

CyNetDiff: A Python Library for Accelerated Implementation of Network Diffusion Models

Eliot W. Robson
University of Illinois
Urbana-Champaign
Urbana, Illinois
erobson2@illinois.edu

Dhemath Reddy
University of Illinois
Urbana-Champaign
Urbana, Illinois
dhemath2@illinois.edu

Abhishek K. Umrawal
University of Illinois
Urbana-Champaign
Urbana, Illinois
aumrawal@illinois.edu

ABSTRACT

In recent years, there has been increasing interest in network diffusion models and related problems. The most popular of these are the independent cascade and linear threshold models. Much of the recent experimental work done on these models requires a large number of simulations conducted on large graphs, a computationally expensive task suited for low-level languages. However, many researchers prefer the use of higher-level languages (such as Python) for their flexibility and shorter development times. Moreover, in many research tasks, these simulations are the most computationally intensive task, so it would be desirable to have a library for these with an interface to a high-level language with the performance of a low-level language. To fill this niche, we introduce CyNetDiff, a Python library with components written in Cython to provide improved performance for these computationally intensive diffusion tasks.

Artifact Availability: CyNetDiff—the Python library introduced in this paper is available at <https://pypi.org/project/cynetdiff/>. The source code, data, and/or other artifacts for this paper are available at <https://github.com/eliotwrobson/CyNetDiff/blob/main/examples>. Refer to the Jupyter notebook titled `visualizations.ipynb` for a demonstration of the performance of CyNetDiff.

1 INTRODUCTION

Motivation. Network diffusion is central to studying information propagation [20, 21] and epidemic spreading [3] over social networks. There are several discrete-time stochastic models of diffusion over social networks. In this work, we focus on the *independent cascade* (IC) [6, 7] and *linear threshold* (LT) [9, 18] models of diffusion (discussed in Section 2). In particular, many research tasks related to these models involve simulating their execution over large networks. This can become computationally expensive as both the size of the graphs and the number of simulations grow. Of particular note is the task of influence maximization (IM), introduced by Domingos and Richardson [5]. This involves selecting users on social networks to sponsor to maximize influence under some network diffusion models. This has been widely studied in different settings, namely the discrete [12, 19, 21], continuous [4, 20], and online [1, 16]. Many algorithms for this task involve computing the influence of a given set under a network diffusion model that requires a large number of simulations.

Related Work. Other Python libraries have been written for this task, most notably NDLIB [17]. This is a library that can simulate various network diffusion models (including the IC and LT models described here), written in pure Python, on top of the NetworkX

graph library [10]. NDLIB suffers from shortcomings inherent to many pure Python libraries, including large memory overhead, and slower execution of iterative algorithms than a compiled language. In addition, NDLIB simulates these models by looping through every node in each time step of the model, meaning that this computation is inefficient when only a few nodes are active.

Contribution. In this work, we introduce the CyNetDiff library for simulating these diffusion tasks. We describe the technologies, data structures, and algorithms used in the implementation. We demonstrate this greater performance with a detailed set of benchmarks. We apply this library to influence maximization by implementing the CELF algorithm [13] and reporting the performance.

Organization. The rest of this paper is organized as follows. In Section 2, we discuss the background for the use cases served by CyNetDiff. In Section 3, we discuss the implementation details of the package and how it is optimized for the previously discussed use cases. In Section 4, we detail the demonstration scenarios.

2 PRELIMINARIES

In this section, we give formal statements for the network diffusion models and the influence maximization problem.

Diffusion Models. Diffusion models describe how the cascade takes place in a social network. In *linear threshold* (LT) model, given a (possibly directed) graph $G = (V, E)$, the process starts at time 0 with an initial set of active nodes S , called the *seed set*. When a node $v \in S$ first becomes active at time t , it will be given a single chance to activate each currently inactive neighbor w . The activation succeeds with probability $p_{v,w}$ (independent of the history thus far). If w has multiple newly activated neighbors, their attempts occur in an arbitrary order. If v succeeds, then w will become active at time $t + 1$; but whether or not v succeeds, it cannot make any further attempts to activate w in subsequent rounds. The process runs until no further activation is possible. In the *linear threshold* (LT) model, given a (possibly directed) graph $G = (V, E)$, a node v is influenced by each neighbor w according to a weight $p_{v,w}$ such that $\sum_{w \in \partial v} p_{v,w} \leq 1$, where ∂v represents the set of (in-)neighbors of v . Each node v chooses a *threshold* θ_v uniformly from the interval $[0, 1]$; this represents the weighted fraction of v 's neighbors that must become active for v to become active. The process starts with a random choice of thresholds for the nodes, and an initial set of active nodes S , called the *seed set*. In step t , all nodes that were active in step $t - 1$ remain active, and we activate any node v for which the total weight of its active neighbors is at least θ_v . The process runs until no more activation is possible.

