Fast GPU-based Influence Maximization within Finite Deadlines via Node-level Parallelism

Koushik Pal[1], Zissis Poulos[2], Edward Kim[1], and Andreas Veneris[2]

[1] Sysomos L.P., Toronto, ON M5J 2V5, Canada
{kpal, ekim} @sysomos.com
[2] University of Toronto, Toronto, ON M5S 3G4, Canada
{zpoulos, veneris} @eecg.toronto.edu

Abstract. Influence maximization in the continuous-time domain is a prevalent topic in social media analytics. It relates to the problem of identifying those individuals in a social network, whose endorsement of an opinion will maximize the number of expected follow-ups within a finite time window. This work presents a novel GPU-accelerated algorithm that enables node-parallel estimation of influence spread in the continuous-time domain. Given a finite time window, the method involves decomposing a social graph into multiple local regions within which influence spread can be estimated in parallel to allow for fast and low-cost computations. Experiments show that the proposed method achieves up to x85 speed-up vs. the state-of-the-art on real-world social graphs with up to 100K nodes and 2.5M edges. In addition, our optimization solutions are within 98.9% of the influence spread achieved by current state-of-the-art. The memory consumption of our method is also substantially lower. Indicatively, our method can achieve, on a single GPU, similar running time performance as the state-of-the-art, when the latter distributes execution across hundreds of CPU cores.

Keywords: Social Media Analytics, Influence Maximization, GPUs

1 Introduction

Influence maximization is one of the dominant topics in viral marketing. It pertains to the problem of identifying a subset of the population that, within a certain time window (deadline), can trigger the maximum number of expected follow-ups in a given network. Understanding the temporal dynamics of influence diffusion is of paramount importance to marketing departments, as it enables them to plan their campaigns operating within strict time-sensitive constraints.

The problem of influence maximization has been extensively studied in the discrete-time domain with infinite deadlines [15, 9, 1, 10, 7]. However, optimizing influence spread over infinitely long time horizons does not always reflect realistic scenarios. For example, a marketer often wishes for an opinion to become viral in a matter of minutes or days, not decades.

P. Perner (Ed.): ICDM 2017, LNAI 10357, pp. 151–165, 2017.
DOI: 10.1007/978-3-319-62701-4_12

As such, maximizing influence spread within finite (and often short) time windows is a variant of the problem which is closer to real world needs. Enforcing such time-sensitive constraints requires influence diffusion models that accurately capture the temporal dynamics of the process to predict how future events unfold in time. A sequence of recent studies on real world data highlights the superiority of continuous-time models over discrete-time ones in expressing the temporal properties of influence diffusion [4, 5, 12, 16].

Motivated by these findings, recent work [6, 3] introduced continuous-time generative models to address influence maximization within finite time windows. The authors have modelled node-to-node influence propagation by transmission rates obeying densities over time, and designed methods for computing exact and approximate influence spread. The method that computes exact spread [6] is not scalable; influence spread from a particular node is computed over the whole network. Yet, in the continuous-time setting, influence decays rapidly towards the network regions that are further away from the source. As such, a big fraction of the computations is wasted analyzing regions of the graph where influence is minuscule or zero, especially when the influence deadline is relatively short.

Based on this observation, we propose a novel approximation method that uses deadline constraints to identify, for each node, a local graph region where the volume of its influence is restricted. The method entails an inexpensive preprocessing step that extracts a decomposition of the social graph into possibly overlapping trees, where the influence of each node is restricted within its own local tree region. This enables us to avoid exhaustive graph inference, thus speeding-up computations with minimal impact on accuracy. Further, it enables GPU-based parallelization, since the influence spread of each node can be computed independently within each local tree region. We build upon this node-level parallelism and harness the parallel processing capacity of commercial-level GPUs to achieve orders of magnitude faster computations than the current state-of-the-art.

Efforts to address the scalability issue have also been taken up by the authors in [3], where an approximation method is developed and shown to be orders of magnitude faster compared to exact inference [6]. This speed-up, however, comes at the cost of enormous memory consumption, which hampers GPU-based acceleration. Indicatively, for social graphs with millions of edges, the method necessitates the instrumentation of massive clusters consisting of 192 CPU cores. Consequently, it is not suitable for parallelization using inexpensive alternatives, such as (multi-) GPU systems. The application of GPUs to this problem has been previously explored [13], but is restricted solely to the discrete-time domain.

2 Preliminaries

Our work is based on the continuous-time generative model for network diffusion that has been introduced in [6]. Given a social network, modeled as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the influence propagation process begins from an initial set $\mathcal{S} \subset \mathcal{V}$ of source nodes, referred to as the *seed set*. The seed set is assumed to be influenced by means of adopting an opinion at time zero.

Influence propagates via directed edges from the seed nodes towards their out-neighbours. The newly influenced nodes influence their out-neighbours in turn, and this process continues. An influenced node is assumed to remain influenced for the entire duration of the diffusion process. Consequently, the node that influences a given node at the earliest time will be its parent in the induced influence propagation graph (also called the *cascade*), effectively imposing a Directed Acyclic Graph (DAG) cascade structure, even if $\mathcal{G}$ contains cycles.

