NeutronStream: A **Dynamic** GNN Training Framework with **Sliding Window** for Graph Streams

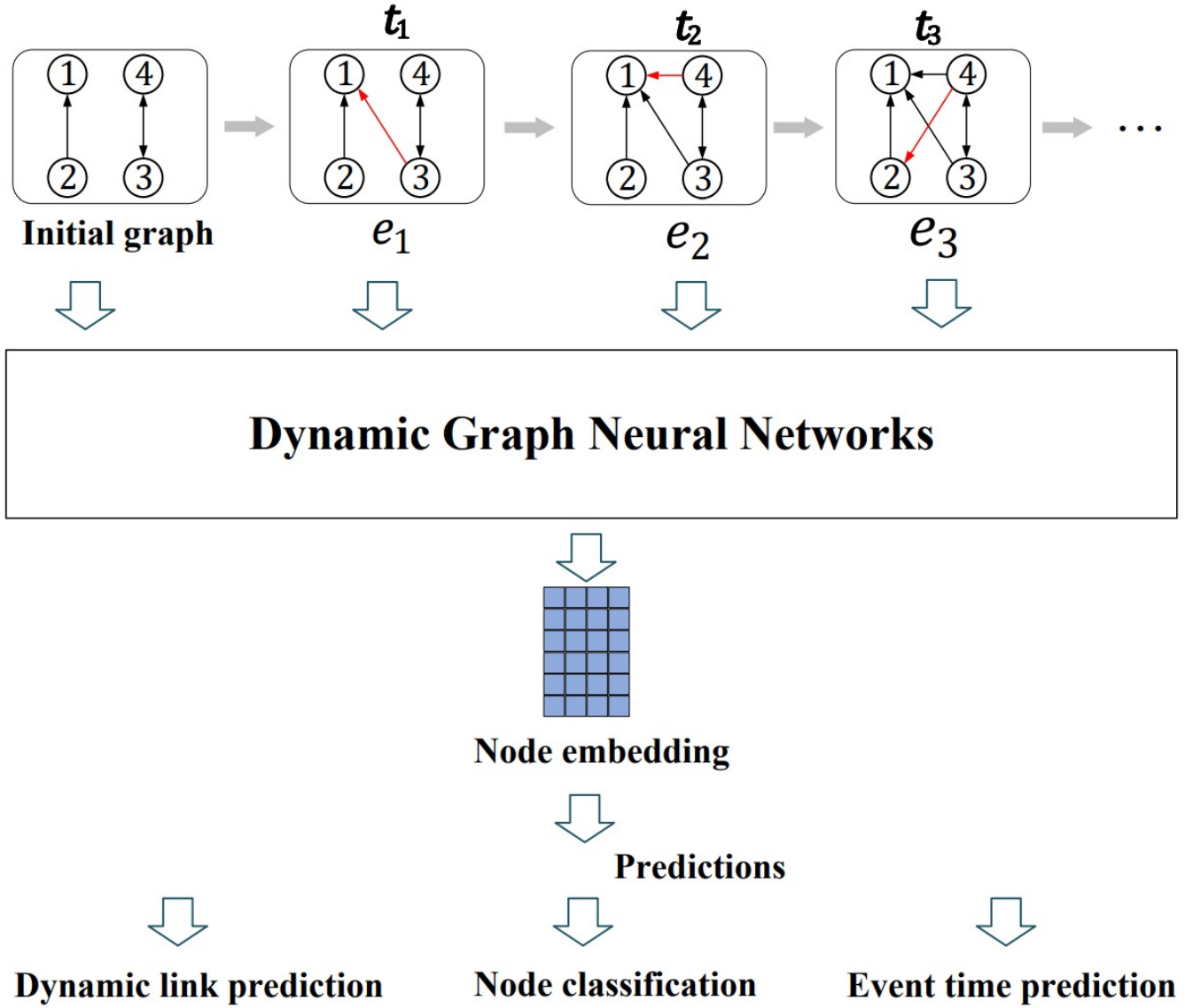
Chaoyi Chen, Dechao Gao, Yanfeng Zhang, Qiange Wang,
Zhenbo Fu, Xuecang Zhang, Junhua Zhu, Yu Gu, Ge Yu

Northeastern University, China
Huawei Technologies Co., Ltd.

VLDB 2024

Dynamic Graph Neural Network

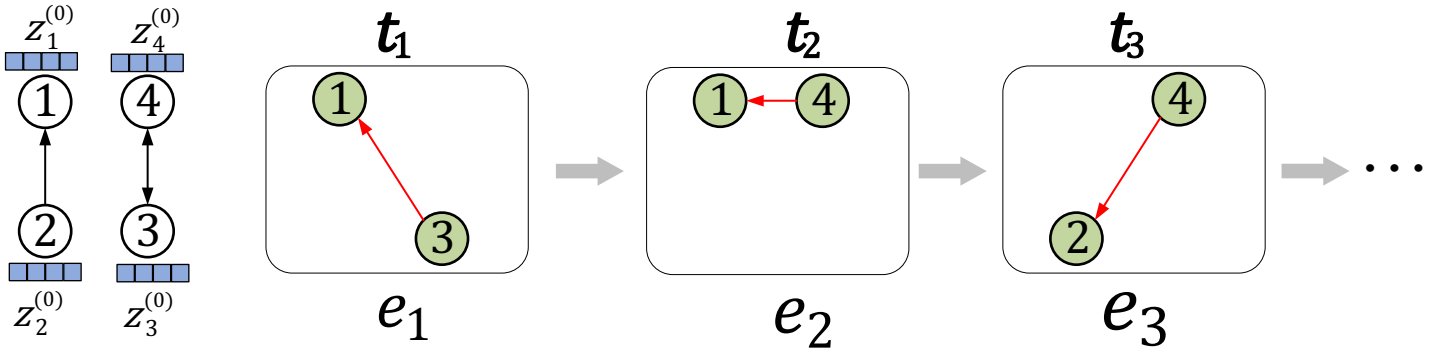
Dynamic graph representation



Execution Pattern of Dynamic GNN

Forward computation (1-event):

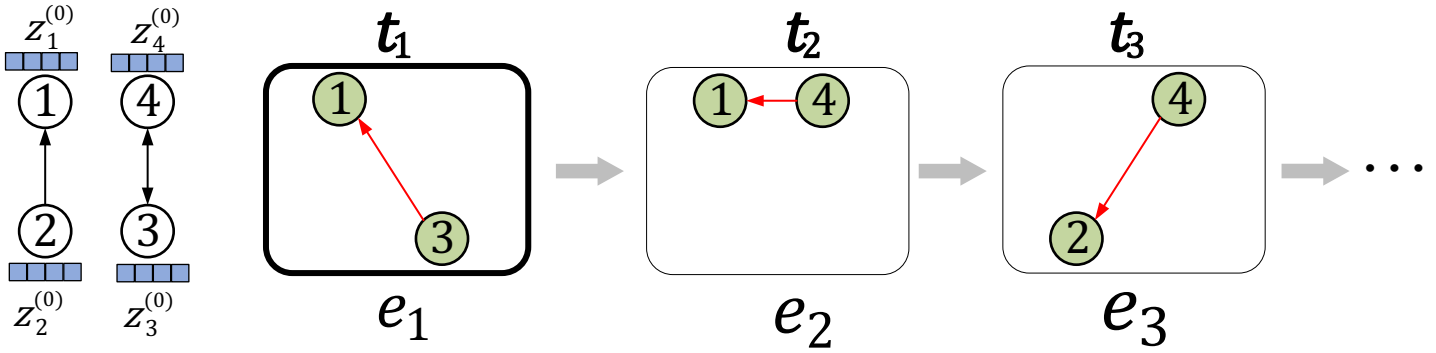
Input data:



Execution Pattern of Dynamic GNN

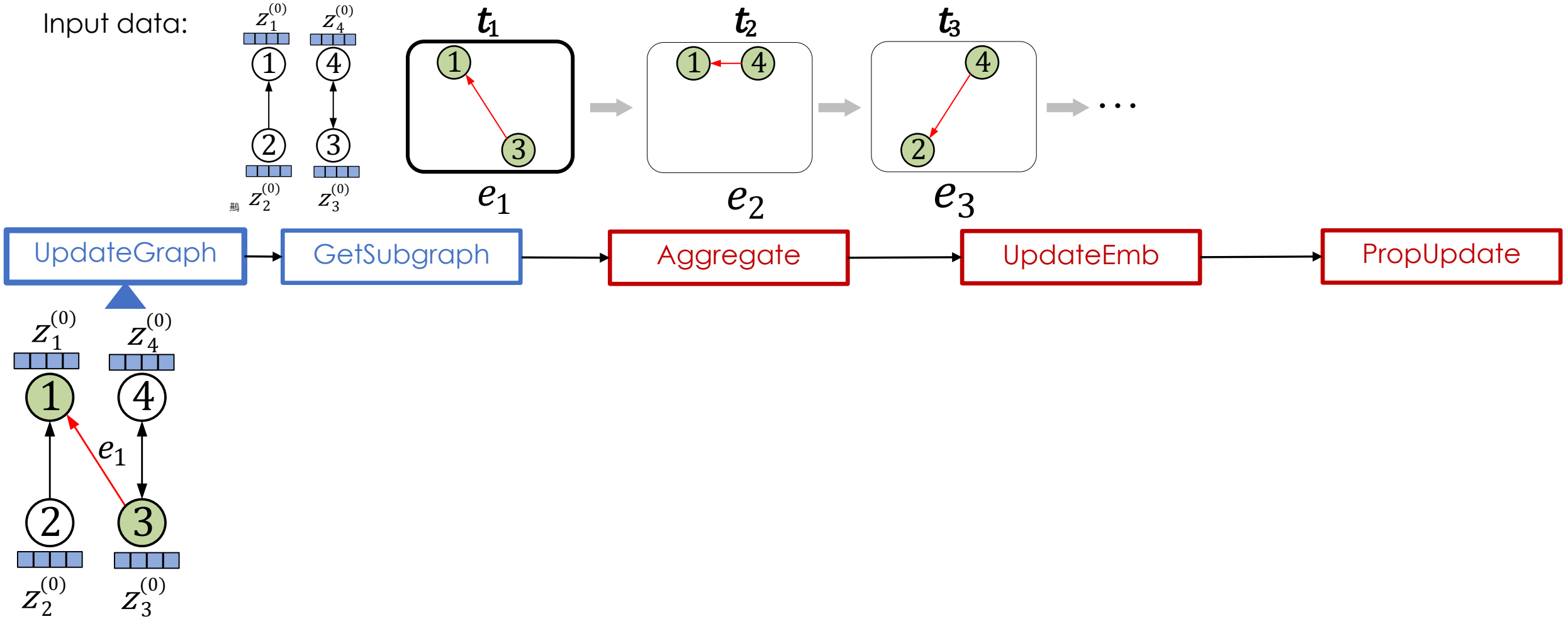
Forward computation (1-event):

Input data:



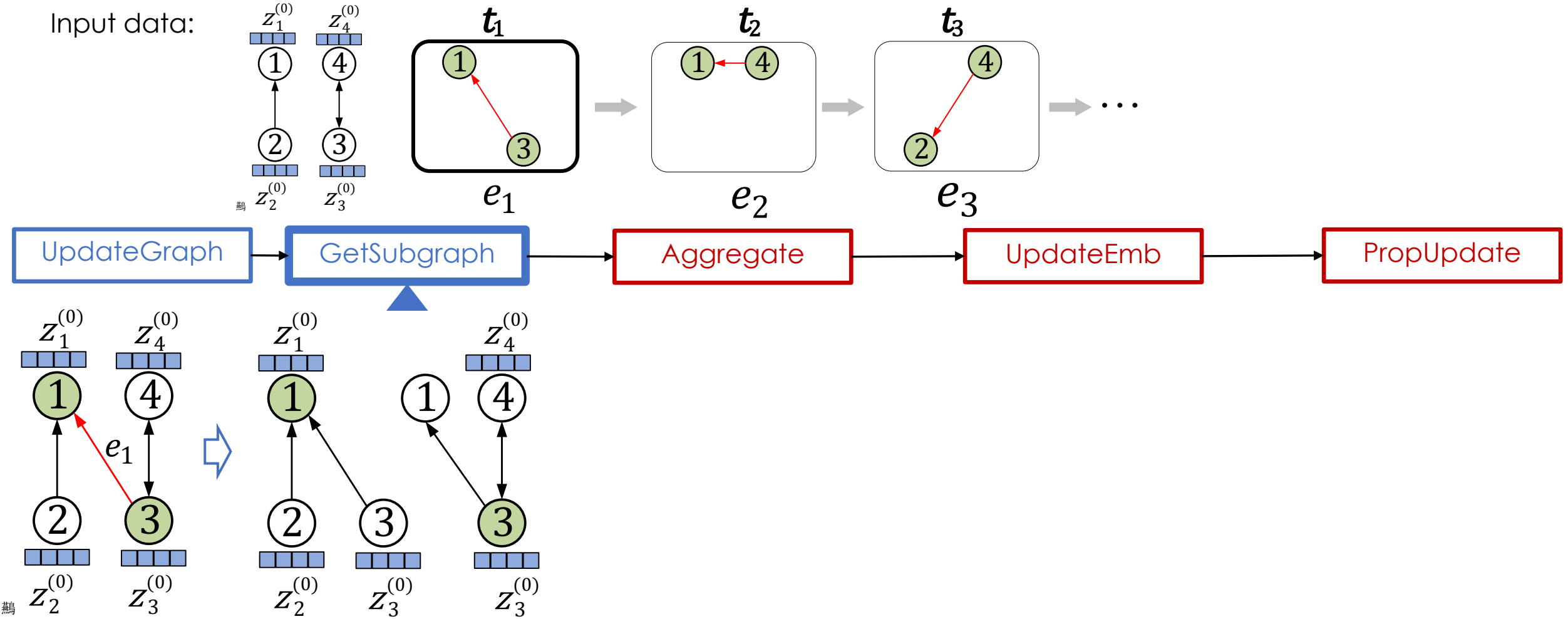
Execution Pattern of Dynamic GNN

Forward computation (1-event):



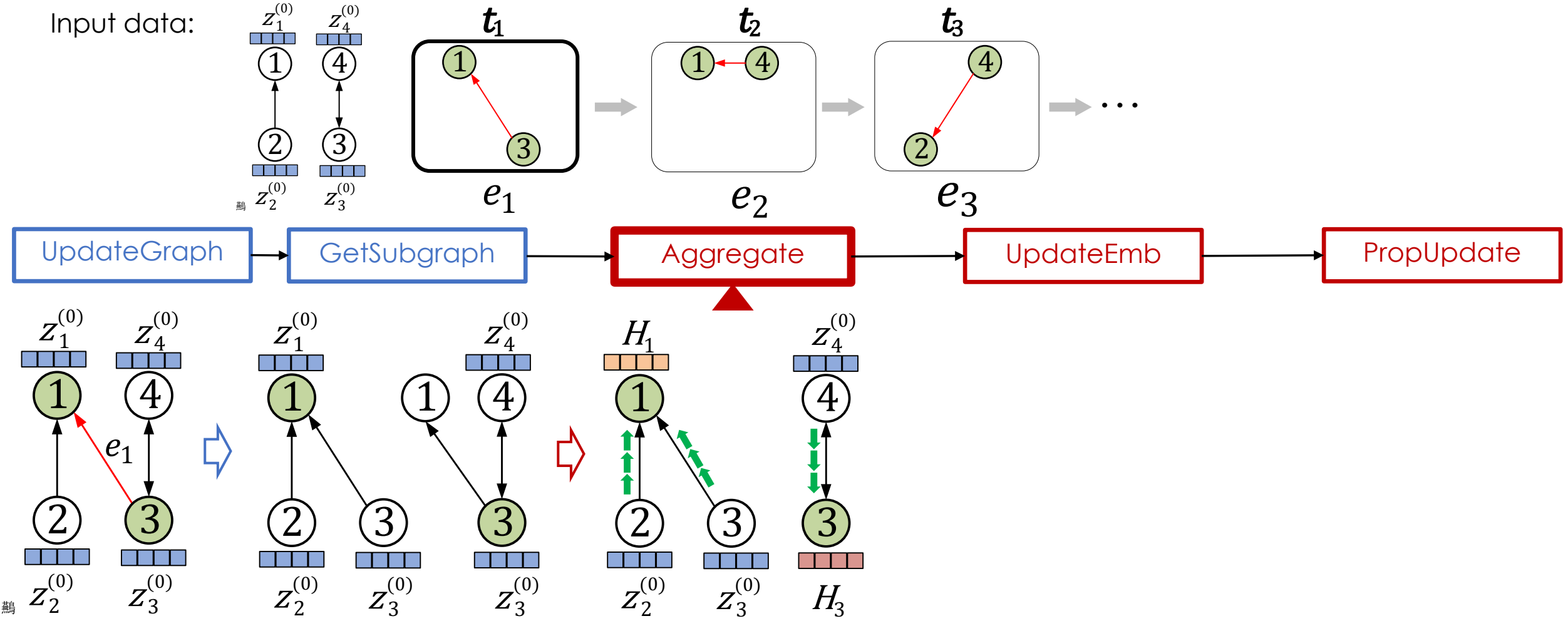
Execution Pattern of Dynamic GNN

Forward computation (1-event):



