Maximum k-Plex Finding: Choices of Pruning Techniques Matter!

Akhlaque Ahmad
Indiana University Bloomington
akahmad@iu.edu

Da Yan
Indiana University Bloomington
yanda@iu.edu

Xiao Chen
Indiana University Bloomington
xc56@iu.edu

Lyuheng Yuan
Indiana University Bloomington
lyyuan@iu.edu

Qin Zhang
Indiana University Bloomington
qzhangcs@iu.edu

Saugat Adhikari
Indiana University Bloomington
adhiksa@iu.edu

ABSTRACT

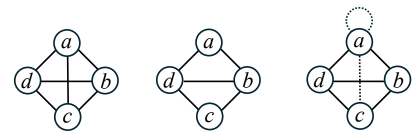
A k-plex is a dense subgraph structure where every vertex can be disconnected with at most k vertices. Finding a maximum k-plex (MkP) in a big graph is a key primitive in many real applications such as community detection and biological network analysis. A lot of MkP algorithms have been actively proposed in recent years in top AI and DB conferences, featuring a broad range of sophisticated pruning techniques. In this paper, we study the various pruning techniques from nine recent MkP algorithms including kPlexT, Maple, Seesaw, DiseMKP, kPlexS, KpLeX, Maplex, BnB and BS by unifying them in a common framework called U-MkP. We summarize their proposed techniques into three categories, those for (1) branching, (2) upper bounding, and (3) reduction during subgraph exploration. We find that different pruning techniques can have drastically different performance impacts, but there exists a configuration of the techniques dependent on k that leads to the best performance in vast majority of the time. Interestingly, extensive experiments with our unified framework reveal that some techniques are not effective as claimed in the original works, and we also discover an unmentioned technique that is actually the major performance booster when $k > 5$. We also study problem variants such as finding all the MkPs and finding the densest MkP (i.e., with the most edges) to cover community diversity, and effective algorithm parallelization. Our source code is released at https://github.com/akhlaqueak/MKP-Study.

PVLDB Reference Format:
Akhlaque Ahmad, Da Yan, Xiao Chen, Lyuheng Yuan, Qin Zhang, and Saugat Adhikari. Maximum k-Plex Finding: Choices of Pruning Techniques Matter!. PVLDB, 14(9): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/akhlaqueak/MKP-Study.

1 INTRODUCTION

Finding cohesive subgraphs in a large graph is useful in various applications, such as finding protein complexes or biologically relevant functional groups [28, 33, 43, 52] and social communities [42, 48].



(a) A Clique (b) A 2-Plex (c) Missing Edges of a

Figure 1: Examples of a Clique and a 2-Plex

One classic notion of cohesive subgraph is *clique* which requires every pair of distinct vertices to be connected by an edge (see Figure 1(a)). However, in real graphs, communities rarely appear in the form of cliques due to various reasons such as the existence of data noise [37, 38, 49, 54]. As a relaxed clique model, k-plex was first introduced in [51], which is a graph where every vertex v is adjacent to all but at most k vertices (including v itself). Figure 1(b) shows a 2-plex where every vertex is not adjacent to at most $k = 2$ vertices. For example, a is not connected to $\{a, c\}$ (see dotted edges in Figure 1(c)). It has found extensive applications in the analysis of social networks [51], especially in the community detection [38, 49]. However, mining k-plex structures is NP-hard [29, 47], so existing algorithms rely on branch-and-bound search which runs in exponential time, but utilize effective pruning techniques to make the search process tractable on medium-sized graphs.

Let us focus on the problem of finding a maximum k-plex (MkP), i.e., when there are ties, an algorithm only needs to return one of the maximum k-plexes (other variants will be studied in Section 6). Surprisingly, in recent years, there is a surge of algorithms with new pruning techniques proposed by the AI and DB communities to significantly speed up MkP computation. However, they share a lot of common pruning techniques, many of which are just reinventing the wheels. Moreover, some new techniques may not improve performance at all, or only improve performance for small k values but can lead to catastrophic performance when k is large. However, these issues were not explicitly reported by the respective papers. Even worse, there is a work [53] whose implementation is totally different from what was proposed in its paper, and its performance gain mainly comes from an unmentioned technique. Without a thorough experimental study of these techniques under a unified framework, claims in some of these papers can mislead users to adopt the pruning techniques that result in performance pitfalls.

This paper provides a timely (and in-depth) summary and experimental study of the various techniques proposed by the recent MkP algorithms, by placing them into a unified algorithmic framework. We categorize the pruning techniques applied during subgraph exploration into three categories, those for (1) branching, (2) upper

Proceedings of the VLDB Endowment, Vol. 14, No. 9 ISSN 2150-8097.
doi:XX.XX/XXX.XX

bounding, and (3) reduction. To be self-contained, we provide the proofs of all techniques in our full technical report [27] with intuitive diagrams and consistent notations, so that the main paper can focus on providing intuitions about the idea behind these techniques. The goal is to provide a benchmark of MkP with which future works can avoid reinventing the wheel and focus on what are really new, and to serve as a comprehensive testbed of the pruning techniques on their performance impacts. We also provide algorithms for variants of MkP that find all the MkPs or the MkP with the most edges to cover community diversity, and parallel versions of all our algorithms.

The insightful experimental findings are summarized as follows:

(1) Different pruning techniques can have drastically different performance impacts (e.g., by thousands of times), but there exists a configuration of the techniques (dependent on k only) that leads to the best performance in vast majority of the time.
(2) The AI community actively designs the upper-bound-based techniques to prune an entire branch of unpromising subgraph search space, but those techniques are only useful for branching instead [44] when k is small. Moreover, a sophisticated strategy such as the one by Seesaw [62] is not more beneficial than a simple one. Even worse, branching in this way can backfire when k is large (not tested and reported in [44]).
(3) When k is large, an unmentioned pivot-based branching method works the best (up to thousands of times faster than upper-bound-based branching), which we extract from the code of Maple [53]. In fact, we find that [53] describes a totally different algorithm which is not actually implemented.
(4) The DB community focuses on designing reduction rules to reduce the size of candidate sets for subgraph expansion during exploration. The latest algorithm kPlexT [35] proposes a new branching method to improve worst-case time complexity, but empirically we find it not competitive to those proposed by the AI community. In contrast, kPlexT manages to find a new reduction rule that continues to significantly improve the search performance. Its prior version kPlexS [34] advocates an incremental reduction technique called CTCP; but we find that CTCP is only worthwhile at the top-level subgraph exploration, but it backfires if it is further applied with the lower-level branches due to the incurred overheads.

The main contributions of this paper are summarized as follows:

(1) To enable the discovery of the 4 findings above, we summarize the pruning techniques of nine state-of-the-art MkP algorithms into three categories, and place them into a carefully-designed unified algorithmic framework called Unified MkP (abbr. U-MkP) to facilitate the flexible configuration of techniques. We utilize U-MkP for extensive experimental studies.
(2) Through the experimental studies, we obtain the above 4 findings which clearly show what techniques work and what do not, which the existing papers fail to reveal. We also identify a configuration of the techniques (dependent on k only) that leads to the best performance in vast majority of the time, and recommend concrete alternatives to try in the rare cases when this configuration is slow (which are difficulty to forecast).
(3) We formalize an unmentioned pivot-based branching method that is the key to the performance of MkP when $k > 5$. We also generalize [44]'s partition-based branching method (using S-based upper bounding) to work with R- and SR-based upper bounding, and it supports additional branch pruning.
(4) To ensure efficiency, we design efficient container structures such as dual-array and auxiliary buffers that are preallocated and incrementally reused/updated during the recursive subgraph exploration to maintain the necessary vertex sets.
(5) We provide algorithms for variants of MkP that find all MkPs or the MkP with the most edges, to avoid missing important dense communities due to returning only one MkP.
(6) We parallelize U-MkP using a task-based approach with time-out mechanism for load balancing to scale up almost ideally.

In the sequel, Section 2 introduces our notations and the branch-and-bound framework adopted by MkP algorithms for subgraph exploration. Then, Section 3 overviews our U-MkP framework and introduces the types of pruning techniques focusing on upper bounding ones. Subsequently, Section 4 summarizes the various branching methods, and Section 5 summarizes the various reduction methods. We discuss the MkP variants and parallelization in Section 6. Finally, Section 7 reports our comprehensive experiments, Section 8 reviews the related works, and Section 9 concludes this paper.

2 PRELIMINARIES

Notations. We consider an undirected and unweighted simple graph $G = (V, E)$, where V is the vertex set, and E is edge set. The degree of a vertex v is denoted by $d_G(v) = |N_G(v)|$. We also define the concept of *non-neighbor*: a vertex u is a non-neighbor of v in G if $(u, v) \notin E$. Accordingly, the set of non-neighbors of v is denoted by $\overline{N_G}(v) = V - N_G(v)$, and we denote its cardinality by $\overline{d_G}(v) = |\overline{N_G}(v)|$. Given a vertex subset $S \subseteq V$, we denote by $G[S] = (S, E[S])$ the subgraph of G induced by S, where $E[S] = \{(u, v) \in E \mid u, v \in S\}$. We simplify the notation $N_{G[S]}(v)$ to $N_S(v)$, and define other notations such as $N_S(v)$, $\overline{N_S}(v)$ and $d_S(v)$ in a similar manner. For an arbitrary graph g, $V(g)$ and $E(g)$ denote the vertex set and edge set of g, respectively. The diameter of G, denoted by $\Delta(G)$ is the shortest-path distance of the farthest pair of vertices in G, measured by the number of hops.

Problem Definition. We next define the concept of k-plex and MkP.

Definition 2.1. (k-*Plex*) A graph g is a k-plex if every vertex $v \in V(g)$ has at least $|V(g)|-k$ neighbors in g, i.e., $d_g(v) \geq |V(g)|-k$. Equivalently, g is a k-plex if every vertex $v \in V(g)$ has at most k non-neighbors in g (including v itself as a non-neighbor), i.e., $\overline{d_g}(v) \leq k$.

Definition 2.2. (*Maximum* k-*Plex Finding*) Given a graph G, the maximum k-plex finding problem finds a largest vertex set $P \subseteq V$ such that the subgraph $G[P]$ induced by P is a k-plex.

The above MkP finding problem only finds one of the potentially many maximum k-plexes (i.e., MkPs) in G, but all of them could be interesting since they may correspond to different (and even non-overlapping) communities in a social network. Moreover, even all MkPs are ties in terms of vertex number, some may have more edges than others and it would be interesting to find the densest one among them. We, therefore, also consider two problem variants below:

Definition 2.3. (*Finding All MkPs*) Given a graph G, the problem finds all largest vertex sets $P \subseteq V$ such that subgraph $G[P]$ is a k-plex.

Definition 2.4. (*Finding the Densest MkP*) Given a graph G, the problem finds an MkP in G with the largest number of edges.

Algorithm 1: Basic Branch-and-Bound Search

```
1 function BB_basic(S, R, g)    # The basic BB(.) variant
2     if (reduce_and_prune(S, R, g) = true) then return
3     for each v ∈ R do
4         BB_basic(S ∪ {v}, R − {v}, g)
5         R ← R − {v}
```

In Appendix A of our technical report [27], we show a case study where it is necessary to mine multiple MkPs to cover different important communities. Note that the algorithms for these problem variants are just variants of our MkP algorithm (see Section 6).

Hereditariness and Diameter of k-Plex. Note that k-plex satisfies the hereditary property which says that: any induced subgraph (denoted by g') of a k-plex g is also a k-plex, since a vertex in g' cannot miss more neighbors than those already missed in g.

THEOREM 2.5. (Hereditariness) *Given a k-plex $P \subseteq V$, any subset $P' \subseteq P$ is also a k-plex.*

The proof is in Appendix C of our technical report [27].

Moreover, as proved by [55], the diameter of k-plexes with a reasonably large size (which is usually the case for MkPs) is bounded:

THEOREM 2.6. *For a k-plex P and any integer $c \geq 2$, if $|P| > 2k - c$, then $\Delta(P) \leq c$ [55].*

Most works [34, 53] only consider the case when $c = 2$ by assuming $|P| \geq 2k - 1$, which gives the following corollary:

THEOREM 2.7. *Given a k-plex P, if $|P| \geq 2k - 1$, then $\Delta(P) \leq 2$.*

We provide the proof of this special case in Appendix D [27].

The assumption $|P| \geq 2k - 1$ is reasonable for a maximum k-plex P, since natural communities that our MkP problems aim to discover are connected, and $2k - 1$ is relatively small. For example, when $k = 5$, we only require $|P| \geq 9$. Note that a k-plex with $|P| = 2k - 2$ may be disconnected (e.g., formed by two disjoint $(k-1)$-cliques).

Subgraph Exploration by Branch and Bound. All existing MkP algorithms follow (but are variants of) the branch-and-bound search framework shown in Algorithm 1, where S is the set of vertices already included into the current subgraph, and R is the set of candidate vertices yet to be added to S to form a larger subgraph that can become an MkP. Specifically, given a vertex-set pair $\langle S, R\rangle$ in graph g, Line 2 first calls a function reduce_and_prune(S, R, g) to apply reduction and upper-bounding rules, which may decide that the entire search branch to extend S is unpromising so that *true* is returned, in which case Line 2 returns directly to terminate the extension of S. We will discuss this function in more detail when we introduce Algorithm 3 in Section 3. If the branch is not pruned, Line 3 then takes the next candidate $v \in R$, and split the search space into two cases: (1) v is in the MkP to find, in which case Line 4 further extends $\langle S \cup \{v\}, R - \{v\}\rangle$ by recursion; (2) v is not in the MkP to find, in which case Line 5 removes v so that it will not appear in future iterations of the for-loop in Line 3.

Figure 2 illustrates the search process of Algorithm 1 on a toy graph with four vertices a, b, c and d, which corresponds to a set-enumeration search tree where each node denotes S and we also

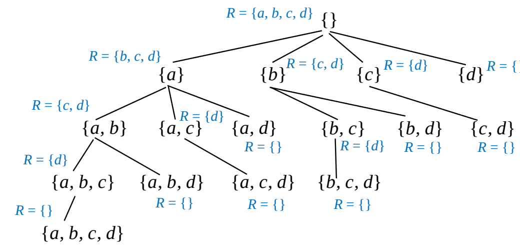


Figure 2: Set-Enumeration Search Tree

annotate its corresponding R near the node (assuming that no node is pruned, and that vertices in R are always ordered with $a < b < c < d$). We can see that the search space is perfectly partitioned without redundancy. Note that calling Algorithm 1 on S basically grows the set-enumeration subtree under node S, which we denote by T_S.

Also note that we do not need to create new input sets $\langle S \cup \{v\}, R - \{v\}\rangle$ at Line 4, but can reuse the space of $\langle S, R\rangle$, and operations like removing v from R can be done in $O(1)$ time, using the dual-array data structure introduced in Appendix I [27] with Figure 15.

Degeneracy Ordering. The k-core of a graph G is its largest induced subgraph with minimum (vertex) degree k. The degeneracy of G, denoted by $D(G)$, is the largest value of k for which a k-core exists in G. It is well-known that the degeneracy of a graph can be computed in linear time by a peeling algorithm that repeatedly removes a vertex with the minimum current degree at a time [30], which produces a degeneracy ordering of vertices. Appendix B of our technical report [27] provides an illustration of this process. Note that $D(G)$ is usually a small value (see Table 2) since while a high-degree vertex tends to appear later in the degeneracy ordering, so many of its neighbors could have already been removed by peeling.

3 U-MkP: A UNIFIED MkP FRAMEWORK

Initialization. Before subgraph exploration, we first aim to find a large (though may not be maximum) k-plex P as well as an upper bound ub on the size of any MkP in G, so that (1) if $|P| = ub$, then P is already an MkP and can be directly returned, otherwise (2) we can still prune those search branches that cannot lead to a larger k-plex with size at least $(|P|+1)$. We follow kPlexS [34] to compute $\langle P, ub\rangle$ while running the peeling algorithm in linear time, and Appendix E provides the detailed algorithm and its explanations.

Top-Level Branching. Recall from Theorem 2.6 that reasonably large k-plexes ($|P| > 2k - c$) have diameter $\Delta(P) \leq c$. In other words, for the top-level nodes v_i ($v_i = a, b, c, d$) in Figure 2, the branch $T_{\{v_i\}}$ under it only needs to consider a subgraph of G, denoted by g_i, where vertices are within c hops from v_i. It is thus worthwhile to shrink G to g_i, and explore subgraphs in the branch $T_{\{v_i\}}$ over g_i since (1) g_i is much smaller than G so checking the neighbors of each vertex becomes much faster, and (2) this is an efficient one-time processing that can benefit the entire branch $T_{\{v_i\}}$. So, we follow this approach which is also adopted by recent algorithms such as kPlexT [35], kPlexS [35], and Maple [53]. In contrast, when processing $T_{\{v_i\}}$, even though whenever a vertex v is added to S, we can shrink g_i further to remove vertices $> c$ hops away from v, this cost is associated with each expansion of S and so not worthwhile.