Influence Maximization (IM). For a given model, let $\sigma(S)$ denote the expected number of nodes activated after running the diffusion process to completion with initial seed set S . Given a diffusion model and a budget k , we wish to choose the set S such that $|S| = k$ and $\sigma(S)$ is maximized. Notably, for both the IC and LT models, Kempe et al. [12] showed that the influence function $\sigma(\cdot)$ is sub-modular, and thus its maximum value can be approximated with the greedy algorithm [15]. The greedy algorithm for this problem is computationally expensive, as it requires a large number of evaluations of the influence function $\sigma(\cdot)$. To improve this, the CELF algorithm [13] was introduced as an optimized version of the greedy algorithm, requiring fewer evaluations of $\sigma(\cdot)$. Despite this optimization, the CELF algorithm still requires a substantial number of evaluations of $\sigma(\cdot)$, and optimizing the speed of these evaluations is a point of focus for implementations of this algorithm.

3 CYNETDIFF

In this section, we detail the technologies and implementation techniques used in CYNETDIFF—the library introduced in this paper.

Cython. The desire for greater speed and lower memory usage immediately suggests the use of a compiled language instead of an interpreted one like Python. However, we would like our software package to maintain the greater flexibility provided by Python and its ecosystem. To accomplish both of these goals, we wrote the performance-critical portions of our library in Cython [2] and included some supporting Python utilities.

Cython is a Python language extension that allows for compilation in C and C++ while still providing code callable from Python. This made Cython the ideal technology for providing a high-level Python interface with similar performance to a compiled language.

Data Structures. To take full advantage of the additional performance provided by Cython, we represent graphs within the library using array-based data structures tailored to lower-level languages. These data structures have lower memory overhead and allow for faster execution time by Cython.

We opted to store the underlying graphs in the *compressed sparse row* (CSR) format [11], using the built-in Cython array data structure. At a high level, this format stores the out-neighbors for each node in contiguous memory (unlike an adjacency list, which uses pointers), with an additional indexing array indicating where the neighbors for each node start.

Although the CSR format makes it difficult to modify the graph once it is stored, it has a lower memory footprint than the adjacency list, and allows efficient queries for the outgoing neighbors of a node without the need for pointer lookups. Thus, CSR format is conducive to efficient, repeated traversals, making it ideal for the internal graph representation used by the library.

We also provide utility functions for the creation of model classes directly from NetworkX graphs. These functions convert NetworkX graphs into the CSR format, instantiate the corresponding model, and return the resulting model class to the client code. This has a substantial impact on the usability of the package, as NetworkX is a well-established library for graph analysis tasks, allowing for easy integration of CYNETDIFF into existing research pipelines. This makes CYNETDIFF an effective drop-in replacement for NDLIB.

Algorithms. To facilitate an efficient implementation, we need the following folklore observation about the locality of node activation.

Observation 1. In both the IC and LT models, any node v activated at time t must have at least one in-neighbor u activated at time $t - 1$, unless v is a seed node.

This immediately suggests that the newly activated nodes in each iteration can be determined from the out-neighbors of the activated nodes from the previous iteration. We applied this observation in our implementation of these models, as we use a BFS-based traversal algorithm to determine which nodes are activated in each iteration. As a result, the work performed during the simulation of these models by CYNETDIFF is proportional to the number of edges incident to activated nodes. This can be much smaller than the size of the entire graph when the number of seed nodes is small.

This optimization is very important for improving the runtime in workloads like the CELF algorithm, where many simulations have very few nodes activated. This is especially the case at the beginning of the algorithm’s execution, since the marginal gains for every individual seed must be computed, and only a small portion of the graph is traversed in each iteration as a result.

4 DEMONSTRATION

We describe the demonstration scenarios for CYNETDIFF. As this package is intended to augment existing research applications in Python, the demo will focus on how the package integrates into the Python ecosystem for data analysis, and how it handles the increased scale of datasets. These will be conducted within a Jupyter notebook. For brevity, we focus on the IC model only for our demonstration, although CYNETDIFF can simulate the LT model.