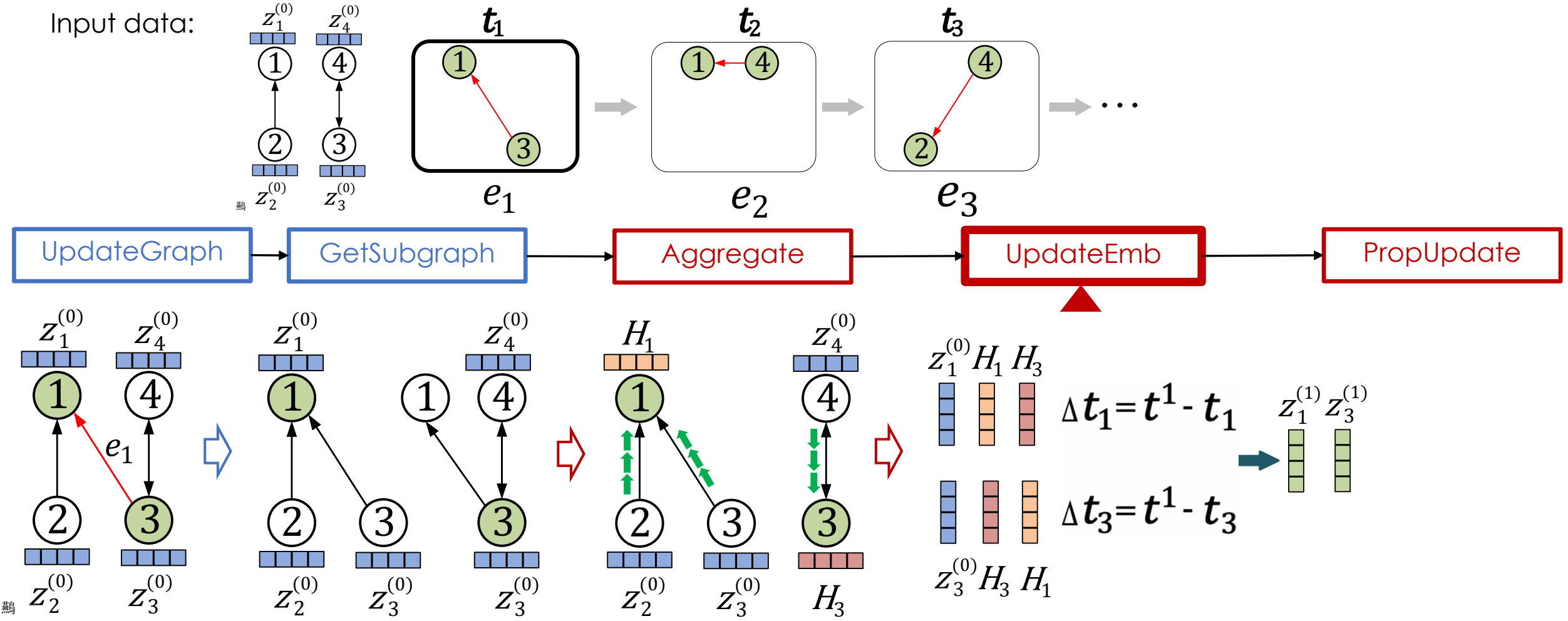
Execution Pattern of Dynamic GNN

Forward computation (1-event):



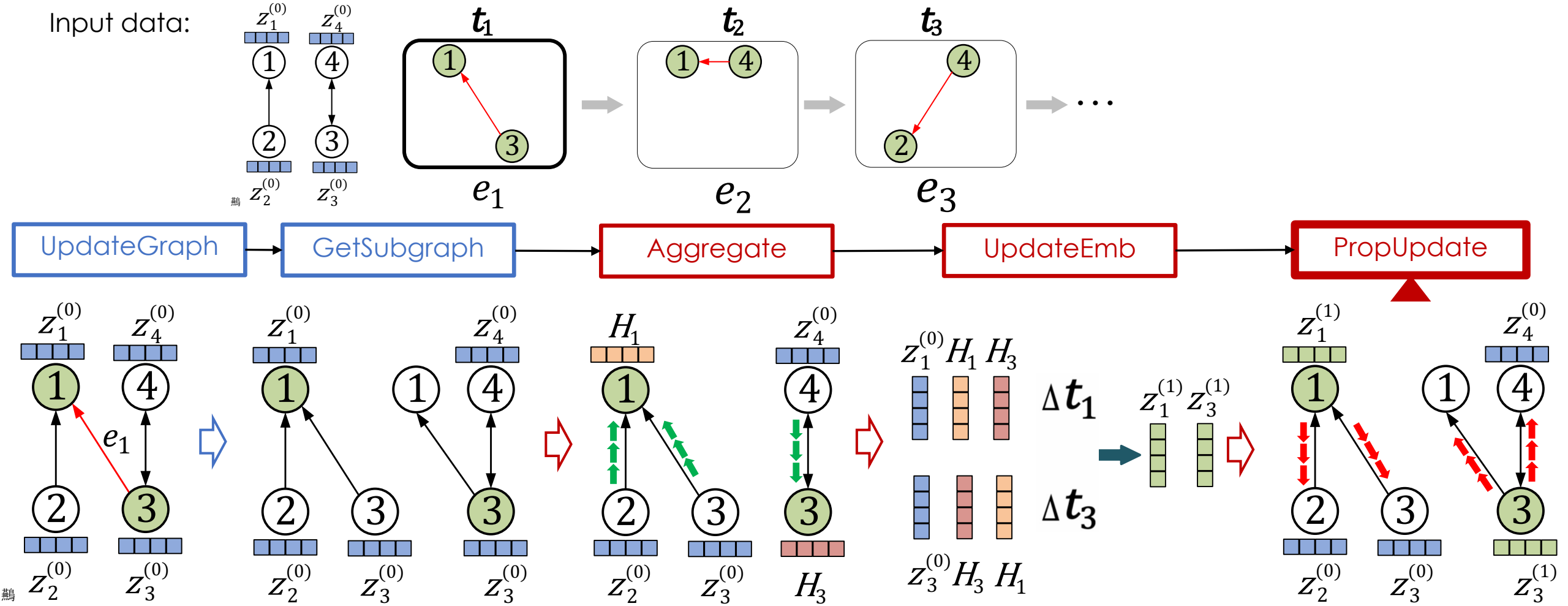
Execution Pattern of Dynamic GNN

Forward computation (1-event):



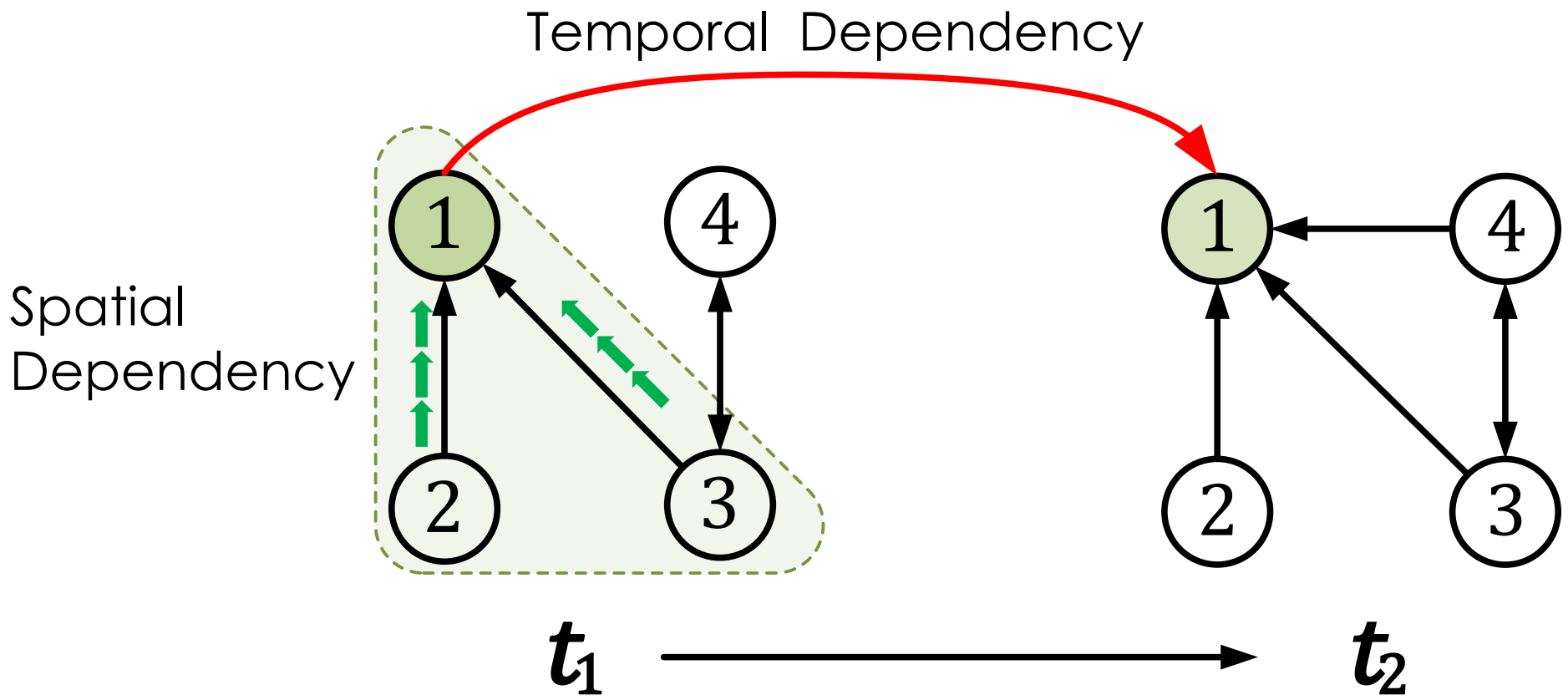
Execution Pattern of Dynamic GNN

Forward computation (1-event):



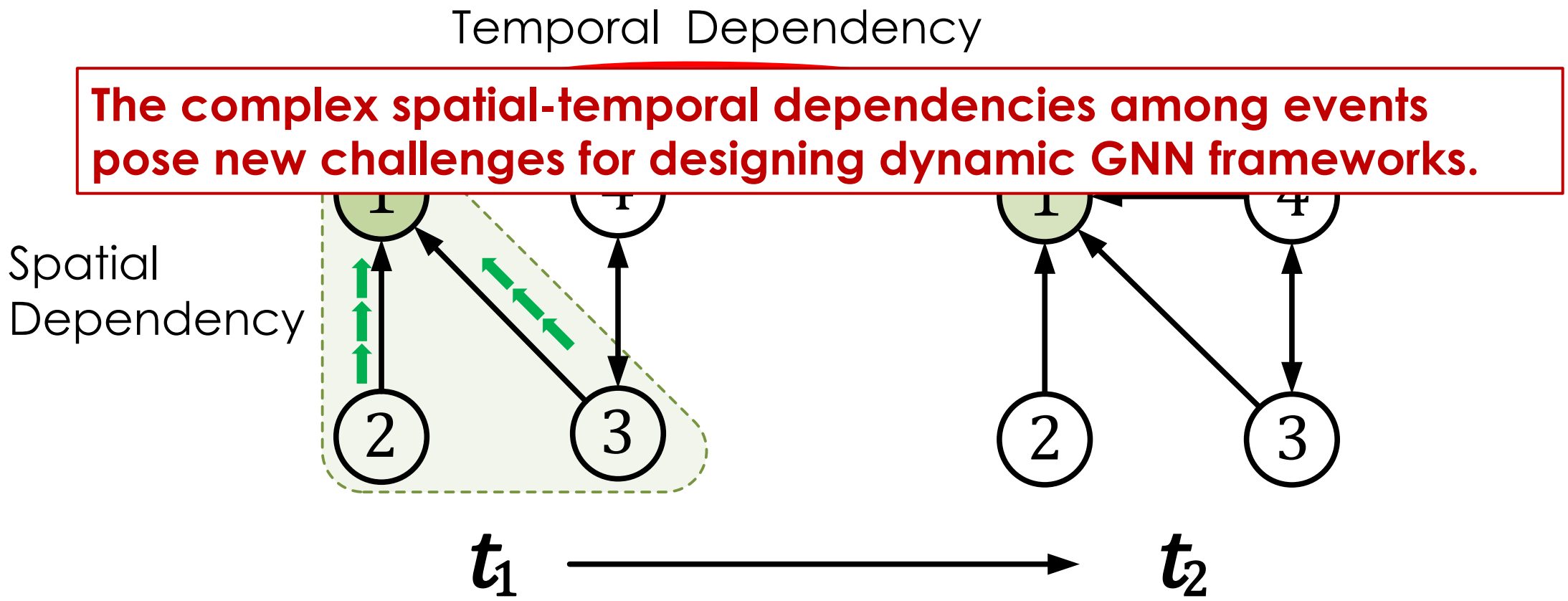
Characteristic: Spatial-Temporal Dependency

- **Spatial Dependency**: a node status depends on its neighbors
- **Temporal Dependency**: a node status depends on its previous status



Characteristic: Spatial-Temporal Dependency

- **Spatial Dependency**: a node status depends on its neighbors
- **Temporal Dependency**: a node status depends on its previous status

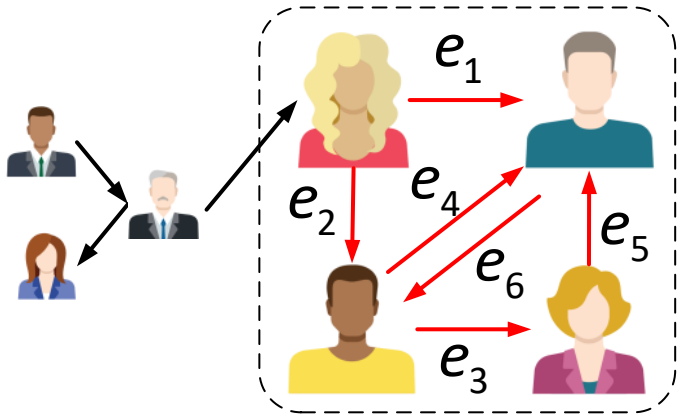


Challenge 1: The training method needs to capture real-time structural information

Key Observations : Events in the same area should update parameters collaboratively to better capture spatial dependencies

Update event stream

e_1 e_2 e_3 e_4 e_5 e_6 ...

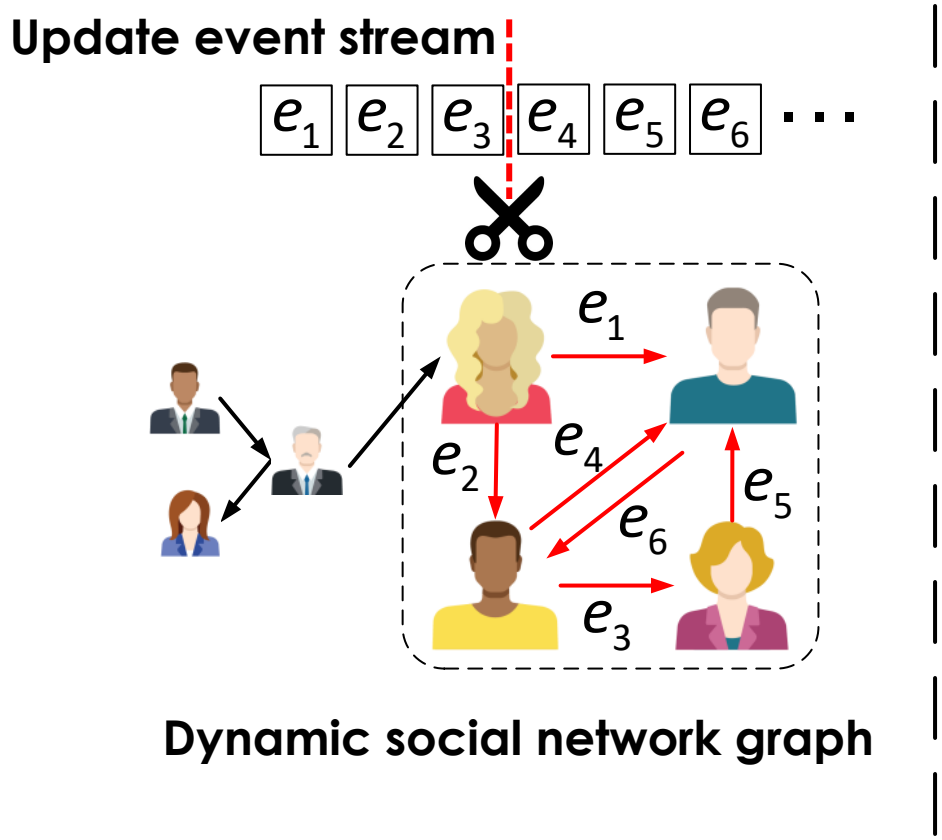


Dynamic social network graph

Six events are concentrated in one area

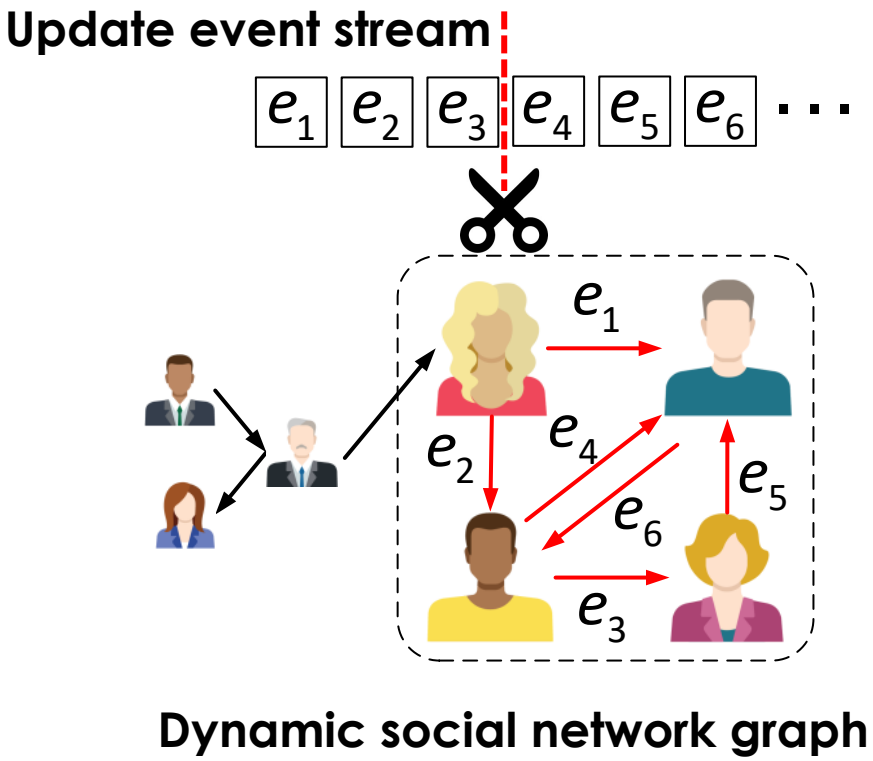
Challenge 1: The training method needs to capture real-time structural information