The spread of influence from a node u to an out-neighbour v is assumed to consume *random* time, drawn from a conditional density function $f_{uv}(t_v|t_u)$. This models the time it takes for node u to influence node v at time-stamp t_v given that node u has been previously influenced at time-stamp t_u. These *transmission times* can be distributed differently across the edges, but they are assumed to be mutually independent. We further assume that the transmission function $f_{uv}(t_v|t_u)$ is *shift invariant*: $f_{uv}(t_v|t_u) = f_{uv}(\tau_{uv})$, where $\tau_{uv} := t_v - t_u$, and *nonnegative*: $f_{uv}(\tau_{uv}) = 0$ if $\tau_{uv} < 0$. Examples include exponential and Rayleigh distributions. Consequently, each directed edge $(u, v) \in \mathcal{E}$ is associated with a density function $f_{uv}(\tau_{uv})$, which models the time it takes for u to influence v (independent of the actual timestamps when u and v are influenced). Because of the mutual independence assumption, one obtains a fully factorized joint density of the set of transmission times $p(\{\tau_{uv}\}_{(u,v)\in\mathcal{E}}) = \prod_{(u,v)\in\mathcal{E}} f_{uv}(\tau_{uv})$.

An useful property of the above continuous-time *Independence Cascade* (IC) model is that, for a given sample of edge weights corresponding to their respective transmission times, the time t_u taken to influence a node u is the length of the shortest path in $\mathcal{G}$ from the seed set $\mathcal{S}$ to node u. This *shortest path property* is leveraged for influence spread estimation in [3] and is also utilized in the work presented here, as it reduces the problem of approximating influence spread to a well-studied graphical optimization problem, namely that of finding shortest paths. Because of this property, the infection times $\{t_u\}_{u\in\mathcal{V}}$ can be obtained from the transmission times $\{\tau_{uv}\}_{(u,v)\in\mathcal{E}}$ via the transformation $t_u = g_u(\{\tau_{vw}\}_{(v,w)\in\mathcal{E}}) := \min_{q\in\mathcal{Q}_u} \sum_{(v,w)\in q} \tau_{vw}$, where $\mathcal{Q}_u$ is the collection of all directed paths in $\mathcal{G}$ from each of the source nodes to u, and $g_u(\cdot)$ is the value of the shortest-path minimization. With this setup, one can then compute the probability of u being influenced within the deadline T as

$$\Pr\{t_u \le T\} = \Pr\{g_u(\{\tau_{vw}\}_{(v,w)\in\mathcal{E}}) \le T\}.$$

By standard definition [6], the *influence spread* $\iota(\mathcal{S}, T)$ of the seed nodes $\mathcal{S}$ in the deadline T can then be computed as

$$\begin{aligned}\iota(\mathcal{S},T) &= \mathbb{E}\left[\sum_{u\in\mathcal{V}} \mathbb{I}\{t_u \le T\}\right] = \sum_{u\in\mathcal{V}} \mathbb{E}\left[\mathbb{I}\{t_u \le T\}\right] = \sum_{u\in\mathcal{V}} \Pr\{t_u \le T\} \\ &= \sum_{u\in\mathcal{V}} \Pr\{g_u(\{\tau_{vw}\}_{(v,w)\in\mathcal{E}}) \le T\} = \mathbb{E}\left[\sum_{u\in\mathcal{V}} \mathbb{I}\{g_u(\{\tau_{vw}\}_{(v,w)\in\mathcal{E}}) \le T\}\right],\end{aligned}$$

where $\mathbb{I}\{\cdot\}$ is the indicator function, $\mathbb{E}\{\cdot\}$ is the expectation function, and the expectation is taken over the set of independent variables $\{\tau_{vw}\}_{(v,w)\in\mathcal{E}}$. The sum

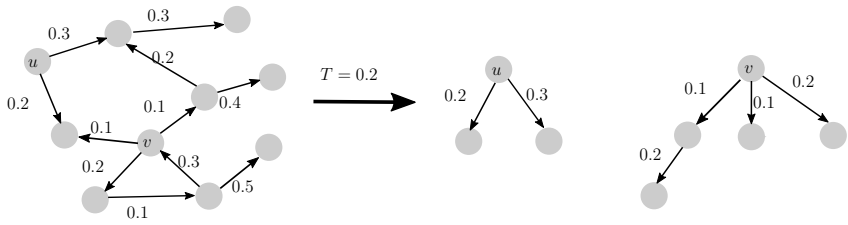


Fig. 1: Extracting local regions in the IC model

$\sum_{u \in \mathcal{V}} \mathbb{I}\{\cdot\}$ in the above formula is essentially the influence spread of the seed nodes for a given sample of transmission times $\{\tau_{vw}\}_{(v,w)\in\mathcal{E}}$.

Finally, *influence maximization* is the problem of finding an optimal set $\mathcal{S}$ of seeds of a fixed size $\mathcal{C}$ such that $\iota(\mathcal{S}, T)$ is maximized, i.e., we seek to solve

$$\mathcal{S}^\star = \underset{\{\mathcal{S} \,:\, |\mathcal{S}| \leq \mathcal{C}\}}{\arg\max} \quad \iota(\mathcal{S}, T). \tag{1}$$

We take $\mathcal{C}$ (determined by budgetary constraints) as an input in this paper. It is noteworthy that the above optimization problem is NP-hard in general.

