Soc. Netw. Anal. Min. (2017)7:4
DOI 10.1007/s13278-017-0424-7

ORIGINAL ARTICLE

Hashkat: large-scale simulations of online social networks

Kevin Ryczko[1] · **Adam Domurad**[2] · **Nicholas Buhagiar**[3] · **Isaac Tamblyn**[4]

Received: 7 October 2016 / Revised: 20 January 2017 / Accepted: 22 January 2017

Abstract Hashkat (http://hashkat.org) is a free, open source, agent-based simulation software package designed to simulate large-scale online social networks (e.g., Twitter, Facebook, LinkedIn). It allows for dynamic agent generation, edge creation, and information propagation. The purpose of hashkat is to study the growth of online social networks and how information flows within them. Like real-life online social networks, hashkat incorporates user relationships, information diffusion, and trending topics. Hashkat was implemented in C++ and was designed with extensibility in mind. The software includes Bash and Python scripts for easy installation and usability. In this report, we describe all of the algorithms and features integrated into hashkat before moving on to example use cases. In general, hashkat can be used to understand the underlying topology of social networks, validate sampling methods of such networks, develop business strategy for advertising on online social networks, and test new features of an online social network before going into production.

✉ Kevin Ryczko
kevin.ryczko@uoit.net

Adam Domurad
adomurad@uwaterloo.ca

Nicholas Buhagiar
nbuhagiar@ryerson.ca

Isaac Tamblyn
isaac.tamblyn@nrc.ca

1 University of Ontario Institute of Technology, Oshawa, Canada

2 University of Waterloo, Waterloo, Canada

3 Ryerson University, Toronto, Canada

4 National Research Council of Canada, Ottawa, Canada

Keywords Agent-based modeling · Kinetic Monte Carlo · Online social network · Network evolution · Information propagation · Simulation

1 Introduction

Hashkat is a simulation package designed to study the growth and time evolution of online social networks. As online social networks continue to grow in relevance, it has become increasingly important to quantitatively analyze their behavior. We show that hashkat can be used to produce existing analytical graph models, as well as new, unstudied graphs with similar topologies to online social networks. Hashkat falls under the field of agent-based social simulations (Li et al. 2008). Hashkat takes a kinetic Monte Carlo (kMC) approach, allowing a user to explore the characteristics of online social networks as they evolve through time. A wealth of recent work (Gleeson et al. 2014; Brach et al. 2014; Myers and Leskovec 2014; Doerr et al. 2011) has focused on analyzing the topology of online social networks, in order to make inferences about information flow, interaction mechanisms, and network stability. Brach et al. (2014) were able to make predictions about the behavior of information diffusion within social networks based on a random network topology. The model they developed described the evolution of rumors on social networks and gave qualitatively good results with respect to how messages propagate within Twitter. Gleeson et al. (2014) focused on meme diffusion for a directed social network. They found that the popularity growth of each meme can be described by a critical branching process. The popularity distributions of the memes had heavy tails similar to the distribution of links on the Internet (Albert et al. 1999) as well many degree distributions for online social networks (Mislove et al. 2007; Ugander et al. 2011; Java et al. 2007).

Currently, there are several commercial and non-commercial simulation packages available that simulate some aspect of online social networks. SMSim (de C Gatti et al. 2013) simulates the network surrounding one agent and studies how information propagates from the central agent throughout the local neighborhood. SeSAm (Klügl and Puppe 1998) is able to treat a moderately small number of agents (10s of 1000s) with an agent-based modeling approach. Zeng et al. (2015) discussed a simulation method where random sampling techniques were used to build the initial network. Once the initial network was constructed, they then used a "close degree algorithm" to obtain snapshots of the social network at certain points in time. An R package called NetSim (Stadtfeld et al. 2013) allows users to simulate the coevolution of social networks and individual attributes. The engine of this package uses a generic Markov model, and the simulation incorporates social pressure (a user will connect to friends of a user they are connected to) and likeness of agents when making connections.

Hashkat represents a significant step forward in the area of simulating online social behavior due to its combination of scalability and features. Despite the complexity of the numerous features integrated into hashkat, it simplifies the user-defined modeling process to the determination of rates for a given system. If the rates of events in a system are known (and correct), hashkat will time evolve the system along the true trajectory.

In this report, we first describe the design and high-level structure of the algorithms used within hashkat. This includes a description of all events that can occur in the model. We then use hashkat to reproduce existing analytical graph models as test cases. This includes constructing random and preferential attachment graphs. Lastly, we walk through example use cases for which no analytical solutions are known.

2 Algorithms, design, and features of hashkat

2.1 Engine

We begin by first discussing the core engine of hashkat. Prior to using the software, rates at which events occur within the network must be set. These include the rate of content generation/propagation (i.e., tweeting and retweeting), the rate of agents connecting (user following), and the rate of agent creation (users joining the network). These rates are used to evolve the simulation through time using the kMC algorithm (Voter 2007). Such rates can be obtained from publicly available social network APIs. kMC, popular in molecular simulations, is a generic and highly scalable (Schulze 2008) algorithm used for generating event sequences based on input rates. It should be noted that kMC is only valid if the rates are correct, the events associated with the rates are Poisson type events, and the different events are independent of one another. The kMC algorithm is as follows:

```
begin
  comment: array of rates for $n$ events
  $rates := [r_1, r_2, ..., r_n]$
  while $simulation\_time < maximum\_time$ :
      comment: Get random number $u_1 \in (0, 1]$
      $u_1 := rand()$
      comment: Create binary tree for efficient event selection
      $binary\_tree = create\_binary\_tree(rates)$
      $R := binary\_tree.sum()$
      comment: Perform a weighted random selection using our random number $u_1$
      $event := binary\_tree.event\_select(u_1)$
      $carry\_out\_event(event)$
      comment: Get random number $u_2 \in (0, 1]$, update simulation time and rates if needed
      $u_2 := rand()$
      $simulation\_time := simulation\_time - ln(u_2)/R$
      $update\_rates(rates)$
  end
end
```

At each simulation time step, a list of *all* possible events is generated. From these events, a random choice is made among all possible outcomes. The probability of choosing a particular event is weighted based on how frequently (i.e., based on the defined rate) that event is expected to occur. Given a set of possible events X, we define the cumulative rate function as

$$R = \sum_{x \in X} r(x). \qquad (1)$$

An event $x \in X$ occurring with $r(x)$ frequency has selection probability of $r(x) / R$. The time step used is inversely proportional to the sum of the event rates in the system. In the standard kMC algorithm, two random variables, $u_1, u_2 \in (0, 1]$ are used for event selection and to progress through time, respectively. After event execution, the kMC method advances the simulation time by

$$dt = -\ln(u_2)/R. \quad (2)$$

As the number of events in the system gets arbitrarily large, over a time frame t, the expected value of an event x will be $r(x)t$. To further simplify constructing the total rate within the simulations, we have applied the notion of homogeneous agent behavior and collective events. Homogenous agent behavior implies that agents within the same category will have equivalent rates associated with their behavior. That is, two users with an identical categorization will rebroadcast a message with the same probability and connect with other users with the same rate. Although the agents within the same category act identically, they may follow different agents diversifying information diffusion. Hashkat can be used with an arbitrary number of agent profiles, but for optimal performance, 10–100 agent profiles are recommended.

