A Details on the MaxKCD algorithm

In this section, we expand on the details of the proposed MaxKCD algorithm for identifying k-edge-connected components, i.e. for Problem 3. Aside from connections to completability, k-connected components have been recognized as an important structure in applications ranging from behavior mining, social network analysis, e-commerce, biology, and more [26]. For a fixed small k (e.g. $k = 1, 2, 3$) and a graph $G = (V, E)$, all maximal k-CCs can be found in $O(k(|V| + |E|))$ time by algorithms based on depth-first-search [21, 28]. For a general k, the global min-cut routine can be iteratively applied to remove cuts smaller than k; however, the worst case run-time for such a vanilla algorithm is $O(|V|T)$, where T is the run-time of the global min cut algorithm, e.g., $T = O(|V|(|E| + |V| \log |V|))$ for Stoer-Wagner algorithm [25]. Heuristic improvements have been proposed, for example in [33].

Several groups have proposed modifying the black box min-cut algorithms [1, 7, 26]. While the worst-case performance is generally unchanged, the empirical runtime decreases significantly. The improvement stems from the fact that an algorithm for finding k-CCs relies largely on finding a cut of size specifically less than k, and that this subroutine can be implemented more efficiently than the black-box min cut algorithms. In [7, 26], the authors proposed an algorithm based on the Stoer-Wagner global min-cut algorithm incorporating the knowledge of k for a gain in efficiency. Instead, Akiba et al. in [1] proposed a randomized approach based on Karger's algorithm; however, it has been shown to be inefficient compared to Stoer-Wagner-based algorithms.

The algorithm we propose, which is called MaxKCD, works by iteratively finding a cut of size less than k in G until such a cut does not exist, and only k-CCs or singleton vertices remain; the pseudocode is shown in Algorithm 2. At step 5, the algorithm applies a subroutine called kCut to find a cut of size less than k. The subroutine is based on the Stoer-Wagner algorithm [25] and improves upon the works in [26, 7]. Using the returned cut, the graph is split at step 9 of MaxKCD and each part is pushed for the next iteration.

In MaxKCD, the main kCut subroutine utilizes three core operations: EarlyStop, ForceContraction, and Batch-EarlyMerge. First, we observe that the core component, *maximal adjacency search* (MAS), can be accelerated by a batch-like approach, Batch-EarlyMerge, which allows kCut to be more efficient while retaining accuracy for the the K-Connected Component Decompositon Problem. We also incorporate a vertex-merging technique, called Force-Contraction, which results in a more aggressive merging technique than those in [26, 1]. We also incorporated the EarlyStop routine proposed in [7, 26]. As a side note, the kCut subroutine can be used in turn to speed up the original Stoer-Wagner algorithm by actively updating the k, starting from infinity, as the current smallest cut. In what follows we dive into the details of the kCut algorithm proposed in this work.

Algorithm 2: Maximal k-CC Decomposition (MaxKCD)

```
Input : Graph $G = (V, E)$ and the target cut size $k$
Output : The vertex partition $\Phi$
1  Initialize $\Phi = \emptyset$ and $\Gamma_0$ to be the whole graph $G$
2  while $\Gamma_i \neq \emptyset$ do
3      Initialize $\Gamma_{i+1} = \emptyset$
4      forall $G' = (V', E') \in \Gamma_i$ do
5          Find a cut $C = (V_1', V_2') = \texttt{kCut}(G', k)$
6          if $C = \emptyset$ or $|V'| = 1$ then
7              Add $V'$ to $\Phi$
8          else
9              Add $G'_{V_1}$ and $G'_{V_2}$ to $\Gamma_{i+1}$
10         end
11     end
12 end
13 return $\Phi$
```

A.1 The kCut algorithm

The kCut algorithm is based on the Stoer-Wagner algorithm for finding a global min-cut. Shown in Algorithm 3, kCut is composed of two parts: the kMAS algorithm in step 4, and the Merge routine in step 5. The former obtains information on the (s, t)-min-cut via graph traversal, and the later utilizes the target value k to leave a strictly smaller graph for the next step.