3 Methodology

A fundamental step in maximizing influence is to compute the influence spread of each node of the graph $\mathcal{G}$. The most basic way of doing that, as suggested by Equation 1, is via Naïve Sampling, where one generates a random sample of $\{\tau_{uv}\}_{(u,v)\in\mathcal{E}}$ from the corresponding edge distributions $\{f_{uv}(\tau_{uv})\}_{(u,v)\in\mathcal{E}}$, runs a *single source shortest path* (SSSP) algorithm from each node, and computes the influence spread of that node for that sample as the number of nodes whose shortest distance from the source node is less than the deadline T. The process repeats for several iterations (say, N_s times). On termination, the average spread of each node across all N_s samples is computed. Due to its exhaustive nature, Naïve Sampling is a costly process, as it runs a single source shortest path algorithm from each node for each sample. Assuming Dijkstra's algorithm at the core of the process, the overall time complexity is $O(N_s|\mathcal{V}|(|\mathcal{E}|+|\mathcal{V}|\log|\mathcal{V}|))$, which is prohibitively expensive. Space complexity is, however, minimal — $O(|\mathcal{V}|+|\mathcal{E}|)$.

Naïve Sampling is massively parallelizable across samples, but with shortcomings. All samples need to be generated at once, which requires $O(N_s(|\mathcal{V}|+|\mathcal{E}|))$ space. Moreover, it needs to run a SSSP algorithm from every node of the graph to every other reachable node, which is redundant for smaller deadlines. Thus, it is reasonable to identify, for each node, a "large enough, yet small" subgraph wherein its influence within the deadline is primarily restricted to, and search for its influence there (instead of the whole graph). We refer to this method as Localized Naïve Sampling (`LNS`) (cf. Algorithm 1 and Algorithm 2). We obtain such a subgraph for each node u by running Dijkstra's algorithm from u with the means of the edge distributions as the corresponding weights for each edge, and keeping those nodes in the subgraph of u whose shortest distance from u in $\mathcal{G}$ is "slightly greater" than T. Figure 1 depicts the process, and shows the subtrees

extracted for nodes u and v in the graph. The precise condition for deciding whether to include a node v in the subgraph of u is mentioned in Algorithm 2 (Line 25). The explanation of the criterion and the choice of the free parameter σ are explained further in the Appendix. The quantity $Var(u, v)$ in Line 20 of Algorithm 2 is the variance of f_{uv} corresponding to the edge $(u, v) \in \mathcal{E}$.

Looking for the influence spread of each node in a smaller subgraph potentially reduces computations. The subgraph we choose is a subtree of the shortest path tree obtained when running Dijkstra's algorithm. Computing shortest paths in a tree for a given sample of weights is cost-effective, since there is exactly one path from the source node to any other node. For a tree of size $|L|$, the time complexity of computing the distance of each node from the root is $O(|L|)$. Once we extract a local subtree for each node, we generate multiple weight samples for that subtree and compute the average influence spread of that node in that subtree over all iterations. This process gives us an approximate spread for each node, as opposed to its true spread under the IC model. However, it does not largely affect the quality of the seeds obtained, as results in Section 4 show.

Assuming the size of each subgraph is bounded by $|L|$, the time complexity of the serial version of `LNS` is $O(|\mathcal{E}| + |\mathcal{V}| \log |\mathcal{V}| + N_s|V||L|)$, which in the worst case is $O(N_s|V|^2)$, as $|L| = O(|V|)$ in the worst case. But `LNS` is node-level parallelizable (cf. Parallel Block in Algorithm 1), which leads, for the parallel version, to a worst case runtime of $O(|\mathcal{E}| + |\mathcal{V}| \log |\mathcal{V}| + N_s|V|)$. This enables us to achieve large reductions in time complexity primarily on large graphs (see Section 4). The space complexity of our method is $O(|V||L|)$, which in the worst case is $O(|V|^2)$, as we have to store all the local subgraphs for all the nodes.

Algorithm 1 Localized Naïve Sampling (`LNS`)

```
1: procedure LocalizedNaiveSampling(G, T, N_s, σ)
2:     for u = 1 : |V| do
3:         spread[u] = 0
4:         Assign weights W to all edges of G equal to the
   means of the corresponding edge distributions
5:         dijkstraTree[u] = DijkstraTree(u, G, W, T, σ)
6:
       Parallel Block:
7:     for u = 1 : |V| do
8:         for n = 1 : N_s do
9:             Generate a sample of {τ_vw}_{(v,w)∈dijkstraTree[u]}
10:            for each node v in dijkstraTree[u] do
11:                distance[v] = distance of v from u in dijkstraTree[u]
12:                if distance[v] < T then
13:                    spread[u] = spread[u] + 1
14:        spread[u] = spread[u] / N_s
15:    return spread[]
```

In comparison, the state-of-the-art framework, `ConTinEst` [3], achieves its superior runtime performance compared to traditional methods by employing a

randomized version of Dijkstra's algorithm [2]. This comes, however, at the cost of space complexity. Specifically, the runtime to compute expected influence across all nodes and all samples is $O(N_s(|\mathcal{V}| + |\mathcal{E}|))$, while space complexity is $O(N_s|\mathcal{E}|)$ — a significant memory bottleneck, particularly for large graphs.

