Same bang, fewer bucks: efficient discovery of the cost-influence skyline

Matthijs van Leeuwen* Antti Ukkonen†

Abstract

Influence maximization aims to find a set of persons in a social network that can be used as *seeds* for a viral marketing campaign, such that the expected spread of influence is maximized. Standard approaches to this problem produce a single seed set that either maximizes influence, or the "bang for the buck" if the vertices are associated with a cost.

In this paper we consider the problem of finding the *cost-influence skyline*, i.e., the collection of all seed sets that are Pareto optimal w.r.t. seeding cost and expected influence. Computing the cost-influence skyline has a number of advantages over finding a single solution only. First, it provides a better understanding of the trade-off between cost and influence, which enables the user to make an informed choice regarding the budget. Second, by computing the cost-influence skyline we obtain the optimal seed set for any given seeding budget, not only the one that corresponds to singleton solutions found by existing algorithms.

In practice, the problem is to discover the skyline w.r.t. two functions spanned by all subsets of size k of a set of vertices. Due to the extremely large number of such subsets, this is a very hard problem. We present an efficient heuristic algorithm for computing the skyline when one of the functions is linear (e.g., the seeding cost) and the other submodular (e.g., expected influence). The experiments show that the cost-influence skyline can be computed in reasonable time for networks with up to a million vertices.

Keywords: influence maximization, Pareto front, skyline, algorithms, viral marketing

1 Introduction

Viral marketing aims to use existing social networks for marketing purposes, e.g., to increase product sales. By introducing a message that is designed *to go viral*, the goal is that persons will share the message with their contacts. An important parameter is the set of persons that is initially targeted: by choosing a set of *influential* persons that are *widely distributed* in the network, it becomes more likely for the message to go viral. The problem of selecting an influential set of persons has been formalized as a discrete optimization problem in the seminal paper by Kempe et al. [11]. Informally, the goal of the influence maximization (InfMax) problem is to select k persons, often called *seeds*, such that the expected spread of influence is maximized.

*Machine Learning, Department of Computer Science, KU Leuven, Leuven, Belgium, *email:* matthijs.vanleeuwen@cs.kuleuven.be.

†Helsinki Institute for Information Technology HIIT, Aalto University, and, Finnish Institute of Occupational Health, *email:* antti.ukkonen@ttl.fi

The standard problem does not take into account the *costs* of targeting individual seeds, which may be variable in practice. Consider, for example, a company giving away for free k products. Although the product costs are constant and can therefore be ignored, mailing costs might depend on the geographical locations of the persons. When there are no direct costs, costs could represent the lost sales due to giving away products, i.e., the expected loss of income due to the k seeds *not* buying the product. As a final example, consider a company that aims to convince k potential buyers to adopt a certain product by offering personalized rebates. Here, the cost associated with a seed is the amount of the rebate that is deemed necessary to convince this person to buy the product.

Clearly, this (literally) adds a new dimension to the problem: higher seeding costs are likely to permit larger expected influence spreads. In general, different campaign budgets will correspond to different influence-maximizing seed sets. This raises a hard question: *how to choose a budget without having any knowledge about the trade-off between cost and expected influence?* The standard greedy approach [11] is cost-oblivious and its solution may be more expensive than the budget allows. An alternative would be to optimize the cost-influence ratio [12], but this "bang-for-the-buck" (BFTB) approach also gives you just one solution – if the budget exceeds the costs for this seed set, the method cannot identify other, more appropriate solutions.

1.1 Approach and contributions

To solve this problem, we introduce the *cost-influence skyline*, i.e., the Pareto optimal front of seed sets of size k with respect to cost and expected influence spread. Each solution on the skyline is non-dominated, meaning that there exists no seed set for which either cost or influence can be improved without adversely affecting the other quantity. Hence, each Pareto optimal seed set by definition provides a favorable trade-off between cost and influence and is therefore a desirable solution.

We consider two skyline variants. The first variant consists of seed sets of size exactly k, which could be interpreted as, e.g., giving away for free exactly k products. It can also be useful, however, to interpret k as the *maximum* number of desired seeds and thus the

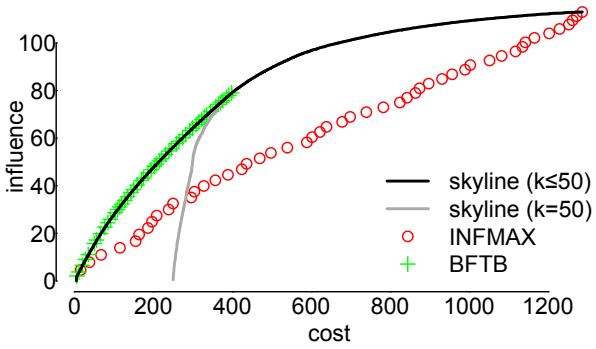

Figure 1: Cost-influence skylines for *com-dblp*, obtained using the FAST-SKYLINE algorithm that we introduce in this paper (black: $k \leq 50$, gray: $k = 50$). The red dots resp. green crosses show the sequences of solutions for $k = 1$ to 50 found by the cost-oblivious greedy influence maximization algorithm [11] resp. the bang-for-the-buck greedy algorithm [12]. Vertex costs are random and $p = 0.01$; see Section 6 for more details.

second variant is the *global* skyline over all seed sets of sizes 1 up to k. As we will show, our approach allows to naturally obtain both skylines at the same time.

Figure 1 shows example cost-influence skylines that were obtained by the algorithm that we introduce. For comparison, the figure also depicts sequences of consecutive solutions obtained with the standard and bang-for-the-buck greedy algorithms. This shows that the $k \leq 50$ skyline includes all Pareto optimal solutions found by the two existing methods, but additionally provides a number of potentially interesting alternatives spanning a large cost range. Moreover, the skyline reveals that it is possible to choose a seed set with much lower cost than 1200, the cost of the InfMax solution, while hardly giving in on influence – we can get almost the same, maximum bang for much fewer bucks.

More formally, we consider given a (social) network G, the number of seeds k, an influence function $I(X)$ that returns the expected number of influenced (and activated) nodes when seeding X, and a cost function $C(X)$ that gives the seeding cost of X. Furthermore, the seeding cost of a seed set is simply the sum of the individual costs, i.e., $C(X) = \sum_{x \in X} C(x)$, where $C(x)$ is the seeding cost of x. The problem is to find the cost-influence skyline, that is, all seed sets that are Pareto optimal with respect to $I(\cdot)$ and $C(\cdot)$.