Algorithm 2 shows the pseudocode to create top-level search branches $T_{\{v_i\}}$, where we assume there are two globally maintained

Algorithm 2: Top-Level Branching ($c = 2$)

Input: A graph $G = (V, E)$ and an integer $k \geq 2$
Output: A maximum k-plex in G

```
1  function top_branching(G)
2      P, ub ← find_init(G)    # global variables set by Algo. 6
3      if |P| ≥ ub then return P
4      CTCP(G, ∅, |P| + 1 − k, |P| + 1 − 2k)
5      Sort [v_1, ..., v_n] following degeneracy ordering of G
6      for i = 1 to |V| do
7          H_1 ← N(v_i) ∩ {v_{i+1}, ..., v_n}
8          H_2 ← N(H_1) ∩ {v_{i+1}, ..., v_n}
9          g ← the subgraph of G induced by ({v_i} ∪ H_1 ∪ H_2)
10         BB({v_i}, V(g) − {v_i}, g)    # call a BB variant
11         Remove v_i from G
12         CTCP(G, {v_i}, |P| + 1 − k, |P| + 1 − 2k)
13     return P
```

variables accessible any time during subgraph exploration: (1) P: the current largest k-plex found, and (2) ub: size upper bound of an MkP. These two variables are initialized in Line 2 using Algorithm 6 in Appendix E, and during subgraph exploration, ub stays the same while P will be updated whenever a larger k-plex is found, so that if we know that a branch cannot generate a k-plex larger than $|P|$, it can be pruned without exploration. If $|P|$ equals ub, then the initial P is already a MkP and is thus returned in Line 3. Otherwise, Line 4 then further prunes unpromising vertices and edges from G using the CTCP reduction technique from kPlexS [34] which we will introduce soon, so that top-level branches are created from the pruned G.

We then create branches $T_{\{v_i\}}$ in the degeneracy ordering as shown in Lines 5-6, which keeps the sizes of all g_i's small to avoid having a giant branch that needs to search a deep set-enumeration subtree with potentially high node fanouts. This is because a high-degree vertex v_i tends to appear later in the degeneracy ordering, so V_i has excluded many neighbors of v_i originally in V.

Without loss of generality, let us assume that $c = 2$, that is, an MkP P has $|P| \geq 2k-1$ (i.e., $|P| > 2k-2$). To create g_i for each v_i, Line 7 first obtains one-hop neighbors of v_i in V_i, and Line 8 then uses them to obtain the two-hop neighbors (where $N(H_1) = \cup_{v \in N(H_1)} N(v)$); finally, Line 9 creates g_i using them (i.e., excluding other vertices in V_i that are > 2 hops away from v_i). Note that it is easy to extend g_i for the general case of c. For example, when $c = 3$, we can additionally compute $H_3 \leftarrow N(H_2) \cap \{v_{i+1}, \ldots, v_n\}$ and then compute g_i to be the subgraph of G induced by $(\{v_i\} \cup H_1 \cup H_2 \cup H_3)$.

Once g_i is created, Line 10 then explores it by extending $S = \{v_i\}$ using branch and bound. For now, we can think of BB(.) simply as BB_basic(.) presented in Algorithm 1, and Section 4 will describe two variants using more efficient branching methods. We then remove v_i from G in Line 11 (similar to Line 5 of Algorithm 1) so that later branches will not consider v_i to avoid redundancy. Note that we do not actually need to do the intersections in Lines 7 and 8 since $\{v_1, \ldots, v_{i-1}\}$ have already been removed in previous iterations.

Core-Truss Co-Pruning (CTCP). kPlexS [34] proposes a CTCP technique to prune the vertices and edges using the fact that to find a larger k-plex with size at least $|P|+1$, we only need to consider those vertices with degree at least $\tau_v = (|P| + 1 - k)$ and those edges (u, v) where u and v share at least $\tau_e = (|P| + 1 - 2k)$ common neighbors (see Theorem D.1 in our technical report [27]). Figure 3 shows an illustration where vertices and edges below thresholds are alternately pruned. kPlexS [34] shows that the reduced graph by CTCP is guaranteed to be no larger than that computed by BnB [40], Maplex [63] and KpLeX [63] and was then the most efficient MkP algorithm.

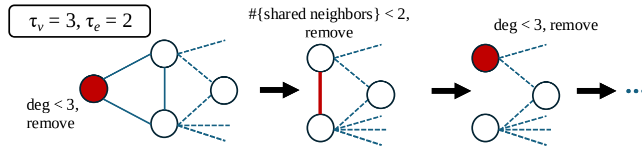


Figure 3: Illustration of CTCP

Algorithm 3: Reduction and Pruning Function

```
1  function reduce_and_prune(S, R, g)  # returns if T_S is pruned
2      if (|P| = ub) then return true
3      Apply reduction rules to (S, R, g)
4      if g_union = g[S ∪ R] is a k-plex then
5          if |S| + |R| > |P| then P ← S ∪ R
6          return true
7      if partition(S, R, g) = ∅ then return true  # Seesaw UB
8      return false
```

We adopt the efficient implementation of CTCP(G, Q_v, *lb_changed*, τ_v, τ_e) from kPlexS [34], where Q_v keeps the set of vertices that must be removed from G, and *lb_changed* is a flag indicating if the current largest k-plex P has changed (so that τ_e is increased and more edges may be pruned). By maintaining all $d_G(v)$ for all v and the triangle counts for all edges (u, v) (denoted by $d_G(u, v)$), and dynamically updates them while propagating pruning from vertices in Q_v (and if *lb_changed* = *true*, also from new edges with $d_G(u, v) < \tau_e$), CTCP can minimize its graph update footprint and overhead to shrink G.

Refer back to Algorithm 2 where for simplicity, we omit argument *lb_changed* of CTCP. Line 4 conducts CTCP over the entire G before creating branches (with *lb_changed* = *false*). Moreover, after Line 11 removes v_i from G, new edges (e.g., v_i's adjacent ones) may have $d_G(u, v)$ reduced below τ_e, so CTCP is called in Line 12 with $Q_v = \{v_i\}$ to shrink G further for use by future iterations (Here, *lb_changed* is determined by whether BB(.) called in Line 10 has found a larger k-plex to update P).

Reduction and Upper-Bound-Based Branch Pruning. Recall from Line 2 of Algorithm 1 that BB(.) first calls reduce_and_prune(.) over the instance (S, R, g) for further reduction before extending S with vertices from R, the pseudocode of which is shown in Algorithm 3. Specifically, Line 1 first checks if the current P already reaches the maximum possible size ub, and if so, it returns *true* to notify BB(.) to terminate its subgraph exploration (see Line 2 of Algorithm 1). Right after the call of BB(.) in Line 10 of Algorithm 2, we can check if $|P| = ub$ and return P directly if so to terminate early.

Otherwise, we apply reduction rules (to be introduced in Section 5) in Line 3 to reduce the candidate size of R. Before extending S with vertices in R (e.g., Lines 3–5 in Algorithm 1), we first conduct a look-ahead pruning in Line 4 to see if the entire $g_{\text{union}} = g[S \cup R]$ is a k-plex, and if so, Line 6 returns true so that Line 2 of Algorithm 1

Table 1: MkP Algorithms and Their Pruning Techniques

Algorithm	Upper Bounding (UB)					Branching					Reduction						
	S-Based	R-Based	SR-Based	UBR1	UBR2	Binary	Pivot	S-Based	R-Based	SR-Based	Two-Hop	RR1	RR2	RR3	CTCP	BR1	BR2
U-MkP (Our Work)	✓	✓	✓		✓		✓	✓	✓	✓	✓ ^{g}	✓	✓	✓	✓ ^{T}	✓	✓
kPlexT (SIGMOD'24)					✓	✓					✓ ^{g}	✓	✓	✓	✓ ^{T}	✓	✓
Maple (IJCAI'23)				✓			✓				✓ ^{g}	✓	✓	✓	✓ $^{+}$	✓	✓
Seesaw (arXiv'23)	✓ $^{+}$	✓ $^{+}$	✓	—	—	—	—	—	—	—	—	—	—	—	—	—	—
DiseMKP (IJCAI'23)	✓ $^{+}$							✓ $^{+}$				✓	✓	✓	✓ ^{T}		
kPlexS (PVLDB'22)				✓		✓					✓ ^{g}	✓	✓	✓	✓ $^{+}$	✓	✓
KpLeX (IJCAI'21)	✓							✓				✓	✓	✓			
Maplex (AAAI'21)		✓				✓					✓	✓	✓	✓	✓ ^{T}		
BnB (IJCAI'18)						✓						✓	✓	✓			
BS (AAAI'17)							✓				✓	✓	✓	✓			✓

*** Note:** ✓ ^{g}: Two-hop graph constructed; ✓ $^{+}$: Improved version; ✓ ^{T}: Top-level only
— : Seesaw only proposes UB rules, no open-source code;

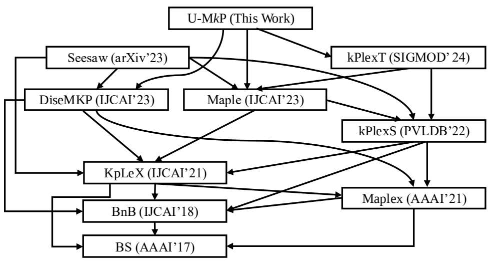


Figure 4: Reported Algorithm Dominance Relationships

will return directly to terminate its exploration of T_S. Moreover, if $S \cup R$ is larger than the current P, P is updated with $S \cup R$ in Line 5.

Finally, we check if T_S can still produce a k-plex larger than P, and if not, Line 7 returns *true* to let BB(.) terminate its exploration of T_S. The function partition(.) (to be introduced in Section 4) uses upper-bound-based pruning to return the subset of vertices in R, denoted by B, that is still worth branching with (i.e., to extend S only with a vertex in B in the next step), and if $B = \emptyset$, there is nothing worth extending with so the entire T_S is pruned.

Overview of Pruning Techniques. We now place the various pruning techniques of nine recent algorithms under our U-MkP framework established in Section 3, including kPlexT [35] Maple [53], Seesaw [62], DiseMKP [44], kPlexS [34], KpLeX [45], Maplex [63], BnB [40] and BS [55]. Table 1 summarizes the pruning techniques of these algorithms into three categories, those for upper bounding, branching, and reduction. We can see that most algorithms are from the AI community (AAAI and IJCAI), since we can regard MkP finding as a search problem. Specifically, the set-enumeration tree defines a state space where each node S is a state, extension set R defines the successor function, and an MkP for S to expand into is a goal state. Moreover, the two works kPlexT [35] and kPlexS [34] are from the DB community, actually the same group. Figure 4 shows the performance dominance relationships reported by the respective papers to their compared algorithms.

We have covered two reduction rules "Two-Hop" (or more generally, "c-Hop") and "CTCP" since they are essential for describing our U-MkP framework, and we will introduce the other reduction rules in Section 5. Branching rules will be introduced in Section 4.

Regarding upper-bounding rules, one usage is to prune an entire unpromising branch T_S as mentioned in Line 7 of Algorithm 3, but since it is also applied by a branching strategy, we will present the details in Section 4.1 next. Note that there are still two more upper-bounding rules UBR1 and UBR2 in Table 1, and since they are used for reduction, we will present them in Section 5.

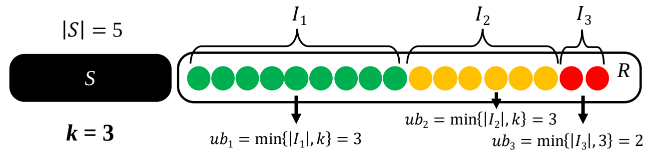


Figure 5: Illustration of Partition-Based Upper Bounding

Also, Table 1 shows that BS uses more techniques than BnB but is slower. This is because instead of finding a large initial k-Plex as in Algorithm 6 to enable effective pruning (e.g., by CTCP in Line 4 of Algorithm 2), BS solves a decision version of MkP that guesses its size, and relies on binary search on the size range to locate MkP by running explorations for up to $O(\log |V|)$ times [55].

4 BRANCHING METHODS

This section reviews two competitive branching methods, one using **partition-based upper bounding**, and the other using **pivoting**, both significantly beating a baseline binary branching method. For the former, we show that although the latest cost-model-driven Seesaw upper bound [62] was originally proposed only for entire-branch pruning (see Line 7 of Algorithm 3), it can actually be adapted for branching. For the latter, we actually reverse engineered this unmentioned technique from the code of Maple [53], which unfortunately, implements something totally different from what their paper described. Recall our contribution (3) in Section 1.

4.1 Partition-Based Upper Bounding

The partitioning-based upper bounding technique was proposed and improved by series of works: Maplex [63], KpLeX [45], DiseMKP [44], Seesaw [62]. Given $\langle S, R\rangle$, the goal is to compute a size upper bound UB_S on the largest k-plex that S can be extended into using candidates from R (i.e., within branch T_S), by partitioning vertices of R into t subsets $I_1, I_2, \ldots, I_t$. If for each I_i, we can show that at most ub_i vertices from I_i can be added to S without breaching the k-plex requirement, then UB_S can be computed as $|S| + \sum_{i=1}^{t} ub_i$.

We illustrate by the example in Figure 5 that uses Maplex's color-based method, which repeatedly removes a maximal independent set I_i from R at a time. Since vertices in I_i do not have any mutual edges (so can share the same color during graph coloring, hence the name), if more than k vertices are added from I_i to S, then each added vertex will have more than k non-neighbors in S, breaching the k-plex requirement and the set cannot be further extended into a k-plex due to the hereditary property. Therefore, $ub_i = \min\{|I_i|, k\}$. In Figure 5, the upper bound is given by $UB_S = |S| + \sum_{i=1}^{3} ub_i = 5+3+3+2 = 13$.

R-Based Upper Bounding. We present the above algorithm to obtain each $\langle I_i, ub_i\rangle$ in Appendix F of our technical report [27]. We call this method as **R-based strategy**, since it computes independent sets directly from R without referring to S's content.

Seesaw [62] further tightens this bound. To explain its method, we first define the concept of vertex support.

Definition 4.1. (*Support*) The support of a vertex v is defined as $\delta_S(v) = k - |\overline{N_S}(v)|$, which indicates the maximum number of non-neighbors of v that can be added to S.

Intuitively, $\delta_S(v)$ serves as the non-neighbor "quota" of v that can be added to S (including v itself if $v \notin S$) without breaching the k-plex requirement. This support definition is also used a lot in the reduction rules, as we shall see in Section 5.

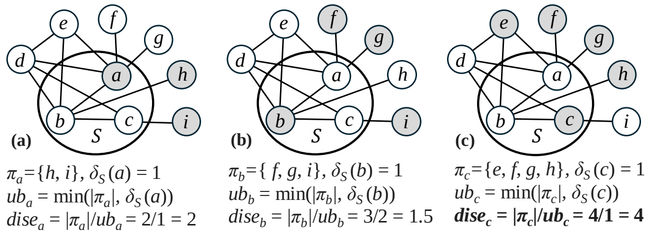


Figure 6: An Example Graph for S-Based Partitioning

We actually maintain $d_S(v)$ incrementally and keep it up to date whenever we move vertices around R and S, so $\delta_S(v) = k - (|S| - d_S(v))$ can always be obtained in $O(1)$ time for use by our pruning techniques. See Appendix J [27] for the details.

Now we are ready to present Seesaw's R-based strategy. Seesaw tightens $ub_i = \min\{|I_i|, k\}$ into $ub_i = \max\{i \mid \delta_S(v_i) \geq i\}$ (see Lemma 1 of [62]). Moreover, for each $\langle I_i, ub_i \rangle$ obtained where I_i is a maximal independent set, Seesaw [62] further relaxes I_i to include additional vertices from R without increasing the value of ub_i (see Lemmas 2–3 of [62]). This reduces the number of remaining vertices in R (from which future partitions are obtained) hence tends to reduce the upper bound UB_S. Our current work uses this improved approach to obtain $\langle I_i, ub_i \rangle$ rather than the simple approach of Maplex, and we denote this operation by

$$\langle I_i, ub_i \rangle \leftarrow \text{get\_R\_part}(R, S, g)$$

where get_R_part(.) is specified by Algorithm 2 of [62].

S-Based Upper Bounding. KpLeX [45] proposes a different way to obtain $\langle I_i, ub_i \rangle$ by partitioning vertices of R based on vertices in S, so we call this approach **S-based strategy**. Specifically, it partitions R by obtaining from R (1) the common neighbors of all vertices in S as π_0, (2) for each $v_i \in S$, obtain π_i as all the remaining non-neighbors of v_i in R. Consider the graph in Figure 6(a) where $S = \{a, b, c\}$ and vertices d to i belong to R: if we check vertices in S by the order $[a, b, c]$, we obtain $\pi_0 = \{d\}$, $\pi_1 = \{h, i\}$, $\pi_2 = \{f, g\}$, $\pi_3 = \{e\}$.

KpLeX treats each π_i ($i \geq 1$) as I_i, we have $ub_i = \delta_S(v_i)$ since it is the quota of non-neighbors of v_i that can be pulled into S. Clearly, we can compute $UB_S = |S| + |\pi_0| + \sum_{i=1}^{|S|} \min\{|\pi_i|, \delta_S(v_i)\}$. However, the value of UB_S depends on the checking order of vertices in S.

To find an ordering that leads to small UB_S, DiseMKP [44] proposes the concept of distribution efficiency (dise) where $dise(\pi_i) = |\pi_i|/ub_i$. Intuitively, we prefer high $dise(\pi_i)$ since we want to remove a large set π_i from R while adding a small ub_i to UB_S. Based on this idea, DiseMKP proposes to greedily check $dise(\pi_i)$ for all the remaining $v_i \in S$ whose π_i has not been selected, and choose the one with the highest $dise(\pi_i)$ as the next I_i. Figure 6 illustrates how I_1 is determined by computing the dise scores, and π_c is picked as I_1 since it has the highest dise. Note that π_i's are initialized as all non-neighbors of v_i that could overlap, so after $I_1 = \pi_c$ is picked, we need to update $\pi_a = \pi_a - I_1 = \{i\}$ and $\pi_b = \pi_b - I_1 = \{i\}$, and then pick I_2 by comparing their dise scores.