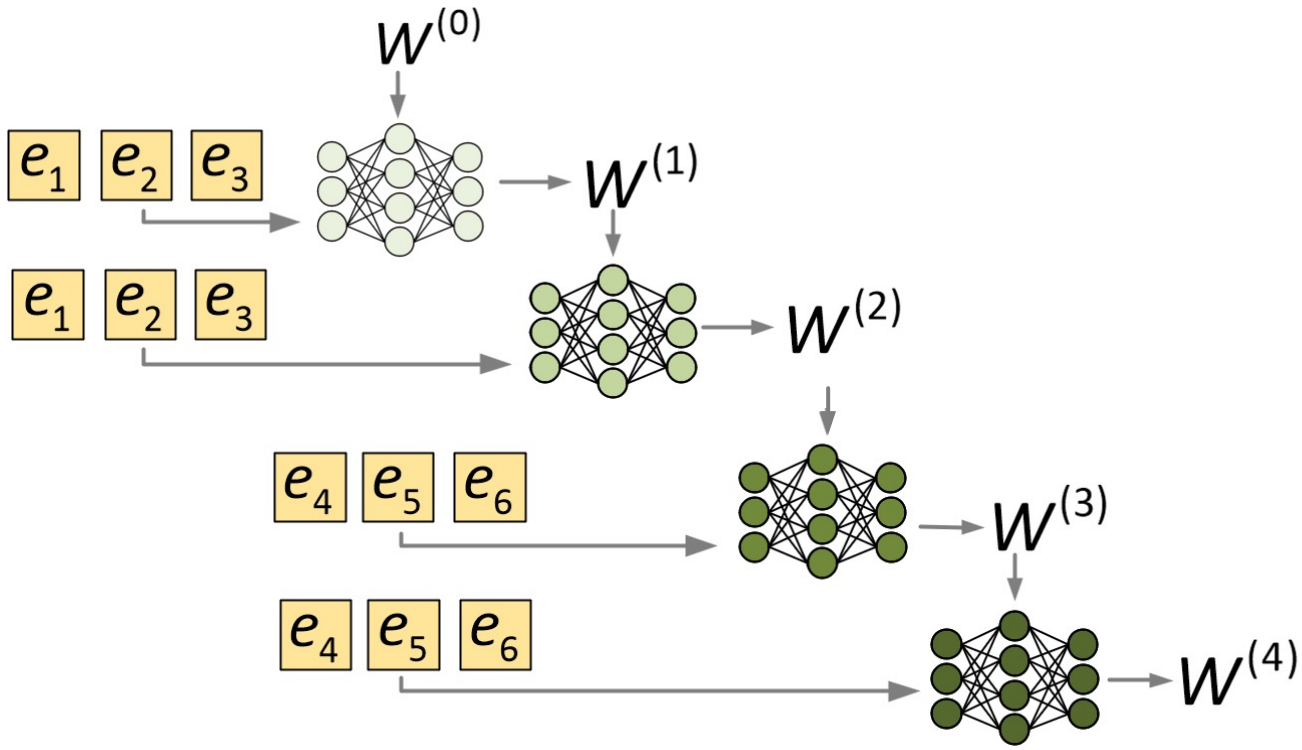
The traditional batched training method forcibly cuts off the event stream and ignores the spatial locality between events



Challenge 1: The training method needs to capture real-time structural information

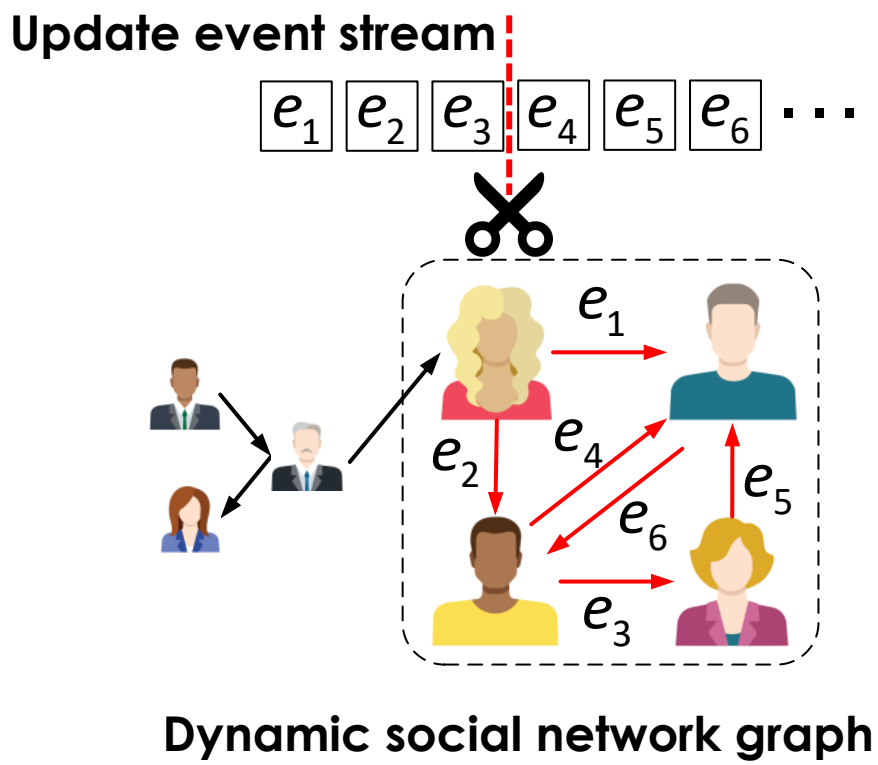
The traditional batched training method forcibly cuts off the event stream and ignores the spatial locality between events

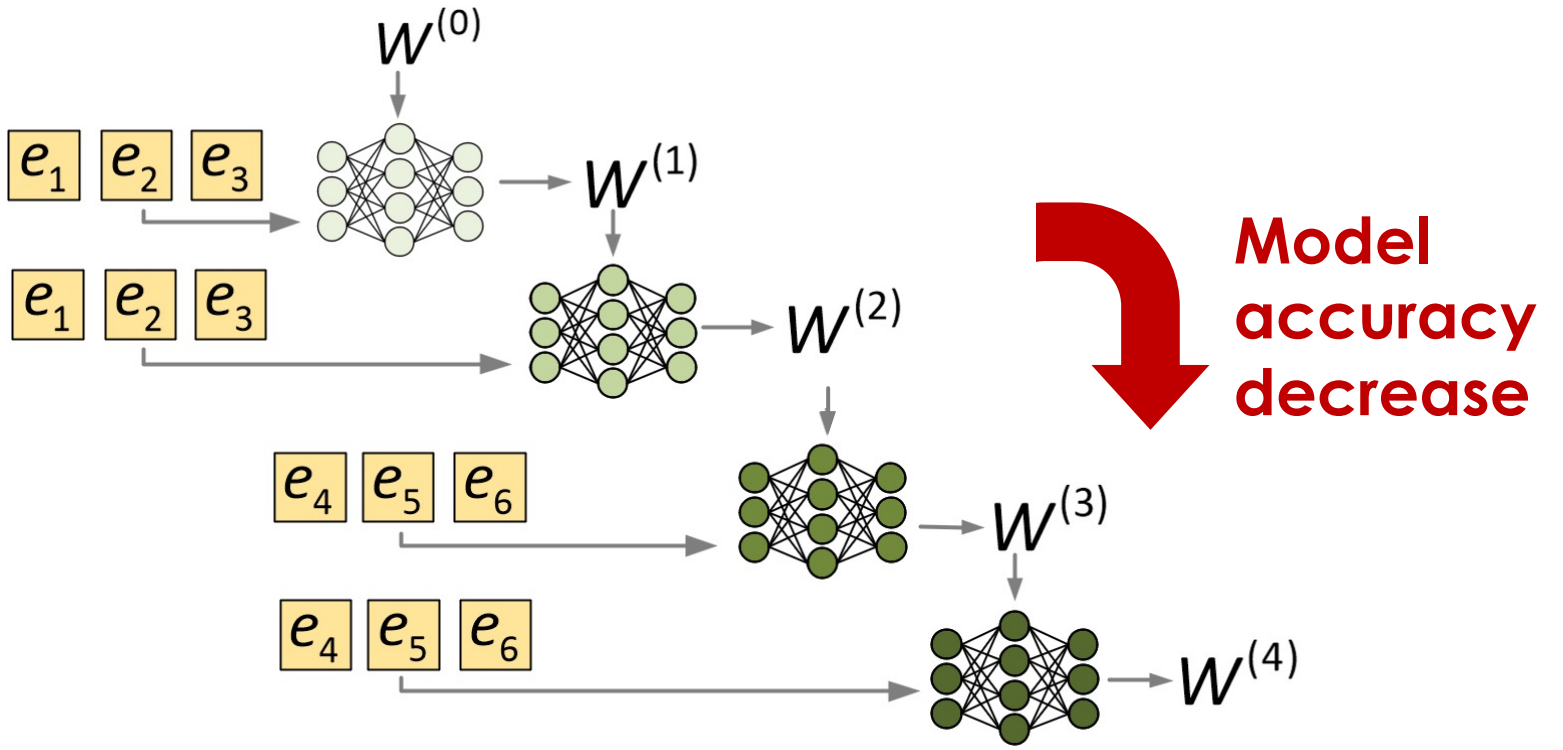




Challenge 1: The training method needs to capture real-time structural information

The traditional batched training method forcibly cuts off the event stream and ignores the spatial locality between events





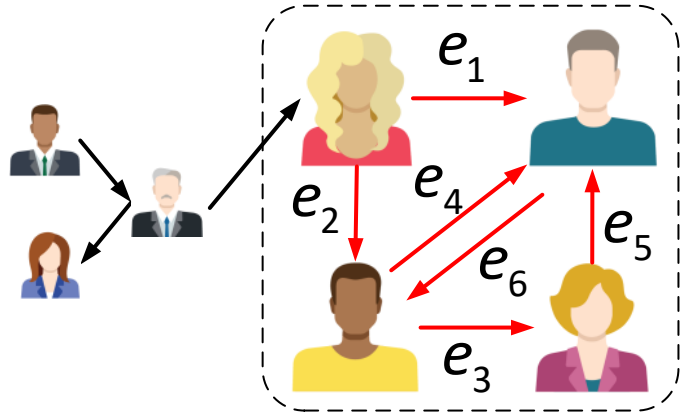
Challenge 2: The temporal dependency nature makes parallelism optimization hard to design

- **Temporal Dependency: a node status depends on its previous status**

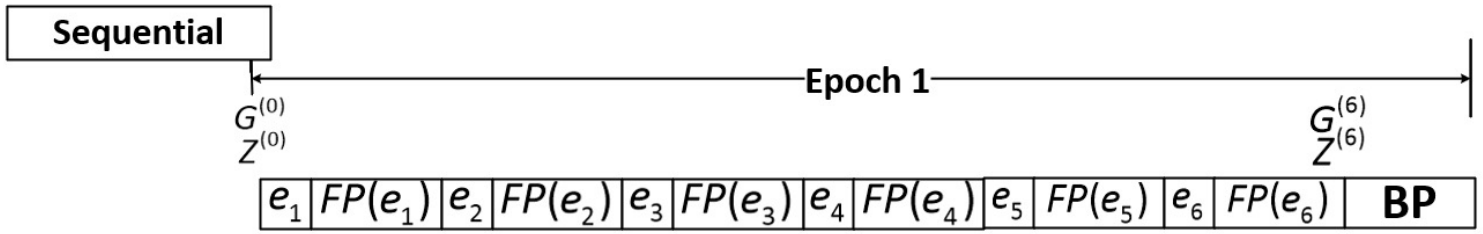
Existing frameworks sequential iterative processing

Update event stream

e_1 e_2 e_3 e_4 e_5 e_6 ...



Dynamic social network graph



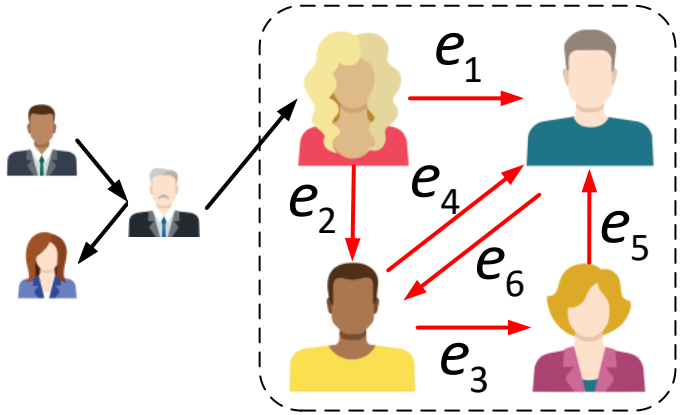
Challenge 2: The temporal dependency nature makes parallelism optimization hard to design

- **Temporal Dependency: a node status depends on its previous status**

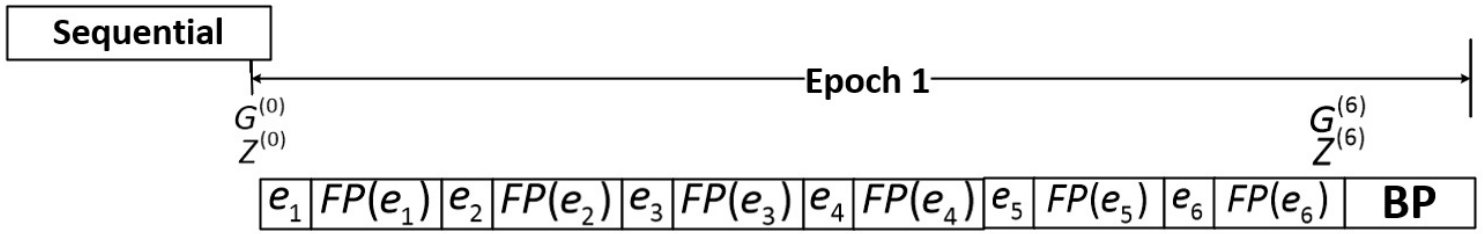
Existing frameworks sequential iterative processing

Update event stream

e_1 e_2 e_3 e_4 e_5 e_6 ...



