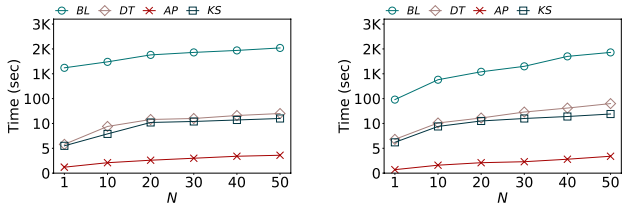
(a) IMDB (b) SNAP (c) Wiki-Fr (d) DBLP

Fig. 9 Effect of state pruning varying f

For *DT*, the number of vertices and edges and the length of time instants have an impact on the efficiency of discrete timestamp algorithm. So, with the increase of $|\Gamma_q|$, the memory of *DT* is increasing. Therefore, *KS* is much faster than *DT*. For *AP*, it only needs to calculate 100 queries without time information, so it needs less memory than *KS*. For *GURD*, it has no pruning technique, it need more memory than *BL* and *KS*.

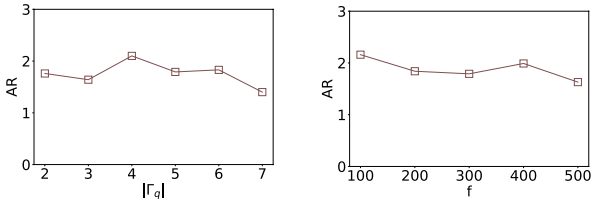
Effect of f on running time As shown in Fig. 6, the running time of *KS* decreases with the increase of f. With the increase of f, there are more vertices containing the query labels. Then, it is faster for the algorithm to achieve the optimal solution. However, for *BL*, with the increase of f, *BL* needs to find more "best" paths, and costs more time. In general, the running time of *KS* is obviously smaller than *BL* and *GURD*. As shown in other figures, *KS* can obtain the optimal solution in 10 s in most cases. In particular, for $f = 300$, in Fig. 6a, *KS* takes 6.3 s, *BL* needs more than 373 s, *GURD* takes 1288s. *KS* is 59 times faster than *BL*, and is 204 times faster than *GURD*.

For *KS*, in order to get the optimal solution, each state need to do the edge-growth operation and tree-merger operation. As f grows, so does the number of initialized states. Then, two states merge into a new state will be earlier. And pruning techniques can also find the weak state and fake state earlier. Then, it is faster for *KS* to achieve the optimal solution. For *BL*, with the increase of f, more best paths between two vertices with query labels in each snapshot need be found. As a result, the time of *BL* increases. Figure 6 illustrates how the time of *BL* increases as f increases. For *DT*, it call a dynamic programming algorithm on keyword search which edge does not contain time. With the increase of f, the dynamic programming algorithm needs less time. And the length of time instants have an impact on



(a) SNAP (b) DBLP

Fig. 10 Effect of N on running time



(a) Effect of $|\Gamma_q|$ (b) Effect of f

Fig. 11 Effect of approximation algorithm

the efficiency of discrete timestamp algorithm. Therefore, *KS* is faster than *DT*. For *AP*, it only needs to calculate 100 queries without time information, so it is still faster than *KS*. For *GURD*, it has no pruning technique and it is slower than *BL* and *KS*.

Effect of f on memory consumption In this experiment, the value of $|\Gamma_q|$ is the default value 4, and the results are shown in Fig. 7. The memory consumption of *KS* decreases as the increase on f as shown in Fig. 7. This is because, with the increase of f, there are few states need to be stored in the algorithms. For *BL*, the memory consumption increases with the increase of f. This is because, with the increase of f, there are more best path iterators need to be stored in the algorithms. In particular, as shown in Fig. 7c, for $f = 500$, the baseline algorithm *BL* algorithm requires 316.7MB, but our proposed algorithms *KS* only take 54.8MB.

For *KS*, two states merge into a new state will be earlier. And pruning techniques can also find the weak state and fake state earlier. The total number of states is smaller. Then, *KS* need less memory to get the optimal solution. For *BL*, with the increase of f, more best paths between two vertices with query labels in each snapshot need be found. So, *BL* need more memory for the "best" paths. For *DT*, it call a dynamic programming algorithm on keyword search which edge does not contain time. With the increase of f, the dynamic programming algorithm needs less memory. Although, the state in *DT* does not contain time, and it need less memory than

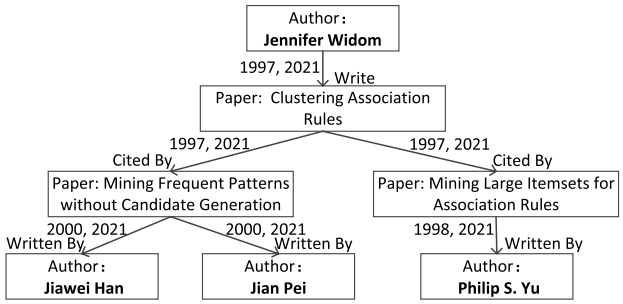


Fig. 12 A case study

the state in *KS*. The length of time instants has an effect on *DT*. The length of time instants becomes larger, the memory of *DT* will be larger. Therefore, *KS* need less memory than *DT*. For *AP*, it only needs to calculate 100 queries without time information, so the memory of it is still smaller than *KS*. For *GURD*, it has no pruning technique and it need more memory than *BL* and *KS*.

