On Temporal Bipartite Graphs and Their Application in Disease Spread Prediction

by

Ruilin Su

A thesis submitted to the Faculty of Graduate Studies in partial fulfillment of the requirements for the Master of Science degree.

Department of Applied Computer Science

Master of Science in Applied Computer Science and Society

The University of Winnipeg

Winnipeg, Manitoba, Canada

April 2024

ABSTRACT

The original *temporal bipartite graph* is flawed in the context of disease spreading models as it does not account for concepts such as virus incubation and recovery periods. In this thesis, a new graph structure, referred to as the *improved temporal bipartite graph* is introduced with these two concepts incorporated to enhance accuracy in predicting disease spreading. To facilitate arbitrary reachability queries, another concept, the *transmission graph*, is introduced. It is derived from a temporal bipartite graph based on a series of reachability query evaluation. We distinguish between two types: *single-path transmission graph* and *multi-path transmission graph*. Based on them, four algorithms are proposed for evaluating reachability queries on a temporal bipartite graph, with a label-based technique used to achieve high efficiency. Both single-path transmission graphs and multi-path transmission graphs are in fact a kind of extension of the reachability query evaluation. By establishing indexes over them, the reachability query evaluation for disease spreading prediction can be very efficiently conducted.

Keywords: Temporal Bipartite Graph, Disease Spreading Model, Reachability Queries.

ACKNOWLEDGEMENT

I would like to express my heartfelt gratitude to Dr. Yangjun Chen for his invaluable guidance as my thesis advisor. Dr. Chen not only served as my mentor throughout this research but also taught the Advanced Data Structure and Algorithms course, which played a significant role in shaping this study. His guidance and insights are very helpful to me for the completion of this work.

I would also like to extend my appreciation to the other professors who taught me various aspects of computer science during my graduate studies, namely Dr. Sheela Ramanna, Dr. Christopher Henry, and Dr. Simon Liao. Their expertise and teachings have broadened my knowledge and enriched my academic journey.

A special note of thanks goes to my parents for their unwavering support and for providing me with the means to pursue my education abroad. Their sacrifices and encouragement have been instrumental in my academic achievements.

Lastly, I want to express my deepest gratitude to my wife for her unwavering support, understanding, and companionship throughout this academic endeavor. Her presence has been a constant source of motivation and strength.

I am truly grateful to all these individuals and their help for my academic and personal growth.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Bipartite graphs are used to model relationships between two different types of entities. Some examples include the people-location network [2], author-paper network [3, 5], and user-item network [4]. When edges in a bipartite graph are assigned timestamps, it becomes a Temporal Bipartite Graph [6]. As shown in Figure 1, it is a people-location network derived from the article [2], where each edge is associated with two timestamps, representing an individual's arrival and departure times at a location.

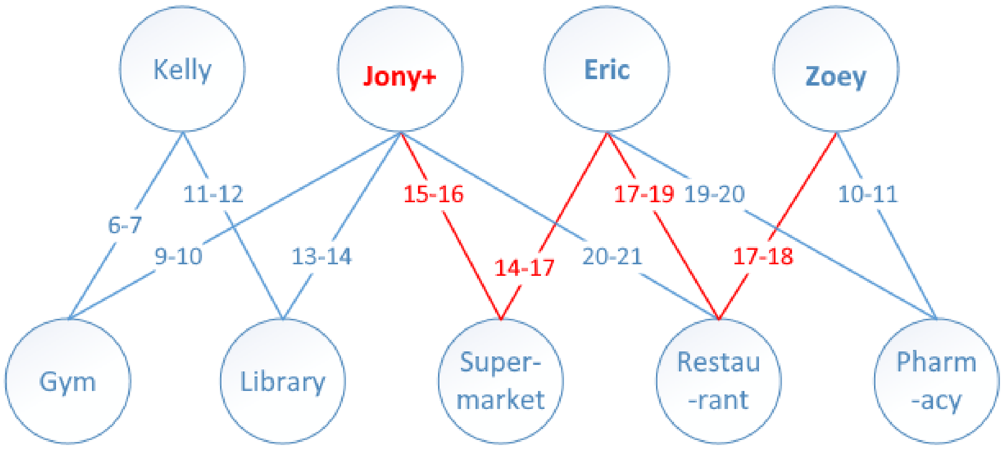


Figure 1 A people-location network for simulating disease outbreaks

This model simulates the movement of people between various locations in the real world, making it highly suitable for tracking disease spread during outbreaks [2]. This model records when individuals simultaneously visit the same location, and it is precisely during such simultaneous visits that disease transmission may occur. For instance, the carrier Jony visits the supermarket between 15 and 16, while Eric visits there between 14 and 17, then Eric may be transmitted. Another significant application scenario for Temporal Bipartite Graphs is in tracking cell metabolism pathways [1, 7, 8, 9].

The primary challenge in utilizing Temporal Bipartite Graphs to track and predict disease spread lies in performing reachability queries on these graphs, which is more complex than traditional bipartite graphs due to the consideration of timestamps on edges. Xiaoshuang Chen et al. first implemented reachability queries on Temporal Bipartite Graphs [1], and they also explored how to use these graphs to track and predict disease spread. However, we observed that real-world infections typically have an incubation period, during which infected individuals generally cannot transmit the disease to others [10]. Furthermore, diseases can be cured, and recovered individuals lose their ability to transmit the disease. Consequently, it becomes evident that individuals in the infectious state can only transmit the disease to others after a certain incubation period, or in other words, they are reachable by other individuals during this time. This aspect was not considered in Xiaoshuang Chen's research.

Therefore, we introduced two timestamps for human vertices to represent the time of infection and the time of recovery. With this improvement, the accuracy of tracking and prediction will be enhanced.

Regarding the collection of an individual's arrival and departure times at a location, some countries have provided solutions. For instance, in China, individuals are required to scan a QR code for registration when visiting public places during the outbreak of COVID-19 [11]. As for the incubation period of the pathogen and the time it takes to be cleared from the human body, these can be referenced from research findings specific to the corresponding disease [10].

To accelerate the performance of random reachability queries, we introduced the concept of a Transmission Graph. Initially, a comprehensive reachability query is conducted on the original Temporal Bipartite Graph, and based on each query result, the Transmission Graph is gradually generated while also excluding the vertices representing locations. This approach makes it appear as if the Transmission Graph is a simplification of the original Temporal Bipartite Graph, and it provides a clear representation of the transmission chains. Reachability queries can be directly performed on the Transmission Graph, which is faster than conducting reachability queries directly on the original Temporal Bipartite Graph.

In Chapter 2, we will provide a detailed introduction to the improved Temporal Bipartite Graph and the Transmission Graph. Chapter 3 will delve into comprehensive descriptions of the reachability query algorithms and the algorithms for generating the Transmission Graph. The algorithm results and performance will be presented and discussed in Chapter 4.

2 Preliminary

Temporal Bipartite Graph

The temporal bipartite graph $G(V, E)$ is described as an undirected graph, with its vertices $V(G)$ divided into two groups: $V(G) = U(G) \cup L(G)$. $U(G)$ resides on the upper layer, while $L(G)$ resides on the lower layer. In $G(V, E)$, we have

$$U(G) \cap L(G) = \emptyset, \text{and } U(G) \cup L(G) = V(G) \qquad (1)$$

$$E(G) \subseteq U(G) \times L(G) \qquad (2)$$

where *E(G)* denotes the set of temporal edges of the form (u, v, t_s, t_e) or (v, u, t_s, t_e), representing an edge connecting two vertices *u* and *v* respectively from the two different layers in a time span $t_s \sim t_e$.

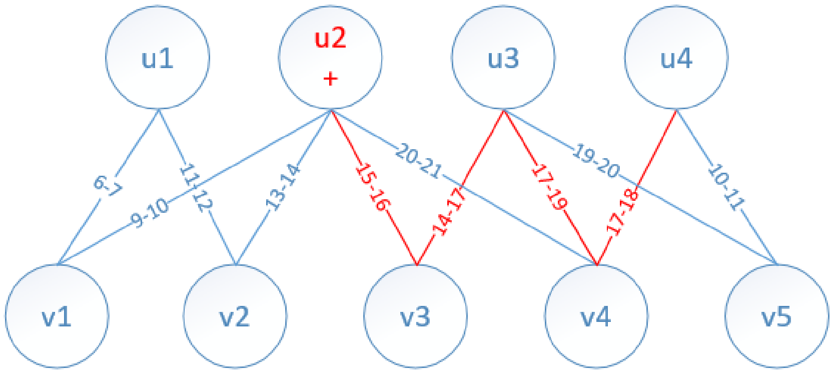


Figure 2 A temporal bipartite graph modeled from figure 1

As an example, consider the graph $G(V, E)$ shown in Figure 2. This is a typical temporal bipartite graph with $U(G) = \{u_1, u_2, u_3, u_4\}$and $L(G) =$

$\{v_1, v_2, v_3, v_4, v_5\}$. Each edge connecting a vertex from $U(G)$ to a vertex from $L(G)$ is referred to as a temporal edge. For example, the left most edge from u_1 to v_1 is labeled with a time span 6 - 7, and represented by $(u_1, v_1, 6, 7)$.

By a *wedge*, we mean a path consisting of three vertices *u*, *v*, *w*, and two connecting temporal edges (u, v, t_{s1}, t_{e1}), $(v\ , w, t_{s2}, t_{e2})$. In Figure 2, a path starting from u_1, going along $(u_1, v_1, 6, 7)$ and reaching v_1, then going along $(v_1, u_2, 9, 10)$ and reaching u_2 is a wedge, denoted as $W = ((u_1, v_1, 6, 7), (v_1, u_2, 9, 10))$. Next, by a *time-overlapping wedge*, we mean two connected temporal edges with overlapped time spans. For example, the edge $((u_2, v_3, 15, 16), (v_3, u_3, 14, 17))$ shown in Figure 2 is a time-overlapping wedge, and $((u_3, v_4, 17, 19), (v_4, u_4, 17, 18))$ is another. Assume that u_2 is a virus carrier. Then, a series of time-overlapping wedges will cause a spread of disease: u_3 will be infected by u_2 in location v_3, and u_4 will be infected by u_3 in v_4.

By a *time-respecting path* we mean a series of consecutive time-overlapping wedges $P = (W_1, W_2, \ldots, W_n)$ such that the very ending time of W_i is not later than the beginning time of W_{i+1} ($i = 1, \ldots, n - 1$). For instance, the three red wedges in Figure 2 make up a time-respecting path.

The time-respecting path is used to track the disease transmission, based on which the following two notations are defined:

$u \rightsquigarrow_{\mathrm{I}} v$ - denotes a single-pair reachability, representing that *u* can reach *v* via a time-respecting path W within a time span I, where *u* and *v* are two vertices on a same layer in *G*; and

$u \rightsquigarrow_{\mathrm{I}} \{u_1, u_2, \ldots, u_k\}$ - denotes a single-source reachability, representing that *u* can reach $u_1, u_2, \ldots, u_k$ via some time-respecting paths within I, where *u* and all $u_1, u_2, \ldots, u_k$ are on a same layer in *G*.

For example, $u_2 \rightsquigarrow_{(14,19)} u_4$ in Figure 3 is a single-pair reachability while $u_2 \rightsquigarrow_{(14,19)} \{u_3, u_4\}$ is a single-source reachability.

In addition, an earliest-arrival path is a time-respecting path from *u* to *v*, (on a same layer in *G*), which has the earliest ending time among all time-respecting paths.

For example, in Figure 2, the path from u_2 to u_4: $P = (W_1, W_2)$ is an earliest-arrival path, where $W_1 = ((u_2, v_3, 15, 16), (v_3, u_3, 14, 17))$and $W_2 = ((\mathrm{u}_3, \mathrm{v}_4, 17, 19), (\mathrm{v}_4, \mathrm{u}_4, 17, 18))$.

Figure 3 is similar to Figure 2, but added with a new edge from u_2 to v_5 with time span 9-11, in which another earliest-arrival path from u_2 to u_4 can be found: $P = (W_1)$, where $W_1 = ((u_2, v_5, 9, 11), (v_5, u_4, 10, 11))$.

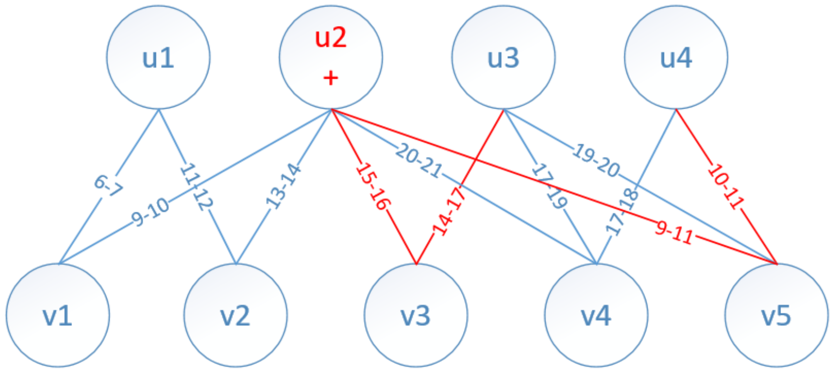


Figure 3 Earliest-arrival path

All the above concepts were first introduced in [1]. However, using such a temporal bipartite graph for monitoring a disease spread, such as COVID-19, can be problematic and very inaccurate since two important factors are not taken into consideration. First, a newly infected carrier becomes capable of transmitting the disease to others only when an incubation period has passed. Secondly, carriers can eventually recover and lose their ability to transmit the disease, or become non-infected themselves. For these reasons, we have also assigned two timestamps to U(G), which are used to indicate the infection time t_s and recovery time t_e of the vertex. An upper-layer vertex is represented in the following format: $u = (t_s, t_e, E(u))$ where *E(u)* denotes the set of edges associated with u.

Figure 4 is a modification of the transmission model shown in Figure 3, by which both incubation and recovery are considered. Assuming that u_2 is carrying the virus from the time 0, but recovered from the disease and loses the ability to infect others from the time 72, that is $u_2 = (0, 72, E(u_2))$. Hence, although there is a time-overlapping wedge $W = ((u_1, v_1, 80, 82), (v_1, u_2, 80, 82))$, connecting u_1 and u_2, u_1 will not be infected as u_2 is already cured from the time 72.

On the other hand, u_2 transmits the disease to u_3 at v_3 in the time interval 14 - 17, but u_3 becomes infectious only after the time 63. Therefore, u_4 is, rather than at v_4 in 17 - 19, infected at v_5 in 65 - 67.

Clearly, our refined approach greatly increases the accuracy of tracking carriers when compared to the original method [1].

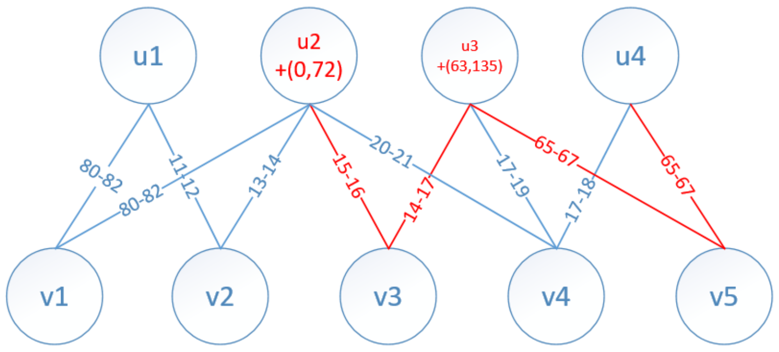


Figure 4 Transmission model of a disease

In our implementation of the above method using C++ language, a class named "Graph" is defined to accommodate all the temporal bipartite graphs. This Graph class has various attributes, such as the sizes of the upper and lower layers, the incubation period, and recovery time. The vertices are stored in a list, but divided into two distinct parts: the upper layer vertices occupy positions from 0 to the size of the upper layer minus one while the lower layer vertices are stored from the location size of the upper layer to the size of the lower layer minus one. In addition, the vertices from different layers are further partitioned into separate groups.

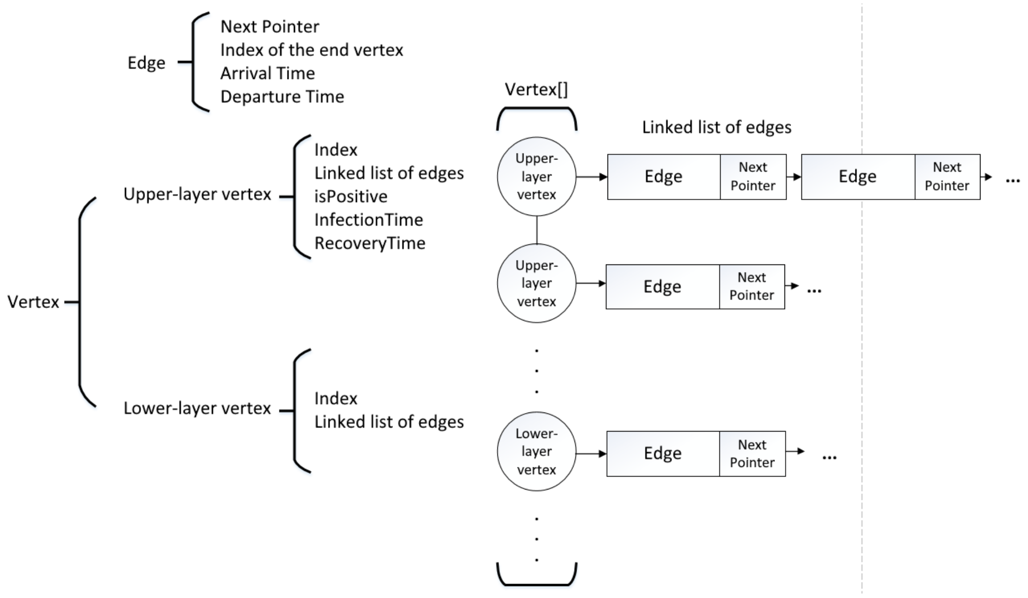


Figure 5 Structure of vertices and edges

Figure 5 provides an overview of the vertex and edge structure. Here, "Upper Vertex" and "Lower Vertex" are two subclasses of the "Vertex" class. The "Upper Vertex" subclass encompasses various attributes to represent its infection status and period, whereas the "Lower Vertex" subclass does not since it is only used to represent different locations. Both "Upper Vertex" and "Lower Vertex" subclasses possess an index denoting their position in the list and an adjacency list to store pointers to edges. The edges of upper vertices maintain information about the index of lower vertices, arrival time, and departure time while the edges of lower vertices store details about the index of upper vertices, arrival time, and departure time.

Transmission Graph

A transmission graph is derived from a temporal bipartite graph, which serves to show the relationship of infection within a disease transmission scenario, offering a simplified representation of disease infection. In contrast to the original temporal bipartite graph, the transmission graph excludes all lower-layer vertices and is structured as a directed graph, providing a clear depiction of which vertex infects. In our implementation, a transmission graph is stored in a separate file so that the monitoring of disease outbreaks can be done by searching this separate file alone, rather than searching the original temporal bipartite graph every time. This clearly has a better performance.

Two kinds of transmission graphs are introduced in our study: single-path transmission graph and multi-path transmission graph.

Definition 1 (multi-path transmission graph) Let $G(V, E)$ be a temporary bipartite graph with bipartite $V = U \cup L$. The multi-path transmission graph G' of G is an induced graph from G by changing each wedge $W = (u, v, t_{s1}, t_{e1}), (v, w, t_{s2}, t_{e2})$ to an edge (u, v, t_s, t_e), where $[t_s, t_e] = [t_{s1}, t_{e1}] \cap [t_{s2}, t_{e2}]$.

The multi-path transmission graph stores all possibilities of transmissions.

Definition 2 (single-path transmission graph) Let G' be an induced graph from a temporary bipartite graph G. A single-path transmission graph of G is a forest created by a depth first search of G', with $L(G)$ being removed. Searched paths are time-respecting and earliest-arrival paths.

