Project1: Historical Connectivity Query

Sun Yuxiang

April 2025

1 Bidirectional BFS(Final Solution)

1.1 Algorithm Idea

The Bidirectional BFS algorithm is an efficient approach for checking connectivity between two vertices, u and v, within a given time window $[t_1, t_2]$ in a temporal graph. Instead of performing a BFS starting from just the source node, the algorithm performs two BFS searches, one from u and one from v, simultaneously. These searches proceed until they meet in the middle, reducing the search space significantly.

In the context of historical connectivity, we need to check whether there is a path between u and v that exists at any point within the specified time interval. The bidirectional BFS helps minimize the number of edges explored by dividing the search space.

The pseudocode for Bidirectional BFS is as follows:

Algorithm 1 Online-Bidirectional-BFS

Require: Temporal graph G, vertices u, v, time interval $[t_1, t_2]$
Ensure: Whether u and v are reachable within $[t_1, t_2]$

```
if u = v then
    return true
end if
Initialize two queues Q_u ← {u}, Q_v ← {v}
Mark u and v as visited from their respective sides
while Q_u and Q_v are not empty do
    Expand one layer from Q_u
    if any node is visited from v-side then
        return true
    end if
    Expand one layer from Q_v
    if any node is visited from u-side then
        return true
    end if
end while
return false
```

1.2 Algorithm Analysis

The time complexity of Bidirectional BFS is $O(m + n)$, where m is the number of edges and n is the number of vertices in the graph. The space complexity is $O(n + m)$ as the algorithm stores the visited vertices and the queues for both directions. Bidirectional BFS is suitable for conducting a small number of queries on large graphs. On the one hand, it significantly reduces exploration space and time compared to traditional BFS, on the other hand, it does not incur additional memory or unnecessary data structure construction, which will save lots of time. Therefore, we choose it as the final solution to this task.

2 TILL-Index based Span-Reach

2.1 Solution Overview

Wen et al. [1] propose an index-based method, TILL-Index, to efficiently answer span-reachability queries in large temporal graphs. The core idea is to adopt a two-hop labeling strategy: for each

vertex u, a label set is constructed containing triplets $\langle w, t_s, t_e \rangle$ indicating that u can reach (or be reached by) w in the projected graph over time interval $[t_s, t_e]$. A query is answered by checking whether there exists an intermediate vertex w such that u and v are both span-reachable to/from w in the given interval. Compared with online search methods, this index enables fast query processing with precomputed label information.

In this work, we adapt their method to **undirected temporal graphs**, where each vertex maintains only one label set.

2.2 TILL-Index Construction

(1) Index Structure. For each vertex u, we store a label set $L(u)$ representing the span-reachability between u and other vertices. Instead of storing raw triplets $\langle w, t_s, t_e \rangle$, we follow the compressed structure shown in Fig.3 of [1]:

- $L(u)$.`first` stores the list of reachable vertices from u, along with the start position of their intervals in $L(u)$.`second`.
- $L(u)$.`second` is a flat array storing all intervals $[t_s, t_e]$ in chronological order.

This structure allows efficient query processing by supporting fast two-level lookup: first locating a vertex w in the label, and then scanning the corresponding intervals. Intervals for each vertex are sorted by their start and end timestamps, which enables binary search.
(2) Construction Idea. Constructing the TILL-Index involves computing all canonical reachability tuples (CRTs). The algorithm proceeds by a fixed vertex order. For each vertex u, we compute all skyline reachability tuples (SRTs) from u using a priority queue, prune dominated intervals, and check whether a common vertex of higher rank already covers them. For each valid CRT $\langle u, v, t_s, t_e \rangle$, we add the interval $[t_s, t_e]$ to $L(v)$.`second` and update $L(v)$.`first` accordingly.
(3) Pseudocode. The simplified algorithm for undirected graphs is as follows:

Algorithm 2 TILL-Construct* (Undirected Version)

Require: Temporal undirected graph $G(V, E)$, vertex order O
Ensure: Label set $L(u)$ for each vertex u

for all $u \in V$ in order O **do**
  $Q \leftarrow \{\langle u, +\infty, -\infty \rangle\}$
  while Q not empty **do**
    Pop $\langle v, t_s, t_e \rangle$ with smallest interval
    if $u \neq v$ and u already covers v in $[t_s, t_e]$ **then**
      continue
    else
      Add $[t_s, t_e]$ to $L(v)$.`second`; update $L(v)$.`first` for u
    end if
    for all (v', t) such that $(v, v', t) \in E$ and $O(v') > O(u)$ **do**
      $t'_s \leftarrow \min(t_s, t)$, $t'_e \leftarrow \max(t_e, t)$
      if $t'_e - t'_s + 1 \leq \vartheta$, push $\langle v', t'_s, t'_e \rangle$ into Q
    end for
  end while
end for