The kMAS routine is based on a core procedure in Stoer-Wagner: the MAS algorithm for finding an (s, t)-min-cut in G. Unlike other (s, t)-min-cut algorithms, the vertices s and t are not received as input. Instead, MAS visits vertices in a particular order, and outputs the size of the (s, t)-min-cut between the two vertices visited last. The order is dictated by connectivity to the already-visited set, and we refer the reader to [25] for more details. The key property of MAS is that at any iteration, MAS records the (s, t)-min-cut between two vertices most recently visited on the subgraph induced by all visited vertices. More formally, let $L_i = (u_1, \ldots, u_i)$ be the ordered set of already visited vertices at iteration i and u_{i+1} be the vertex to be visited next. Then we have that $w(L_i, u_{i+1})$ is the (s, t)-min-cut between u_i and u_{i+1} on the induced subgraph $G_{L_{i+1}}$. Note that $w(L_i, u_{i+1})$ is the sum of the edges weights

Algorithm 3: Finding a cut with size less than k

```
1  $C$ =kCut($G, k$)
   Input  : $G = (V, E)$ and the target cut size $k$
   Output : Cut $C$
2  Initialize $G_1 = (V_1, E_1)$ to $G$, and $i = 1$
3  while $|V_i| > 1$ where $G_i = (V_i, E_i)$ do
4      $P_i$ =kMAS($G_i, v_j, k$), for arbitrary $v_j \in V_i$
5      $G_{i+1}$ =Merge($G_i, P_i, k$)
6      $i = i + 1$
7  end
8  return $C = \emptyset$
```

between L_i and u_{i+1} and $G_{L_n} = G$. For the propose of kCut, s and t can be safely merged if the (s,t)-min-cut on G is at least k. This will not destroy any cut of size less than k since such a cut would place these two vertices on the same side. The EarlyMerge operation proposed in [26] monitors $w\left(L_i, u_{i+1}\right)$, which is a lower bound on the (s,t)-min-cut, and merges u_i and u_{i+1} as soon as it reaches k.

The key to modifying MAS for the K-Connected Component Decompositon Problem and later improving on efficiency is based on two main observations. First, the existence of a target rank k implies that we do not care to differentiate between cuts of size larger than k in Algorithm 2. This provide the extra room for us to modify the MAS procedure as we do not require the (s,t)-min-cut lower bound to as accurate. As a result, a batch-like approach Batch-EarlyMerge can be utilized in the kMAS procedure to speed up the process. Second, we observe that a separation between the visitation and the merging allows for incorporation of more aggressive merging techniques, known as ForceContraction, resulting in a more efficient algorithm. These two observations are captured in three core operations, EarlyStop, Batch-EarlyMerge, and ForceContraction, which we describe below.

Early Stopping: The first operation, EarlyStop, utilizes the information of the target cut size k to terminate the visitation procedure of kMAS and directly return the solution for kCut. If at any point the cut between the visited set of vertices L_i and the unvisited set $V \setminus L_i$ is smaller than k, we can terminate kMAS and return the cut $\{L_i, V \setminus L_i\}$ directly.

Batch Early-Merging: The original MAS routine visits one vertex at a time; however, we observe that this one-by-one visitation is too strict for the K-Connected Component Decompositon Problem since we care only whether the cut is smaller than k. We introduce the Batch-EarlyMerge operation to work on multiple vertices at a time, and prove that it retains correctness with respect to the K-Connected Component Decompositon Problem.

Algorithm 4: Accelerated MAS for k-CCs

```
1  $P$ =kMAS($G, u_1, k$)
   Input  : $G = (V, E)$, start vertex $u_1$, target cut size $k$
   Output : Sets of vertices $P$ for Batch-EarlyMerge
2  Initialize $i = 1$, $P = \emptyset$, and $L_1 = \{u_1\}$
3  while $i < |V|$ do
4      if $w(L_i, V \setminus L_i) < k$ then
5          Apply EarlyStop
6      else
7          if $\max_{v \in V \setminus L_i} \{w(L_i, v)\} \geq k$ then
8              Batch-EarlyMerge:
9              $U_{i+1} = \{v | w(L_i, v) \geq k, v \in \{V \setminus L_i\}\}$
10             $L_{i+|U_{i+1}|} = L_i \cup U_{i+1}$
11             $P = P \cup \{U_{i+1} \cup \{u_i\}\}$
12             $u_{|L_i|}$ = any vertex from $U_{i+1}$
13         else
14             $u_{i+1} = \arg\max_{v \in V \setminus L_i} \{w(L_i, v)\}$
15             $L_{i+1} = L_i \cup \{u_{i+1}\}$
16         end
17         $i = |L_i|$
18     end
19 end
20 return $P$
```

