No Time to Interact: Simulating Population Protocols at Scale

Lukas Hintze
University of Hamburg, Germany

Manuel Penschuck
University of Southern Denmark, Odense, Denmark

Abstract

We consider the simulation of population protocols with a random scheduler. A population protocol consists of n agents each associated a state $q_i \in Q$ where $|Q| = \mathcal{O}(\log n)$ for many protocols. An execution consists of a large number $I \gg n$ of interactions, each time selecting two agents uniformly at random and updating their state solely based on the current states of the pair.

We propose – to the best of our knowledge – a novel and faithful simulation technique: we show that a protocol's (undirected) interaction multi-graph follows the configuration model with a Poisson degree distribution (up to some rejection). We then introduce a sampling algorithm for the component structure of the configuration model which we believe to be of independent interest. In the subcritical regime, its runtime scales in the number of component types rather than the graph size. Finally, we show how to accelerate the simulation of a protocol by using the component structure as an execution plan template. For any fixed integer parameter $\tau \geq 2$ and $I \gg n$, the time taken per n interactions is in $\mathcal{O}(|Q|^{2-2/\tau} n^{1/\tau})$ (where the hidden constant depends heavily on τ).

In an experimental evaluation, we give evidence to the practicality of our approach. For large systems, we observe a speedup of several orders of magnitude over state-of-the-art simulators.

2012 ACM Subject Classification Computing methodologies → Agent / discrete models

Keywords and phrases Stochastic Simulation, Population protocols, Configuration model, Component structure, Sampling

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.19

Supplementary Material *Software*: `https://codeberg.org/manpen/graph-based-pop-prot-sim`
Software (Archived Source Code and Data): `https://zenodo.org/records/21135860`

Funding This work was partially funded by the Deutsche Forschungsgemeinschaft FOR 2975.
Manuel Penschuck: This work was partially funded by Novo Nordisk Foundation grant NNF21OC0066551.

Acknowledgements This material is based upon work initiated on a workshop of the DFG FOR 2975 "Algorithms, Dynamics, and Information Flow in Networks". A substantial part of the computation for this project was performed on the UCloud interactive HPC system, which is managed by the eScience Center at the University of Southern Denmark.

1 Introduction

Population protocols are a model for massively parallel computation introduced in [1]. There are n anonymous *agents*, each in a *state* in some *state space* Q. The agents' states are updated in discrete time-steps through *interactions* of agent pairs. We consider a *probabilistic scheduler* where the interacting pairs are chosen uniformly at random (u.a.r.).

When an ordered pair of agents (called *initiator* and *responder*) interacts, they observe each others' states and update them according to protocol-specific *transition function* $\delta\colon Q^2 \to Q^2$. For simplicity, we assume that δ is deterministic and can be evaluated in constant time (we remove this restriction in Section 5.2). As agents are anonymous, the system's state is entirely given by the *state histogram*, or, equivalently, the multiset of states.

To simulate a population protocol, one may simply follow this specification and execute discrete interactions. This approach takes a linear time in the number of interactions [2], which is prohibitive to simulating the very large systems needed to gain insight into slow-growing observables (say order $\sqrt{\log(n)}$ or even $\log\log(n)$) when designing new protocols [2].

However, one can often do better by exploiting the anonymity of agents. Call two interactions *non-colliding* if they do not have an agent in common. Then sequences of non-colliding interactions may be reordered arbitrarily without changing their outcome. Thus, if one knew that the next ℓ interactions are pairwise non-colliding, one can execute them as a *batch*: As agents are anonymous, it suffices to know how often any given pair of states (as opposed to agents) interact in a batch. If there are much fewer state pairs than ℓ, batched execution is much faster than the linear-time approach. If the $(\ell{+}1)$-th interaction results in a collision, the colliding state can be sampled from the states we just updated.

The number of interactions until a collision occurs can be efficiently sampled [2]. To simulate I interactions, this approach takes $\mathcal{O}(I(|Q|^2 + \log(n))/\sqrt{n})$ time in expectation. In addition, a small number of subsequent batches can be merged into *epochs*. For $|Q| \in \omega(\sqrt{\log(n)}) \cap o(\sqrt{n\log(n)})$ this approach takes expected time $\mathcal{O}(I|Q|\sqrt{\log(n)}/\sqrt{n})$ to simulate I interactions. This is equivalent to $\mathcal{O}(|Q|\sqrt{n\log(n)})$ time per n interactions, a natural measure since that is the degree of parallelism in the simulated system.

On a higher level, we can interpret this state-of-the-art approach as follows. (i) commit to a collision type (in [2] second interaction of an agent), (ii) sample the number of interactions until this collision occurs, and (iii) exploit the lack of these collisions for efficient simulation.

Our contribution

We flip the previous approach. Instead of asking how many interactions pass until a collision occurs, we fix the epoch length first and then sample the collision types that occur. This comes at the price of having to support more complex collision types involving many interactions. However, as long as the number of interactions is not too large compared to n, the collision types remain manageable. This intuition is captured in the following theorem, which we prove (using results from throughout) in Appendix B.2.

▶ **Theorem 1.** *For each $\tau \geq 2$, $I = \Omega(n)$ interactions of a population protocol on n agents with $|Q| = o(\sqrt{n})$ states can be simulated in expected time $\mathcal{O}(\frac{I}{n}|Q|^{2-2/\tau}n^{1/\tau})$.*

This theorem relies on a series of contributions:

- We model collision types as an undirected *interaction (multi)graph* that can be sampled via the configuration model with degrees following a (conditioned) Poisson distribution.
- We propose an algorithm to sample the component structure of this multigraph (Sec. 4).
- We propose an algorithm to execute a protocol based on the interaction graph (Sec. 5).
- We show the correctness and bound the runtime rigorously of both parts.
- We give empirical evidence that our novel simulator is orders of magnitude faster than state-of-the-art implementations (Sec. 6).

2 Model and notation

Notation. We denote multisets using calligraphic letters (like $\mathcal{S}$, $\mathcal{T}$ or $\mathcal{M}$) and write them with double curly braces (e.g., $\{\!\{0, 0, 2\}\!\}$). Formally, a multiset $\mathcal{S}$ is a set S together with a multiplicity function $m_{\mathcal{S}} \colon S \to \mathbb{N}^+$: each $s \in S$ is contained $m_{\mathcal{S}}(s)$ times in $\mathcal{S}$. We write $\in_m$ to indicate that multiplicity of elements is taken into account, whereas $\in$ operates on the underlying set and considers each element just once. The union $\cup_m$ over multisets adds their multiplicities, and the submultiset relation $\mathcal{S} \subseteq_m \mathcal{S}'$ holds iff $m_{\mathcal{S}}(s) \leq m_{\mathcal{S}'}(s)$ for all $s \in \mathcal{S}$.

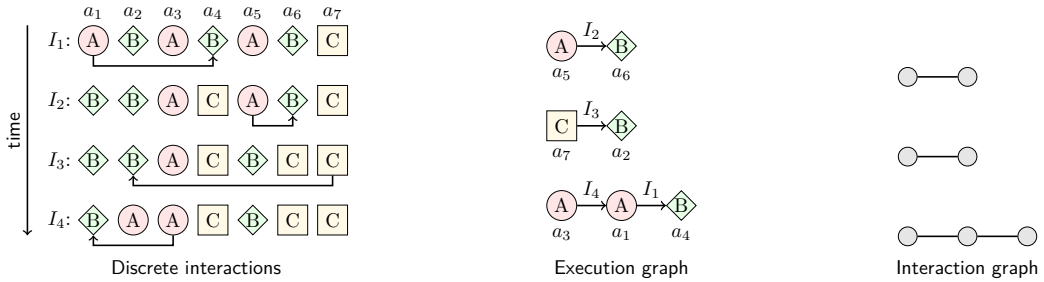

Figure 1 Left: Four discrete interactions (from top to bottom) between 7 agents with $Q = \{A, B, C\}$. **Middle:** The *execution graph* equivalent to the interactions on the left: agents are labeled with their initial states, edges with the order and direction they are executed. **Right:** The results *interaction graph*. Interactions I_1 and I_4 collide, as they share agent a_1; hence they belong to the same components. The agent ids in the left and middle figure are for illustration purposes only.

Sampling. We analyze our algorithms in the commonly accepted RAM model with support added to sample from the following distributions in constant time: Uniform integers $\mathcal{U}([a, b))$ and categories $\mathcal{U}(X)$, Binomial $\mathrm{Bin}(n, p)$, Hypergeometric $\mathrm{Hyp}(n, N, k)$.[1] Sampling from the Multinomial distribution $\mathrm{Mult}(n, p_1, \ldots, p_k)$, can be reduced to taking $k-1$ conditional binomial variates (and analogously for multivariate Hypergeometric distributions). Sampling from $\mathcal{U}(\{\mathcal{S}' \subseteq_m \mathcal{S} \mid |\mathcal{S}'| = s\})$ (for any multiset $\mathcal{S}$ and size s), which we do often, can be done in time $\mathcal{O}(\min\{s, |S|\})$ (with $|S|$ being $\mathcal{S}$ viewed as a set) by either sampling s uniform integers or sampling from a multivariate Hypergeometric distribution.

Population protocols. We have already given most of the definition of population protocols above. In addition, they also have an output function; however, this is not relevant for the simulation of protocols. Typically, under the probabilistic scheduler, the agents interacting in each step are chosen u.a.r. among all agents *without replacement*. We will, for reasons we give in Section 4.1, consider a scheduler sampling *with replacement*, but where "self-interactions" are recognized and ignored. The only difference is that these ignored interactions still count as timesteps. Formally, a *schedule* of M interactions on n agents is given by a sequence of agents $a_1, a_2, \ldots, a_{2M} \in [n]$, where in the ith interaction, agents a_{2i-1} and a_{2i} interact. Under the uniformly random scheduler, with self-interactions allowed, the sequence is i.i.d., with each a_i chosen u.a.r. We define a schedule's *interaction (multi)graph* as follows: Its nodes are the n agents $[n]$, and for each interaction $i \in [M]$, there is an edge $\{a_{2i-1}, a_{2i}\}$ between the agents participating in that interaction (and hence a loop if it was a self-interaction).

Protocol execution. We switch between the three (partial) representations of a protocol execution illustrated in Figure 1. The most obvious representation is an explicit enumeration of interacting state pairs. If we want to capture the dependencies within an execution plan, we typically use ephemeral agent ids to distinguish otherwise anonymous agents.

The *execution (multi)graph* is an equivalent representation. Its nodes are the agents, and they are colored with the agents' initial states. Edges are ordered according to their execution sequence and point from the initiating agent to the responder. Since dependencies only occur within components, the relative edge order between components is meaningless.