Dynamic social network graph

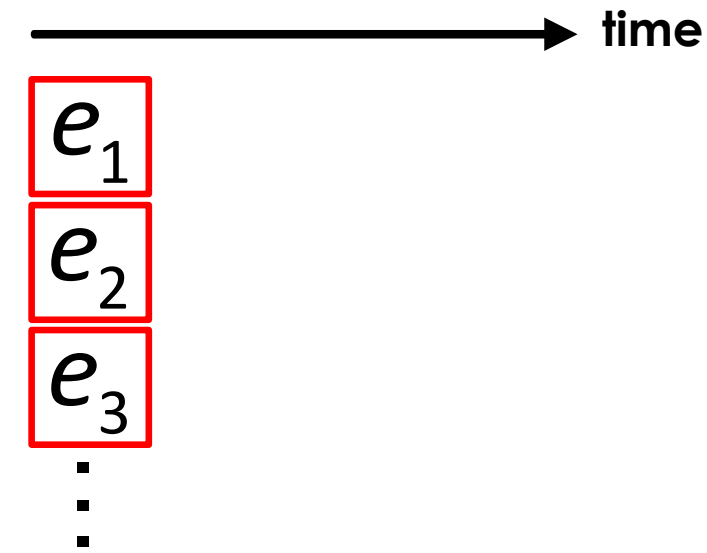


DyRep on GitHub dataset

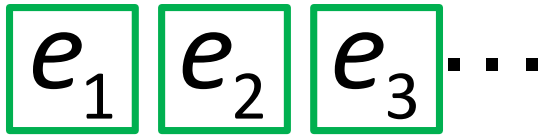
- **Single thread for an epoch : 60.45s**
- **20 threads for an epoch : 60.89s**

Challenge 2: The temporal dependency nature makes parallelism optimization hard to design

- **Without Temporal Dependency** : Ignore time, execute events in parallel
- **With Temporal Dependency** : Keep time, execute events chronologically



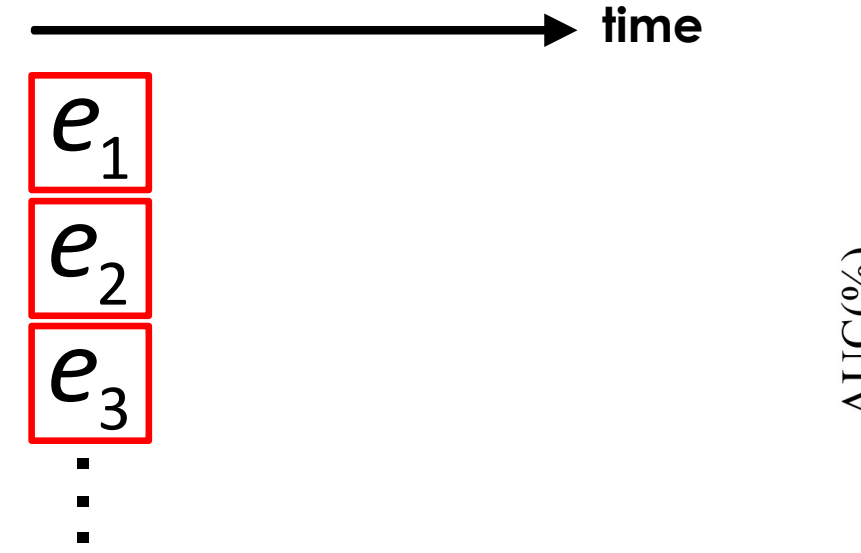
(a) Without Temporal Dependency



(b) With Temporal Dependency

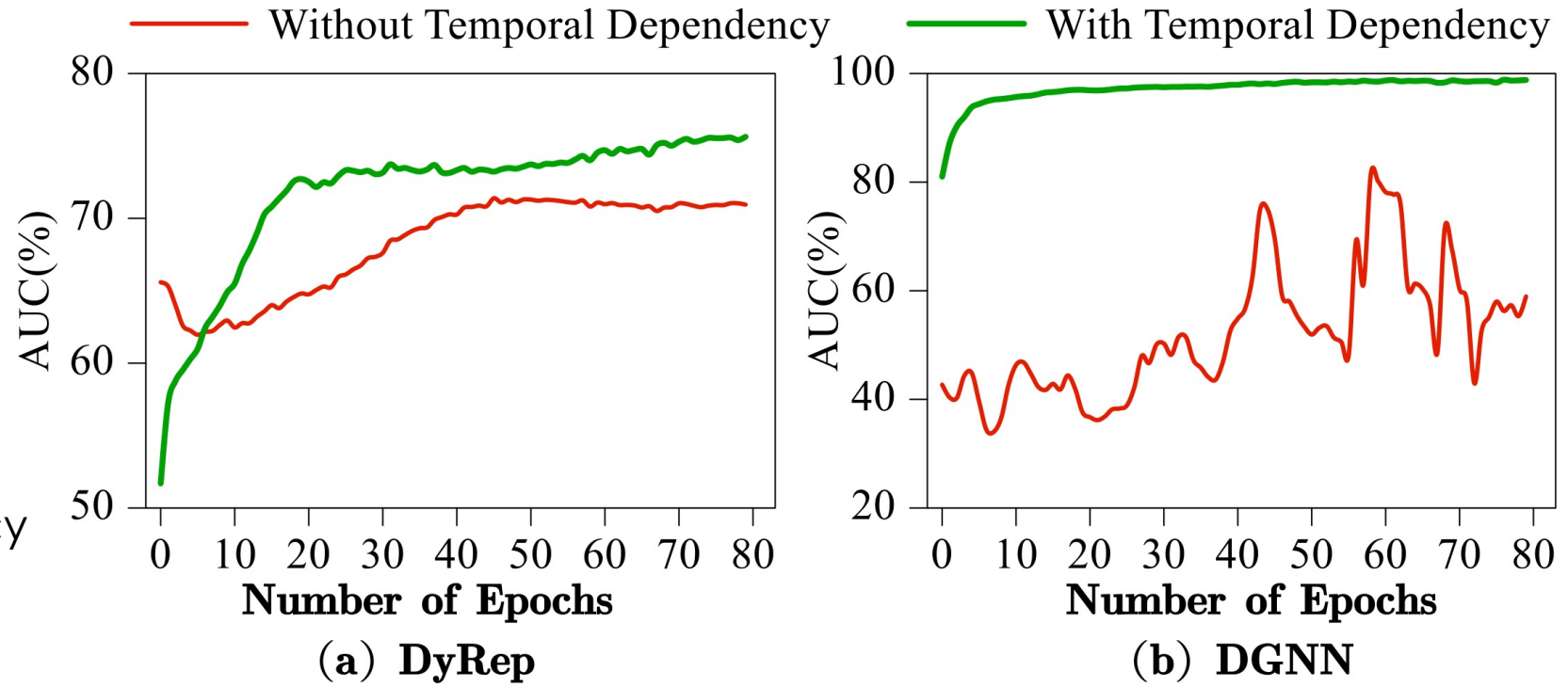
Challenge 2: The temporal dependency nature makes parallelism optimization hard to design

- **Without Temporal Dependency** : Ignore time, execute events in parallel
- **With Temporal Dependency** : Keep time, execute events chronologically



(a) Without Temporal Dependency

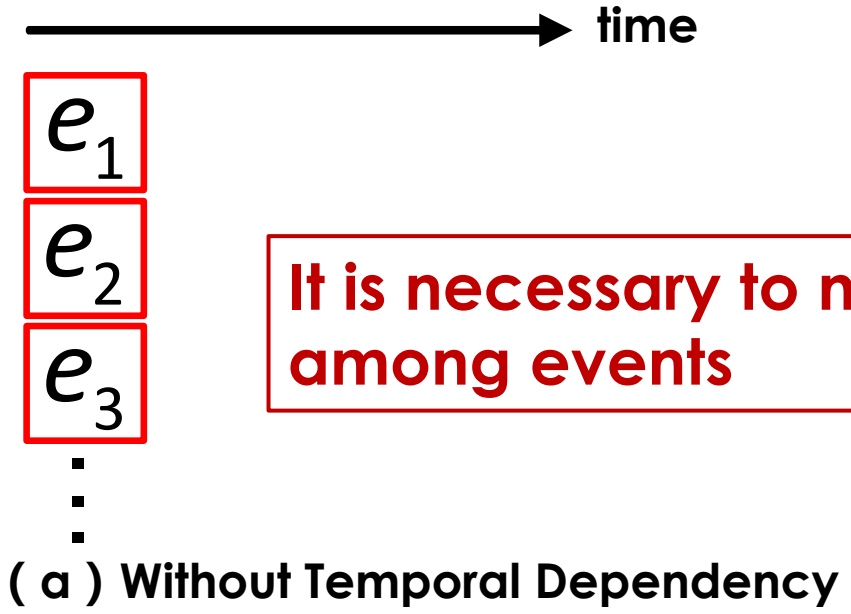
(b) With Temporal Dependency



AUC Comparison with dependency and without dependency

Challenge 2: The temporal dependency nature makes parallelism optimization hard to design

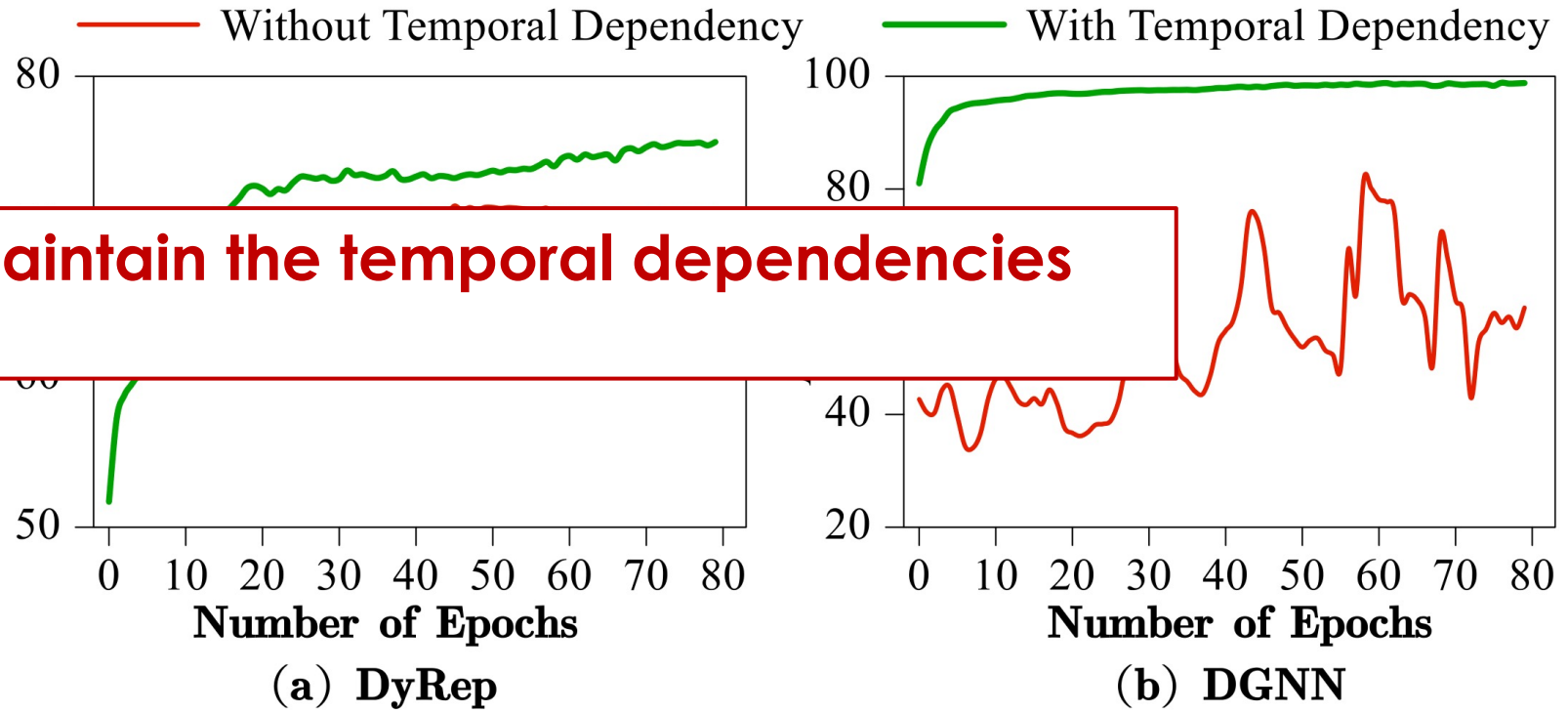
- **Without Temporal Dependency** : Ignore time, execute events in parallel
- **With Temporal Dependency** : keep time, execute events chronologically



(a) Without Temporal Dependency

(b) With Temporal Dependency

It is necessary to maintain the temporal dependencies among events



AUC Comparison with dependency and without dependency

Summary

- **Challenge 1:** The traditional batched training mode fail to capture the real-time structural evolution information

Model accuracy decrease

- **Challenge 2:** The temporal dependency nature makes parallelism optimization hard to design

Training performance decrease

Our Contributions

We propose NeutronStream, a high-performance Dynamic GNN framework

- **Solution 1:** Propose a new incremental learning mode with a sliding window for training on graph streams → Challenge 1 → Model accuracy improve

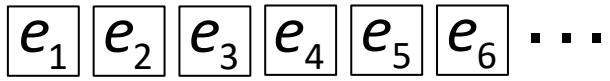
- **Solution 2:** Propose a fine-grained event parallel execution scheme → Challenge 2 → Training performance improve

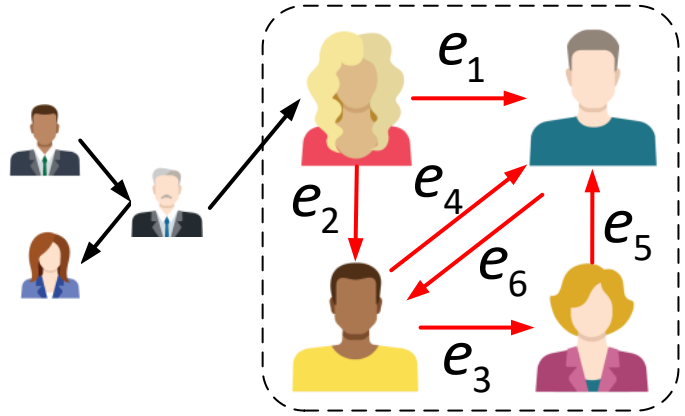
Solution 1: Sliding-window-based Training Method

Incremental Mode with Sliding Window (denoted as **NS-Slide)**

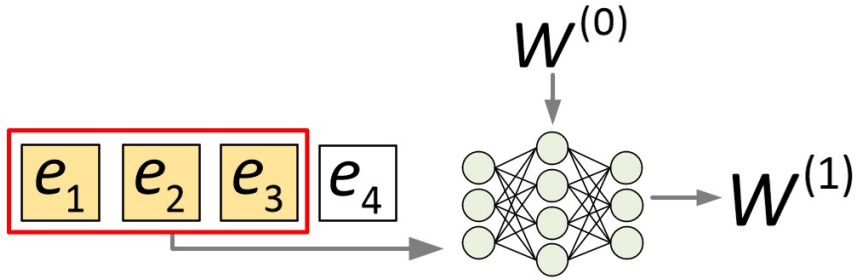
We design a sliding window to select consecutive events from the input event stream

Update event stream





Dynamic social network graph

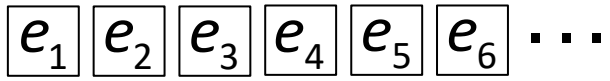


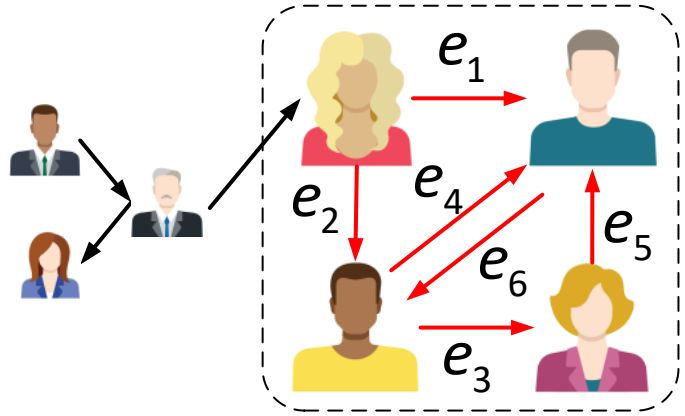
Solution 1: Sliding-window-based Training Method

Incremental Mode with Sliding Window (denoted as **NS-Slide)**

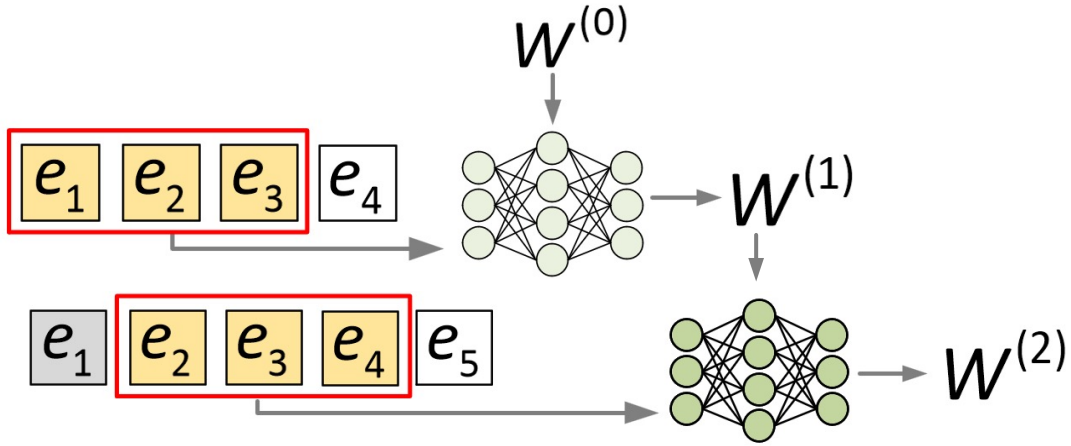
We design a sliding window to select consecutive events from the input event stream