Recall that MAS merges two vertices $u_i \in L_i$ and $u_{i+1} \notin L_i$ at step i if $w\left(L_i, u_{i+1}\right) \geq k$. The Batch-EarlyMerge operation works by merging all v with $w\left(L_i, v\right) \geq k$, as opposed to only the u_{i+1} that is scheduled to be visited next; more precisely the set $\{v | w\left(L_i, v\right) \geq k\}$ is merged together with u_i. The main idea of the proof of correctness (available in the extended version) is that although kMAS does not follow the order specified by MAS, no cut of size at least k will be omitted and no cut of size less than k will be affected: The resulting graph after Merge operation will be the same as that with EarlyMerge. The computational benefit of Batch-EarlyMerge is that it reduces the number of IncreaseKey operations for the *Max Heap* used in MAS, which is the computational bottleneck of the MAS algorithm for dense graph.

Force Contraction: The ForceContraction operation was first introduced in [1]. The operation merges any vertices u and v whose edge weight $w\left(u, v\right)$ exceeds k. The operation has been omitted from previous exact algorithms because applying it to nodes outside of the visited set L_i may lead to a violation of the MAS property. However, by separating the visitation and merging into kMAS and Merge, the kCut algorithm we propose allows ForceContraction to be utilized. The combination of ForceContraction with the other two operations further reduces the size of the graph, lead-

Algorithm 5: Vertex merging with ForceContraction

```
1 $G'$ =Merge$(G, P, k)$
  Input  : $G = (V, E)$, set of pairs $P$, target cut size $k$
  Output : Modified graph $G'$
2 Initialize $G' = G$
3 forall $U \in P$ do
4     Batch-EarlyMerge: Merge all vertices in $U$ on $G'$
5 end
6 while $\exists$ edge with $w(v_1, v_2) \geq k$ in $G'$ do
7     ForceContraction: Merge $v_1$ and $v_2$ on $G'$
8 end
9 return $G'$
```

ing to a faster convergence of the recursion. Moreover, since the edges within a super-vertex can be ignored during vertex merging and `ForceContraction` creates larger *super-vertex*, the cost of the merging operation is also often reduced considerably. For example, the cost was reduced by 30% on the Amazon review data.

The correctness of Algorithm 3 largely follows that of the `MAS` procedure. Additional optimization of `kCut` are discussed in the extended version, e.g., remove vertices with degree less than k.

Computational Complexity: Since the `kCut` algorithm can be applied to solve the global min-cut problem with binary search, the worst-case computational complexity of `kCut` algorithm would be similar to that of the global min-cut algorithm, differed by a $\log|V|$ factor at most. In fact, the worst-case computational complexity of `kCut` and its analysis is the same as the Stoer-Wagner algorithm: $O\left(|V||E| + |V|^2 \log|V|\right)$.

However, unlike the Stoer-Wagner algorithm, the empirical runtime of the `kCut` algorithm is much less than its worst-case. If the input graph is sparse and loosely connected, like those in the matrix completion setting, `EarlyStop` will likely terminate the algorithm in the first few calls. If the input graph is tightly connected, `Batch-EarlyMerge` ensures that each `kMAS` can be conducted efficiently, and together with `ForceContraction`, the graph size shrinks quickly. In our experiments, less than 10 calls to `kMAS` are observed even in graph with million of vertices and billions of edges, where the empirical runtime scales with $|E|\log|V|$ due to usage of binary heaps.

B Proof of correctness for Batch-EarlyMerging

Unlike previous algorithms for Problem 3 which required the exact `MAS` procedure, we propose a brand-new accelerated `MAS` routine, where multiple vertices can be visited in a single step. This *batch* approach reduces the time-consuming operation of updating the max-heap for the traditional `MAS`, leading to potentially significant speed-up. The key observation is that we are not restricted to the order of visitation specified by `MAS` since we do not need to distinguish cuts of size greater or equal to k.