We denote the above operation to obtain a partition I_i as

$$\langle I_i, ub_i \rangle \leftarrow \text{get\_S\_part}(R, S, g, \Pi)$$

where $\Pi = \cup_{v_i \in S} \pi_i$ is an auxiliary set that keeps the (potentially overlapping) non-neighbor sets π_i of all $v_i \in S$, which is initialized before computing UB_S, and tracks the candidate partitions to choose next. Note that whenever a partition π_i is chosen, its vertices are removed from all partitions in Π, so if π is previously chosen, it will become empty. Function get_S_part(.) simply chooses the next partition among those non-empty π_i's in Π, and the detailed algorithm is specified in Appendix G of our technical report [27].

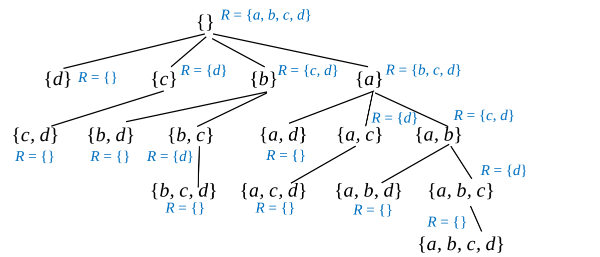


Figure 7: Set-Enumeration Search Tree (Horizontal Flip)

SR-Based Upper Bounding. Since $dise(\pi_i) = |\pi_i|/ub_i$ is also well defined for R-based partitions, Seesaw [62] proposes to run both method when picking the next partition from R:

$$\langle I_i^R, ub_i^R \rangle \leftarrow \text{get\_R\_part}(R, S, g),$$
$$\langle I_i^S, ub_i^S \rangle \leftarrow \text{get\_S\_part}(R, S, g, \Pi),$$

and pick the partition with the higher dise as the next partition I_i. Of course, vertices of I_i need to then be removed from R and all candidate sets π_j maintained for the S-based strategy, and ub_i is added to UB_S. This is repeated until R becomes empty (so π_0 will be refined by R-based strategy at the end), after which we add $|S|$ to UB_S to obtain the final UB_S. This method is the **SR-based strategy**.

4.2 Partition-Based Branching

Branching Set. The above partition-based upper bounding technique can actually be used for branching that allows some branches to be pruned, as proposed by DiseMKP [44]. Specifically, let us consider the horizontal flip of the set-enumeration tree in Figure 2, which we show in Figure 7. We can observe that when we scan the candidates from right to left till reaching a candidate $v \in R$, the set-enumeration subtrees rooted at v and all the candidates in front of v in R can only involve vertices in S, v, and those vertices before v in R. For example, for node $S = \{\}$ in Figure 7, when we scan R and reach b, all the three subtrees rooted at nodes $\{d\}$, $\{c\}$ and $\{b\}$ contain vertices in $\{b, c, d\}$. Similarly, for node $S = \{a\}$, when we scan R and reach c, the two subtrees rooted at nodes $\{a, d\}$ and $\{a, c\}$ contain vertices in $\{a, c, d\}$. Now let us consider Figure 8 with $\langle S, R \rangle$, and the last few vertices constitute a set B. Following the above discussion, we know that subtrees rooted at the vertices in $R' = R - B$ can only involve those vertices in $S \cup R'$. If we can show that the size upper bound of a k-plex obtained by extending S with candidates in R', denoted by $UB_S(R')$, cannot exceed $|P|$ (i.e., cannot produce a larger k-plex), then we do not need to extend S with those vertices in R' in the next level. In other words, we only need to branch over vertices in B, hence we call B the branching set.

Note, however, that vertices of R' can still appear in the subtrees of those nodes that extend S with vertices in B, so they are not removed from g (i.e., not reduction) but just skipped for the level of branching below S. To illustrate with Figure 7 again, assume that $R' = \{d, c\}$ for $S = \{\}$, then even if we skip the subtrees under $\{d\}$ and $\{c\}$, c and d can still appear in the subtrees under $\{b\}$ and $\{a\}$.

We can use the partition-based upper bounding techniques presented in Section 4.1 to construct R' (and hence $B = R - R'$). As

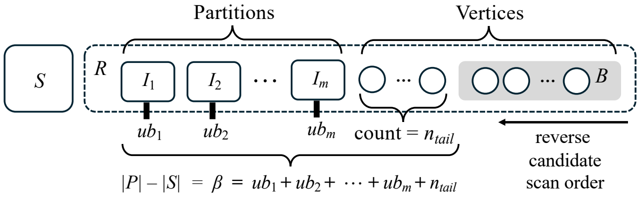


Figure 8: Illustration of Branching Set Computing

Algorithm 4: Partitioning-Based Branch and Bound

Input: Flag σ_{seed}: true if called by the top level.
Flag σ_{up}: a global variable indicating branch pruning

```
1  {Updates P if a larger maximum k-plex is found}
2  function BB_part(S, R, g, σ_seed)
3      if (reduce_and_prune(S, R, g) = true) then return
4      B ← partition(S, R, g)
5      R′ ← R \ B
6      Sort B = [v_1, ..., v_n] following degeneracy ordering of g
7      for i = |B| to 1 do
8          if (σ_seed = true) then σ_up ← false
9          if (σ_up = true) then return
10         τ_old ← |P|
11         BB_part(S ∪ {v_i}, R′, g, false)
12         if |P| > τ_old then σ_up ← true
13         R′ ← R′ ∪ {v_i}
```

Figure 8 illustrates, our goal is to find R' as a subset of R that is as large as possible so that $UB_S(R') \le |P|$. Let us view the computation of $UB_S(R')$ as the following process: initially, $UB_S(R')$ is set as $|S|$, and we then partition R' into partitions $\langle I_i, ub_i \rangle$ and add all ub_i values to $UB_S(R')$ to obtain the final value of $UB_S(R')$.

We can use any of the S-based strategy, R-based strategy, or SR-based strategy presented in Section 4.1 to find $\langle I_i, ub_i \rangle$ one at a time and expand R' with I_i. Since we require $UB_S(R') \le |P|$, we have the quota $\beta = |P| - |S|$ for the summation of ub_i values when we choose as many partitions I_i into R' as possible. Whenever a partition i_i is chosen, we deduct ub_i from β so that β is always the remaining quota. We denote the above process to compute B ($R' = R - B$) as

$$B \leftarrow \text{partition}(S, R, g),$$

which keeps obtaining $\langle I_i, ub_i \rangle$ as long as $\beta \ge ub_i$ (during S-based partition selection as shown in Figure 6, we also do not consider those candidates π_i with $ub_i > \beta$). Finally, as Figure 6 illustrates, assume that $ub_{m+1} > \beta$ for the next partition I_{m+1} (if exists), we then stop obtaining this and future partitions (with potentially lower dise scores), but rather taking β more vertices from R (if exists) to prune more branches by letting $UB_S(R')$ reach the allowed value $|P|$.

Algorithm 9 in Appendix H of our technical report [27] shows the pseudocode of partition(S, R, g) when we apply the most sophisticated SR-based strategy, and the counterparts for the other two strategies can be similarly derived (but much simpler).

Partition-Based Branch and Bound. Operating on the horizontally flipped set-enumeration tree as shown in Figure 7 allows additional branch pruning as follows: assume that we have processed the top-level branches $T_{\{v_1\}}, T_{\{v_2\}}, \ldots, T_{\{v_{i-1}\}}$ and the current largest k-plex is P, then we can find a k-plex with at most $(|P| + 1)$ vertices in branch $T_{\{v_i\}}$, so as soon as we find such a k-plex, we can skip the rest of $T_{\{v_i\}}$ (where we can find at most ties) and move on to $T_{\{v_{i+1}\}}$.

To see this, consider Figure 7 again, and assume that $T_{\{d\}}$, $T_{\{c\}}$ and $T_{\{b\}}$ have been processed so that P is an MkP found over $\{b, c, d\}$. Then in $T_{\{a\}}$, we show that we cannot we find a k-plex P' of size more than $|P| + 1$ by contradiction: assume that $|P'| > |P| + 1$, then since a is the only additional vertex beyond $\{b, c, d\}$, we must have $a \in P'$ (otherwise, P is not an MkP over $\{b, c, d\}$); by the hereditary property of k-plex, $P' - \{a\} \subseteq \{b, c, d\}$ is also a k-plex but has size more than $|P|$, which contradicts with the fact that P is an MkP over $\{b, c, d\}$. This proof can easily be adapted to the general case.

Based on this idea, Algorithm 4 shows the partition-based branch-and-bound algorithm to be called by Algorithm 2 with $\sigma_{seed} = true$. Here, σ_{seed} indicates if the function is called by the top level, and within each branch under a top-level node, the recursive call passes *false* to σ_{seed} as Line 11 shows. As a result, when computation just enters a top-level branch $T_{\{v_i\}}$, Line 8 will initialize the global flag variable σ_{up} to *false* to indicate that a k-plex of size $(|P| + 1)$ has not been found in $T_{\{v_i\}}$ yet. Line 11 then recursively extends $S \cup \{v_i\}$ for a vertex $v_i \in B$, and Line 12 checks if the recursive call has increased $|P|$ beyond its old value recorded in Line 10. If so, a larger k-plex is found (and must have size $|P| + 1$ based on the previous discussion), so we terminate the exploration of $T_{\{v_i\}}$ by directly returning in Line 9 along the backtracking path all the way to the top level, which will then proceed the exploration to $T_{\{v_{i+1}\}}$.

Note from Lines 6–7 that we only branch on vertices in B, and we check vertices from $v_{|B|}$ to v_1 in the reverse degeneracy ordering, so that dense regions are examined first in hope that a larger P can be identified early to prune a lot of unpromising branches.

Implementing Auxiliary Buffers Π. Recall that the S- and SR-based strategies require an auxiliary set $\Pi = \cup_{v_i \in S} \pi_i$ that keeps the (potentially overlapping) non-neighbor sets π_i of all $v_i \in S$. Creating and initializing it for each time when partition(.) is called in Line 4 of Algorithm 4 is very expensive, so we propose to preallocate buffer space for Π to be reused during recursive subgraph exploration in a space-efficient manner. The details can be found in Appendix K [27].

As our experiments shall show, the partition-based branching method is usually the most efficient branching method for the small values of k (2 to 5) that are the focus of most papers (only Maple [53] tested large values for k), but only adopted by S-based methods KpLeX [45] and DiseMKP [44] with the scheme not clearly explained, so later works such as kPlexT [35] still uses binary branching that is less efficient. By clearly explaining this partition-based scheme and extending it to support all 3 variants of partition-based strategies (S-, R-, and SR-based), we hope to motivate future MkP works to consider this partition-based scheme when k is small.

Also note that instead of evaluating if $UB_S \le |P|$ in Line 7 of Algorithm 3, we evaluate if $B = \emptyset$ in Line 7, which is equivalent.

4.3 Binary Branching

As Table 1 shows, most algorithms adopt simple binary branching that given instance $\langle S, R \rangle$, chooses a vertex $v \in R$ with the smallest degree (to follow the degeneracy ordering) and divides into two instances $\langle S \cup \{v\}, R - \{v\} \rangle$ (i.e., P containing v) and $\langle S, R - \{v\} \rangle$ (i.e., P not containing v). However, there is no branch pruned like R' in partition-based method, so simple binary branching is inefficient.

Algorithm 5: Pivoting-Based Branch and Bound

Input: P, B are global variables, v_i is the top-level vertex
$\mathcal{S}$ is the candidate stack
Output: Updates P if a larger maximum k-plex is found

1 **function** BB_pivot(S, R, g)
2   **if** (reduce_and_prune(S, R, g) = *true*) **then return**
3   **if** $\mathcal{S} = \emptyset$ **or** $\mathcal{S}.top() \in S$ **or** $\mathcal{S}.top() \notin R$ **then**
4   **if** $\exists v \in R$, *s.t.* $(v, v_i) \notin E(g)$ **and** $\big(\delta_S(v) = 1$ **or** $\delta_S(v_i) = 1$ **or** $d_g(v) + k \leq |P| + 1\big)$ **then**
5       $\mathcal{S} \leftarrow \{v\}$
6   **else**
7   v_p = A vertex with minimum $d_g(.)$ in R
8       $\mathcal{S} \leftarrow R - N_g(v_p)$
9   Sort $\mathcal{S}$ in descending order of $d_g(.)$
10   $v \leftarrow \mathcal{S}.pop()$
11   BB_pivot($S \cup \{v\}, R - \{v\}, g$)
12   Remove v from g
13   $\mathcal{S}.clear()$
14   BB_pivot(S, R, g)

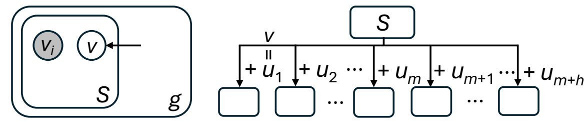
Interestingly, kPlexT [35] recently proposes a slightly improved binary branching method to choose v more smartly, which is essential for proving their improved worst-case time complexity. However, our experiments show that their branching method does not obviously improve performance, and efficiency of kPlexT is mainly contributed by a new reduction rule UBR2 and avoiding CTCP in non-top-level BB recursion (in contrast to their prior algorithm kPlexS).

In contrast, our experiments show that the pivot-based binary branching method of Maple [53], as shown in Algorithm 5, can significantly improve performance, and is often many orders of magnitude faster for branching when k is large. Interestingly, Algorithm 5 is reverse-engineered from Maple's code, and their paper [53] proposes something totally different that finds MkP by solving a complement problem of k-plex called d-BDD, which is not actually implemented in their GitHub repo, so cannot be compared.

Our experiments reveal that while partition-based branching is the most efficient for small k values, it is not competitive to BB_pivot(.) (Algorithm 5) when k becomes larger. This is mainly because each $ub_i = \min\{|\pi_i|, \delta_S(v_i)\}$ becomes loose when k is large (since $\delta_S(v_i) = k - |\overline{N_S}(v_i)|$), reducing the pruning power of BB_part(.) (Algorithm 4). We recommend BB(.) to choose BB_part(.) when $k \leq 5$ and BB_pivot(.) otherwise (see Appendix L [27]).

We now explain Algorithm 5, which treats MkP finding as a constraint satisfaction problem (CSP) and applies the most-constraining-variable heuristic to find candidates to extend S that tend to reduce the remaining candidates in R the most. As we can see from Lines 11, 12 and 14, BB_pivot(.) is still a binary branching method that selects a pivot v at a time to divide the instance. The key success lies in its two methods to select v to be the most constraining.

In Case 1, Line 4 aims to find v as a non-neighbor of top-level vertex v_i (i.e., we are exploring $T_{\{v_i\}}$) which decrements $\delta_S(v_i)$ to promote pruning. For such a v we also require it to fall in one of the three candidate-constraining cases (see Figure 9(a)): **(1)** $\delta_S(v) = 1$, so after v is added to S (hence $\delta_S(v) = 0$), all non-neighbors of v can be pruned from R; **(2)** $\delta_S(v_i) = 1$, so after v is added to S



(a) The "if" Case (b) The "else" Case, $N_g(v) = \{u_{m+1}, \ldots, u_{m+h}\}$

Figure 9: Illustration of Pivot Selection

(hence $\delta_S(v_i) = 0$), all non-neighbors of v_i can be pruned from R; **(3)** $d_g(v) + k \leq |P| + 1$, so extending $S \cup \{v\}$ cannot lead to a k-plex larger than $|P| + 1$ (see reduction rule RR3 in Appendix M [27]) and is more likely to be pruned ($d_g(v) + k > |P|$ must hold, or RR3 should have pruned v in Line 2). If v is found, Line 5 will set $\mathcal{S}$ to contain only v, and v will be popped in Line 10 for binary branching.

Otherwise, the else-branch (Case 2) finds $v_p \in R$ as the vertex with minimum $d_g(.)$, and pushes the candidates in R to $\mathcal{S}$ so that they are popped in Line 10 in non-decreasing order of $d_g(.)$ (to follow the degeneracy ordering), with v_p being the first to pop. Note that as Line 8 does, we do not add $N_g(v_p)$ to $\mathcal{S}$ since as Figure 9(b) shows, the last h branches that correspond to candidates from $N_g(v_p)$ can only produce k-plexes $P' \subseteq N_g(v_p)$; since v_p connects to every vertex in P', $P' \cup \{v_p\}$ is also a k-plex, so P' cannot be an MkP.

As we shall see in Section 5, reduction rules in Line 2 may move some candidates directly from R to S, leading to $\mathcal{S}.top() \in S$; and may prune some candidates already added to $\mathcal{S}$, leading to $\mathcal{S}.top() \notin R$, in which case Line 3 will activate the selection of a new pivot. Otherwise, the if-branch in Line 3 will be skipped and next candidate in $\mathcal{S}$ will be popped for binary branching.

5 REDUCTION METHODS

This section briefly overviews the reduction techniques summarized in Table 1, focusing on categorizing them. Due to the space limit, the detailed rules and their proofs can be found in Appendix M [27].

The general idea of reduction rules is to remove unpromising vertices from g. These rules check $d_g(v)$ and $\delta_S(v) = k - (|S| - d_S(v))$ for vertices v frequently, so we incrementally maintain $d_g(v)$ and $d_S(v)$ and keep them up to date whenever we move vertices around R and S, so that $d_g(v)$, $\overline{d_g}(v) = V(g) - d_g(v)$ and $\delta_S(v)$ can be always be obtained in $O(1)$ time for use by our reduction rules below. See Appendix J [27] for the details on incremental degree maintenance.

Section 3 discussed **TwoHop** and **CTCP**. We will now discuss the remaining ones: RR1–RR3, BR1–BR2, and UBR1–UBR2. Note that while kPlexS [34] promotes CTCP for each subgraph extension, we find its overhead to be high, and using the other more efficient reduction rules for reduce_and_prune(.) in BB(.) is more favorable. This is also what was done in their follow-up work kPlexT [35].