In short, the main contributions of this paper are:

1. We formalize the problem of discovering cost-influence skylines and prove that this is an NP-hard problem when $I(\cdot)$ is submodular.
2. We introduce FAST-SKYLINE, an efficient and effective heuristic algorithm for computing skylines. The algorithm performs levelwise search and the solutions it finds are guaranteed to include the solution found by the standard greedy method.
3. We empirically demonstrate that cost-influence skylines can be computed in reasonable time for networks containing up to a million vertices. We compare its performance to that of the naive skyline algorithm that we introduced in the context of pattern mining [19] and show that FAST-SKYLINE is orders of magnitude faster. Furthermore, we show that the proposed pruning technique is often nearly optimal.
4. We show that cost-influence skylines provide useful insight into the trade-off between cost and expected influence, and provides a wider range of (budgetary) options than existing methods.

Note that a propagation model is needed to compute the expected spread of influence for a given seed set. One of the commonly used propagation models is the independent cascade model [11], which has the large advantage that influence spread is submodular. As a consequence, the standard greedy algorithm is guaranteed to give good approximations [16]. In the remainder of this paper, we consider the independent cascade model with constant probabilities, but our methods are agnostic w.r.t. the propagation model as long as the influence function $I(\cdot)$ is submodular, and $C(\cdot)$ is a linear function of vertex-specific costs.

We continue with a discussion of related work in Section 2. After providing definitions and background information in Sections 3 and 4 respectively, Section 5 introduces FAST-SKYLINE. Our approach is empirically evaluated in Section 6 and we conclude in Section 7.

2 Related work

Conceptually the cost-influence skyline that we propose is inspired by optimal portfolio theory [15]: for any given cost (risk) we should find the seed set (portfolio of securities) that maximizes influence (return). The word "skyline" is inspired by database literature [4], that uses the same expression to describe a set of *Pareto optimal* solutions. However, the main technical challenge we face is that of finding the *Pareto optimal subsets of size* k subject to two functions, not computing the Pareto optimal solutions of aggregates over a database table.

This problem also appears in the context of *multi-objective combinatorial optimization*, see [7, 8] for surveys on that topic. The main difference to that line of work is that we want to compute *all* Pareto optimal solutions rather than, e.g., use a utility function (i.e., *scalarization)* to select one of the Pareto optimal solutions. Instead, we substantially extend our work in [19]

to handle much larger problem instances, and show how skylines can be applied in the context of viral marketing.

The seed selection problem for influence maximization was originally introduced by Kempe et al. [11], and has attracted considerable attention during the past ten years. Giving a comprehensive survey is beyond the scope of this paper. For a recent survey we refer to [2].

Influence maximization under the independent cascade model can be seen as a SET-COVER type of problem, and hence a greedy algorithm that maximizes marginal gains is a common solution. A trivial implementation, however, is not scalable for large networks. Therefore, several approaches to speed up influence estimation have been proposed [12, 5, 10]. Most recent in this line of work are [6, 3, 18]; all avoid simulating the independent cascade model during the seed selection phase by using pre-computed data structures. We make use of the same technique (in particular the one proposed in [6]) to speed up evaluating $I(\cdot)$. However, any other technique could be applied by our algorithm as well.

Seed selection has also been considered in the context of several *marketing scenarios*. Recently, Lu and Lakshmanan [14] introduced the ProMax problem, which consists of both assigning individual product prices to each person and selecting a seed set that maximizes profit. Since the influence function is still submodular, the greedy algorithm can still be used. Apart from that, methods are introduced to determine the price vectors. Lu et al. [13] consider the seed selection problem when marketing campaigns for several *competing products* are launched at the same time. Finally, [9] consider the seed selection problem when *propagation time* is taken into account.

3 Problem definition and complexity

We assume given a graph, denoted by $G = (V, E)$, with V the set of vertices and E the set of edges. Let $V_i = \{X \subset V \mid |X| = i\}$. We assume given an influence function $I(\cdot)$ that is *submodular*, and a cost function $C(\cdot)$ that is the *sum of vertex-specific costs*. We use the independent cascade model [11] in our experiments, but any submodular influence function can be used.

DEFINITION 1. *The influence function $I(\cdot)$ is submodular whenever*

$$I(X \cup u) - I(X) \leq I(Y \cup u) - I(Y)$$

for every $u \in V$ and $Y \subseteq X \subseteq V$.

In general, the *skyline*, or Pareto front, of a set of points are those points that are not *dominated* by any other point in the set. We define dominance for two sets of vertices in terms of cost and influence, as follows.

DEFINITION 2. *Let $X, Y \subset V$. The set Y* dominates *the set X in terms of $C(\cdot)$ and $I(\cdot)$, denoted $Y \succ X$, iff*

$$\big(C(Y) < C(X) \quad \textit{and} \quad I(Y) \geq I(X)\big) \ \textit{or}$$
$$\big(I(Y) > I(X) \quad \textit{and} \quad C(Y) \leq C(X)\big).$$

That is, the set Y dominates the set X if it has strictly lower cost or higher influence than X, and is at least at the level of X w.r.t. the other function.

DEFINITION 3. *The* cost-influence skyline *of all i-sized subsets of V is defined as the set*

$$P_i = \{X \in V_i \mid \nexists Y \in V_i \textit{ s.t. } Y \succ X\}.$$

Our main problem is that of computing the cost-influence skyline of all k-sized vertex sets[1].

PROBLEM 1. (COST-INFLUENCE SKYLINE) *Given the graph $G = (V, E)$, an integer k, the cost function $C(\cdot)$, and the influence function $I(\cdot)$, compute the cost-influence skyline P_k as defined in Definition 3.*

Observe that the complexity of finding the skyline for a given set of points is polynomial in the number of the points. However, in this case the size of the input is in fact *exponential* in the size of V, because we are computing the skyline over V_k, and $|V_k| = \binom{|V|}{k}$. Any trivial algorithm for computing skylines can thus not be used. Moreover, we can easily show that this problem is NP-hard for submodular influence functions.

PROPOSITION 3.1. *For submodular influence functions $I(\cdot)$, the* COST-INFLUENCE SKYLINE *problem (Problem 1) is NP-hard.*

Proof. Recall that the problem of finding the set $X^* \in V_k$ that maximizes influence is NP-hard when $I(\cdot)$ is submodular. Now, observe that X^* must belong to P_k, because X^* has the largest influence among all sets in V_k by definition. Hence, computing P_k must be at least as hard as computing X^*. □

We are thus not aiming to find optimal skylines, but focus on finding good skylines instead.

4 Levelwise skyline search

In this section we discuss a greedy, *levelwise* approach to computing the skyline of V_k.