Benchmarks. The first scenario guides participants through interactive benchmarks that introduce the IC model, and compare the performance of CYNETDIFF with other implementations for these tasks. This is done through the introduction of the `simple_benchmark` function, which can be used to run comparative benchmarks for network diffusion on an arbitrary input graph. This function outputs performance information for different diffusion implementations over a configurable number of trials. This will allow participants to easily benchmark additional graphs of their choosing and observe the performance of CYNETDIFF. Output from running this is shown in Figure 1a. The benchmarks in this section cover both synthetic data (generated using graph generation functions provided by NetworkX) and real-world data obtained from SNAP [14]. Participants are encouraged to experiment with different benchmark parameters and observe the effect this has on the performance of the different implementations. An example of a benchmark run on a real-world dataset is shown in Figure 1b.

We next provide some sample benchmarks in Table 1. These benchmarks were conducted on a Dell Precision Tower 5810 with a 3.0 GHz Intel Xeon E5-1660 v3 processor. The three implementations were the CYNETDIFF library, NDLIB library, and a fast pure Python implementation of the diffusion model written for comparison purposes. The benchmarks consisted of running the IC model 1,000 times over the input graph using different edge-weight models (EWM). The first is the *trivalency* (TV) model [8], where

```
In [4]: db.simple_benchmark(graph)

Number of randomly chosen seed nodes: 10
Graph nodes: 1,000
Graph edges: 4,958
Number of trials: 1,000
Starting diffusion with NDlib.

100% ██████████ 1000/1000 [00:14<00:00, 70.51it/s]

func: 'diffuse_ndlib' took: 14.0245 sec
Starting diffusion with pure Python.

100% ██████████ 1000/1000 [00:00<00:00, 1110.33it/s]

func: 'diffuse_python' took: 0.9989 sec
Starting diffusion with CyNetDiff.

100% ██████████ 1000/1000 [00:00<00:00, 19604.96it/s]

func: 'diffuse_CyNetDiff' took: 0.0672 sec
NDlib computed influence: 103.457
Pure Python computed influence: 101.203
CyNetDiff computed influence: 103.357
```

(a) Benchmark output on a synthetic random graph.

```
In [28]: downloader = db.GraphDownloader()

# Print all graphs available to download
print("Available Graphs:", downloader.list_graphs())

# Choose the Epinions graph and set weighted cascade.
print("Preparing to benchmark on Epinions graph.")
big_graph = downloader.get_graph("soc-Epinions1.txt.gz").to_directed()
set_activation_weighted_cascade(big_graph)

# Run a simple benchmark on this graph.
db.simple_benchmark(big_graph, backends_to_run=["cynetdiff"])

Available Graphs: ['facebook_combined.txt.gz', 'twitter_combined.txt.gz', 'wiki-Vote.txt.gz', 'soc-Epinions1.txt.gz']
Preparing to benchmark on Epinions graph.
Number of randomly chosen seed nodes: 10
Graph nodes: 75,879
Graph edges: 811,480
Number of trials: 1,000
Starting diffusion with CyNetDiff.

100% ██████████ 1000/1000 [00:00<00:00, 17543.52it/s]

func: 'diffuse_CyNetDiff' took: 1.4325 sec
CyNetDiff computed influence: 111.288
```

(b) Benchmark output on a real-world graph.

Figure 1: Outputs from `simple_benchmark` in the demonstration. The it/s in the output refers to the number of independent model simulations executed per second.

each edge-weight was drawn uniformly at random from a small set of constants $\{0.1, 0.01, 0.001\}$. The second is the *uniformly random* (UR) model, where the edge-weight was drawn uniformly at random in the interval $[0, 1]$. The third is the *weighted cascade* (WC) model [12], where for each node $v \in V$, the weight of each edge entering v was set to $1/\text{in-degree}(v)$. In all of these weighting schemes, undirected edges in the graph were treated as two directed edges. We used two synthetic networks and a real-world network for our benchmarks. An Erdős-Rényi graph with parameters $n = 2,000$ nodes, $p = 0.002$, leading to 4,018 edges. A Watts-Strogatz Small-World graph with parameters $n = 10,000$ nodes, $k = 10$, $p = 0.007$, leading to 50,000 edges. More details about these graph models and corresponding parameters can be found in the NetworkX documentation. The Facebook graph was obtained from the SNAP dataset and has 4,039 nodes and 88,234 edges.