[1] Strictly speaking, sampling from these distributions takes *expected* constant time on a real-RAM machine (e.g., [13, 11, 16]). However, given the large number of samples taken here, the average sampling times are sufficiently concentrated to justify this common simplification.

Algorithm 1 MultiwayMSetMatch$(\mathcal{S}, d)$, where $|\mathcal{S}|$ must be a multiple of d.

if $d = 1$ **then**
  return $\mathcal{S}$
$d_l \leftarrow \lceil \frac{d}{2} \rceil$; $d_r \leftarrow \lfloor \frac{d}{2} \rfloor$
$\mathcal{S}_l \sim \mathcal{U}\big(\{\mathcal{S}' \subseteq_m \mathcal{S} \mid |\mathcal{S}'| = \frac{d_l}{d} \cdot |\mathcal{S}|\}\big)$; $\mathcal{S}_r \leftarrow \mathcal{S} - \mathcal{S}_l$
$\mathcal{M}_l \leftarrow$ MultiwayMSetMatch$(\mathcal{S}_l, d_l)$; $\mathcal{M}_r \leftarrow$ MultiwayMSetMatch$(\mathcal{S}_r, d_r)$
$\mathcal{M} \leftarrow \{\!\{\}\!\}$
for $s \in \mathcal{M}_l$ **do**
  $\mathcal{T} \sim \mathcal{U}(\{\mathcal{S}' \subseteq_m \mathcal{M}_r \mid |\mathcal{S}'| = m_{\mathcal{M}_l}(s)\})$
  $\mathcal{M} \leftarrow \mathcal{M} \cup_m \{\!\{(s,t) \mid t \in_m \mathcal{T}\}\!\}$
  $\mathcal{M}_r \leftarrow \mathcal{M}_r - \mathcal{T}$
return $\{\!\{u \cup_m v \mid (u,v) \in_m \mathcal{M}\}\!\}$

Lastly, we consider *interaction (multi)graphs* which remove the node labels and colors, as well as edge direction and order from the *execution graph*. When we refer to the *component structure* of the interaction graph, we refer to the multiset of its unlabeled components.

3 Preliminaries

In Section 4, we will describe how to sample a uniformly random schedule's *interaction multigraph*. The procedure conceptually works in the well-known configuration model, and gradually reveals the component structure of the graph as in the iterative leaf-stripping process to determine the 2-core of the graph; in doing so, it heavily relies on sampling matchings over multisets. We describe the basics of all this in the remainder of this section.

3.1 Random multiway matchings over multisets (up to symmetry)

A d-way matching over a multiset $\mathcal{S}$ is given, up to symmetry, by the multiset of multisets $\{\!\{\{\!\{S_{i,j} \mid j \in [d]\}\!\} \mid i \in [|\mathcal{S}|/d]\}\!\}$ where and $(S_{i,j})$ (for $i \in [|\mathcal{S}|/d]$ and $j \in [d]$) is a permutation of $\mathcal{S}$ chosen u.a.r. (where of course d must divide $|\mathcal{S}|$).

Algorithm 1, MultiwayMSetMatch$(\mathcal{S}, d)$, samples such a random matching. It first partitions $\mathcal{S}$ into two sets $\mathcal{S}_l$ and $\mathcal{S}_r$ of appropriate size (via sampling $\mathcal{S}_l$ as a submultiset of fixed size of $\mathcal{S}$ chosen u.a.r.), recursively sampling matchings (for sizes $\lfloor d/2 \rfloor$ and $\lceil d/2 \rceil$) over these multisets, matches these together as a bipartite matching, and then "forgets" the order of the matching. For S being $\mathcal{S}$ viewed as a set, there are at most $|S|^{\lfloor d/2 \rfloor}$ kinds of groups from $\mathcal{S}_l$ and $|S|^{\lceil d/2 \rceil}$ kinds of groups from $\mathcal{S}_r$, so matching them takes at most $|S|^d$ hypergeometric samples, and $\mathcal{O}(d|S|^d)$ time. If there are fewer elements $|\mathcal{S}|$ to distribute than there are potential $|S|^d$ kinds, we may –as an optimization– assign them individually using $|\mathcal{S}|$ uniform samples without replacement.

3.2 The configuration model

The *configuration model*, introduced by Bollobás in [3], generates a random (multi)graph of a given degree sequence roughly as follows. For each node i with degree d_i, we create exactly d_i so-called *stubs* (or *half-edges*). The stubs are then perfectly matched u.a.r. (each such matching is called a *configuration*), and stubs belonging to the same node are contracted to obtain the multigraph. Conditioned on the resulting multigraph being a simple graph, the distribution is uniform; however, it is not in general (see, e.g., [9]).

The configuration model exactly matches the distribution of the interaction multigraph obtained from a uniform random population protocol scheduler. Given the counts c_i of how often each agent i appears in the schedule, a uniform scheduler matches *agents* equivalently to how the configuration model matches *stubs* with degree sequence $(c_i)_i$.

The key idea of our sampling algorithm is to exploit the many symmetries (automorphisms) found in very sparse configuration models with $o(n)$ edges. In this subcritical regime, most nodes will be singletons (inactive agents) or part of a pair (two agents interacting exactly once). Instead of sampling each pair's edge individually, we sample the total number of pairs.

To handle more complex structures efficiently, we process sets of stubs that have not yet been shown to be non-isomorphic in bulk. We do so by first partitioning stubs only by the degree of their associated node. Then, we let the contraction of stubs into nodes within each partition class be u.a.r. among all partitions into contractions of the appropriate size (c.f. Section 3.1). The partition is then iteratively refined. This is licit as the distribution of the multigraph is the same for any contraction of the stubs into nodes, as long as the nodes have the right degrees.

3.3 Iterative leaf removal and the 2-core

We sample the matching of the configuration model in bulk using the same order in which edges are marked in the following *iterative leaf removal* process:

Consider an initially unmarked multigraph. Define a node's *residual degree* as its degree in the subgraph of unmarked edges. *While there is a node of residual degree at most 1: Let S be a non-empty set of such nodes; mark S, the unmarked edges between nodes in S, and the unmarked edges in the cut $(S, \overline{S})$.*

When the process (which is usually given for the special case $|S| = 1$) terminates, the subgraph of unmarked nodes forms the graph's *2-core*, i.e., the unique maximal subgraph having minimal degree 2 [4]. Note that the components containing the 2-core are exactly those which contain a cycle (including loops and multiedges).

This marking order is useful for us because stubs of nodes of residual degree 1 must necessarily belong to different nodes, so that we can immediately discern the component structure resulting from matches between such stubs. If the residual degree were larger, we would have to differentiate between edges between different nodes, multiedges, and loops. The marking process has the following property, proven in Appendix B:

▶ **Observation 2.** *After each iteration of the above process, the subgraph of marked edges (but all nodes) is a forest whose trees are all induced subgraphs with 0 or 1 unmarked nodes.*

4 Sampling the interaction multigraph up to isomorphism

In this section, we discuss the details of our procedure to sample the structure of the interaction multigraph of the uniformly random schedule for one epoch. We give the pseudocode of the main outer routine in Algorithm 3; in addition to the number of agents n and a target epoch length $\hat{m}$, it takes as input a parameter τ which controls which components of the graph are considered "small". First, we sample the *degree histogram* of the graph, i.e., the counts of the number of nodes having a particular degree (Section 4.1). Afterwards, we iteratively sample the bulk of the graph, including all trees of $\leq \tau$ nodes (Section 4.2). Finally, the remainder of the graph (which are its "large" trees and 2-core) is sampled one edge of the time as in standard implementations of the configuration model (barring some accounting for already-sampled edges). We outline the routine SampleRemainingGraph

Algorithm 2 SampleDegreeHistogram$(n, \hat{m})$: Samples degree histogram for an interaction graph with n nodes and target edge count $\hat{m}$, guaranteeing at most $2\hat{m}$ edges.

repeat
  $H_0 \sim \mathrm{Bin}(n, \mathbb{P}[Y = 0])$ where $Y \sim \mathrm{Pois}(2\hat{m}/n)$
  $\hat{n} \leftarrow n - H_0, \quad i \leftarrow 1$
  while $\hat{n} > 0$ **do**
    $H_i \sim \mathrm{Bin}(\hat{n}, \mathbb{P}[Y = i \mid Y \geq i])$ where $Y \sim \mathrm{Pois}(2\hat{m}/n)$
    $\hat{n} \leftarrow \hat{n} - H_i, \quad i \leftarrow i + 1$
  $\Delta \leftarrow i - 1, \quad m \leftarrow \sum_{i=0}^{\Delta} H_i \cdot i$ `// Max degree and degree sum`
  if m *is even* $\wedge\, m \leq 4\hat{m}$ **then**
    return $(\Delta,\ (H_i)_{i=0}^{\Delta})$

in the appendix's Section 4.2.3, as the direct sampling from the configuration model is standard and the remainder of the routine consists of bookkeeping to take the matches already determined into account.

4.1 Sampling the degree histogram

To sample the degree histogram, we crucially use the fact that we may freely choose a distribution for the epoch length M. Recall that the $2M$ agents interacting in an epoch are chosen u.a.r. So the counts of how often agents appear in the sequence (i.e., their *degrees* in the interaction multigraph) have the multinomial distribution $\mathrm{Mult}(2M, \vec{1}/n)$ – the reason we allowed self-interactions in the schedule is so that this holds. So the nodes' degrees are not independent, no matter M's distribution, as they must always add up to an even number.

We can remove these dependencies by making the total number of interactions a random variable. Given some target length $\hat{m}$, consider a sequence of $L \sim \mathrm{Pois}(2\hat{m})$ agents chosen u.a.r. Then each agent occurs in the sequence $\mathrm{Pois}(2\hat{m}/n)$ times, and these are i.i.d. for all agents. Even if we condition on $L \in S$ for any non-empty $S \subseteq \mathbb{N}_0$, the degrees have distribution $\mathrm{Mult}(L, \vec{1}/n)$ (cf. Corollary 9).

So let M have distribution $\mathrm{Pois}(2\hat{m})/2$ conditioned on that being an integer at most $2\hat{m}$. Then the degree sequence $(D_i)_i \in [n]$ has the distribution of n independent $\mathrm{Pois}(2\hat{m}/n)$ random variables, conditioned on their sum being even and at most $4\hat{m}$. Note that, by Corollary 9, this is the *exact* distribution of the degree sequence and not an approximation.

