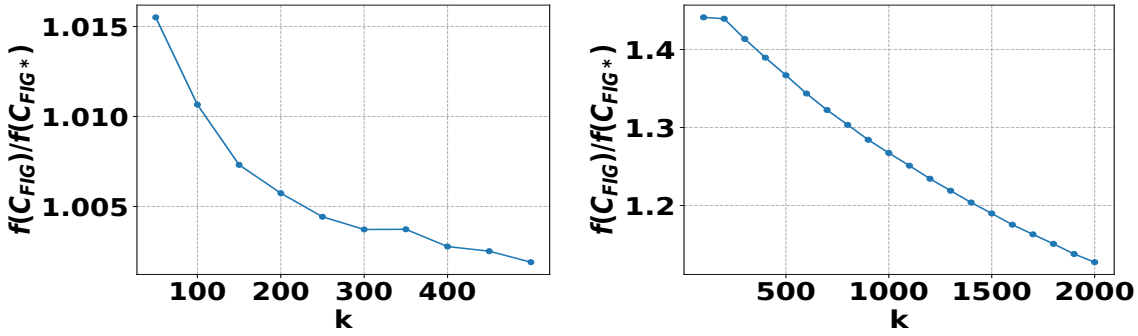

(a) ER instance, $n = 1000$ (b) BA instance, $n = 10000$

Figure 1: Ablation study, effect of stealing behavior.

We thank all reviewers for the comprehensive feedback. In the following, we address specific concerns of each reviewer individually. In addition to the below items, the source code link and other minor issues pointed out by the reviewers will be fixed in the next version.

Reviewer 1: There are several related works which are missing from this manuscript: ... Thank you for bringing these works to our attention. We will add discussion of these works to the manuscript.

[The algorithms of] [FMZ2019], Feldman et al. (2017), [FKK2019] could be beneficial to be compared with the algorithms presented in this paper. We agree that more empirical comparisons, especially with [FMZ2019], would be interesting. If we had been aware of this work earlier, it may have provided a better contrast than BLITS, since it has low query complexity as well as adaptivity.

Reviewer 2: In the proof of Lemma 1, how is the last term in the second to last equation upper bounded by δM? All usage of ε in the pseudocode of FIG, ADD should be δ, which then yields the upper bound. Thank you for catching this typo.

For the BLITS algorithm, why do the number of queries to f consistently decrease for larger k? BLITS works by guessing successively smaller values for OPT. With larger k, it is able to terminate earlier as the value of any solution it obtains is a lower bound on OPT.

How much of the performance of FIG is due to the stealing trick (Appendix C)? Is the proposed algorithm still competitive with Gupta et al. without the stealing trick? In Fig. 1 above, we show the effect of removing the stealing procedure on the random graph instances evaluated in the manuscript. Let C_{FIG} be the solution returned by FIG, and C_{FIG*} be the solution returned by FIG with the stealing procedure removed. Fig. 1(a) shows that on the ER instance, the stealing procedure adds at most 1.5% to the solution value; however, on the BA instance, Fig. 1(b) shows that the stealing procedure contributes up to 45% increase in solution value, although this effect degrades with larger k. This behavior may be explained by the interlaced greedy process being forced to leave good elements out of its solution, which are then recovered during the stealing procedure.

How does the algorithm perform on monotone submodular functions? If the submodular function is restricted to be monotone, IG obtains an approximation ratio of at least $1/2$. A factor of 2 is saved from the analysis for general submodular, since $f(O \cup A) + f(O \cup B) \geq 2f(O)$ in the monotone case. It is possible a better analysis could show a ratio of better than $1/2$. We did not evaluate empirically on monotone submodular functions.

Reviewer 3: Are there instances where the interlacing greedy or the fast variant achieve an approximation ratio of exactly 1/4? In other words, is 1/4 the best approximation to hope for with this technique? Yes, the ratio $1/4$ is tight for non-monotone submodular functions. Tight examples will be added to the manuscript showing, for each $\epsilon > 0$, an instance A_ϵ where InterlaceGreedy has ratio less than $1/4 + \epsilon$ on A_ϵ.

The proofs of the algorithms are terse in a few places. ... We agree, and we will add the details indicated, the equivalent definition of submodularity, and highlight the places where non-negativity is used.

Can you comment on why might the run time be slowly increasing with k? In FIG, we terminate a greedy subroutine when the marginal gain falls below $\delta M/n$, which leads to the $\log n$ term in the query complexity. However, it is sufficient in the proof to terminate when the gain is below $\delta M/k$, which would lead to a $\log k$ term in the query complexity. We will update the algorithm and discussion to reflect this fact.