[1] For reasons of brevity, we only define and analyze the skyline problem for seed sets of exactly size k, but we will also show how our approach finds skylines of seed sets of sizes 1 up to k.

Algorithm 1 NAIVE-SKYLINE$(X_1, \ldots, X_n)$

```
initialize the list L ← [X_1, X_2, ... X_n]
sort L in increasing order of C(X_i)
I_max ← 0
P ← ∅
for i = 1 to |L| do
    X ← L[i]
    if I(X) > I_max then
        I_max ← I(X)
        P ← P ∪ X
return P
```

4.1 A naive skyline algorithm We start by discussing a simple algorithm for computing the skyline of a given set of vertex subsets $X_1, \ldots X_n$ with respect to $C(X_i)$ and $I(X_i)$. This algorithm, called NAIVE-SKYLINE, shown in Algorithm 1, scans over the X_i in increasing order of $C(X_i)$, and keeps track of the largest influence, denoted $I_{\max}$, seen so far. A subset X enters the skyline P if $I(X)$ is larger than $I_{\max}$. It is easy to see that this algorithm will only include non-dominated points into P, and moreover, it will include *all* non-dominated points into P.

4.2 A greedy strategy for skyline search The brute force solution to Problem 1 of first materializing all $\binom{|V|}{k}$ subsets of size k and running Algorithm 1 on these is clearly infeasible even for relatively small $|V|$ and k. The LEVELWISE heuristic proposed in [19] avoids considering all possible subsets by employing a greedy strategy. It computes P_i given P_{i-1} by first combining every point in P_{i-1} with every possible vertex in V, and then finds the skyline of the resulting set of points using, e.g., Algorithm 1. More formally, we define the *expansion* of P_{i-1} as

$$\mathcal{E}(P_{i-1}) = \{X \cup u \mid X \in P_{i-1} \wedge u \in V \setminus X\}.$$

Every subset in $\mathcal{E}(P_{i-1})$ is thus of the form $X \cup u$, where $X \in P_{i-1}$, and u is some vertex not in X. LEVELWISE starts with $P_0 = \{\emptyset\}$ and iterates the following equation from $i = 1$ until $i = k$:

$$P_i = \text{NAIVE-SKYLINE}(\mathcal{E}(P_{i-1})). \tag{4.1}$$

The skyline P_i is thus defined as the skyline of the expansion of P_{i-1}. In [19] it was shown that, in the context of pattern set mining, this heuristic produces skylines that are very close to the exact skylines.

The levelwise search procedure procedure is illustrated in Figure 2, which shows (hypothetical) cost-influence skylines for $k = 1$ to 3. In practice the skyline moves towards higher cost and influence as k increases.

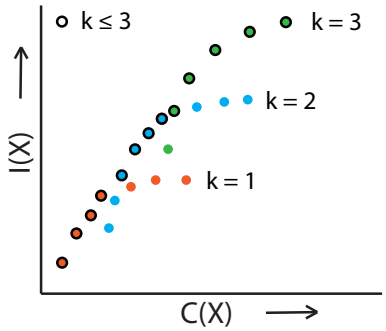

Figure 2: Levelwise cost-influence skyline search. Starting from the skyline for $k = 1$ (orange dots), each point on the skyline is expanded in every possible way. From the generated candidates, the skyline for $k = 2$ is obtained (blue dots). This procedure is repeated until the desired k is reached; in this case $k = 3$ (green dots). The black circles indicate the global skyline for $k \leq 3$.

A key advantage of levelwise search is that the process of obtaining the skyline for a fixed k actually provides *all skylines for sizes* 1 *up to* k. As a consequence, the online and iterative computation of the global skyline is trivial: start with $P_1^{\text{global}} = P_1$ and then compute P_i^{global} as the skyline of P_{i-1}^{global} and P_i, which can be done, e.g., using Algorithm 1. Hence, we obtain not only the skyline for a fixed seed set size, but also the global skyline for seed sets up to that size. Figure 2 shows an example of such a global skyline for $k \leq 3$.

4.3 Theoretical results for levelwise skyline search Next we present some results that show how the (single-level) skyline found by the levelwise approach relates to the solutions found by standard greedy methods. First, for any k, the skyline P_k found by the levelwise algorithm contains the solution found by the simple greedy algorithm that maximizes marginal gain in influence on every step.

PROPOSITION 4.1. *Let* X_k^{InfMax} *denote the solution of size* k *found by the algorithm that greedily maximizes marginal gain in* $I(\cdot)$ *on every step, and let* P_k *denote the solution found by levelwise skyline search. When* $I(\cdot)$ *is submodular, we have*[2] $X_k^{\text{InfMax}} \in P_k$.

Proof. Please see supplementary information.

[2] When multiple solutions with the same influence exist, the greedy algorithm finds just one of those. Since it is cost-agnostic, this may be one with higher cost than the one found by the skyline algorithm, as is the case in Figure 1. Since the effect is negligible in practice, we ignore this subtlety.

This implies that the $(1-1/e)$ approximation that holds for the solution found by the greedy algorithm [16] also holds for the point of largest influence in P_k. Hence, *that part of the resulting skyline cannot be arbitrarily bad* when maximizing influence.

Of course we can also maximize the "bang for the buck" using a greedy algorithm, as discussed in the Introduction. This algorithm, which we call BFTB, adds at every step the item u that maximizes the ratio $\frac{I(X\cup u)-I(X)}{C(u)}$, where X is the current solution set. The cost-influence skyline P_k that is found by the levelwise method can dominate the solution found by BFTB for k.

PROPOSITION 4.2. *Let X_k^{BFTB} denote the solution of size k found by the algorithm that greedily maximizes $\frac{I(X\cup u)-I(X)}{C(u)}$ on every step, and let P_k denote the solution found by levelwise skyline search. When $I(\cdot)$ is submodular, there can exist some $X \in P_k$ such that $X \succ X_k^{\text{BFTB}}$.*

Proof. Please see supplementary information.

In practice we observe that the BFTB solution tends to belongs to the skyline, however.

5 The FAST-SKYLINE algorithm

We continue by introducing a levelwise algorithm that exploits the linearity of $C(\cdot)$ as well as the submodularity of $I(\cdot)$. These allow us to address the scalability issues that the LEVELWISE approach suffers from.