Algorithm 2 Dijkstra Trees

```
1: procedure DijkstraTrees(source, 𝒢, W, T, σ)
2:     distance[source] = 0
3:     variance[source] = 0
4:     create vertex set Q
5:     for u = 1 : |𝒱| do
6:         if u ≠ source then
7:             distance[u] = ∞
8:             variance[u] = ∞
9:             parent[u] = UNDEFINED
10:        Q.add_with_priority(u, distance[u])
11:    while Q is not empty do
12:        u = Q.extract_min()
13:        if distance[u] − σ * sqrt(variance[u]) ≥ T then
14:            break
15:        for each out-neighbor v of u in 𝒢 do
16:            alt_distance = distance[u] + W(u, v)
17:            if alt_distance < distance[v] then
18:                distance[v] = alt_distance
19:                parent[v] = u
20:                variance[v] = variance[u] + Var(u, v)
21:                Q.decrease_priority(v, alt_distance)
22:    dijkstraTree = []
23:    dijkstraTree.add([source, source])
24:    for u = 1 : |𝒱| except source do
25:        if distance[u] − σ * sqrt(variance[u]) < T then
26:            dijkstraTree.add([u, parent[u]])
27:    return dijkstraTree[]
```

To compute the joint influence spread of a set $\mathcal{S}$ of nodes, for each sample, we mark the nodes that are influenced by each element $s \in \mathcal{S}$. Then, we count all such nodes that are influenced by at least one element of $\mathcal{S}$; this count is exactly the influence spread of $\mathcal{S}$ for that sample. We repeat the process N_s times and take an average to obtain the expected influence spread of $\mathcal{S}$. It can then be shown that, over the local regions where our method operates, $\iota(\mathcal{S}, T)$ satisfies a diminishing returns property referred to as *submodularity* : for $S_1, S_2 \subset \mathcal{V}$ with $S_1 \subseteq S_2$ and $u \in \mathcal{V} \setminus S_2$, it holds that $\iota(S_1 \cup \{u\}, T) - \iota(S_1, T) \geq \iota(S_2 \cup \{u\}, T) - \iota(S_2, T)$. This implies that our method can be used as a subroutine in the greedy algorithm that we describe below (cf. Algorithm 3).

Our goal is to find a set $\mathcal{S}$ of nodes of size $\mathcal{C}$ such that their combined influence spread is maximum. Due to its intractability, the problem calls for

an approximation algorithm. For monotonic submodular functions, the greedy algorithm described by Kempe et al. [9] is one such well-known approximation algorithm. The algorithm is iterative, and at the i^{th} iteration, adds to the seed set $\mathcal{S}_{i-1}$ the node $s \in \mathcal{V} \setminus \mathcal{S}_{i-1}$ that maximizes the *marginal gain* $\iota(\mathcal{S}_{i-1} \cup \{s\}, T) - \iota(\mathcal{S}_{i-1}, T)$. We use `LNS` in each iteration of the greedy algorithm to find such nodes. Because we approximate the influence spread $\iota(\mathcal{S}, T)$ by using random samples drawn from edge distributions, we introduce a sampling error. Fortunately, the greedy algorithm is tolerant to such sampling noise (see [3]).

Algorithm 3 Overall Algorithm

procedure InfluenceMaximization($\mathcal{G}$, T, $\mathcal{C}$, N_s, σ)
  for $u = 1 : |\mathcal{V}|$ **do**
    Assign weights W to all edges of $\mathcal{G}$ equal to the means of the corresponding edge distributions
    $dijkstraTree[u]$ = DijkstraTree($u, \mathcal{G}, W, T, \sigma$)
  $\mathcal{S} = \emptyset$
  for $i = 1 : \mathcal{C}$ **do**
    for $u = 1 : |\mathcal{V}|$ **do**
      Call the Parallel Block in `LNS` to compute :
      $marginal\_spread[u] = \iota(\mathcal{S} \cup \{u\}, T) - \iota(\mathcal{S}, T)$
    $s = \arg\max_{u \in \mathcal{V} \setminus \mathcal{S}} marginal\_spread[u]$
    $\mathcal{S} = \mathcal{S} \cup \{s\}$
  return $\mathcal{S}$

As shown in [14], the above greedy approach obtains a seed set which achieves at least a constant fraction $(1 - \frac{1}{e})$ of the optimal spread, provided the influence spread function is a monotonic submodular function. In the proposed method, however, the influence spread function is not sub-modular under the IC model. Thus, the approximation ratio cannot be claimed. However, there are loose bounds that we can claim by showing that $\iota(\mathcal{S}, T)$ is approximately submodular, based on the following definition.

Approximate Submodularity: For given $\epsilon \geq 0$, we say that a function $F : 2^V \to \mathbb{R}$ is *ϵ-approximately submodular* if there exists a submodular function $f : 2^V \to \mathbb{R}$ such that for every $S \subseteq V$:

$$(1 - \epsilon) f(S) \leq F(S) \leq (1 + \epsilon) f(S).$$

Theorem 1 *The influence spread $\iota(\mathcal{S}, T)$ is ϵ-approximately submodular under the continuous IC model.*