The degree histogram then follows a multinomial distribution with n trials, where each class i has probability $\mathbb{P}[\mathrm{Pois}(2m/n) = i]$, conditioned on the degree sum being even and less than $4\hat{m}$. The number of binomial samples needed to sample the multinomial histogram (before potential rejection) is equal to the maximum value of the n i.i.d. random variables, i.e., the maximum degree contained in the histogram. We give the pseudocode of this routine SampleDegreeHistogram as Algorithm 2. We show in Appendix B that $\mathbb{E}[M]$ is indeed very close to $\hat{m}$:

▶ **Lemma 3.** *For M chosen as above, $\hat{m}(1 - \Theta((e/4)^{2\hat{m}})) \leq \mathbb{E}[M] \leq \hat{m}(1 - \Theta(e^{-4\hat{m}}))$.*

In the proof, we also see that the rejection probability is asymptotically $\frac{1}{2} - o(1)$.

4.2 Sampling the component structure of most of the graph

Next is the actual sampling of the component structure of the interaction multigraph given its degree histogram. Recall from Section 3.2 that the interaction multigraph has exactly the distribution obtained from the configuration model with the given degree sequence. In addition to the matching of stubs being u.a.r., we also do not determine which stubs belong to the same node at the very beginning. Stubs are instead partitioned into classes where stubs belonging to the same node always belong to the same class, and where such nodes "look the same" given what we have sampled of the graph structure so far. At any point, the grouping of ungrouped stubs to nodes within a class is u.a.r.

The sampling of the graph structure is done iteratively using two operations, with the sequencing similar to that in which edges are marked in the iterative leaf removal process (see Section 3.3 above). Recall that in said process, the subgraph containing only marked edges is a forest, and each component is an induced subgraph containing at most one unmarked node (Observation 2). This directly informs the exact way we partition stubs into classes: We identify each class with a *node kind* $k = (d_k, T_k)$, where d_k is the residual degree of the unmarked node, and T_k is the rooted tree which is the component of the unmarked node, which acts as the root; we also call a stub belonging to a node of kind k a k-stub. Once a component is fully marked, its structure is be fully determined, and it can be output immediately. So it is sufficient to only keep track of nodes with non-zero residual degree, which are in one-to-one correspondence with the components of said subgraph, all of which are trees.

Given this, we can outline the two operations: The first, ProcessLeafKind(k), determines the numbers of all kinds of matches of all k-stubs of a kind k with residual degree $d_k = 1$ (these nodes are the set S in the leaf removal process). While we do consider the edges in this matching as marked, note that the operation does not yet determine which of the stubs on the other side of the matches are grouped to which nodes. This grouping of matched-but-yet-ungrouped stubs, which is deferred until there are no unmatched k-stubs with $d_k = 1$ left, is done by the second kind of operation, Refine(k). The grouping is only done up to symmetry, refining the kinds of unmatched stubs. We do this until there are no longer any kinds (d_k, T_k) with $d_k + |T_k| \leq \tau$ left for which either operation is possible. This limits the kinds processed in this way to those which may potentially still lie in a component of at most τ nodes.

We describe the two operations in more detail in the following subsections, giving their pseudocode in Algorithms 4 and 5. The algorithm uses the following state:

- K is the set of node kinds with non-zero residual degree in the graph of marked edges; we call kinds $k \in K$ *active*. Initially, K contains kinds $(i, \bullet)$ for all positive degrees i present in the tree, where $\bullet$ represents the trivial tree that only has a root.
- $\mathcal{S}$ (for stubs) is the multiset of node kinds containing one copy of each node kind for every unmatched stub belonging to a node of that kind. Recall that we refer to stub belonging to a node of kind k as *k-stub*. Initially, it contains i $(i, \bullet)$-stubs per node of degree i.
- $\mathcal{P}_k$ (the matched pool for node kind $k \in K$) is the multiset of rooted trees which were matched to stubs belonging to nodes of kind k (but whose groupings into nodes has not yet been determined). So each element of $\mathcal{P}_k$ stands for a matched k-stub and contains the tree rooted at the node to which its matching partner belongs.

We prove the following in Appendix B:

▶ **Lemma 4.** *There is a function f so that for $\tau \geq 2$ and $\hat{m}(n) < \frac{n}{4e+\varepsilon}$ for some constant $\varepsilon > 0$, SampleInteractionGraph$(n, \hat{m}(n), \tau)$ runs in expected time $\mathcal{O}(\mathbb{E}[\Delta] + f(\tau) + \mathbb{E}[|\mathcal{E}|])$ where Δ is the maximum degree of the graph, and where $\mathcal{E}$ is the multiset of edges lying in components containing cycles or having more than τ nodes.*

Algorithm 3 SampleInteractionGraph$(n, \hat{m}, \tau)$.

$(\Delta, (H_i)_{i=0}^{\Delta}) \leftarrow$ SampleDegreeHistogram$(n, \hat{m})$
$K \leftarrow \{(i, \bullet) \mid i \in [\Delta]\}$ `/* • is the trivial tree (root with no children) */`
$\mathcal{S} \leftarrow$ multiset with $m_{\mathcal{S}}((k, \bullet)) = H_i \cdot i$ `/* mset of stubs */`
$\mathcal{P}_{(i,\bullet)} \leftarrow \{\!\{\}\!\}$ for all $i \in [\Delta]$ `/* initialize (empty) pools of matched trees */`
while $\exists (d_k, T_k) = k \in K : (d_k + |T_k| \leq \tau) \wedge (d_k = 1 \vee |\mathcal{P}_k| > 0)$ **do**
  if $\exists (d_k, T_k) = k \in K : (d_k + |T_k| \leq \tau) \wedge d_k = 1$ **then**
    ProcessLeafKind(k) `/* e.g., ` $m_{\mathcal{S}}(k)$ `maximal */`
  else
    $k \leftarrow$ some $k \in K$ with $d_k + |T_k| \leq \tau$ and $|\mathcal{P}_k| > 0$ `/* e.g., ` $|\mathcal{P}_k|$ `maximal */`
    Refine(k)
SampleRemainingGraph()

Algorithm 4 ProcessLeafKind(k): Processes a leaf kind $k = (1, \cdot)$.

$(1, T_k) \leftarrow k$
for $T' \in \mathcal{P}_k$ **do**
  output $m_{\mathcal{P}_k}(T')$ components obtaining from attaching T' to the root of T_k
$L \sim \text{Hyp}(|\mathcal{S}|, |\mathcal{S}|/2, m_{\mathcal{S}}(k))$ `/* "left" k-stubs */`
$T \sim \text{Hyp}(|\mathcal{S}|/2, L, m_{\mathcal{S}}(k) - L)$ `/* "twinned" matches between k-stubs */`
$O \leftarrow m_{\mathcal{S}}(k) - 2I$ `/* k-stubs matched to others */`
output T components obtained by attaching T_k to the root of another T_k
remove all copies of k from $\mathcal{S}$
$\mathcal{T} \sim \mathcal{U}(\{\mathcal{S}' \subseteq_m \mathcal{S} \mid |\mathcal{S}'| = O\})$
$\mathcal{S} \leftarrow \mathcal{S} - \mathcal{T}$
for $t \in \mathcal{T}$ **do**
  add $m_{\mathcal{T}}(t)$ copies of T_k to $\mathcal{P}_t$
remove k from K

4.2.1 ProcessLeafKind

First, for each tree in k's pool $\mathcal{P}_k$, ProcessLeafKind outputs the component obtained by attaching this tree to T_k's root, since these components have already been completely determined. To process the $m_{\mathcal{S}}(k)$ unmatched k-stubs, it then determines the number T of matches between these stubs: Consider the stubs in $\mathcal{S}$ being partitioned u.a.r. into two equal-sized sets (one "left" and one "right"). The number L of k-stubs belonging to the left set follows a Hypergeometric distribution: There are $|\mathcal{S}|$ total unmatched stubs, of which the $m_{\mathcal{S}}(k)$ k-stubs are marked, and we draw the $|\mathcal{S}|/2$ stubs of the left side u.a.r. without replacement. Keeping these stubs fixed, we know there are $m_{\mathcal{S}}(k) - L$ stubs on the right side. For each of the k-stubs on the left side, now draw a stub from the right side (u.a.r. without replacement) and see how many of them are k-stubs. This number T follows the hypergeometric distribution $\text{Hyp}(|\mathcal{S}|/2, L, m_{\mathcal{S}}(k) - L)$. Finally, there then are $O = m_{\mathcal{S}}(k) - 2T$ still-unmatched k-stubs which must be matched to non-k-stubs. Hence, remove all k-stubs from $\mathcal{S}$, and sample O stubs u.a.r. without replacement from $\mathcal{S}$; these are the stubs the k-stubs are matched to. It now remains to update the pools accordingly and remove k from K since all nodes/stubs of kind k have been processed now.

Algorithm 5 REFINE(k).

input : Node kind $k = (d, T_k)$ with $|\mathcal{P}_k| > 0$

add $m_{\mathcal{S}}(k)$ copies of $\perp$ to $\mathcal{P}_k$
remove k from K and all copies of k from $\mathcal{S}$
$\mathcal{M} \leftarrow$ MULTIWAYMSETMATCH($\mathcal{P}_k, d$)
$\mathcal{P}_k \leftarrow \{\!\{\}\!\}$
for $M \in \mathcal{M}_k$ **do**
  $d_{k'} \leftarrow m_M(\perp)$
  $T_{k'} \leftarrow$ result of adding the trees in M as children to the root of T_k
  if $d_{k'} = 0$ **then**
    output $m_{\mathcal{M}_k}(M)$ copies of $T_{k'}$
  else
    add $k' = (d_{k'}, T_{k'})$ to K, and $d_{k'} \cdot m_{\mathcal{M}_k}(M)$ copies of k' to $\mathcal{S}$

Algorithm 6 SAMPLEREMAININGGRAPH().

while $\exists k \in K : |\mathcal{P}_k| > 0$ **do**
  REFINE(k)
$\mathcal{S}' \leftarrow \{\!\{\}\!\}$
$G \leftarrow$ empty graph
for $(d_k, T_k) = k \in K$ **do**
  for $i \in [m_{\mathcal{S}}(k)/d_k]$ **do**
    add a copy of T_k to G, with the root node labeled (k, i)
    add d_k copies of (k, i) to $\mathcal{S}'$
$\mathcal{M} \leftarrow$ MULTIWAYMSETMATCH($\mathcal{S}', 2$)
for $M = \{\!\{(k_1, i_1), (k_2, i_2)\}\!\} \in_m \mathcal{M}$ **do**
  add an edge between (k_1, i_1) and (k_2, i_2) in G
output G `/* might have multiple components */`

4.2.2 Refine

REFINE first adds one copy of the placeholder $\perp$ to $\mathcal{P}_k$ for each unmatched k-stub (i.e., each $k \in_m \mathcal{S}$). It then groups together the members of $\mathcal{P}_k$ in a uniformly random multi-way match to determine which of the matched sub-trees (or placeholders) belong together. The rest of the routine consists of bookkeeping: First, k is removed from K since, for now, the node kind k has been processed completely. Note, however, that k may be added again later if one of the matched groups consists only of unmatched stubs (i.e., $\perp$s). Then, we compute, for each kind of matched group, the refined node kind $k' = (d_{k'}, T_{k'})$: d'_k is the number of $\perp$s, and the $T_{k'}$ is obtained by attaching the trees in the matched groups to the root of k's original tree. If $d'_k = 0$, the component of k' is now fully determined; otherwise, k' is added to K and the appropriate number of copies of k' is added to $\mathcal{S}$ to represent its unmatched stubs.