Effect of state pruning There are two state pruning techniques in Algorithm 2, namely weak state and fake state. In Figs. 8 and 9, *DP* means the dynamic programming algorithm without any pruning, *WS* means the case that we only enable the weak state pruning in Algorithm 2, and *FS* means the case that we only enable the fake state pruning to Algorithm 2. The running time can be dramatically reduced since lots of states can be pruned by the prosed techniques. As shown in Fig. 9c, we can rank them as *DP*, *FS*, *WS*, *KS* by the ascending of efficiency. It shows that both pruning are efficient, and the weak state pruning is more efficient than the fake state efficient. Limited in 6K seconds, *DP* cannot finish when $|\Gamma_q| = 5, 6, 7$, and *FS* cannot finish when $|\Gamma_q| = 6, 7$, while *WS* and *KS* only need 1,611 and 1,319 s even when $|\Gamma_q| = 7$. It shows that the pruning techniques are important for our algorithms in terms of efficiency.

Effect of top-N In the set of experiments, we test queries on the datasets varying values of N with top-N solutions, ranging from 1 to 50. The ranking function is ascending by relevance. The results are shown in Fig. 10. In general, the running time increases with the increase of N. The running time of our algorithm *KS* is smaller than that of *BL*. In particular, as shown in Fig. 10a, for $N = 50$, the baseline algorithm *BL* algorithm requires 1,349 s, but our proposed algorithm *KS* only takes 9.6 s.

Effect of approximation algorithm Approximation algorithm reduces the search space by simple time sampling, thus speeding up the query. Through the above experiments, we can find that AP is the fastest, but it is not the best solution. For the default ranking factor of ascending weight, we set the approximation ratio(AR) as the weight of the

approximate solution divided by the weight of the optimal solution. To further illustrate the effect of approximation algorithm, we test the approximation ratio on SNAP dataset by changing $|\Gamma_q|$ and f. The experimental results are shown in Fig. 11.

Case Study The case study is on the DBLP datasets. 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 Fig. 12, which shows the relationship between the specified authors and their papers during the specified time, and will be useful for network analysis. 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 paper 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.

8 Related Work

Keywords Search on Graph/Social Network Keyword query on the graph/social network has always attracted the attention of scholars [19]. The majority of researches adopt the minimal tree semantics.

By regarding entities as nodes and relationships as edges, relational, XML and HTML data can be represented as graphs, then keyword search on graph starts to be popular. Bhalotia et al. [12] proposed a system named BANKS to support keyword-based search on relational database by modeling tuples as nodes in a graph. Hristidis et al. [20] adapted IR-style document-relevance ranking strategies to the problem of processing free-form keyword queries over RDBMSs. Kacholia et al. [13] combine top-down search from roots and bottom-up search from leaves, and proposed a novel frontier prioritization technique based on spreading activation to guide the search. Kimelfeld et al. [11] modified the general procedure of Lawler to reduce the problem of enumerating in ranked order to the problem of finding an optimal answer under constraints, but they suffer from the "Steiner-tree bottleneck". Luo et al. [21] proposed efficient query processing methods that have minimal accesses to the database, but it was based on their new ranking method. Sayyadian et al. [22] studied the keyword-search problem over heterogeneous relational databases and proposed a solution named Kite to solve it. Reference [23] proposed a simple yet flexible query language, and develop its semantics to enable intuitively appealing extraction of relevant fragments of information. A search engine based on an incremental algorithm for enumerating subtrees in a 2-approximate order for keyword proximity search in complex data graphs [14]. A principled probabilistic approach to query rewriting was proposed in [24].

Some works use other result definitions for keyword query on the graph, e.g. subgraphs [25], database tuples [26].

The best-known exact algorithm for keyword query in a relational database is introduced in [7], it formulated the query as a group steiner tree(GST) problem in a directed graph. An improved algorithm for a relatively large graph or label set is proposed in [17]. Reference [27] further considers the GST problem with node and edge weights, and proposes an approximate algorithm by using a dynamic programming approach. Reference [28] proposes a query relaxation algorithm which solves the small even emptyresult sets when performing query operations in temporal graph databases. How to handle the time information is not included in these works.