2.2 Program structure

Due to the use of the homogenous agents, a collection of identical events can be treated as a single event when constructing the cumulative rate function (collective events). This is shown in Fig. 1. Every collective event summarizes the total rate of children events (possibly collective themselves), forming a tree. Furthermore, collective events that only require selecting an agent from a set of homogeneous agents can be fulfilled in $O(1)$ (on average) time throughout our simulation. Our implementation first chooses between either the event categories of information diffusion (message broadcasts and rebroadcasts), social graph growth (users joining), or an agent connection forming (user following). For new content broadcasting, and new followings, hashkat makes use of homogeneous agent behavior to reason about large lists of similar users. For message rebroadcasting, hashkat utilizes binary trees to choose a piece of content during information diffusion.

Within the code base, there are four important C++ classes that should be mentioned. These are shown in Fig. 2. These classes were used to separate important parts of the simulation into different files, thereby organizing the code base. As briefly explained in Fig. 2, the Analyzer class (analyzer_main.cpp) is where the core of the simulation takes place. Here, the simulation either begins or continues (if a restart file was written to disk, i.e., serialized), all of the event functions are called, and statistics are output to monitor the simulation. Since some events are more complex than others (i.e., following and retweeting are more complex than tweeting), we have created separate

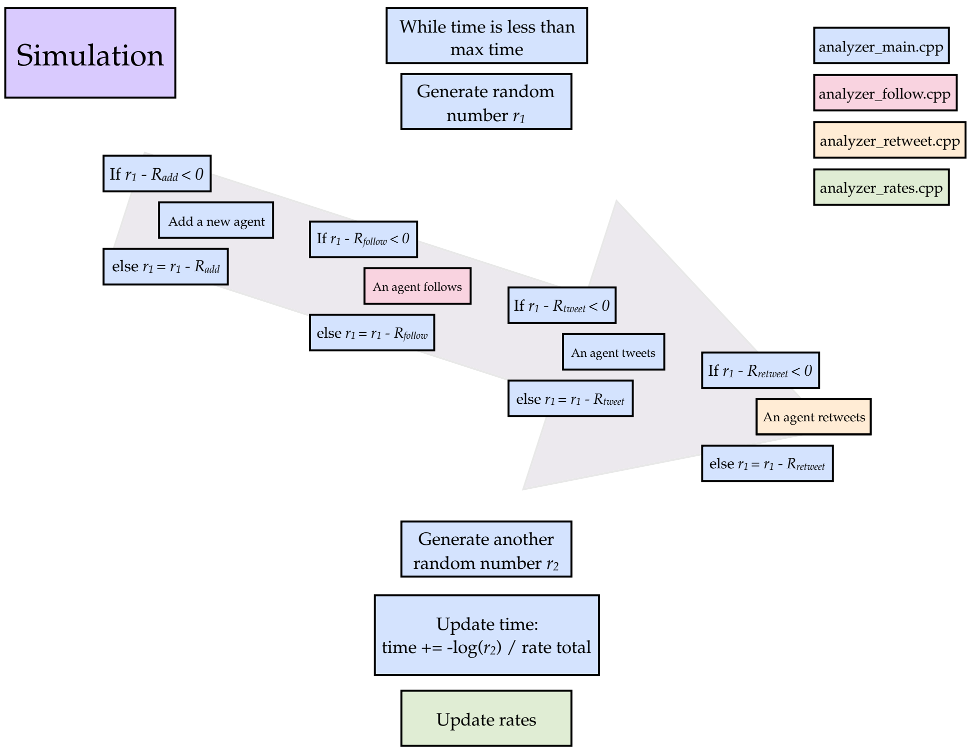
Simulation
While time is less than max time
Generate random number r_1
analyzer_main.cpp
analyzer_follow.cpp
analyzer_retweet.cpp
analyzer_rates.cpp
If r_1 - R_{add} < 0
Add a new agent
else $r_1 = r_1 - R_{add}$
If r_1 - R_{follow} < 0
An agent follows
else $r_1 = r_1 - R_{follow}$
If r_1 - R_{tweet} < 0
An agent tweets
else $r_1 = r_1 - R_{tweet}$
If r_1 - $R_{retweet}$ < 0
An agent retweets
else $r_1 = r_1 - R_{retweet}$
Generate another random number r_2
Update time: time += -log(r_2) / rate total
Update rates

Fig. 1 Illustration of kMC event selection with "collective events," as applied in hashkat

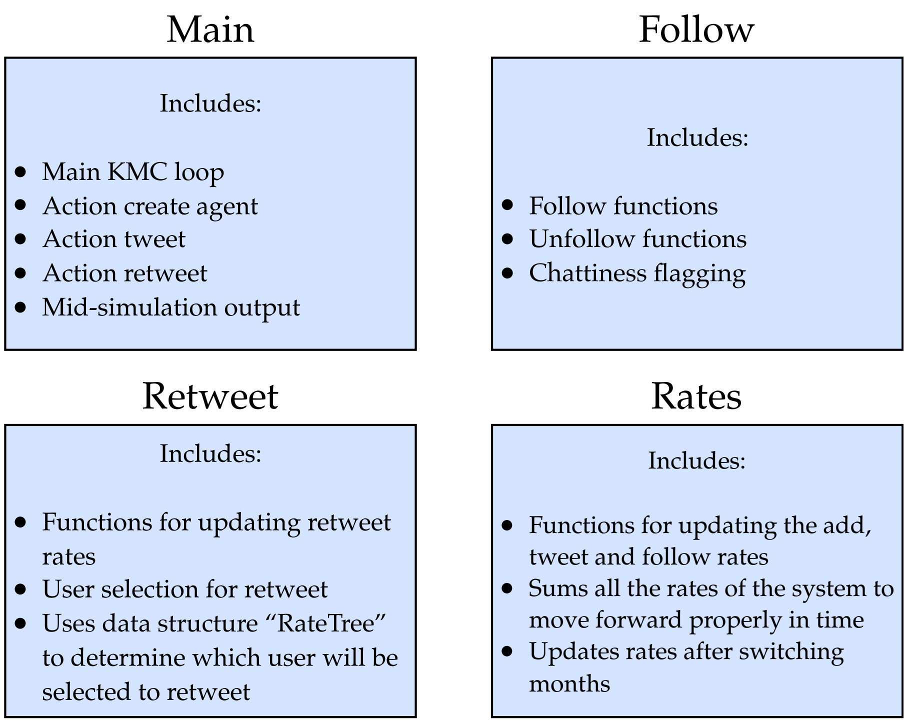
Main
Includes:
Main KMC loop
Action create agent
Action tweet
Action retweet
Mid-simulation output
Follow
Includes:
Follow functions
Unfollow functions
Chattiness flagging
Retweet
Includes:
Functions for updating retweet rates
User selection for retweet
Uses data structure "RateTree" to determine which user will be selected to retweet
Rates
Includes:
Functions for updating the add, tweet and follow rates
Sums all the rates of the system to move forward properly in time
Updates rates after switching months

Fig. 2 Schematic view of the four most important classes used while a simulation is active

classes for following and retweeting. These classes are AnalyzerFollow (analyzer_follow.cpp) and AnalyzerRetweet (analyzer_retweet.cpp), respectively. All of the connection functions are held within AnalyzerFollow. This includes functions to handle creating and destroying connections between agents. Also included here is the flagging of agents who may appear to send messages too often. This information could be used at a later time in the simulation for unfollowing (destroying a connection between agents). The AnalyzerRetweet class contains all functions and data structures for handling information propagation in the network. These functions include updating the cumulative retweet rate (which contributes to the cumulative rate function) and agent selection for a retweet event. The data structure used to organize agents is also here; this structure allows for an efficient agent selection when a retweet event occurs. Lastly, the AnalyzerRates (analyzer_rates.cpp) class handles building the cumulative rate used in the kMC algorithm within the analyzer class.