4.2.3 SampleRemainingGraph

The routine SAMPLEREMAININGGRAPH does what it says on the tin. First, it uses the REFINE operation on all node kinds which have a non-empty pool to group together matched-but-yet-ungrouped stubs. Then, it determines how many nodes of each node kind k there

must be from the pool of stubs $\mathcal{S}$, numbers them, and adds, for each numbered node, the corresponding number of stubs into a new pool $\mathcal{S}'$. It then follows the configuration model's specification: it samples a uniformly random (2-way) matching between the stubs in $\mathcal{S}'$, constructs the corresponding graph, and outputs it. Here, it also takes care of the fact that the node kinds k have trees attached. Note that SampleRemainingGraph can, in principle, be used at any point, although of course, its runtime is linear in the number of edges in the residual graph.

5 Protocol execution

The execution of a protocol can be interpreted as the process of modifying the multiset $\mathcal{Q}$ of states. In this section, we treat $\mathcal{Q}$ as an urn and repeatedly extract states u.a.r. without replacement. In order to prevent updating the same agent accidentally multiple times, we place the modified states into a second urn $\mathcal{Q}'$ and merge both urns at the end of the epoch.

We build the *execution graph* based on the interaction graph of the previous section. It is represented by $(\mathcal{T}, \mathcal{E}_c)$ where $\mathcal{T}$ is the multiset of trees and $\mathcal{E}_c$ is the multiset of edges in components containing cycles. To this end, we need to sample the following information:

- Nodes need colors (i.e., initial agent states) drawn u.a.r. from $\mathcal{Q}$ without replacement.
- Each edge has to be randomly oriented to encode the interaction's initiator and responder.
- We need to shuffle the edges to specify a random order in which interactions take place.

Observe that node colors can be assigned in arbitrary order due to the exchangeability of "uniform sampling without replacement". Further, only nodes within the same component can affect each other, and orientations are i.u.r. for each edge.

We process the components using two algorithms: most interactions take place in small trees in $\mathcal{T}$, and we execute them in the *bulk processing path* (Bpp). All remaining components are simulated step-by-step using the *discrete interaction path* (Dip). The runtime complexity of both parts is given by the following lemmas (proven in the respective subsections):

▶ **Lemma 5.** *Given a deterministic transition function $\delta\colon Q^2 \to Q^2$, Bpp for all trees in $\mathcal{T}$ with up to τ nodes runs in time $\mathcal{O}(\tau! \cdot \tau^{\tau-2} \cdot |Q|^2)$.*

▶ **Lemma 6.** *Given a deterministic transition function $\delta\colon Q^2 \to Q^2$, Dip takes time $\mathcal{O}(|\mathcal{E}|)$ where $\mathcal{E}$ is the multiset of edges in components containing cycles or more than τ nodes.*

5.1 Bulk interactions on trees

In this section, we outline the bulk processing path. We focus on establishing a runtime that scales (i) in $\mathcal{O}(|Q|^2)$, and (ii) independently of the number of interactions carried out. Since we consider τ to be a small constant ($\tau \leq 5$ in our implementation), we accept a rather crude bound on τ for ease of description (see Section 5.2 for practical improvements).

Fix $2 \leq t \leq \tau$ as the number of nodes in a tree. The remainder of this section repeatedly splits the class of all trees on t nodes until execution becomes (almost) trivial. Observe that there are $N_t = t^{t-2}$ labeled undirected trees on t nodes [5, 6]. Hence, there are at most N_t relevant entries in $\mathcal{T}$. Fix an arbitrary tree $T_t \in \mathcal{T}$ with t nodes and multiplicity $m = m_{\mathcal{T}}(T_t)$.

To faithfully execute the protocol, we have to consider all possible interaction orders. There are $(t-1)!$ equally likely ways to totally order the tree's $t-1$ edges. Further, each edge has to be independently oriented to encode which endpoint acts as the interaction's initiator and responder, respectively. This accounts for 2^{t-1} equally likely orientations. Since both considerations are independent, there are a total of $N_{T_t} = (t-1)! \cdot 2^{t-1}$ ordering and orientation classes of T_t. We can distribute the m copies of T_t into classes by sampling the multinomial $(m_1, \ldots, m_{N_{T_t}})$ from $\mathrm{Mult}(m, \vec{1}/N_{T_t})$ in time $\mathcal{O}(N_{T_t})$.

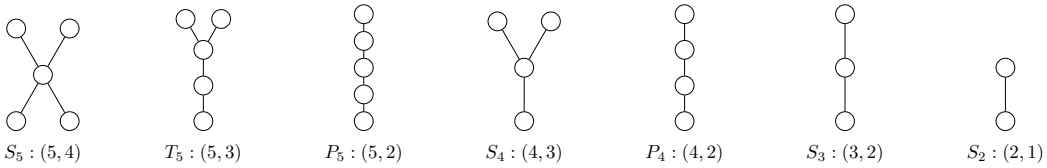

Figure 2 Tree topologies (up to isomorphism) supported for bulk processing. They can be distinguished by a fingerprint (n, Δ) where n is the number of nodes and Δ the maximum degree.

Let T_i be a fixed ordered and oriented tree on t nodes with multiplicity m_i. For simplicity, assume that we have a single tree, i.e., $m_i = 1$. We assign states drawn u.a.r. from $\mathcal{Q}$ without replacement to the τ nodes; we say we *color* the nodes. Then we apply the protocol's transition function to update the node's colors in order and orientation encoded by the tree's edges. Finally, we move the updated states in to $\mathcal{Q}'$ and discard T_i. For $m_i > 1$, we repeat the process for each copy of T_i resulting in a runtime of $\Theta(m_i \cdot t)$.

To remove the runtime's dependency in m_i (which would result in $\Omega(1)$ time per interaction), we move to process T_i in bulk. Naively, observing that there are $\mathcal{O}(|Q|^t)$ ways to color the t nodes, we could consider each in isolation resulting in a runtime of $\mathcal{O}(|Q|^t t)$.

However, we can do much better: instead of considering the tree's complete coloring with $|Q|^t$ classes, we assign each node $v \in T_i$ its own urn $\mathcal{Q}_v$ with $|\mathcal{Q}_v| = m_i$, and then carry out the interactions in T_i's order. To execute interaction (u, v), we evaluate the transition function δ on a random 2-way matching between $\mathcal{Q}_u$ and $\mathcal{Q}_v$ and assign the updated states to the two urns respectively.[2] This reduces the runtime to the acclaimed $\mathcal{O}(|Q|^2 t)$.

The approach's correctness hinges on the absence of cycles in the processed tree. As each edge (u, v) is executed exactly once, we can conceptually delete it after processing. This splits the tree into two subtrees T_u and T_v that cannot affect each other anymore. Hence, instead of considering the joint distribution of the updated state pairs, and we only retain the marginal distributions of the updated $\mathcal{Q}_u$ in T_u and the updated $\mathcal{Q}_v$ in T_v, respectively.

As an optimization, observe that the subtrees obtained by cutting edge (u, v) are independent of the edge's direction (up to naming). Hence, instead of splitting the undirected tree T_t into all edge orders and orientations, it suffices to consider only the orders. Then before executing m_i interactions along the edge $\{u, v\}$, we sample $(m_i^{\leftarrow}, m_i^{\rightarrow})$ from $\mathrm{Mult}(m_i, \vec{1}/2)$ and execute both orientations in the obvious way. This only doubles the cost per interaction while reducing the number of classes considered by 2^{k-1}. We can now finally prove Lemma 5:

Proof of Lemma 5. Based on this section's discussion, we can process all N_t trees with t nodes in isolation. For each, we consider its $N_{T_t}/2^{k-1}$ classes. For each execute the protocol in time $\mathcal{O}(|Q|^2 t)$. This results in a total runtime $W_t = \mathcal{O}(t^{t-2} \cdot (t-1)! \cdot |Q|^2 t) = \mathcal{O}(t! \cdot t^{t-2} \cdot |Q|^2)$. Summing over all size $2 \le t \le \tau$ yields the acclaimed bound. ◀

5.2 Remarks on practical bulk processing

Until now, we assumed the transition function $\delta \colon Q^2 \to Q^2$ to be deterministic. In practice, our simulator expects an arbitrary function `t((a,b), n)` translating n iterations of the state pair (a, b) into multiset M of state-pair with $|M| = n$. The function may even have side-effects (e.g., to collect statistics), as long as the simulator remains free to reorder invocations.

[2] This is the obvious generalization of the $|Q| \times |Q|$ matrix bulk processing technique of [2] to two urns.

Our implementation features a bulk processing path (Bpp) for all trees of up to five nodes. These trees have a large number of symmetries that we ignored before. We begin with isomorphic trees: Let $T \in \mathcal{T}$ be a tree on t nodes and Δ_T be its largest degree. Then, we can identify its isomorphism class by comparing its (t, Δ_T) to the values given in Figure 2. This alone reduces the number of tree classes from $\sum_{t=2}^{5} N_t = 145$ to 7 isomorphism classes.

Next, we make heavy use of the principle of deferred decisions using partially colored trees, which we call *fragments*. Let $\mathcal{F}$ be the multiset of fragments, which we initialize to contain all uncolored trees from $\mathcal{T}$ with up to τ nodes. We "execute" fragment f on t nodes with multiplicity m as follows: Instead of considering all $(t-1)!$ possible edge orders, we only concern ourselves with the *next* interaction (and defer the order of the remaining edges until later). Let m_i be the number of copies, in which the edge $e_i = \{u_i, v_i\}$ is the next to be executed. We sample the m_is according to $\mathrm{Mult}(m, \vec{1}/(t-1))$ and handle each of the cases independently. If u_i or v_i are yet uncolored, we sample m_i states from $\mathcal{Q}$ and then carry out the m_i interactions as outlined in Section 4.2. Deleting the edge then results in two new partially colored fragments, which we insert into $\mathcal{F}$.