Update event stream





Dynamic social network graph

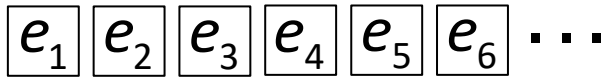


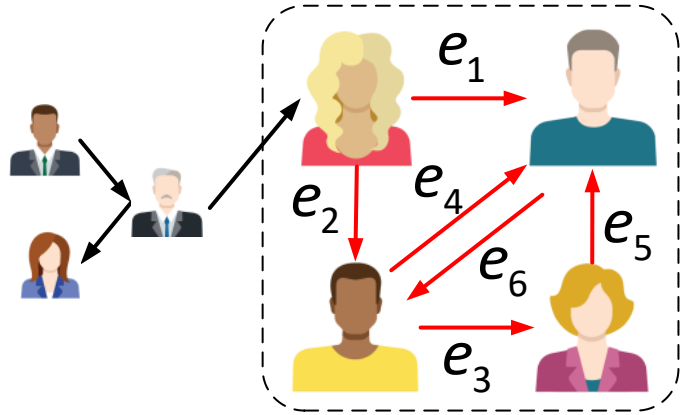
Solution 1: Sliding-window-based Training Method

Incremental Mode with Sliding Window (denoted as **NS-Slide)**

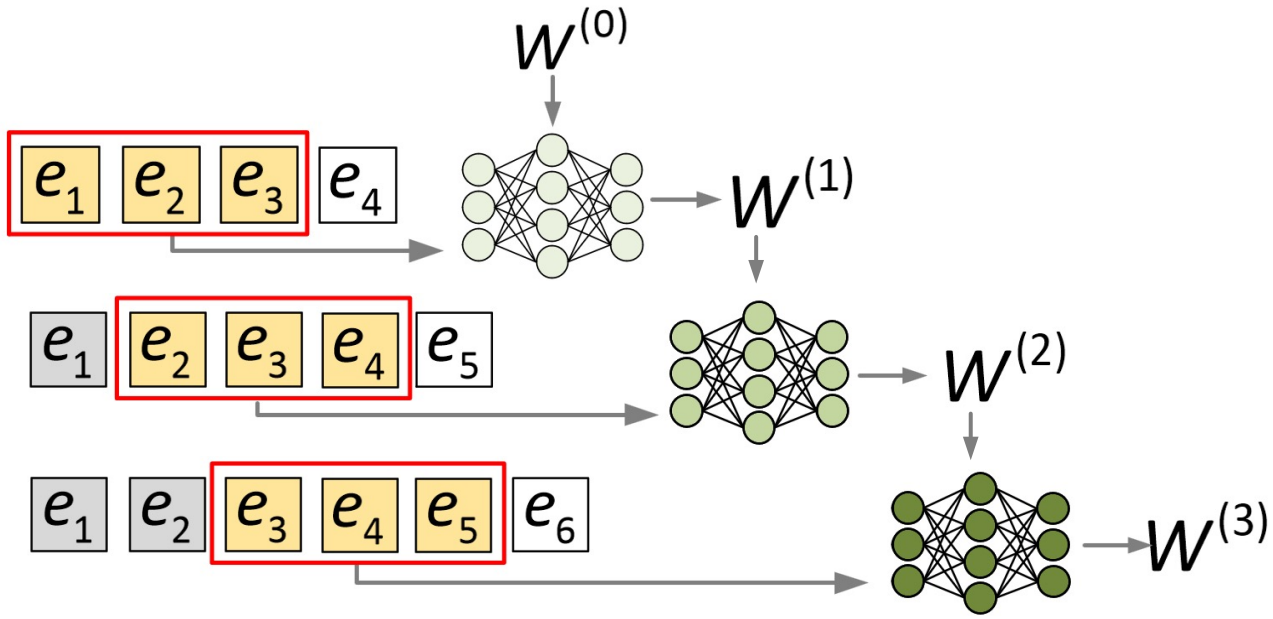
We design a sliding window to select consecutive events from the input event stream

Update event stream





Dynamic social network graph

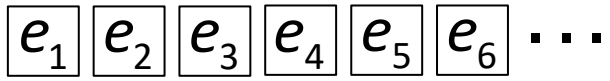


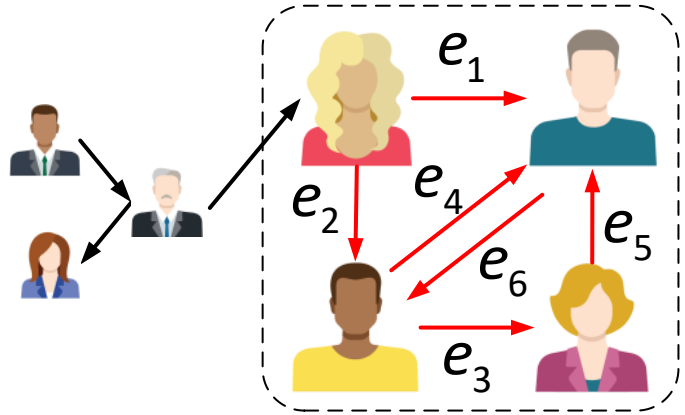
Solution 1: Sliding-window-based Training Method

Incremental Mode with Sliding Window (denoted as **NS-Slide)**

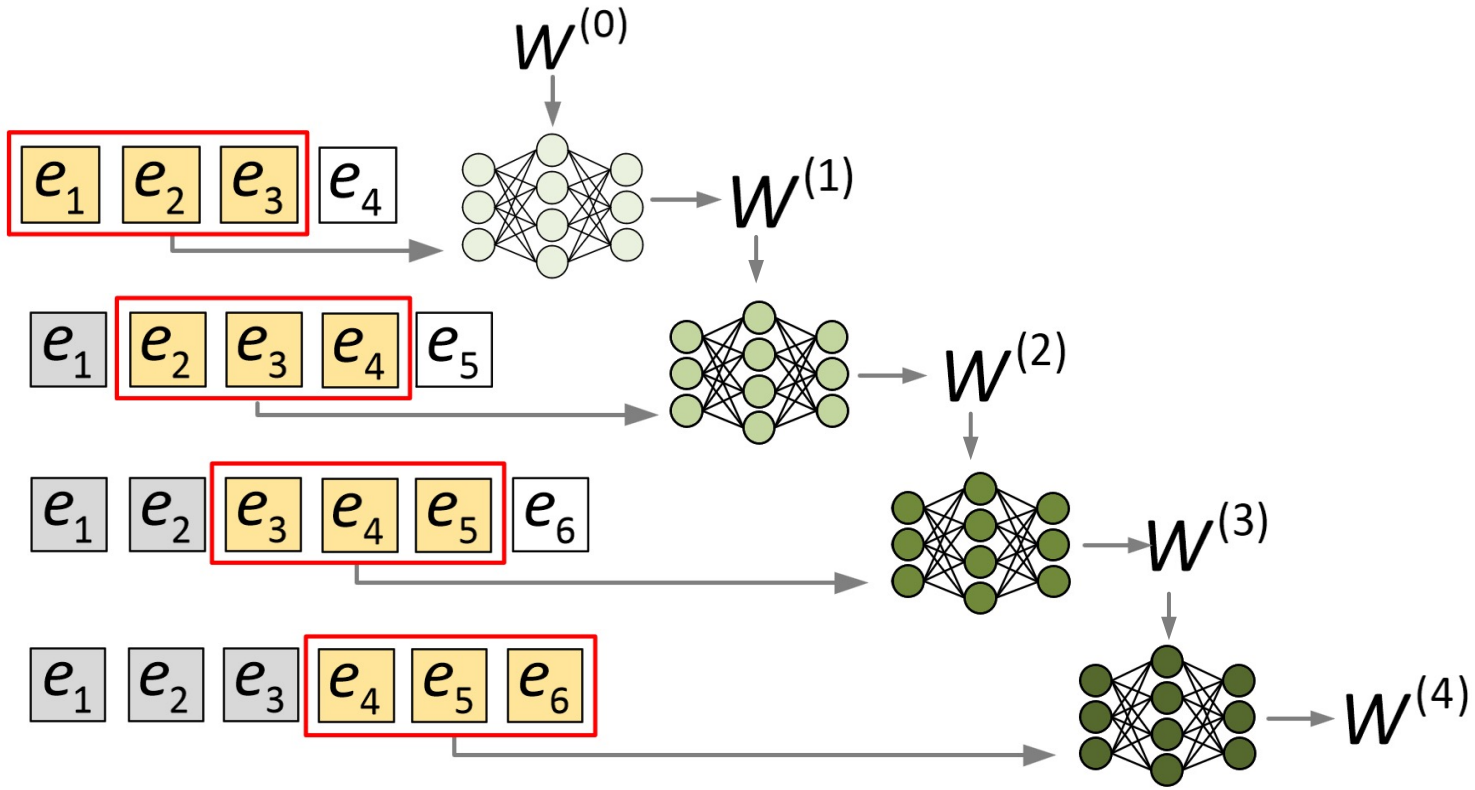
We design a sliding window to select consecutive events from the input event stream

Update event stream





Dynamic social network graph

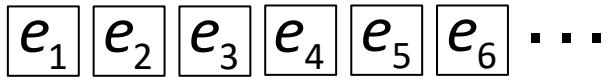


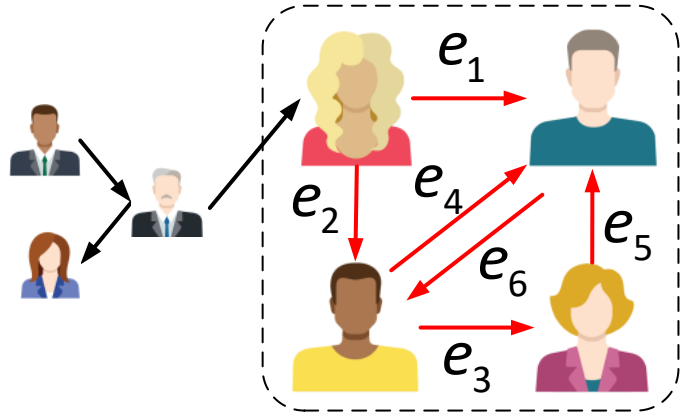
Solution 1: Sliding-window-based Training Method

Incremental Mode with Sliding Window (denoted as **NS-Slide)**

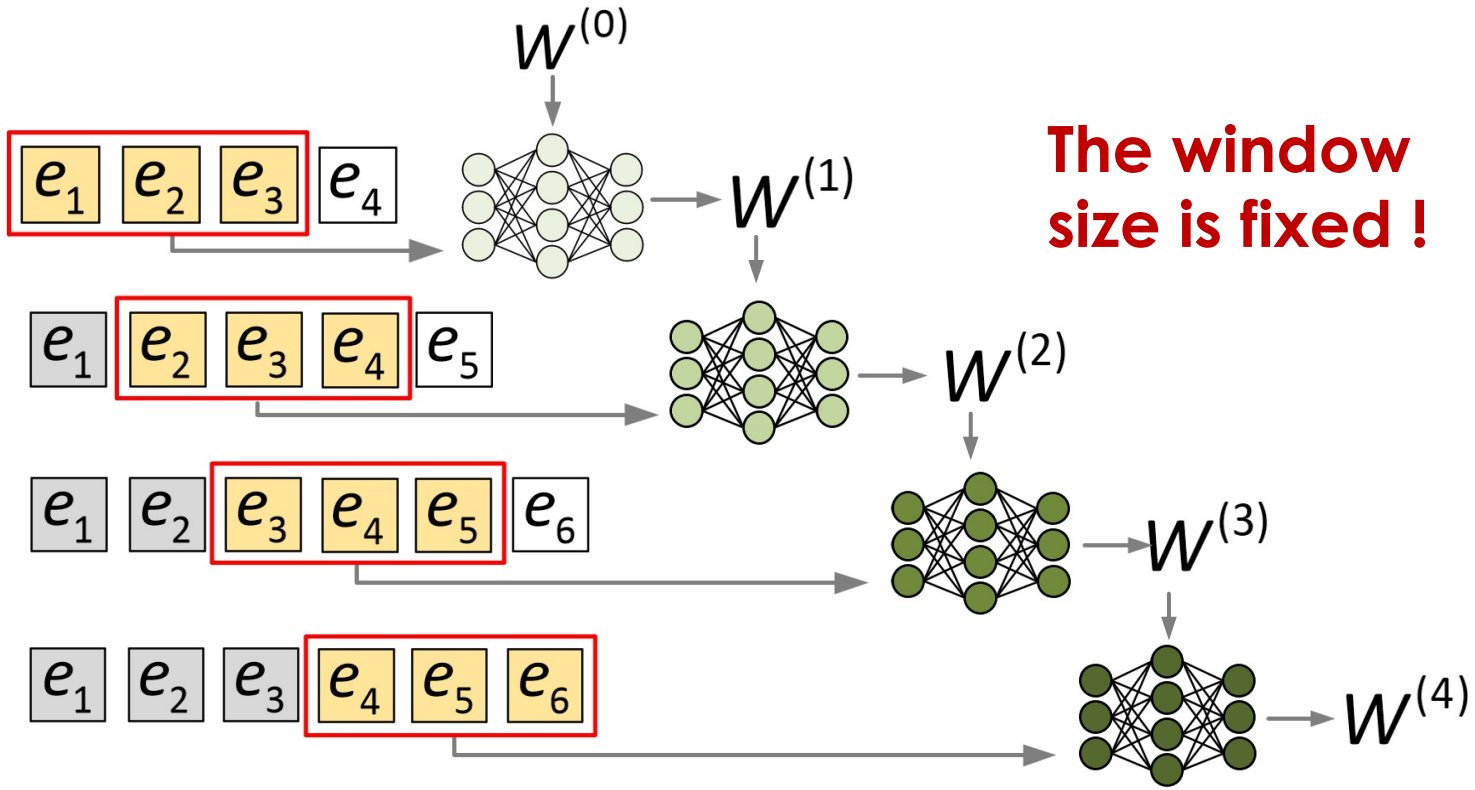
We design a sliding window to select consecutive events from the input event stream

Update event stream





Dynamic social network graph

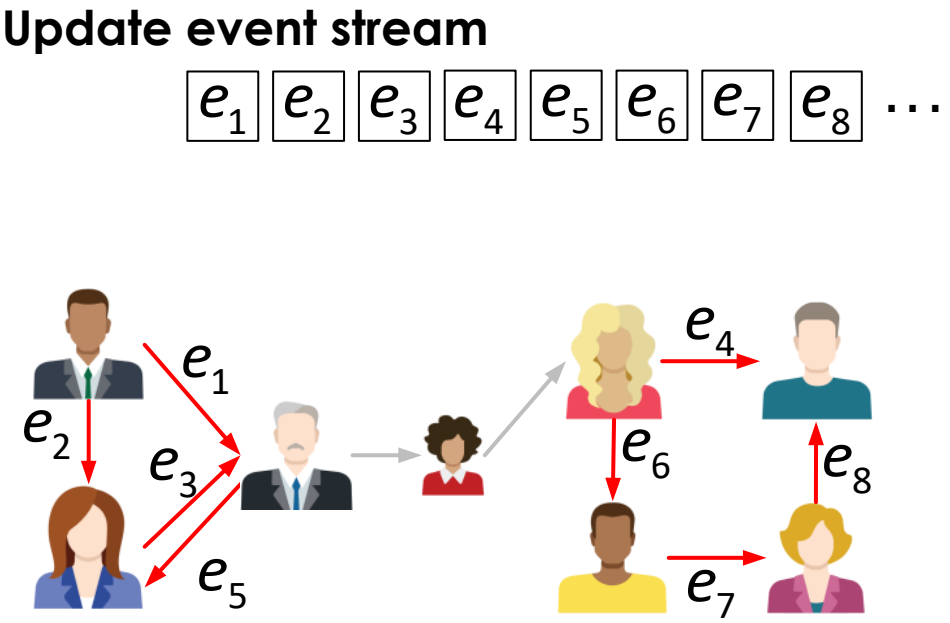


The window size is fixed !