Intuitively, a single-path transmission graph is like a set of multi-way trees, in which the in-degree of each vertex is 0 or 1, and there is no limit to the out-degree.

In transmission graphs, all paths are time-respecting and earliest-arrival paths. Therefore, we have the following lemma.

Lemma 1 All successors $S(P)$ of a predecessor P are *single-source reachable* from this predecessor. That is, $P \rightsquigarrow_{[t_{sP},t_{eS}]} \{S(P)\}$, where t_{sP} is the earliest starting time of edges associated with P, and t_{eS} is the latest ending time of edges associated with $S(P)$.

Figure 4 is further changed, in order to clearly explain what a transmission graph can be derived, as illustrated in Figure 6, in which u_0 and u_5 are added. u_0 and u_2 are initially set as carriers, and their carrying time spans are (24, 96) and (0, 84), respectively. u_1 is infected by u_0 at v_1 in the time span 80 - 82, even though another carrier u_2 also visited v_1 in the time span 81 -82. u_0 arrived and contacted with u_1 earlier than u_2. Hence u_2 has lower possibility to transmit the disease to u_1. u_2 transmits the disease to u_3, and then both u_4 and u_5 can be infected by u_2 or u_3 since all of them (u_2, u_3, u_4, u_5) have visited v_5 in the time span 65 - 67. Therefore, there are 2 possibilities of infection. This shows the difference between single-path transmission graphs and multi-path transmission graphs.

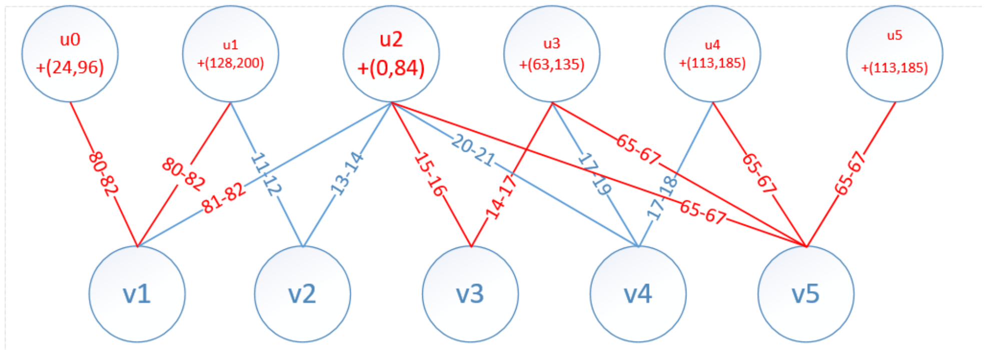


Figure 6 Changed transmission model

As mentioned above, a single-path transmission graph is like a set of multi-way trees, as illustrated in Figure 7 and 8, in which we show three possible single-path transmission graphs generated from figure 6. They clearly show how the disease is spreading, where the carrying time span is displayed within the circle representing each vertex. In addition, the time span on edges show the direction and disease spreading time. From the figures, we can see that u_4 and u_5 can be infected by both u_2 or u_3. Hence, there are 3 possible single-path transmission graphs.

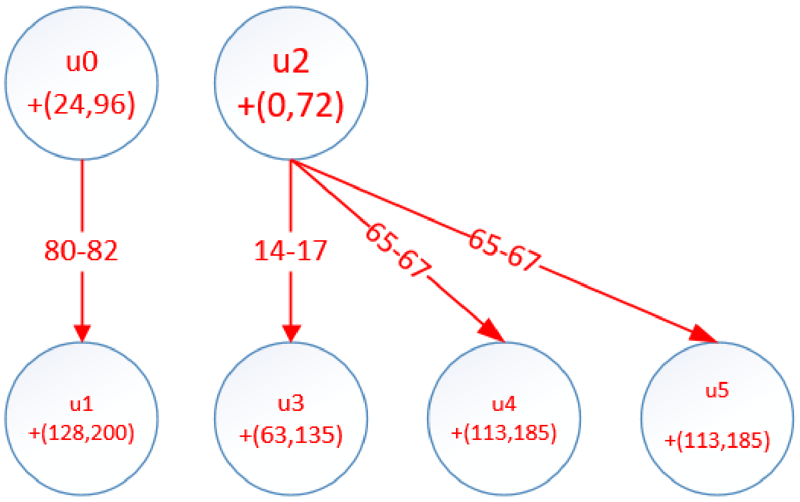


Figure 7 An example of single-path transmission graph, generated from the model in figure 6

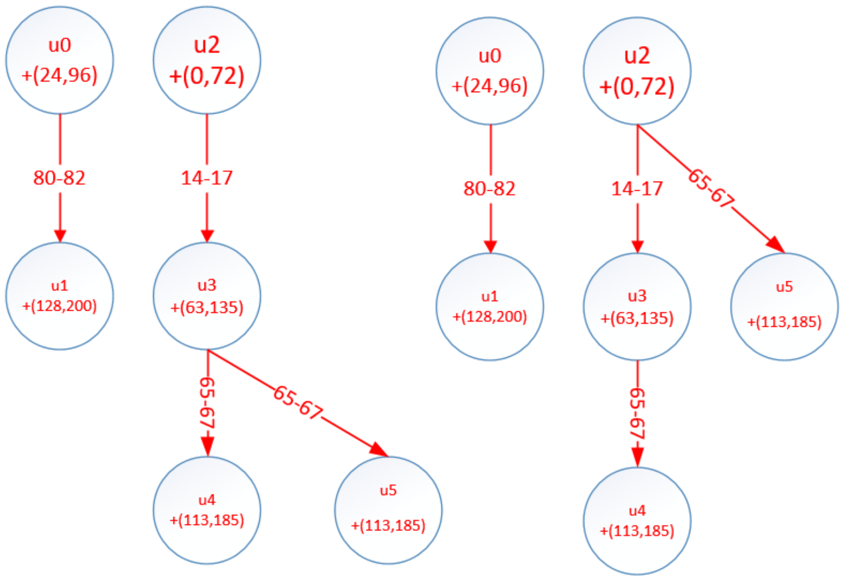


Figure 8 Other possible single-path transmission graphs generated from the model in figure 6

All the possible spreading paths can be stored in one multi-path transmission graph, as shown in Figure 9. However, a multi-path transmission graph is more complicated than a single-path transmission graph. Thus, more time are needed to evaluate a reachability query. Note that although u_2 arrives v_1 later than u_0, u_0 is more likely to spread the disease to u_1 than to u_2 in our solution. It is because our algorithm (for generating multi-path transmission graphs) considers all carriers accessing a certain location in an overlapping time span as the sources of infection. Hence u_2 is connected with u_1 in Figure 9 while for single-path transmission graph generation only the carrier that has the earliest contact time will be considered the source of infection, unless this carrier cannot be determined (when all carriers arrive at the same time).

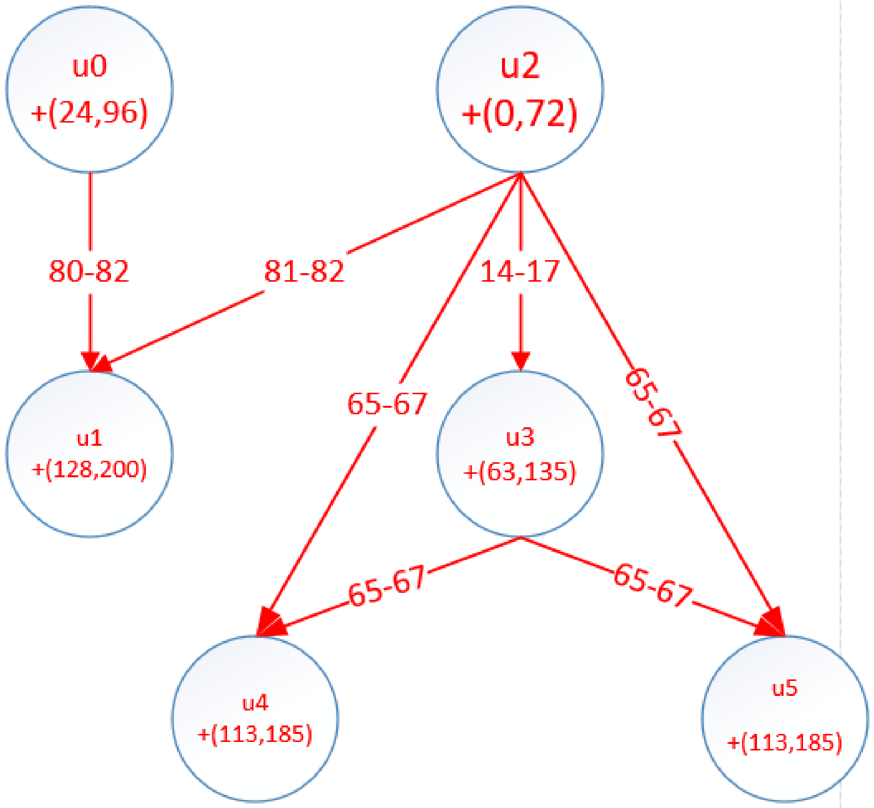


Figure 9 An example of multi-path transmission graph generated from the model in figure 6

In our implementation, the adjacency list representation method is used to represent both single-path transmission graphs and multi-path transmission graphs, as shown in Figure 10. They have the same structure as regular graphs.

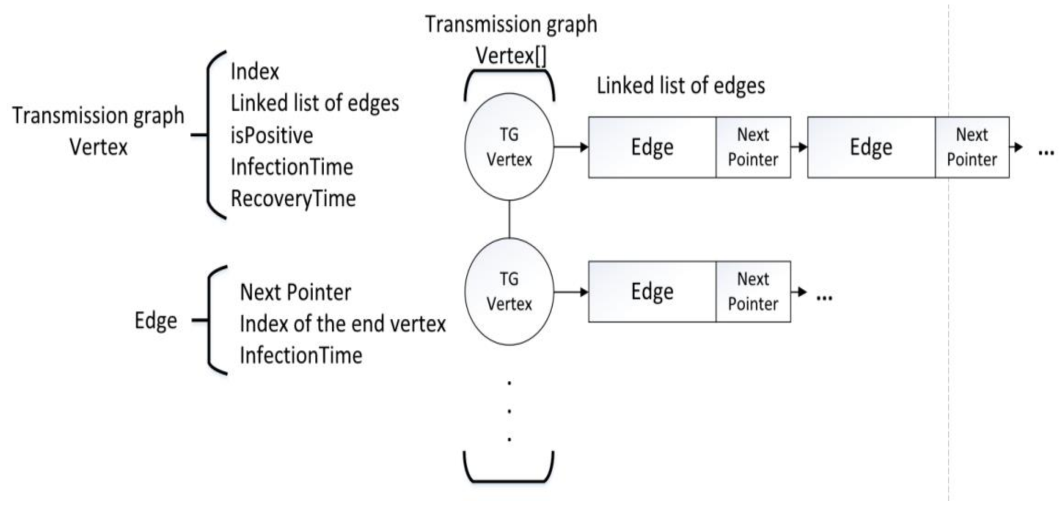


Figure 10 Structure of a single-path transmission graph or a multi-path transmission graph

Problem Statement

Given a temporal bipartite graph $G(V = (U, L), E)$, we will search all reachable pairs that can be listed in $U(G)$, update the carrying time spans and status of all new carrier, repeat this procedure until there is no status change of any carrier, and eventually generate a single-path transmission graph and a multi-path transmission graph. Then, answer any reachability query, by which we ask whether a vertex v is reachable from another vertex u through a path in a transmission graph.

In the following, we will first discuss several algorithms for answering reachability queries and updating the status and time spans of carriers. Then, the algorithm for generating transmission graphs. The experiment results will be shown and discussed in Chapter 5.

3 SOLUTION OVERVIEW

In this chapter, algorithms of predicting disease spread and generating transmission graphs are introduced, the table below paraphrases some variants in the pseudo code of proposed algorithms.

Table 1 Definitions of some variables in the pseudo code

Variable Name	Definition
flag	A boolean flag, used to detect whether any value is changed or not in one iteration.
UpperG/LowerG	$U(G)$ or $L(G)$, the set of all upper-layer vertices or lower-layer vertices.
isPositive	The infection status of an upper-layer vertex, the values can be NEGATIVE, POSITIVE, INCUBATING and IMMUNE.
edges	The set of edges associated with a vertex.
VisitTime	Two timestamps t_s, t_e of an edge $\mathrm{E} = (u\ , v, t_s, t_e)$, denoting the arrival time and departure time.
CarryTime	Two timestamps t_s, t_e of an upper-layer vertex $u = (t_s, t_e, E(u))$, denoting the infection time and re-

	covery time.
CurrentTime	The recorded time when the "isPositive" of an upper-layer vertex is changed, used to update "isPositive" in some algorithms below.
WaitTime	The incubation time of the pathogen.

Breadth-First-Search-Based Algorithm (BF)

The **B**readth-**F**irst-Search-Based algorithm is a naive method to search all reachable pairs, called **Algorithm 1**, by which all the single-source reachability of every positive upper-layer vertex is searched.

This algorithm works as follows.

It starts from the first "POSITIVE" or "INCUBATING" vertex on the upper layer. Then, for all lower-layer nodes incident on this vertex, all the other upper-layer nodes reachable from them through an edge will be explored (see lines 3 - 4). These two upper-layer vertices and one lower-layer vertex form a wedge (see lines 5 - 13). If edges of these wedges are time-overlapping and the status of the other vertex is "NEGATIVE" (see lines 14 - 15), changes the status of this vertex to "INCUBATING" and updates the carrying time span (see lines 16 -26). Here, the carrying starting time is the earliest contact time plus incubation time, and the ending time is the starting time plus the recovery time). In a next step, we will search a next "POSITIVE" or "INCUBATING" vertex, and repeat the above procedure until no "POSITIVE" vertex can be found (see lines 3 - 4).

The current time will be recorded, which is the time of changing the status of the last non-carrier vertex in one iteration (see line 16).

In the last step of this iteration, the states of all vertices based on the current time will be updated. If the current time falls into the carrying time span of a vertex, the state of this vertex is set to "POSITIVE". If the current time is after the ending time, the state of the vertex is set to "IMMUNE" (see lines 34 - 36). Repeat the whole process, until there is no state changing of any vertex (see line 1).

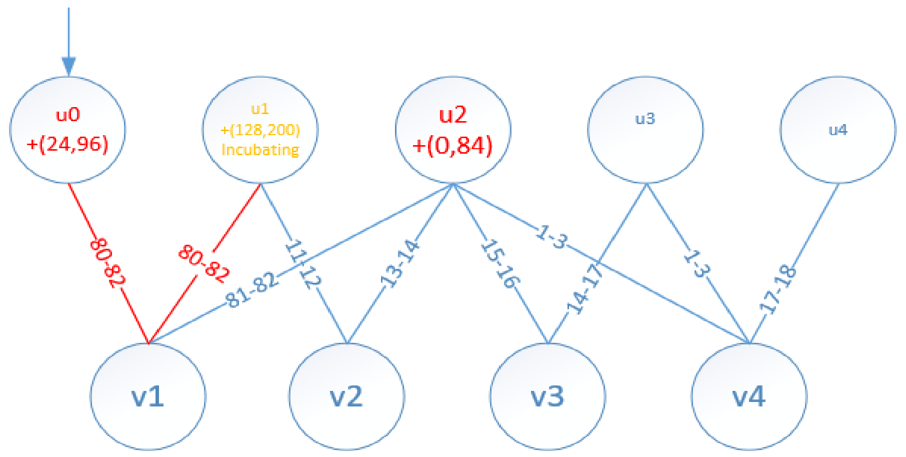
ALGORITHM 1: NAIVE BFS-BASED ALGORITHM

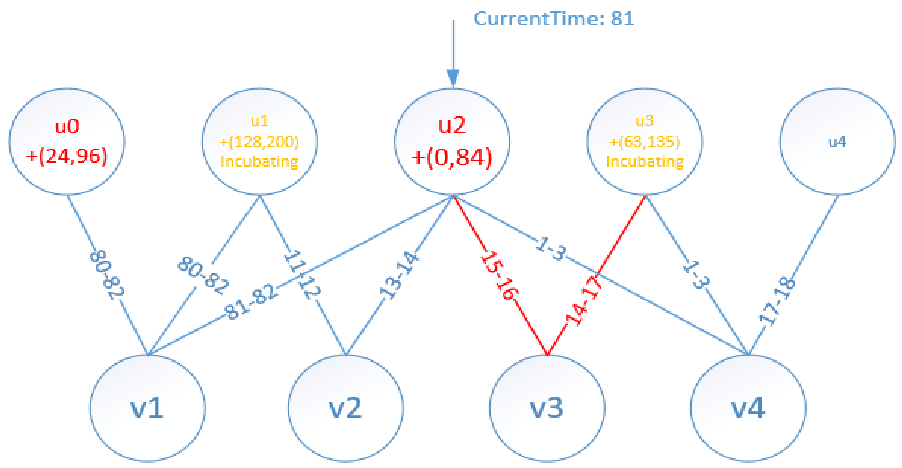
Input: All upper vertices UpperG and all lower vertices LowerG in a temporal bipartite graph G

```
1  while flag
2    flag ← false
3    foreach v in UpperG do //UpperG - U(G)
4      if v.isPositive is not NEGATIVE or IMMUNE then
5        foreach e in v.edges do //v.edges - all edges incident to v
6          if v.CarryTime is overlapped with e.VisitTime then
7            //v.CarryTime - [$v.t_s, v.t_e$]
8            //e.VisitTime - [$t_s, t_e$] of the edge e [v, lv, $t_s, t_e$]
9            LowerVertex lv ← GetVertexById(e.id)
10           //GetVertexById(e.id) - a function to the vertex
11           //corresponding to the id. Here the id is stored in the edge e
12             foreach le in lv.edges do
13               UpperVertex uv ← GetVertexById(le.id)
14               if e.VisitTime is overlapped with le.VisitTime
15                  && uv.isPositive is NEGATIVE then
16                CurrentTime ← e.VisitTime($t_e$)
17                //When it is bigger than CurrentTime, set
18                // it to the ending time of e
19                uv.CarryTime ← VisitTime + WaitTime
20                //VisitTime is picked from 4 times in
21                //e.VisitTime and le.VisitTime, which is the
22                //possible earliest contact time
23                //WaitTime is IncubationTime and RecoveryTime
24                //Here, add $t_s$ with IncubationTime as $uv.t_s$,
25                //then add $uv.t_s$ with RecoveryTime as $uv.t_e$
26                uv.isPositive ← INCUBATING
27                flag ← true
28               end if
29             end foreach
```

```
      end if
    end foreach
  end if
end foreach
foreach v in UpperG do
  Set v.isPositive based on CurrentTime and v.CarryTime
end foreach
end while
```

isPositive: NEGATIVE/POSITIVE/INCUBATING/IMMUNE





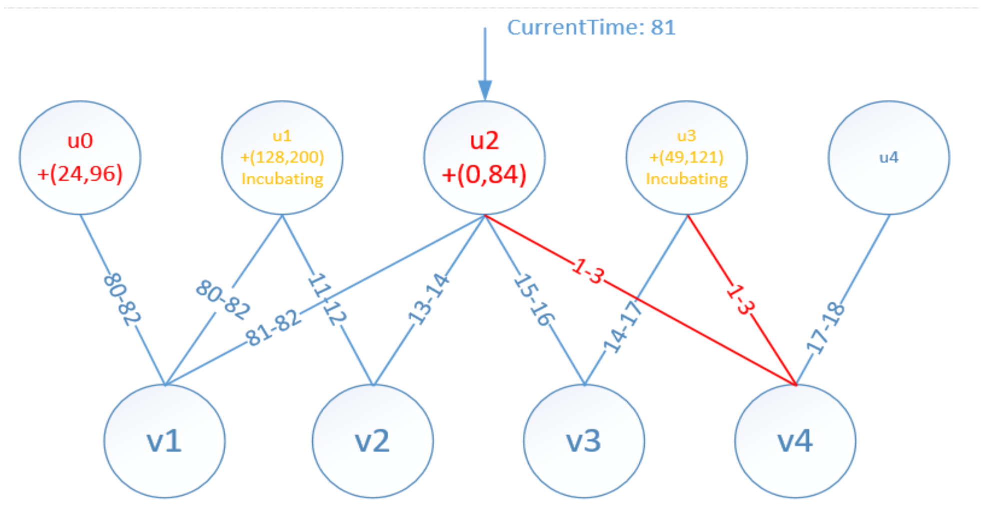


Figure 11 One round of the loop of Algorithm 1(Example)

Figure 11 shows a toy example demonstrating the core procedure of **Algorithm 1**, in which there are two time-overlapping wedges between u_2 and u_3: $W_1 = ((u_2, v_3, 15, 16), (v_3, u_3, 14, 17))$ and $W_2 = ((u_2, v_4, 1, 3), (v_4, u_3, 1, 3))$. Clearly, W_1 is later than W_2. If there are several possible carrying time spans for one vertex, the earliest one will be determined.