Proof. Consider $f : 2^V \to \mathbb{R}$ to be the influence spread function under the continuous IC model when the whole graph is considered for computations. The function f abides to the standard spread definition, which is known to be submodular. In the decomposition we propose, a finite set of graph nodes $\emptyset \subseteq R_S \subsetneq V$ are rendered unreachable by set S (for finite T), when in fact they might be reachable in the IC model. Thus, it follows that $f(S) - \iota(\mathcal{S}, T) \leq |R_S|$,

or $f(S)-|R_S| \leq \iota(\mathcal{S},T)$. It also trivially holds that $\iota(\mathcal{S},T) \leq f(S)$, which implies $\iota(\mathcal{S},T) \leq f(S)+|R_S|$. Hence,

$$f(S)\Big(1-\frac{|R_S|}{f(S)}\Big) \leq \iota(\mathcal{S},T) \leq f(S)\Big(1+\frac{|R_S|}{f(S)}\Big).$$

Set $\epsilon := max\left\{\frac{|R_S|}{f(S)} \;\middle|\; S \subseteq V\right\}$. Then, $0 \leq \epsilon < 1$ (as $|R_S| < f(S)$ for every $S \subseteq V$). Also, we obtain

$$(1-\epsilon)f(S) \leq \iota(\mathcal{S},T) \leq (1+\epsilon)f(S).$$

Thus, $\iota(\mathcal{S},T)$ is ϵ-approximately submodular. □

By Theorem 5 of [8], we obtain the following approximation result.

Corollary 1. *For $\mathcal{C} \geq 2$ and any constant $0 \leq \delta < 1$ with $\epsilon = \frac{\delta}{\mathcal{C}}$, the greedy algorithm obtains a seed set which achieves at least a constant fraction $1-\frac{1}{e}-16\delta$ of the optimal value.*

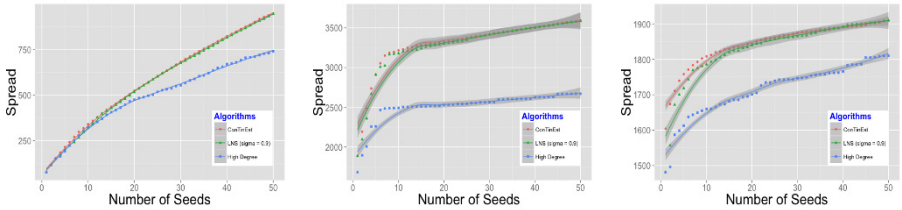
Since $\epsilon = \frac{\delta}{\mathcal{C}}$, we have $\epsilon = O(\delta)$, provided $\mathcal{C}$ is constant. It follows that for lesser values of ϵ (equivalently, for smaller values of $|R_S|$), we achieve better approximations of the influence spread, which are closer to the optimal value. Of course, the values of $|R_S|$ and ϵ are entirely dependent on the depth of the local subtrees we choose. Deeper subtrees offer potentially better approximations, but induce extra computational cost. In Section 4, we show that when we select deep enough trees, without still covering the whole graph, we obtain quality results on par with the state-of-the-art methods while maintaining runtime benefits.

4 Experiments

We base our evaluation on real world networks found in the Stanford Network Analysis Project (SNAP) [11]. Table 1 shows some of the network characteristics. In each network, we associate each directed edge with a transmission function obeying an exponential density, whose scale parameter is drawn uniformly at random from the open-closed interval $(0, 5]$.

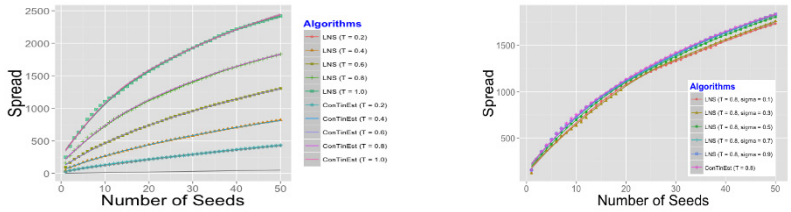
Table 1: Network Statistics

Network	**# nodes**	**# edges**	**density**
ego-Facebook	4,039	88,234	21.84
gnutella08	6,301	20,777	3.29
wiki-vote	7,115	103,689	14.57
gnutella04	10,876	39,994	3.68
soc-Epinions1	75,879	508,837	6.71
ego-twitter	81,306	2,468,149	30.35
soc-Slashdot0922	82,168	948,464	11.54



(a) gnutella08 (b) facebook (c) wiki-vote

Fig. 2: Seed set spread vs. seed set size with $T = 0.2$



(a) gnutella04 (varying T) (b) gnutella04 (varying σ)

Fig. 3: Effect of T and σ on spread respectively

4.1 Quality of Seed Sets

To compare the quality of seed sets that are produced by our methodology, we perform an immediate comparison between our method and `ConTinEst` [3], since the latter has been shown to statistically outperform other approximation methods on real world data. To do so, we need a procedure that can receive these seed sets as input and obtain near ground-truth estimates of their influence spread. We perform the comparison by running Naïve Sampling for 10,000 iterations on each seed set that is obtained. For practical reasons, we restrict this comparison on the four smallest graphs in our dataset, since Naïve Sampling requires at least 8,000 hours to produce results for the larger graphs.

For this set of results, the algorithms are configured as follows: abiding to author guidelines in [3], `ConTinEst` is set to perform 10,000 sampling rounds for transmission times. For each sampling round, it is also configured to run 5 iterations for the randomization required by the neighborhood size estimation subroutine. `LNS` is also set to 10,000 iterations, while the σ parameter is tuned to 0.9 to produce large enough local regions.