RR1–RR3 shrink R. Given (S, R, g), a vertex $v \in R$ is unpromising if $S \cup \{v\}$ is not a k-plex, so **RR1** and **RR2** prune such v from R, based on conditions that check $\delta_S(.)$. Also, v is unpromising if extending $S \cup \{v\}$ cannot produce a k-plex larger than P, so **RR3** prunes such v from R based on a condition that checks $d_g(v)$.

RR1-RR3 are used by all MkP algorithms, so we put them in the same category. There are two more sophisticated upper-bound-based reduction rules that utilize tighter upper bounds (of the size of a k-plex extending $S \cup \{v\}$), so are more powerful in pruning candidates in R: **UBR1** is proposed by kPlexS [34], and **UBR2** by kPlexT [35], both from the same group in the DB community. Notably, our experiments reveal that UBR2 is particularly effective as a reduction rule even though it is only discovered very recently.

Finally, there are also two reduction rules that directly add a vertex $v \in R$ to S, based on conditions related to $d_g(.)$. Specifically, **BR1** is first proposed by kPlexS [34] based on conditions related to $d_g(v)$ which ensures that, if there exists an MkP P containing S, then there must also exist an MkP P' containing $S \cup \{v\}$. Therefore, if we only need to find one MkP, we can directly move such a v into S, but if our goal is to find all MkPs, we also need to consider the other branch where v is removed from R. In contrast, **BR2** is first proposed by BS [55] based on conditions related to $d_g(.)$ of v and all $u \in \overline{N_g}(v)$ which ensures that, every MkP in g must contain v, so we can safely move such v to S, even when we are finding all MkPs.

6 MkP VARIANTS AND PARALLELIZATION

MkP Variants. For the variant that finds all MkPs (rather than an arbitrary one) which we call as **All-MkP**, we propose a two-phase approach. Phase 1 finds the size of MkPs (denoted by p_{max}) using our MkP framework. Then, Phase 2 loads G again and prunes it using CTCP with $\tau_v = p_{max} - k$ and $\tau_e = p_{max} - 2k$. During subgraph exploration, we lock the threshold of the current largest k-plex size to be $p_{max} - 1$, so that the pruning techniques will not prune the search space that can lead to any k-plex of size p_{max}; and whenever such an MkP is found, it is immediately emitted as a result without incrementing the threshold. Also, we cannot enable BR1 in reduce_and_prune(S, R, g) to avoid missing any MkP.

For finding the MkPs with the most edges (**Densest-MkP**), Phase 2 instead only update the current densest MkP if a newly found one has more number of edges.

All-MkP can find diversified communities that have a tie in size but may not overlap much, while Densest-MkP can further identify a denser community among them. Appendix A [27] provides a case study to show how these variants can avoid missing important dense communities due to returning only one MkP. Note that the algorithms for these problem variants are just variants of our MkP algorithm.

The Parallel Algorithm. Our MkP framework can be easily parallelized with a task-based model [46, 57] that has been extensively applied in compute-intensive graph search and mining problems [36, 41, 46, 56, 58–61]. The idea is to add all $v_i \in V$ into a task queue Q, where each element v_i defines a top-level branch $T_{\{v_i\}}$ as an independent task that can be assigned to an idle CPU core for recursive mining by first creating the two-hop graph (denoted by g_i) created for v_i as shown in Lines 7–9 of Algorithm 2. At any time, we only process a window of θ tasks and hence maintain θ two-hop graphs, where θ is the number of computing threads.

Since $T_{\{v_i\}}$ can have drastically different sizes with some giant branches becoming stragglers, we adopt the timeout mechanism [36, 41, 46, 60, 61] where a task recursively mining branch T_S can time out after running beyond a time threshold τ_{time} (= 0.1 ms by default), after which any explored node S' during backtracking will become a new task for mining $T_{S'}$. New tasks are directly added back to the task queue to be scheduled for processing by idle threads. Since tasks created from the same top-level branch $T_{\{v_i\}}$ share the same graph g_i, we adopt the approach detailed in Section 6 of [36] to group these tasks into one task group: each initial task with $S = \{v_i\}$ creates a task group that keeps g_i for use by its tasks, while new tasks created due to the decomposition of an existing task in v_i's task group are added to the same task group. A task group keeps track of its active tasks with its own queue, and is released from memory together with g_i when a thread finishes the last task in the group. To keep memory bounded, only a window of θ task groups are processed at any time, and when there are already θ task groups, the next new root-level task $T_{\{v_i\}}$ can only start its evaluation when some existing task group is completed.

Recall that Line 11 of Algorithm 2 removes v_i when $T_{\{v_i\}}$ is processed, and Line 12 subsequently applies CTCP(.) to further shrink the graph G. However, this is not thread-safe in our parallel implementation since it is possible that when a thread is still creating g_i from G that needs a vertex u, another thread may have already finished a different branch $T_{\{v_j\}}$ and deleted u (either because $u = v_j$ or due to CTCP). We, therefore, disable Lines 11–12 but instead prune those vertices v_j with $j < i + 1$ in Lines 7–8 (which is not needed in the serial algorithm due to Line 11) when constructing g_i.

Table 2: Datasets

Dataset	$\|V\|$	$\|E\|$	d_{max}	d_{avg}	$D(G)$	Category
johnson8-4-4	70	1855	53	53	53	Synthetic Graph
keller4	171	9435	124	110	102	Synthetic Graph
socfb-Duke14	9885	506,437	1887	102	85	Facebook Network
ia-wiki-Talk	92,117	360,767	1220	8	58	Interaction Network
soc-buzznet	101,163	2,763,066	64,289	55	153	Social Network
soc-LiveMocha	104,103	2,193,083	2980	42	92	Social Network
soc-gowalla	196,591	950,327	14,730	10	51	Social Network
soc-digg	770,799	5,907,132	17,643	15	236	Social Network
sc-ldoor	909,537	20,770,807	76	46	34	Scientific Computing
soc-youtube-snap	1,134,890	2,987,624	28,754	5	51	Social Network
soc-lastfm	1,191,805	4,519,330	5150	8	70	Social Network
soc-orkut	2,997,166	106,349,209	27,466	71	230	Social Network
wikipedia-link-en	13,593,032	669,183,050	1,052,326	49	1114	Hyperlink Network
dbpedia-link	18,268,991	253,780,418	612,308	14	149	Hyperlink Network
wikipedia-link-en13	25,921,548	1,086,367,222	4,271,341	42	1120	Hyperlink Network
delicious-ui	33,778,221	203,595,714	29,319	6	193	Social Network
soc-sinaweibo	58,655,849	522,642,066	278,489	9	193	Social Network
web-ClueWeb09	147,925,593	893,533,906	308,477	6	192	Web Graph

7 EMPIRICAL STUDIES

This section reports our comprehensive experiments to evaluate the various pruning techniques under the proposed U-MkP framework, as well as its comparison with the state-of-the-art MkP algorithms. Our code is written in C++17 and compiled by g++ version 12.3.0 with optimization flag -O3. All the experiments are conducted on a platform with 24 cores (Intel Xeon Gold 6248R) and 256GB RAM. The unit of time we report is "second" unless stated otherwise.

Datasets and Experimental Settings. We use 26 datasets as described in Appendix N [27]. Due to page limit, we only show the results of 18 graphs as summarized in Table 2, and results on the full datasets can be found in our technical report [27]. In Table 2, d_{max} and d_{avg} indicate the maximum degree and average degree, respectively, and $D(G)$ is the degeneracy. These datasets span a wide range of graph sizes and categories, including 6 with over 10^7 vertices.

We set the time limit as 1800 s (s = seconds). We use × in tables and "OOT" in text descriptions to indicate that execution exceeds this time limit. We tested the datasets for 11 different values of k, i.e., 2, 3, 4, 5, 6, 7, 8, 9, 10, 15 and 20. See Table 10 in our technical report [27] for the MkP sizes of all datasets for all values of k.

Default Configuration of Pruning Techniques: A Summary. Recall all our pruning techniques in Table 1. Interestingly, we find that there exists a configuration of the these techniques (dependent on k only) that leads to the best performance in vast majority of the time: we do not apply upper bounding; for branching, we adopt S-based method when $k \le 5$ and pivot-based method when $k > 5$; for reduction, we enable Two-Hop, top-level CTCP, RR1–RR3, BR1, BR2, UBR2, but disable recursive CTCP and UBR1. We will report

Table 3: Comparison of Branching Techniques (Second Best is Underscored)

Dataset	$k=2$				$k=3$				$k=8$				$k=10$			
	S-Br	R-Br	SR-Br	Pivot-Br	S-Br	R-Br	SR-Br	Pivot-Br	S-Br	R-Br	SR-Br	Pivot-Br	S-Br	R-Br	SR-Br	Pivot-Br
johnson8-4-4	**0.7**	1.1	0.8	1.0	**5.7**	18.9	8.1	10.3	×	×	×	×	×	×	×	**1271.9**
keller4	25.2	49.9	**17.7**	69.8	**28.9**	87.9	31.3	274.0	×	×	×	×	×	×	×	×
socfb-Duke14	**1.2**	1.9	2.1	1.8	**0.8**	1.0	1.7	0.8	656.9	6.2	×	**0.6**	×	52.0	×	**0.6**
ia-wiki-Talk	0.8	1.1	**0.6**	1.3	**0.6**	0.8	0.6	0.8	44.7	2.8	22.8	**0.9**	436.6	4.3	101.0	**1.5**
soc-buzznet	**358.0**	933.6	389.1	1365.3	**189.1**	589.5	770.2	456.0	×	1438.2	×	**75.4**	×	×	×	**150.3**
soc-LiveMocha	3.3	5.6	**2.7**	6.6	**2.2**	3.5	2.6	3.2	492.7	62.2	246.0	**4.1**	×	16.9	388.6	**2.8**
soc-gowalla	0.1	0.1	0.1	**0.1**	0.1	0.1	**0.1**	0.1	**0.1**	0.1	0.1	0.1	0.4	0.5	0.5	**0.1**
soc-digg	**121.9**	226.5	204.9	304.0	**49.4**	162.5	1373.8	80.2	1296.2	×	×	**49.1**	1580.5	×	×	**35.1**
sc-ldoor	14.1	14.7	14.4	**13.6**	24.6	26.4	26.3	**24.2**	**6.9**	7.0	7.2	7.1	9.0	**8.1**	8.8	8.6
soc-youtube-snap	0.4	0.4	0.4	**0.4**	0.5	0.5	**0.5**	0.5	1.4	0.8	1.1	**0.4**	21.5	1.0	9.9	**0.5**
soc-lastfm	3.6	4.6	**2.4**	7.5	**1.9**	2.9	2.1	3.4	3.4	2.0	45.4	**1.1**	54.2	6.8	68.4	**1.6**
soc-orkut	96.1	135.1	179.0	**96.0**	79.1	103.0	511.1	**77.3**	×	×	×	**49.9**	×	×	×	**46.1**
wikipedia-link-en	12.1	12.5	11.8	**11.7**	12.0	**11.9**	14.3	13.4	**67.5**	68.3	69.4	69.0	12.4	12.6	13.5	**12.0**
dbpedia-link	76.0	64.8	**54.8**	233.4	52.2	64.5	**48.2**	404.0	**51.3**	×	53.1	×	**81.1**	×	×	×
wikipedia-link-en13	249.0	260.3	288.5	**240.5**	290.7	277.7	269.8	**247.3**	302.8	304.6	297.7	**284.1**	330.6	283.1	**265.1**	292.3
delicious-ui	62.1	57.8	63.9	**51.9**	99.0	**77.1**	94.1	161.7	1320.7	987.1	**971.4**	×	×	×	×	×
soc-sinaweibo	**141.2**	1036.8	1805.6	306.3	**126.3**	1397.6	×	651.5	×	×	×	×	×	×	×	×
web-ClueWeb09	**43.8**	167.5	155.0	192.0	**70.5**	×	×	×	×	×	×	×	×	×	×	×

the ablation studies to reach this conclusion next, before we compare this default configuration of U-MkP with other existing algorithms.

Choice of Upper-Bounding (UB) Techniques: Recall that Section 4.1 presented three upper bounding (UB) techniques proposed by the AI community: S-Based, R-Based and SR-Based. We studied the effect of these upper bounding techniques and found that they have a marginal benefit, and in some settings, the cost of upper bound computation itself is quite high and can backfire. For example, the running time on *soc-digg* when $k = 5$ is **102.4** s, 218.5 s, 1266.5 s, and OOT for no UB pruning, S-Based, R-Based, and SR-Based UB pruning, respectively. The full results are shown in Table 11 of our technical report [27], based on which we recommend to apply no bounding technique by default. We do notice an exception that S-Based is quite effective on the two largest graphs, especially *web-ClueWeb09* when $k = 4, 5$. Moreover, S-Based does not bring much slowdown in most cases, so it could be safe alternative for large graphs (e.g., $|V| = 10^7$) and worth trying out if OOT happens.

Choice of Branching Method: Section 4 presented two competitive branching methods: partition-based and pivoting-based, where partition-based methods are based on the above-mentioned 3 UB techniques. Table 3 shows a comparison of these branching methods when $k = 2$, 3, 8 and 10, and the full results are shown in Table 12 of our technical report [27]. The baseline binary branching method (where pivot is selected simply based on the degeneracy order) is not competitive and runs OOT most of the time, so is not included in Table 3 (but is shown in Table 12). Among the 3 partition-based methods, there is no clear winner, but S-based method is the most stable and often performs the best for more time-consuming jobs when k is small, so we adopt S-based branching when $k \leq 5$ by default. On the other hand, the pivoting-based branching method by Maple is a clear winner when k is large, so we adopt pivoting-based branching when $k > 5$ by default. This default scheme of U-MkP generally works very well. For example, as Table 3 shows, when $k = 3$, it took 28.7 s on *keller4* when applying S-based branching but 274.0 s (~10× slower) when applying pivoting-based branching. On the other hand, when $k = 10$, it did not finish within the time limit on *socfb-Duke14* when applying S-based branching but finished in only 0.6 s (> 3000× faster) when applying pivoting-based branching. Exceptions exist: the advantage of S-based branching goes beyond $k = 5$ on *dbpedia-link* and *delicious-ui*, so can be an alternative to try when OOT happens with pivot-based branching.

Finally, as we shall show in Tables 5–6, kPlexT [35], which uses an improved binary branching method (important for their worst-case time complexity proof), can be a few orders of magnitude slower than our default setting with U-MkP. Since the only difference in pruning techniques between kPlexT and U-MkP is the branching method, it shows that kPlexT's branching method is not competitive in practice where we rarely care about the worst-case performance.

Choice of Reduction Rules: Now, let us consider the 7 reduction rules in Table 1. Section 3 introduced **Two-Hop** and **CTCP** that are applied to prune each top-level subgraph g_i extended from $v_i \in V$. Table 13 in our technical report [27] shows that disabling **Two-Hop** is disastrous and cause most experiments to run OOT, so it should always be enabled. Surprisingly, Table 14 in our technical report [27] shows that disabling **CTCP** at the top level does not cause much slowdown, indicating that its pruning effect is mostly covered also by other techniques. However, since top-level CTCP has a low overhead and is slightly beneficial in most cases, we still enable it by default.

However, we observed that applying CTCP inside the BB(.) procedure (i.e., in reduce_and_prune(.)), as kPlexS [34] does, is expensive. Specifically, Table 15 in our technical report [27] shows that, when enabling CTCP inside BB(.), the execution time was increased by one to two orders of magnitude for many datasets. Thus, U-MkP disables it in BB(.) by default.

Next, we consider the reduction rules presented in Section 5. We tested the efficiency of those rules by disabling each rule at a time and observing the performance difference. Tables 16–18 in our technical report [27] report the effect of **RR1**–**RR3**, and we can see that they can significantly speed up computation, so we enable them by default. Note that conditions of RR1–RR3 are efficient to check.

Reduction rules **BR1** and **BR2** aim to add some vertices directly to S. We report the effect of BR1 and BR2 in Table 19 of our technical report [27], and to our surprise, we can see that they merely make any difference for almost all datasets and values of k, except for *delicious-ui* where a significant speedup is observed. This shows that their conditions seldom hold to allow pruning, but since they are efficient to check, U-MkP enables them by default.