Visualizations. The next part of the demonstration focuses on visualizations that can be created using CyNETDIFF. The graphics are created using other libraries with data provided from simulations run with CyNETDIFF. Each of these visualizations is created in real-time with reasonably large graphs and a large number of simulations. This demonstrates how the speed of CyNETDIFF can be used to enhance different applications, as the creation of these visualizations would not be possible using a slower library.

The first visualization we create is a *heatmap* based on how many times a node was activated across many simulations. The demonstration provides an example of how to run simulations and create a graph with the results, where a node displays redder if it has been activated in more simulations as seen in Figure 2.

As a second visualization, we use MATPLOTLIB to plot the *mean number of activated nodes over time* for different initial seeds. Each of these plots shows the mean number of activated nodes in each iteration across 1,000 independent simulations. As before, participants are given multiple scenarios and encouraged to experiment with different parameters to see how this affects the output.

Influence Maximization. The final scenario in the demonstration focuses on the task of influence maximization. To demonstrate the

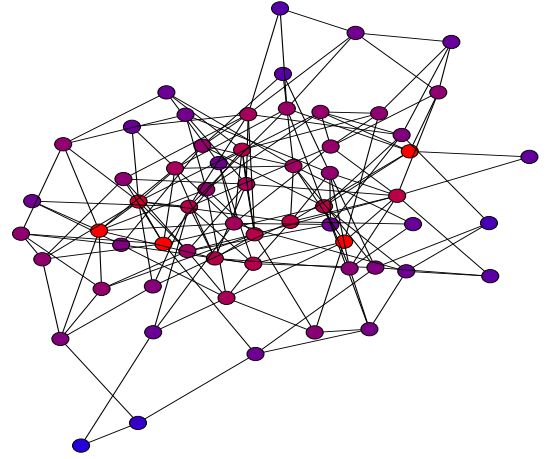


Figure 2: A heatmap created by running CyNETDIFF and coloring nodes based on how often they were activated. The seed nodes appear completely red, as they were always active.

performance of our implementation of network diffusion, we implement the CELF [13] algorithm using all of the network diffusion implementations mentioned here as backends. We then compare the influence of the seed set generated by this algorithm to that of the other methods using the plotting functions introduced earlier.

Participants are encouraged to try different backend implementations for this algorithm, but CyNETDIFF is used in the demonstration to keep a reasonable runtime. The plot generated by this is shown in Figure 3.

For completeness, we also include a sample comparative benchmark for the CELF algorithm, see Table 2. This was performed on a random 7-regular (each node has degree 7) graph with 5,000 nodes and 35,000 edges, generated by NetworkX.

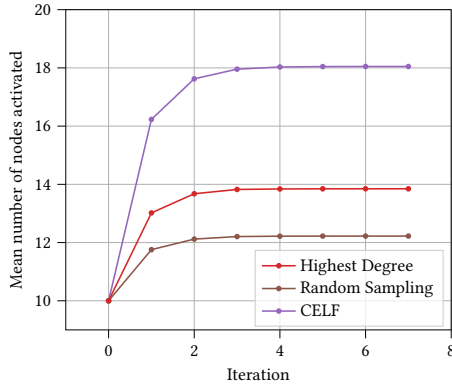


Figure 3: Mean number of nodes activated over time. The data for this plot was dynamically generated during the demonstration on a random regular graph generated by NetworkX.

Graph	EWM	CyNETDIFF	pure Python	NDLIB
Erdős-Rényi	TV	1	11	194
	UR	1	12	203
	WC	1	11	198
Watts-Strogatz	TV	1	9	283
	UR	1	11	327
	WC	1	9	312
Facebook	TV	1	8	81
	UR	1	12	45
	WC	1	8	71

Table 1: Comparison of run-times for independent cascade run with 100 seeds on different graphs. Runtimes are normalized and rounded over each row so that the fastest benchmark in each row is 1.

Graph	EWM	CyNETDIFF	pure Python
Random 7-regular	TV	2	26
	WC	10	153

Table 2: Comparison of run-times for the CELF algorithm run with 10 seeds. Runtimes are in seconds. Results for NDLIB are not reported because they did not finish within 5 minutes.