For searching single-source reachability, the time complexity is dominated by performing the main **for**-loop in line 5 - 23. Since the algorithm will traverse all wedges formed for this vertex, in the worst case, this vertex reaches all lower-layer vertices in one step, each of which in turn reaches all the other upper-layer vertices. The time complexity depends on the number of edges incident on this vertex and all its reachable lower-layer vertices. Thus, the time complexity of the main **for**-loop in **Algorithm 1** is bounded by $O(|E(U)| \sum_{i=1}^{|E(U)|} |E(L_i)|)$, where $E(U)$ is the number of edges incident on all the positive upper-layer vertices, and L is the set of the positive upper-layer vertices.

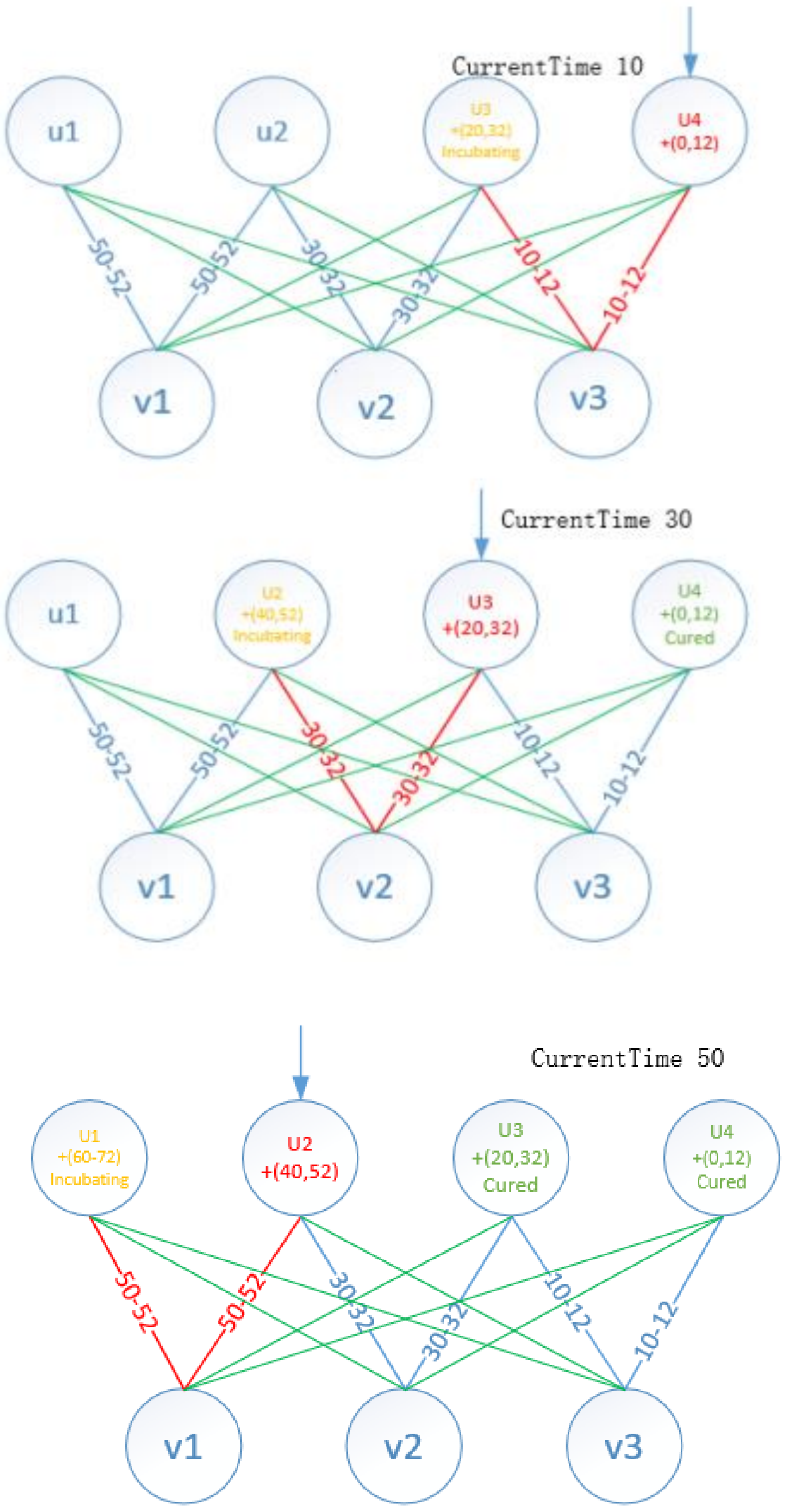


Figure 12 An example of the worst case of algorithm 1

For the time complexity of **Algorithm 1**, it can be more complicated. The iteration count of the while loop starting from *Line 1* depends upon the number of new carriers in each iteration. In the worst case, given a set of upper-layer vertices $U(G) = (u_1, u_2, \ldots, u_n)$, the initial positive vertex is the last vertex, and $\mathrm{u_n}$ only infected u_{n-1} in this iteration, each vertex will only infect its previous one vertex, such that only one vertex turns positive in one iteration. In this case, it requires *n* plus one iterations to end the loop. Thus the time complexity of **Algorithm 1** is $O(U(G) \sum_{i=1}^{U(G)} E(U_i) \sum_{j=1}^{E(U_i)} E(L_j))$.

Figure 12 illustrates one worst case of **Algorithm 1**. To simplify this example, the recovery time is set to 12 and the incubation time is set to 10, and there are some green edges, which mean this edge is not time-overlapping with other edges. Because the algorithm starts from front to back, in this example, when the previous vertex turns positive, its previous vertices will not be scanned again in this iteration. Hence, it requires 5 iterations to end the loop. If the algorithm starts from back to front, it only needs 2 iterations to end the loop.

Time-Labeling Method (TL)

Algorithm 2 is for the **T**ime-**L**abeling (TL), which is used to gather all positive visitors' time spans and merge overlapping time spans.

Similar to **Algorithm 1**, this algorithm traverses all those edges (*u*, *v*) starting from each vertex *u* initially labeled "POSITIVE" (or "INCUBATING", as long as it has a carrying time span), and stores the indexes (in order to generate transmission graphs) and time spans of these edges into a time span list associated with *v*, called a *visit record* (see lines 4 – 8 and lines 31 - 39). Note that only the time spans overlapped with the carrying time spans will be stored. *(**Remark:** Starting from lower-layer vertices is also feasible but it may consume more time since edg-*

es from "NEGATIVE" vertices are also traversed.) Then, all overlapping visit records will be merged in order to increase the performance, but it may decrease accuracy as the indexes will be also merged (see line 9 and lines 41 - 44).

After storing and merging all visit record, the algorithm starts searching from the first upper-layer vertex to the last one. For any vertex encountered, if it is "NEGATIVE" and one of the time spans of its edges is overlapped with the stored time spans of the corresponding vertex, then sets it to be "INCUBATING" and update its carrying time span (see lines 11 - 24). Also, the current time will be recorded (the current time will only be modified if and only if the old current time is later than the new current time). If there is a change of the status of an upper-layer vertex in one iteration, then continue to a next iteration.

In addition, a "NEGATIVE" vertex can visit several lower-layer vertices and have several overlapped time spans. However, possessing multiple carrying time span makes no sense in a short term (it can be useful for a very long time span, such as a data set containing 1-year data). In this case, the carrying time span is calculated based upon the earliest-arrival path.

ALGORITHM 2: TIME-LABELING

```
   Input: All upper vertices UpperG and all lower vertices LowerG in
   a temporal bipartite graph G
1 while flag
2   flag ← false
3   clear all timespans //Clear Visit Records of LowerG - L(G)
4   foreach v in UpperG do
5     if v.isPositive is not NEGATIVE or IMMUNE then
6       setVisitRecord(v)
7     end if
8   end foreach
9   mergeTimespan(foreach LowerVertex lv in LowerG)
10   foreach v in UpperG do
11    if v is NEGATIVE then
12      foreach e in v.edges
13       LowerVertex lv ← GetVertexById(e.id)
14         foreach timespan in lv.TimeSpan
15           if e.VisitTime is overlapped with lv.TimeSpan
16             CurrentTime ← e.VisitTime($t_e$)
17        //Set CurrentTime if VisitTime is bigger than CurrentTime
18             uv.CarryTime ← VisitTime + WaitTime
19             uv.isPositive ←   INCUBATING
20             flag ← true
21           end if
22         end foreach
23      end foreach
24    end if
25   end foreach
26  foreach v in UpperG do
27    Set v.isPositive based on CurrentTime and v.CarryTime
28  end foreach
29 end while
30
```

```
31 procedure setVisitRecord(UpperVertex v)
32   foreach e in v.edges
33     if v.CarryTime is overlapped with e.VisitTime then
34       $(t_s, t_e) \leftarrow$ e.VisitTime
35       LowerVertex lv ← GetVertexById(e.id)
36       lv.TimeSpan ← $(v.index, t_s, t_e)$
37     end if
38   end foreach
39 end procedure
40
41 procedure mergeTimespan(LowerVertex lv)
42   Merge all overlapping visit record,
43   //such as [$\boldsymbol{u_1}$,1,3],[$\boldsymbol{u_2}$,2,6] to [[$\boldsymbol{u_1}$, $\boldsymbol{u_2}$],1,6]
44 end procedure
```

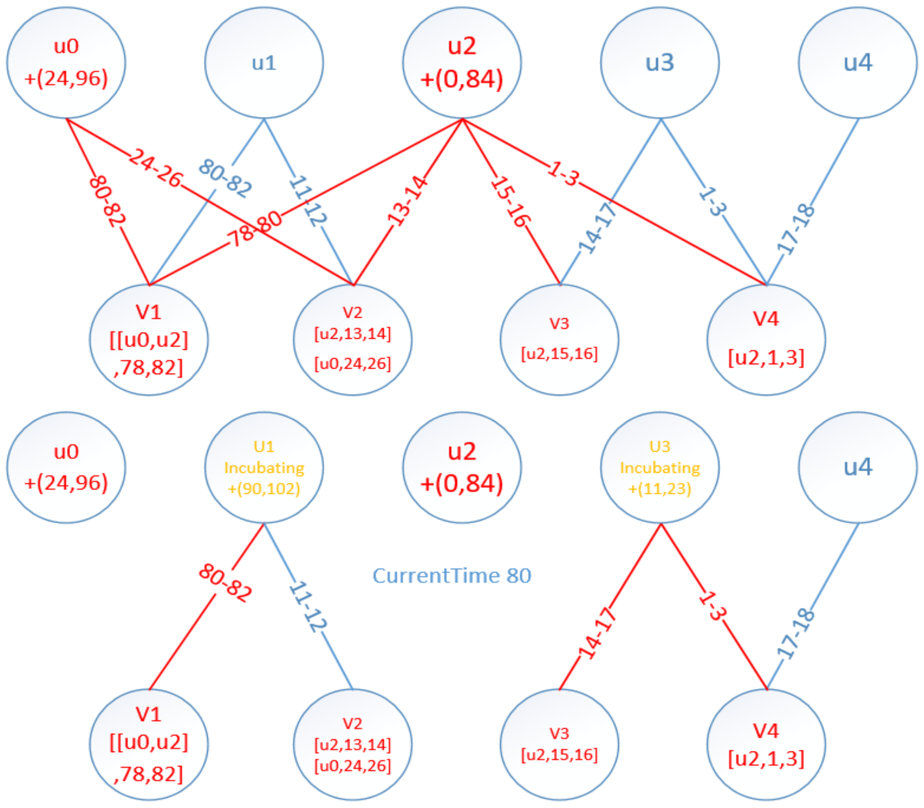


Figure 13 An example of Algorithm 2

As illustrated in Figure 13, unlike **Algorithm 1**, **Algorithm 2** firstly scans all edges starting from "POSITIVE" or "INCUBATING" vertices: u_0 and u_2, and then adds the time spans which are overlapped with their corresponding carrying time spans of the edges to their respective end vertices: v_1 and v_2 for u_0, and v_1, v_2, v_3 and v_4 for u_2. So [u_0, 80, 82] and [u_0, 24, 26] will be added to v_1 and v_2 while [u_2, 78, 80], [u_2, 13, 14], [u_2, 15, 16] and [u_2, 1,3] added to v_1, v_2, v_3 and v_4. Note that since there are overlapping visit records [u_1, 78, 80] and [u_2, 80, 82] for v_1, these two visit records will be merged to form a combined visit record: <[u_1, u_2], 78, 82> although it is not necessary. However, the merging can increase the performance by sacrificing accuracy.

After adding and merging all time spans for v_1, the algorithm will scan all other vertices u_1, u_3 and u_4, as illustrated in Figure 13. If the time span of an edge is overlapped with a visit record of a lower-layer vertex, then update the status and carrying time span of the end vertex (at the upper-layer) of the corresponding edge. One of the edges out of "NEGATIVE" vertex u_1 is with the time span 80 - 82, and its end vertex has a time span 78-82. So, the status and carrying time span of u_1 are set to be "INCUBATING" and 90 – 102, respectively. For u_3, there are two time-overlapping edges out of it, and the earlier one is selected. So, the status and carrying time span of u_3 are set to be "INCUBATING" and 11 – 23, respectively. u_4 does not have time-overlapping edges out of it. Therefore, its status is not changed. In the next iteration, the time spans of u_1 and u_3 will be added to the end vertices of the corresponding edges.

For adding time spans, the algorithm will scan all edges out of any "POSITIVE" or "INCUBATING" vertices in each iteration. In most cases, all edges out of each upper-layer vertices will have to be scanned eventually. So, the time complexity of this part is bounded by $O(U(G)\sum_{i=1}^{U(G)} E(U_i))$ on average.

For status-updating, the algorithm will scan all edges out of any "NEGATIVE" vertices. Then, for the corresponding lower-layer vertex of each of such edges, its visit records will be scanned to compare with the time span of the corresponding edge. Hence, the time complexity of this part is bounded by $O(U(G)\sum_{i=1}^{U(G)} E(U_i) \sum_{j=1}^{E(U_i)} I(E_{U_i}))$, where I is the visit records associated with the lower-layer vertices. I can be equal to $U(G)$ on average. Therefore, the time complexity is bounded by $O(U(G)^2 \sum_{i=1}^{U(G)} E(U_i))$.

From the above analysis, we can see that the time complexity of **Algorithm 2** is $O(U(G)^2 \sum_{i=1}^{U(G)} E(U_i) + U(G) \sum_{i=1}^{U(G)} E(U_i)) = O(U(G)^2 \sum_{i=1}^{U(G)} E(U_i))$.

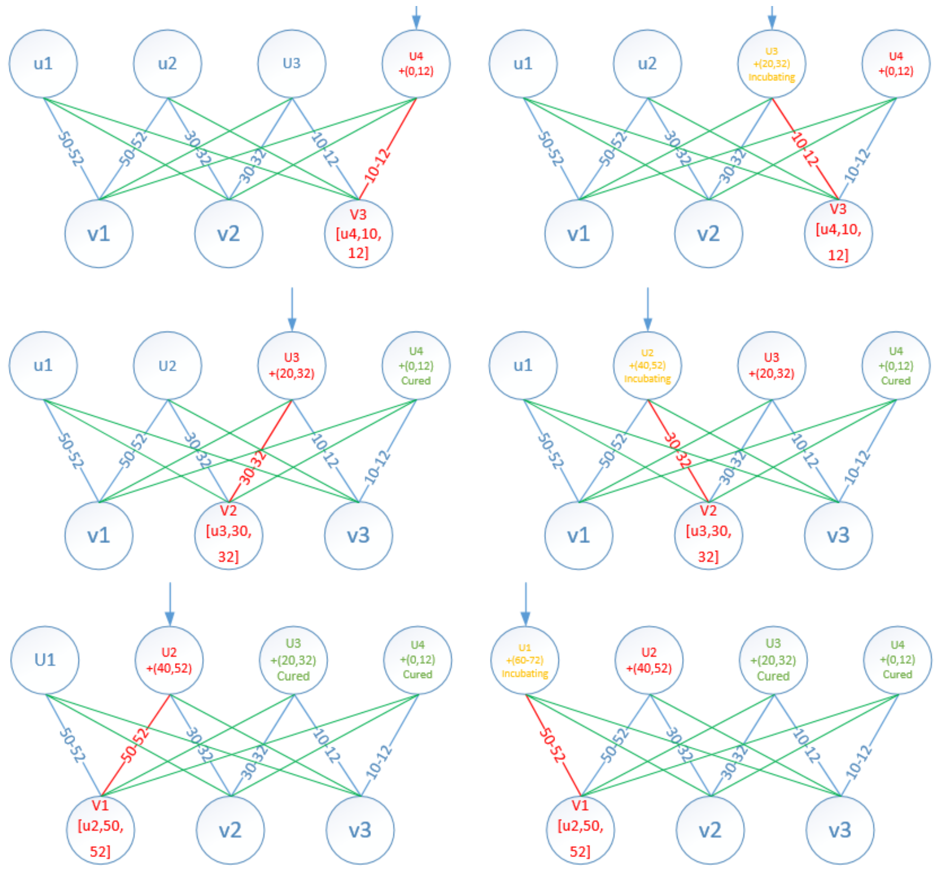


Figure 14 A worst case for Algorithm 2

Similar to **Algorithm 1**, the count of iterations is restricted by the number of status-changed vertices in each iteration. Figure 14 shows this example, in each iteration, only a time span of one upper-layer is added to a lower-layer vertex. In the step of updating statuses, the pointer traverses through the upper-layer rather than the lower-layer. Hence, all edges starting from upper-layer vertices will be scanned, which is time-consuming.

Dynamic Adjacency Arrays (DAA)

For a normal graph, an adjacency matrix or a collection of adjacency lists is used to represent the graph. In general, Arrays or matrices are more flexible than linked-lists, but require more space. **D**ynamic **a**djacency **a**rrays (DAA for short) combines their respective advantages. They not only keep the flexibility of arrays, but also have a lower memory usage. When using a language like C++, which provides encapsulated dynamic arrays such as **std::vector**, it is easy and convenient to implement this mechanism.

In our method, the DAA is used to store a temporal bipartite graph and will be scanned. So, in the following algorithm, it will be first created when loading the temporal bipartite graph is required. (See **Algorithm 3-1**.)

ALGORITHM 3-1: CREATING ADJACENCY ARRAYS

```
   Input: A file f describing the temporal bipartite graph

1 File f ← open_file(f)
2 upper_size, lower_size ← read_line(f)
3 Graph g ← initialize(upper_size, lower_size)
4 while(edge ← read_line(f))
5   $(idx_u, idx_l, t_s, t_e)$ ← edge //Time span, indexes of upper
6                  //and lower vertices
7   store vertices and edges to Graph g
8   g[$idx_u$].adj.add($idx_l, t_s, t_e$)
9   sort all adj by idx //Sort in ascending order
10 end while
```

Algorithm 3-1 is the brief description of loading a temporal bipartite graph from a file, which is stored as a set of edges. In the file, the first line is the upper-layer and lower-layer sizes. (See Table 1 for illustration.)