Reduction rule **UBR1** requires access to the number of common neighbors of every pair of vertices in g for condition checking, as is also required by recursive CTCP, so they are usually used together if enabled in BB(.). However, as we have shown in Table 15 in our technical report [27], the overhead of dynamically maintaining these counts is expensive, so U-MkP disables UBR1 by default along with recursive CTCP. On the other hand, we observed that **UBR2** is highly

Table 5: Execution Time for Small Values of k (Unit: seconds)

Dataset	$k=2$				$k=3$				$k=4$				$k=5$			
	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP
johnson8-4-4	**0.7**	1.7	1.7	1.1	**5.7**	24.0	34.4	6.6	**29.6**	210.9	926.3	239.4	**2.0**	123.0	×	76.1
keller4	25.2	134.6	111.6	**19.7**	**28.9**	2073.1	1251.1	64.0	**1504.8**	×	×	×	**1120.0**	×	×	×
socfb-Duke14	**1.2**	6.4	2.3	129.2	**0.8**	14.4	43.7	×	**1.4**	9.2	30.9	×	**2.6**	21.3	4.8	×
ia-wiki-Talk	0.8	4.1	4.1	**0.7**	**0.6**	2.7	10.1	4.1	**0.4**	0.8	1.3	6.9	**0.7**	0.9	4.1	70.7
soc-buzznet	**358.0**	1577.7	1589.5	×	**189.1**	2744.2	×	×	**153.0**	×	×	×	**1771.0**	×	×	×
soc-LiveMocha	**3.3**	24.6	8.0	8.2	**2.2**	7.8	178.6	19.2	**2.3**	10.5	120.2	120.2	**7.0**	28.1	62.3	×
soc-gowalla	**0.1**	0.1	0.2	1.1	**0.1**	0.1	0.2	0.8	0.1	**0.1**	0.1	2.3	**0.1**	0.3	0.2	8.9
soc-digg	**121.9**	392.2	331.1	×	**49.4**	×	×	×	**33.2**	252.0	×	×	**102.4**	503.7	×	×
sc-ldoor	14.1	**9.9**	11.7	×	24.6	**14.9**	40.8	×	**56.6**	159.9	221.9	×	**71.3**	298.9	572.2	×
soc-youtube-snap	**0.4**	1.3	0.6	1.6	**0.5**	1.4	0.7	2.2	**0.3**	0.4	1.6	2.8	**0.3**	1.0	0.5	10.6
soc-lastfm	**3.6**	8.1	7.9	4.9	**1.9**	6.2	63.9	17.8	**1.5**	6.4	28.8	24.8	**2.2**	7.5	34.1	11.7
soc-orkut	**96.1**	505.2	220.0	×	**79.1**	706.9	603.0	×	**611.5**	×	1047.9	×	×	×	**1473.2**	×
wikipedia-link-en	12.1	**10.7**	65.9	×	12.0	**9.9**	560.5	×	**12.9**	17.6	1485.6	×	**15.3**	44.9	×	1520.1
dbpedia-link	**76.0**	198.4	332.2	×	**52.2**	700.5	×	×	**50.5**	1371.4	×	×	**49.3**	×	×	×
wikipedia-link-en13	**249.0**	321.0	×	×	**290.7**	364.6	×	×	**249.3**	470.5	×	×	**290.9**	552.6	×	×
delicious-ui	**62.1**	109.3	189.6	326.9	**99.0**	441.9	215.1	395.3	**101.2**	1368.2	215.2	394.3	**119.3**	×	×	1452.4
soc-sinaweibo	**141.2**	674.8	512.0	×	**126.3**	×	×	×	**207.3**	×	×	×	**1661.7**	×	×	×
web-ClueWeb09	**43.8**	355.3	224.6	×	**70.5**	×	×	×	**613.2**	×	×	×	×	×	×	×

Table 6: Execution Time for Large Values of k on Real Graphs

Dataset	$k=7$				$k=10$				$k=15$				$k=20$			
	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP
johnson8-4-4	×	×	×	×	**1271.9**	×	×	×	**20.4**	36.3	159.5	401.2	0.0	0.0	0.0	**0.0**
keller4	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	**0.6**	307.8	6.1	×	**0.6**	×	9.0	×	**2.7**	×	18.3	×	**1.7**	×	25.2	×
ia-wiki-Talk	**0.5**	4.7	0.6	×	**1.5**	×	2.5	×	**3.6**	×	25.5	×	**23.7**	×	×	×
soc-buzznet	**76.1**	×	×	×	**150.3**	×	×	×	**520.7**	×	×	×	**701.7**	×	×	×
soc-LiveMocha	**4.1**	495.9	365.5	×	**2.8**	×	24.9	×	**0.9**	×	1.9	×	**3.2**	×	335.7	×
soc-gowalla	**0.1**	0.4	0.1	15.1	**0.1**	66.5	0.1	×	**0.2**	×	0.6	×	**0.4**	×	4.5	125.0
soc-digg	**54.5**	1529.0	1683.5	×	**35.1**	×	608.2	×	**7.9**	×	28.1	×	**7.4**	×	26.5	×
sc-ldoor	**11.3**	29.8	16.3	×	8.6	**5.7**	5.7	×	7.3	**3.0**	4.3	×	10.1	**5.2**	6.5	×
soc-youtube-snap	**0.3**	1.5	0.5	186.4	**0.5**	143.8	0.5	×	**1.2**	×	3.6	×	**4.6**	×	1347.6	×
soc-lastfm	**3.5**	168.8	7.0	534.1	1.6	2014.8	**1.3**	2958.2	**5.6**	×	6.7	×	**21.3**	×	824.1	×
soc-orkut	**68.8**	×	1200.9	×	**46.1**	659.4	195.5	×	**33.6**	519.9	142.5	×	**37.2**	×	98.3	×
wikipedia-link-en	**247.1**	580.8	×	×	**12.0**	179.5	1639.1	39.1	9.9	**8.4**	18.7	101.7	**7.2**	8.9	15.3	14.4
dbpedia-link	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	**304.4**	652.2	×	×	**292.3**	×	×	×	**297.2**	×	×	×	**277.6**	×	×	×
delicious-ui	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

Table 4: Effect of Reduction Rule UBR2

Dataset	$k=3$		$k=4$		$k=8$		$k=10$	
	w/ UBR2	w/o UBR2	w/ UBR2	w/o UBR2	w/ UBR2	w/o UBR2	w/ UBR2	w/o UBR2
johnson8-4-4	**5.7**	7.1	**29.6**	74.4	×	×	**1271.9**	×
keller4	**28.9**	43.5	**1504.8**	×	×	×	×	×
socfb-Duke14	**0.8**	3.7	**1.4**	14.1	**0.6**	×	**0.6**	×
ia-wiki-Talk	**0.6**	1.4	**0.4**	1.4	**0.9**	×	**1.5**	×
soc-buzznet	**189.1**	×	**153.0**	1701.1	**75.4**	×	**150.3**	×
soc-LiveMocha	**2.2**	6.7	**2.3**	7.5	**4.1**	×	**2.8**	×
soc-gowalla	0.1	**0.1**	0.1	**0.1**	0.1	**0.1**	**0.1**	908.4
soc-digg	**49.4**	558.3	**33.2**	918.2	**49.1**	×	**35.1**	×
sc-ldoor	**24.6**	25.2	**56.6**	64.6	**7.1**	7.3	8.6	**8.4**
soc-youtube-snap	**0.5**	0.6	**0.3**	0.4	**0.4**	×	**0.5**	×
soc-lastfm	**1.9**	7.7	**1.5**	5.3	**1.1**	×	**1.6**	×
soc-orkut	**79.1**	809.8	**611.5**	×	**49.9**	×	**46.1**	×
wikipedia-link-en	**12.0**	16.1	**12.9**	468.6	**69.0**	×	**12.0**	×
dbpedia-link	**52.2**	68.6	**50.5**	50.8	×	×	×	×
wikipedia-link-en13	290.7	**229.0**	249.3	**217.6**	284.1	**247.2**	292.3	**284.5**
delicious-ui	**99.0**	168.2	**101.2**	171.8	×	×	×	×
soc-sinaweibo	**126.3**	×	**207.3**	×	×	×	×	×
web-ClueWeb09	**70.5**	×	**613.2**	×	×	×	×	×

effective on many datasets. Table 4 shows a comparison between our default U-MkP that enables UBR2 and the version that disables it, when $k = 3$, 4, 8 and 10. The full results are shown in Table 20 of our technical report [27]. We can see that enabling UBR2 can significantly speed up computation. For example, on *socfb-Duke14*, U-MkP w/o UBR2 cannot finish within 1800 s when $k = 8$, but with UBR2, U-MkP finishes in 0.6 s, so the speedup is 3000×.

Comparison of MkP Algorithms. Recall from the algorithm dominance graph shown in Figure 4 that among existing MkP algorithms, only kPlexT [35], Maple [53] and DiseMKP [44] are competitive baselines (Seesaw has no code released). We, therefore, select these three baselines to compare with our U-MkP framework with the previously mentioned default configuration of pruning rules. Table 5 shows the running time of the compared algorithms when $k = 2$, 3, 4 and 5 (so U-MkP adopts S-based branching), where we can see that U-MkP performs the best in vast majority of the datasets and often beats the second best by many times to over an order of magnitude for time-consuming jobs. Even when U-MkP is not the best, it is very close to the best (except for the only case of soc-orkut when $k = 5$ where Maple wins but is not too far from OOT). For example, U-MkP is 38.5× faster than DiseMKP (the second best) on *johnson8-4-4* when $k = 5$, and only 1.28× slower than the winner algorithm for *keller4* when $k = 2$. Among the other algorithms, there is no clear second best, but kPlexT seems to win on more datasets. However, due to its ineffective binary branching method, kPlexT is still much slower than U-MkP. Maple is slow in most time, showing that its pivot-based branching method is not effective for small k values. Finally, although DiseMKP uses S-based branching, it does not utilize effective reduction rules like UBR2, so is not competitive.

Table 6 shows the running time of the compared algorithms when $k = 7$, 10, 15 and 20 (so U-MkP adopts pivot-based branching), where we can see that U-MkP still performs the best and often beats the second best by a few orders of magnitude. Maple is the only other algorithm that can properly handle large k values, thanks to its pivot-based branching. However, it can still be a few orders of magnitude slower than U-MkP since it does not utilize effective reduction rules like UBR2. Among the other two algorithms, DiseMKP simply cannot handle large values of k and runs OOT on most datasets; while even though kPlexT is the latest algorithm that proposed UBR2, it generally cannot handle the cases when $k = 15, 20$ due to its suboptimal approach for binary branching.

While Tables 5 and 6 cover only a subset of k on 18 datasets, Table 21 of our technical report [27] shows results on all 26 datasets for all values of k, where we can see that U-MkP is a clear winner.

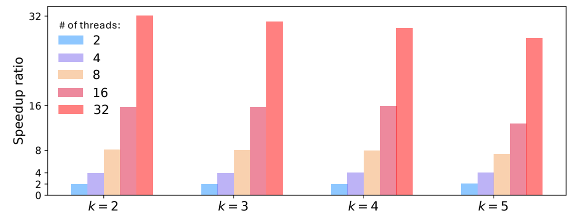
Performance of MkP Variants. Section 6 presented a two-phase approach to compute all MkPs as well as the densest MkP. Table 7 reports the results of running this variant for $k = 5$ on 18 graphs (full results on all our 26 graphs and all k values are shown in Table 22

Table 8: Running Time of Parallel U-MkP on Representative Datasets with Varying Number of Threads

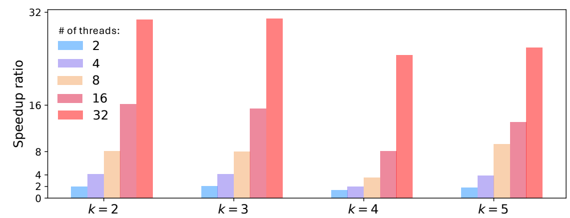
Dataset	$k=2$						$k=3$						$k=4$						$k=5$					
	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32
keller4	27.6	13.9	6.7	3.4	1.7	0.9	247.0	120.3	60.7	30.7	16.0	8.0	1480.2	1028.2	728.3	423.0	182.1	60.1	1120.2	620.4	287.5	120.3	85.3	43.2
soc-buzznet	301.6	150.5	74.9	36.7	19.1	9.4	145.7	73.3	36.5	18.0	9.2	4.7	143.4	71.1	35.0	17.9	9.0	4.8	1720.7	824.2	421.2	231.4	134.2	61.2
soc-LiveMocha	2.4	1.2	0.6	0.3	0.2	0.1	1.5	0.7	0.4	0.2	0.1	0.1	3.7	1.7	0.8	0.4	0.2	0.2	18.8	10.1	5.5	3.0	1.5	1.5
soc-digg	91.6	45.9	22.2	11.5	5.7	2.9	29.5	15.2	7.6	3.7	1.9	1.0	42.5	16.0	6.6	3.0	1.3	0.9	100.2	54.3	27.3	14.9	8.2	4.3
sc-ldoor	5.9	3.0	1.6	0.8	0.5	0.7	24.2	12.3	5.8	3.7	2.8	3.2	48.2	24.3	15.2	8.9	4.3	3.2	72.8	38.2	20.9	10.3	6.7	3.2
delicious-ui	18.8	9.9	5.3	2.7	1.4	0.6	95.7	48.4	25.4	13.3	6.8	2.8	345.2	178.7	94.8	49.5	24.9	10.2	1282.2	834.2	612.1	406.9	205.2	88.4
soc-sinaweibo	32.0	18.5	10.3	5.3	3.6	2.6	30.2	18.1	10.1	5.2	3.7	2.6	60.1	34.1	19.0	8.7	6.2	5.1	1812.2	918.3	345.2	121.8	74.3	45.7
web-ClueWeb09	9.2	5.9	3.2	1.7	1.2	1.0	16.2	8.8	4.8	2.4	1.9	1.6	57.2	29.4	15.8	7.8	5.7	4.8	52.2	28.9	17.4	8.4	4.9	3.7

Table 7: Finding All MkPs and Densest MkP

Dataset	$k=5$			
	Time	#MkPs	$\lvert E_{\text{Phase 1}}\rvert$	$\lvert E_{\text{Densest}}\rvert$
johnson8-4-4	9	226	644	644
keller4	×	×	×	×
socfb-Duke14	12	75	2166	2172
ia-wiki-Talk	3	95	530	534
soc-buzznet	×	×	×	×
soc-LiveMocha	15	24	664	676
soc-gowalla	0	40	972	980
soc-digg	90	15	4976	4986
sc-ldoor	×	×	×	×
soc-youtube-snap	1	8	584	590
soc-lastfm	3	2	622	622
soc-orkut	×	×	×	×
wikipedia-link-en	31	24,837	1,219,880	1,219,880
dbpedia-link	91	2	2776	2776
wikipedia-link-en13	67	1	186170	186,170
delicious-ui	882	23	110	132
soc-sinaweibo	×	×	×	×
web-ClueWeb09	×	×	×	×



(a) *soc-buzznet*



(b) *keller4*

Figure 10: Speedup Ratio of Parallel U-MkP

in our technical report [27]), including (1) the running time, (2) the number of MkPs found, (3) the number of vertices and the number of edges in the MkP found by Phase 1, and (4) the number of edges of the densest MkP found in Phase 2. We can see that some graphs have many MkPs, so finding one of them is not sufficient to catch all dense communities. While *wikipedia-link-en* has 24,837 MkPs so are not selective, many graphs have several to tens of MkPs which are reasonably selective and interesting for users to examine all these structures. We can also see that MkPs found in Phase 1 are not the densest on many graphs. For example, on *soc-LiveMocha*, Phase 1 finds an MkP with 664 edges but the densest has 676 edges.

Performance of U-MkP Parallelization. We implemented the parallel version of U-MkP based on the description in Section 6. Table 8 reports the running time on 8 datasets when using 1, 2, 4, 8, 16 and 32 threads, respectively, where we use the default timeout threshold $\tau_{time} = 0.1$ ms for load balancing (which consistently works well). We chose representative datasets where the job time in serial execution is at least 10 seconds when $k = 5$, so that it is worth for parallelization. As Table 8 shows, our parallel algorithm is efficient and scales up well with the number of CPU cores on all the datasets. We also show the speedup ratio on 2 representative datasets in Figure 10. We can see that our parallelization can achieve a speedup ratio of up to 28.1× with 32 threads. The complete results for all datasets can be found in Table 23 of our technical report [27].

8 RELATED WORK

This paper has surveyed the algorithms for **maximum** k-plex finding as summarized in Table 1. There are also works for finding **maximal** k-plexes (i.e., those without a supergraph that is also a k-plex) [31, 36, 38, 39, 54, 64], often with a size threshold q to find only those with at least q vertices. Among them, ListPlex [54] proposes to create initial tasks each of which consists of a top-level vertex v_i and a subset of its two-hop neighbors, and extend these vertices with candidates from v_i's one-hop neighbors. This approach not only reduces the worst-case time complexity [54], but is also efficient and hence adopted by a later work [36]. The authors of ListPlex aim to apply this idea to the MkP problem by proposing Maple [53], but we found that their implementation does not follow their paper description. In general, finding maximal k-plexes is more expensive than MkP since the size lower bound remains at q rather than $|P| + 1$, so many pruning techniques are not as effective; finding maximal k-plexes also needs to avoid emitting non-maximal results with the help of an exclusion set following the Bron-Kerbosch algorithm [32].

9 CONCLUSION

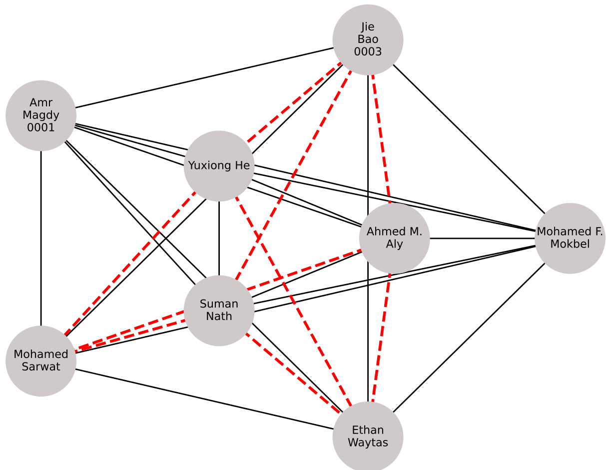
We proposed U-MkP, a framework for finding a maximum k-plex that can be adapted to find all maximum k-plexes or the one with the most edges. Our framework can integrate the various pruning techniques from nine recent algorithms including kPlexT, Maple, Seesaw, DiseMKP, kPlexS, KpLeX, Maplex, BnB and BS, which were summarized into three categories: those for (1) branching, (2) upper bounding, and (3) reduction during subgraph exploration. We found that different pruning techniques can have drastically different performance impacts, and obtained interesting new insights about these techniques not studied by prior works. Moreover, we found that there exists a configuration of the techniques dependent on k that leads to the best performance in vast majority of the time, which we recommended as the default configuration of U-MkP.