1. Equation 4.1 requires to combine every item in V with every subset in P_{i-1}. (Except those that are already contained in a particular subset X.) The set $\mathcal{E}(P_{i-1})$ is thus of size $O(|P_{i-1}||V|)$, which is still linear in $|V|$, but in practice very large. Rather than materializing the list L in Algorithm 1, we will construct an iterator that will produce the list L *in increasing order of* $C(\cdot)$. This is possible because $C(\cdot)$ is linear.
2. Algorithm 1 computes $I(X)$ for every $X \in \mathcal{E}(P_{i-1})$. This becomes prohibitively slow even if recently proposed fast influence estimation algorithms (such as [6, 18]) are used. The submodularity of $I(\cdot)$ allows us to define an upper bound to $I(X \cup u)$ that is very fast to compute in comparison to the actual value of $I(X \cup u)$. We will use this upper bound on skip some of the X that will not satisfy the $I(X) > I_{\max}$ condition on line 7 of Algorithm 1.

These improvements allow us to scale up the levelwise search strategy to *several orders of magnitude larger inputs*.

On a high level the proposed algorithm works as Algorithm 1. It iterates over sets in $\mathcal{E}(P_{i-1})$ in increasing order of $C(\cdot)$, and adds the set X to P_i whenever $I(X)$ is greater than $I_{\max}$. The main difference is in how the set $\mathcal{E}(P_{i-1})$ is accessed, and that $I(X)$ is not computed for all X.

5.1 Iterating over $\mathcal{E}(P_{i-1})$ in increasing order of $C(X)$

Consider the *sorted* list L as defined in Algorithm 1. Next we show how to generate L in sorted order without first computing $C(X)$ for every $X \in \mathcal{E}(P_{i-1})$ and then sorting the resulting list.

A first observation is that, for some fixed $X \in P_{i-1}$, the set $X \cup u$ appears before the set $X \cup v$ in L if and only if $C(u) \leq C(v)$. Therefore, we do not have to consider the set $X \cup v$ until all sets $X \cup u$ with $C(u) < C(v)$ have been processed.

Let U denote a list of all vertices $u \in V$ that is *sorted in increasing order of* $C(u)$, and denote by $U[l]$ the vertex at position l in this list. Moreover, let X_j denote the the j:th set in the skyline P_{i-1} for every $j = 1, \ldots |P_{i-1}|$. We associate to every X_j an index to the list U, denoted pos_j. If there was only one subset in P_{i-1}, denoted X_1, we could iterate over the list L in increasing order of cost simply by outputting the sets $X_1 \cup U[\text{pos}_1]$ for $\text{pos}_1 = 1, \ldots, |V|$ (and omitting those $u \in V$ that are already contained in X_1).

When there is more than one subset X_j in P_{i-1}, the next subset in L is always $X_j \cup U[\text{pos}_j]$ for the j that minimizes $C(X_j \cup U[\text{pos}_j])$. To find this j in an efficient manner, we maintain a (min) priority queue Q that contains the sets $X_j \cup U[\text{pos}_j]$, for $j = 1, \ldots |P_{i-1}|$, with $C(X_j \cup U[\text{pos}_j])$ as the priority. (The lower the cost, the higher the "priority".) The queue Q is initialized by letting $\text{pos}_j = 1$ for all j, and inserting the sets $X_j \cup U[1]$. When the set $X_j \cup U[\text{pos}_j]$ is obtained from the queue, we increment pos_j by one, and insert the new $X_j \cup U[\text{pos}_j]$ into the queue. The next subset of L is always the one at the head of Q.

5.2 Upper bound for expected influence

Even if fast influence estimation algorithms are used, computing $I(X)$ for every $X \in \mathcal{E}(P_{i-1})$ is still a bottleneck. A well-known optimization to the greedy algorithm [12] makes use of submodularity to avoid computing $I(X)$ when X can not enter the current solution. In particular, the trick is to to find an upper bound on the marginal gain $\Gamma(X, u) = I(X \cup u) - I(X)$ of u when added to X. Recall that the greedy algorithm always adds the $u \in V \setminus X$ that maximizes $\Gamma(X, u)$. If this upper bound is lower than the current best marginal gain, we know that u can not enter the solution.

We use a similar technique to prune unnecessary evaluations of $I(X)$ for a large part of the $X \in \mathcal{E}(P_{i-1})$ in our algorithm. Recall that Algorithm 1 keeps track

of $I_{\max}$, the largest influence seen so far, and the set X can enter P_i only if $I(X)$ is larger than $I_{\max}$. However, if already the upper bound of Eq. 5.2 is lower than $I_{\max}$, we can skip computing $I(X)$.

We assume $I(\cdot)$ to be submodular, thus Definition 1 implies that

$$I(X \cup u) \leq I(X) + \Gamma(Y, u), \tag{5.2}$$

where $Y \subseteq X$. Since every set in $\mathcal{E}(P_{i-1})$ is of the form $X \cup u$, where $X \in P_{i-1}$, and $I(X)$ is known for every $X \in P_{i-1}$, we can compute the upper bound by finding the smallest known $\Gamma(Y, u)$ where $Y \subseteq X$. We do this by maintaining a *marginal gain cache* of all marginal gains $\Gamma(Y, u)$ that have been computed.

5.3 Marginal gain cache The marginal gain cache is a data structure that represents (explicitly or implicitly) a set of triplets $(u, Y, \Gamma(Y, u))$. It provides the operations MGC-PUT$(u, Y, \Gamma(Y, u))$ and MGC-GET(u, X). The insert operation inserts the triplet $(u, Y, \Gamma(Y, u))$ into the cache. The retrieve operation is defined as

$$\text{MGC-GET}(u, X) = \min_{(v, Y, \Gamma(Y,v))} \{\Gamma(Y, u) \mid v = u \wedge Y \subset X\}, \tag{5.3}$$

where minimization is over all triplets in the cache.

Different implementations of the cache can be devised. The only real requirement is that the cache performs substantially faster than computing $I(\cdot)$. We use a simple approach where the cache associates to every $u \in V$ a list of only those triplets that contain u. Inserting is done by appending to this list, while MGC-GET scans the list and finds the triplet with the smallest marginal gain $\Gamma(Y, u)$ where $Y \subset X$.

5.4 Combining everything The final algorithm, FAST-SKYLINE, is shown in Algorithm 2. To compute the skyline at level k, we first compute the first level skyline P_1, e.g., using Algorithm 1, and then iterate FAST-SKYLINE until the given k.