After the temporal bipartite graph is initialized, each upper-layer vertex will be assigned a dynamic array of the form (idx_l, t_s, t_e), stored as a structure. Edges in the file are stored in the form of (idx_u, idx_l, t_s, t_e). After an edge is read, idx_l, t_s, t_e from this edge will be assigned to the dynamic array associated with vertex idx_u.

All the dynamic arrays for edges will be sorted in ascending order of idx_l. In this way, the comparison of elements from two adjacency arrays can be efficiently performed.

Table 2 A file storing a temporal bipartite graph

5		4	
0(24,96)	1	80	82
0(24,96)	2	24	26
1	1	80	82

1	2	11	12
2(0,84)	1	78	80
2(0,84)	2	13	14
2(0,84)	3	15	16
2(0,84)	4	1	3
3	3	14	17
3	4	1	3
4	4	17	18

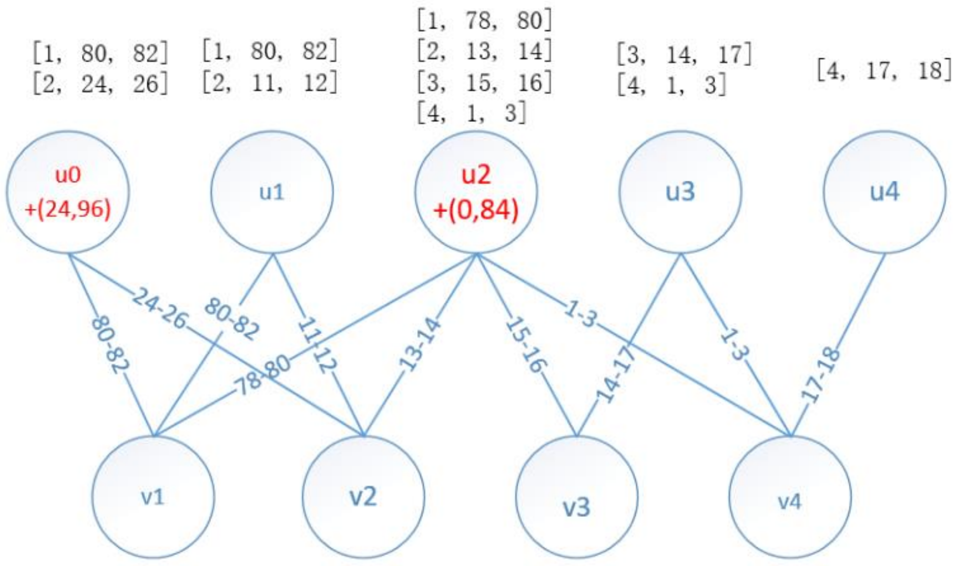


Figure 15 The temporal bipartite graph and adjacency arrays generated from Table 1

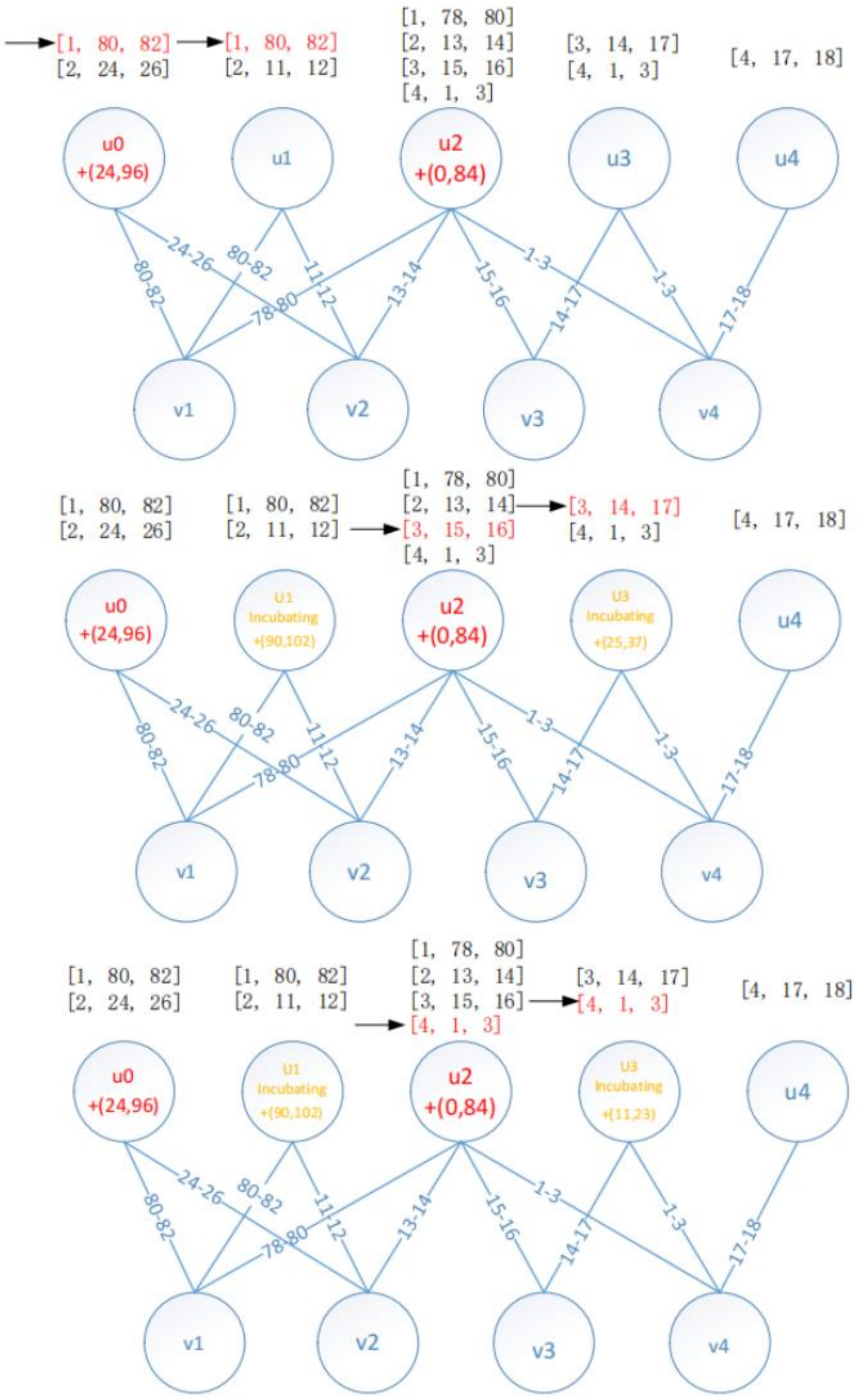
Table 2 is an example file storing a temporal bipartite graph, in which 5 upper-layer vertices with a dynamic array and 4 lower-layer vertices are initialized and assigned to a list of 9 entries. While creating linked-lists and their end vertices, the information of edges will also be added into the dynamic adjacency arrays. As illustrated in Figure 15, associated with the upper-layer vertices, are their respective adjacency arrays, storing indexes of end vertices (of edges), starting times and ending times.

In a next step, the computation will be based upon such adjacency arrays, as depicted in **Algorithm 3**. Like **Algorithm 1** and **2**, **Algorithm 3** still needs a loop and flag to iterate until there is no update on any upper-layer vertex occurs. In this process, the upper-layer vertices will be scanned one by one. If a "POSITIVE" or "INCUBATING" upper-layer vertex is encountered, then the lower-layer indexes in its adjacency arrays will be compared with the lower-layer indexes in the adjacency arrays of the other upper-layer vertices. In lines 8 - 19 of **Algorithm 3,** such index-comparison is conducted. Two indexes from two arrays will be compared to see if they are equivalent. We scan both arrays, starting from their respective first index. In this process, the pointer to the smaller index will move forward until the indexes are equivalent. In addition, if the carrying time span of a "POSITIVE" vertex is not overlapped with the time span of another vertex to which the "POSITIVE" vertex's pointer points, the pointer of "POSITIVE" vertex's array will move forward. When indexes are equivalent, their time spans will be compared. If the time spans are overlapped, update the status and the carrying time span of the target upper-layer vertex, as well as the current time. The principle of updating the status and carrying time span is same as **Algorithm 1** and **2**. Then, move to the next index or next vertex to start a new comparison. When all comparisons are done, updating statuses of all the upper-layer vertices based on their carrying time spans and the current time. If there is any update on an upper-layer vertex, start a new iteration until there is no update is needed.

ALGORITHM 3: DYNAMIC ADJACENCY ARRAYS

Input: All upper vertices UpperG in a temporal bipartite graph G

```
1 while flag
2   flag ← false
3   foreach $v_i$ in UpperG do
4    if $v_i$.isPositive is POSITIVE or INCUBATING then
5      foreach $v_j$ && $v_j$ != $v_i$ in UpperG do
6        if $v_j$.isPositive is NEGATIVE or INCUBATING then
7          i, j ← 0
8          while i < $v_i$.adj.size && j < $v_j$.adj.size
9            if $v_i$.adj[i].$(t_s, t_e)$ is not overlapped with $v_i$.CarryTime or
10             $v_i$.adj[i] < $v_j$.adj[j] then
11             i++
12           else if $v_i$.adj[i] > $v_j$.adj[j] then
13             j++
14           else if $v_i$.adj[i].$(t_s, t_e)$
15               is overlapped with $v_j$.adj[j].$(t_s, t_e)$ then
16             CurrentTime ← $t_e$ //if $\boldsymbol{t_e}$ is bigger than CurrentTime
17             $v_j$.isPositive ← INCUBATING
18             $v_j$.CarryTime ← $t_s$ + WaitTime
19             flag ← true
20           end if
21         end while
22       end if
23     end foreach
24   end if
25  end foreach
26  foreach v in UpperG do
27    Set v.isPositive based on CurrentTime and v.CarryTime
28  end foreach
29 end while
```



1.
Figure 16 An example of Algorithm 3

Figure 16 shows an iteration of Algorithm 3. At first, [1, 80, 82] of u_0 will compare with elements of u_1's adjacency arrays. Since the first index of u_1 is also 1 and the time spans are overlapped, the status and carrying time span will be updated. Next, since u_1 is incubating and has carrying time span, its adjacency array will be compared with some others'. u_0 and u_2 have the same indexes and overlapping time spans. However, because they both are positive, their status will not be changed. Then, the arrays of u_2 will be compared. For the first entry, u_0 and u_1 have the overlapping time spans, but it will not change their status. For the third entry, u_0's time span is overlapped with the first entry of u_3. So, u_3 turns to incubating and its carrying time span is set to be [25, 37]. However, after the forth entry of u_2 and the second entry of u_3 have been compared, there will be a new carrying time span [11, 23], which is earlier than the old one. Thus, eventually, the carrying time span of u_3 is set to be [11, 23].

Because each edge incident to a positive or an incubating vertex needs to be compared with edges incident to all the other vertices, the time complexity of Algorithm 3 is bounded by $\mathrm{O}(\sum_{\mathrm{i}=0}^{\mathrm{U_i}} \mathrm{E}(\mathrm{U_i}) \sum_{\mathrm{i}=0}^{\mathrm{U_i}} \mathrm{E}(\mathrm{U_i}))$.

It is still hard to deal with the worst case illustrated in Figure 17. Like Algorithm 1 and 2, the number of iterations in Algorithm 3 depends on the number of status-changing vertices in an iteration.

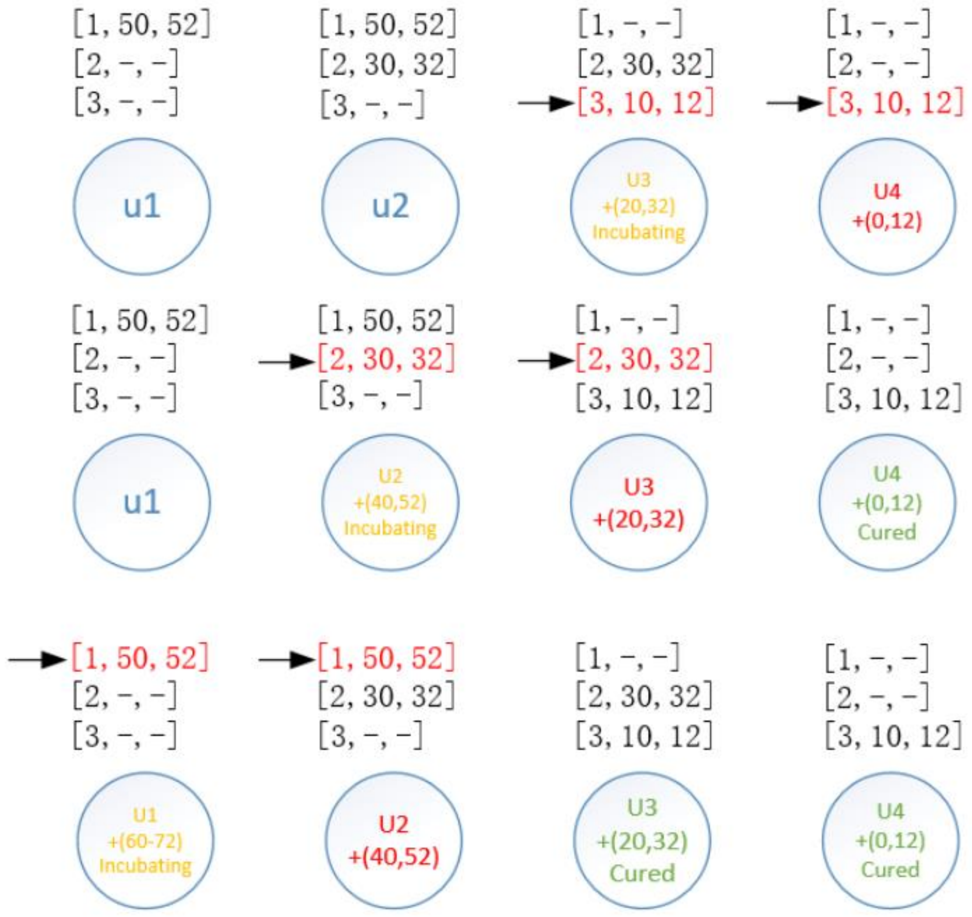


Figure 17 A worst case for Algorithm 3

Depth-First Strategy (DF)

Depth-**F**irst Strategy (DF) is also a brute-forcing and naive strategy. It is simple, but not so efficient.

Mainly, it contains two components: one is a recursive process to search the whole graph (**Algorithm 4**), and the other is responsible for passing all initially-positive vertices to the recursive function (**Algorithm 4-1**).

ALGORITHM 4: DEPTH-FIRST RECURSION PART

Input: An upper-layer vertex v

```
1 foreach e in v.edges do
2   if e.VisitTime is overlapped with v.CarryTime then
3     LowerVertex lv ← GetVertexById(e.id)
4     foreach le in lv.edges do
5       UpperVertex uv ← GetVertexById(le.id)
6         if e.VisitTime is overlapped with le.VisitTime then
7           CarryingStartTime ← the larger one between
8 le.VisitStartTime and e.VisitStartTime + IncubationTime
9         end if
10          if  uv.isPositive is NEGATIVE or uv.CarryingStartingTime
11  is bigger than CarryingStartTime then
12          uv.CarryingStartTime ← CarryingStartTime
13          uv.CarryingEndTime ← CarryingStartTime + RecoveryTime
14          Algorithm4(uv)
15        end if
16      end if
17    end foreach
18  end if
19 end foreach
```

ALGORITHM 4-1: DEPTH-FIRST MAIN PART

Input: All upper-layer vertices in a temporal bipartite graph

```
1 foreach POSITIVE upper-layer vertex v do
2   Algorithm4(v)
3 end foreach
4 foreach upper-layer vertex v do
5   Set v.isPositive based on latest ending time and v.CarryTime
6 end foreach
```

The input of **Algorithm 4-1** is set of initial positive upper-layer vertices. In **Algorithm 4**, for the set of edges associated with the passed vertices, if there is an edge whose visiting time span is overlapped with the positive time span of any input vertex, the corresponding lower-layer vertex will be found based on the ID stored in that edge (see lines 1-3). Then, the edges associated with this lower-layer vertex are traversed, forming wedges with the edge that store the ID of this lower-layer vertex (see line 4). Next, for each edge, the corresponding upper-layer vertex will be found based on the ID stored in that edge (see line 5). If the visiting time spans of two edges within the wedge overlap, the status of the corresponding upper-layer vertex will be checked, and the carrying starting time is computed (see lines 6 - 8). If the upper-layer vertex is "NEGATIVE" or the carrying starting time is earlier than the upper-layer vertex's carrying starting time, the carrying starting time of the upper-layer vertex is modified to the computed time, and the carrying ending time is calculated based on the average recovery time (see lines 9 - 11). Simultaneously, a new round of recursion is initiated with the upper-layer vertex as the parameter (see line 12). The termination condition for recursion is reached when no more infected vertices are founded, and the set of edges associated with the initial input (positive) vertex has been fully traversed (see line 1).

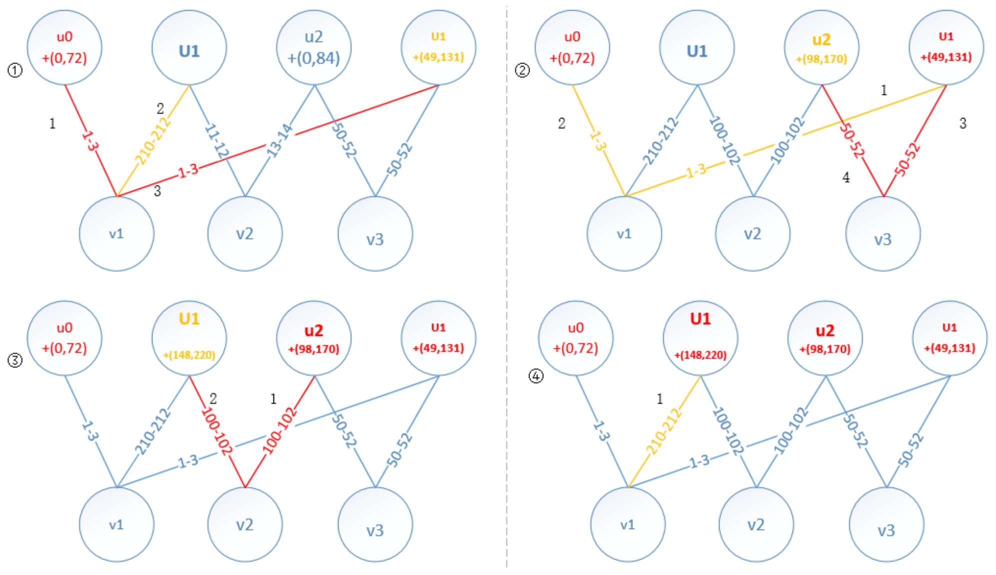


Figure 18 An example procedure of Depth-First Strategy

In Figure 18, the numbers associated with vertices (numbers put in a small circle) represent the current recursion level; and the numbers next to the edges indicate the order in which the edges are being examined. Furthermore, a red-colored edge indicates that the status or infection time of the end vertex has been modified while a yellow-colored edge signifies that there is no modification. When the status or infection time of the end vertex is changed, the corresponding vertex in this Figure turns yellow, and this vertex is used as a parameter in a next recursive call. In the next recursion round, this vertex will be changed to red.