ACKNOWLEDGMENTS

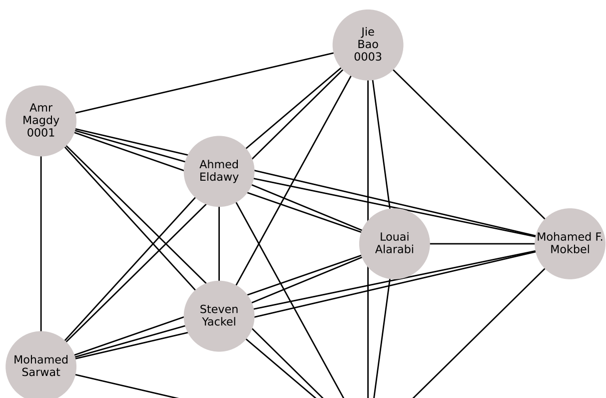
This work was supported by DOE ECRP Award DE-SC0025228, NSF OAC-2414474, NSF OAC-2414185 and 2024–2025 Luddy Faculty Fellow Award from Indiana University Bloomington.

REFERENCES

[1] brock200-2. https://networkrepository.com/brock200-2.php.
[2] dbpedia-link. http://konect.cc/networks/dbpedia-link/.
[3] delicious-ui. http://konect.cc/networks/delicious-ui/.
[4] hamming6-2. https://networkrepository.com/hamming6-2.php.
[5] ia-wiki-talk. https://networkrepository.com/ia-wiki-Talk.php.
[6] johnson8-4-4. https://networkrepository.com/johnson8-4-4.php.
[7] keller4. https://networkrepository.com/keller4.php.
[8] p-hat500-1. https://networkrepository.com/p-hat500-1.php.
[9] sc-ldoor. https://networkrepository.com/sc-ldoor.php.
[10] sc-msdoor. https://networkrepository.com/sc-msdoor.php.
[11] soc-buzznet. https://networkrepository.com/soc-buzznet.php.
[12] soc-digg. https://networkrepository.com/soc-digg.php.
[13] soc-gowalla. https://networkrepository.com/soc-gowalla.php.
[14] soc-lastfm. https://networkrepository.com/soc-lastfm.php.
[15] soc-livemocha. https://networkrepository.com/soc-LiveMocha.php.
[16] soc-orkut. https://networkrepository.com/soc-orkut.php.
[17] soc-pokec. https://networkrepository.com/soc-pokec.php.
[18] soc-sinaweibo. https://networkrepository.com/soc-sinaweibo.php.
[19] soc-youtube. https://networkrepository.com/soc-youtube.php.
[20] soc-youtube-snap. https://networkrepository.com/soc-youtube-snap.php.
[21] socfb-a-anon. https://networkrepository.com/socfb-A-anon.php.
[22] socfb-b-anon. https://networkrepository.com/socfb-B-anon.php.
[23] socfb-duke14. https://networkrepository.com/socfb-Duke14.php.
[24] web-clueweb09. https://networkrepository.com/web-ClueWeb09.php.
[25] wikipedia-link-en. http://konect.cc/networks/wikipedia_link_en/.
[26] wikipedia-link-en13. https://networkrepository.com/web-wikipedia-link-en13-all.php.
[27] Full Technical Report. https://github.com/akhlaqueak/MKP-Study/Technical_Report.pdf, 2024.
[28] G. D. Bader and C. W. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC bioinformatics*, 4(1):2, 2003.
[29] B. Balasundaram, S. Butenko, and I. V. Hicks. Clique relaxations in social network analysis: The maximum *k*-plex problem. *Oper. Res.*, 59(1):133–142, 2011.
[30] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.
[31] D. Berlowitz, S. Cohen, and B. Kimelfeld. Efficient enumeration of maximal k-plexes. In *SIGMOD*, pages 431–444. ACM, 2015.
[32] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.
[33] D. Bu, Y. Zhao, L. Cai, H. Xue, X. Zhu, H. Lu, J. Zhang, S. Sun, L. Ling, N. Zhang, et al. Topological structure analysis of the protein-protein interaction network in budding yeast. *Nucleic acids research*, 31(9):2443–2450, 2003.
[34] L. Chang, M. Xu, and D. Strash. Efficient maximum k-plex computation over large sparse graphs. *Proc. VLDB Endow.*, 16(2):127–139, 2022.
[35] L. Chang and K. Yao. Maximum k-plex computation: Theory and practice. *Proc. ACM Manag. Data*, 2(1):63:1–63:26, 2024.
[36] Q. Cheng, D. Yan, T. Wu, L. Yuan, J. Cheng, Z. Huang, and Y. Zhou. Efficient enumeration of large maximal k-plexes. In *EDBT*, pages 53–65. OpenProceedings.org, 2025.
[37] A. Conte, D. Firmani, C. Mordente, M. Patrignani, and R. Torlone. Fast enumeration of large k-plexes. In *KDD*, pages 115–124. ACM, 2017.
[38] A. Conte, T. D. Matteis, D. D. Sensi, R. Grossi, A. Marino, and L. Versari. D2K: scalable community detection in massive networks via small-diameter k-plexes. In *KDD*, pages 1272–1281. ACM, 2018.
[39] Q. Dai, R. Li, H. Qin, M. Liao, and G. Wang. Scaling up maximal *k*-plex enumeration. In *CIKM*, pages 345–354. ACM, 2022.
[40] J. Gao, J. Chen, M. Yin, R. Chen, and Y. Wang. An exact algorithm for maximum k-plexes in massive graphs. In *IJCAI*, pages 1449–1455. ijcai.org, 2018.
[41] G. Guo, D. Yan, M. T. Özsu, Z. Jiang, and J. Khalil. Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach. *Proc. VLDB Endow.*, 14(4):573–585, 2020.
[42] J. Hopcroft, O. Khan, B. Kulis, and B. Selman. Tracking evolving communities in large linked networks. *Proceedings of the National Academy of Sciences*, 101(suppl 1):5249–5253, 2004.
[43] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics*, 21(suppl_1):i213–i221, 2005.
[44] H. Jiang, F. Xu, Z. Zheng, B. Wang, and W. Zhou. A refined upper bound and inprocessing for the maximum k-plex problem. In *IJCAI*, pages 5613–5621. ijcai.org, 2023.
[45] H. Jiang, D. Zhu, Z. Xie, S. Yao, and Z. Fu. A new upper bound based on vertex partitioning for the maximum k-plex problem. In *IJCAI*, pages 1689–1696. ijcai.org, 2021.
[46] J. Khalil, D. Yan, G. Guo, and L. Yuan. Parallel mining of large maximal quasi-cliques. *VLDB J.*, 31(4):649–674, 2022.
[47] J. M. Lewis and M. Yannakakis. The node-deletion problem for hereditary properties is np-complete. *J. Comput. Syst. Sci.*, 20(2):219–230, 1980.
[48] J. Li, X. Wang, and Y. Cui. Uncovering the overlapping community structure of complex networks by maximal cliques. *Physica A: Statistical Mechanics and its Applications*, 415:398–406, 2014.
[49] J. Pattillo, N. Youssef, and S. Butenko. On clique relaxation models in network analysis. *European Journal of Operational Research*, 226(1):9–18, 2013.
[50] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.
[51] S. B. Seidman and B. L. Foster. A graph-theoretic generalization of the clique concept. *Journal of Mathematical Sociology*, 6(1):139–154, 1978.
[52] D. Ucar, S. Asur, U. Catalyurek, and S. Parthasarathy. Improving functional modularity in protein-protein interactions graphs using hub-induced subgraphs. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 371–382. Springer, 2006.
[53] Z. Wang, Y. Zhou, C. Luo, and M. Xiao. A fast maximum k-plex algorithm parameterized by the degeneracy gap. In *IJCAI*, pages 5648–5656. ijcai.org, 2023.
[54] Z. Wang, Y. Zhou, M. Xiao, and B. Khoussainov. Listing maximal k-plexes in large real-world graphs. In *WWW*, pages 1517–1527. ACM, 2022.
[55] M. Xiao, W. Lin, Y. Dai, and Y. Zeng. A fast algorithm to compute maximum *k*-plexes in social network analysis. In *AAAI*, pages 919–925. AAAI Press, 2017.
[56] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, W. Ku, and J. C. S. Lui. G-thinker: A distributed framework for mining subgraphs in a big graph. In *ICDE*, pages 1369–1380. IEEE, 2020.
[57] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, J. C. S. Lui, and W. Tan. T-thinker: a task-centric distributed framework for compute-intensive divide-and-conquer algorithms. In J. K. Hollingsworth and I. Keidar, editors, *PPoPP*, pages 411–412. ACM, 2019.
[58] D. Yan, G. Guo, J. Khalil, M. T. Özsu, W. Ku, and J. C. S. Lui. G-thinker: a general distributed framework for finding qualified subgraphs in a big graph with load balancing. *VLDB J.*, 31(2):287–320, 2022.
[59] D. Yan, W. Qu, G. Guo, X. Wang, and Y. Zhou. Prefixfpm: a parallel framework for general-purpose mining of frequent and closed patterns. *VLDB J.*, 31(2):253–286, 2022.
[60] L. Yuan, D. Yan, J. Han, A. Ahmad, Y. Zhou, and Z. Jiang. Faster depth-first subgraph matching on gpus. In *ICDE*, pages 3151–3163. IEEE, 2024.
[61] L. Yuan, D. Yan, W. Qu, S. Adhikari, J. Khalil, C. Long, and X. Wang. T-FSM: A task-based system for massively parallel frequent subgraph pattern mining from a big graph. *Proc. ACM Manag. Data*, 1(1):74:1–74:26, 2023.
[62] J. Zheng, M. Jin, Y. Jin, and K. He. Two new upper bounds for the maximum k-plex problem. *CoRR*, abs/2301.07300, 2023.
[63] Y. Zhou, S. Hu, M. Xiao, and Z. Fu. Improving maximum k-plex solver via second-order reduction and graph color bounding. In *AAAI*, pages 12453–12460. AAAI Press, 2021.
[64] Y. Zhou, J. Xu, Z. Guo, M. Xiao, and Y. Jin. Enumerating maximal *k*-plexes with worst-case time guarantee. In *AAAI*, pages 2442–2449. AAAI Press, 2020.



((a)) An MkP



((b)) Another denser MkP

Figure 11: Case Study of MkP Variants

A A CASE STUDY WITH TWO MkPS

We obtain all papers from ICDE 2014 and their authors from DBLP to create a co-authorship graph with 442 authors and 860 edges, and then find maximum 4-plexes on it. Figure 11(a) shows an MkP that would be returned if we only request for finding one MkP, where the red dashed lines denote missing edges. However, it will miss the other MkP shown in Figure 11(b) which is clearly denser (with more edges). Moreover, the two MkPs are different by the three authors shown in the inner circles. This shows the necessity of All-MkP and Densest-MkP to avoid missing important communities.

B k-CORE DECOMPOSITION

The graph degeneracy can be computed by the process of k-core decomposition [50]. Formally, the k-core of a graph $G = (V, E)$ is the largest induced subgraph with minimum degree k (i.e., where every vertex has degree $\geq k$). For example, Fig. 12 shows the 1-core,

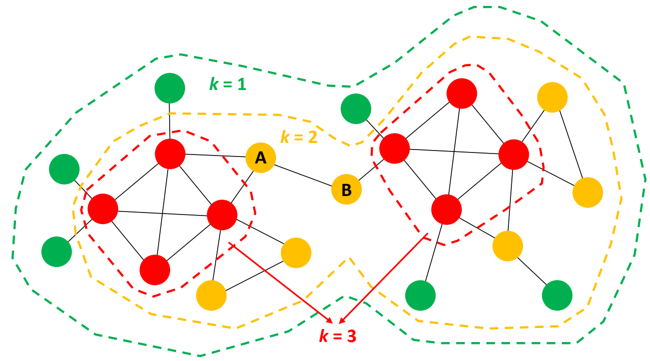


Figure 12: Illustration of k-Cores

2-core and 3-core of a graph. Specifically, the 2-core contains all yellow and red nodes in the yellow dashed contour, since it is the largest induced subgraph where every vertex has degree ≥ 2, as any green vertex has degree 1. Note that even though vertex A has degree 3, it is not in 3-core since its neighbor B has degree 2 so cannot be in 3-core, hence A has at most 2 neighbors in 3-core. k-core decomposition finds the core number of every $v \in V$, denoted by $core(v)$, which is the largest value of k that v belongs to a k-core. For example, $core(A) = 2$ in Fig. 12 since A is in 2-core but not 3-core.

The process of k-core decomposition [50] can be computed by a peeling algorithm that repeatedly removes a vertex with the minimum current degree at a time [30]. For the graph G in Fig. 12, green vertices will first be removed, followed by yellow ones, and finally the red ones, after which no vertex is remaining so we have $D(G) = 3$. Accordingly, the algorithm also generates the degeneracy ordering of vertices in G where green vertices go first, followed by yellow ones and then red ones.

C PROOF OF THEOREM 2.5

For any $u \in P'$, we have $u \in P$. Since P is a k-plex, $\overline{d_P}(u) = |\overline{N_P}(u)| \leq k$.

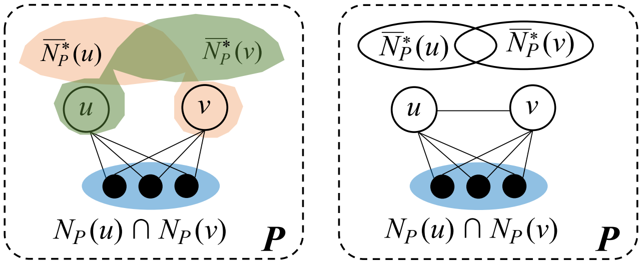
Since $\overline{N_{P'}}(u) \subseteq \overline{N_P}(u)$, we have $\overline{d_{P'}}(u) = |\overline{N_{P'}}(u)| \leq |\overline{N_P}(u)| \leq k$, so P' is also a k-plex. □

D SECOND-ORDER PROPERTY OF k-PLEX

The theorem below states the second-order property of two vertices in a k-plex with size constraint:

THEOREM D.1. *Let P be a k-plex with $|P| \geq q$. Then, for any two vertices $u, v \in P$, we have (i) if $(u, v) \notin E$, $|N_P(u) \cap N_P(v)| \geq q - 2k + 2$, (ii) otherwise, $|N_P(u) \cap N_P(v)| \geq q - 2k$.*

PROOF. This can be seen from Figure 13. Let us first define $\overline{N_P^*}(v) = \overline{N_P}(v) - \{v\}$, so $|\overline{N_P^*}(v)| \leq k - 1$. In Case (i) where $(u, v) \notin E$, any vertex $w \in P$ can only fall in the following 3 scenarios: (1) $w \in \overline{N_P^*}(u)$, (2) $w \in \overline{N_P^*}(v)$, and (3) $w \in N_P(u) \cap N_P(v)$. Note



(a) $(u, v) \notin E$ (b) $(u, v) \in E$

Figure 13: Second-Order Pruning

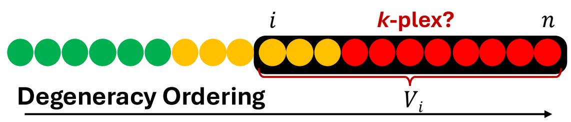


Figure 14: Illustration of an Iteration of Algorithm 6

that w may be in both (1) and (2). Thus, we have:

$$\begin{aligned} |P| &= |N_P(u) \cap N_P(v)| + |\overline{N_P^*}(u) \cup \overline{N_P^*}(v)| \\ &\leq |N_P(u) \cap N_P(v)| + |\overline{N_P^*}(u)| + |\overline{N_P^*}(v)| \\ &\leq |N_P(u) \cap N_P(v)| + 2(k-1) \\ &= |N_P(u) \cap N_P(v)| + 2k - 2, \end{aligned}$$

so $|N_P(u) \cap N_P(v)| \geq |P| - 2k + 2 \geq q - 2k + 2$.

In Case (ii) where $(u, v) \in E$, any vertex $w \in P$ can only be in one of the following 4 scenarios: (1) $w = u$, (2) $w = v$, (3) $w \in N_P(u) \cap N_P(v)$, and (4) $w \in \overline{N_P}(u) \cup \overline{N_P}(v)$, therefore:

$$\begin{aligned} |P| &= 2 + |N_P(u) \cap N_P(v)| + |\overline{N_P^*}(u) \cup \overline{N_P^*}(v)| \\ &\leq 2 + |N_P(u) \cap N_P(v)| + |\overline{N_P^*}(u)| + |\overline{N_P^*}(v)| \\ &\leq 2 + |N_P(u) \cap N_P(v)| + 2(k-1) \\ &= |N_P(u) \cap N_P(v)| + 2k, \end{aligned}$$

so $|N_P(u) \cap N_P(v)| \geq |P| - 2k \geq q - 2k$. □

Note that by setting $q = 2k - 1$, Case (i) gives $|N_P(u) \cap N_P(v)| \geq (2k-1) - 2k + 2 = 1$, i.e., for any two vertices $u, v \in P$ that are not mutual neighbors, they must share a neighbor and is thus within 2 hops, which proves Theorem 2.7.

E FINDING A LARGE INITIAL k-PLEX

Before subgraph exploration, we first aim to find a large (though may not be maximum) k-plex P as well as an upper bound ub on the size of any MkP in G, so that (1) if $|P| = ub$, then P is already an MkP and can be directly returned, otherwise (2) we can still prune those search branches that cannot lead to a larger k-plex with size at least $(|P| + 1)$. We follow kPlexS [34] to compute $\langle P, ub \rangle$ while running the peeling algorithm in linear time.

Algorithm 6 shows this process, where a vertex v with the minimum current degree is peeled at a time (see Line 4) and removed from G (see Line 9). Lines 5–6 are to find a large k-plex P, while Lines 7–8 are to compute ub, and we explain them next.