Solution 1: Sliding-window-based Training Method

The window size determines how much data can be fed into the model

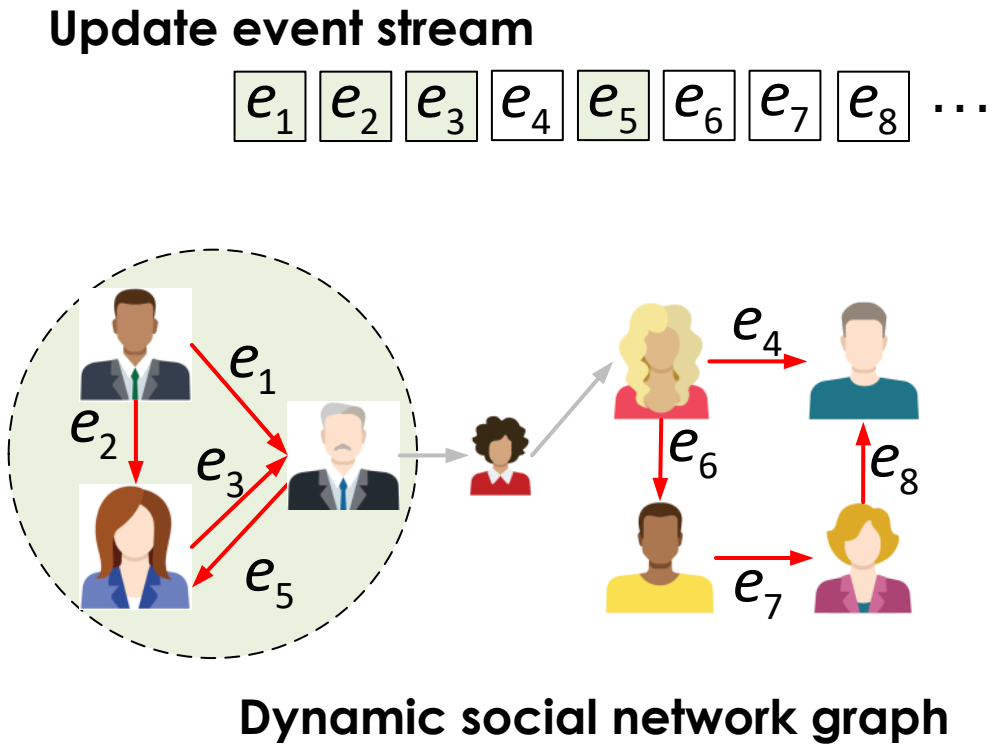
Key Observations : A continuous segment of events is usually concentrated in a local area in the graph



Solution 1: Sliding-window-based Training Method

The window size determines how much data can be fed into the model

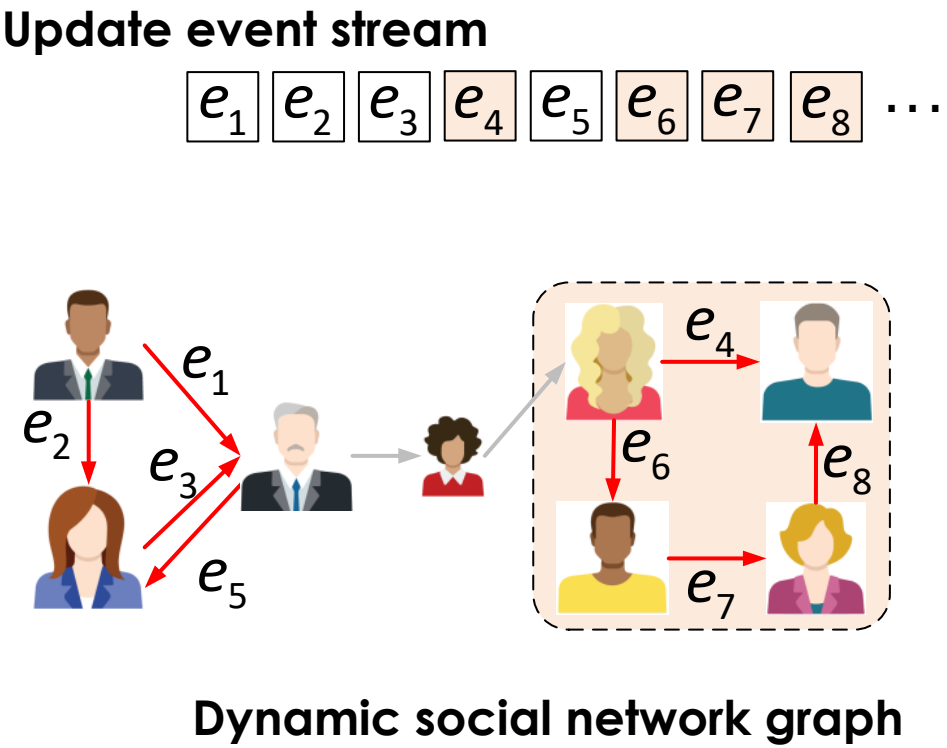
Key Observations : A continuous segment of events is usually concentrated in a local area in the graph



Solution 1: Sliding-window-based Training Method

The window size determines how much data can be fed into the model

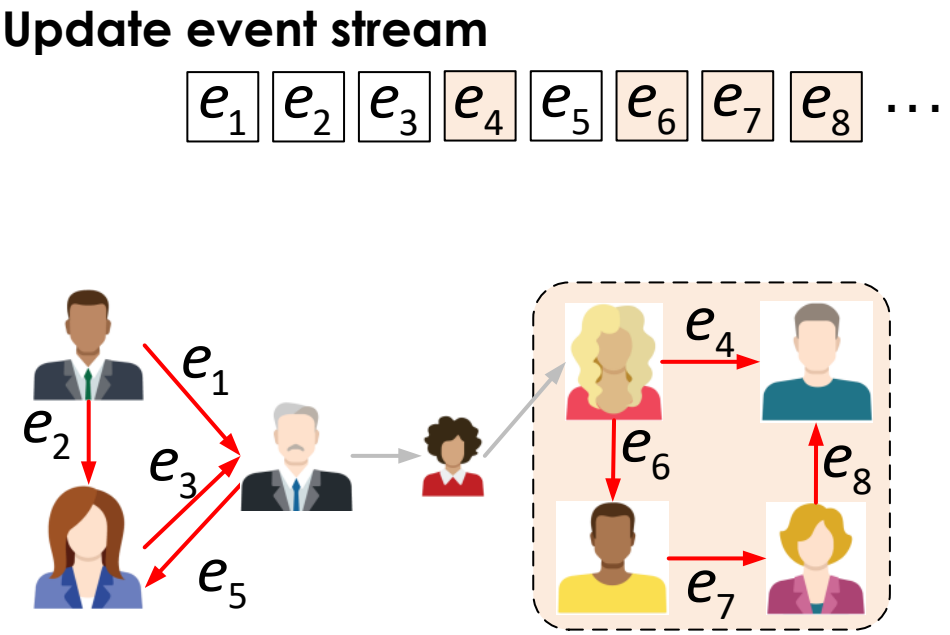
Key Observations : A continuous segment of events is usually concentrated in a local area in the graph



Solution 1: Sliding-window-based Training Method

The window size determines how much data can be fed into the model

Key Observations : A continuous segment of events is usually concentrated in a local area in the graph



The window size should be dynamically adjustable

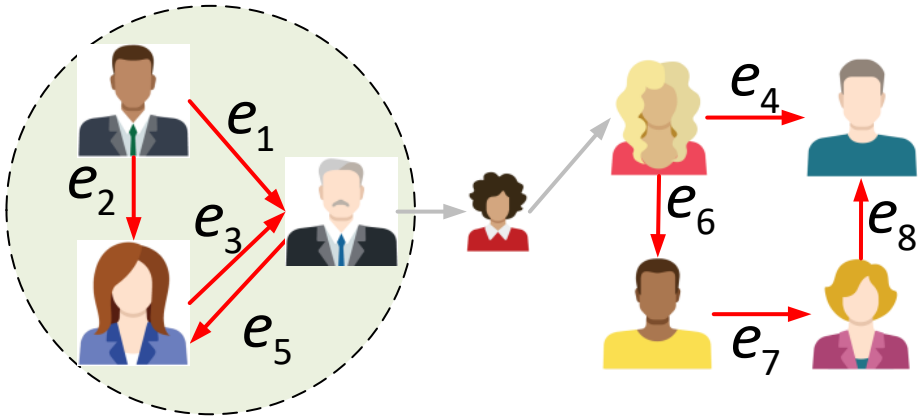
Solution 1: Sliding-window-based Training Method

Adaptive Adjustment of Window Size (denoted as **NS-AdaSlide)**

The window size can be adaptively adjusted for capturing the **complete spatial dependencies**

Update event stream

e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 …



Dynamic social network graph

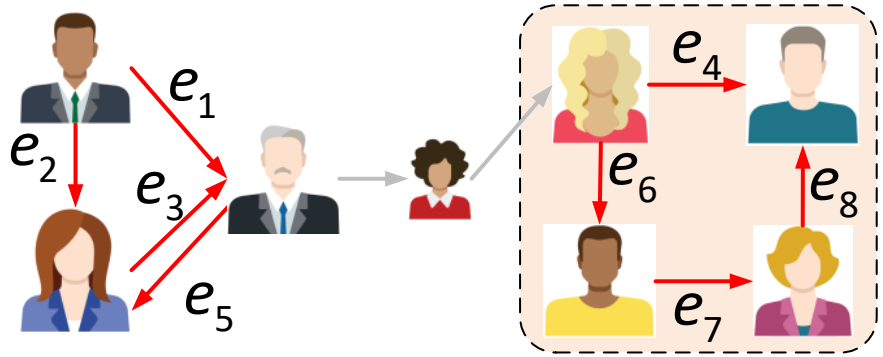
e_1 e_2 e_3 e_4 e_5

Solution 1: Sliding-window-based Training Method

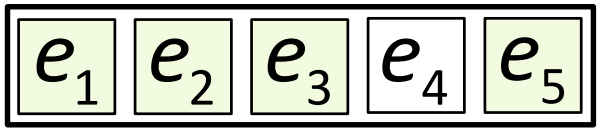
Adaptive Adjustment of Window Size (denoted as **NS-AdaSlide)**

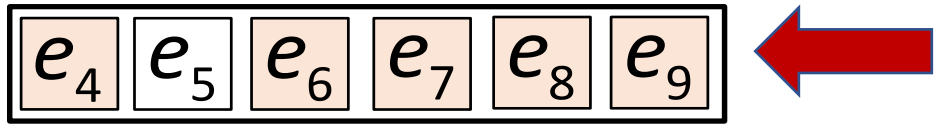
The window size can be adaptively adjusted for capturing the **complete spatial dependencies**

Update event stream



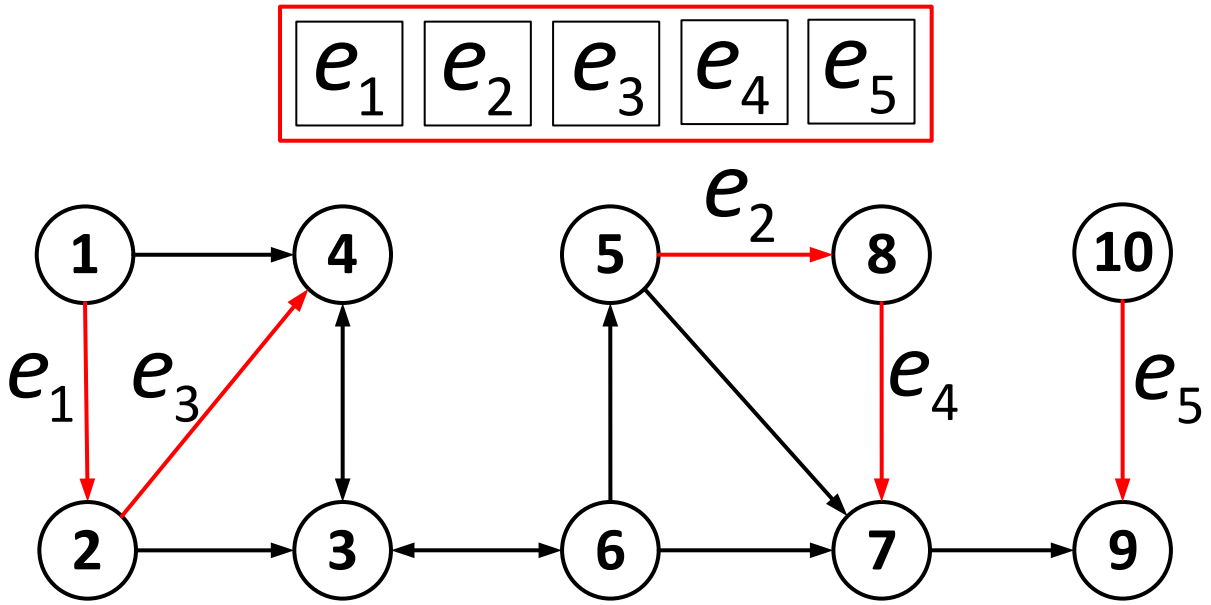
Dynamic social network graph





Solution 2: Dependency-Graph-Driven Event Parallelizing

Key Observations : An event only involves a subgraph. If there are no read-write conflicts between two events, they can be executed in parallel.

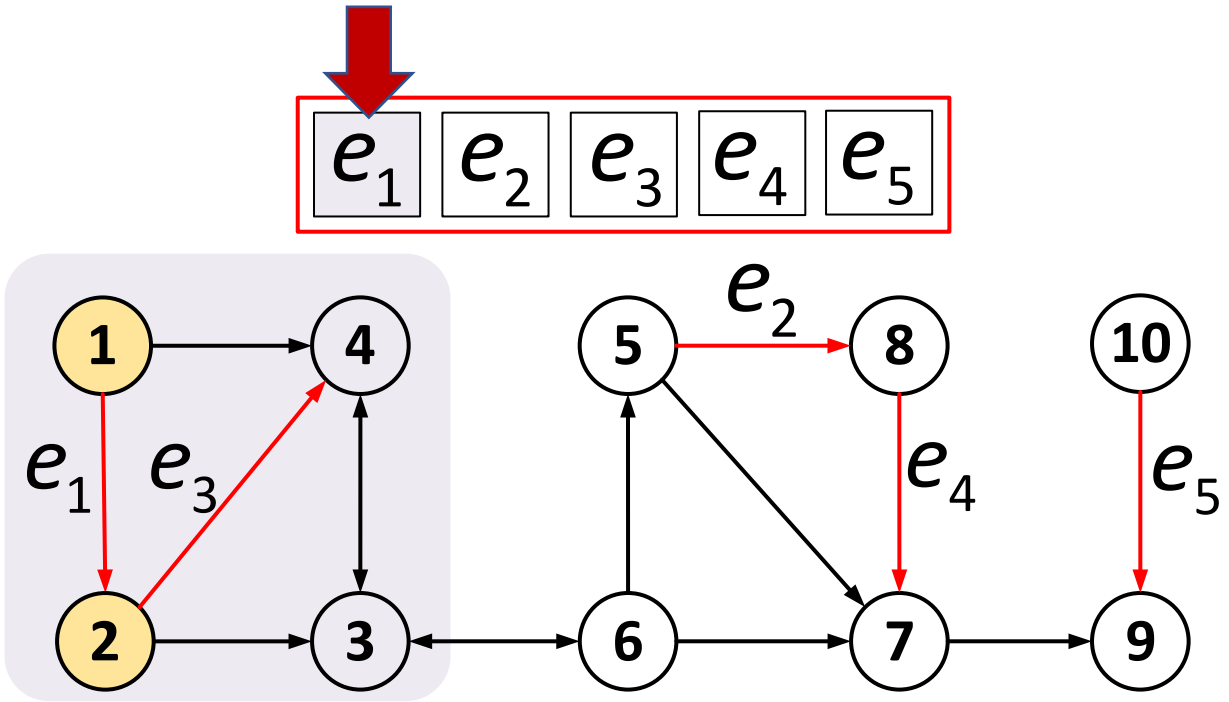


A subgraph consists of event nodes and their neighbors, where

- **Event nodes are write sets**
- **Event nodes and their neighbors are read sets**

Solution 2: Dependency-Graph-Driven Event Parallelizing

Key Observations : An event only involves a subgraph. If there are no read-write conflicts between two events, they can be executed in parallel.

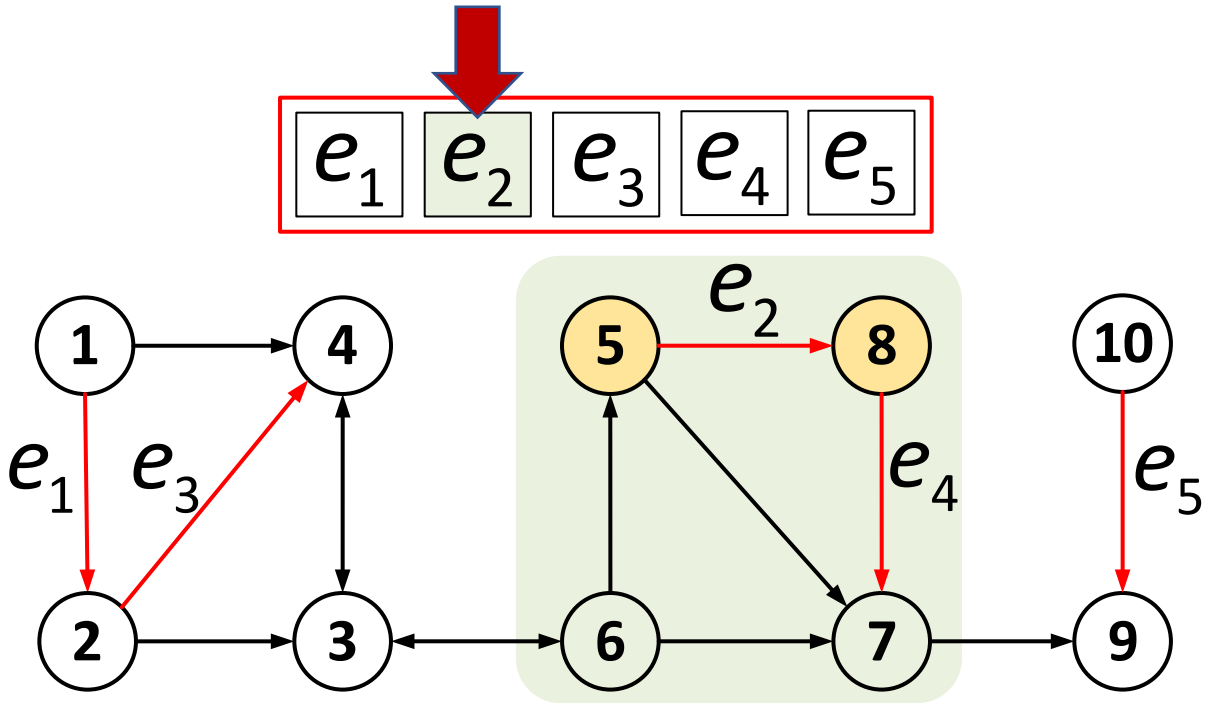


The purple area is the subgraph of events e_1

- Write set: {**1**, **2**}
- Read set: {**1**, **2**, **3**, **4** }

Solution 2: Dependency-Graph-Driven Event Parallelizing

Key Observations : An event only involves a subgraph. If there are no read-write conflicts between two events, they can be executed in parallel.