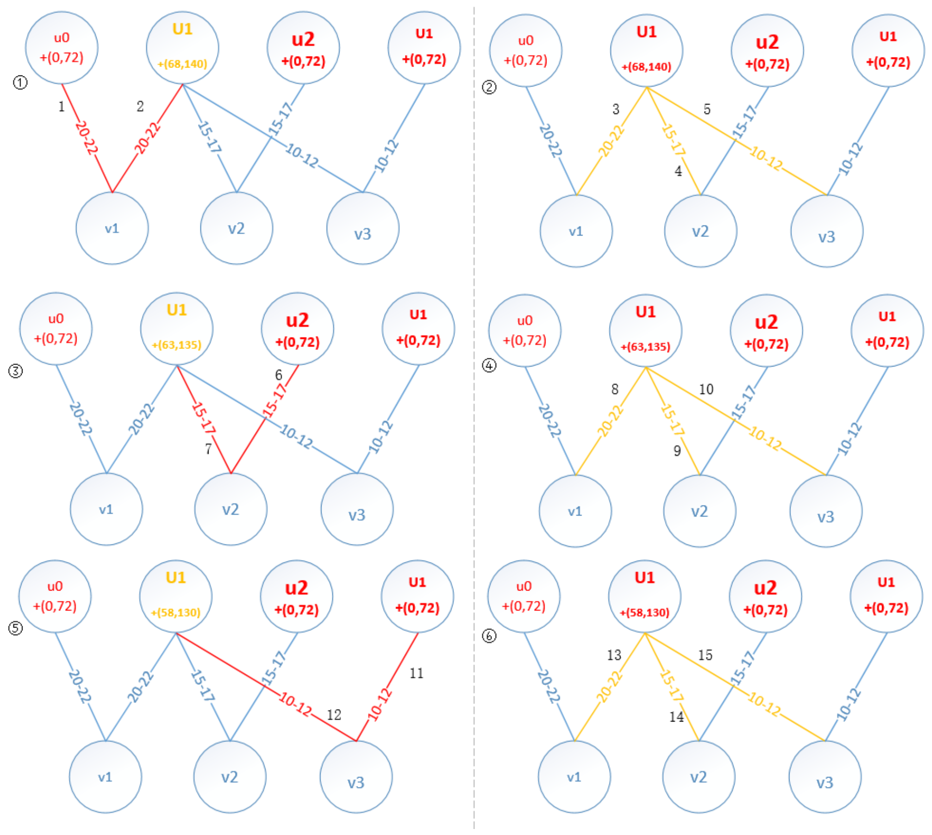


Figure 19 A worst case example of Depth-First Strategy

Assuming that each upper-layer vertex connects to every lower-layer vertex. The number of edges that incident to an upper-layer vertex equals to the number of lower-layer vertices, while the number of edges that incident to a lower-layer vertex equals to the number of upper-layer vertices. Since all upper-layer vertices' status is changed only once, the time complexity is bounded by $O(U(G)^2 L(G))$.

Figure 19 illustrated a worst case. In this case, the infection time of the vertex u_1 is changed more than one time. Every time its infection time is changed, all edges incident to it will be checked.

Finally, we notice that all those worst cases respectively illustrated in Figure 12, 14 and 17 can be easily dealt with **Algorithm 4** as its iteration number is not restricted by the number of status-changing vertices.

Single-Path Transmission Graph

Generation of Single-Path Transmission Graph

A single-path transmission graph is generated when searching reachability. So, the code for generating it needs to be inserted into several parts of **Algorithm 1** (see **Algorithm 1-1**).

Each time loading a temporal bipartite graph from a file, there are possibly some positive vertices. Normally, the number of such positive vertices is not predictable (since initially not all upper-layer vertices will be infected).

Positive upper-layer vertices will firstly be initialized (see line 1).

ALGORITHM 1-1: GENERATION OF SINGLE-PATH TRANSMISSION GRAPH

Input: All upper vertices UpperG and all lower vertices LowerG in a temporal bipartite graph G

Output: Single-Path Transmission Graph

```
7 initialize some vertices of SPTG based on initially-positive upper-
  layer vertices, set the upper-layer index and carrying time
8 //Root nodes
9 while flag
10   flag ← false
11   foreach v in UpperG do
12     if v.isPositive is not NEGATIVE or IMMUNE then
13       foreach e in v.edges do
14         if v.CarryTime is overlapped with e.VisitTime then
15           LowerVertex lv ← GetVertexById(e.id)
16           foreach le in lv.edges do
17             UpperVertex uv ← GetVertexById(le.id)
18             if e.VisitTime is overlapped with le.VisitTime
19                 && uv.isPositive is NEGATIVE then
20               Set flag, uv and current time (Lines 12-19 in Algorithm 1)
21               if GetVertexByIdx(uv.idx) not exists then
22                 father node ← GetVertexByIdx(v.idx)
23                 initialize SPTG vertex, set its index uv.idx, father
                   node v.idx, set edge storing transmission time and index
                   of this vertex, add the edge to father node
24               else
25                 father node ← GetVertexByIdx(v.idx)
26                 SPGV ← GetVertexByIdx(uv.idx)
27                 set index of SPGV, father node v.idx, set edge storing
                   transmission time and index of this vertex, add the edge
                   to father node
28             push SPGV to the single-path transmission graph list
29             end if
```

```
30          end foreach
31        end if
32      end foreach
33    end if
34  end foreach
35  foreach v in UpperG do
36    Set v.isPositive based on CurrentTime and v.CarryTime
37  end foreach
38 end while
39 return Single-Path Transmission Graph
```

The if-statement in the algorithm is the core part of generating single-path transmission graph (see lines 14 - 19). The algorithm will first acquire the positive or incubating vertex's index from the initial file. Then, it will check if the new "POSITIVE" vertex exists. If not, it will be initialized, its carrying time span will be set and an edge linking from old positive vertex to new positive vertex will be created. If exists, rather than initializes it, the algorithm will first find it by its index and then perform the same operations over it.

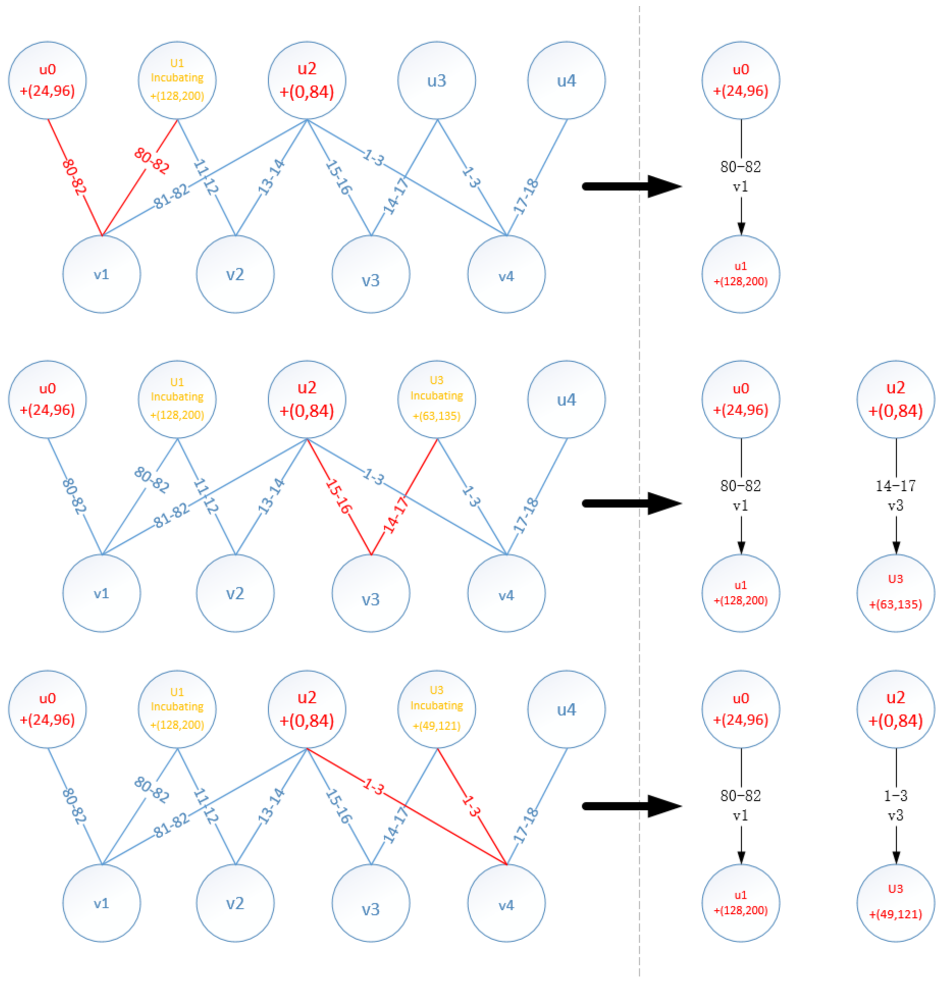


Figure 20 An example of generating a single-path transmission graph

Figure 20 illustrates how a single-path transmission graph is created by **Algorithm 1-1**. At first, u_0 transmits the disease to u_1 in the time span 80 – 82. So, new vertices u_0 and u_2 are initialized in the single-path transmission graph with an edge directed from u_0 to u_2, storing where and when the vertices contacted, Then, v_1 and with 80 - 82 is created. Next, u_2, u_3 with an edge going from u_2 to u_3 will be created. Because there is an earlier time span, the time span stored in the edge will also be changed to the earlier one. Note that not all upper-layer vertices will be created in the single-path transmission graph since some of them may not be infected. For example, u_4 is neither positive nor infected in this example.

Reachability Answering

In general, there are two simple approaches to answer reachability queries: *top-down* and *bottom-up*.

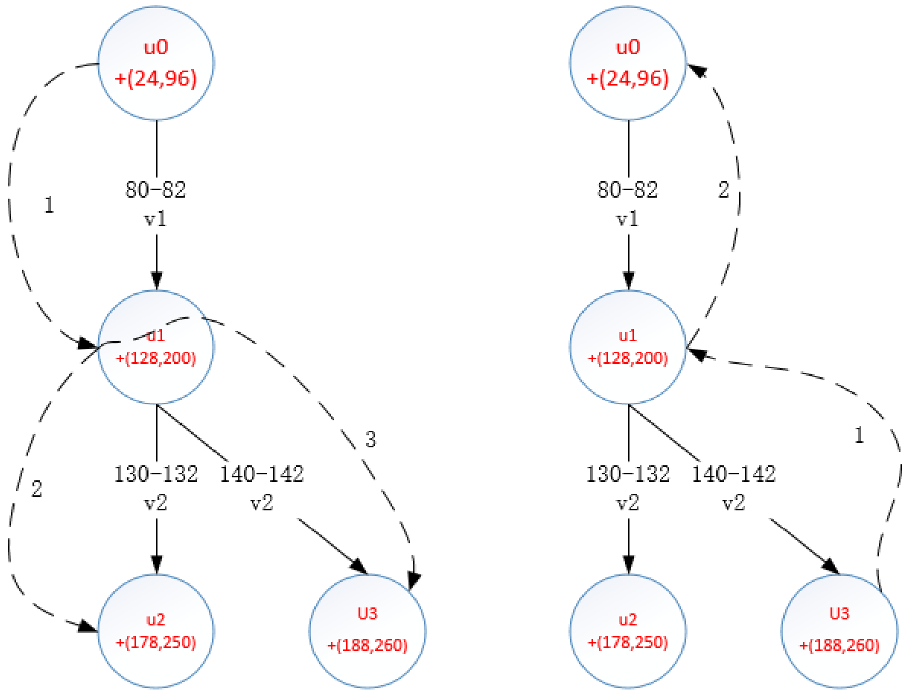


Figure 21 Top-Down and Bottom-Up Searching

The top-down is a naive method to answer the reachability. Given two vertices, u_i and u_j, this method initiates a depth-first search from u_i, exploring all its successor nodes until it reaches u_j, and returning True if u_j is found, and False otherwise.

On the other hand, the bottom-up is an optimized approach that makes use of the characteristic of single-path transmission graphs, where each node has an in-degree of 1 at maximum. Similarly, given two vertices u_i and u_j, it begins the search from u_j and progressively moves upwards until it encounters u_i, returning True if u_i is found and False otherwise. In comparison to the top-down approach, the bottom-up typically reduces the number of search operations in most cases, achieving a better theoretical performance.

In Figure 21, we compare these two methods, in which the dashed arrows represent the search paths while the number beside the dashed arrows represent the order of the searching. As shown in the figure, to answer the reachability between u_1 and u_3, the bottom-up reduces the number of search operations.

The bottom-up method can be further improved by associating an array (or a string) with each vertex to represent the unique path from a *root* vertex to the vertex.

Given two vertices, u_i and u_j, where u_j is a descendant of u_i, associated with an array $[j, \ldots, 1]$. Search the array or the string, if i is found in the array or u_i is found in the string, *true* will be returned. The time complexity of searching the array or the string is bounded by $O(\log N)$, where N is the number of vertices.

In addition, using the array or the string to answer a reachability query has another advantage that the transmission path can easily be displayed to the user. However, this algorithm needs to assemble a string to display, which can be time-consuming.

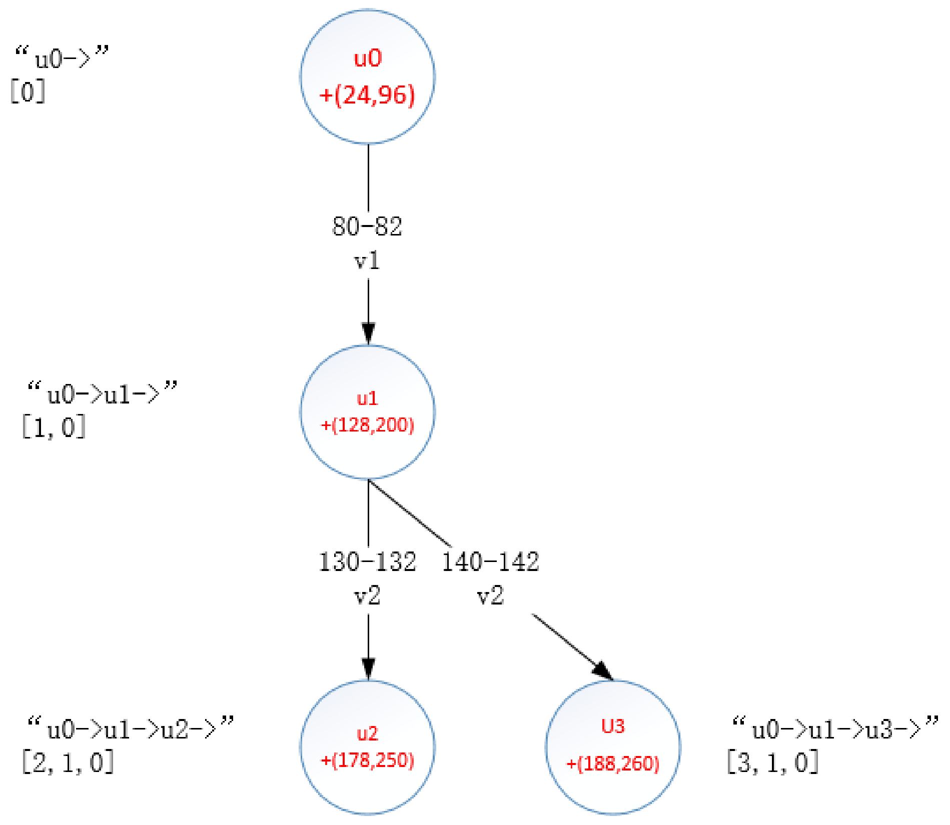


Figure 22 String and array buffers

In Figure 22, we show the arrays (or strings) associated with vertices. When answering the reachability from u_1 to u_3, for example, the algorithm matches "u1 $\rightarrow$" or 1 in the string or in the array.

Multi-Path Transmission Graph

Generation of Multi-Path Transmission Graph

Algorithm 2 can be slightly improved by using a multi-path transmission graph (see **Algorithm 2-1**).

Similar to **Algorithm 1-1**, those initial positive upper-layer vertices will be the root vertices of the multi-path transmission graph (see lines 5 - 8). If a negative upper-layer vertex reaches a lower-layer vertex within any time span of its visit records, a multi-path transmission graph vertex will be created. This vertex carries the carrying time span and index of its original upper-layer vertex, and its possible parent pointers. Its possible parents can be found in the visit records of the lower-layer vertex (see line 16 - 20). A data structure storing the pointer of this new vertex will be inserted into the children list of parents if the new vertex does not exist in the children list (see lines 21 - 27). The created vertex will be stored in a list storing all multi-path transmission graph vertices.

Next, following the second if-statement (see line 29), if the upper-layer vertex is not negative, but its carrying time can be updated to an earlier carrying time (which means the multi-path transmission graph vertex has been already created), then this multi-path transmission vertex will be founded by its index. The same operations (see lines 20 - 27) will be conducted on this vertex.

ALGORITHM 2-1: GENERATION OF MULTI-PATH TRANSMISSION GRAPH

Input: All upper vertices UpperG and all lower vertices LowerG in a temporal bipartite graph G

Output: Multi-Path Transmission Graph

```
1 while flag
2  flag ← false
3  clear all timespans
4  foreach v in UpperG do
5    if v.isPositive is not NEGATIVE or IMMUNE and it is the first iteration then
6      setTimespan(v)
7      initialize MPTG vertex(v.CarryTime, v.index)
8      push MPTG vertex to the Multi-Path Transmission Graph List
9    else if v.isPositive is not NEGATIVE or IMMUNE
10     setVisitRecord(v)
11   end if
12 end foreach
13 (Lines 9-15 in Algorithm 2)
14   CurrentTime ← e.VisitTime($t_e$)
15   CarryTime ← VisitTime + WaitTime
16   if v is NEGATIVE then
17     v.isPositive ← INCUBATING
18     v.CarryTime ← CarryTime
19     MPTGV ← initialize MPTG vertex(v.CarryTime, v.index)
20     insert parent indexes into MPTGV.parents from lv.TimeSpan
21     foreach parent in MPGTV.parents
22       if MPTGV not exists in children of parent then
23         Node ← initialize a Node(MPTGV_pointer)
24         Node.nextNode ← parent.nextNode
25         parent.nextNode ← Node
26       end if
```

```
27      end foreach
28      push MPTG vertex to the Multi-Path Transmission Graph List
29    else if v.CarryTime is earlier than CarryTime then
30      v.isPositive ← INCUBATING
31      v.CarryTime ← CarryTime
32      MPTGV ← findMPTGVertexById(v.index)
33      (Lines 19-27 Algorithm 2-1)
34 Continued from previous page
35 (Lines 21-29 in Algorithm 2)
36 return Multi-Path Transmission Graph
```

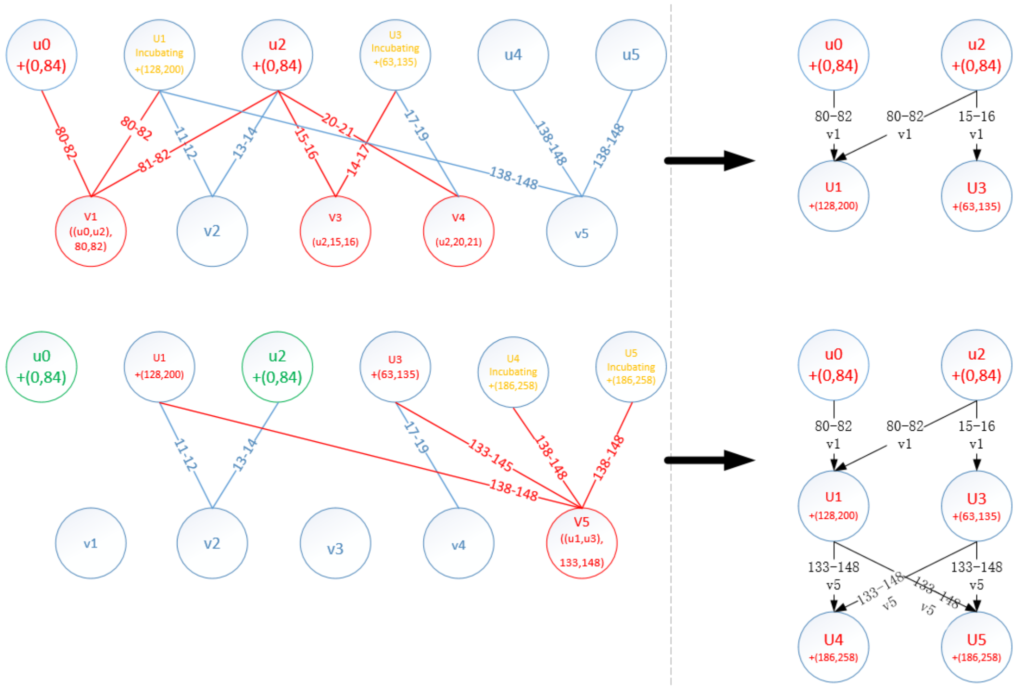


Figure 23 An example of generating a multi-path transmission graph

Figure 23 shows an example of the basic process to generate a multi-path transmission graph by using **Algorithm 2-1**. Two initial positive vertices u_0 and u_2, are the root vertices of the multi-path transmission graph. Since they are positive and reach v_1 and v_3 respectively, their visit time spans are added into the visit records of the two lower-layer vertices.