Algorithm 6: Finding Large Initial k-Plex and Upper Bound

1 **function** find_init($G = (V, E)$)
2   $P \leftarrow \emptyset$, $ub \leftarrow 0$
3   **while** $V \neq \emptyset$ **do**
4     $v \leftarrow$ A vertex with minimum $d_G(.)$
5   **if** $P = \emptyset$ **and** $d_G(v) + k \geq |V|$ **then**
6       $P \leftarrow V$
7   **if** $\min\{d_G(v) + k, |V|\} > ub$ **then**
8       $ub \leftarrow \min\{d_G(v) + k, |V|\}$
9   Remove v from G
10   **return** P, ub

Algorithm 7: Basic R-Based Partition Computing

1 **function** get_R_part(S, R, g)
2   $I \leftarrow \emptyset$
3   **for each** $v \in R$ **do**
4   **if** $N(v) \cap I = \emptyset$ **then** $I \leftarrow I \cup \{v\}$
5   $ub \leftarrow \min\{|I|, k\}$
6   **return** I, ub

Figure 14 illustrates the i^{th} iteration of the while-loop in Algorithm 6, assuming that the degeneracy ordering is given by $[v_1, v_2, \ldots, v_n]$. Specifically, after v_i is fetched by Line 4, vertices $v_1, \ldots, v_{i-1}$ have already been removed from V by Line 9 in previous iterations, so V in iteration i becomes $V_i = [v_i, \ldots, v_n]$. Now, if Line 5 finds that $d_{V_i}(v_i) + k \geq |V_i|$, then we have that $G[V_i]$ must be a k-plex, so it is assigned to P (Line 6) to return (Line 10). This is because based on the definition of degeneracy ordering, for any $j \geq i$, we have $d_{V_j}(v_j) \geq d_{V_i}(v_i) \geq |V_i| - k$; and since $d_{V_i}(v_j) \geq d_{V_j}(v_j)$, we have $d_{V_i}(v_j) \geq |V_i| - k$ for all $v_j \in V_i$, so $G[V_i]$ is a k-plex.

As for Lines 7–8, we have the following theorem:

THEOREM E.1. *Any k-plex P with $v \in P$ has $|P| \leq d_G(v) + k$.*

PROOF. We show this by contradiction. Assume that $|P| > d_G(v) + k$, then $|P| - k > d_G(v)$; however, since $v \in P$ and P is a k-plex, we have $d_G(v) \geq |P| - k$, leading to a contradiction. □

Let us denote P_{max} as an MkP. So, in each iteration i, Line 7 computes an upper bound $ub_i = \min\{d_{V_i}(G) + k, |V_i|\}$ for the case when $v_i \in P_{max}$ but $v_j \notin P_{max}$ for all $j < i$. Since P_{max} must exist in one of all the cases, we have $|P_{max}| \leq \max_{v_i \in V}\{ub_i\} = ub$, which is computed by Lines 7–8 and returned by Line 10.

F BASIC R-BASED PARTITION COMPUTING

Algorithm 7 shows the pseudocode of Maplex [63] to compute a partition and the largest number of its vertices that can be added to S after which S can still be extended into a k-plex. In this algorithm. Lines 2–4 computes I as a maximal independent set of R.

Note that our current work actually uses an improved version of get_R_part(.) by Seesaw as shown in Algorithm 2 of [62] rather than this simple version.

Algorithm 8: S-Based Partitioning of Candidate Set R

1 # Initializing auxiliary structure Π
2 $\Pi \leftarrow \emptyset$
3 **for each** $v_i \in S$ **do**
4     $\pi_i \leftarrow R \setminus N(v_i)$
5     $\Pi \leftarrow \Pi \cup \{\pi_i\}$
6 # The S-based strategy to obtain the next partition
7 **function** get_S_part(S, R, g, Π)
8     $\pi^* \leftarrow \emptyset$, $ub^* \leftarrow 0$, $dise^* \leftarrow 0$
9   **for each** $\pi_i \in \Pi$ **and** $\pi_i \neq \emptyset$ **do**
10         $ub_i \leftarrow \min\{|\pi_i|, \delta_S(v_i)\}$
11         $dise \leftarrow |\pi_i|/ub_i$
12   **if** $dise > dise^*$ **then**
13             $\pi^* \leftarrow \pi_i$, $ub^* \leftarrow ub_i$, $dise^* \leftarrow dise$
14   **if** $\pi^* \neq \emptyset$ **then**
15         $R \leftarrow R \setminus \pi^*$
16   **for each** $\pi_i \in \Pi$ **do** $\pi_i \leftarrow \pi_i - \pi^*$
17   **return** π^*, ub^*

G S-BASED PARTITION COMPUTING

Algorithm 8 shows the pseudocode to compute an S-based partition and the largest number of its vertices that can be added to S after which S can still be extended into a k-plex. In this algorithm. Lines 2–5 prepares the set Π of non-neighbors of vertices in S. It is used throughout the calls of get_S_part(.) to obtain partitions.

Also note that when the partition with the highest dise score is picked as π^*, Lines 13–15 need to remove its vertices from all the remaining sets in Π (those already picked before will already have $\pi_i = \emptyset$ anyway due to the vertex removal in the previous calls).

H COMPUTING SR-BASED BRANCHING SET

Algorithm 9 shows the pseudocode to compute the branching set B. Specifically, Line 2 first initializes B as R so that later, partitions are taken from B (into R') to shrink B. This is conceptual since our implementation basically reuses the space of R to keep both R' and B using the dual-array structure to be introduced in Appendix I with Figure 15, where moving a vertex from B to R' takes only $O(1)$ time.

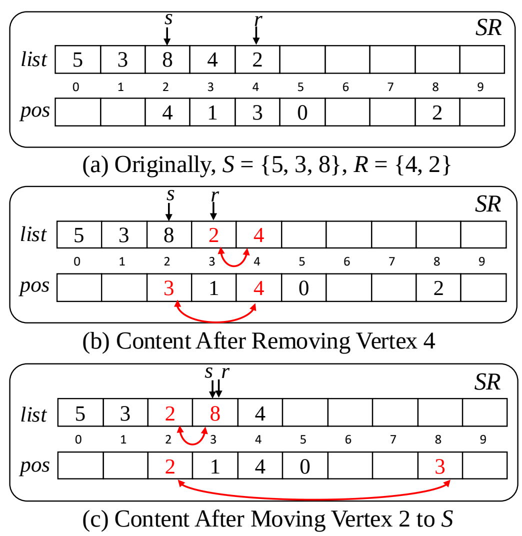
Line 2 also initializes the remaining quota $beta$, and Line 3 computes the auxiliary set required by S-based strategy. The while-loop from Line 4 then keeps obtaining one partition in each iteration. Specifically, Lines 6–9 finds the S-based partition with the highest dise score as the chosen partition (but without considering any candidate partition with $ub_i > \beta$, as shown in Line 8). Lines 10–13 then finds the R-based partition and if $ub_i > \beta$ and it has a higher dise score, then it is used instead as the chosen partition. Line 14–16 then decides if the chosen partition is still within budget. If so, a new iteration starts, while otherwise, the while-loop exits and the control goes to Line 18 to move more vertices from B to R' to reach the allowed quota.

I INCREMENTAL SET MAINTENANCE

Recall that each of our branch-and-bound (BB) algorithm variants expands $\langle S, R\rangle$ by recursively calling BB itself over $\langle S', R'\rangle$ where

Algorithm 9: SR-Based Candidate Set Partitioning

1 **function** partition(S, R, g)
2     $B \leftarrow R$, $\beta \leftarrow |P| - |S|$
3   prepare $\pi_v = R - N(v)$ for all $v \in S$; $\Pi = \bigcup_{v \in S} \pi_v$
4   **while** $B \neq \emptyset$ **and** $\beta > 0$ **do**
5         $\pi^* \leftarrow \emptyset$, $ub^* = 0$, $dise^* \leftarrow 0$
6   **for each** $\pi_i \in \Pi$ **and** $\pi_i \neq \emptyset$ **do**
7             $ub \leftarrow \min\{|\pi_i|, \delta_S(v_i)\}$, $dise \leftarrow |\pi_i|/ub$
8   **if** $ub \leq \beta$ **and** $dise > dise^*$ **then**
9                 $\pi^* \leftarrow \pi_i$, $ub^* \leftarrow ub$, $dise^* \leftarrow dise$
10         $I, ub \leftarrow$ get_R_part(S, B, g)
11   **if** $ub \leq \beta$ **then**
12   **if** $|I|/ub > dise^*$ **or** $(|I|/ub = dise^*$ **and** $|I| > |\pi^*|)$ **then**
13                 $\pi^* \leftarrow I$, $ub^* \leftarrow ub$
14   **if** $ub^* \leq \beta$ **then**
15             $B \leftarrow B - \pi^*$, $\beta \leftarrow \beta - ub^*$
16   **else break**
17   **for each** $\pi_i \in \Pi$ **do** $\pi_i \leftarrow \pi_i - \pi^*$
18   **if** $\beta > 0$ **then** remove $\min\{\beta, |B|\}$ vertices from B
19   **return** B



(a) Originally, $S = \{5, 3, 8\}$, $R = \{4, 2\}$

(b) Content After Removing Vertex 4

(c) Content After Moving Vertex 2 to S

Figure 15: Dual-Array Data Structure for Set Maintenance

$S' = S \cup \{v_i\}$ is the extended set, and R' is a pruned version of R (see, for example, Line 11 of Algorithm 4).

Instead of creating new sets S' and R' as the inputs into each recursive call of BB, we propose to reuse the same containers for S and R throughout the recursive execution, by initializing their spaces at the very beginning, and populating them with proper elements for use in each recursion body.

In particular, we maintain a single containers for both S and R (let us call it SR) using the dual-array data structure shown in Figure 15.

Specifically, two arrays with capacity $|V|$ are maintained: (1) *list*[] which keeps the list of elements for scanning and adding new elements; and (2) *pos*[] which maps vertex ID back to its position in *list*[], to facilitate element search and removal. Note that element addition, search and removal can all be done in $O(1)$ time. To illustrate using Figure 15, the vertices in the *list* array indexed between $[0, s)$ constitute the S set and those between $[s, r]$ constitute the R set.

As Figures 15(a)–(b) show, when removing vertex 4 from g, we first find its position in *list* as $pos[4] = 3$, and then swap this vertex with the last vertex $list[r]$ and finally decrement r. Note that their position values are also swapped in the array *pos*. In this way, the vertices are not completely erased from the structure and can be restored when returning back from the recursive call.

Similarly, as Figures 15(b)–(c) show, when moving vertex 2 from R into S, we first find its position in *list* as $pos[2] = 3$, and then swap this vertex with the first vertex of R, $list[s]$, and finally increment the value of s. Note that their position values are also swapped in the array *pos*. In this way, we can incrementally populate S and R for calling BB(.) with minimal cost.

For example, in Line 11 of Algorithm 5, we add vertex v from R to S using the above mentioned method, and then at Line 12 we remove v from g. However, since v is not completely removed from SR, it can be recovered when this recursive call is finished so that the caller can recover v back into its candidate set R.

As for Algorithm 4, since R is now split into two sets R' and B, we use a variant of our dual-array data structure, this time with 3 position pointers with b placed between s and r: $[0, s)$ is for S, $[s, b)$ is for R', and $[b, r]$ is for B. The corresponding $O(1)$-time operations for moving vertices around the sets can be similarly derived.

J INCREMENTAL DEGREE MAINTENANCE

Note that support $\delta_S(.)$ as defined in Definition 4.1 is frequently needed by the pruning techniques, such as $ub_i = \min\{|\pi_i|, \delta_S(v_i)\}$ in S-based strategy, Line 4 of Algorithm 5 for pivot-based branching, and reduction rules RR1, RR2, UBR1 and UBR2 (see Appendix M for the reduction rules). Also, $d_g(.)$ are also used in many places such as Line 4 of Algorithm 5 for pivot-based branching, and reduction rules RR3, BR1 and BR2 (which needs $\overline{d_g}(v) = |V(g)| - d_g(v)$). Therefore, we maintain $d_g(.)$ and $d_S(.)$ with arrays and keep them up to date by incrementally updating them while moving vertices around the sets S and R, so that $d_g(v)$ and $\delta_S(v) = k - |S| - d_S(v)$ can be immediately obtained for any vertex v in $O(1)$ time to check the conditions of the pruning techniques.

Specifically, whenever we add (resp. remove) a vertex v to (resp. from) g, we need to increment (resp. decrement) $d_g(u)$ for all $u \in N_g(v)$. Similarly, whenever we add (resp. remove) a vertex to (resp. from) S, we increment (resp. decrement) $d_S(u)$ for all $u \in N_g(v)$.

K IMPLEMENTING AUXILIARY BUFFERS Π

Recall that the S-based strategy requires an auxiliary set $\Pi = \cup_{v_i \in S} \pi_i$ that keeps the (potentially overlapping) non-neighbor sets π_i of all $v_i \in S$. So when S-based or SR-based strategy is used by Line 4 of Algorithm 4 (BB_part(.)) to compute the branching set, the partition(.) function needs to first initialize $\Pi = \bigcup_{v \in S} \pi_v$ where for each $v \in S$, $\pi_v = R - N(v)$.

Algorithm 10: Adaptive Branch-and-Bound Method

```
function BB(S, R, g)
    if k ≤ 5 then
        BB_part(S, R, g, true)
    else
        BB_pivot(S, R, g)
```

Since BB_part(.) is a recursive function, it is inefficient to allocate new space to create **Π** when calling partition(.), and delete the space when partition(.) returns, as such memory allocation overhead is incurred in each recursion body. Note that BB_part(.) is supposed to be called by Line 10 of Algorithm 2 at the top level, with $S = \{v_i\}$ and $g = (V_i, E_i)$ being v_i's 2-hop neighborhood graph which is much smaller than G. We, therefore, propose to associate each g with an array of $|V_i|$ buffers, each with capacity $|V_i|$. Whenever partition(.) is called in the branch under $\{v_i\}$, we simply reuse the allocated buffers to initialize Π instead of creating new space.

For each buffer, we support the locating and removal of a vertex from π_i in $O(1)$ time, using the dual-array data structure introduced in Appendix I with Figure 15. This is essential to support the efficient removal of vertices of a chosen partition I from each $\pi_i \in \Pi$ as needed by the S-based and SR-based strategies.

L ADAPTIVE BRANCH AND BOUND

Algorithm 10 shows our adaptive branch-and-bound algorithm that calls BB_part(.) (Algorithm 4) when $k \leq 5$ and calls BB_pivot(.) (Algorithm 5) when $k > 5$.

M DESCRIPTION OF REDUCTION RULES

We first present RR1-RR3 which shrink R by removing unpromising candidates. RR1–RR3 are used by all of the MkP algorithms.

RR1: For a vertex $v \in R$, if v has at least k non-neighbors in S, i.e., $\delta_S(v) \geq k$, then we can remove v from g as $S \cup \{v\}$ is not a k-plex.

This is because moving v to S will lead to $\delta_S(v) \geq k{+}1$ (as v is also a non-neighbor of v), but in a k-plex, we have $\delta_S(v) = k - \overline{N_S}(v) \leq k$, a contradiction. By the hereditary property of k-plex, if $S \cup \{v\}$ is not a k-plex, extending $S \cup \{v\}$ cannot produce a k-plex.

RR2: For a vertex $v \in R$, if v has a non-neighbor $u \in S$ such that $\delta_S(u) = k$, then we can remove v from g as $S \cup \{v\}$ is not a k-plex. This is because u already have k non-neighbors in S, hence it cannot have any further non-neighbors e.g., v.

This is because moving v to S will lead to $\delta_S(u) \geq k + 1$ (as $(u, v) \in E(g)$), but in a k-plex, we have $\delta_S(u) = k - \overline{N_S}(u) \leq k$, a contradiction.

RR3: For a vertex $v \in R$ if $d_g(v) + k \leq |P|$, then we can remove v from g as any k-plex containing $S \cup \{v\}$ will be of size at most $|P|$.

Note that this reduction rule is straightforward using Theorem E.1.

We next present two more effective upper-bound-based reduction techniques, UBR1 and UBR2, both proposed by the same group from the DB community.

UBR1 is first proposed by kPlexS, which is based on the following theorem [34]:

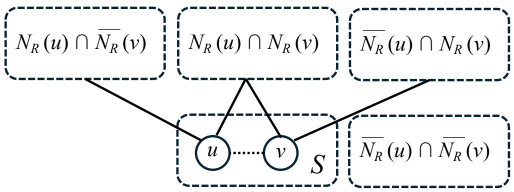


Figure 16: Upper Bound Illustration of Theorem M.1

THEOREM M.1. *Given an instance (S, R, g) and any two vertices $u, v \in S$, the maximum size of a k-plex containing S is at most $|S| + \delta_S(u) + \delta_S(v) + |N_R(u) \cap N_R(v)|$.*

UBR1 is simply a corollary following Theorem M.1:

UBR1: Consider a vertex $v \in R$, and let us denote $S' = S \cup \{v\}$. If there exists another vertex $u \in S'$ (i.e., $u \in S$) such that $|S'| + \delta_{S'}(v) + \delta_{S'}(u) + |N_R(u) \cap N_R(v)| \leq |P|$, then we can remove v from g as any k-plex containing $S \cup \{v\}$ will be of size at most $|P|$.

This is because for $R' = R - \{u, v\}$, we have $N_{R'}(u) \cap N_{R'}(v) = N_R(u) \cap N_R(v)$ since $u \notin N_R(u)$ and $v \notin N_R(v)$.