Results are shown in Figure 2. We also report the spread achieved by greedily selecting the node with highest out-degree each time (`High Degree`). Finally, the deadline is fixed to $T = 0.2$. One observes that `LNS` is on par with `ConTinEst` in terms of seed quality. In fact, the relative error never surpasses 11% across all four graphs. On the average, the relative error of our method is 1.2%. In contrast, a simplistic method such as `High Degree`, produces solutions that are, on the average, up to 21.2% off in terms of influence spread compared to `ConTinEst`.

We also evaluate the effect that the deadline constraint has on our method's accuracy. Figure 3(a) compares the spread obtained by our method and that obtained by `ConTinEst` under various deadlines (between $T = 0.2$ and $T = 1.0$) for $\sigma = 0.9$ on a graph with 10,876 nodes (**gnutella04**). One observes that the proposed method maintains the quality of seed sets even when the deadline increases. Indicatively, the average relative error ranges between 0.2% for $T = 0.2$ and 0.7% for $T = 1.0$. Finally, we study the effect of σ on seed quality. For a fixed $T = 0.8$, and for σ ranging between 0.1 and 0.9, it can be seen in Figure 3(b) that accuracy drops with lower values of σ, as expected. However, for $\sigma \geq 0.4$, our average relative error falls between 0.8% and 3.5%.

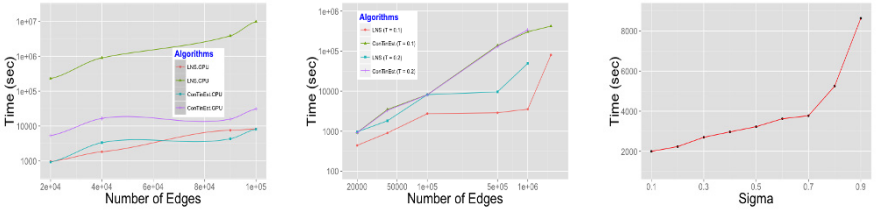
4.2 Runtime Evaluation

Our runtime evaluation entails two parts. First, we empirically expose that GPU-based acceleration for `ConTinEst` is severely hampered by memory bottlenecks. We do so by porting a sample-parallel version of the implementation in [3] into a GPU utility. Experiments show that, for graphs of substantial size, the parallel version performs poorly. Second, we report running times under different deadlines, and demonstrate a comparison between the `ConTinEst` engine and a node-parallel version of `LNS` implemented on GPUs. Any implementation mentioned henceforth is carried out on a single 2.6GHz processor, and a GPU card with 4GB RAM on an NVidia K520 Grid platform.

First, we report runtime comparisons between four implementations: two CPU-based implementations of `ConTinEst` and `LNS`, a sample-parallel GPU implementation of `ConTinEst`, and a node-parallel GPU implementation of `LNS`. Note that `ConTinEst`, by construction, does not allow node-level parallelism. All implementations are set to retrieve seed sets of size 50.

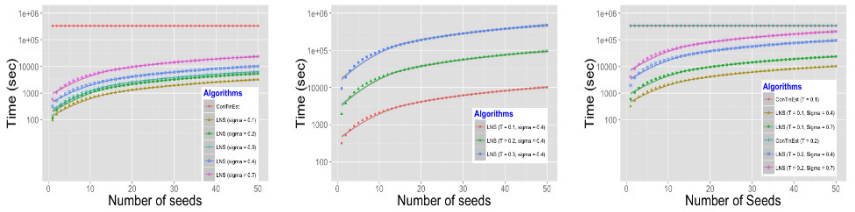
Figure 4(a) demonstrates results for the three smallest graphs in the dataset. As it can be seen, the CPU implementation of `LNS` is the slowest one. This is expected as the method is designed for parallel computing and is not a good fit for serial execution. Indicatively, the GPU implementation of `LNS` is dramatically faster by a factor ranging from x100 to x1000, justifying the above argument. One interesting finding is that for graph sizes like the ones in Figure 4(a), the sample-parallel implementation of `ConTinEst` is slower by a factor ranging between x2 and x5 compared to the serial implementation. This can be justified by the fact that only a small batch of samples can be ported into the GPU each time, due to the large space complexity of the process. This, in turn, necessitates multiple kernel calls which incur significant communication overhead between the host system and the GPU device. Finally, it can be seen that the GPU-based implementation of `LNS` and the CPU-based implementation of `ConTinEst` have similar run-times for the set of smaller networks.

However, as seen in Figure 4(b), the performance of `LNS` surpasses that of `ConTinEst` for larger networks ($>$ 100K edges). Specifically, for deadline T fixed to 0.2, `LNS` is x13.7 and x6.8 times faster than `ConTinEst` for **soc-Epinions1** and **soc-Slashdot0922**, respectively. Further, the performance of `LNS` improves significantly as the deadline becomes smaller (stricter constraints), while `ConTinEst`



(a) CPU/GPU implementations (b) Runtime vs. number of edges (c) Effect of σ

Fig. 4: Runtime results



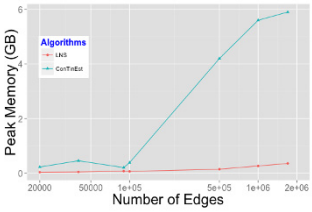
(a) LNS Runtime vs. # seeds (varying σ) (b) LNS Runtime vs. # seeds (varying T) (c) LNS vs. ConTinEst (varying T and σ)