Alg. 2 takes P_{i-1} as input and computes P_i. The loop on lines 6–19 corresponds to the **for** loop on lines 5–9 of Algorithm 1. The **if** statement on line 10 queries the marginal gain cache to check if $X_j \cup u$ can enter the solution. If yes, we compute the actual marginal gain and insert this into the cache (lines 11–13). On lines 14–16 the algorithm determines if $X_j \cup u$ enters the new skyline. Finally, we update the queue Q on lines 17–19. This part implements an additional optimization that avoids inserting candidates into Q that we can prune already at this point. That is, we keep incrementing pos_j until we either reach the end of U, or we find a case that has an upper bound above the current $I_{\max}$. This leads to a substantial speedup in practice, as we can prune many of the points in $\mathcal{E}(P_{i-1})$ from entering the queue in the first place[3].

Algorithm 2 FAST-SKYLINE(P_{i-1})

```
1:  Q ← empty (min) priority queue
2:  for j = 1 to |P_{i-1}| do
3:      X_j ← j:th set in P_{i-1}
4:      insert (j, 1) into Q with priority C(X_j ∪ U[1])
5:  P_i ← ∅, I_max ← 0
6:  while Q is not empty do
7:      (j, pos_j) ← QUEUE-POP()
8:      X_j ← j:th set in P_{i-1}
9:      u ← U[pos_j]
10:     if MGC-GET(u, X_j) + I(X_j) > I_max then
11:         inf ← I(X_j ∪ u)
12:         gain ← inf − I(X_j)
13:         MGC-PUT(u, X_j, gain)
14:         if inf > I_max then
15:             I_max ← inf
16:             P_i ← P_i ∪ {X_j ∪ u}
17:     increment pos_j until pos_j > |U| or
        MGC-GET(U_{pos_j}, X_j) + I(X_j) > I_max
18:     if pos_j ≤ |U| then
19:         insert (j, pos_j) into Q with
            priority C(X_j ∪ U[pos_j])
20: return P_i
```

5.5 Filtering intermediary skylines Finally, the skylines have a lot of redundancy from a practical point of view. Often adjacent points on the skyline differ only by a very small amount in terms of $C(\cdot)$ and $I(\cdot)$. Moreover, the size of P_k (in term of the number of sets) tends to increase as k increases. As the complexity discussion below shows, the size of P_{i-1} plays an important part. We thus propose to use simple *filtering* heuristics to reduce the size of P_{i-1} before running FAST-SKYLINE. The one we use in the experiments considers P_{i-1} in increasing order of $C(\cdot)$, and omits a set X_j when the difference $C(X_j) - C(X_{j-1})$ is below some threshold θ. This turns out to have a substantial effect to the running time, and only a very small effect to the resulting skyline.

5.6 A note on complexity We assume that managing the priority queue, as well as the marginal gain cache is orders of magnitude faster than evaluating $I(\cdot)$. From a practical point of view, the relevant measure of complexity[4] is thus the number of calls to $I(\cdot)$.

[3] Some additional notes on optimizing the implementation can be found in the supplementary information.

[4] We let Ω and O denote (not necessarily tight) lower and upper bounds, respectively.

First, it is easy to observe that this number is lower bounded by the size of P_i: we must compute the influence of every set $X_j \cup u$ that enters P_i (line 11 of Alg. 2). Second, in the absence of pruning (or if nothing was pruned), we would call $I(\cdot)$ for every set in $\mathcal{E}(P_{i-1})$. We obtain thus:

PROPOSITION 5.1. *The number of times* FAST-SKYLINE *(or* any *greedy levelwise algorithm) calls* $I(\cdot)$ *is* $\Omega(|P_i|)$ *and* $O(|V||P_{i-1}|)$.

In the experiments we will measure how close to this lower bound the algorithm gets in practice. Note that the LEVELWISE algorithm discussed in Section 4.2 will make $\Omega(|V||P_{i-1}|)$ calls to $I(\cdot)$. Even if FAST-SKYLINE has the same worst-case performance, it can be up to $|V|$ times faster, assuming that P_{i-1} and P_i are roughly of the same size.

Finally, note that the size of the priority queue Q can not be larger than $|P_{i-1}|$. This is because initially Q contains $|P_{i-1}|$ items, and at every iteration of the **while** loop we pop one element from Q and insert at most one new element into Q. Moreover, at least every item in P_i must have passed through Q, and there are at most $|\mathcal{E}(P_{i-1})| = O(|V||P_{i-1}|)$ items that can possibly enter Q. This yields:

PROPOSITION 5.2. *The complexity of managing the priority queue* Q *in* FAST-SKYLINE *is* $\Omega(|P_i| \log |P_{i-1}|)$ *and* $O(|\mathcal{E}(P_{i-1})| \log |P_{i-1}|)$.

6 Experiments

We ran a number of experiments to compare FAST-SKYLINE with LEVELWISE as well as the standard greedy influence maximization algorithm. Our implementation of the greedy algorithm uses the optimizations (CELF) discussed in [12]. Moreover, to speed up simulating the independent cascade model (computing $I(X)$), we use the technique described in [6] with $R = 200$. In short, this avoids having to run a large number of simulations for each X by using a precomputed database of reachable sets for every node $u \in V$. Running times that we report do not include this preprocessing, as we want to focus on algorithm performance given an efficient method for computing $I(X)$. We also use filtering with $\theta = 1$ whenever the size of P_{i-1} is larger than 200 points. These values were set after some initial experimentation[5], and in practice filtering has negligible effect on the resulting skyline. In all experiments, individual vertex costs were drawn uniformly at random from $[5, 50]$. All algorithms were implemented[6] in JavaScript, and the experiments were run with Node.js[7] (version 0.10.28) on a 2.8GHz Intel Xeon CPU having 32GB RAM.

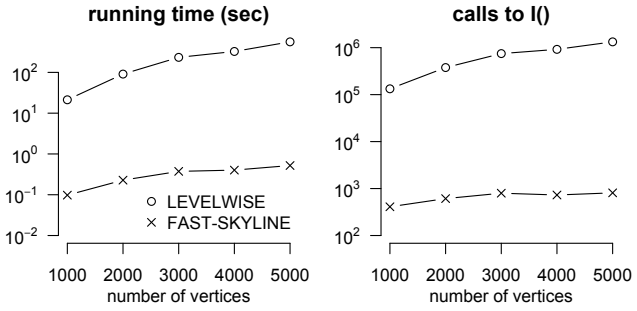

Figure 3: Running time and number of calls to $I(\cdot)$ for LEVELWISE and FAST-SKYLINE (log scale) on small Barabasi-Albert graphs as a function of network size.

6.1 Comparison with LEVELWISE We start by comparing FAST-SKYLINE with the LEVELWISE algorithm originally proposed in [19], and briefly discussed in Section 4.2. The main difference is that LEVELWISE incorporates no pruning, and simply materializes all of $\mathcal{E}(P_{i-1})$ to compute P_i. We created small random graphs using the Barabasi-Albert model [1], allocated random costs to the vertices, and ran both algorithms up to $k = 5$. Propagation probability was set to 0.1.