2.3 Simulation workflow

To use hashkat (Fig. 3), a user starts by modifying an input file (INFILE.yaml). Here, the user can modify, add, or remove variables for their simulation. Once the user is satisfied with their simulation configuration, they execute a Bash script (run.sh) in the top-level directory of hashkat. The Bash executable first calls a Python script (hashkat_pre.py) which generates other files needed for a simulation. As an example, the Python script generates the numerical grid from a user-defined probability density function used in the retweeting algorithm. Once the python script has finished, the C++ executable is called. This executable first reads in the files generated by hashkat_pre.py where the configuration information is stored within a class called AnalysisState (analyzer.cpp). If the user has a restart network file (from a previous run) and has selected the restart option, then the existing network will be loaded into memory and will continue running. If not, initial agents are created, memory is declared, the cumulative rate function is calculated, time-dependent rates are calculated for the simulation time duration, and the main kMC loop is entered. Once the simulation has concluded (either a maximum wall clock time or simulation time has been reached), basic analysis is done on the network (io.cpp) and the program exits. The analysis includes summary statistics of the network, degree distributions, and visualization files (all of which are located in the output directory).

2.4 Agents

All agents within hashkat are classified by several attributes which describe their region, political ideology, language, humor preferences, and musical tastes. These attributes define both the semantic nature of content they create, as well as their reactions to other agent's content. These attributes are shown in Fig. 4. An agent's preferences for content discovery (i.e., finding like minded agents within the same region) are also determined by these attributes. Each of the attributes are discrete and have user-

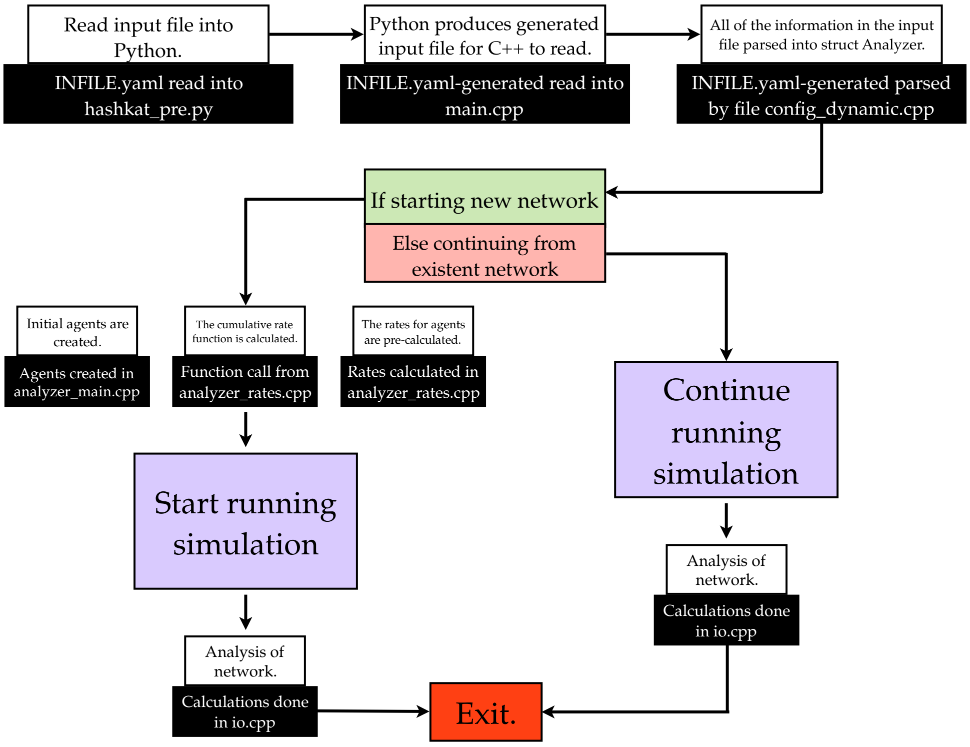
Read input file into Python.
INFILE.yaml read into hashkat_pre.py
Python produces generated input file for C++ to read.
INFILE.yaml-generated read into main.cpp
All of the information in the input file parsed into struct Analyzer.
INFILE.yaml-generated parsed by file config_dynamic.cpp
If starting new network
Else continuing from existent network
Initial agents are created.
Agents created in analyzer_main.cpp
The cumulative rate function is calculated.
Function call from analyzer_rates.cpp
The rates for agents are pre-calculated.
Rates calculated in analyzer_rates.cpp
Start running simulation
Continue running simulation
Analysis of network.
Calculations done in io.cpp
Analysis of network.
Calculations done in io.cpp
Exit.

Fig. 3 Simulation workflow of hashkat. The *white boxes* are a high-level description of the process and *black boxes* indicate which parts of the code are called

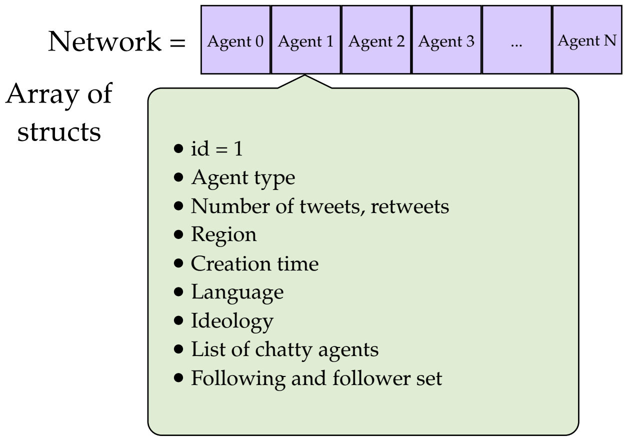

Fig. 4 Schematic view of the network built in hashkat outlining the attributes of agents within hashkat. Each agent has attributes which define both their content preferences and their generation profiles

defined weights associated with them. This allows for simulations with agents from multiple countries, speaking different languages, and even different personal preferences. These options allow for the possibility of creating networks for a wide variety of sociologically relevant cases. For example, a multilingual country with a wide variety of political beliefs can be described as a collection of multilingual agents with the same region attribute and a spread of political views. Conversely, agents within the same region may use different (incompatible) languages and exhibit strongly opposite political ideology. Such a simulation could describe a country where strife and civil discourse are possible. The structure of the graph generated under these two examples will be quite different. It should be noted that although we have used terms like ideology, humor, and music, these attributes can be mapped to arbitrary concepts or types of information. The important distinction between these labels is their scope, discoverability, and transmission factor. An example of transmission factor is shown in Fig. 5. In hashkat, messages transcend artificial barriers if certain conditions are met when comparing two agents. This is given in Table 1.

As an example of what kind of agent types can be created, we have constructed and visualized a network with four agent types (see Sect. 2.9, Fig. 8). We have labeled these agent types standard, celebrity, organization, and bot. Their labels are based on their external attributes (i.e., attributes outside the network that define their behavior within the network) and are motivated by Twitter. On Twitter, real-life celebrities are more highly connected than other users, have a much higher in-degree than out-degree, and make up very little of the network. Real-life organizations tend to have more connections than standard users, have a similar out- and in-degree, and also make up very little of the network. Lastly, standard users and bots are usually less connected, have varied behavior, but make up most of the network. Bots are the phenomena on Twitter where the account is driven by scripts rather than a person. An example is an account that follows only verified users on Twitter (either celebrities or well-know organizations).