In practice, we establish a flow graph similar to Figure 3. Each uncolored fragment (input) is represented by a counter, and fragments with $k > 0$ colored nodes by a tuple of k independent state histograms. The flow-graph contains one uncolored source node per input topology, and exactly one sink, namely the colored singleton fragment. It collects the states of all processed agents and is identified with $\mathcal{Q}'$. Since each interaction deletes exactly one edge, all interaction edges point towards strictly smaller fragments. The flow-graph is hence acyclic, and we "execute" it in a topological order. We exploit several symmetries:

- Two directed edges may produce isomorphic subproblems. For instance, any interaction on the star graph S_k with $k > 2$ produces the same fragments: the satellite becomes a colored singleton, while the rest becomes a smaller star graph S'_{k-1} with a colored center.
- Different topologies may lead to identical fragments; e.g., S_5 and S_4 both have S'_3 in their execution path. If the fragment has only one colored node, we can merge their executions.
- Observe that any bulk interaction is expensive as it involves the enumeration of $|Q|^2$ pairs of states (ignoring possibly unoccupied states). On the other hand, coloring a node without interaction is cheap as it involves only $\mathcal{O}(|Q|)$ states. Now suppose that there exists an uncolored fragment f, such that coloring a single node u yields another fragment f' which is to be processed anyhow (e.g., $f = S_4$ and $f' = S'_4$ in Figure 3). Then, instead of simulating all interactions of f, we only color u with states from $\mathcal{Q}$ and add them to f'.

As an optimization, we implemented a code generator to produce a "symbolic execution" of the flow-graph. This process is completely generic over the protocol and only has to be performed once while implementing the simulator. The resulting code uses 18 fragment types (including 7 uncolored inputs), and involves 31 random 2-way matchings of states.

5.3 Discrete interactions path

The discrete interaction path (Dip) carries out the interaction on all trees in $\mathcal{T}$ with more than τ nodes, and the edges in $\mathcal{E}_c$; let $\mathcal{E}$ be the union of their edges. Its implementation is much simpler than the bulk processing path, as we are allowed to do constant work per interaction. This allows us to individualize each agent: First initialize an empty set of ephemeral node ids V and an empty (growing) edge array L. For each edge $e \in_m \mathcal{E}$, add a copy of e to L and update V with all new nodes encountered.

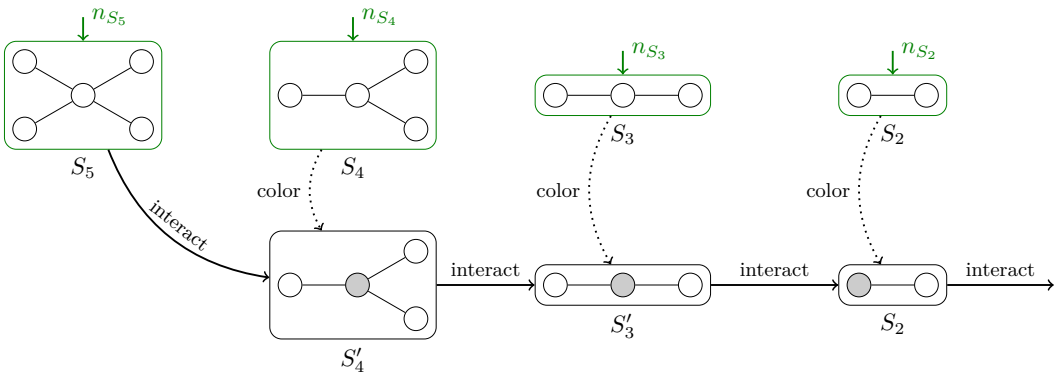

Figure 3 Each rounded box represents a fragment. The top row contains uncolored input fragments which are encoded by a scalar each. The colored fragments on the bottom are represented by a state distribution each. We execute the protocol by following the fragments in some topological order. Each depicted interaction results in at least one singleton fragment (omitted for clarity).

As a result, we now have list L containing exactly one entry for every interaction to be carried out. To carry out the interactions, we shuffle L (i.e., apply a permutation drawn uniformly at random) and randomly orient each edge independently. We then draw a state for each agent in V from $\mathcal{Q}$ u.a.r. without replacement, carry out the interactions prescribed by L, and finally place the updated states in $\mathcal{Q}'$.

Proof of Lemma 6. Assume that we delete all node kinds from $\mathcal{T}$ already processed in Bpp. Then, we can build the edge list L trivially in time $\mathcal{O}(|L|)$. Shuffling L takes time $\mathcal{O}(L)$ [12]. Sampling the initial states takes time $\mathcal{O}(|V|) = \mathcal{O}(|L|)$. Applying the constant-time transition function to each entry takes time $\mathcal{O}(|L|)$. Finally observe that $|L| = |\mathcal{E}|$. ◀

6 Experimental evaluation

We empirically study our novel *graphical population-protocol simulation* Gps and compare it to the state-of-the-art. To the best of our knowledge, there are only two implementations with subconstant interaction times available, namely (i) the C++ reference implementation of [2], to which we refer as Pps, and (ii) the Cython implementation of the same algorithm for the Ppsim framework [8] focusing on user-friendliness. We consider only Pps, as it outperformed Ppsim in preliminary experiments. We also include $\text{Seq}^{\text{Linear}}$ of [2], here referred to as Seq, as a highly engineered discrete interaction simulator.

We implement Gps in Rust and prioritize insights into the algorithmic framework over being highly optimized for production use. As such, we include internal profiling tools and avoid heuristics that change the scaling behavior. For instance, the Pps features control loops to auto-tune parameters on the fly and to adapt to changing state histograms. In contrast, Gps uses only preset values to reduce variance in our measurements. To further ease the comparison between Pps and Gps, we port the single-most expensive primitive, namely sampling from the Hypergeometric distribution, from C++ to Rust.[3]

[3] The code of [2] is adopted from numpy's ratio-of-uniforms method [16]. We only add a special case for small populations or taking few samples, which occurs much more frequently in Gps.

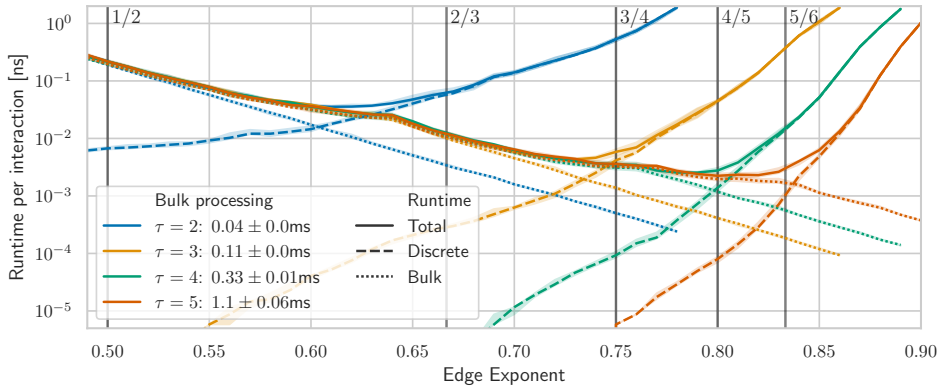

Figure 4 Execution of GPS with $n = 2^{35}$ agents, $|Q| = 8$ states, and $\hat{m} = n^{\gamma}$ edges per epoch using a timeout of 1000s. The legend reports the average runtime of bulk processing per epoch.

If not stated differently, we use $n = 2^{35}$ agents and the `random` protocol (see below) starting from a uniform histogram of $|Q| = 8$ states [2]. We repeat each measurement exactly 10 times and report the mean of the measurements. All plots indicate the 95% confidence interval by the light shaded area; they are typically too small to be visible. We often report the *time per interaction* t/I, where t is the wall time of a simulation of $I = 10n$ interactions.

The `random` protocol forces a large number of updates to the state histogram, while keeping the distribution balanced on average. This simplifies comparing PPS and GPS as it counteracts state-reorder heuristics of PPS. It is implemented as a lookup table mapping each state pair to a state pair drawn uniformly at random.

6.1 Number of interactions per epoch

We start by analyzing the effect of the two primary tuning parameters of GPS, namely the number of edges per epoch m and the number of nodes τ in the largest tree that we execute using bulk processing. Recall that m is sampled from an almost Poisson distribution with expected value $\approx \hat{m} = n^{\gamma}$ – hence we control m indirectly in terms of the exponent γ.

To maximize the number of interactions per epoch, γ should be maximized. However, this leads to more complex components and thereby requires more costly bulk processing flows. Specifically, our implementation uses 1, 4, 8, and 18 fragments for $\tau = 2$ to 5, respectively. To investigate this interaction of parameters τ and γ, we execute the algorithm for $\tau \in [2, 3, 4, 5]$ and $0.49 \leq \gamma \leq 0.9$ and record the total runtime, as well as the contributions of the bulk processing path (BPP) and the discrete interaction path. Averaged over the all measurements, these two account for 91.6% of total runtime. The results are summarized in Figure 4.

Let us first consider the **bulk processing engine** (τ-BPP) using trees with up to τ nodes. As established in Lemma 7, we expect $\mathcal{O}(1)$ edges incompatible with a τ-BPP for $\gamma \leq (\tau-1)/\tau$. This suggests that τ-BPP is only efficient for $\gamma \geq (\tau-1)/\tau$, which is supported by the measurements. For all values of τ, the τ-BPP runtime *per epoch* approaches a constant reported in the legend of Figure 4. This empirically confirms our simulator's key idea, namely that the BPP's runtime is independent of the number of interactions it carries out.

The **discrete interaction engine** (DIP) executes all interactions that are not handled by the τ-BPP. Our implementation combines the sampling of the 2-core (Algorithm 6) and the execution of discrete interactions. Consistently, we report their combined runtime here; averaged over all measurements, the execution accounts for 69.1% of the reported times.