Temporal Databases/XML/Social Network Temporal databases extend the relational model by allowing a tuple to have a valid time and/or a transaction time. Some query languages have been proposed for temporal relational data, such as TQUEL [29], TSQL [30], SQL3 [9], and ChronoGraph [31]. However, due to the complexity of query syntax and the low efficiency of graph processing, these methods are not suitable for ordinary users to search temporal graphs.

Many temporal graph storage techniques consider a temporal graph as a sequence of graph snapshots, where each snapshot depicts the state of the historical graph at a past time point [3, 4, 18, 32–34]. These works focus on the storage and query engine to deal with the outgrowing size of data while being able to query efficiently. However, [18, 33] do not consider the relationship between the keywords of the query. [3] studies the temporal keyword search problem with temporal label in temporal social networks. [32] focus on searching path on temporal graph. However, [3, 32] does not consider the weight of vertex and edge, ranking factors, which are different from our works. Reference [4] focus on the solution of keyword searching on graph database, where time information is associated with both the vertices and edges of the temporal graphs. They divide the time information on the temporal graph into time snapshots, and then finds the temporal solution. The number of time snapshots seriously affects the efficiency of the algorithm. Reference [34] discusses the definition and topological structure of time-dependent graphs, as well as models for their relationship to dynamic systems. Our work can address keyword query on temporal graph directly without dividing time information.

For the temporal graph/social network, some scholars have studied specific queries on it, such as single-node queries that ask for historical information of a vertex in a graph [35], reachability and matching query [36]. Algorithms for these queries are not applicable for processing keyword queries except that [37] studying the shortest path problem. There is another type of temporal graph, which is suitable for traffic network and communication network problems [38–41]. In this type of temporal graph, edges in the path must satisfy the time constraint. That is, the arrival time of the previous edge is no later than the start time of the next edge. Time constraint does not apply to our problem.

9 Conclusion

In this paper, we first propose the discrete timestamp algorithm and the approximation algorithm, which turn the keyword search problem on temporal social networks to the traditional keyword search problem on graph. Then we propose a more effective dynamic programming algorithm aiming to resolve the issue of keyword search on temporal social networks. In order to speed up the searching process, we adopt the two powerful pruning to greatly reduce the searching space. Lastly, we extend the scope of search questions, so as to get top-N efficient solutions. We conduct a series of experiments on the real temporal social networks. The results prove the efficiency and validity of our algorithm. In the future, we will further optimize the technologies on the dynamic changes in search conditions.

Author Contributions YG and ZC wrote the main manuscript text and prepared all the figures. All authors reviewed the manuscript.

Funding This paper is supported by the National Nature Science Foundation of China (61572537, U1501252).

Data Availability The data that support the findings of this study are openly available.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Ethical Approval This article does not contain any studies with human participants or animals performed by any of the authors. Results are gotten through simulation and tested number of times to take final value.