Then, u_1 and u_3 reach these lower-layer vertices, marked as incubating and created in the multi-path transmission graph as the child vertex of u_0 and u_2 respectively, where u_1 is the child vertex of both u_0 and u_2, as the visit record of v_1 has been merged.

In the next iteration, u_1 and u_3 as the new positive vertices, reach v_5 and a merged visit record of v_5 is created. u_4 and u_5 reaches v_5 within the time span of the visit record, and they therefore are created as both the child vertices of u_1 and u_3 in the multi-path transmission graph.

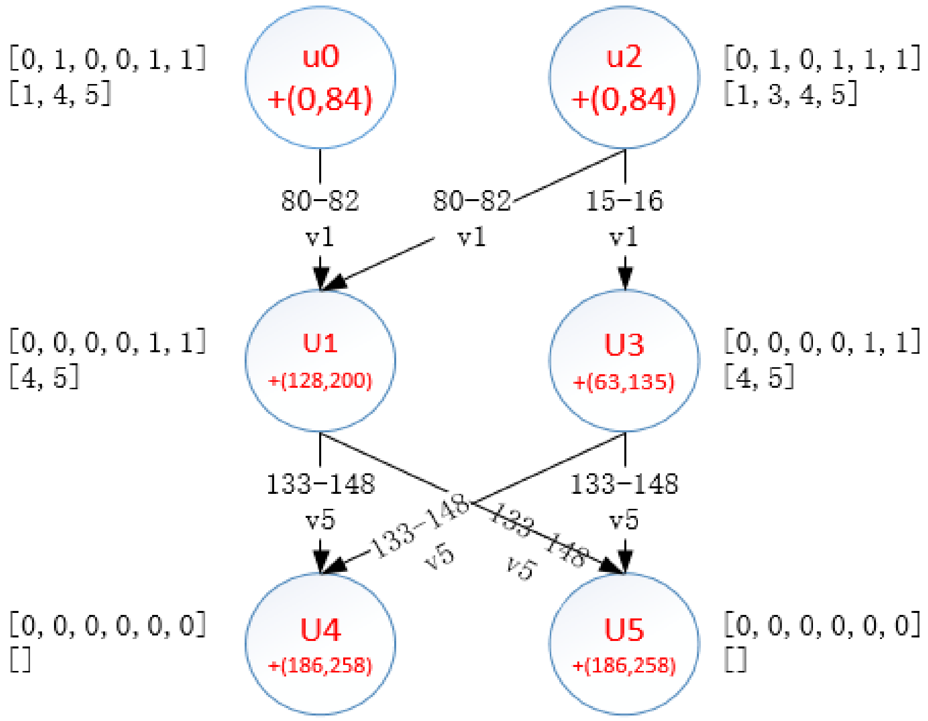


Figure 24 Answering reachability by indexes

Other than the top-down and the bottom-up techniques, which have the same principle as in single-path transmission graphs, adjacency arrays storing indexes of vertices can be created to speed up the searching of reachability. Given a vertex and an adjacency array with the length equal to the number of vertices, each index in the array represents the index of a vertex. In the array, "0" indicates that the corresponding vertex cannot be reached from the original vertex, while "1" indicates that it can. These arrays can also be compressed to arrays directly storing indexes. The arrays next to the vertices in Figure 24 are an example of such arrays. However, creating these arrays requires first searching all possible reachable pairs, which can be very time-consuming. When generating a multi-path transmission graph, the compressed arrays can be created. In addition, in **Algorithm 2-1**, the parents' indexes are inserted into a parent index list of each end vertex other than root vertices (see line 20). Every time a vertex in a multi-path transmission graph is created, all predecessors are found through the parent index list, the parent index lists of parents, et cetera. The index of the vertex is added into the adjacency arrays of all predecessors.

However, this indexing technique cannot show the transmission path of a pair.

A transmission path can be shown only by using traversal methods like the top-down or the bottom-up. They have the same performance since the in-degree of a vertex can be larger than 1. Also, it is possible that there is more than 1 transmission path, and therefore a graph traversal can be extremely time-consuming.

3.8 Evaluation of Reachability Queries Based on Indexes

A more efficient method for querying reachability is based on an indexing technique discussed in [12, 13]. For this method, we will first perform a depth-first traversal for the graph, generating a pre-order rank for each vertex. Then, assign a semi open closed interval $[i, j)$ to each vertex. See Figure 25 for illustration, in which each vertex v is associated with a pair of two values: its pre-order rank i, denoted as $p(v)$; and its interval $[i, x)$, denoted as *interv*(v), where x is equal to $d + 1$ and d is the largest rank among all the vertices in $G[v]$.

For example, vertex U_4's pre-order rank $p(U_4)$ is 2 while its interval *interv*(U_4) is [2, 4). It is because the largest rank among all the vertices in $G[U_4]$ is 3 and 3 + 1 = 4. For the same reason, U_6's pre-order rank is 3 while its interval is [3, 4). In the same way, you can check all the other vertices.

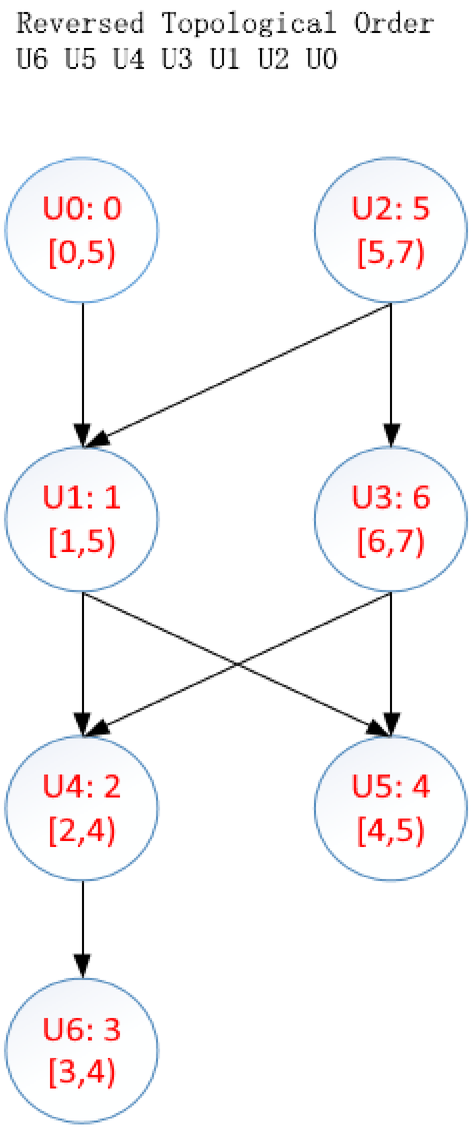


Figure 25 Example of intervals and Reversed topological order

Then, for each vertex v, we will create an interval sequence $s(v) = [a_1, b_1) \dots [a_k, b_k)$ for some k with the following property.

Let $u \neq v$ be a vertex in G. u is a descendant of v if and only if there exists an integer j ($1 \leq j \leq k$) such that $p(u) \in [a_j, b_j)$.

To create such an interval sequence for each vertex, we will first find a topological order for G, in which for any $(u, v) \in E$ u appears before v.

For example, for G shown in Figure 25, one of its topological sequences is shown below.

$$U_0 \longrightarrow U_2 \longrightarrow U_1 \longrightarrow U_3 \longrightarrow U_4 \longrightarrow U_5 \longrightarrow U_6$$

Then, along its reversed topological order, for each vertex v, we will merge the interval sequences of all its children with $interv(v)$ as follows.

Let $v_1, \dots, v_l$ be the child vertices of v. We will merge $s(v_1), \dots, s(v_l)$ with $interv(v)$ in turn. The result is stored in L. Initially, $L = interv(v)$.

Let $L = [a_1, b_1) \dots [a_k, b_k)$ be the result after $s(v_1), \dots, s(v_i)$ have been merged with $interv(v)$. The merging of L with $s(v_{i+1})$ will be done as below.

Let $s(v_{i+1}) = [c_1, d_1) \dots [c_r, d_r)$. We will scan both of L and $s(v_{i+1})$ from left to right.

Let $[a_i, b_i)$ (from L) and $[c_j, d_j)$ (from $s(v_{i+1})$) be the interval currently encountered. The following checkings will be conducted:

- If $c_j > a_i$, we go to the index next to $[a_i, b_i)$ and compare it with $[c_j, d_j)$ in a next step.
- If $a_i > c_j$, insert $[c_j, d_j)$ just before $[a_i, b_i)$. Go to the index next to $[c_j, d_j)$ and compare it with $[a_i, b_i)$ in a next step.
- If $a_i = cj$, we will compare b_i and d_j. If $b_i > d_j$, nothing will be done. If $d_j > b_i$, replace b_i with d_j. In both cases, we will go to the indexes next to $[a_i, b_i)$ and $[c_j, d_j)$, respectively.

As an example, consider the graph shown in Figure 25 again. Applying the above process to the vertices of the graph along a reversed topological order, we will create all the interval sequences for them as shown in the third column in Table 3.

Table 3. Merging the intervals

Vertex	Child Nodes	Intervals to be merged	Merged Intervals
U6		[3,4)	[3,4)
U5		[4,5)	[4,5)
U4	U6	[2,4)[3,4)	[2,4)
U3	U4 U5	[6,7)[2,4)[4,5)	[2,5)[6,7)
U1	U4 U5	[1,5)[2,4)[4,5)	[1,5)
U2	U1 U3	[5,7)[1,5)[2,5)[6,7)	[1,7)
U0	U1	[0,5)[1,5)	[0,5)

This method can be further improved by shortening each interval sequence associated with a vertex as follows:

Let $[a_1, b_1) \ldots [a_k, b_k)$ be a sequence associated with a certain vertex. For any two consecutive intervals: $[a_i, b_i)$ and $[a_{i+1}, b_{i+1})$ $(1 \leq i \leq k - 1)$ if $b_i = a_{i+1}$, we can replace them by a single interval $[a_i, b_{i+1})$ without impacting the correctness of reachability checking.

In this way, both querying time and space overhead can be greatly reduced, as illustrated by the fourth column for Table 3. From this, we can see that the whole space requirement for storing interval sequences is reduced by half.

4 EXPERIMENTAL RESULT

This chapter introduces the testing datasets and methods, and illustrates and analyze the experimental results of the proposed algorithms. Programs are implemented with C++, compiled by Microsoft Visual Studio 2019 with cl.exe being used. All the programs run on a Microsoft Windows 10 machine with 32 GB of memory and a 2.3 GHz 16-core processor.

Testing Datasets

Given the sizes of upper-level and lower-level vertices, as well as a maximum time value, the test datasets are generated according to the following rules:

1) Each upper-layer vertex is randomly connected to a certain number of lower-level vertices, determined by a provided maximum time value;

2) Within the visiting time span of the first edge for each vertex, the baseline starting time is set to 15, with a random offset in the range of [-14, 10]. The ending time is created by randomly adding 1, 2, or 3 to the corresponding starting time. Subsequently, for each incident edge, the baseline starting time is incremented by 50 until the baseline starting time exceeds the provided maximum time value. Then, no more edges will further be generated for that upper-layer vertex.

Multiple test datasets have been generated based on different input values, and, as depicted in Figure 25, they are named according to the following format: Upper-layer Size - Lower-layer Size (total number of edges). The running time of predicting transmission (**Algorithms 1-4**), of generating a transmission graph (**Algorithm 1.1 and 2.1**), as well as of answering reachability queries are all tested. In particular, the prediction results are compared with those yielded by the existing method.

For all the tests, the incubation time is set to 72, while the recovery time is set to 168.

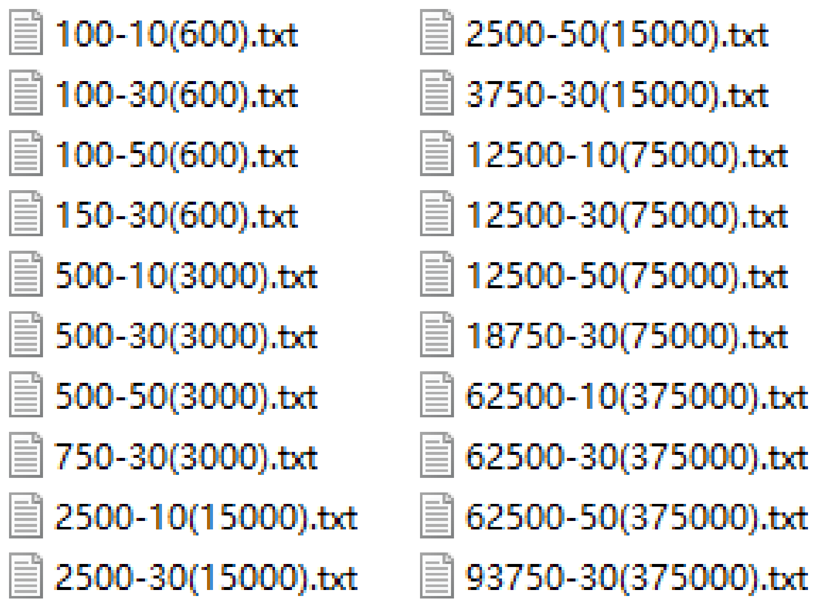

100-10(600).txt
100-30(600).txt
100-50(600).txt
150-30(600).txt
500-10(3000).txt
500-30(3000).txt
500-50(3000).txt
750-30(3000).txt
2500-10(15000).txt
2500-30(15000).txt
2500-50(15000).txt
3750-30(15000).txt
12500-10(75000).txt
12500-30(75000).txt
12500-50(75000).txt
18750-30(75000).txt
62500-10(375000).txt
62500-30(375000).txt
62500-50(375000).txt
93750-30(375000).txt

Figure 26 Naming of test datasets

Transmission Prediction Efficiency (Algorithms 1-4)

This section conducts performance testing on the transmission prediction algorithms. All the datasets are shown in Figure 26. Over some of them, all the four algorithms discussed in the previous sections (named BF, TL, DAA and DF, respectively) are executed. Their execution time is respectively demonstrated in Tables 4 to 7.

Table 4. Computation time (in milliseconds) of Algorithms 1-4

Edge\Algorithm	BF	TL	DAA	DF
600	0	0	0	0
3000	1	1	7	0
15000	28	4	38	22
75000	1681	60	568	6027
375000	189605	1284	13999	929597

In Table 4, the first column is the number of edges, the numbers of upper-layer vertices are respectively 100, 500, 2500, 12500 and 62500 while the numbers of lower-layer vertices are all 10.

Table 5. Computation time (in milliseconds) of Algorithms 1-4

Edge\Algorithm	BF	TL	DAA	DF
600	0	1	1	0
3000	0	3	41	0
15000	8	3	101	4
75000	577	26	798	902
375000	29205	251	17710	116508

In Table 5, the numbers of upper-layer vertices remain unchanged while the numbers of lower-layer vertices are all 30.

Table 6. Computation time (in milliseconds) of Algorithms 1-4

Edge\Algorithm	BF	TL	DAA	DF
600	1	1	1	0
3000	1	1	1	0
15000	2	4	183	1
75000	173	22	1182	121
375000	13230	190	21237	29436

In Table 6, the numbers of upper-layer vertices remain unchanged while the numbers of lower-layer vertices are all 50.

Table 7. Computation time (in milliseconds) of Algorithms 1-4

Edge\Algorithm	BF	TL	DAA	DF
600	0	1	2	1
3000	1	1	45	0
15000	11	4	174	7
75000	604	40	1701	477
375000	97001	249	36898	96709

In Table 7, the numbers of upper-layer vertices are respectively 150, 750, 3750, 18750 and 93750 while the numbers of lower-layer vertices are all 30.

From Tables 4 – 7, we can see that overall TL has the best performance, significantly outperforming all the other algorithms. Both BF and DF perform poorly. In general, as the number of edges increases, the execution time of each algorithm increases.

Comparing Tables 4, 5, and 6, it can also be observed that BF, TL, and DF have shorter execution time as the number of lower-layer vertices increases. This is quite opposite to DAA.

In addition, the dataset shown in Table 6 has 1.5 times the number of upper-level vertices in the dataset shown in Table 4. But they have the same number of edges. This implies that in the dataset of Table 6, upper-layer vertices are connected to fewer lower-layer vertices than Table 4. By comparing these two tables, we can see that BF and DAA both need longer execution time as the number of upper-level vertices increases while DF behaves oppositely. However, TL's execution time remains unchanged.

In order to further verify the performance of TL, we created a large test set. This test set has 900000 upper-layer vertices, 60000 lower-layer vertices and 4500000 edges. The execution time is 3196ms.

In summary, TL demonstrates the best performance.

Transmission Prediction Comparison

In this section, we compare the transmission prediction results between our strategy and the existing method [1]. The existing strategy, as compared to ours, does not take virus incubation and recovery times into consideration. It simply searches for the reachability of all pairs of upper-layer vertices in a brute-force manner. In the context of disease spread modeling, this approach assumes that as soon as a person comes into contact with any carrier, they will immediately become a carrier and remain infectious forever. This is obviously not so reasonable.

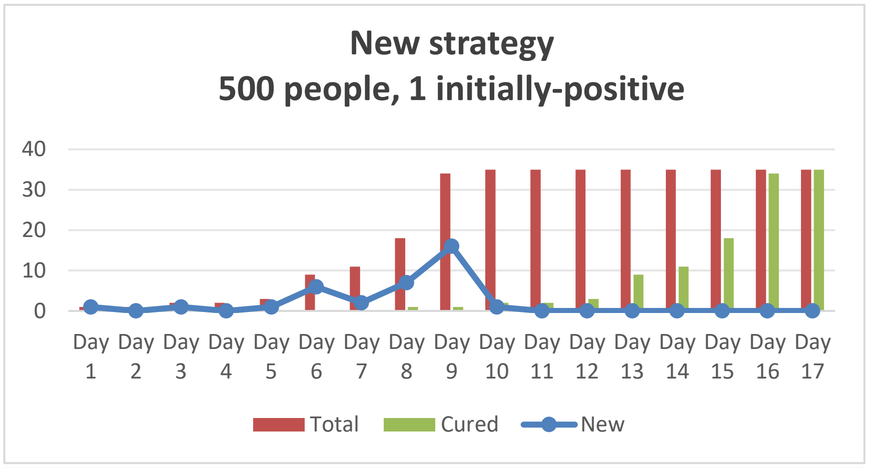
The test dataset used in this section consists of 500 upper-layer vertices, 30 lower-layer vertices, and 3000 edges. In this test, the BF algorithm is executed on the test dataset to compute infection time for two scenarios: one is our strategy with virus incubation and recovery times considered, and the other is the existing method, by which virus incubation and recovery time are not taken into account.