Figure 3 shows the results. We can observe that FAST-SKYLINE is over two orders of magnitude faster than LEVELWISE, and the difference increases even further as the network grows. It is hardly surprising that LEVELWISE is slower: the pruning really has a huge effect on the number of calls to $I(\cdot)$. It is easy to see that using LEVELWISE on networks with millions of vertices is simply not possible, especially for larger values of k.

6.2 Running times on real networks We then take a look at how FAST-SKYLINE performs on real data. We use six publicly available networks from the Stanford Large Network Dataset Collection[8]. Vertex costs were assigned uniformly at random from $[5, 50]$. Their names and basic statistics are shown in Table 1. The propagation probability of the independent cascade model was set to a uniform value of $p = 0.01$ in all cases (except $p = 0.005$ for soc-pokec). Table 1 also shows the running times (in sec) of the proposed algorithm with $k = 50$ (column FSL). As we can see, for small networks the algorithm runs in a couple of minutes,

[5] Please see the supplementary information for a study of the effect of the filtering heuristic.

[6] http://anttiukkonen.com/skyline

[7] http://nodejs.org/

[8] http://snap.stanford.edu/data/

Table 1: Dataset statistics and results for $k = 50$: skyline size, running time, and pruning efficiency

				running time			calls to $I(\cdot)$ $/\lvert P_i\rvert$		
Network	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert P_{50}\rvert$	FSL	FSL$/\lvert P_{50}\rvert$	GREEDY	min	median	max
ca-HepTh	9.8K	52.0K	1572	1m31s	<0.1s	<1.0s	1.12	1.21	1.33
ca-CondMat	23.1K	186.9K	2047	3m17s	<0.1s	<1.0s	1.18	1.24	1.54
soc-Epinions1	75.9K	508.8K	2269	46m25s	1.2s	33.8s	2.99	4.47	14.32
com-dblp	317.1K	1.05M	2294	1h38m36s	2.5s	6.8s	1.02	1.05	1.30
amazon0601	403.4K	3.39M	646	15m30s	1.4s	8.9s	1.00	1.10	1.63
soc-pokec	1.63M	30.62M	3161	4h15m59s	4.9s	34.0s	1.01	1.02	1.25

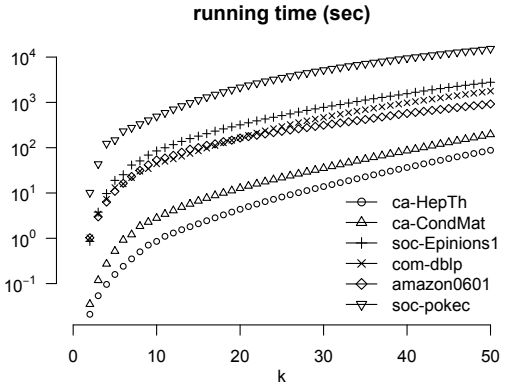

Figure 4: Running time (log scale) of FAST-SKYLINE as a function of k on real networks.

while for medium-sized networks computing the skyline takes about 1-2 hours. For the largest network (*soc-pokec*) the algorithm needs a little over four hours.

While the greedy influence maximization algorithm is orders of magnitude faster (column GREEDY), it only computes a single point for every k. FAST-SKYLINE produces a large set of Pareto optimal solutions, including the solution found by the greedy algorithm. Table 1 also shows the running time of the skyline algorithm divided by the number of points in the final solution (columm FSL$/\lvert P_{50}\rvert$), which shows that the time needed *per seed set* is very competitive. FAST-SKYLINE thus outperforms an approach based on scalarization where the greedy algorithm is invoked several times to compute P_{50} one set at a time.

It is also interesting to see how the running time of FAST-SKYLINE behaves as a function of k. This is shown in Figure 4 for all networks up to $k = 50$. We can observe that as k grows the current implementation becomes progressively slower. For small k the algorithm finds P_k easily in less than 1000 seconds in all cases except again the largest network (*soc-pokec*).

6.3 Pruning efficiency on real networks Next, we consider the efficiency of our pruning strategy. Recall that the main objective is to reduce the number of calls to $I(\cdot)$, as that is the main bottleneck of the LEVELWISE method. According to Proposition 5.1, *any* greedy levelwise algorithm that computes P_i from P_{i-1} *must* make at least $\lvert P_i\rvert$ calls to $I(\cdot)$. The pruning techniques will never be perfect, but it is interesting to see how close we get to this lower bound. We recorded the number of times FAST-SKYLINE calls $I(\cdot)$ on every level i, and divided this by the size of P_i, for $i = 2$ up to k. Results are shown in the rightmost columns of Table 1.

We can see that for every network, except *soc-Epinions1*, the maximum ratio observed is less than 2 for every i up to k, and the medians are very close to 1. This means that *the pruning strategy of* FAST-SKYLINE *is nearly optimal*. An even better pruning strategy would not lead to substantial speedups in most cases. The fact that the algorithm has problems with *soc-Epinions1* is an interesting finding as well, as it suggests that there are inputs for which the mechanism is suboptimal.

6.4 Examples of cost-influence skylines Figure 5 shows examples of global skylines we found with $k \leq 50$. The plots also show the consecutive solutions found by the greedy influence maximization algorithm (INFMAX), as well as the greedy bang-for-the-buck (BFTB) algorithm. We can see that for a given k, the cost-influence skyline contains a wide range of solutions that lie between the BFTB and INFMAX solutions. All of these seed sets have higher influence than the BFTB solution, and a lower cost than the INFMAX solution.