References

1. David K, Jon K, Amit K (2000) Connectivity and inference problems for temporal networks. In: STOC, pp 504–513
2. Guan-Yi J, Yi-Cheng C, Hung-Ming L (2021) Evolution pattern mining on dynamic social network. J Supercomput 77:6979–6991
3. Xiaoying C, Chong Z, Bin G, Weidong X (2017) Temporal query processing in social network. In: JIIS, pp 147–166
4. Liu Z, Wang C, Chen Y (2017) Keyword search on temporal graphs. TKDE 29(8):1667–1680
5. Jingwen S, Chaokun W, Changping W, Gaoyang G, Jun Q (2020) An attribute-based community search method with graph refining. J Supercomput 76:7777–7804
6. Youngho J, Hyunwoo L, Ayoung C, Mincheol W (2021) Web behavior analysis in social life logging. J Supercomput 77:1301–1320
7. Ding B, Yu JX, Wang S, Qin L, Zhang X, Lin X (2007) Finding top-k min-cost connected trees in databases. In: ICDE, pp 836–845
8. Ma S, Hu R, Wang L, Lin X, Huai J (2017) Fast computation of dense temporal subgraphs. In: ICDE, pp 361–372
9. Rizzolo F, Vaisman AA (2008) Temporal xml: modeling, indexing, and query processing. PVLDB 17(5):1179–1212
10. He H, Wang H, Yang J, Yu PS (2007) Blinks: ranked keyword searches on graphs. In: SIGMOD, pp 305–316
11. Kimelfeld B, Sagiv Y (2006) Finding and approximating top-k answers in keyword proximity search. In: SIGMOD, pp 173–182
12. Bhalotia G, Hulgeri A, Nakhe C, Chakrabarti S, Sudarshan S (2002) Keyword searching and browsing in databases using banks. In: ICDE, pp 431–440
13. Kacholia V, Pandit S, Chakrabarti S, Sudarshan S, Desai R, Karambelkar H (2005) Bidirectional expansion for keyword search on graph databases. In: VLDB, pp 505–516
14. Golenberg K, Kimelfeld B, Sagiv Y (2008) Keyword proximity search in complex data graphs. In: SIGMOD, pp 927–940
15. Dreyfus SE, Wagner RA (1971) The Steiner problem in graphs. Networks 1(3):195–207
16. Reich G, Widmayer P (1989) Beyond Steiner's problem: a VLSI oriented generalization. In: WG, pp 196–210
17. Li R-H, Qin L, Yu JX, Mao R (2016) Efficient and progressive group steiner tree search. In: SIGMOD, pp 91–106
18. Wentao H, Kaiwei L, Shimin C, Wenguang C (2019) Auxo: a temporal graph management system. BDMA 2(1):58–71
19. Jianye Y, Wu Y, Wenjie Z (2021) Keyword search on large graphs: a survey. DSE 6(2):142–162
20. Hristidis V, Papakonstantinou Y, Gravano L (2003) Efficient ir-style keyword search over relational databases. In: VLDB, pp 850–861
21. Luo Y, Lin X, Wang W, Zhou X (2007) Spark: top-k keyword query in relational databases. In: SIGMOD, pp 115–126
22. Sayyadian M, LeKhac H, Doan A, Gravano L (2007) Efficient keyword search across heterogeneous relational databases. In: ICDE, pp 346–355
23. Thirunarayan K, Immaneni T (2009) A coherent query language for XML. JIIS 32(2):139–162

24. Zhang L, Tran T, Rettinger A (2013) Probabilistic query rewriting for efficient and effective keyword search on graph data. PVLDB 6(14):1642–1653
25. Qin L, Yu JX, Chang L, Tao Y (2009) Querying communities in relational databases. In: ICDE, pp 724–735
26. Balmin A, Hristidis V, Papakonstantinou Y (2004) Objectrank: authority-based keyword search in databases. VLDB 4:564–575
27. Sun Y, Xiao X, Cui B, Halgamuge K, Lappas T, Luo J (2021) Finding group Steiner trees in graphs with both vertex and edge weights. PVLDB 7(14):1137–1149
28. Luyi B, Xinyi D, Bin Q (2022) Adaptive query relaxation and top-k result sorting of fuzzy spatiotemporal data based on XML. IJIS 3(37):2502–2520
29. Snodgrass R (1987) The temporal query language tquel. TODS 12(2):247–298
30. Jensen CS, Snodgrass RT, Soo MD (1995) The tsql2 data model. In: The TSQL2 temporal query language. Springer, pp 157–240
31. Jaewook B, Sungpil W, Daeyoung K (2020) hronoGraph: enabling temporal graph traversals for efficient information diffusion analysis over time. TKDE 32(3):424–437
32. Ariel D, Eliseo P, Matas P, Valeria S, Alejandro V (2021) A model and query language for temporal graph databases. JVLDB 30(5):825–858
33. Maria M, Zolt M, Philipe P Pierre M (2022) Clock-G: a temporal graph management system with space-efficient storage technique. ICDE, pp 2263–2276
34. Yishu W, Ye Y, Yuliang M, Guoren W (2019) Time-dependent graphs: definitions, applications, and algorithms. DSE 4(4):352–366
35. Koloniari G, Souravlias D, Pitoura E (2013) On graph deltas for historical queries. arXiv preprint arXiv:1302.5549
36. Fard A, Abdolrashidi A, Ramaswamy L, Miller JA (2012) Towards efficient query processing on massive time-evolving graphs. In: CollaborateCom, pp 567–574
37. Huo W, Tsotras VJ (2014) Efficient temporal shortest path queries on evolving social graphs. In: SSDBM, pp 1–4
38. Wu H, Cheng J, Huang S, Ke Y, Lu Y, Xu Y (2014) Path problems in temporal graphs. PVLDB 7(9):721–732
39. Rozenshtein P, Gionis A, Prakash BA, Vreeken J (2016) Reconstructing an epidemic over time. In: KDD, pp 1835–1844
40. Xiao H, Rozenshtein P, Tatti N, Gionis A (2018) Reconstructing a cascade from temporal observations. In: SDM, pp 666–674
41. Lei L, Kai Z, Sibo W, Wen H, Xiaofang Z (2018) Go slow to go fast: minimal on-road time route scheduling with parking facilities using historical trajectory. JVLDB 27:321–345