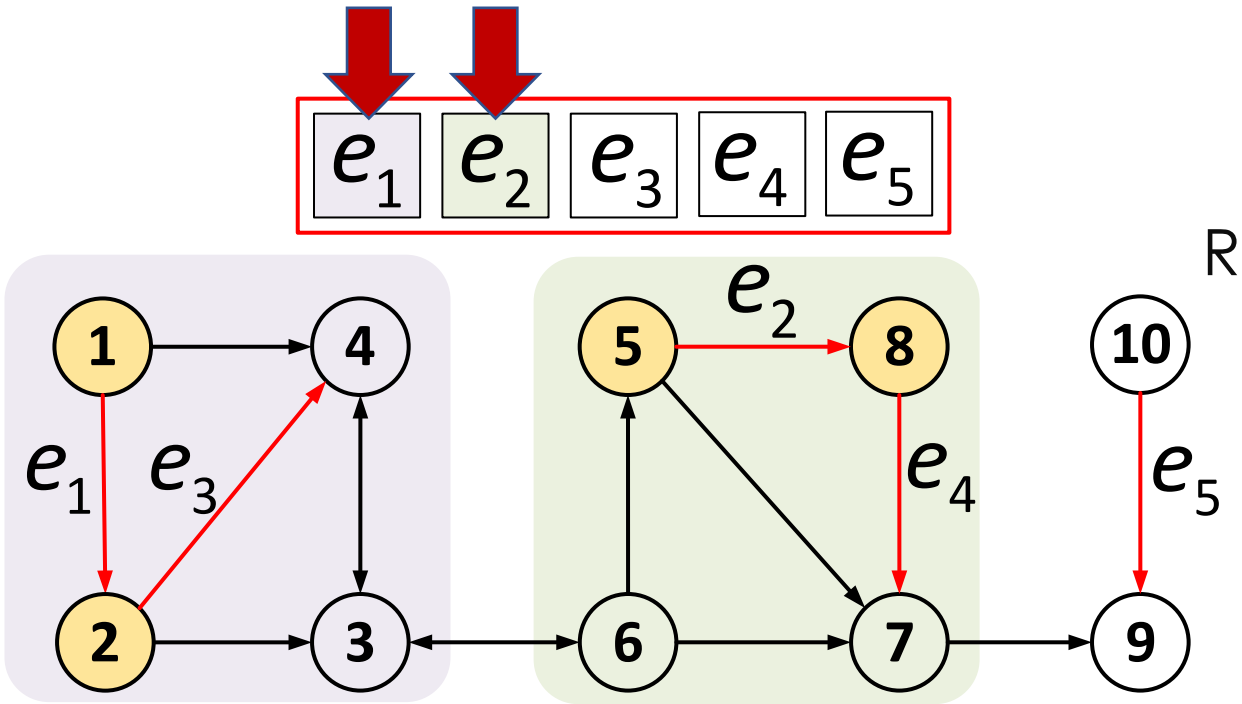


The green area is the subgraph of events e_2

- Write set: {**5**, **8**}
- Read set: {**5**, **8**, **6**, **7** }

Solution 2: Dependency-Graph-Driven Event Parallelizing

Key Observations : An event only involves a subgraph. If there are no read-write conflicts between two events, they can be executed in parallel.

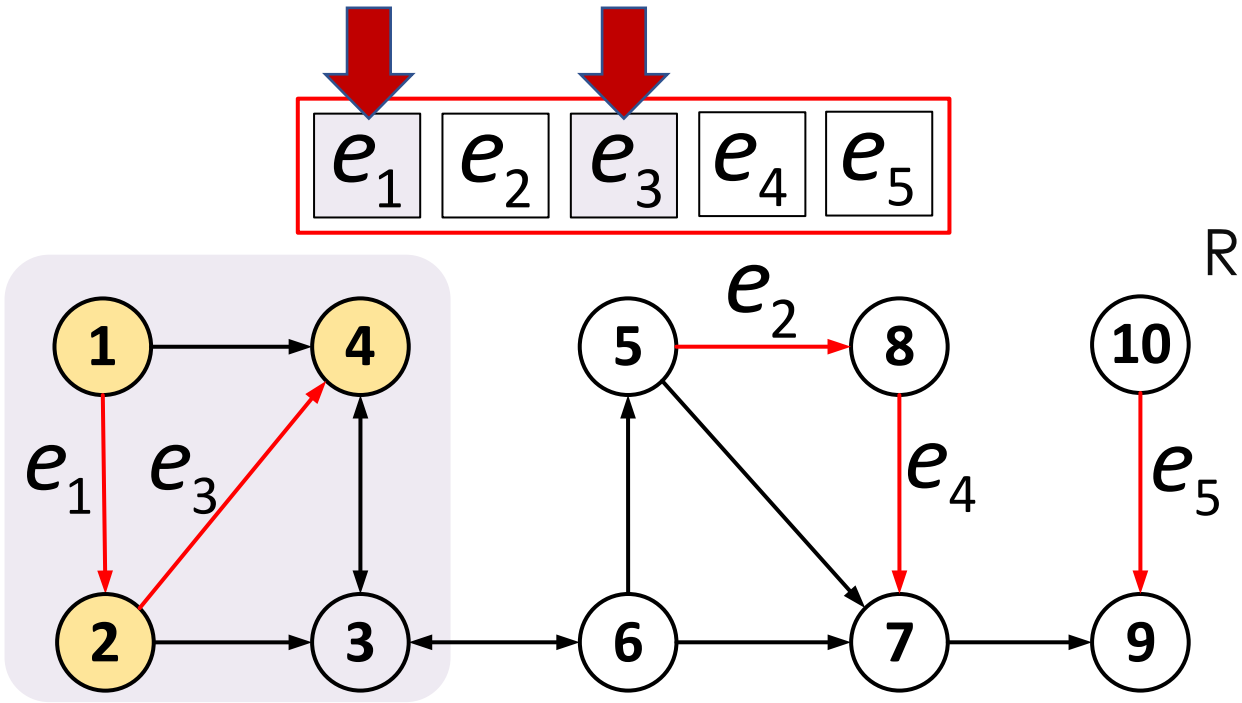


Read set: {**5**, **8**, **6**, **7** } ⋂ Write set: {**1**, **2**} = ∅

e_1 **and** e_2 **can be executed in parallel**

Solution 2: Dependency-Graph-Driven Event Parallelizing

Key Observations : An event only involves a subgraph. If there are no read-write conflicts between two events, they can be executed in parallel.



Read set: {**1**, **2**, **3**, **4** } ⋂ Write set: {**1**, **2**} ≠ ∅

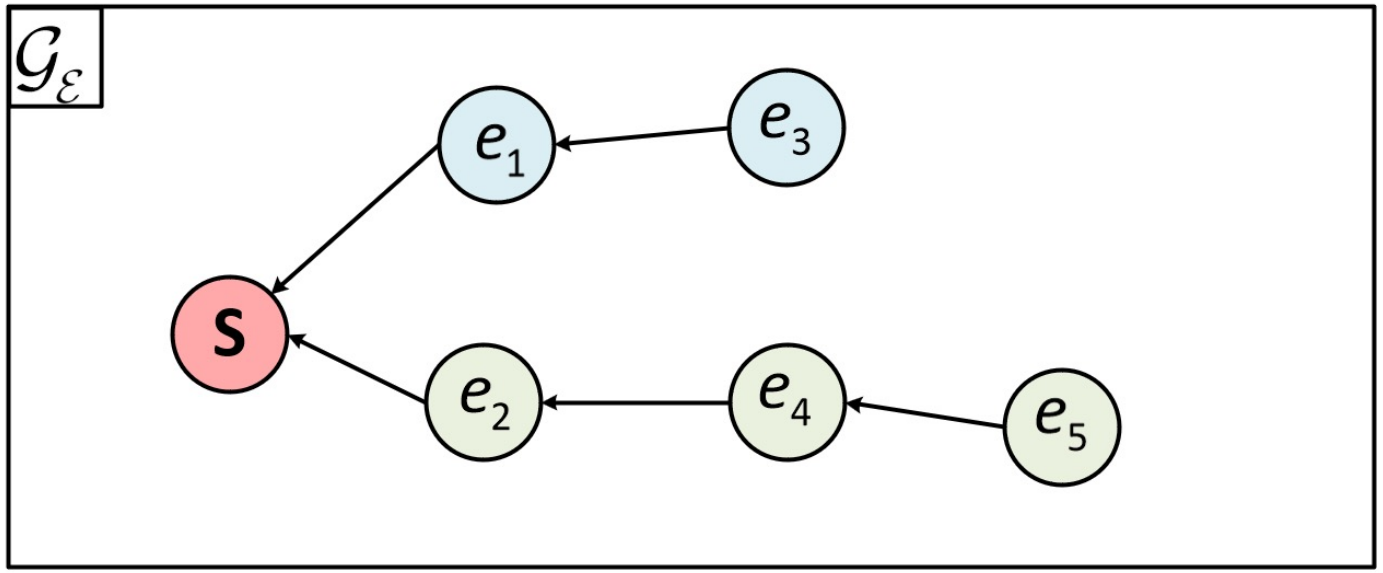
e_1 **and** e_3 **cannot be executed in parallel**

$$e_3 \Rightarrow e_1$$

Solution 2: Dependency-Graph-Driven Event Parallelizing

Event Dependency Graph $\mathcal{G}_{\mathcal{E}} = (\mathcal{V}, \mathcal{E})$

Each node represents an event. Each edge represents a dependency.

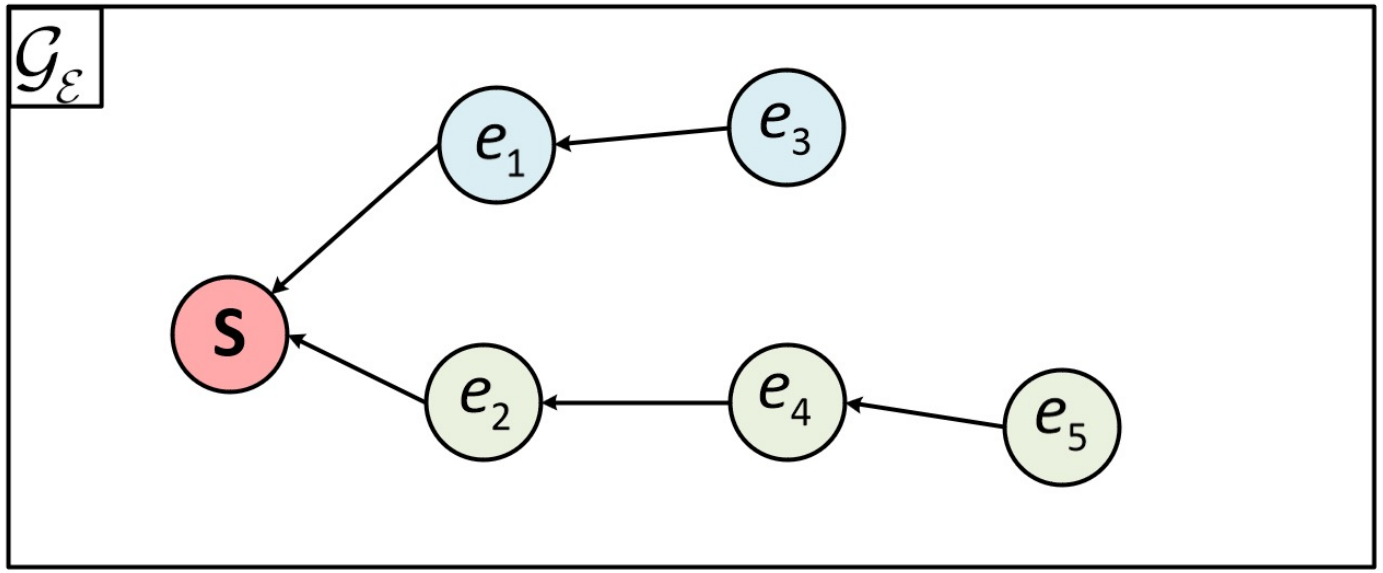


Event dependency graph

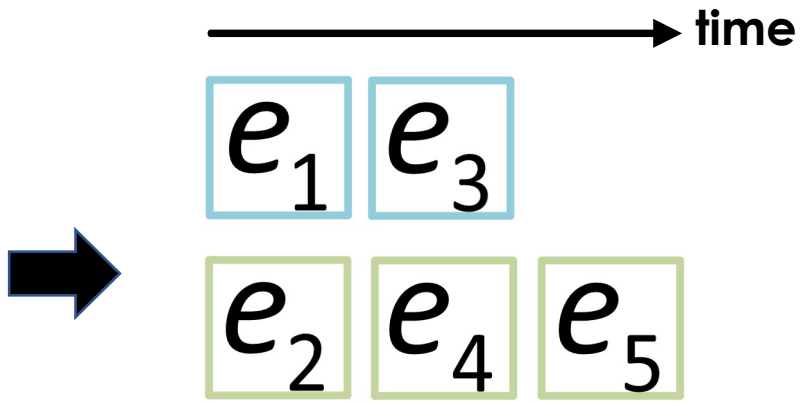
Solution 2: Dependency-Graph-Driven Event Parallelizing

Event Dependency Graph $\mathcal{G}_{\mathcal{E}} = (\mathcal{V}, \mathcal{E})$

Each node represents an event. Each edge represents a dependency.



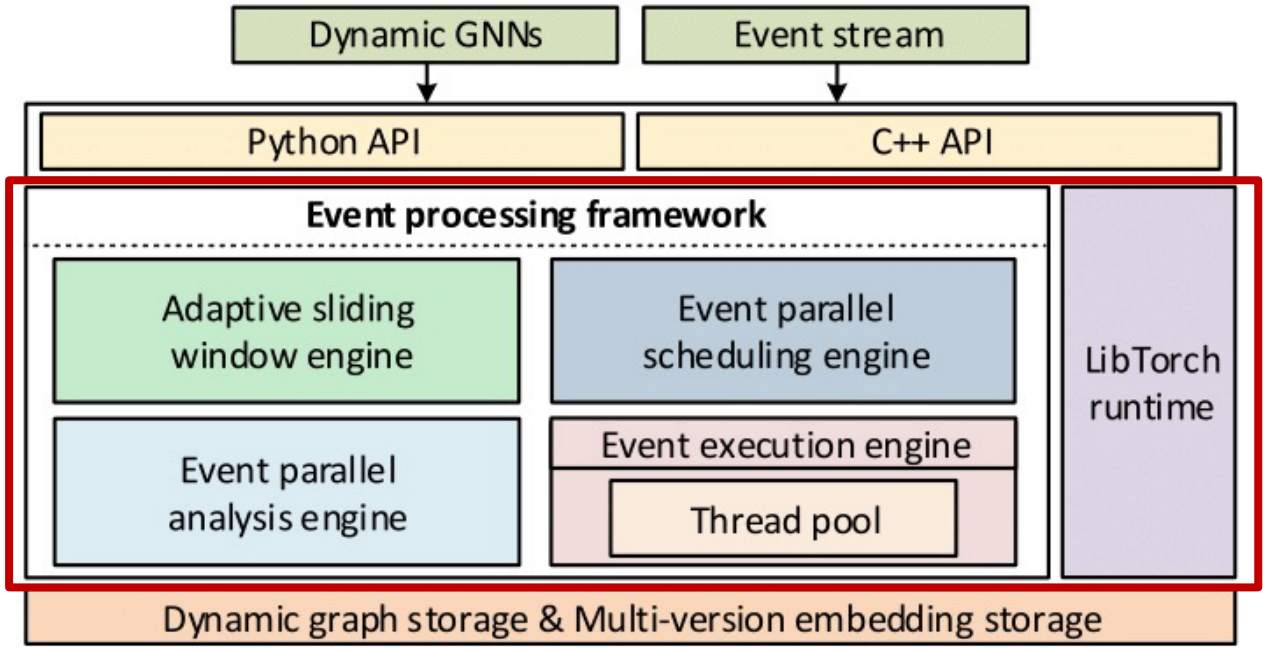
Event dependency graph



Parallel plan

NeutronStream

We deliver a dynamic GNN training framework NeutronStream , which consists of 10,336 LoC