Fig. 5: Effect of T and σ on runtime

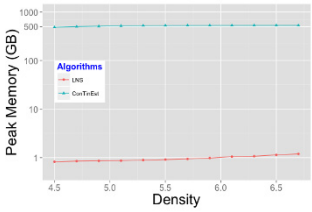
largely remains unaffected. In more detail, for T fixed to 0.1, `LNS` is faster across all social graphs, with improvements ranging between x2.1 and x85.7. In fact, the greatest gains appear for the three largest graphs in our dataset — **soc-Epinions1**, **soc-Slashdot0922** and **ego-twitter** — for which the runtime improvements are x47.9, x85.7 and x5.2, respectively. For **ego-twitter**, `ConTinEst` consumes $\approx$ 420K seconds, while `LNS` terminates after $\approx$ 80K seconds. In absolute terms, these savings correspond to days of computations (*i.e.,* savings amounting to 4 days for **ego-twitter**), which is substantial in a realistic campaign planning process. Finally, note that `ConTinEst` runtime remains similar moving to shorter deadlines, whereas `LNS` is 66.9% faster, on the average. This stresses the merits of leveraging deadline information prior to influence estimation.

We also report how the choice of σ affects runtime for the GPU-based implementation of `LNS`.Results are obtained on **soc-Epinions1**. Figure 4(c) shows that runtime grows exponentially with σ. This is expected, since for a larger σ, the borders of the local regions extend far beyond the deadline, possibly to a point where the local regions for multiple vertices cover the entire reachable set of nodes from the corresponding source nodes.

Finally, Figure 5 illustrates the relation between runtime and the number of seeds (logarithmic scale). Results are obtained on **soc-Slashdot0922**. In Figure 5(a), we fix $T = 0.1$ and let $\sigma \in \{0.1, \ldots, 0.7\}$. One observes that `LNS` runtime is linear in the number of seeds to obtain, irrespective of σ. Further, in Figure 5(b), we fix $\sigma = 0.4$ and let $T \in \{0.1, 0.2, 0.3\}$. Again, T does not affect



(a) Different graphs, Number of simulations = 100



(b) Different densities, Number of simulations = 10000

Fig. 6: Peak memory vs. density (deadline $T = 0.2$)

linearity. However, the impact of increasing T is much larger than increasing σ. Finally, Figure 5(c) contrasts the behavior of `LNS` to that of `ConTinEst` when the deadline varies. One can observe the following: (a) `ConTinEst` runtime is unaffected by the deadline constraint, since this is only used as a query after all computations are completed, and (b) it is also largely unaffected by the number of seeds to be obtained, which is beneficial when obtaining a relatively large sized seed set. In summary, `LNS` outperforms `ConTinEst` for relatively short deadlines and reasonably strict budget constraints (i.e., challenging viral marketing cases).

4.3 Memory Consumption

In this section we empirically evaluate to what extent `LNS` and `ConTinEst` stress memory when implemented using their faster variants. We report peak memory consumption for GPU-based `LNS` and CPU-based `ConTinEst`, which are set to perform 100 sampling rounds. Figure 6(a) confirms the memory intensiveness of `ConTinEst`, especially for larger graphs, while it shows that `LNS` maintains memory consumption relatively low, even as graph size increases. Specifically, `ConTinEst` consumes between 0.21GB and 5.9GB, while peak memory for `LNS` ranges between 0.04GB and 0.36GB, which corresponds to an average improvement of 1260%.

Finally, we discuss the effect of graph density on memory consumption for 10,000 iterations on both methods, and a social graph with 70K nodes and 500K edges. Figure 6(b) demonstrates that `ConTinEst` requires approximately 500GB of space to accommodate computations. The bulk of this space is occupied by the least label list structures, which store information for all 10,000 sampling iterations to support faster queries. As the density of the graph drops (from 7.0 to 4.5), peak memory remains at similar levels (8.3% drop). In contrast, `LNS` starts off at 1.2GB and for the smallest density it only consumes 0.8GB; a 32% reduction by virtue of producing shallower local subgraphs.

5 Conclusion

We present a novel approximation framework for influence maximization in the continuous-time domain. Our work addresses two drawbacks of existing meth-

ods: the lack of node-level parallelism, and the memory intensive nature of the dominant methodologies. The proposed approximation algorithm is the first to enable node-parallel influence estimation in the continuous-time setting, while maintaining memory requirements relatively low. By employing commercial-level GPUs we dramatically speed-up computations with minimal impact on accuracy.

6 Appendix

In this section, we explain our criterion for selecting local subgraphs. As mentioned in Section 3, the transmission times $\{f_{uv}(\tau_{uv})\}_{(u,v)\in\mathcal{E}}$ are differently distributed across the edges, but are mutually independent. Consequently, the joint distribution of the transmission times is fully factorized. Also, the variance of a path q is the sum of the variances of the distributions corresponding to the edges on q, i.e., $Var(q) = \sum_{(u,v)\in q} Var(u,v)$, where $Var(u,v)$ is the variance of the distribution f_{uv} associated with the edge $(u,v) \in \mathcal{E}$. We use these properties to define our criterion for selecting the local subgraphs corresponding to each node. By linearity of expectation, if q is a path from node u to node v, the expected time for u to influence v along q equals the sum of the means of the distributions corresponding to the edges on q. It also holds that the expected shortest distance from u to v equals the length of the shortest path from u to v with edge weights equal to the means of the corresponding distributions. Thus, if there is a path q from u to v whose expected length is less than the deadline T, then v is most likely to be within the influence spread of u for an arbitrary given sample, and should be included in the local subgraph of u. As such, running Dijkstra with edge weights being the distribution means allows us to extract these subgraphs.