Let $u_1 \to u_2 \to \cdots \to u_n$ be the order of visitation by `MAS` as stated in Algorithm 6, and $L_i = \{u_1, \ldots, u_i\}$. Then, at the i-th step, we can list the remaining vertices $v \in V \setminus L_i$ by $w(L_i, v)$ in decreasing order: $v_{i+1} \to v_{i+2} \to \cdots \to v_n$; note that this order is not necessarily the same as $u_{i+1} \to \cdots \to u_n$ except that $v_{i+1} = u_{i+1}$. The batch approach can be applied for the maximum possible $s \geq i$, if it exists, such that

$$w(L_i, v_r) \geq k, \quad \forall r, \quad i < r \leq s.$$

That is, the set $U_{i+1} = \{v_r | i < r \leq s\}$, can be visited all at once as follows: first merge the vertices inside U_{i+1} to $u_{i+1} = v_{i+1}$, then add u_{i+1} to L_i by merging u_{i+1} with u_i. Take $L_s = L_i \cup U_{i+1}$ and continue the visit.

Though the final order of visitation may differ from that of `MAS`, we will prove that no cut of size less than k will be destroyed, hence end result is the same. The computational benefit over the original `MAS` procedure comes from the reduced number of `IncreaseKey` operations in heap maintenance for all the edges inside U_{i+1}, which is the dominant (first) factor in the $O\left(|E| + V\log|V|\right)$ cost of `MAS`. The improvement is especially significant when `MAS` is implemented with *binary heap*, where the cost would be $O\left(|E|\log|V| + |V|\log|V|\right)$.

Proof of Correctness: To prove the correctness of the *batch* approach, we prove that P =`MAS`(G, s, k) and P^+ =`kMAS`(G, s, k) lead to the same graph after merging, i.e., `Merge`(G, P, k) = `Merge`(G, P^+, k). Here `MAS` denotes the simple `MAS` with `EarlyMerge` with vertices visited one at a time. Since the `ForceContraction` is applied in the last stage of `Merge` and uniquely depends on the results of `EarlyMerge`, we need only consider the results of `EarlyMerge`.

Note that the elements in P =`MAS`(G, s, k) are pairs of vertices that will be merged together. Moreover, elements of P may overlap if they contain the same vertex, which suggests a chain of vertices that will all be merged together. For example, P may contain both $\{u_i, u_{i+1}\}$ and $\{u_{i+1}, u_{i+2}\}$ which means all three vertices will be merged together. Hence, P can be simplified by merging the overlapped elements until all elements in P are non-overlapping; we call these resulting non-overlapping elements *early merging blocks*. More formally:

Definition 4 (Early Merging Block)**.** *In Algorithm 6,*

Algorithm 6: Maximum Adjacency Search for `kCut`

```
1  $P$ =MAS $(G, u_1, k)$;
   Input  : $G = (V, E)$, start vertex $u_1$, target cut size $k$
   Output : Sets of vertices $P$ for Batch-EarlyMerge
2  Initialize $i = 1$, $P = \emptyset$, and $L_1 = \{u_1\}$
3  while $i < |V|$ do
4      $u_{i+1} \leftarrow \arg\max_{u \in V \setminus L_i} \{w(L_i, u)\}$;
5      $w_{i+1} \leftarrow w(L, u_{i+1})$
6      if $w_{i+1} \geq k$ then
7          $P = P \cup \{\{u_{i+1}, u_i\}\}$
8      end
9      $L_{i+1} \leftarrow L_i \cup \{u_{i+1}\}$
10     $i \leftarrow i + 1$
11 end
12 return $P$
```

an early merging block is a subset of vertices $R^k_{i:j}$ that satisfies

$$R^k_{i:j} = \{u_r | i \leq r \leq j, w_i < k, w_{j+1} < k, \text{ and } w_s \geq k, \forall s, i < s \leq j\},$$

where $1 \leq i < j \leq n$, and $w_1 = w_{n+1} = 0$. Note that $|R^k_{i:j}| \geq 2$ by definition.

If two vertices belong to the same *early merging block*, they will be merged together. Further, without `ForceContraction`, if two vertices do not belong to the same block, they will remain distinct vertices after `Merge`. We will prove that P^+, obtained by `Batch-EarlyMerge`, contains the same information as P.