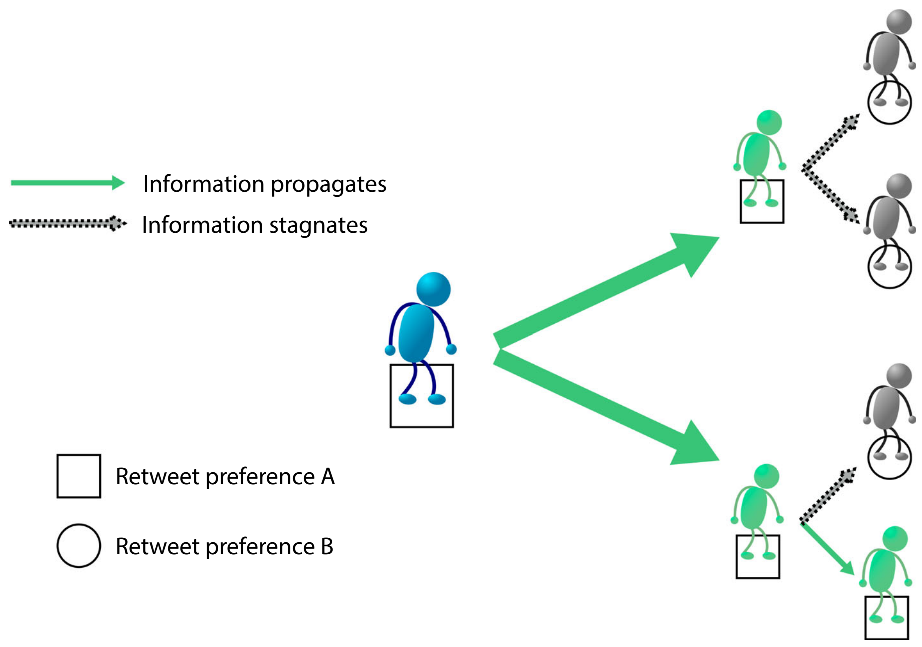

Fig. 5 Toy example outlining transmission factor in a network. In this example, users can either have a retweet preference for *A* (*squares*) or *B* (*circles*). This could be ideology, religion, or some other interest. The agent shown on the *left* sends out a message, and only agents where they have the same preference rebroadcasts the original message. The transmission probability is 1 for preference *A* and 0 for preference *B*. Although binary in this example, hashkat allows continuous transmission probability values when rebroadcasting

Table 1 Table outlining how different tweets transcend artificial boundaries on online social networks

Type of tweet	Rebroadcast condition
Political	Two agents share the same ideology, region, and language
Humorous	Two agents share the same language
Musical	No condition

In hashkat, if these conditions are met, there is an additional transmission probability (that can be set prior to a simulation) of the rebroadcast occurring

This bot follows 212,000 users indicating that celebrities and well-known organizations only make up 0.06% of the Twitter population (Twitter has a reported 313 active million users in the second quarter of 2016). The distinction between a celebrity and standard agent can be understood based on how other agents react to them. Standard agents are judged based only on their observed activity within the network, whereas celebrity agents are intrinsically attractive or persuasive irrespective of their "in-simulation behavior."

2.5 Trending topics

Trending topics on Twitter tend to be synonymous with "hashtags." Hashtags are keywords that begin with the "#" character. Twitter parses the user generated content stream to find hashtags and uses them to classify messages into trending topic lists. These topic lists allow for global information diffusion throughout the network. In hashkat, we model these global topic lists as circular buffers of fixed size. When a new message is added to a full buffer, old messages are discarded. The size of the buffer can be determined experimentally based on the lifetime of content relevancy. It is assumed that the effect of messages older than those in this list can be neglected. The probability of an agent adding a hashtag into their message can be set prior to a simulation. If an agent adds a hashtag to their message, they are added to the circular buffer. This allows for agents within the network to search for specific hashtags (network-wide content discovery) and find other agents with similar interests.

2.6 Adding agents

Throughout a simulation, it is possible for agents to continuously join the network at a rate set by the user in the configuration file (similar to a real online social community). The agent add rate is one of many rates that can be set prior to a simulation. All rates in hashkat can be set to vary throughout the simulation (this requires additional user input). This kind of variation of the input rates is analogous to the changing of rules in cellular automata (Wolfram et al. 1986). Simple changes in the rules can form very different complex systems as time evolves. An example of a realistic agent add rate may be low in the early days of a network, increase as it gains in size, and eventually decrease as the set of possible users is exhausted. When hashkat performs an "add agent" event, it selects an agent type and region based on the configured input and creates a new agent from the particular region. The new agent's region may be used to decide other attributes depending on the user's direction. For example, an agent from a particular region may be more likely to speak one language than another and may be more likely to hold a particular political view.

2.7 Connecting agents

There are many ways for agents to connect within hashkat to mimic the complexity of relationships in real online social networks. In hashkat, the follow rates use the homogeneous agent assumption for users which have the same classification. These correspond to the expected rate that a given agent makes new connections in the network. When hashkat performs a "follow agent" event, it selects a follow model based on the user configuration. Several connection mechanisms have been integrated into hashkat: random, twitter suggest (preferential attachment), agent (selecting a certain agent type), agent type with nested preferential attachment, compatible content (likeness of agents), reciprocal connection, and a combination of all follow models. Hashkat also allows for connections to be removed within the network (unfollowing). Currently, there are two unfollow mechanisms available. The first unfollow mechanism is random, and the second unfollow mechanism takes into account how often an agent tweets with respect to other agents that are being followed (excessive tweeting/chattiness). In hashkat, when an agent is followed, the tweet rate of the agent is compared with the average tweet rate of the agents previously followed. If the tweet rate exceeds the average, the newly followed agent is flagged and can be later unfollowed. Online social networks have the interesting property that subscribers may disagree about which agent is dominating their screen depending on the number and characteristics of agents they follow (Fig. 6). In this regard, discussions in an online setting can be quite different than what transpires in real life. For example, the overly talkative person at a dinner party is easy for everyone to identify; their rate of content generation is large compared to the other attendees. In an online setting, however, different users have different subscription lists; they are all effectively attending many dinner parties simultaneously. In practice, this means that while one observer may be overwhelmed by the content generated from a talkative user, to another observer, there may be a constant flow of messages from the rest of the subscription list making the talkative user seem less intrusive.