5 CONCLUSION AND FUTURE WORK

In this work, we have described and demonstrated the effectiveness of CyNETDIFF on a variety network diffusion tasks. The demonstration scenarios and benchmarks show how the speed of the library enables running larger experiments and the creation of interesting visualizations. This will enable faster and more comprehensive experiments going forward by researchers studying network diffusion and influence maximization.

There are several avenues for further work. In one direction, there is potential to continue to improve the performance of CyNETDIFF by adding parallelism. In another, introducing this library enables performing larger-scale experiments, enabling an expansion of the scope of experiments conducted in prior work.

REFERENCES

- [1] Mridul Agarwal, Vaneet Aggarwal, Abhishek K Umrawal, and Christopher J Quinn. 2022. Stochastic Top K-Subset Bandits with Linear Space and Non-Linear Feedback with Applications to Social Influence Maximization. *ACM/IMS Transactions on Data Science (TDS)* 2, 4 (2022), 1–39.
- [2] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. 2011. Cython: The Best of Both Worlds. *Computing in Science & Engineering* 13, 2 (2011), 31–39. <https://doi.org/10.1109/MCSE.2010.118>
- [3] Rebekka Burkholz and John Quackenbush. 2021. Cascade size distributions: Why they matter and how to compute them efficiently. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 6840–6849.
- [4] Wei Chen, Ruihan Wu, and Zheng Yu. 2020. Scalable lattice influence maximization. *IEEE Trans. Comp. Soc. Sys.* 7, 4 (2020), 956–970.
- [5] Pedro Domingos and Matt Richardson. 2001. Mining the network value of customers. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 57–66.
- [6] Jacob Goldenberg, Barak Libai, and Eitan Muller. 2001. Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Marketing Letters* 12, 3 (2001), 211–223.
- [7] Jacob Goldenberg, Barak Libai, and Eitan Muller. 2001. Using complex systems analysis to advance marketing theory development: Modeling heterogeneity effects on new product growth through stochastic cellular automata. *Academy of Marketing Science Review* 9, 3 (2001), 1–18.
- [8] Amit Goyal, Francesco Bonchi, and Laks VS Lakshmanan. 2011. A data-based approach to social influence maximization. *Proceedings of the VLDB Endowment* 5, 1 (2011), 73–84.
- [9] Mark Granovetter. 1978. Threshold models of collective behavior. *Amer. J. Sociology* 83, 6 (1978), 1420–1443.
- [10] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11–15.
- [11] Terence Kelly. 2020. Programming Workbench: Compressed Sparse Row Format for Representing Graphs. *login Usenix Mag.* 45, 4 (2020). <https://www.usenix.org/publications/login/winter2020/kelly>
- [12] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 137–146.
- [13] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. 2007. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 420–429.
- [14] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>.
- [15] George L. Nemhauser, Laurence A. Wolsey, and Marshall L. Fisher. 1978. An analysis of approximations for maximizing submodular set functions—I. *Mathematical Programming* 14, 1 (1978), 265–294.
- [16] Guanyu Nie, Mridul Agarwal, Abhishek Kumar Umrawal, Vaneet Aggarwal, and Christopher John Quinn. 2022. An Explore-then-Commit Algorithm for Submodular Maximization Under Full-bandit Feedback. In *The 38th Conference on Uncertainty in Artificial Intelligence*.
- [17] Giulio Rossetti, Letizia Milli, Salvatore Rinzivillo, Alina Sirbu, Dino Pedreschi, and Fosca Giannotti. 2018. NDLIB: a python library to model and analyze diffusion processes over complex networks. *Int. J. Data Sci. Anal.* 5, 1 (2018), 61–79. <https://doi.org/10.1007/S41060-017-0086-6>
- [18] Thomas C Schelling. 2006. *Micromotives and Macrobehavior*. WW Norton & Company.
- [19] Abhishek K Umrawal and Vaneet Aggarwal. 2023. Leveraging the community structure of a social network for maximizing the spread of influence. *ACM SIGMETRICS Performance Evaluation Review* 50, 4 (2023), 17–19.
- [20] Abhishek K Umrawal, Vaneet Aggarwal, and Christopher J Quinn. 2023. Fractional Budget Allocation for Influence Maximization. In *2023 62nd IEEE Conference on Decision and Control (CDC)*. IEEE, 4327–4332.
- [21] Abhishek K Umrawal, Christopher J Quinn, and Vaneet Aggarwal. 2023. A community-aware framework for social influence maximization. *IEEE Transactions on Emerging Topics in Computational Intelligence* (2023).