2.3 TILL-Index based Span-Reach

(1) Query Logic. Given a query $(u, v, [t_1, t_2])$, we check if there exists a common intermediate vertex w such that u is connected to w in some $[t_s, t_e] \subseteq [t_1, t_2]$ and v is also connected to w in some $[t'_s, t'_e] \subseteq [t_1, t_2]$.

(2) Implementation. Since all label sets are sorted by degree and each vertex only maintains one label set, we use a two-pointer merge-sort-like strategy to find common vertices w in $L(u)$ and $L(v)$. For each common w, we check if there exists a pair of intervals from $L(u)$ and $L(v)$ that both fall within the query interval.

(3) Pseudocode. The query algorithm is adapted from Algorithm 4 in the original paper:

Algorithm 3 Span-Reach (Undirected, Final Structure with Fast Pruning)

Require: TILL-Index L, temporal graph G, query vertices u, v, interval $[t_1, t_2]$
Ensure: Whether u span-reaches v in $[t_1, t_2]$

if $u = v$ **then**
  return true
end if
if $L(u).\texttt{first} = \emptyset$ and $L(v).\texttt{first} = \emptyset$ **then**
  return false
end if
▷ Prune using Lemma 9 and 10
$u_{t_{\min}} \leftarrow +\infty$, $u_{t_{\max}} \leftarrow -\infty$
$v_{t_{\min}} \leftarrow +\infty$, $v_{t_{\max}} \leftarrow -\infty$
for all $(w, t) \in G.\texttt{adj}[u]$ **do**
  update $u_{t_{\min}}, u_{t_{\max}}$
end for
for all $(w, t) \in G.\texttt{adj}[v]$ **do**
  update $v_{t_{\min}}, v_{t_{\max}}$
end for
if $u_{t_{\min}} > t_2$ or $u_{t_{\max}} < t_1$ or $v_{t_{\min}} > t_2$ or $v_{t_{\max}} < t_1$ **then**
  return false
end if
▷ Proceed with merge-sort-like strategy
$i \leftarrow 1$, $j \leftarrow 1$
$L_u \leftarrow L(u).\texttt{first}$, $I_u \leftarrow L(u).\texttt{second}$
$L_v \leftarrow L(v).\texttt{first}$, $I_v \leftarrow L(v).\texttt{second}$
while $i \le |L_u|$ and $j \le |L_v|$ **do**
  $w_u, p_u \leftarrow L_u[i]$, $w_v, p_v \leftarrow L_v[j]$
  if $w_u = w_v$ **then**
    $p_u^{\text{end}} \leftarrow$ next start pos or $|I_u|$, $p_v^{\text{end}} \leftarrow$ next start pos or $|I_v|$
    for $k = p_u$ to $p_u^{\text{end}} - 1$ **do**
      for $l = p_v$ to $p_v^{\text{end}} - 1$ **do**
        if $I_u[k] \subseteq [t_1, t_2]$ and $I_v[l] \subseteq [t_1, t_2]$ **then**
          return true
        end if
      end for
    end for
    $i \leftarrow i + 1$, $j \leftarrow j + 1$
  else if $w_u < w_v$ **then**
    $i \leftarrow i + 1$
  else
    $j \leftarrow j + 1$
  end if
end while
return false

2.4 Algorithm Analysis

The construction time of TILL-Construct* is bounded by $O(ndc_{\le\vartheta}(\log(dc_{\le\vartheta}) + c_{\le\vartheta}))$, where d is the maximum degree and $c_{\le\vartheta}$ is the maximum number of canonical reachability tuples per vertex within interval length ϑ. In practice, it alwagys costs lots of time.

Query processing time is bounded by $O(|L(u)| + |L(v)|)$ due to the merge-sort-like scan. Compared with online search, this approach greatly reduces runtime, especially on large graphs and frequent queries.

References

[1] D. Wen, Y. Huang, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "Efficiently answering span-reachability queries in large temporal graphs," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, IEEE, 2020, pp. 1153–

1164. DOI: 10.1109/ICDE48307.2020.00104. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00104.