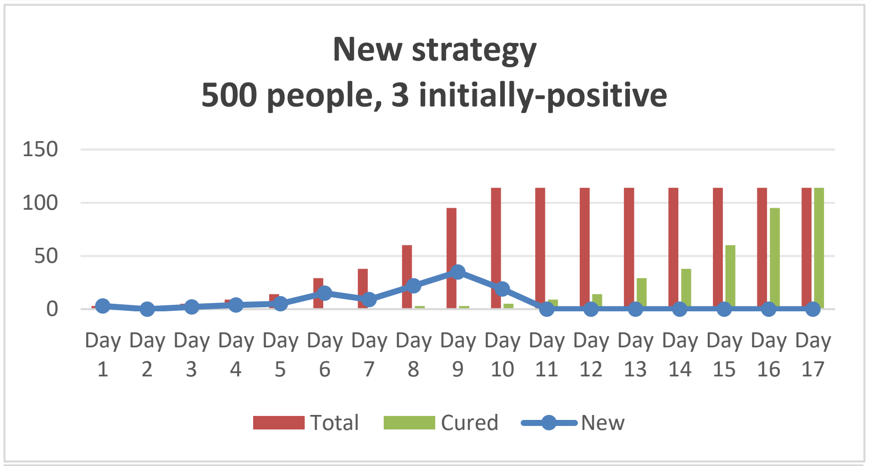
In Chart 1, we show the infection time by our strategy. In Chart 2, the infection time by the existing method is demonstrated, by which both the incubation and recovery time are set to 0.

Both of these charts contain three subplots, with the numbers of initial carriers set at 1, 3, and 5, resulting in different spreading scales. By these subplots, it can be observed that the difference in the number of initial carriers indeed leads to varying infection scales, i.e., more initial carriers ultimately result in more infected individuals. Besides, comparing the six blue curves labeled "New" in both charts, we can see a similar pattern, generally showing a peak in the middle with lower values on either side. In other words, the addition of carriers reaches its peak after some time and gradually declines.

Now, let us examine the red bars labeled "Total" in each subplot. In the first subplot of Chart 1, with only 1 initial carrier, it indicates that ultimately there are slightly more than 30 individuals becoming carriers. In contrast, in the first subplot of Chart 2 under the same conditions, there are nearly 400 individuals becoming carriers.

Finally, comparing the third subplots of both charts, where there are 5 initial carriers, we can see that by our strategy only slightly more than 200 individuals become carriers. In contrast, the existing strategy algorithm results in all individuals becoming carriers. No useful information is delivered.





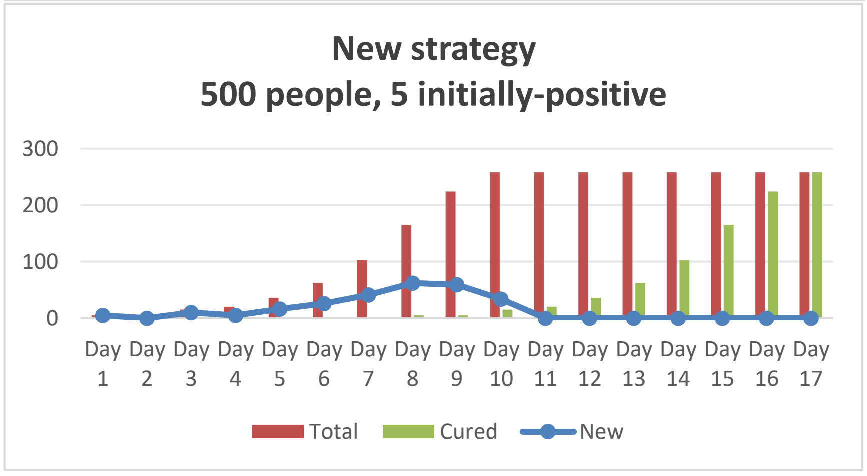


Chart 1. Transmission prediction by employing Algorithm 2

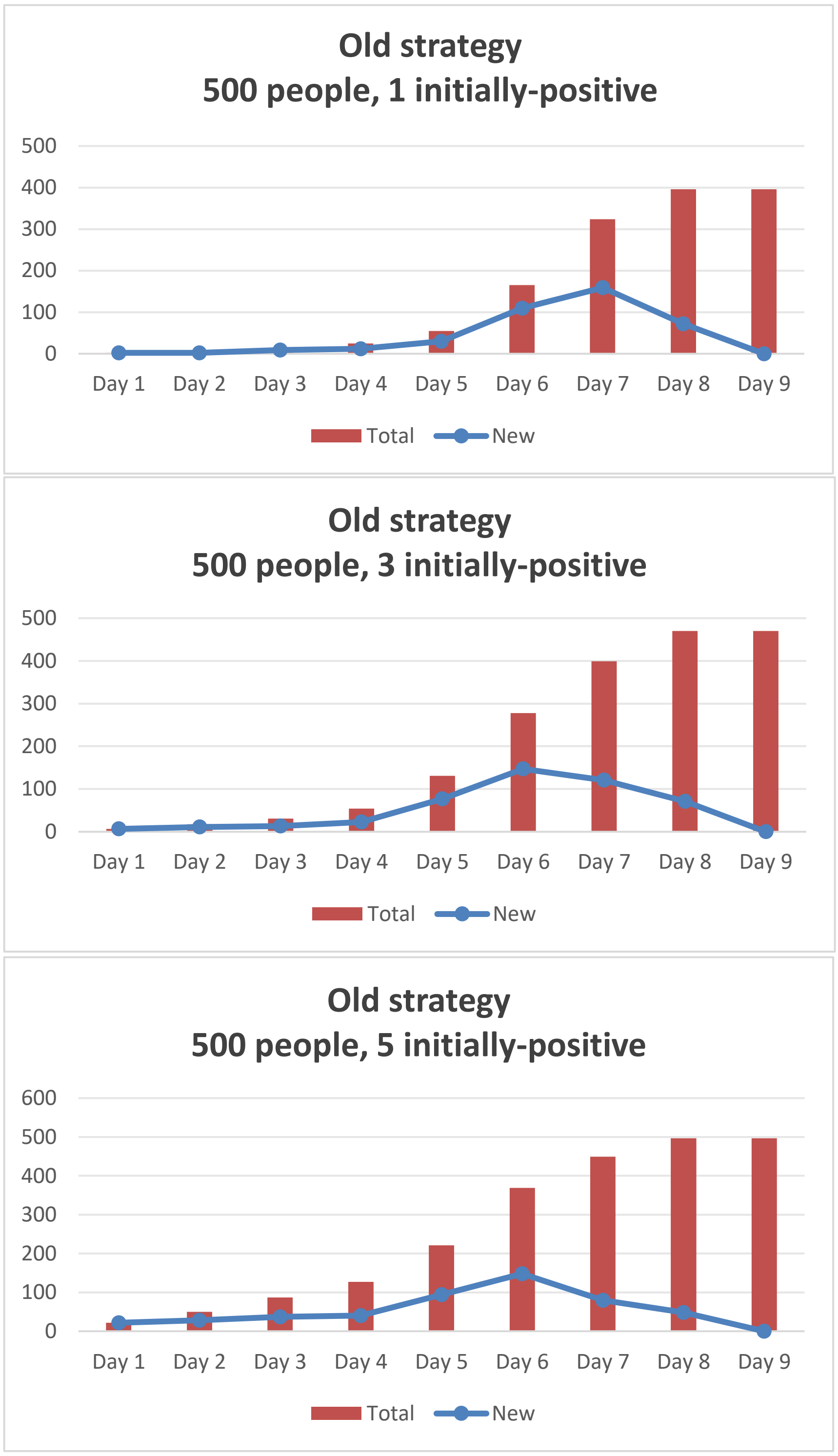


Chart 2. Transmission prediction by employing the old strategy

Transmission Graph Generation Testing (Algorithms 1.1 and 2.1)

In this section, the time for generating Transmission Graphs (Single-Path Transmission Graph SPTG and Multi-Path Transmission Graph MPTG) is tested, as shown in Table 7. They are the running time of Algorithm 1.1 (BF) and Algorithm 2.1 (TL). The dataset used for this test is the same as for Table 5.

Table 8. Generation time (in milliseconds) of Algorithms 1.1 and 2.1

Edge\Algorithm	SPTG(BF)	MPTG(TL)
600	0	0
3000	0	0
15000	3	10
75000	483	914
375000	13570	30200

In Table 8, the numbers of upper-layer vertices are respectively 100, 500, 2500, 12500 and 62500 while the numbers of lower-layer vertices are all 50.

Observing Table 8, it can be seen that for generating an MPTG approximately twice as much time as for generating an SPTG is taken, despite the results in Section 4.2 suggesting that the TL algorithm for predicting transmission is much faster than the BF algorithm for predicting transmission. This discrepancy is due to more edges in the MPTG than the SPTG.

It is also noted that when the total number of edges is less than 15,000, both types of Transmission Graphs can be generated very quickly. However, even though not much additional time for generating a large SPTG is incurred (from 13230 ms to 13570 ms), much more time is required for generating a large MPTG (from 190 ms to 30200 ms). This can be observed by comparing the data in the BF and TL columns from Table 6 in Section 4.2

Reachability Answering in Transmission Graph Testing

This section shows the performance testing for evaluating reachability queries on both Single-Path Transmission Graphs and Multi-Path Transmission Graphs. Due to the time-consuming nature of performing all possible reachability queries on a Transmission Graph, this test is done only on part of vertex pairs.

Table 9. Reachability answering time (in milliseconds) of different strategies

Pair\Strategy	Top-Down	Bottom-Up	Bottom-Up-Path	String-Buffer	Array-Buffer
100000	512	514	591	605	522
200000	1026	1030	1698	1212	1640
300000	1540	1526	3306	1807	2739
400000	2118	2065	5382	2519	3846
500000	2582	2674	7984	3015	5096
600000	3079	3073	11125	3615	6081
700000	3619	3595	15411	4230	7190

In Table 9, we show the time for evaluating reachability of upper-layer-vertex-pairs in a single-path transmission graph, the first column is the number of pairs.

Table 9 presents the test results for reachability queries on Single-Path Transmission Graphs. From this table, it can be seen that there is only a little difference in execution time between the top-down and bottom-up algorithms, with the bottom-up a little bit better. This is somewhat unexpected since the bottom-up is theoretically much better than its counterpart.

The "Bottom-Up-Path" column illustrates the additional time taken by the bottom-up algorithm to display the transmission path. It shows that generating a path string consumes a significant amount of time. On the other hand, the string-buffer algorithm, although slightly slower than the bottom-up, is significantly faster than the bottom-up-path when it comes to displaying transmission paths.

Unexpectedly, the "Array-Buffer" algorithm is slower than the other algorithms, suggesting that this approach may not be as successful.

Table 10. Answering time (in milliseconds) of different strategies

Pair\Strategy	Top-Down	Indexing	Index-Merging
100000	500	940	140
200000	1928	1971	292
300000	3244	2747	427
400000	4757	3815	606
500000	6016	4672	732
600000	7523	5838	824
700000	8784	6627	982

In Table 10, we show the time for evaluating reachability of upper-layer-vertex-pairs in a multi-path transmission graph by using indexes, in which the first column is the number of tested pairs.

From this, it can be seen that the indexing algorithm is generally faster than the top-down algorithm and the index-merging method is overwhelmingly faster than the other 2 algorithms.

However, when examining a bunch of 100,000 pairs of vertices with fewer reachable pairs, the top-down algorithm works better. One possible reason for this could be that there are fewer edges, but more pairs of vertices with smaller depth differences are selected. In this case, the execution time of the top-down algorithm is expected to be positively correlated with the depth difference between two vertices. In general, the indexing algorithm's execution time tends to be more consistent across different pairs of vertices.

Table 11. Comparison numbers of 2 strategies

Pair\Strategy	Indexing	Index-Merging
100000	5886672	907241
200000	11772811	1808868
300000	17659800	2712877
400000	24600881	3652890
500000	30233151	4547949
600000	36277074	5456668
700000	42324663	6360781

In Table 11, we show the comparison numbers of indexing and index-merging algorithms. Comparing with Table 10, we can see that both the answering time and comparison numbers of indexing are around 6 - 7 times of their index-merging counterparts. In addition, the sizes of indexes of indexing and index-merging are respectively 79.8MB and 72.2MB. Index-merging can compress the size of indexes.

The index-merging method has obviously the best performance among the 3 algorithms. A possible reason for this could be that multi-path transmission graphs are relatively simple, and for most vertices the number of intervals in the corresponding sequences is much smaller than the sequences for the indexing method.

5 CONCLUSION

In this thesis, we have examined the structure of temporal bipartite graphs, as well as transmission graphs, along with the associated reachability query problems. As an enhancement to the original temporal bipartite graph [1], we introduced a new disease spreading model, by which the incubation time and recovery time are considered to predict disease spreading more accurately.

For disease spread prediction, which involves specialized reachability queries across an entire temporal bipartite graph, we have designed several algorithms. Especially, a label-based algorithm is developed, based on a kind of time-labeling, by which high efficiency can be achieved.

In general, we distinguish between two kinds of transmission graphs: one is single-path transmission graphs and the other is multi-path transmission graphs. The key difference between them is whether a vertex can be infected by multiple predecessor vertices, leading to multiple infection paths. Both single-path and multi-path transmission graphs are created based on reachability query results. But the multi-path transmission graphs are better suited for generation using Time-Labeling. Both types of transmission graphs can be created efficiently.

We have also compared disease spread prediction results between the methods based on the original temporal bipartite graph and those based on the improved temporal bipartite graph. The test results on various sets of random datasets indicate that the methods based on the original temporal bipartite graph tend to label a large part of vertices or even all vertices as infected, whereas the methods based on the improved temporal bipartite graph label fewer vertices as infected. In this way, the accuracy is improved.

Finally, a new method to evaluate reachability queries on transmission graphs is designed based on Index-Merging, which significantly improves performance compared to the brute-force algorithms.

6 FUTURE WORK

For future prospects, we believe that the time-labeling algorithm may be suitable for GPU acceleration. As an example, see Figure 13. There are 5 upper-layer vertices in the graph. So, we can create 5 threads with each scanning edges starting from corresponding upper-layer vertices so that visiting records can be added to each corresponding end vertices simultaneously. Then, create another 4 threads for the 4 lower-layer vertices in order to check if there is any negative upper-layer vertex access during time spans of visiting records.

Additionally, the proposed dynamic adjacency arrays and depth-first algorithms can also be used to create single-path transmission graphs.

Furthermore, we can try to store transmission graphs into a file or a database so that we do not need to regenerate transmission graphs each time.

To enhance flexibility, we are considering a new algorithm that would enable the updating of the original transmission graph by reading in datasets from another time periods. For example, if a transmission graph based on data from January has been created, then reading in a dataset from February would extend the original transmission graph.

In the future work, further improvements to the time-labeling algorithm and extensions of dynamic adjacency arrays, as well as depth-first algorithms will be further explored.

REFERENCES

[1] Chen, Xiaoshuang, et al. "Efficiently answering reachability and path queries on temporal bipartite graphs." Proceedings of the VLDB Endowment (2021).

[2] Eubank, Stephen, et al. "Modelling disease outbreaks in realistic urban social networks." Nature 429.6988 (2004): 180-184.

[3] Goldstein, Michel L., Steven A. Morris, and Gary G. Yen. "Group-based Yule model for bipartite author-paper networks." Physical Review E 71.2 (2005): 026108.

[4] Xu, Jinliang, et al. "Latent interest and topic mining on user-item bipartite networks." 2016 IEEE international conference on services computing (SCC). IEEE, 2016.

[5] Tong, Hanghang, et al. "Proximity tracking on time-evolving bipartite graphs." Proceedings of the 2008 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics, 2008.

[6] Zeng, An, et al. "Trend prediction in temporal bipartite networks: the case of Movielens, Netflix, and Digg." Advances in Complex Systems 16.04n05 (2013): 1350024.

[7] Kasukawa, Takeya, et al. "Human blood metabolite timetable indicates internal body time." Proceedings of the National Academy of Sciences 109.37 (2012): 15036-15041.

[8] O'Connor, Clare M., Jill U. Adams, and Jennifer Fairman. "Essentials of cell biology." Cambridge, MA: NPG Education 1 (2010): 54.

[9] Smart, Ashley G., Luis AN Amaral, and Julio M. Ottino. "Cascading failure and robustness in metabolic networks." Proceedings of the National Academy of Sciences 105.36 (2008): 13223-13228.

[9] Wei, Yongyue, et al. "Comprehensive estimation for the length and dispersion of COVID-19 incubation period: a systematic review and meta-analysis." Infection 50.4 (2022): 803-813.

[11] Tyson, Ann Scott. "Why China's COVID-tracking QR codes raise surveillance concerns." The Christian Science Monitor, December 6, 2022. https://www.csmonitor.com/World/Asia-Pacific/2022/1206/Why-China-s-COVID-tracking-QR-codes-raise-surveillance-concerns.

[12] Wang, Haixun, et al. "Dual labeling: Answering graph reachability queries in constant time." 22nd International Conference on Data Engineering (ICDE'06). IEEE, 2006.

[13] Y. Chen, and Y.B. Chen. and Y. Zhang, Evaluation of Reachability Queries Based on Recursive DAG Decomposition, IEEE Transactions on Knowledge and Data Engineering (TKDE), Vol. 35, No. 8, Aug. 2023, pp. 7935 - 7952.

APPENDIX

SOURCE CODES

Appendix A provides source codes of generating single-path transmission graph (Algorithm 1.1) and codes of generating multi-path transmission graph (Algorithm 2.1) and codes of Algorithm 3. Refer to Snippet 1, 2, 3.