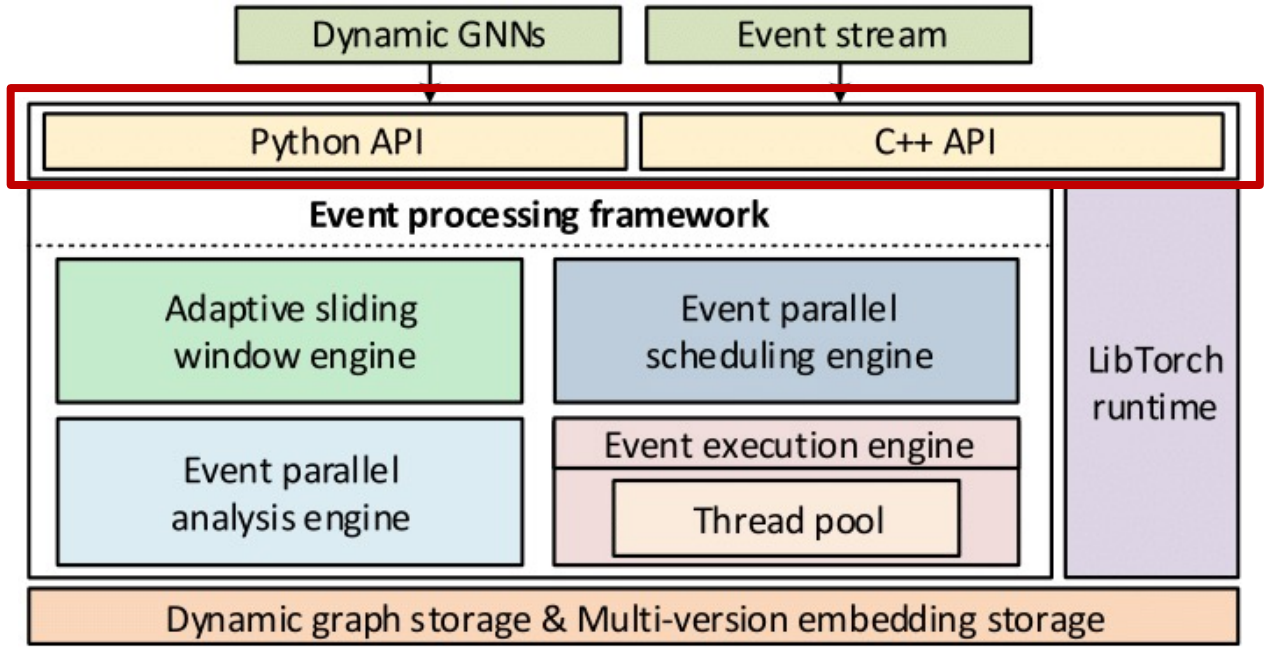


NeutronStream architecture

1. Sliding-window-based Training Method
2. Dependency-Graph-Driven Event Parallelizing

NeutronStream

We provides a set of easy-to-use APIs



NeutronStream architecture

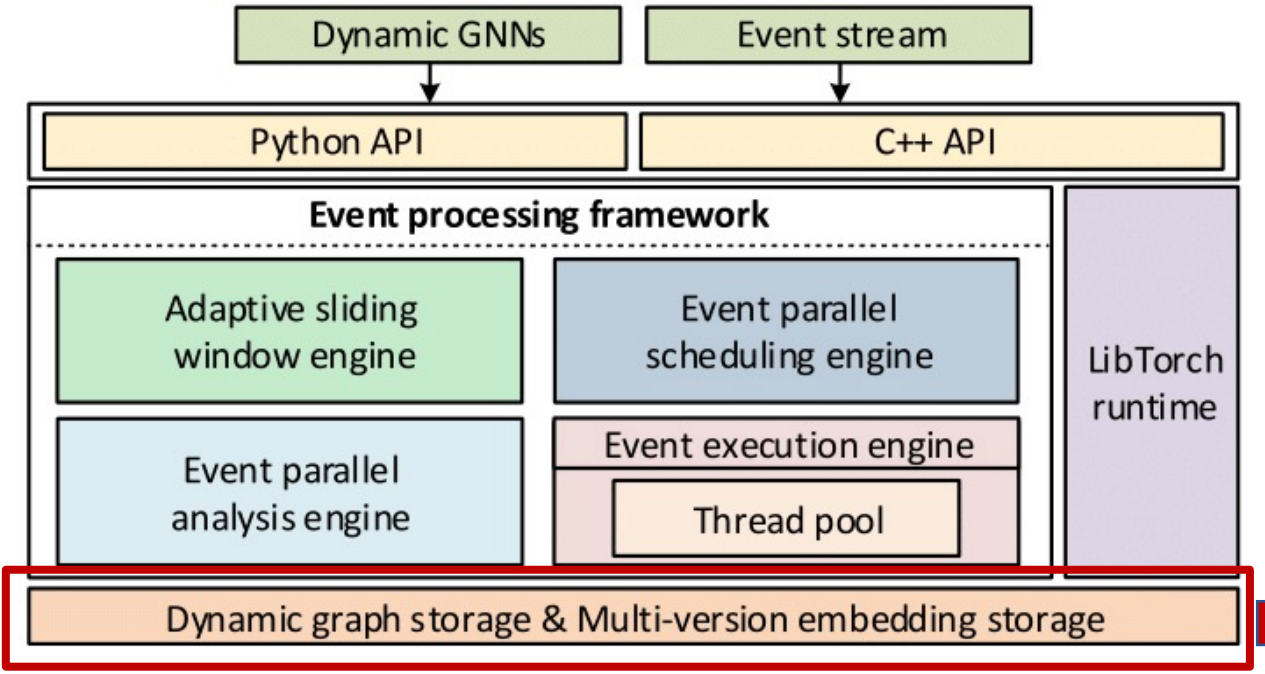
```
Class DyRep(nn.module):
   def __init__(self, h_dim):
      # initial parameters include W_h, W_struct, W_rec, W_t
and attention martix S

# update graph structure based an event
   def UpdateGraph(evt, dynGraph):
        # compute intensity
        # update attention matrix self.S
        if evt.type == 'Association'
           dynGraph.add_edge(evt.u,evt.v, undir=True)
        return dynGraph.get_subgraph(evt)
# compute aggregated neighbor information of an event node
   def Aggregate(dynSubgraph, dynEmb, nid):
        nbr = dynSubgraph.get_nbr(nid, dir='in')
        nbr_emb = self.W_h(dynEmb.index(nbr))
        q_ni = torch.exp(self.S[nid, nbr])
        q_ni = q_ni/(torch.sum(q_ni) + 1e-7)
        agg = torch.max(nn.sigmoid(q_ni.view(1,1)*nbr_emb),dim=0)[0]
        return self.W_struct(agg)
# compute updated embedding of an event node
   def UpdateEmb(dynSubgraph, dynEmb, nid, h_agg, time):
        delta_time = time - dynSubgraph.query_time(nid)
        h_exogenous = self.W_t(delta_time)
        h_self = self.W_rec(dynEmb.index(nid))
        upd_emb = nn.sigmoid(h_agg + h_self + h_exogenous)
        dynEmb.update(nid, upd_emb)
# forward function
   def forward(evt_list, dynGraph, dynEmb):
        for evt in evt_list:
           dynSubgraph = UpdateGraph(evt, dynGraph)
           h_u_agg = Aggregate(dynSubgraph, dynEmb, evt.u)
           h_v_agg = Aggregate(dynSubgraph, dynEmb, evt.v)
           UpdateEmb(dynGraph, dynEmb, evt.u, h_v_agg, evt.time)
           UpdateEmb(dynGraph, dynEmb, evt.v, h_u_agg, evt.time)
```

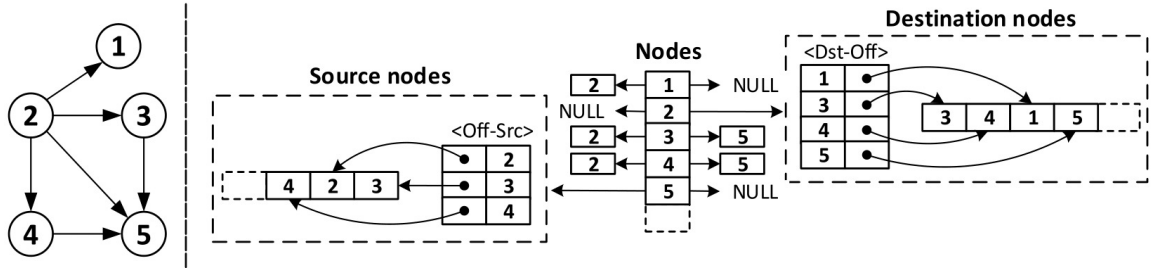
DyRep implementation with NeutronStream APIs

NeutronStream

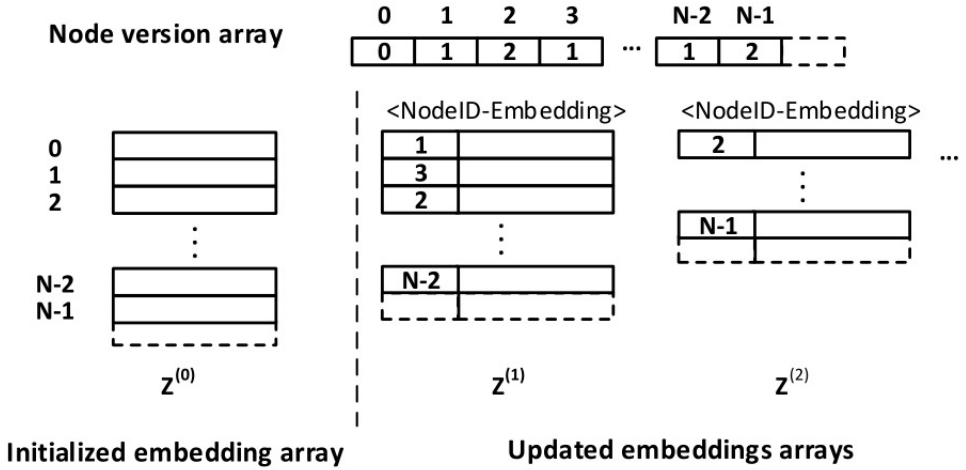
We design a built-in graph storage structure that supports efficient dynamic updates



NeutronStream architecture



(a) In-out-edge-separated indexed adjacency lists



(b) Multi-version node embedding storage

Dynamic graph storage design

Experimental Setups

Baseline: **Torch-Batch**, **NS-Batch** (NeutronStream), **NS-Slide** (NeutronStream) and **NS-AdaSlide** (NeutronStream).

Platforms:

An Aliyun ECS cluster (ecs.g5.6xlarge instance, 24 vCPUs, 96GB RAM)

Models and datasets:

- 3 Dynamic GNN DyRep, LDG, DGNN
- 6 real-world graphs.

Environment

- Ubuntu 18.04 LTS
- PyTorch 1.12.1

Table 2: Dataset statistics

Dataset	$\|V\|$	$\|E\|$.init	$\|E\|$.final	$evt.num$
Social Evolution [29]	84	575	794	54,369
Github [4]	284	298	4,131	20,726
DNC [21]	2,029	0	5,598	39,264
UCI [21]	1,899	0	20,296	59,835
Reality [7]	6,809	0	9,484	52,052
Slashdot [12]	51,083	0	131,175	140,778

AUC Comparison

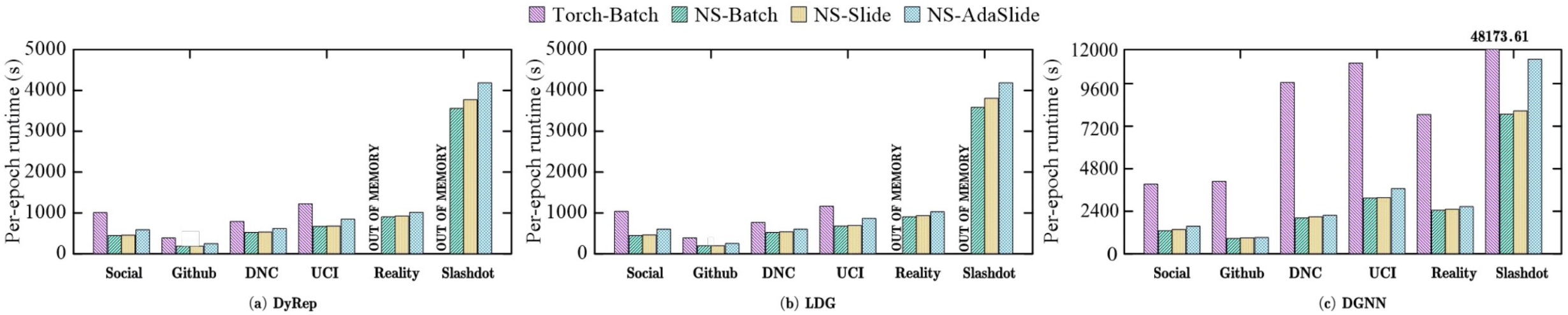
Table 3: AUC comparison of three training methods

Model	Dataset	Training Method			
		Torch-Batch	NS-Batch	NS-Slide	NS-AdaSlide
DyRep	Social	83.01%	83.01%	86.15%	**89.32%**
	Github	73.46 %	73.46%	77.54%	**79.28%**
	DNC	63.15%	63.15%	63.49%	**66.18%**
	UCI	62.46%	62.46%	63.53%	**65.68%**
LDG	Social	87.07%	87.07%	88.52%	**92.98%**
	Github	74.34%	74.34%	78.10%	**79.16%**
	DNC	64.62%	64.62%	66.50%	**69.41%**
	UCI	62.16%	62.16%	64.38 %	**66.57%**
DGNN	Social	97.21%	97.21%	97.61%	**97.67%**
	Github	81.94%	81.94%	82.54%	**84.45%**
	DNC	86.04%	86.04 %	87.72%	**88.81%**
	UCI	78.45%	78.45%	81.14 %	**82.09%**

(1) The batch method (**Torch-Batch** and **NS-Batch**) has the **lowest AUC** because it cuts off the data stream, resulting in the loss of training information.

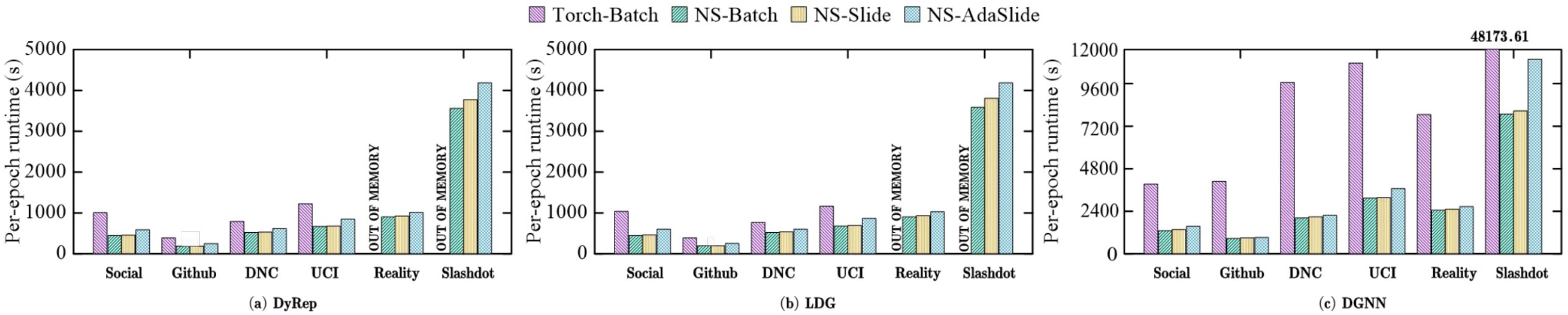
(2) The adaptive sliding method (**NS-AdaSlide**) achieves the **highest AUC** by effectively capturing the spatial-temporal locality between events.

Runtime Comparison



Compared with the **Torch-Batch**, **NS-Batch** achieves **1.48x – 5.87x** on 3 Dynamic GNNs and six real datasets.

Runtime Comparison



Compared with the **Torch-Batch**, **NS-AdaSlide** have lowest parallelism. However, thanks to our system optimization, **NS-AdaSlide** can achieve **1.27X-4.44X** speedups on 3 Dynamic GNNs and six real datasets.

Summary

NeutronStream: A Dynamic GNN Training Framework with Sliding Window for Graph Streams.

- **Proposing a new incremental learning mode with a sliding window for training on graph streams**

  We design a sliding-window-based method to incrementally train models for capturing the spatial-temporal dependency of events.

Summary

NeutronStream: A Dynamic GNN Training Framework with Sliding Window for Graph Streams.

- **Proposing a new incremental learning mode with a sliding window for training on graph streams**

  We design a sliding-window-based method to incrementally train models for capturing the spatial-temporal dependency of events.

- **Proposing a fine-grained event parallel execution scheme**

  We build a dependency graph analysis method that identifies the events having no node-updating conflicts and processes them in parallel.

Summary

NeutronStream: A Dynamic GNN Training Framework with Sliding Window for Graph Streams.

- **Proposing a new incremental learning mode with a sliding window for training on graph streams**

  We design a sliding-window-based method to incrementally train models for capturing the spatial-temporal dependency of events.

- **Proposing a fine-grained event parallel execution scheme**

  We build a dependency graph analysis method that identifies the events having no node-updating conflicts and processes them in parallel.

- **Delivering a dynamic GNN system**

  We design and implement NeutronStream, a dynamic GNN system that achieves 1.48X-5.87X speedups over PyTorch.

Summary

NeutronStream: A Dynamic GNN Training Framework with Sliding Window for Graph Streams.

- **Proposing a new incremental learning mode with a sliding window for training on graph streams**

  We design a sliding-window-based method to incrementally train models for capturing the spatial-temporal dependency of events.

- **Proposing a fine-grained event parallel execution scheme**

  We build a dependency graph analysis method that identifies the events having no node-updating conflicts and processes them in parallel.

- **Delivering a dynamic GNN system**

  We design and implement NeutronStream, a dynamic GNN system that achieves 1.48X-5.87X speedups over PyTorch.

- **The codes are publicly available on github**

https://github.com/iDC-NEU/NeutronStream