Theorem 5. *Given P =`MAS`(G, s, k) and P^+ =`kMAS`(G, s, k), then `Merge`(G, P, k) = `Merge`(G, P^+, k).*

Proof. As discussed before, the merging result of P can be described by early merging blocks. We will now prove that P^+ yields the same merging result as suggested by the early merging blocks. Let $u_1, u_2, \ldots, u_n$ be the visitation order taken by `MAS`, then at step i we have $L^{\texttt{MAS}}_i = \{u_1, \ldots, u_i\}$, and similarly for `kMAS` we have $L^{\texttt{kMAS}}_i$.

Note that if P^+ is empty, this implies that the condition on step 7 in `kMAS` is never true. If this is the case then there is no early merging block, implying that P is empty as well. Similarly if P^+ is not empty, but all $|U_{i+1}| = 1$, then by definition that $P = P^+$.

When there exists $|U_{i+1}| > 1$, we prove Theorem 5 by induction, starting both `MAS` and `kMAS`, at the first step, and matching P and P^+ step by step. Before the first step, P and P^+ for `MAS` and `kMAS` are all empty, and $L^{\texttt{kMAS}}_0 = L^{\texttt{MAS}}_0 = \emptyset$, so the conclusion holds. In fact, P and P^+ (also $L^{\texttt{kMAS}}_i$ and $L^{\texttt{MAS}}_i$) are equivalent before encountering the first nontrivial ($|U_{i+1}| > 1$) early merging block. Here, we assume that P and P^+ are equivalent up to the p-th iteration. We break the proof into three parts:

1. Let U_{i+1} with $i \geq p$ be the first U_{i+1} with $|U_{i+1}| > 1$, then we know that $L^{\texttt{kMAS}}_i$ is the same as $L^{\texttt{MAS}}_i$. Here we also merge u_i into U_{i+1} corresponding to the step 11 in `kMAS` algorithm. Pick any vertex u_t in U_{i+1}, which is the t-th ($t > i$) visited vertex in `MAS`. By definition in `kMAS`, we have that $w(L^{\texttt{MAS}}_i, u_t) \geq k$. Then for any r such that $i < r \leq t$, we have that:

   $$w_r = w(L^{\texttt{MAS}}_{r-1}, u_r) \geq w(L^{\texttt{MAS}}_{r-1}, u_t) \geq w(L^{\texttt{MAS}}_i, u_t) \geq k,$$

   which implies that there exists $R^k_{i:j}$ such that:

   $$\{u_i, \ldots, u_t\} \subset R^k_{i:j} \Rightarrow u_t \in R^k_{i:j}$$

   Since u_t is picked arbitrarily, same conclusion holds for all elements in U_{i+1}, and there is a unique early merging block that starts from i. Therefore, we have that $U_{i+1} \subset R^k_{i:j}$ meaning that the batch found by `Batch-EarlyMerge` is always a subset of an early merging block.

2. When $R^k_{i:j} = U_{i+1}$, we know that P and P^+ are equivalent up to the j-th iteration, and that $L^{\texttt{kMAS}}_j$ is the same as $L^{\texttt{MAS}}_j$. Therefore, we can replace the results of `kMAS` by that of `MAS` and continue both algorithms by repeating the argument in (1) from the j-th iteration.

3. When $R^k_{i:j} \neq U_{i+1}$, we will prove that the next element $U_{i'+1}$ in P^+ satisfies $U_{i'+1} \subset R^k_{i:j}$ and $U_{i'+1} \cap U_{i+1} \neq \emptyset$. Let $s = \min\{s' | u_{s'} \in R^k_{i:j} \setminus U_{i+1}\}$, then we have that

   $$w(L^{\texttt{kMAS}}_{i'}, u_s) \geq w(L^{\texttt{MAS}}_{s-1}, u_s) \geq k,$$

   because that $L^{\texttt{MAS}}_{s-1} \subset L^{\texttt{kMAS}}_{i'} = L^{\texttt{MAS}}_i \cup U_{i+1}$. Therefore, we know that $u_s \in U_{i'+1}$ and $U_{i'+1}$ was created immediately after U_{i+1}, which implies that $U_{i'+1} \cap U_{i+1} \neq \emptyset$.