Snippet 1 C++ implementation of Algorithm 1.1

```
SinglePathGraph* Graph::Naive()
{
    bool flag = false;
    SinglePathGraph* spg = new SinglePathGraph();
    for (int i = 0; i < upperSize; i++)
    {
        UpperVertex* v = (UpperVertex*)vertices[i];
        if (v->isPositive == 1)
        {
            int t = getEarliestPositiveTime(v);
                            //Get earliest Ts of initially positive V
            if (t != -1)//initialize vertex of SPTG
            {
                v->positiveStartTime = t;
                v->positiveEndTime = t + recoveryTime;
                SPGVertex* spgv = new SPGVertex();

                spgv->idx = v->number;
                spgv->next = NULL;
                spgv->path = "->";
                spgv->path.append(to_string(spgv->idx));
                spgv->path.append("->");
                spgv->path1.push_back(spgv->idx);
                spgv->father = NULL;
                spgv->positiveStartTime = t;
                spgv->positiveEndTime = t + recoveryTime;
                spg->heads.push_back(spgv);
            }
        }
    }
    do
    {
        flag = false;
        int currentTime = -1;
        for (int i = 0; i < upperSize; i++)
        {
            UpperVertex* v = (UpperVertex*)vertices[i];
            if (v->isPositive == 1 || v->isPositive == 3)
            {//Find positive and incubating
                EdgeNode* e = v->firstEdge;
                while (e != NULL)
                {
                    if (isTimeOverlapping(v-
>positiveStartTime, v->positiveEndTime, e->startTime, e-
>endTime))
                    {//Find associating L
                        LowerVertex* lv = (LowerVer-
tex*)vertices[e->vertexIdx];
                        EdgeNode* le = lv->firstEdge;
                        while (le != NULL)
                        {//Find associating U
                            UpperVertex* uv = (UpperVer-
tex*)vertices[le->vertexIdx];
                            if (isTimeOverlapping(e-
>startTime, e->endTime, le->startTime, le->endTime) &&
uv->isPositive == 0)
                            {//If overlapping, calculate
carrying time span
                                int positiveStartTime =
(le->startTime > e->startTime ? le->startTime : e-
>startTime) + incubationTime;
                                int positiveEndTime =
positiveStartTime + recoveryTime;
                                if (positiveStartTime <
uv->positiveStartTime || uv->positiveStartTime == -1)
{//If there is an earlier carrying time span, update it
                                    currentTime = e-
>endTime > currentTime ? e->endTime : currentTime;
                                    uv->isPositive = 3;
                                    uv->positiveStartTime
= positiveStartTime;
                                    uv->positiveEndTime =
positiveEndTime;
                                    flag = true;
                                    if (spg-
>findVertexById(uv->number) == NULL)
                {//If vertex of SPTG does not exist, initialize it with carrying time span and location.
                    SPGVertex* spgvpre = spg->findVertexById(v->number);
                    SPGVertex* spgvnew = new SPGVertex();
                    SPGNode* spgn = new SPGNode();
                    int id = uv->number;
                    spgvnew->idx = id;
                    spgvnew->next = NULL;
                    spgvnew->path = spgvpre->path;
                    spgvnew->path.append(to_string(id));
                    spgvnew->path.append("->");
                    spgvnew->path1 = spgvpre->path1;
                    spgvnew->path1.push_back(id);
                    spgvnew->father = spgvpre;
                    spgn->transmissionTime = positiveStartTime - incubationTime;
                    spgn->spgv = spgvnew;
                    spgn->next = spgvpre->next;

                    spgvpre->next = spgn;

                    spg->vertices.push_back(spgvnew);
                }
                else {//If exists, find it and update it
                    int id = uv->number;
```

```
                                        SPGVertex* spgvpre = spg->findVertexById(id);
                                        SPGVertex* spgvnew = spg->findVertexById(v->number);
                                        SPGNode* spgn = new SPGNode();
                                        spgvnew->idx = id;
                                        spgvnew->next = NULL;
                                        spgvnew->path = spgvpre->path;
                                        spgvnew->path.append(to_string(id));
                                        spgvnew->path.append("->");
                                        spgvnew->path1 = spgvpre->path1;
                                        spgvnew->path1.push_back(id);
                                        spgvnew->father = spgvpre;
                                        spgn->transmissionTime = positiveStartTime - incubationTime;
                                        spgn->spgv = spgvnew;
                                        spgn->next = spgvpre->next;
                                        spgvpre->next = spgn;
                                    }
                                }
                            }
                            le = le->next;
                        }
                    }
                    e = e->next;
                }
            }
        }
        if (currentTime == -1)
        {//Update isPositive based on CurrentTime
            currentTime = latestEndTime;
        }
        cout << "By the time " << currentTime << ", the situation is:" << endl;
        for (int i = 0; i < upperSize; i++)
        {
            UpperVertex* v = (UpperVertex*)vertices[i];
            if (v->positiveEndTime > currentTime && v->positiveStartTime > currentTime - incubationTime && v->isPositive != 1)
            {
                v->isPositive = 3;
            }
            else if (v->positiveEndTime <= currentTime && v->isPositive != 0)
            {
                v->isPositive = 2;
            }
            else if (v->positiveEndTime > currentTime)
            {
                v->isPositive = 1;
            }
            switch (v->isPositive)
            {
                case 0:
                    cout << "Individual " << v->getNumber() << ": is negative" << endl;
                    break;
                case 1:
                    cout << "Individual " << v->getNumber() << ": is positive" << ", will recover from " << v->positiveEndTime << endl;
                    break;
                case 2:
                    cout << "Individual " << v->getNumber() << ": has been immune, recovered from " << v->positiveEndTime << endl;
                    break;
                case 3:
                    cout << "Individual " << v->getNumber() << ": is in incubation period, will be positive at " << v->positiveStartTime << " and recover at " << v->positiveEndTime << endl;
                    break;
            }
        }
```

```cpp
    } while (flag);
    int num1 = 0, num2 = 0, num3 = 0, num4 = 0;
    for (int i = 0; i < upperSize; i++)
    {//Calculate numbers of U with different status
        UpperVertex* v = (UpperVertex*)vertices[i];
        switch (v->isPositive)
        {
        case 0:
            num1++;
            break;
        case 1:
            num2++;
            break;
        case 2:
            num3++;
            break;
        case 3:
            num4++;
            break;
        }
    }
    cout << "Number of negative:" << num1 << endl;
    cout << "Number of positive:" << num2 << endl;
    cout << "Number of immune:" << num3 << endl;
    cout << "Number of incubating:" << num4 << endl;
    return spg;
}
```

Snippet 2: C++ implementation of Algorithm 2.1

```cpp
MultiPathGraph* Graph::algorithm2_1(int upperSize)
{
    bool flag = false;
    MultiPathGraph* mpg = new MultiPathGraph();
    mpg->upperSize = upperSize;
    do
    {
        clearRiskyTimespan();
        for (int i = 0; i < upperSize; i++)
        {//Find initially positive U
            UpperVertex* v = (UpperVertex*)vertices[i];
            if (v->isDisabled == false && (v->isPositive == 1 || v->isPositive == 3))
            {
                if (!flag)
                {
                    int t = getEarliestPositiveTime(v);
                    if (t != -1)
                    {

                        v->positiveStartTime = t;
                        v->positiveEndTime = t + recoveryTime;
                        //Add visit record and initialize vertex of MPTG
                        setRiskyTimespan_1(v);
                        MPGVertex* mpgv = new MPGVertex();
                        mpgv->idx = v->number;
                        mpgv->positiveStartTime = t;
                        mpgv->positiveEndTime = t + recoveryTime;
                        mpgv->next = NULL;
                        mpg->vertices.push_back(mpgv);
                        cout << "Carrier" << v->number << ": positiveStart:" << v->positiveStartTime << "positiveEnd: " << v->positiveEndTime << endl;

                    }
                }
                else
                {
                    setRiskyTimespan_1(v);
                    cout << "Carrier" << v->number << ": positiveStart:" << v->positiveStartTime << "positiveEnd: " << v->positiveEndTime << endl;
                }
                v->isDisabled = true;
            }
        }
        flag = false;
        int currentTime = -1;
        for (int i = 0; i < upperSize; i++)
        {
            UpperVertex* v = (UpperVertex*)vertices[i];
            if (v->isPositive == 0 || v->isPositive == 3)
            {//Find negative and incubating U
                EdgeNode* e = v->firstEdge;
                while (e != NULL)
        {//Find associating L
          LowerVertex* lv = (LowerVertex*)vertices[e->vertexIdx];
          for (int j = 0; j < lv->riskyTime.size(); j++)
          {
            if (isTimeOverlapping(lv->riskyTime[j][1], lv->riskyTime[j][2], e->startTime, e->endTime))
            {//If overlapped with visit record
              currentTime = e->endTime > currentTime ? e->endTime : currentTime;
              int positiveStartTime = (lv->riskyTime[j][1] > e->startTime ? lv->riskyTime[j][2] : e->startTime) + incubationTime;
              if (v->positiveStartTime == -1)
              {//Calculate carrying time span and if MPTG vertex does not exists, initialize it.
                v->positiveStartTime = positiveStartTime;
                v->positiveEndTime = positiveStartTime + recoveryTime;
                v->isPositive = 3;
                flag = true;
                MPGVertex* mpgv = new MPGVertex();
                mpgv->father.insert(lv->riskyTime[j][0]);
                mpgv->idx = v->number;
                mpgv->positiveStartTime = positiveStartTime;
                mpgv->positiveEndTime = v->positiveEndTime;
                mpg->insertReachableIntoFathers(mpgv, v->number);

                for (auto father : mpgv->father)
                {//Update indexing
                  auto fatherN = mpg->findVertexById(father);
                  if (checkSingle(fatherN, mpgv->idx)) {
                    MPGNode* mpgn = new MPGNode();
```

```
                                        mpgn->next = fatherN->next;
                                        mpgn->vertex = mpgv;
                                        fatherN->next = mpgn;
                                    }
                                }
                                mpg->vertices.push_back(mpgv);
                            }
                            else if(v->positiveStartTime > positiveStartTime && v->number != lv->riskyTime[j][0])
                            {//If exists, update it
                                v->positiveStartTime = positiveStartTime;
                                v->positiveEndTime = positiveStartTime + recoveryTime;
                                v->isPositive = 3;
                                MPGVertex* mpgv = mpg->findVertexById(v->number);
                                if (mpgv != NULL)
                                {
                                    mpgv->father.insert(lv->riskyTime[j][0]);
                                    mpgv->positiveStartTime = positiveStartTime;
                                    mpgv->positiveEndTime = v->positiveEndTime;
                                    mpg->insertReachableIntoFathers(mpgv, v->number);
                                    for (auto father : mpgv->father)
                                    {
                                        auto fatherN = mpg->findVertexById(father);
                                        if (checkSingle(fatherN, mpgv->idx)) {
                                            MPGNode* mpgn = new MPGNode();
                                            mpgn->next = fatherN->next;
                                            mpgn->vertex = mpgv;
                                            fatherN->next = mpgn;
                                        }
                                    }
                                    flag = true;
                                }
                            }
                        }
                    }
                    e = e->next;
                }
            }
        }
        if (currentTime == -1)
        {
            currentTime = latestEndTime;
        }
        cout << "By the time " << currentTime << ", the situation is:" << endl;
        for (int i = 0; i < upperSize; i++)
        {
            UpperVertex* v = (UpperVertex*)vertices[i];
            if (v->positiveEndTime > currentTime && v->positiveStartTime > currentTime - incubationTime && v->isPositive != 1)
            {
                v->isPositive = 3;
            }
            else if (v->positiveEndTime <= currentTime && v->isPositive != 0)
            {
                v->isPositive = 2;
            }
            else if (v->positiveEndTime > currentTime)
            {
                v->isPositive = 1;
            }
            switch (v->isPositive)
            {
                case 0:
                    cout << "Individual " << v->getNumber() << ": is negative" << endl;
                    break;
                case 1:
                    cout << "Individual " << v->getNumber() << ": is positive" << ", will recover from "
```

```
<< v->positiveEndTime << endl;
                    break;
                case 2:
                    cout << "Individual " << v->getNumber() << ": has been immune, recovered from " << v->positiveEndTime << endl;
                    break;
                case 3:
                    cout << "Individual " << v->getNumber() << ": is in incubation period, will be positive at " << v->positiveStartTime << " and recover at " << v->positiveEndTime << endl;
                    break;
            }
        }
    } while (flag);
    int num1 = 0, num2 = 0, num3 = 0, num4 = 0;
    for (int i = 0; i < upperSize; i++)
    {
        UpperVertex* v = (UpperVertex*)vertices[i];
        switch (v->isPositive)
        {
        case 0:
            num1++;
            break;
        case 1:
            num2++;
            break;
        case 2:
            num3++;
            break;
        case 3:
            num4++;
            break;
        }
    }
    cout << "Number of negative:" << num1 << endl;
    cout << "Number of positive:" << num2 << endl;
    cout << "Number of immune:" << num3 << endl;
    cout << "Number of incubating:" << num4 << endl;
    mpg->initReachableVertex();//Initialize index-merging
    return mpg;
}
```

Snippet 3 C++ implementation of Algorithm 3

```cpp
void Graph::Algo3()
{
    bool flag = false;
    for (int i = 0; i < upperSize; i++)
    {
        UpperVertex* v = (UpperVertex*)vertices[i];
        if (v->isPositive == 1 )
        {
            int t = getEarliestPositiveTime(v);
            if (t != -1)
            {//Find initially positive U to update recovery time
                v->positiveStartTime = t;
                v->positiveEndTime = t + recoveryTime;
            }
        }
    }
    //STEP1
    do
    {
        flag = false;
        int currentTime = -1;
        for (int i = 0; i < upperSize; i++)
        {
            UpperVertex* vi = (UpperVertex*)vertices[i];
            if (vi->isPositive == 1 || vi -> isPositive == 3)
            {//Find positive and incubating U
                for (int j = 0; j < upperSize; j++)
                {
                    UpperVertex* vj = (UpperVertex*)vertices[j];
                    if (j != i && vj->isPositive == 0)
                    {//Compare DAA of two vertices
                        int k1 = 0, k2 = 0;
                        auto arrI = vi->arr;
                        auto arrJ = vj->arr;
                        while (k1 < TwoHopI.size() && k2 < TwoHopJ.size())
                        {
                            if (!isTimeOverlapping(arrI[k1]->startTime, arrI[k1]->endTime, vi->positiveStartTime, vi->positiveEndTime) || arrI[k1]->idx < arrJ[k2]->idx)
                            {//If not overlapping, move pointer
                                k1++;
                            }
                            else if (arrI[k1]->idx > arrJ[k2]->idx)
                            {
                                k2++;
                            }
                            else
                            {
                                if (isTimeOverlapping(arrI[k1]->startTime, arrI[k1]->endTime, arrJ[k2]->startTime, arrJ[k2]->endTime))
                                {//If overlapping, update carrying time span
                                    currentTime = arrI[k1]->endTime > currentTime ? arrI[k1]->endTime : currentTime;
                                    vj->isPositive = 3;
                                    vj->positiveStartTime = (arrI[k1]->startTime < arrJ[k2]->startTime) ? TwoHopI[k1]->startTime : TwoHopJ[k2]->startTime;
                                    vj->positiveStartTime = vj->positiveStartTime + incubationTime;
                                    vj->positiveEndTime = vj->positiveStartTime + recoveryTime;
                                    flag = true;
                                }
                                k1++;
                                k2++;
                            }
                        }
                    }
                }
            }
        }
        if (currentTime == -1)
        {
            currentTime = latestEndTime;
        }
        cout << "By the time " << currentTime << ", the situation is:" << endl;
        for (int i = 0; i < upperSize; i++)
        {
            UpperVertex* v = (UpperVertex*)vertices[i];
```

```
          if (v->positiveEndTime > currentTime && v->positiveStartTime > currentTime - incubationTime && v->isPositive != 1)
          {
              v->isPositive = 3;
          }
          else if (v->positiveEndTime <= currentTime && v->isPositive != 0)
          {
              v->isPositive = 2;
          }
          else if (v->positiveEndTime > currentTime)
          {
              v->isPositive = 1;
          }
          switch (v->isPositive)
          {
              case 0:
                  cout << "Individual " << v->getNumber() << ": is negative" << endl;
                  break;
              case 1:
                  cout << "Individual " << v->getNumber() << ": is positive" << ", will recover from " << v->positiveEndTime << endl;
                  break;
              case 2:
                  cout << "Individual " << v->getNumber() << ": has been immune, recovered from " << v->positiveEndTime << endl;
                  break;
              case 3:
                  cout << "Individual " << v->getNumber() << ": is in incubation period, will be positive at " << v->positiveStartTime << " and recover at " << v->positiveEndTime << endl;
                  break;
          }
      }
    } while (flag);
    int num1 = 0, num2 = 0, num3 = 0, num4 = 0;
    for (int i = 0; i < upperSize; i++)
    {
        UpperVertex* v = (UpperVertex*)vertices[i];
        switch (v->isPositive)
        {
        case 0:
            num1++;
            break;
        case 1:
            num2++;
            break;
        case 2:
            num3++;
            break;
        case 3:
            num4++;
            break;
        }
    }
    cout << "Number of negative:" << num1 << endl;
    cout << "Number of positive:" << num2 << endl;
    cout << "Number of immune:" << num3 << endl;
    cout << "Number of incubating:" << num4 << endl;
}
```

Snippet 3 C++ implementation of Algorithm 4

```
void Graph::depth_recursion(UpperVertex* v)
{
    auto e = v->firstEdge;
    while (e != NULL)
    {//Find associating L
        int st = e->startTime;
        int et = e->endTime;
        if (isTimeOverlapping(v->positiveStartTime, v-
>positiveEndTime, st, et))
        {//If overlapped, find associating U
            LowerVertex* lv = (LowerVertex*)vertices[e-
>vertexIdx];
            auto le = lv->firstEdge;
            while (le != NULL)
            {
                UpperVertex* uv = (UpperVertex*)vertices[le-
>vertexIdx];
                if (isTimeOverlapping(st, et, le->startTime,
le->endTime))
                {//If overlapped, update carrying time span
                    int positiveStartTime = (st < le-
>startTime ? le->startTime : st) + incubationTime;
```

```
                    if (uv->positiveStartTime == -1 || uv->positiveStartTime > positiveStartTime)
                    {
                        uv->positiveStartTime = positiveStartTime;
                        uv->positiveEndTime = positiveStartTime + 168;
                        if (latestEndTime < uv->positiveEndTime)
                        {
                            latestEndTime = uv->positiveEndTime;
                        }
                        depth_recursion(uv);
                    }
                }
                le = le->next;
            }
        }
        e = e->next;
    }
}
void Graph::depth()
{
    for (int i = 0; i < upperSize; i++)
    {
        UpperVertex* v = (UpperVertex*)vertices[i];
        if (v->isPositive == 1)
        {//Find initially positive U to update recovery time
            int t = getEarliestPositiveTime(v);
            if (t != -1)
            {
                v->positiveStartTime = t;
                v->positiveEndTime = t + recoveryTime;
                v->isInit = true;
            }
        }
    }
    for (int i = 0; i < upperSize; i++)
    {//Find initially positive U to call recursion function
        UpperVertex* v = (UpperVertex*)vertices[i];
        if (v->isInit == true)
        {
            depth_recursion(v);
        }
    }
    for (int i = 0; i < upperSize; i++)
    {
        UpperVertex* v = (UpperVertex*)vertices[i];
        if (v->positiveEndTime > latestEndTime && v->positiveStartTime > latestEndTime - incubationTime)
        {
            v->isPositive = 3;
        }
        else if (v->positiveEndTime <= latestEndTime && v->positiveEndTime != -1)
        {
            v->isPositive = 2;
        }
        else if (v->positiveEndTime > latestEndTime)
        {
            v->isPositive = 1;
        }
    }
    int num1 = 0, num2 = 0, num3 = 0, num4 = 0;
    for (int i = 0; i < upperSize; i++)
    {
        UpperVertex* v = (UpperVertex*)vertices[i];
        switch (v->isPositive)
        {
        case 0:
            num1++;
            break;
        case 1:
            num2++;
            break;
        case 2:
            num3++;
            break;
        case 3:
            num4++;
            break;
        }
    }
    cout << "Number of negative:" << num1 << endl;
    cout << "Number of positive:" << num2 << endl;
    cout << "Number of immune:" << num3 << endl;
    cout << "Number of incubating:" << num4 << endl;
```

```
}
```

REFERENCES

[10] Chen, Xiaoshuang, et al. "Efficiently answering reachability and path queries on temporal bipartite graphs." Proceedings of the VLDB Endowment (2021).

[11] Eubank, Stephen, et al. "Modelling disease outbreaks in realistic urban social networks." Nature 429.6988 (2004): 180-184.

[12] Goldstein, Michel L., Steven A. Morris, and Gary G. Yen. "Group-based Yule model for bipartite author-paper networks." Physical Review E 71.2 (2005): 026108.

[13] Xu, Jinliang, et al. "Latent interest and topic mining on user-item bipartite networks." 2016 IEEE international conference on services computing (SCC). IEEE, 2016.

[14] Tong, Hanghang, et al. "Proximity tracking on time-evolving bipartite graphs." Proceedings of the 2008 SIAM International Conference on Data Mining. Society for Industrial and Applied Mathematics, 2008.

[15] Zeng, An, et al. "Trend prediction in temporal bipartite networks: the case of Movielens, Netflix, and Digg." Advances in Complex Systems 16.04n05 (2013): 1350024.

[16] Kasukawa, Takeya, et al. "Human blood metabolite timetable indicates internal body time." Proceedings of the National Academy of Sciences 109.37 (2012): 15036-15041.

[17] O’Connor, Clare M., Jill U. Adams, and Jennifer Fairman. "Essentials of cell biology." Cambridge, MA: NPG Education 1 (2010): 54.

[18] Smart, Ashley G., Luis AN Amaral, and Julio M. Ottino. "Cascading failure and robustness in metabolic networks." Proceedings of the National Academy of Sciences 105.36 (2008): 13223-13228.

[9] Wei, Yongyue, et al. "Comprehensive estimation for the length and dispersion of COVID-19 incubation period: a systematic review and meta-analysis." Infection 50.4 (2022): 803-813.

[11] Tyson, Ann Scott. "Why China's COVID-tracking QR codes raise surveillance concerns." The Christian Science Monitor, December 6, 2022. https://www.csmonitor.com/World/Asia-Pacific/2022/1206/Why-China-s-COVID-tracking-QR-codes-raise-surveillance-concerns.

[12] Wang, Haixun, et al. "Dual labeling: Answering graph reachability queries in constant time." 22nd International Conference on Data Engineering (ICDE'06). IEEE, 2006.

[13] Y. Chen, and Y.B. Chen. and Y. Zhang, Evaluation of Reachability Queries Based on Recursive DAG Decomposition, IEEE Transactions on Knowledge and Data Engineering (TKDE), Vol. 35, No. 8, Aug. 2023, pp. 7935 - 7952.