2.8 Information propagation

One of the differentiating features of hashkat is the fact that agents within the simulation have the ability to create content (e.g., tweets or wall posts) in a topologically

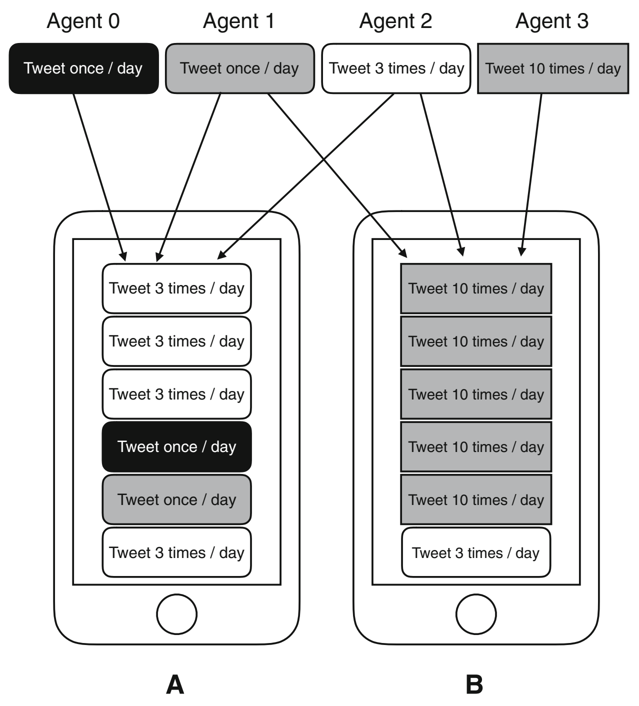

Fig. 6 Example of excessive tweeting and how it must be measured relative to a crowd. On screen **a**, agent 2 dominates the feed in comparison to agents 0 and 1. However, on screen **b** agent 3 dominates the feed. Even though agent 2 dominates the feed on screen **a**, they do not dominate the feed on screen **b**. Chattiness is in the eye of the beholder

dynamic network. Moreover, other agents within the simulation have the ability to rebroadcast this content to their subscriber lists. The propensity for an agent to rebroadcast observed content depends on both the nature of the content and the receiving agent's own preference for such content. Different agents within the simulation can generate content with different meanings. If another agent agrees with the sentiment, they have a higher rebroadcasting probability. This probability can be any value on the interval [0, 1]. To perform content rebroadcasts, we first make a selection between all active broadcasts. We select content m with probability proportional to the collective rebroadcast rate $r(m)$ of the broadcasts' audience. This rate depends on the makeup of the subscriber list. If subscribers agree with the content, the retransmission rate will be high. Otherwise, there will be a low probability of rebroadcasting. We consider the audience to be the subscribers of the broadcasting user u_b. The probability density function of rebroadcasting a given piece of content at a certain time after the content is created is an input to the system $[\Omega(t)]$. The expected number of rebroadcasts can be calculated given $\Omega(t)$, and a transmission probability α,

$$N_{\text{rebroadcasts}} = \int \alpha N_{\text{subscribers}} \Omega(t) \mathrm{d}t. \tag{3}$$

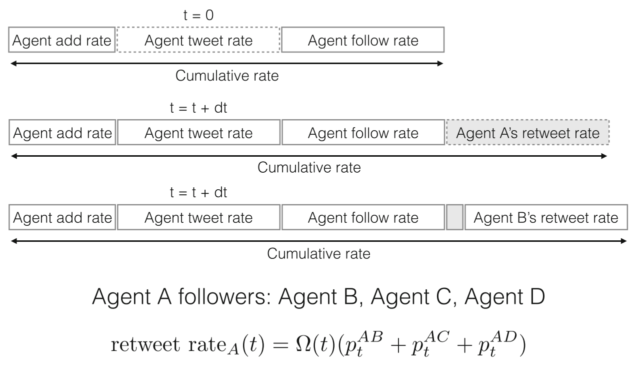

Fig. 7 Example of how retweeting is incorporated into the cumulative rate function. At $t = 0$, there are no current retweets and the agent tweet event is selected. Agent A is then selected and sends out a message. Moving forward in time, the retweet rate of agent A is first added into the cumulative rate function and then selected as the next event to be carried out. Moving forward in time once again, the retweet rate of agent A has decreased [due to $\Omega(t)$], agent B was selected to retweet, and therefore agent B's retweet rate is now added onto the cumulative rate function. Note that in the expression for the retweet rate of agent A, if $p_t = p_t^{AB} = p_t^{AC} = p_t^{AD}$, then the retweet rate becomes $\Omega(t)N^A_{\text{subscribers}}p_t$ as shown in Eq. 3. Here, $p_t \equiv \alpha$, the transmission probability. Different functional forms of $\Omega(t)$ can be provided by the user. The consequences of this selection are explored in Sect. 3.3

where $N_{\text{subscribers}}$ is the number of subscribers of the original broadcaster u_b. Note that the time frame is reset for each rebroadcast and the rebroadcast is then treated in the same manner as an original broadcast. This allows for rebroadcasts themselves to be rebroadcast, analogous to how infections spread through a population. A schematic explanation of this is shown in Fig. 7. The susceptible-infected-recovered (SIR) epidemic model (Pastor-Satorras et al. 2015) is very similar to the mechanism we have applied to propagate messages. In this model, they consider the effective spreading rate of a disease to be $\lambda = \beta/\mu$, where β is the rate of infection and μ is the cure rate. In our model, $\mu = 1$, and the effective spreading rate of a disease is therefore α from Eq. 3. For a subscribing agent u_i, the transmission probability α is a piecewise function which depends on both u_i and the broadcasting agent u_b. To facilitate this, the subscriber set of u_b is partitioned into separate structures, each with their own summary data. These structures are implemented using Google's sparsehash (Silverstein 2010) data structure. Rebroadcasting is considered idempotent. After an agent has rebroadcast a given message once, further rebroadcasts by the same agent will have no effect. To this end, every piece of content is also associated with a set of agents who have previously broadcast the content. The set of active broadcasts is implemented as a binary tree containing rate summary information at every node, necessary to locate the message to select. Selection and insertion are $O(\log n)$, where n is the number of live broadcasts. Periodic cleanup is performed on the set to remove broadcasts unlikely to be rebroadcast. Once a message has been chosen, an agent is chosen with a uniform random selection on the sparsehash data structure to rebroadcast the message. Every message broadcast has a unique identifier, allowing hashkat to track a particular message through the network as it propagates. Hashkat logs the information related to this process to the output directory allowing the user to investigate information propagation in the network even further.

2.9 Visualization

When a simulation has completed in hashkat, there are several different output files to analyze the network constructed. Some of these files are distributions including degree distributions, tweet count distributions, retweet count distributions, and network summary statistics. There are also two main files that can be used to visualize the network itself; one is designed for a network visualization tool called Gephi (Bastian et al. 2009), and the other can be used in Python's networkX (Hagberg et al. 2013) package or R's iGraph package (Csardi and Nepusz 2006). Two visualizations using Gephi of networks generated in hashkat are shown in Fig. 8. Apart from visualizing the full graph, hashkat is also capable of tracking the tweet that has been rebroadcasted the most, and creates a directed graph around that tweet. This is shown in Fig. 9.

2.10 Compatibility and performance

Hashkat is an open source project (GNU General Public License version 3) and has roughly 8600 lines of fully commented code. Hashkat has integrated build tests, unit tests, input tests, and has been built on all three major operating systems: Windows 10 (using the Linux subsystem), Linux, and OS X. It should also be noted that it has been run on systems that vary greatly in size. These include systems with very small amounts of memory (Raspberry Pi) to systems with terabytes of memory (supercomputers). Scalability tests showed that hashkat can produce a random graph of five million nodes in roughly an hour, and 30 million nodes in approximately 40 h. The project homepage (http://hashkat.org) has links to documentation (with an extensive set of examples), source code, and a Web interface for sample input file generation.

