1. Introduction

This document defines the simulations an their necessary mathematical background.

2. Context and Problem Definition

In this section we define all the features of the network

The main objective of the simulation is to analyze how do posts behave: how far they travel, how many distinct users interact with that post within the timeline and no custom recommender in action.

For that, we need to implement **Posts** and the relations with themselves, which are (1) a post can be a reply of another post (2) a post can be reposted (3) a post can quote another post. Actions (1) and (2) needs of (4) a post can be created by a user.

Posts are shown to **users** according to the post creation time in a **timeline** and which set of users is the user **following** at a given time. The order of the post is **reverse-chronological**, so the user will see the posts from newest to oldest.

A user can interact with a post in several ways, so we must define **actions** that a user can perform over a post. In the bluesky case those are to like, reply, repost and quote another post. Replying, reposting and quoting will influence the followers of the user's timeline as well as timelines of all other users.

2.1. Notation

To not lose hours and sanity, I will specify the notation to be used in all the document here. If it's in anyway unorthodox, it's because I asked to an LLM to give me an idea.
- User from the set of users: $u \in U$, and $|U| = N$.
- Post (item) from the set of posts (items): $i \in I$, and $|I| = M$.

A user has several actions to perform in the simulation. We denote the set of possible actions as

$$\mathcal{A} = \{\varnothing, \text{like}/l, \text{reply}/c, \text{repost}/r, \text{quote}/q, \text{create}/n\}$$

And we denote user u doing action k over item i as:

$$a_{u,i}^{(k)} \in \mathcal{A}$$

As a user can perform more than one option over a post, we must denote the action as a vector:

$$\boldsymbol{a}_{u,i} = \left(a_{u,i}^{\varnothing}, a_{u,i}^{l}, a_{u,i}^{c}, a_{u,i}^{r}, a_{u,i}^{q}\right) \in \{0,1\}^{|\mathcal{A}|\,-1}$$

where each index corresponds to an action type with the order Like, Reply, Repost and Quote. E.g, $y = [1, 0, 1, 0]$ would be a like and a repost.

Each user follows other users. We call the set of users a given user follows as $\Gamma\text{out}(u)$, which are the sources of its timeline. Γ has been chosen due to this being the neighbours of user u. For the sake of completion, we call the set of users that follow our user $\Gamma\text{in }(u)$.

Every user creates (or has created) posts. We denote the list of posts created by user v as $P_v = \{i \in I : \text{author}(i) = u\}$. We define A_u as the activity stream, which is all the posts that will end up in the timeline by user u.

$$A_u(t) = P_u(t) \cup \left\{i \in I \mid \exists \tau < t : a_{u,i}^{(\tau)} \in \{c, r, q\}\right\}$$

The timeline of a user u is all the posts created, replied, reposted and quoted by all the users the user u follows.

$$\mathcal{T}_u(t) = \bigcup_{v \in \Gamma_{\text{out}}(u)} A_v(t)$$

Regarding evaluation metrics, we have to distinguish between several traces of the simulation, one for what the user has seen, another for what the user has done, and the last one for what has happened to the post.

- Impression history of a user: $H_u^{\text{imp}}(t) = \varepsilon_u(t) = (i_1, i_2, ..., i_k)$. It is essentially a subset of $\mathcal{T}_u$, but allows us to review in order what has the used seen. will be needed to count how many users have seen each post.
- User historic activity: $H_u^{\text{act}}(t) = \mathcal{H}_u(k) = \{(i, a, \tau) : a \neq \varnothing, \tau < t\}$. What exactly did the user do, at which time.
- Item trajectory: $T_i(t) = \{(u, a, \tau) \mid \tau < t\} \text{ where } a \in \{c, r, q\}$. That is a list with all the users who have reposted at given time time τ.

3. Version 1: Bare-bones

We are going to define the bare-bones simulation. The objective of this is to build an engine which does not take into account *real data* but only all the concepts that are going to interact and test if the theoretical idea makes sense.

3.1. Scope

This first version of the engine simulation should be used to verify and stablish a solid implementation basis which are verifiable - that is to be as certain as possible of bug's absence. The document *specification_bluesky* specifies what features compose the features Bluesky social network has, which we'll describe (hollisticaly) a subset in the following paragraph.

3.2. Axioms/Assumptions

This lists what the simulations assumes (several simplifications) in order to simplify the implementations. This are not immutable, some of them will be torn down in more advanced versions of the simulation.

1. User/Agent Homogeneity: every user is indistinguishable from the other users, they behave absolutely the same, that is, they have the exact same decision policy π.

$$\forall u_i, u_j \in U : \pi_{u_i} = \pi_{u_j} = \pi$$

2. Action Independence: The agent (user) is memoryless regarding past actions:

$$\mathbb{P}\left(a_{u,i}^t \mid \mathcal{H}_u\right) = \mathbb{P}\left(a_{u,i}^t\right)$$

3. Structure Stability: the underliying structure of the Graph is not going to change during the simulation.

- User Population: no new users are added in the simulation duration.
- Post Population: no new posts are going to be created during the simulation duration.
- User Relationships: no new following/followers are going to be added.