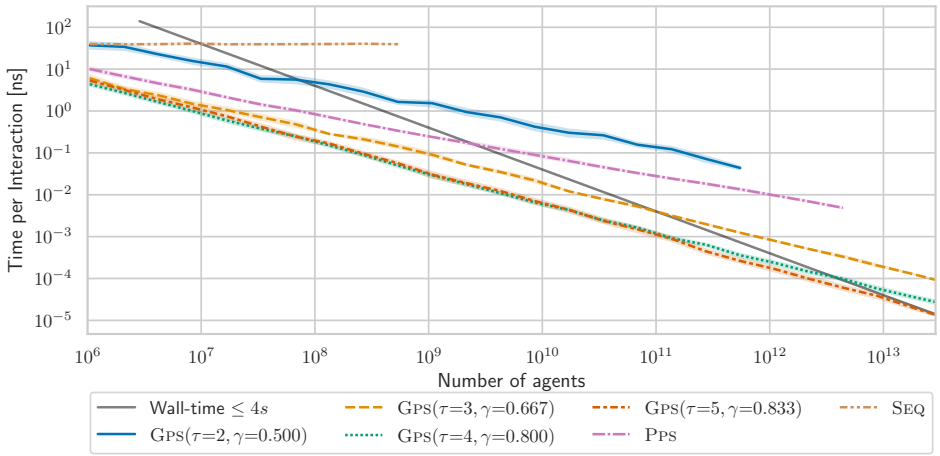

Figure 5 Runtime of GPS, PPS, andSEQ. Protocol: `random` with $|Q| = 8$. Timeout 150s. The grey line indicates the number of agents n, such that $I = 10n$ interactions can be simulated with 4s.

Our measurements align with the analytical prediction: The runtime of DIP scales linearly in the interactions handled by it. Using a τ-BPP for $\gamma \approx (\tau-1)/\tau$ only leaves the remaining $\mathcal{O}(1)$ interactions for DIP. Hence, in this parameter regime, the DIP's runtime is at least one order of magnitude lower than that of τ-BPP. However, our measurements suggest that γ should be chosen slightly below $\tau/(\tau+1)$: In this regime, the increased runtime of DIPis still overcompensated for by the increased number of interactions handled by the τ-BPP.

6.2 Number of agents

Figure 5 summarizes the simulators' scaling behaviors of the time per interaction in the number n of agents. As expected, we observe a constant time per interaction of 39.6ns for SEQ, which scales linearly in the number of iterations I (chosen as $I = 10n$).

For both PPS and GPS($\tau = 2, \gamma = 1/2$), we expect the time per interaction to scale as $\mathcal{O}(1/\sqrt{n})$ which is consistent with our measurements. The run of GPS($\tau = 2, \gamma = 1/2$) exhibits a relatively high noise level. We believe that this is because epochs are quite short in this regime, so that overheads (e.g., to sample the interaction graph) matter much more. As these components are not substantial for the intended use, they are less optimized and, for instance, use more frequent heap allocations. Comparing PPS to GPS($\tau = 2, \gamma = 1/2$) we see that PPS is 5.2 ± 0.7 times faster. This is due to the significantly higher overheads incurred by the more complex GPS that cannot be amortized here. Additionally, PPS operates at its ideal (auto-tuned) parameter settings, while $\tau = 2$ is a mostly irrelevant niche case for GPS.

For all $\tau > 2$, GPS outperforms the competitors – the measured times per interaction are consistent with the scaling predicted by Theorem 1. Let us consider $\tau = 5$ and $\gamma = 5/6$. The largest system that PPS can consistently simulate in less than 150s is at $n = 2^{40}$. At this point, GPS is already 849.7 times faster. Going even bigger, we measure a total runtime of 4.00s for $n = 2^{45}$ – this corresponds to (11 ± 1) femtoseconds per interaction.

As we expect practitioners to select their systems sizes as large as possible to be simulated with a fixed time budget, we indicate the aforementioned 4s barrier in Figure 5. Given this time budget, PPS can simulate a system of agents 2^{31}, which is 2^{14} times smaller than GPS.

7 Conclusion and outlook

We propose the graphical population protocol simulator Gps. It samples the component structure of an interaction graph and uses it to bulk process interactions in expected time $\mathcal{O}(\frac{I}{n}|Q|^{2-2/\tau}n^{1/\tau})$ where τ is a tuning parameter. Gps builds on our novel configuration model sampler, which scales in the number of component types (rather than nodes) in the relevant regime. Our implementation of Gps translates these asymptotic improvements into significant real-world speed-up and outperforms the state-of-the-art even for moderately sized populations. Given a fixed time budget of 4s, it simulates systems four orders of magnitude faster than its competitors.

While we show correctness and bound the runtime of our approach, our analysis is not sufficiently tight to suggest an optimal choice of epoch length and τ given n agents on $|Q|$ states. If the transition function is treated as a black box and updating agents takes time $\Omega(1)$, it is known that executing $I = n$ interactions takes $\Omega(\frac{\log(n)}{\log\log(n)})$ parallel steps, and more generally $\Omega(\frac{I}{n})$ parallel steps for $I = \Omega(n \log n)$ [7]. However, this only accounts for the execution of the protocol when given a decomposition of the schedule into a DAG tracking the dependencies between interactions, so it is open to what extent our $\mathcal{O}(|Q|^{2-2/\tau}n^{1/\tau})$ overhead can be reduced.

Lastly, we expect that our techniques are applicable beyond population protocols. For instance, we leave open whether Gps can be adopted to Gillespie simulations [10] of chemical reaction networks in the same spirit as [15] did for [2]. Another natural question is how to extend Gps to interactions with more than two agents, e.g., using interaction hyper-multigraphs.

References

1 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Comput.*, 18(4):235–253, 2006. doi:10.1007/S00446-005-0138-3.

2 Petra Berenbrink, David Hammer, Dominik Kaaser, Ulrich Meyer, Manuel Penschuck, and Hung Tran. Simulating population protocols in sub-constant time per interaction. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 16:1–16:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.16.

3 Béla Bollobás. A probabilistic proof of an asymptotic formula for the number of labelled regular graphs. *Eur. J. Comb.*, 1(4):311–316, 1980. doi:10.1016/S0195-6698(80)80030-8.

4 Béla Bollobás. The evolution of sparse graphs. *Graph theory and combinatorics (Cambridge, 1983)*, pages 35–57, 1984.

5 Carl Borchardt. Über eine Interpolationsformel für eine Art symmetrischer Functionen und über deren Anwendung. *Mathematische Abhandlungen der Königlichen Akademie der Wissenschaften zu Berlin*, 1860.

6 Arthur Cayley. A theorem on trees. *Quarterly J. of Pure and Applied Mathematics*, 1889.

7 Artur Czumaj and Andrzej Lingas. On parallel time in population protocols. *Inf. Process. Lett.*, 179:106314, 2023. doi:10.1016/J.IPL.2022.106314.

8 David Doty and Eric Severson. Ppsim: A software package for efficiently simulating and visualizing population. In *Computational Methods in Systems Biology: 19th International Conference, CMSB 2021, Bordeaux, France, September 22–24, 2021, Proceedings*, page 245. Springer Nature, 2021.

9 Bailey K. Fosdick, Daniel B. Larremore, Joel Nishimura, and Johan Ugander. Configuring random graph models with fixed degree sequences. *SIAM Review*, 60(2):315–355, 2018. doi:10.1137/16M1087175.

10 Daniel T Gillespie. Exact stochastic simulation of coupled chemical reactions. *The journal of physical chemistry*, 81(25):2340–2361, 1977.

11 Voratas Kachitvichyanukul and Bruce W Schmeiser. Binomial random variate generation. *Communications of the ACM*, 31(2):216–222, 1988. doi:10.1145/42372.42381.

12 Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.

13 Daniel Lemire. Fast random integer generation in an interval. *ACM Trans. Model. Comput. Simul.*, 29(1):3:1–3:12, 2019. doi:10.1145/3230636.

14 Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

15 Joshua Petrack and David Doty. Exactly simulating stochastic chemical reaction networks in sub-constant time per reaction, 2026. doi:10.48550/arXiv.2508.04079.

16 Ernst Stadlober. Ratio of uniforms as a convenient method for sampling from classical discrete distributions. In *WSC*, pages 484–489. ACM Press, 1989. doi:10.1145/76738.76801.

A Auxiliary results

▶ **Lemma 7.** *Let G be the interaction multigraph of a schedule of m interactions chosen u.a.r., and let $\mathcal{E}$ be the multiset of edges of G lying in components containing a cycle (including loops or double-edges) or $\tau+1$ or more nodes. Then*

$$\mathbb{E}[|\mathcal{E}|] \leq \frac{2^\tau e^{\tau+1}}{\tau+1}\frac{m^\tau}{n^{\tau-1}} + \frac{2em/n}{(1-2em/n)^2}.$$

Proof. Let X_1 be the number of edges in component containing at least $\tau+1$ or more nodes, and X_2 be the number of edges in components containing a cycle. Clearly $|\mathcal{E}| \leq X_1 + X_2$, so we will bound the expected values of the latter.

For X_1, see first that every edge contained in a component containing $\tau+1$ or more nodes is also contained in a subgraph which is a tree on $\tau+1$ nodes. For each set of $\tau+1$ nodes, there are $(\tau+1)^{\tau-1}$ labeled trees on that set of nodes, so there are $\binom{n}{\tau+1}\cdot(\tau+1)^{\tau-1}$ subgraphs to consider. Let $e_1,\dots,e_\tau$ be the edges of a particular tree. Then the probability that all of these edges are contained in the graph is at most $(m)_\tau(2/n^2)^\tau$, as there are $(m)_\tau$ possible placements for the edges in the schedule, and the probability of an edge being in a particular place in the schedule is $2/n^2$. So

$$\begin{aligned}\mathbb{E}[X_1] &\leq (\tau+1)\cdot\binom{n}{\tau+1}(\tau+1)^{\tau-1}\cdot(m)_\tau\cdot\left(\frac{2}{n^2}\right)^\tau \\ &\leq (\tau+1)\cdot\frac{e^{\tau+1}n^{\tau+1}}{(\tau+1)^{\tau+1}}\cdot(\tau+1)^{\tau-1}\cdot m^\tau\cdot\frac{2^\tau}{n^{2\tau}} = \frac{2^\tau e^{\tau+1}}{\tau+1}\cdot\frac{m^\tau}{n^{\tau-1}}.\end{aligned}$$