2.11 Limitations of the software

Hashkat was designed to study large online social networks, where agents continuously join the network, connect through various mechanisms, and broadcast/

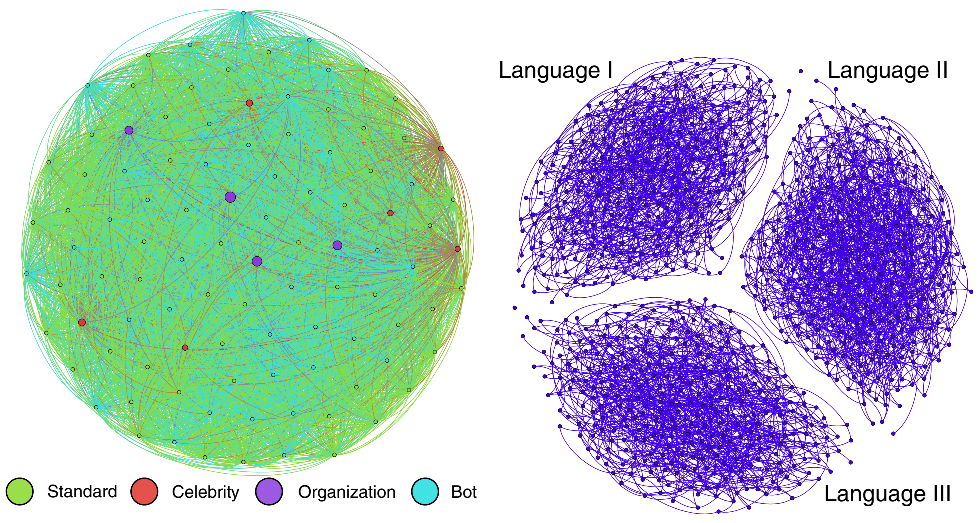

Fig. 8 Hashkat is compatible with common network visualization and analysis tools (e.g., Gephi). On the *left*, we show a random graph with different agent types (described in Sect. 2.4). On the *right*, we show a random graph with one agent type where we have introduced three languages, causing three separate networks to emerge

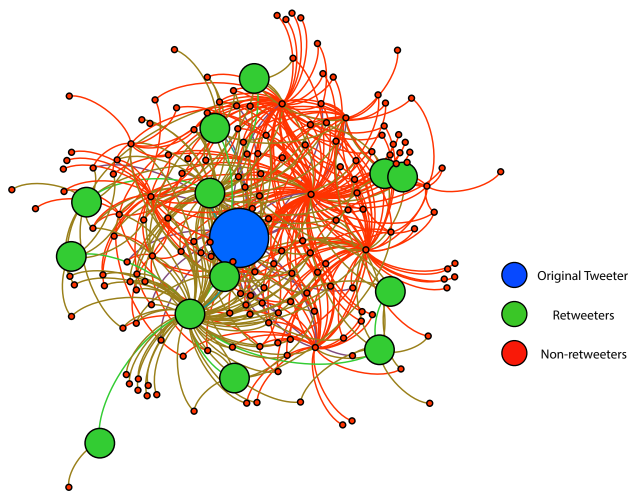

Fig. 9 Visualization of the most rebroadcasted tweet. To create this visualization, hashkat tracks the tweet that has been most rebroadcasted within the simulation. Once the simulation concludes, all possible paths are drawn from the original broadcaster (largest node) down to all the rebroadcasters (medium sized nodes) and viewers (smallest nodes) of the original tweet

rebroadcast messages. If using the preferential attachment model (Twitter suggest) with rebroadcasts, 100s of 1000s of agents is ideal. If creating a dynamic random graph with no rebroadcasting, tens of millions of agents is manageable.

Hashkat has many features, but there are some limitations which can be improved in future development. One limitation of the software is the limited amount of mathematical functions that are accessible to a user for defining the agent add and agent follow rates. Currently, the functions that can be defined in the configuration file are either constant or linear functions. For scenarios where the creation of agents is not linear or constant, hashkat cannot support it without modifying source code.

To produce a network where agents connect via the preferential attachment model, the software must keep track of the in-degree of every agent in the network. Within hashkat, agents are binned together by their in-degree. Prior to a simulation, a user may specify the criteria in which agents are aggregated together. Currently, the bins can be spaced linearly, quadratically, or cubically. The maximum, minimum, and increment value can also be set. If a user would like to bin agents differently, source code must be modified to accommodate the new functionality.

Although hashkat can have an unlimited number of languages, message types (i.e., humorous, ideological, musical), regions, ideologies, and preference classes, a user should be cautious when modifying the number of each feature. The number of each of these features is static constants in the header file “config_static.h” which must be updated when modifying the input file. Currently, the language and message types are also set in this header file, which should directly map to the languages and message types set in the input file.

In hashkat, agents cannot be removed from the networks during a simulation. In real-life online social networks, users have the ability to delete their account and therefore remove themselves from the network. Future development work includes building this feature into hashkat.

The most significant approximation in hashkat is that agent preferences are static in time. Although it is possible to vary certain rates (e.g., the tweet rate) for any class of agents, their language, political ideology, musical tastes, and sense of humor does not change over the course of a simulation. Irrespective of the information (i.e., messages) an agent is exposed to, they will not change their “opinions.” In a real-world scenario, compelling messages or connections to persuasive individuals can alter a person’s worldview and behavior. Agents in hashkat always behave according to their user-defined initialization personality (although we intend to remove this limitation in future work).

Another limitation of the model is that agents are only able to communicate with one another publicly. Almost all major online social networks offer some form of private communication, be it in the form of member-only groups or direct messages. In its current implementation, all messages generated by an agent in hashkat are visible to its entire subscriber list.

A final limitation of the code is that currently it only can take advantage of multi-core architectures in the trivial, seed-parallelism sense. Like all Monte Carlo-based simulations, it is important to generate several independent trajectories (with different random seeds) in order to negate the impact of initial conditions and provide a realistic sampling of possible outcomes. In our work, we have used between 10 and 100 parallel seeds (depending on the size of the network). Making effective use of more than 100 CPU cores is not currently possible within the framework.

3 Test cases

3.1 Analytical graph models

We now use hashkat to firstly generate networks with topologies matching existing theoretical graph models. These models include the random graph and the preferential attachment graph. These models are important because both have been solved analytically. As a result, their corresponding degree distributions can be expressed in closed form. This makes them an ideal test case for validating hashkat. For a random graph (Erdös and Rényi 1959), the degree distribution is known to be the binomial distribution

$$P(k) = \binom{n-1}{k} p^{\mathrm{k}} (1-p)^{n-1-k}, \tag{4}$$

where $P(k)$ is the probability of finding a node with degree k. As n (the total number of nodes) gets large, $P(k)$ can be approximated by the Poisson distribution

$$P(k) = \frac{\lambda^k \mathrm{e}^{-\lambda}}{k!}, \tag{5}$$

where $\lambda = \langle k \rangle$. When constructing a preferential attachment graph, the probability of creating a connection with a node i is given by

$$P(k_i) = \frac{k_i}{\sum_{j=1}^{N} k_j}, \tag{6}$$

where k_i is the degree of node i. In this model, the more connections a node has, the more probable the node is to obtain another connection. This is analogous to the concept of "the rich get richer." Albert et al. (1999) have shown that the degree distributions for these graphs are

$$P(k) \approx k^{-\gamma}, \tag{7}$$

where $\gamma = 3$ for large graphs. In Figs. 10 and 11, we compare our numerical results for both models with the corresponding analytical degree distribution. This form of validation is a strong indication that hashkat is performing as designed. Note that the degree distributions with which we compare are not input to the simulation.