4. Action Subset: The v1 will restric to ignore a post, like it or repost it (no creation).

$$\mathcal{A} = \{\varnothing, l, r\}$$

5. Algorithm: shows followers posts with a chronological (oldest to newest) order.

3.3. Pseudo Algorithm

We can simulate the networks with a Discrete Event Simulation with the Event Scheduling Algorithm.

First, we'll use synthetic data generated in JSON format to run the simulation. This code can be found in [simulation/synthetic_data_generation/main.py], and generates the following structure:

```json
{
  "posts": [
    { "id": 0, "time": 6.56 },
    { "id": 1, "time": 12.86 }
  ],
  "users": [
    {
      "id": 0,
      "policy": [0.2, 0.2, 0.2, 0.2, 0.2],
      "following": [84, 65, 80],
      "followers": [1, 7, 9],
      "authored_post_ids": ["0_0", "0_1"]
    },
    {
      "id": 1,
      "policy": [0.1, 0.9, 0.0, 0.0, 0.0],
      "following": [0],
      "followers": [],
      "authored_post_ids": []
    }
  ]
}
```

Next step is to create the simulation graph from the synthetic data. That is

- Users: How many, which followers and following, and store them in an array. This is U from the notation section.
- Posts: Every user will have authored posts, and store them in an array. This is I from the notation section.
- Fill timelines: every user has to see the posts of users that already follow him.

A user in the simulation contains the following information:

- id: identifier of the user.
- following: other users which our user follows. They determine the timeline. This is $\Gamma_{\text{out}}(u)$ from our notation.
- followers: other users which follow this user. Those users will be affected when our user interacts with a post. This is the $\Gamma_{\text{in}}(u)$ from the simulation.
- timeline: all the post the user has to see in the future. Is $\mathcal{T}_u(t)$ from the notation section.
- posts: which post did the user author.
- policy: Probabiliy associated to each action. Must add to one.

A user will be able to perform three actions over a post: like, repost or nothing.

To run the simulation we'll use the **Event Scheduling Algorithm**. An event in our simulation is defined as

- time: which timestamp has this event happened.

- action: which action has the user did.
- user: all the information listed in the user category.
- id: number of the event happening in the simulation.

As the simulation wants to focus on posts, it will log out a trace in JSON format, which is a register of all events that happened in the simulation. Contains the same information of the event with the id of the post that has been published.

2. Generate $a^k_{u,(i)}$ where $(i) \in \mathcal{T}_u(0)$ (the first event on the user timeline) $\forall u \in U$
3. Start the loop (event scheduling algorithm)
4. If the action is a repost, quote or reply append this to the other users timeline.
5. Append every action into the trace.

3.4. Implementation details

There must be a list with all the posts, and then each user has a min-heap (?) with a index (or a pointer) to the post the user has to see (the oldest one).

Each user must have both which posts has he written, which posts has he interacted and what action did the user performed (well, that's the trace)

Regarding the time between actions of the user, we will assume a exponential distribution, such as an interarrival time.

Due to axiom 1, user will have all the same weights π, but which action is performed is a weighted probability (uniform from zero to one and in which interval falls)

3.5. Algorithm evaluation using Markov properties

Social networks are path-dependent (user action depends on past actions performed in different posts), we can formalize the Simulation Engine as a Markov Chain to prove convergence.

Markov chains are usually characterized as *memoryless* process, meaning that the probability of going to the next state does not depend on previous states:

$$\mathbb{P}\left(s_j \mid s_1, ..., s_{j-1}, s_j\right) = \mathbb{P}\left(s_j \mid s_{j-1}\right)$$

This does not behave at all as a social network nor is not the statement of the first axiom. The tricky fact is that we can characterize the user history of impressions $\mathcal{H}_u$ and the item states $\mathcal{C}_i$ as a Markov chain with the memoryless property by defining the state as S:

$$\mathcal{S}_t = \left\{\mathcal{H}^t_{u_1}, ..., \mathcal{H}^t_{u_N}\right\} \cup \left\{T^t_{i_1}, ..., T^t_{i_M}\right\}$$

Given n the number of posts seen until now, we can consider all the posts until that point, so we can express the memoryless like this:

$$\mathbb{P}(\mathcal{S}_n \mid \mathcal{S}_{n-1}, ..., \mathcal{S}_1) = \mathbb{P}(\mathcal{S}_n \mid \mathcal{S}_{n-1})$$

Considering this fact gives us all the nice Markov chain properties, such as, giving an explicit probability distribution of changing between states, our simulation will converge to the stationary distribution π of the Markov Chain. Therefore, if the code produces a distribution of interactions that matches π then the code is bug-free. Let's narrow it down with an example.

A user i has it's policy defined by π_u, which is essentially a probability distribution which has to add up to 1.

$$\pi_u = (\pi_\varnothing, p_l, p_r); \sum \pi_j = 1$$

We can do that by axiom 1, which tells us that every user has the same policy.