For X_2, note that if an edge e (including loops) is contained in a component containing a cycle, there is some $1\leq k\leq n$ so that e is contained in a unicyclic connected subgraph, i.e., a subgraph which is a tree with an additional edge. For each set of k nodes, there are k^{k-2} labeled trees on that set of nodes, and there are at most k^2 possible edges to add. So there are at most $\binom{n}{k}\cdot k^k$ subgraphs to consider. The rest of the calculation is as above; so overall

$$\mathbb{E}[X_2] \le \sum_{k=1}^{n} k \cdot \binom{n}{k} \cdot k^k \cdot (m)_k \cdot \left(\frac{2}{n^2}\right)^k \le \sum_{k=1}^{n} \frac{e^k n^k}{k^k} \cdot k^{k+1} \cdot m^k \cdot \frac{2^k}{n^{2k}}$$
$$= \sum_{k=1}^{n} 2^k e^k k \cdot \left(\frac{m}{n}\right)^k = \sum_{k=1}^{n} k \cdot \left(\frac{2em}{n}\right)^k \le \frac{2em/n}{(1-2em/n)^2}.$$

with the last inequality holding if $2em \le n$. ◀

▶ **Lemma 8** (Rephrasing of Thereom 5.6 in [14]). *Let $\vec{X} \sim \mathrm{Mult}(k, \vec{1}/n)$, and let $\vec{Y} = (Y_i)_{i\in[n]}$ with the $Y_i \sim \mathrm{Pois}(\lambda)$ being i.i.d. for some parameter $\lambda > 0$. Then the distribution of $\vec{X}$ is the same as that of $\vec{Y}$ conditioned on $\sum_i Y_i = k$, i.e., for any $\vec{x} \in \mathbb{N}_0^n$ with $\|\vec{x}\|_1 = k$,*

$$\mathbb{P}[\vec{X} = \vec{x}] = \mathbb{P}[\vec{Y} = \vec{x} \mid \|\vec{Y}\|_1 = k].$$

▶ **Corollary 9.** *Let $\lambda > 0$, and let $K \sim \mathrm{Pois}(n\lambda)$, $\vec{X} \sim \mathrm{Mult}(K, \vec{1}/n)$, and $\vec{Y} = (Y_i)_{i\in[n]}$ with the $Y_i \sim \mathrm{Pois}(\lambda)$ being independent. Then for any nonempty $S \subseteq \mathbb{N}_0$ and any $\vec{x} \in \mathbb{N}_0^n$,*

$$\mathbb{P}[\vec{X} = \vec{x} \mid K \in S] = \mathbb{P}[\vec{Y} = \vec{x} \mid \|\vec{Y}\|_1 \in S].$$

Proof. Let $k = \|\vec{x}\|_1$. Now $\mathbb{P}[\vec{Y} = \vec{x} \mid \|\vec{Y}\|_1 = i]$ and $\mathbb{P}[\vec{X} = \vec{x} \mid K = i]$ are definitely zero for $i \neq k$. And so by the law of total probability, Lemma 8, and since $\|\vec{Y}\|_1 \sim \mathrm{Pois}(n\lambda)$,

$$\mathbb{P}[\vec{X} = \vec{x} \mid K \in S] = \mathbb{P}[\vec{X} = \vec{x} \mid K = k] \cdot \mathbb{P}[K = k \mid K \in S]$$
$$= \mathbb{P}[\vec{Y} = \vec{x} \mid \|\vec{Y}\|_1 = k] \cdot \mathbb{P}[\|\vec{Y}\|_1 = k \mid \|\vec{Y}\|_1 \in S] = \mathbb{P}[\vec{Y} = \vec{x} \mid \|\vec{Y}\|_1 \in S].$$ ◀

B Deferred proofs

Proof of Observation 2. The claim is trivially true at the beginning of the process as no edges are marked. Then, an edge e is marked only if both ends were unmarked before (since nodes get marked only when all its adjoining edges are marked). So by induction they must have belonged to different components of the subgraph of marked edges, meaning that the subgraph remains a forest. One of e's ends v gets marked, so there is still at most one unmarked node in the merged component. And since e was the only outgoing edge from v's old component to the remainder of the merged component, the latter is an induced subgraph. ◀

Proof of Lemma 3. For $X \sim \mathrm{Pois}(2\hat{m})$, we need to bound $\mathbb{E}[M] = \mathbb{E}\left[\frac{X}{2} \,\middle|\, \frac{X}{2} \in \mathbb{N} \cap [0, 2\hat{m}]\right]$. To that end, let us first determine the probability that $\frac{X}{2}$ is an integer:

$$\mathbb{P}\left[\frac{X}{2} \in \mathbb{N}\right] = \sum_{i=0}^{\infty} \frac{(2\hat{m})^{2i} e^{-2\hat{m}}}{(2i)!} = e^{-2\hat{m}} \cosh(2\hat{m}) = \frac{1}{2} + \frac{e^{-4\hat{m}}}{2}.$$

So then the expectation of $\frac{X}{2}$ conditioned on that being an integer is

$$\mathbb{E}\left[\frac{X}{2} \,\middle|\, \frac{X}{2} \in \mathbb{N}\right] = \sum_{i=0}^{\infty} i \cdot \mathbb{P}\left[\frac{X}{2} = i \,\middle|\, \frac{X}{2} \in \mathbb{N}\right] = \sum_{i=0}^{\infty} i \cdot \frac{\mathbb{P}[X = 2i]}{\mathbb{P}\left[\frac{X}{2} \in \mathbb{N}\right]}$$
$$= \frac{1}{\mathbb{P}\left[\frac{X}{2} \in \mathbb{N}\right]} \cdot \sum_{i=0}^{\infty} i \cdot \frac{(2\hat{m})^{2i} e^{-2\hat{m}}}{(2i)!} = \frac{e^{-2\hat{m}} \hat{m} \sinh 2\hat{m}}{e^{-2\hat{m}} \cosh 2\hat{m}}$$
$$= \hat{m} \cdot \frac{e^{2\hat{m}} - e^{-2\hat{m}}}{e^{2\hat{m}} + e^{-2\hat{m}}} = \hat{m} \cdot \left(1 - \frac{2e^{-2\hat{m}}}{e^{2\hat{m}} + e^{-2\hat{m}}}\right) = \hat{m}\left(1 - \Theta(e^{-4\hat{m}})\right).$$

Clearly $\mathbb{E}[M] = \mathbb{E}\left[\frac{X}{2} \,\middle|\, \frac{X}{2} \in \mathbb{N} \cap [0, 2\hat{m}]\right] \leq \mathbb{E}\left[\frac{X}{2} \,\middle|\, \frac{X}{2} \in \mathbb{N}\right]$, so this is the claimed upper bound.

For the lower bound, see first that

$$\begin{aligned}
\mathbb{E}[M] &= \mathbb{E}\left[\frac{X}{2} \,\middle|\, \frac{X}{2} \in \mathbb{N} \cap [0, 2\hat{m}]\right] = \sum_{i=0}^{\infty} i \cdot \mathbb{P}\left[\frac{X}{2} \,\middle|\, \frac{X}{2} \in \mathbb{N} \cap [0, 2\hat{m}]\right] \\
&= \sum_{i=0}^{2\hat{m}} i \cdot \frac{\mathbb{P}[X = 2i]}{\mathbb{P}\left[\frac{X}{2} \in \mathbb{N} \cap [0, 2\hat{m}]\right]} = \frac{1}{\mathbb{P}\left[\frac{X}{2} \in \mathbb{N} \cap [0, 2\hat{m}]\right]} \sum_{i=0}^{2\hat{m}} i \cdot \frac{(2\hat{m})^{2i} e^{-2\hat{m}}}{(2i)!} \\
&\geq \frac{\sum_{i=0}^{\infty} i \cdot \frac{(2\hat{m})^{2i} e^{-2\hat{m}}}{(2i)!} - \sum_{i=4\hat{m}+1}^{\infty} \frac{i}{2} \cdot \frac{(2\hat{m})^{i} e^{-2\hat{m}}}{i!}}{\mathbb{P}\left[\frac{X}{2} \in \mathbb{N}\right]} \\
&= \mathbb{E}\left[\frac{X}{2} \,\middle|\, \frac{X}{2} \in \mathbb{N}\right] - \frac{\sum_{i=4\hat{m}+1}^{\infty} \frac{i}{2} \cdot \frac{(2\hat{m})^{i} e^{-2\hat{m}}}{i!}}{e^{-2\hat{m}} \cosh(2\hat{m})}
\end{aligned}$$

And

$$\begin{aligned}
\sum_{i=4\hat{m}+1}^{\infty} \frac{i}{2} \cdot \frac{(2\hat{m})^{i} e^{-2\hat{m}}}{i!} &= \frac{1}{2} \cdot \sum_{i=4\hat{m}+1}^{\infty} \frac{(2\hat{m}) \cdot (2\hat{m})^{i-1} e^{-2\hat{m}}}{(i-1)!} = \frac{1}{2} \cdot 2\hat{m} \cdot \sum_{i=4\hat{m}}^{\infty} \frac{(2\hat{m})^{i} e^{-2\hat{m}}}{i!} \\
&= \hat{m} \cdot \mathbb{P}[\mathrm{Pois}(2\hat{m}) \geq 4\hat{m}].
\end{aligned}$$

We bound $\mathbb{P}[X \geq 4\hat{m}]$ using a standard Chernoff bound for Poisson random variables [14, Theorem 5.4]: For $\delta > 1$,

$$\mathbb{P}[\mathrm{Pois}(\lambda) \geq (1+\delta)\lambda] \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\lambda}.$$

So in our case, for $\lambda = 2\hat{m}$ and $\delta = 1$, we get that $\mathbb{P}[X \geq 4\hat{m}] \leq \left(\frac{e}{4}\right)^{2\hat{m}}$. And so combining everything, we get

$$\begin{aligned}
\mathbb{E}[M] &\geq \hat{m} \frac{\sinh(2\hat{m})}{\cosh(2\hat{m})} - \frac{\hat{m}(e/4)^{2\hat{m}}}{e^{-2\hat{m}} \cosh(2\hat{m})} = \hat{m} \cdot \left(1 - \frac{2e^{-2\hat{m}}}{e^{2\hat{m}} + e^{-2\hat{m}}} - \frac{(e/4)^{2\hat{m}}}{(1 + e^{-4\hat{m}})/2}\right) \\
&= \hat{m} \cdot \left(1 - \Theta(e^{-4\hat{m}}) - \Theta((e/4)^{2\hat{m}})\right),
\end{aligned}$$

as claimed. ◀

B.1 Proof of Lemma 4

To prove Lemma 4, we require the following property of the calls to ProcessLeafKind and Refine:

▶ **Lemma 10.** *During an execution of SampleInteractionGraph, there are only finitely many calls to ProcessLeafKind and Refine. Additionally, ProcessLeafKind(k) is called at most once for every kind $k = (1, T_k)$, and for every kind k whose tree is non-trivial, there is a unique k' so that k is created in Refine(k').*

Proof. To see that the loop must terminate, see that the sum $2|\mathcal{S}| + \sum_{k \in K} |\mathcal{P}_k| \geq 0$ is strictly decreasing in each call to the two procedures in question: ProcessLeafKind(k) removes all $m_{\mathcal{S}}(k) > 0$ copies of $k = (1, T_k)$ from $\mathcal{S}$, and adds at most that many copies of T_k to all $\mathcal{P}_t$ combined, so that the sum decreases by at least $m_{\mathcal{S}}(k)$. And in Refine(k), $\mathcal{P}_k$ is decreased from some $|\mathcal{P}_k| > 0$ to 0, but $|\mathcal{S}|$ stays the same; this is because the $m_{\mathcal{S}}(k)$ copies of k are first deleted from $\mathcal{S}$, and this is exactly the number of placeholders $\perp$ added to $\mathcal{P}_k$, and *that* is exactly the sum of residual degrees (accounting for multiplicities) in the new kinds created in Refine(k).