7 Conclusions

We introduced the concept of cost-influence skylines and presented an efficient and effective heuristic algorithm for finding them. The algorithm always finds the solution that the standard greedy influence maximization algorithm finds and in practice also finds the BFTB solution, but it discovers many alternative solutions as well. The experiments show that such skylines can be discovered for fairly large social networks in a reasonable amount of time. Furthermore, we have shown that the

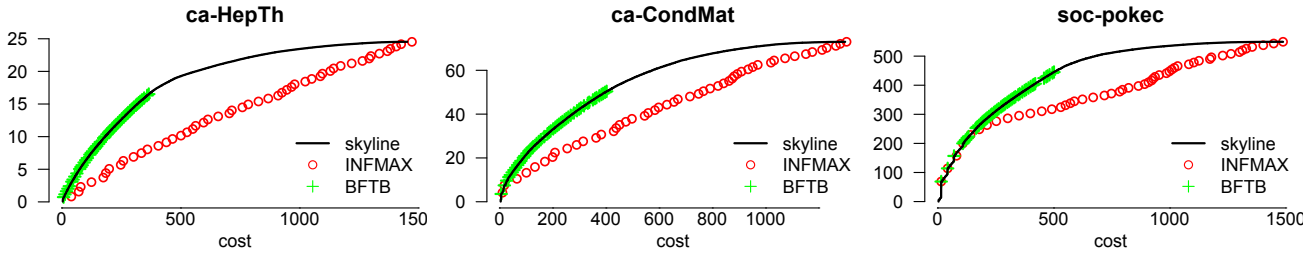

Figure 5: Global cost-influence skylines for three datasets ($k \leq 50$), computed with FAST-SKYLINE (solid black line). Also shown are the consecutive solutions found by a greedy influence maximizing algorithm (red circles), as well as a greedy bang-for-the-buck maximizing algorithm (green circles).

proposed pruning technique is often nearly optimal.

Observe that Algorithm 2 can be applied to *any* set selection problem where a sum must be minimized and a submodular function must be maximized. In this paper we showcased seed selection for viral marketing as an interesting application. Other possible use cases include sensor placement problems [12] (some locations can be more expensive than others due to environmental conditions) or conservation planning [17] (it may be useful to consider the trade-off between the costs to conserve an area and species persistence probability). In general, the problem of computing all Pareto optimal solutions in the context of multi-objective combinatorial optimization is an interesting topic for future work.

Acknowledgements Matthijs van Leeuwen is supported by a Postdoctoral Fellowship of the Research Foundation Flanders (FWO).

References

[1] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[2] F. Bonchi. Influence propagation in social networks: A data mining perspective. *IEEE Intelligent Informatics Bulletin*, 12(1):8–16, 2011.

[3] C. Borgs, M. Brautbar, J. T. Chayes, and B. Lucier. Maximizing social influence in nearly optimal time. In *SODA*, pages 946–957, 2014.

[4] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[5] W. Chen, C. Wang, and Y. Wang. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *KDD*, pages 1029–1038, 2010.

[6] S. Cheng, H. Shen, J. Huang, G. Zhang, and X. Cheng. StaticGreedy: solving the scalability-accuracy dilemma in influence maximization. In *CIKM*, pages 509–518, 2013.

[7] M. Ehrgott and X. Gandibleux. A survey and annoted bibliography of multiobjective combinatorial optimization. *OR Spektrum*, 2000.

[8] M. Ehrgott and X. Gandibleux. Approximative solution methods for multiobjective combinatorial optimization. *TOP: Journal of the Spanish Society of Statistics and Operations Research*, 12(1):1–63, 2004.

[9] A. Goyal, F. Bonchi, L. V. S. Lakshmanan, and S. Venkatasubramanian. On minimizing budget and time in influence propagation over social networks. *Social Netw. Analys. Mining*, 3(2):179–192, 2013.

[10] A. Goyal, W. Lu, and L. V. S. Lakshmanan. Celf++: optimizing the greedy algorithm for influence maximization in social networks. In *WWW (Companion Volume)*, pages 47–48, 2011.

[11] D. Kempe, J. M. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *KDD*, pages 137–146, 2003.

[12] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. M. VanBriesen, and N. S. Glance. Cost-effective outbreak detection in networks. In *KDD*, pages 420–429, 2007.

[13] W. Lu, F. Bonchi, A. Goyal, and L. V. S. Lakshmanan. The bang for the buck: fair competitive viral marketing from the host perspective. In *KDD*, pages 928–936, 2013.

[14] W. Lu and L. V. S. Lakshmanan. Profit maximization over social networks. In *ICDM*, pages 479–488, 2012.

[15] H. Markowitz. Portfolio selection. *The Journal of Finance*, 7(1):77–91, 1952.

[16] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functionsi. *Mathematical Programming*, 14(1):265–294, 1978.

[17] D. Sheldon, B. N. Dilkina, A. N. Elmachtoub, R. Finseth, A. Sabharwal, J. Conrad, C. P. Gomes, D. B. Shmoys, W. Allen, O. Amundsen, and W. Vaughan. Maximizing the spread of cascades using network design. In *UAI*, pages 517–526, 2010.

[18] Y. Tang, X. Xiao, and Y. Shi. Influence maximization: Near-optimal time complexity meets practical efficiency. In *SIGMOD*, 2014.

[19] M. van Leeuwen and A. Ukkonen. Discovering skylines of subgroup sets. In *ECML/PKDD (3)*, pages 272–287, 2013.

Same bang, fewer bucks: efficient discovery of the cost-influence skyline
Supplementary Information

Matthijs van Leeuwen* Antti Ukkonen†

1 Proofs of propositions

Proof. **(of Proposition 4.1)** Let $X_i = X_i^{\text{InfMax}}$ for short. First, observe that $X_1 \in P_1$. This is because the greedy algorithm maximizes marginal gain, and no point on P_1 can dominate this, no matter what the costs are. Next, suppose we have $X_{i-1} \in P_{i-1}$. Since X_i is obtained from X_{i-1} by adding that u that maximizes marginal gain in $I(\cdot)$, no point on P_i can dominate X_i. This is because for some $X' \in P_i$ to dominate X_i, there would have to have been some set in P_{i-1}, such that when some item u' is added to this set, the marginal gain is larger than the one found by greedy when forming X_i. Because for a submodular $I(\cdot)$ the marginal gains are decreasing, such a u' can not exist. If it existed, the greedy algorithm would have added u' at some earlier point. □

Proof. **(of Proposition 4.2)** We provide an example that shows that there can exist some $X \in P_k$ that dominates X_k^{BFTB}. Suppose we have only three vertices, A, B, and C, with the costs and influences as shown in Table 1, and let $k = 2$. Observe first that the influence function is indeed submodular, and that the costs are linear. The greedy BFTB algorithm would first choose B, and then the set BC in the second iteration, as these maximize the bang-for-the-buck ratio. However, the levelwise skyline algorithm would first find that $P_1 = \{A, B, C\}$, and then extend this to $P_2 = \{AB, AC\}$. However, the set AC clearly dominates the set BC as it has both lower cost and higher influence.

Table 1: Example costs and influences for Proof of proposition 4.2

set	Cost	Influence
A	1	2
B	2	5
C	3	6.5
AB	3	5.5
AC	4	8.33
BC	5	8

2 Additional speedups to FAST-SKYLINE

Below we discuss a number of implementation details that are relatively simple but nonetheless useful in practice.