Let's invoke the other axioms to guarantee that this is a markov chain. Axiom 2 makes π_u not change with t (what a user has seen depends on the time it has been seen) which makes the system time-homogenous. Axioms of stability (3 and 4) also makes it homogenous. Axiom 5 ensures that all users see all posts of the users they follow, so defines the state transitions, the topology of the chain. A more sophisticated recommender would change that, and it would not be a markov chain anymore. The definition of memoryless given in this section is the final nail in the coffin, the system can be modelized as a markov chain, therefore it will converge to it's analitical distribution.

As a final note, the User Homogeneity axiom is *not needed* for the system to be modelized as a Markov chain, but it's needed to find an analytical expression to test the code against.

As a final final note, axiom 5 *i think* could be rephrased. That is, imagine that user i sees posts $\{1, 2, 3\}$, a simple markov chain. Now, assume user j reposts post 3, and that's why user i is seeing that. Despite this not seeming markovian, i think it still is? State of user i depends on user j, that means that the path (current state $\mathcal{S}_t$) depends on the state of the user j, so it is still a markov chain with a bigger state which encompasses all users of the social network.

3.6. Correctness & Limitations

The Axioms massively simplify what is a social network in order to provide a verifiable implementation. I want to address two facts which may steer away the simplification too far from being an actual representation.

1. Chronologically sorted or reverse-sorted: most of social network feeds are not given from oldest to newest, but from newest to oldest. Assuming a not reverse chronological order helps not to ask unconfortable questions regarding new timeline added posts (when a newer post should appear in the timeline if you are showing it from newest to oldest) which would clutter a rather simple testing implementation. Additionally, non-reverse order for sure mantains a Markov structure, which I am not sure with reverse chronological order.
2. Timeline definition: the definition of a timeline uses a *union* $\cup$, which implies *non repeated items* appears in the timeline. A normal implementation would just repeat the items (using a union of lists, not sets) which is not also how a social network feed behaves. There should be a system in place that refloats newer posts if they get popular again with some criteria. This will almost certainily break the Markov assumption for sure unless treated with care.

Current known limitations then are:
1. Reverse chronological order instead of non-reversed.
2. Timeline showing similar users popular posts a correct good enough times.
3. Post relation with each other: a reply should show original and replied, as well as a quote, as a single item (so both reply and quote should *create* a post).

4. Version 2: Sessions

The main objective of v2 is to implement a Reverse Chronological Order algorithm (instead of a Chronological Order) with the introduction of user sessions.

A user now can be in two states:

- active: will see its feed and interact according to a policy π_u.
- inactive: its offline touching grass :)

A user will switch between those two states periodically or because it has received a notification.

A notification can bring a user back to active with a given probability.

TODO: Introduce posts relationships and behaviours as bluesky does, described in [timeline_bluesky.typ] or move to v3

4.1. Axioms

1. User/Agent Homogeneity: every user is indistinguishable from the other users, they behave absolutely the same, that is, they have the exact same decision policy π.

$$\forall u, v \in U : \pi_u = \pi_v = \pi$$

2. Action Independence: The agent (user) is memoryless regarding past actions:

$$\mathbb{P}\left(a^t_{u,i} \mid \mathcal{H}_u\right) = \mathbb{P}\left(a^t_{u,i}\right)$$

3. Structure Stability: the underliying structure of the Graph is not going to change during the simulation.

- User Population: no new users are added in the simulation duration.
- Post Population: no new posts are going to be created during the simulation duration. **(?)** (if quote and reply is creating a post, then…)
- User Relationships: no new following/followers are going to be added.

4. Action Subset: The v1 will restric to ignore a post, like it or repost it (no creation).

$$\mathcal{A} = \{\varnothing, l, r\}$$

5. Sessions: user won't be active during all the simulation. When a user is active, will be able to perform any $a_{u,i} \in \mathcal{A}$. The user will be active or inactive periodically.

6. Notifications: when a user interacts with a post of another user, the latter can receive a notification. If its inactive, will come back to active with a given probability. If its active, nothing happens.

7. Algorithm: User u sees its followers posts $\Gamma_{\text{out}}(u)$ in a reverse-chronological order.

4.2. Implementation details

Regarding User Heap Now the Heap associated to every user must output in reverse-chronological order, that is, return the element with the largest timestamp instead of the gloabal Heap, which has to do the opposite.

This makes us consider mainly when we have to "empty" the heap of a user, because if a user timeline keeps getting bigger but its not online a lot, could lead to unbound memory growth of `TimelineEvents` structs.

Regarding Sessions Main problem of the sessions is to make sure an action $a^k_{u,i}$ is performed when the user is inactive. Two ways of taking care of this:

- DoD: store another array called mask with 0 or 1 depeding of when the user is online or not and prevent generation of actions if that is set

- Check everytime with a parameter “change_state”, and if the action would be scheduldded before that just don’t append it.

I think mask would be better because also solves the init problem: generate all the users, generate with the mask who is and isn’t active and start the simulation. Options two seems more convoluted.

Regarding Notificaitons

Oof.