3.2 Reciprocal connections

Recently, a large-scale study of Twitter user profiles (Gleeson et al. 2014) indicated that despite the fact that the service is a directional graph, many of the relation-

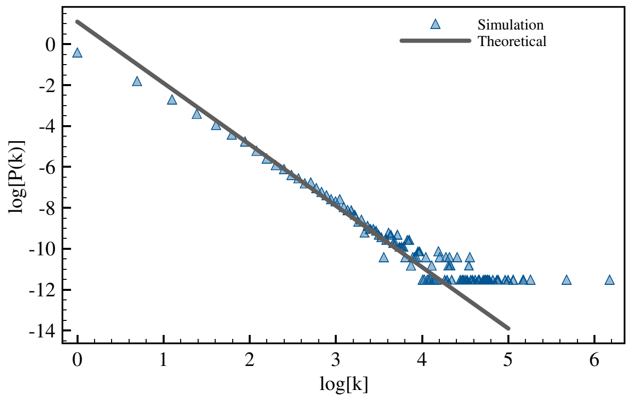

Fig. 10 Comparison of the analytic degree distribution of the preferential attachment model to numerical results generated from hashkat. The points in the *bottom right* of the graph are a result of noise in the distribution

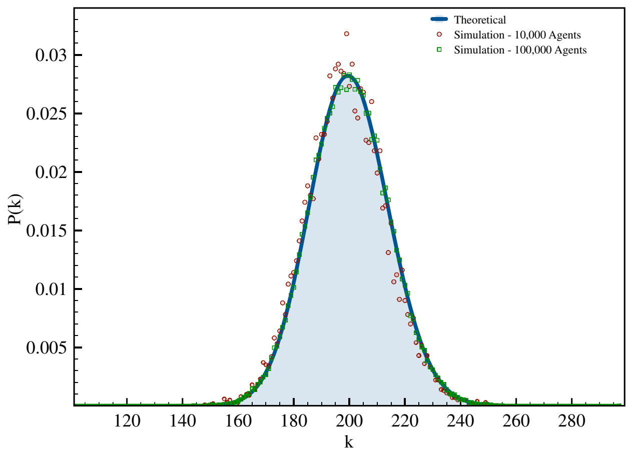

Fig. 11 Comparison of the analytic degree distribution of a random graph to numerical results generated from hashkat

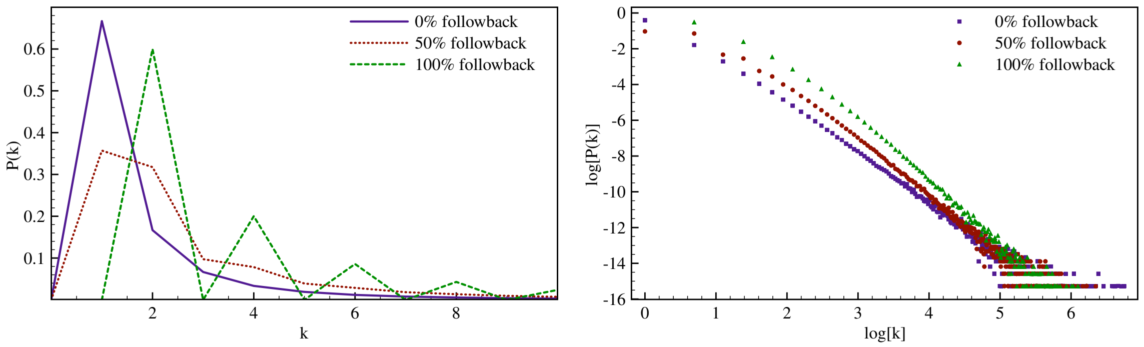

Fig. 12 Cumulative degree distributions (*left*) and log–log plots of the cumulative degree distributions (*right*) for three different simulations where the follow back probabilities were altered from 0 to 1 to show the effects of reciprocal links in a preferential attachment graph

ships are in fact reciprocal. Frequently, when one user subscribes to another, there is a high probability of "followback." For Twitter, as many as 44% of links are reciprocal. For demonstration purposes, we considered the impact that such a behavior has on a preferential attachment network (i.e., where agents have the ability to see the global user list and preferentially follow high-degree users). Hashkat can straightforwardly address this problem. We considered three cases: no followback, 50% followback, and 100% followback. All simulations were run using a constant agent add rate (to a total of 90,000 agents). The results shown in Fig. 12 are based on averages over 100 random initial seeds. As expected, the case of 0% followback results in preferential attachment degree distribution. The 100% results give a distribution which is nonzero only for even numbered degree. This is intuitive, as every subscription will necessarily generate a reciprocal link. The 50% case (close to the real-world Twitter observation) can be understood in terms of these two extremes. Overall the distribution appears to be a preferential attachment one, but with the modification that the fraction of users with odd degree is less, whereas agents with even degree are enhanced. The shift of the curves is due to the follow-back process. For a traditional preferential attachment network, the average cumulative degree is 2. When introducing a 50% followback, the average cumulative degree shifts to 3 and then shifts to 4 for 100% followback. The shifting of the curves is due to the increase in connections from reciprocal links. By considering a controlled system where the followback probability can be arbitrarily varied, it is possible to quantitatively observe the effect such user behavior can have on network structure and time evolution. Note that this approach is substantially more straightforward than developing an extension to the analytic solution for the preferential attachment model, or attempting to gain this insight from experimental data directly, where only one realization of possible parameters can be observed.

3.3 Viral content

Online social networks provide a medium for information to propagate. Information diffusion within these networks is of great interest in the literature (Gleeson et al. 2014; Brach et al. 2014; Myers and Leskovec 2014; Doerr et al. 2011). An interesting phenomenon that emerges in large social networks is the concept of messages "going viral." This is a piece of information that is rebroadcast by many users over a short period of time. From Eq. 3, there are three variables that contribute to viral messages. Firstly, if an agent has many subscribers, the number of other agents who view their information is large. The content has a greater probability of being transmitted due to the audience size. Secondly, if the transmission probability is low, the message lifetime will be short. This is true even for agents with a large number of subscribers. The last determining factor is the probability density function giving the likelihood of a rebroadcast at some time after a message was broadcast. Initially, a message has a greater probability of being rebroadcast as it appears on the top of a user's message board (the feed). At some later time, the message gets pushed further and further down the feed due to newer messages decreasing the rebroadcast probability.

Using hashkat, we experimentally investigated the previously mentioned variables. To do so, we constructed dynamic random graphs with 1000 agents and varied the transmission probability ($0.001 \leq \alpha \leq 0.05$) as well as the probability density distribution $\Omega(t)$. For $\Omega(t)$, we used the functions $\Omega(t) = a\exp(-t)$ and $\Omega(t) = at^{-1}$ where a was chosen such that the functions were normalized. These functions were then integrated over the interval $1 \leq t \leq 600$ min. For each value of α ($\Delta\alpha = 0.001$) ten simulations were run (with different seeds) and were averaged over. The agents in the simulations followed other agents at a constant rate until the total number of follows reached 90,000. Five hundred different simulations were run to produce the contour plots. By increasing the number of follows in the network, the average audience