Unfortunately, the converse is not true, i.e., if the shortest path from u to v has expected length greater than T, it does not mean that v cannot be in the influence spread of u for any sample. Since each sample is a set of random numbers generated from a given set of distributions, it is entirely possible that, for a given sample, the edge weights on a path from u to v are small enough to have v influenced by u for that particular sample. Fortunately, the frequency of such an incident happening decreases as the expected shortest distance between u and v progressively increases beyond the deadline T. Since most of the mass of a probability distribution is concentrated near its mean and is within some multiple of its standard deviation, we use that fact to decide whether v is going to be in the influence spread of u for a significant number of samples. This is the reason behind our selection criterion at Line 25 of Algorithm 2:

$$\textbf{if } distance[u] - \sigma * \sqrt{variance[u]} < T \textbf{ then}$$
$$dijkstraTree.\text{add}([u, parent[u]]).$$

We measure the variance of a node u as the variance of the shortest path between the source node and u. The parameter σ in the above criterion is a free parameter. It suggests how much of the variability in the model we are willing to account for. If we increase σ, we cover wider local neighbourhoods, thereby

improving accuracy, but at the cost of runtime. On the other hand, if we decrease σ, we obtain narrower local neighbourhoods, which leads to faster computations, but at the cost of accuracy. Our experiments suggest that, for moderately big graphs, we can choose $\sigma = 0.9$, while for much larger graphs, $\sigma = 0.3$ suffices.

Finally, the subgraph we choose for each node is a portion of the Dijkstra tree, where we only keep the nodes that satisfy the above selection criterion. Replacing a local subgraph by a local subtree leads to significant under-approximations of the influence spread. But, as our empirical evaluations in Section 4 show, it is a good representative region to sample from for deciding whether v lies in the influence spread of u for an arbitrary given sample.

References

1. Chen, W., Wang, C., Wang, Y.: Scalable influence maximization for prevalent viral marketing in large-scale social networks. In: Proceedings of the 16^{th} ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1029–1038. KDD'10 (2010)
2. Cohen, E.: Size-estimation framework with applications to transitive closure and reachability. J. Comput. Syst. Sci. 55(3), 441–453 (1997)
3. Du, N., Song, L., Gomez-Rodriguez, M., Zha, H.: Scalable influence estimation in continuous-time diffusion networks. In: Advances in Neural Information Processing Systems. NIPS'13 (2013)
4. Du, N., Song, L., Yuan, M., Smola, A.J.: Learning networks of heterogeneous influence. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 25. pp. 2780–2788 (2012)
5. Gomez-Rodriguez, M., Balduzzi, D., Schölkopf, B.: Uncovering the temporal dynamics of diffusion networks. In: Proceedings of the 28^{th} International Conference on Machine Learning. pp. 561–568 (2011), `http://www.icml-2011.org/papers/354_icmlpaper.pdf`
6. Gomez-Rodriguez, M., Schölkopf, B.: Influence maximization in continuous time diffusion networks. In: Proceedings of the 29^{th} International Conference on Machine Learning. pp. 313–320 (2012)
7. Goyal, A., Lu, W., Lakshmanan, L.V.S.: Simpath: An efficient algorithm for influence maximization under the linear threshold model. In: 2011 IEEE 11th International Conference on Data Mining. pp. 211–220 (2011)
8. Horel, T., Singer, Y.: Maximization of approximately submodular functions. In: Lee, D.D., Sugiyama, M., Luxburg, U.V., Guyon, I., Garnett, R. (eds.) Advances in Neural Information Processing Systems 29, pp. 3045–3053. Curran Associates, Inc. (2016)
9. Kempe, D., Kleinberg, J., Tardos, E.: Maximizing the spread of influence through a social network. In: Proceedings of the 9^{th} ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 137–146. KDD'03 (2003)
10. Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J., Glance, N.: Cost-effective outbreak detection in networks. In: Proceedings of the 13^{th} ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 420–429. KDD'07 (2007)
11. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data` (Jun 2014)

12. Li, L., Zha, H.: Learning parametric models for social infectivity in multi-dimensional hawkes processes. In: Proceedings of the 28^{th} AAAI Conference on Artificial Intelligence. pp. 101–107. AAAI'14 (2014)
13. Liu, X., Li, M., Li, S., Peng, S., Liao, X., Lu, X.: Imgpu: Gpu-accelerated influence maximization in large-scale social networks. IEEE Transactions on Parallel and Distributed Systems 25(1), 136–145 (2014)
14. Nemhauser, G.L., Wolsey, L.A., Fisher, M.L.: An analysis of approximations for maximizing submodular set functions. Mathematical Programming 14(1), 265–294 (1978)
15. Richardson, M., Domingos, P.: Mining knowledge-sharing sites for viral marketing. In: Proceedings of the 8^{th} ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 61–70. KDD'02 (2002)
16. Yang, S.H., Zha, H.: Mixture of mutually exciting processes for viral diffusion. In: Proceedings of the 30^{th} International Conference on Machine Learning. vol. 28, pp. 1–9 (2013)