So now assume that the claim that ProcessLeafKind(k) is called at most once for every kind $k = (1, T_k)$. Then there must be a first time where there is a second call to ProcessLeafKind(k) for such a k. After the first call, k is removed from K. So for the second call to happen, k must be readded to K during a call to Refine(k_2) happening inbetween the two calls, for some $k_2 \neq k$. Now T_k cannot be the trivial tree (and thus k be present before any calls to Refine) because it must be possible to create it during a call of Refine(k_2), so that its root must have some children. So there must have been a kind $k_1 \neq k$ so that before the first call of ProcessLeafKind(k), k was added to K in a call to Refine(k_1); so it suffices to show the final claim of the lemma statement under the assumption that in the time span in question, the k for which ProcessLeafKind(k) are called are unique.

It cannot be the case that $k_1 = k_2$, i.e., that before the second call ProcessLeafKind(k), there were two calls Refine(k_1): After the first call, $\mathcal{P}_{k_1}$ is empty. So all trees in $\mathcal{P}_{k_1}$ at the time of the second call are trees T where ProcessLeafKind($(1, T)$) was called in-between the first and second Refine(k_1) calls. But these must all be distinct from the trees where this was the case *before* the first Refine(k_1) call, because by choice of k, before the second call of ProcessLeafKind(k), all kinds for which ProcessLeafKind was called were unique. But hence, the kinds added to K during the second call of Refine(k_1) (except for k_1 itself) must be different from those added to K during the first call; a contradiction.

And so we must have $k_1 \neq k_2$. Because both Refine(k_1) and Refine(k_2) can add k to K, the sum of k_1's residual degree and the number of children in its tree must be the same as that of k_2. So in any case, $T_{k_1} \neq T_{k_2}$ and hence there is some child T_c of T_{k_1}'s root which occurs more often as a child there than in T_{k_2}'s, or vice versa; w.l.o.g. assume the former. But then they cannot be both created in the same Refine(k') call for some kind $k' \notin \{k_1, k_2\}$: This is because just as above, the kinds for which ProcessLeafKind was called before the call to Refine(k') are different from those after (in the time span we care about) by choice of k. Clearly, T_c must have been in $\mathcal{P}_{k'}$ at the time of the call of Refine(k'). But T_c must be in $\mathcal{P}_{k_2}$ at the time of the call to Refine(k_2), which can only have been added after Refine(k'), a contradiction. So there must be $k_1' \neq k_2'$ so that k_1 was added to K during a call Refine(k_1'), and k_2 was added to K during a call Refine(k_2'). But by the same argument, there must then be $k_1'' \neq k_2''$ so that k_1' was added to K during a call Refine(k_1''), and k_2' was added to K during a call Refine(k_2''), and so on. So there must have been infinitely many calls to Refine, contradicting our earlier observation that the loop must terminate. ◀

Proof of Lemma 4. As already discussed in Section 4.1, the time to sample a potential degree histogram (before rejection) is bounded by the maximum of the i.i.d. Poisson r.v.s, whose expectation is on the same order as that for a non-rejected histogram. And the number of times T it takes to rejection is a stopping time which follows a Geometric distribution with the parameter being the acceptance probability, which is $\frac{1}{2} + o(1)$ for $\hat{m} = \omega(1)$ as seen in the proof of Lemma 3, and the latter is the case here. So $\mathbb{E}[T] = O(1)$. So by Wald's equation [14, Theorem 13.3], the expected runtime for this step is $\mathbb{E}[\Delta]$. The rest of the initialization (before the central loop) has the same asymptotic runtime as well because it is linear in Δ.

Now, by Lemma 10, ProcessLeafKind(k) is called at most once for each kind $k = (1, T_k)$, and for every k there is a unique $k' \neq k$ where stubs of kind k are created in calls to Refine(k'). Now we only call ProcessLeafKind on kinds $(1, T_k)$ with $1 + |T_k| \leq \tau$, of which there are finitely many (with the number of them a function of τ); a loose bound is that there are $(\tau - 1)^{\tau - 3}$ labeled trees on $\tau - 1$ nodes, and thus at most that many such

kinds. These are the only kinds which can appear in the pools $\mathcal{P}_{k'}$; and since in the central loop of SampleInteractionGraph, Refine is only called on kinds $k' = (d_k, T_k)$ with $d_k + |T_k| \leq \tau$, of which there are again only finitely many (with the number a function of τ), the number of kinds which can possibly be created is also bounded by a function in τ. And as Refine(k') is only called when $\mathcal{P}_{k'}$ is non-empty, and each such call must produce a stub of some kind k for which no stub has been created before, the number of possible Refine calls in the central loop is also bounded by a function in τ.

The runtime of ProcessLeafKind is at most linear in the size of K (i.e., the number of kinds present in $\mathcal{S}$), so again a function only of τ. For Refine, the runtime is dominated by that of the multiway matching step. But even here, the maximum degree of such a matching is τ, and the number of different elements in a pool is a function of τ, so that the runtime of any Refine during the central loop can again by bounded just in τ.

It remains to consider the runtime of SampleRemainingGraph. Now any remaining kinds $k = (d_k, T_k)$ must have $d_k + |T_k| \geq \tau + 1$, and so the associated node must lie in a component that either has a cycle or has $\tau + 1$ or more nodes. The number of stubs at such nodes is at most twice the size of the multiset $\mathcal{E}$ of such edges, which is bounded in expectation by Lemma 7.

For the Refines for remaining kinds with nonempty pools $\mathcal{P}_k$, recall that the multiway matching step which dominates the runtime of Refine can be done in time linear to the multiset of elements which is being matched (i.e., the pool $\mathcal{P}_k$). And the total size of all such pools taken together is at most the number of stubs at the associated nodes, so at most $2|\mathcal{E}|$ by the discussion in the previous paragraph. So by linearity of expectation, the bound on the expectation of $\mathcal{E}$ Lemma 7 also bounds the expected time of these Refine steps (up to constants).

And the time to execute the rest of SampleRemainingGraph is clearly also linear in $\mathcal{E}$, and thus, taking everything together, we have the claim. ◀

B.2 Proof of Theorem 1

Set $\hat{m} = |Q|^{2/\tau} n^{(\tau-1)/\tau}$. As $|Q| = o(\sqrt{n})$, we have $\hat{m} = o(n)$.

We execute *epochs* of the protocol until we have reached I or more interactions. To do this, we sample an interaction graph using the routine SampleInteractionGraph described in Section 4, and then execute it as described in Section 5. For $\mathcal{E}$ being the set of edges in the interaction multigraph which are contained in components having $\tau + 1$ or more nodes or a cycle (including loops and multiedges), and Δ being the maximum degree of the interaction graph, the runtime of the sampling of the interaction graph is in $\mathcal{O}(\mathbb{E}[|\Delta|] + f(\tau) + |\mathcal{E}|)$ in expectation by Lemma 4, with the expected execution taking time being in $\mathcal{O}(g(\tau)|Q|^2 + |\mathcal{E}|)$ by Lemmas 5 and 6, where f and g are some functions.

Now for $\hat{m} = \mathcal{O}(n)$, $\mathbb{E}[\Delta]$ is well-known to be in $\mathcal{O}(\log n)$ in expectation; And for $\hat{m} = \mathcal{O}(n^{1-1/\tau} \log(n))$ (so when $|Q|^{2/\tau} = \mathcal{O}(\log(n))$), a standard union-bound argument using the super-exponential tail of the degree distribution yields that $\mathbb{E}[\Delta]$ is in $\mathcal{O}(\tau)$. So overall, we definitely have $\mathbb{E}[\Delta] = \mathcal{O}(\tau|Q|^2)$ in either case.

The number of edges in the multigraph produced by SampleInteractionGraph is guaranteed to be at most $2\hat{m}$, and hence by Lemma 7,

$$\mathbb{E}[|\mathcal{E}|] \leq \frac{2^\tau e^{\tau+1}}{\tau+1} \cdot \frac{(2\hat{m})^\tau}{n^{\tau-1}} + \frac{4e\hat{m}/n}{(1-4e\hat{m}/n)^2} = h(\tau)|Q|^2 + o(1)$$

for some function $h(\tau)$. By linearity of expectation, this means that the time taken per epoch is in $\mathcal{O}(\hat{f}(\tau)|Q|^2)$ for some function $\hat{f}$.

For the number of epochs needed, first note that by Lemma 3 each epoch has length $\hat{m}(1 - o(1))$ in expectation, and that again the length is guaranteed to be at most $2\hat{m}$. So letting M_i be the i.i.d. epoch lengths, and T the number of epochs needed to reach $\geq I$ interactions, by Wald's equation [14, Theorem 13.3],

$$\mathbb{E}\left[\sum_{i=1}^{T} M_i\right] = \mathbb{E}[T] \cdot \mathbb{E}[M_i].$$

As $M_i \leq 2\hat{m}$ for all M_i, we have $\sum_{i=1}^{T} M_i \leq I + 2\hat{m}$ by definition, and so

$$\mathbb{E}[T] \leq \frac{I + 2\hat{m}}{\hat{m}(1 - o(1))} = \mathcal{O}\left(\frac{I}{\hat{m}} + 1\right) = \mathcal{O}\left(\frac{I}{\hat{m}}\right)$$

with the last step using that $I = \Omega(n) = \omega(\hat{m})$.

While the times to execute the epochs are not independent (as they may depend on, e.g., the number of states present at the beginning of epochs), our bounds are worst-case in the sense that they hold for *any* initial state of the protocol at the beginning of the epoch. Hence, they are bounds on the expected value of i.i.d. random variables which all stochastically dominate the times taken in the respective epochs. So we may apply Wald's equation again to see that the total expected time taken is the product of the expected number of epochs and the bound on the expected time taken per epoch, which is

$$\mathcal{O}\left(\frac{I}{\hat{m}} \cdot \hat{f}(\tau)|Q|^2\right) = \mathcal{O}\left(\frac{I}{|Q|^{2/\tau} n^{(\tau-1)/\tau}} \cdot \hat{f}(\tau)|Q|^2\right) = \mathcal{O}\left(\frac{I}{n} \cdot \hat{f}(\tau)|Q|^{2-2/\tau} n^{1/\tau}\right),$$

just as claimed.