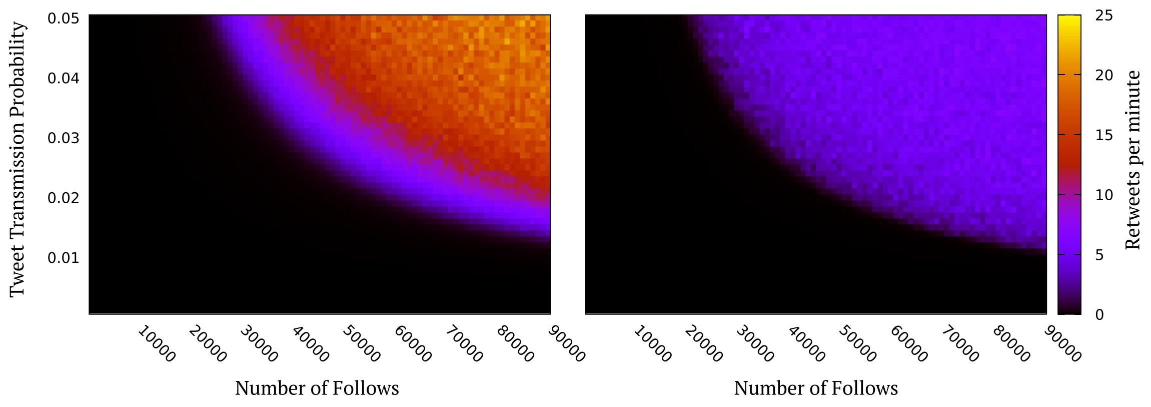

Fig. 13 *Contour plots* showing distinct phases of content rebroadcast behavior. The *left* plot has the probability density distribution (Eq. 3) $\Omega(t) = t^{-1}$, and the *right* has $\Omega(t) = \exp(-t)$

size increases. This allows for more rebroadcasts to occur. With a larger transmission probability, content becomes more likely to be passed on from one agent to another, increasing the effective lifetime of a message within the network. Looking at Fig. 13, we notice a "phase transition" in the contour plots. When $\alpha \leq 0.01$ or the total number of follows ≤20,000, no retweets occur. When $\alpha > 0.01$ or the total number of follows >20,000, we see a transition in the contour plots. Traveling along the transition curve, the retweet rate in the network remains constant while the transmission probability and average degree varies. Any value of α or number of total follows above the transition curve allows for retweets to occur. When $\Omega(t) = at^{-1}$, more retweets occur as you move upward and to the right of the transition curve. When $\Omega(t) = a\exp(-t)$, this is not the case. The exponential function approaches 0 quickly, meaning the probability of a retweet occurring at a time $t_{\text{retweet}} > t_{\text{tweet}}$ is much less than when $\Omega(t) = at^{-1}$. Despite this difference, the structure of the plots is similar for both probability density functions. Again, hashkat can straightforwardly and quantitatively explore information propagation while maintaining complexity.

4 Conclusion

Hashkat is a modern software tool designed for the study and simulation of online social networks. It is an agent-based model with a diverse set of features and capabilities. Hashkat treats network growth and information flow simultaneously, allowing for users to study the interactions between these two phenomena. The kinetic Monte Carlo engine ensures an accurate time evolution of the system and requires only rates as an input. The tool is fully cross-platform (available on Linux, OS X, and Windows 10 via the Linux subsystem) and requires no commercial libraries or tools. Hashkat produces output which is compatible with existing social network analysis packages. The code can run on a wide variety of computing platforms. It can be used to accurately simulate networks ranging from simple random graphs to multi-million agent worlds with a variety of geographical regions, distinct languages, political views, and content preferences. The code is fully open source (GPL v3) and is freely available at the project homepage (http://hashkat.org). In addition to the source code and build instructions, the homepage has extensive documentation, sample visualizations, tutorials, and a Web-based tool to produce sample input files. Hashkat is the most advanced simulation tool for online social networks in existence and is designed to enable fundamental research in this emerging platform of human interactions.

Acknowledgements The authors would like to thank the National Sciences and Engineering Research Council of Canada (NSERC) for funding as well as Compute Canada for computational resources.

References

Albert R, Jeong H, Barabási AL (1999) Internet: diameter of the world-wide web. Nature 401(6749):130
Bastian M, Heymann S, Jacomy M et al (2009) Gephi: an open source software for exploring and manipulating networks. ICWSM 8:361
Brach P, Epasto A, Panconesi A, Sankowski P (2014) Spreading rumours without the network. In: Proceedings of the second ACM conference on online social networks (ACM), pp 107–118
Csardi G, Nepusz T (2006) The igraph software package for complex network research. InterJ Complex Syst 1695(5):1
de C Gatti MA, Appel AP, dos Santos CN, Pinhanez CS, Cavalin PR, Neto SB (2013) A simulation-based approach to analyze the information diffusion in microblogging online social network. In: 2013 winter simulations conference (WSC) (IEEE), pp 1685–1696
Doerr B, Fouz M, Friedrich T (2011) Social networks spread rumors in sublogarithmic time. In: Proceedings of the 43rd annual ACM symposium on theory of computing (ACM), pp 21–30
Erdös P, Rényi A (1959) On random graphs, I. Publ Math 6:290

Gleeson JP, Ward JA, O'sullivan KP, Lee WT (2014) Competition-induced criticality in a model of meme popularity. Phys Rev Lett 112(4):048701

Hagberg A, Schult D, Swart P, Conway D, Séguin-Charbonneau L, Ellison C, Edwards B, Torrents J (2013) Networkx. High productivity software for complex networks. Webová strá nka. https://networkx.github.io. Accessed 2 Feb 2017

Java A, Song X, Finin T, Tseng B (2007) Why we twitter: understanding microblogging usage and communities. In: Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 workshop on Web mining and social network analysis (ACM), pp 56–65

Klügl F, Puppe F (1998) The multi-agent simulation environment sesam. In: Proceedings des workshops simulation in knowledge-based systems, vol tr-ri-98-194 of Reihe Informatik, Paderborn (Citeseer)

Li X, Mao W, Zeng D, Wang FY (2008) Agent-based social simulation and modeling in social computing. In: International conference on intelligence and security informatics. Springer, Berlin, pp 401–412

Mislove A, Marcon M, Gummadi KP, Druschel P, Bhattacharjee B (2007) Measurement and analysis of online social networks. In: Proceedings of the 7th ACM SIGCOMM conference on internet measurement (ACM), pp 29–42

Myers SA, Leskovec J (2014) The bursty dynamics of the twitter information network. In: Proceedings of the 23rd international conference on world wide web (ACM), pp 913–924

Pastor-Satorras R, Castellano C, Van Mieghem P, Vespignani A (2015) Epidemic processes in complex networks. Rev Mod Phys 87(3):925

Schulze TP (2008) Efficient kinetic Monte Carlo simulation. J Comput Phys 227(4):2455

Silverstein C (2010) Google sparsehash package. https://github.com/sparsehash/sparsehash. Accessed 2 Feb 2017

Stadtfeld C (2013) NetSim: a social networks simulation tool in R. http://www.christoph-stadtfeld.com/netsim/. Accessed 2 Feb 2017

Ugander J, Karrer B, Backstrom L, Marlow C (2011) The anatomy of the facebook social graph. arXiv preprint arXiv:1111.4503

Voter AF (2007) Introduction to the kinetic Monte Carlo method. In: Sickafus KE, Kotomin EA, Uberuaga BP (eds) Radiation effects in solids. Springer, Berlin, pp 1–23

Wolfram S et al (1986) Theory and applications of cellular automata, vol 1. World Scientific, Singapore

Zeng R, Sheng QZ, Yao L (2015) A simulation method for social networks. Soc Netw Anal Min 5(1):1. doi:10.1007/s13278-015-0246-4