2.1 Implementing the marginal gain cache In our implementation the marginal gain cache is a collection of lists indexed by vertex id. For every $u \in V$ we have a list R_u of triplets of marginal gains of u for different subsets Y. We scan over R_u until we find a case where the resulting upper bound $\Gamma(Y, u) + I(X)$ is less than $I_{\max}$, as this is enough to prune $X_j \cup u$ from consideration. If no such case is found, we return the smallest valid $\Gamma(Y, u)$ found in R_u.

2.2 Managing the priority queue The QUEUE-PUSH procedure implements an additional optimization that avoids inserting candidates into Q that we can prune already at that point. That is, we keep incrementing pos_j until we either reach the end of U, or we find a case that has a upper bound above the current $I_{\max}$. Notice that when the candidate is popped from Q, the value of $I_{\max}$ might have increased, and we have to check the upper bound again. This leads to a substantial speedup in practice as we can prune many of the points in $\mathcal{E}(P_{i-1})$ from entering the queue in the first place.

2.3 More speedups We mention some further implementation details that help making the algorithm run faster in practice.

1. The algorithm will generate duplicate sets. To avoid these, we conduct an additional check after line 10 of the main algorithm to find out if the resulting set $X_j \cup u$ has already been processed. In practice we do this by checking if the freshly popped set is equal to the set that was processed immediately before.

*Machine Learning, Department of Computer Science, KU Leuven,Leuven, Belgium matthijs.vanleeuwen@cs.kuleuven.be.

†Helsinki Institute for Information Technology HIIT, Aalto University, Finland.

Algorithm 1 Details of subroutines used in FAST-SKYLINE

procedure MGC-GET(u, X):
  $R \leftarrow M_u$
  for $i = 1$ to $|R|$ **do**
    $(u, Y, \Gamma(Y,u)) \leftarrow R_i$
    if $Y \subset X$ and $\Gamma(Y,u) + I(X) < I_{\max}$ **then**
      return $\Gamma(Y,u)$
  return smallest $\Gamma(Y,u)$ found in R_u with $Y \subset X$

procedure QUEUE-PUSH(j):
  repeat
    $\text{pos}_j \leftarrow \text{pos}_j + 1$
  until $\text{pos}_j = |U|$ or MGC-GET(U_{pos_j}, X_j) $+ I(X_j) > I_{\max}$
  if $\text{pos}_j < |U|$ **then**
    insert (j, pos_j) into Q with priority $C(X_j \cup U_{\text{pos}_j})$

2. In MGC-GET, often an item can be pruned without scanning over R_j, but simply by checking if $\Gamma(\emptyset, u) + I(X) < I_{\max}$. That is, for many vertices even the marginal gain that results from adding u to the empty set is enough to prune u when $I_{\max}$ has grown sufficiently large.

3 Effect of the filtering heuristic

We make a note about the effect the filtering heuristic has on the running time, as well as on the resulting skyline. Prior to calling FAST-SKYLINE given P_{i-1}, we prune P_{i-1} by discarding points as described in the main manuscript. We set the parameter $\theta = 1$, meaning the resulting skylines will have a resolution of 1 unit of cost. As the reason for applying the filter is to reduce redundancy as the size of the skyline grows, we only apply the filter when the size of P_{i-1} is above some threshold. In the following experiment we used the thresholds 100 and 200 (denoted by f100 and f200 below); in all other experiments we used 200 as default threshold.

Figure 1 shows the unfiltered skyline, as well as the filtered ones for *soc-Epinions1* and *ca-CondMat*. The plots on the left show the entire skyline, while on the right we zoom into a small region indicated by the dashed rectangle in the left figures. We can observe that the curves for f100 and f200 overlap almost perfectly with the solid black line (no filtering). The small differences that are present appear in the low-cost parts of the skylines, as indicated by the detail plots on the right. The quality of the resulting skylines is hardly affected by applying filtering, and unlikely to make a noticeable difference in practice.

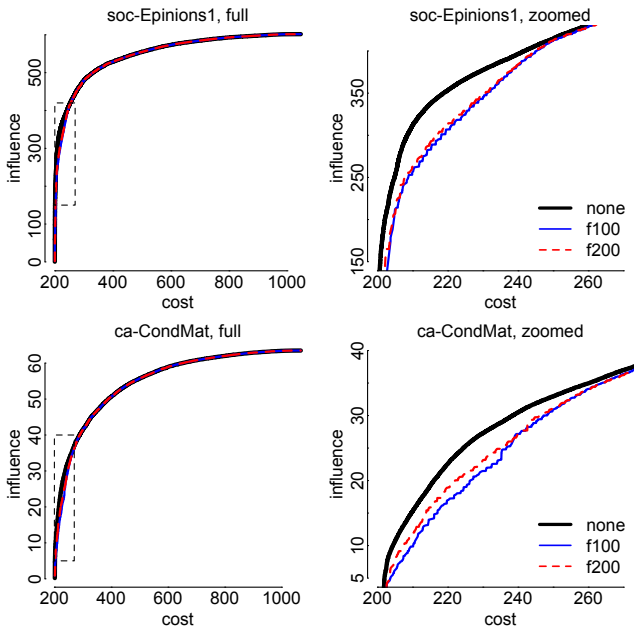

Figure 1: Effect of filtering the skyline P_{i-1} prior to calling FAST-SKYLINE. Black curve shows the unfiltered skyline, while the other lines show filtered ones. Figures on the left show the entire skyline, while figures on the right zoom in on the region indicated by the dashed rectangle.

Table 2: Effect of skyline filtering (with $k = 40$)

Network	filter	time (sec)	P_k
ca-HepTh	none	105	2,612
	f100	29	1,191
	f200	29	1,218
ca-CondMat	none	612	4,795
	f100	69	1,517
	f200	71	1,573
soc-Epinions1	none	7,684	11,128
	f100	1,640	1,648
	f200	1,655	1,654

Table 2 shows both running time (with $k = 40$) and the size of P_{40} for some of our smaller datasets, either without filtering or with one of the two thresholds are applied.[1] We can observe that filtering leads to a substantial reduction in running time, no matter what the number of vertices is. The resulting skylines are obviously substantially smaller.

Note that with filtering, it cannot be guaranteed that the discovered skyline contains exactly the solution found by the greedy InfMax algorithm. In practice, however, the skyline contains at least solutions that are virtually identical in terms of both cost and influence even with (modest) filtering.

[1] The values are computed with a different random allocation of vertex costs, hence the small differences to Table 1 in the article itself.