   Similarly, for any vertex u_s where $s > j$, we have that

   $$w(L^{\texttt{kMAS}}_{i'}, u_s) \leq w(L^{\texttt{MAS}}_j, u_s) \leq w(L^{\texttt{MAS}}_j, u_{j+1}) < k.$$

   Therefore, we have that $U_{i'+1} \subset R^k_{i:j}$. If $U_{i'+1} \cup U_{i+1} = R^k_{i:j}$, we can repeat the argument in (1) from the j-th iteration as in (2). If $U_{i'+1} \cup U_{i+1} \neq R^k_{i:j}$, we assign $U_{i+1} = U_{i'+1} \cup U_{i+1}$ and repeat the argument in (3), and reach (2) in a finite number of steps since $|U_{i'+1} \cup U_{i+1}| > |U_{i+1}|$.

□

		lmafit		**optspace**		**reimann**		**vbmc**		**grouse**	
	k_{MC}	CPLT	overall	CPLT	overall	CPLT	overall	CPLT	overall	CPLT	overall
Bibsonomy	12	0.403	1.257	n/a	n/a	0.590	1.377	0.760	1.061	n/a	n/a
Movielens	40	0.009	0.291	0.074	0.36	0.005	0.287	0.005	0.335	3.45	37.9

Table 4: Comparison of relative frobenius error over completable entries using different matrix completion algorithms.

B.1 Notes on implementation

We have discussed the backbone of the `MaxKCD` algorithm; however, There are a few optimization tricks that can be exploited for the implementation of the `MaxKCD` algorithm which we provide in C++.

k-Core Optimization: Since a k-CC requires each vertex to have degree k, a scan of vertex degrees can remove unnecessary computation. Before searching for the cut with size less than k, the `kCut` algorithm can first check for any vertex whose weighted vertex degrees is less than k, remove it, and add it directly to the final partition P. Such a scan is called a k-`coreOpt` routine, as it decompose the graph into k-Core components.

In-place storage of Γ: In Algorithm [1], each Γ_i can be stored on the original graph, since it corresponds to a list of connected components of G at iteration i. A key and simple observation is that all of the algorithms presented here rely only on the local information of which vertices have been visited and their respective neighbors. Hence, the only information that needs to be stored about each subgraph is its size and a seed vertex.

C Additional experimental results

C.1 Empirical runtime performance for `MaxKCD`

The overall running time of the `MaxKCD` algorithm is a small fraction of the running time of the common matrix completion algorithms. Therefore, the utility of our framework is not limited by the computational efficiency of `MaxKCD`. However, the empirical runtime is of interest since `MaxKCD` can be of independent interest for graph maximal k-connected components decomposition and global minimum cut algorithms.

With our aggressive graph contraction schemes, including both `Batch-EarlyMerge` and `ForceContraction`, the overall running time is all less than 6 minutes on both Amazon and Netflix datasets with $k \geq 10$. We have also recorded the running time of both `kMAS` and `Merge`. Comparing with the version without `ForceContraction` and `EarlyMerge` instead of `Batch-EarlyMerge`, we reduce the computational cost for `Merge` on both Amazon and Netflix data sets by 30% (from 103s to 70s) and 7% (from 59s to 55s), respectively. The source of the gain is mainly from `ForceContraction`, where merging many vertices at once avoids unnecessary edge upkeep.

On sparse data the improvement in the running time for `kMAS` is less dramatic, but on synthetic Erdős-Rényi graph with $60,000$ vertices and $44,995,961$ edges, we observed 3 times acceleration (from 3.7s to 1.2s).

We note that further optimizations can continue to improve the computational performance, but this was not the focus of this work.

C.2 Comparison between matrix completion algorithm

For deeper insight into the implications of our results, we considered how much of our findings are dependent on the particular matrix completion algorithm. In particular, we selected five algorithms with the most competitive efficiency and accuracy: `LMaFit`, `OptSpace`, `LRGeom`, `Riemann`, and `VBMC`. (The interested reader can refer to [34] for a comparison.) We found that using all algorithms, the error over the entries selected by `CompleteID` as completable is significantly smaller than over the non-completable ones. The results in Table [4] demonstrate the robustness of `CompleteID` and that it provides meaningful insight for the task of low-rank matrix completion in general.