We next provide a proof for Theorem M.1:

PROOF. Let $P \subseteq S \cup R$ be an MkP containing S. For two arbitrary vertices $u, v \in S$, the candidate set R can be divided into four subsets as illustrated in Figure 16: (1) $N_R(u) \cap N_R(v)$, (2) $N_R(u) \cap \overline{N_R}(v)$, (3) $\overline{N_R}(u) \cap N_R(v)$, and (4) $\overline{N_R}(u) \cap \overline{N_R}(v)$. Therefore, let $f_S(X)$ be the subset of vertices in set X that are added to S to form an MkP P, we have

$$\begin{aligned} |P| = |S| &+ |f_S(N_R(u) \cap N_R(v))| + |f_S(N_R(u) \cap \overline{N_R}(v))| \\ &+ |f_S(\overline{N_R}(u) \cap N_R(v))| + |f_S(\overline{N_R}(u) \cap \overline{N_R}(v))| \end{aligned}.$$

Note that

$$\begin{aligned} (N_R(u) \cap \overline{N_R}(v)) + (\overline{N_R}(u) \cap \overline{N_R}(v)) &= \overline{N_R}(v), \\ (\overline{N_R}(u) \cap N_R(v)) + (\overline{N_R}(u) \cap \overline{N_R}(v)) &= \overline{N_R}(u). \end{aligned}$$

Since we assume $v \in S$, at most $\delta_S(v)$ more non-neighbors from $\overline{N_R}(v)$ can be added to P. Similarly, at most $\delta_S(u)$ more non-neighbors from $\overline{N_R}(u)$ can be added to P. Therefore, we have

$$\begin{aligned} f_S(N_R(u) \cap \overline{N_R}(v)) + f_S(\overline{N_R}(u) \cap \overline{N_R}(v)) &\leq \delta_S(v), \\ f_S(\overline{N_R}(u) \cap N_R(v)) + f_S(\overline{N_R}(u) \cap \overline{N_R}(v)) &\leq \delta_S(u). \end{aligned}$$

Therefore, since $f_S(N_R(u) \cap N_R(v)) \leq |N_R(u) \cap N_R(v)|$, we have

$$|P| \leq |S| + \delta_S(u) + \delta_S(v) + |N_R(u) \cap N_R(v)|,$$

which completes the proof since $u, v \in S$ are arbitrary. □

UBR2: This is a new upper-bound-based pruning rule proposed by the latest MkP algorithm kPlexT [35], which we find to be particularly effective in pruning R. Specifically, UBR2 first computes the following upper established by Theorem M.2, which will then be further tightened (to be described soon):

THEOREM M.2. *Given an instance (S, R, g) and a vertex $v \in R$, then every k-plex P that contains $S' = S \cup \{v\}$ will contain at most the following number of vertices:*

$$|S'| + \max\left\{i \mid \Sigma_{1 \leq j \leq i} \overline{d_{S_2}}(v_j) \leq \delta_{S'}(S_2)\right\},$$

where $S_2 = \{u \in S : \overline{d_{S_2}}(u) > k\}$, and $v_1, v_2, \ldots$ are vertices in $R' = R - \{v\}$ sorted in non-decreasing order of $\overline{d_{S_2}}(v_i)$.

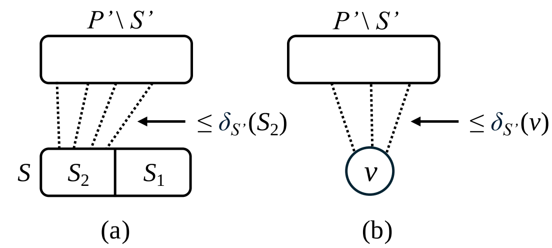


Figure 17: Illustration of UBR2

PROOF. Given an instance (S, R, g) and a vertex $v \in R$, let P' be an MkP containing $S' = S \cup \{v\}$; we aim to find an upper bound of P' so that we remove v from R if $|P'| \leq |P|$, where P is the largest k-plex found so far in the search.

To get a tighter bound of $|P'|$, S is divided into two disjoint subsets S_1 and S_2 where $S_1 = \{u \in S \mid \overline{d_g}(u) \leq k\}$ hence the vertices in S_1 always fulfill the degree requirement of k-plex, and $S_2 = S - S_1$.

Note that $S_2 = \{u \in S \mid \overline{d_g}(u) > k\}$ so that for each $u \in S$, we can only take a subset of $\overline{N_g}(u)$ into S. To compute an upper bound for $|P'|$, we want to move the largest possible number of vertices from R' into S' (so it upper-bounds the actual number), but we can introduce at most $\delta_{S'}(S_2) = \sum_{u \in S_2} \delta_{S'}(u)$ missing edges from the newly introduced vertices moved from R' to those in S_2 (note that $\delta_{S'}(u)$ is the quota of additional missing edges allowed, and we do not need to worry about their missing edges to S_1), or otherwise, some vertex $u \in S_2$ will have more than k missing edges (by the pigeonhole principle), violating the degree requirement of k-plex.

Therefore, to compute the largest possible number $UB_{S_2}(R')$ of vertices that can be moved from R' into S', we sort the vertices in R' in non-decreasing order of $\overline{d_{S_2}}(.)$, and add the vertices one at a time to S' until the missing-edge quota of $\delta_{S'}(S_2)$ is reached (Let $[v_1, \ldots, v_{|R'|}]$ be the sequence of vertices in R' sorted in non-decreasing order with respect to $\overline{d_{S_2}}(.)$). That is, when adding the next vertex $v_i \in R'$ to S' in order, we want to make sure:

$$\sum_{1 \leq j \leq i} \overline{d_{S_2}}(v_j) \leq \delta_{S'}(S_2),$$

In other words, we can calculate the number $UB_{S_2}(R')$ as follows:

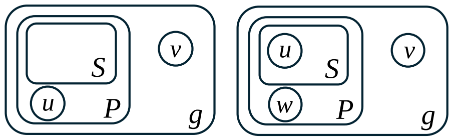
$$UB_{S_2}(R') = \max\left\{i \;\middle|\; \sum_{1 \leq j \leq i} \overline{d_{S_2}}(v_j) \leq \delta_{S'}(S_2)\right\}. \tag{1}$$

Therefore, the upper bound of $|P'|$ is given by

$$\begin{aligned} |P'| &\leq |S'| + UB_{S_2}(R') \\ &= |S| + 1 + \max\left\{i \;\middle|\; \sum_{1 \leq j \leq i} \overline{d_{S_2}}(v_j) \leq \delta_{S'}(S_2)\right\}, \end{aligned} \tag{2}$$

which completes the proof of Theorem M.2. □

To further tighten the bound of Equation (1), we would like to remove some vertices in the list $[v_1, \ldots, v_{|R'|}]$ sorted by $\overline{d_{S_2}}(.)$, so fewer vertices in the reduced list can be added to S', since some



(a) $u \notin S$ (b) $u \in S$

Figure 18: Illustration of the Proof of BR1

vertices with small $\overline{d_{S_2}}(.)$ are removed even when we have the same budget $\delta_{S'}(S_2)$ (so we may need to take fewer vertices in the list).

In fact, we can further reduce the budget, so that even fewer vertices can be added to S'. The insight is to consider the subset of candidates in R' that are neighbors of v, plus $\delta_{S'}(v)$ non-neighbors of v in R' with the smallest $\overline{d_{S_2}}(.)$, and let us denote the set by $\mathcal{R}_v$.

Note that we do not include more than $\delta_{S'}(v)$ vertices from $\overline{N_{R'}}(v)$ into $\mathcal{R}_v$, since if they are all added to S', then v will violate the degree requirement of k-plex.

We define the new missing-edge budget $\delta_{S'}(\mathcal{R}_v) = \sum_{u \in \mathcal{R}_v} \delta_{S'}(u)$, which is no larger than $\delta_{S'}(S_2)$. Then, we can compute a tighter bound similar to Equation (2) but over $\mathcal{R}_v$ rather than R':

$$\begin{aligned} |P'| &\leq |S'| + UB_{S_2}(\mathcal{R}_v) \\ &= |S| + 1 + \max\left\{ i \;\middle|\; \sum_{1 \leq j \leq i} \overline{d_{S_2}}(v'_j) \leq \delta_{S'}(S_2) \right\}, \end{aligned} \tag{3}$$

where $[v'_1, \ldots, v'_{|\mathcal{R}_v|}]$ is the sequence of vertices in $\mathcal{R}_v$ sorted in non-decreasing order with respect to $\overline{d_{S_2}}(.)$

We next present two more reduction techniques, BR1 and BR2, that allow us to move some candidates directly from R into S based on conditions related to $d_S(.)$.

BR1 is first proposed by kPlexS [34] based on conditions related to $d_g(v)$ which ensures that, if there exists an MkP P containing S, then there must also exists an MkP P' containing $S \cup \{v\}$. Therefore, if we only need to find one MkP, we can directly move such a v into S, but if our goal is to find all MkPs, we also need consider the other branch where v is removed from R.

BR1: Given an instance (S, R, g) and a vertex $v \in R$, if $d_g(v) \geq |V(g)| - 2$ and $S \cup \{v\}$ is a k-plex, then v is in some MkP containing S.

PROOF. Note that $d_g(v)$ can be at most $(|V(g)| - 1)$, i.e. connecting to all other vertices in g. Accordingly, we divide the condition $d_g(v) \geq |V(g)| - 2$ into two cases:

- $\mathbf{d_g(v) = |V(g)| - 1}$**:** in this case, v is adjacent to all vertices in g, so an MkP P containing S must contain v, as otherwise, $P \cup \{v\}$ is a larger valid k-plex, leading to contradiction.
- $\mathbf{d_g(v) = |V(g)| - 2}$**:** in this case, v is adjacent to all vertices in g except one vertex. Let u be that vertex not adjacent to v.

  Now consider an MkP, P, that contains S but $v \notin P$ (i.e., proof by contradition). Then, we must have $u \in P$ since otherwise, v is the neighbor of all vertices in P, so $P \cup \{v\}$ is a larger valid k-plex, conflicting with the assumption that P is an MkP.

  Also, we must have $\delta_P(u) = 0$ since otherwise, we can bring v into P without letting u violate the degree requirement of k-plex; moreover, since v is a neighbor of all the other vertices in P, its insertion to P will also not cause them to violate the degree requirement of k-plex. Therefore, $P \cup \{v\}$ is a larger valid k-plex, leading to a contradiction.

  Now that we know $u \in P$, we can further divide it into two cases:

  – **u ∉ S (see Figure 18(a)):** then we can construct $P' = P \cup \{v\} - \{u\}$ by exchanging u and v, which is actually a k-plex of the same size as P (which can be found if we just go to branch $S' = S \cup \{v\}$).

    This is because $v \in P'$ is a neighbor of every vertex $w \in P - \{u\}$ which is exchanged out of S, so v satisfies the degree requirement of k-plex.

    Moreover, for any $w \in P' - \{v\}$ (i.e., $w \in P - \{u\}$), $d_P(w)$ will not decrease (so retain the degree requirement of k-plex): (1) if there is a missing edge (w, u) in P, now we have one more edge (w, v) so $d_P(w)$ is incremented by 1; while (2) if (w, u) was in P, then now it is replaced with (w, v) so $d_P(w)$ remains the same.

    Therefore, all vertices in P' satisfy the degree requirement of k-plex, so P' is a valid k-plex.

  – **u ∈ S (see Figure 18(b)):** in this case, we cannot exchange $u \in S$ with v since we are considering an MkP, P, that contains S. But we can show that there must exist a vertex $w \in P - S$ not adjacent to u. This is because if all vertices in $P - S$ are neighbors of u (i.e., $\delta_P(u) = \delta_S(u)$), then since we have shown $\delta_P(u) = 0$ earlier, we have $\delta_S(u) = 0$ so $S \cup \{v\}$ cannot be a k-plex (since v is a non-neighbor of u but u has no missing-edge quota), violating BR1's condition that $S \cup \{v\}$ is a k-plex.

    Now that we have shown the existence of the vertex $w \in P - S$ not adjacent to u, we can exchange w (out of S) with v, and the resulting subgraph $P' = P \cup \{v\} - \{w\}$ is actually a k-plex of the same size as P (which can be found if we just go to branch $S' = S \cup \{v\}$).

    This is because for $u \in P'$, $d_{P'}(u) = d_P(u)$ since we just exchange its non-neighbor u out of P, but adding another non-neighbor v into P, so the number of missing edges with u remains the same.

    For $v \in P'$, it only misses an edge to u, so will not violate the degree requirement for k-plex for any $k \geq 2$ (for $k = 1$, $S \cup \{v\}$ is not a k-plex as required by the condition of BR1, so no pruning will be done).

    Moreover, for any $x \in P' - \{v, u\}$ (i.e., $x \in P - \{w, u\}$), $d_P(x)$ will not decrease (so retain the degree requirement of k-plex): (1) if there is a missing edge (x, w) in P, now we have one more edge (x, v) so $d_P(x)$ is incremented by 1; while (2) if (x, w) was in P, then now it is replaced with (x, v) so $d_P(x)$ remains the same.

    Therefore, all vertices in P' satisfy the degree requirement of k-plex, so P' is a valid k-plex.

□

Note that BR1 cannot be used if our goal is to find all MkPs. In contrast, BR2 is first proposed by BS [55] based on conditions

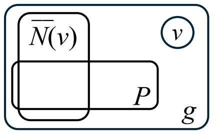


Figure 19: Illustration of the Proof of BR2

Table 9: Full Datasets

Dataset	$\|V\|$	$\|E\|$	d_{max}	d_{avg}	$D(G)$	Category
hamming6-2	64	1824	57	57	57	Synthetic Graph
johnson8-4-4	70	1855	53	53	53	Synthetic Graph
keller4	171	9435	124	110	102	Synthetic Graph
brock200-2	200	9876	114	99	84	Synthetic Graph
p-hat500-1	501	31,570	204	126	86	Synthetic Graph
socfb-Duke14	9885	506,437	1887	102	85	Facebook Network
ia-wiki-Talk	92,117	360,767	1220	8	58	Interaction Network
soc-buzznet	101,163	2,763,066	64,289	55	153	Social Network
soc-LiveMocha	104,103	2,193,083	2980	42	92	Social Network
soc-gowalla	196,591	950,327	14,730	10	51	Social Network
sc-msdoor	404,785	9,378,650	76	46	34	Scientific Computing
soc-youtube	495,957	1,936,748	25,409	8	49	Social Network
soc-digg	770,799	5,907,132	17,643	15	236	Social Network
sc-ldoor	909,537	20,770,807	76	46	34	Scientific Computing
soc-youtube-snap	1,134,890	2,987,624	28,754	5	51	Social Network
soc-lastfm	1,191,805	4,519,330	5150	8	70	Social Network
soc-pokec	1,632,803	22,301,964	14,854	27	47	Social Network
socfb-B-anon	2,937,612	20,959,854	4356	14	63	Facebook Network
soc-orkut	2,997,166	106,349,209	27,466	71	230	Social Network
socfb-A-anon	3,097,165	23,667,394	4915	15	74	Facebook Network
wikipedia-link-en	13,593,032	669,183,050	1,052,326	49	1114	Hyperlink Network
dbpedia-link	18,268,991	253,780,418	612,308	14	149	Hyperlink Network
wikipedia-link-en13	25,921,548	1,086,367,222	4,271,341	42	1120	Hyperlink Network
delicious-ui	33,778,221	203,595,714	29,319	6	193	Social Network
soc-sinaweibo	58,655,849	522,642,066	278,489	9	193	Social Network
web-ClueWeb09	147,925,593	893,533,906	308,477	6	192	Web Graph

related to $d_g(.)$ of v and all $u \in \overline{N_g}(v)$ that ensures that every MkP in g must contain v, so we can safely move such v to S, even when we are finding all MkPs.

BR2: Given an instance (S, R, g) and a vertex $v \in R$, if $\overline{d_g}(v) \leq k$, and $\overline{d_g}(u) \leq k$ for every $u \in \overline{N_g}(v)$, then every MkP in g must contain v.

PROOF. We prove it by contradiction. Assume that there exists an MkP P but $v \notin P$ as shown in Figure 19.

Then, since we assume $\overline{d_g}(v) \leq k$ in BR2's condition, and $\overline{N_{P\cup\{v\}}}(v) \subseteq \overline{N_g}(v)$, we have $\overline{d_{P\cup\{v\}}}(v) \leq \overline{d_g}(v) \leq k$.

For every vertex $u \in P$ that is a non-neighbor of v, we similarly have $\overline{d_{P\cup\{v\}}}(u) \leq \overline{d_g}(u) \leq k$.

For $u \in P$ that is a neighbor of v, $\overline{d_{P\cup\{v\}}}(u) = \overline{d_P}(u) \leq k$ where the last inequality is because P is a k-plex.

In summary, all vertices in $P \cup \{v\}$ have $\overline{d_{P\cup\{v\}}}(.) \leq k$, hence $P \cup \{v\}$ is a valid k-plex, contradicting with the assumption that P is an MkP. □

N DATASET DESCRIPTION

We use 26 datasets as summarized in Table 9, where d_{max} and d_{avg} indicate the maximum degree and average degree, respectively; and $D(G)$ is the degeneracy. These datasets span a wide range of graph sizes and categories: **(1)** synthetic graphs from DIMACS: *hamming6-2* [4], *johnson8-4-4* [6], *keller4* [7], *brock200-2* [1] and *p-hat500-1* [8]; **(2)** social networks: *soc-buzznet* [11], *soc-LiveMocha* [15], *soc-gowalla* [13], *soc-youtube* [19], *soc-digg* [12], *soc-youtube-snap* [20], *soc-lastfm* [14], *soc-pokec* [17], *soc-orkut* [16], *delicious-ui* [3] and *soc-sinaweibo* [18] **(3)** Facebook networks (in specific): *socfb-Duke14* [23], *socfb-B-anon* [22] and *socfb-A-anon* [21]; **(4)** Hyperlink networks: *wikipedia-link-en* [25], *dbpedia-link* [2] and *wikipedia-link-en13* [26]; **(5)** Scientific computing: *sc-msdoor* [10] and *sc-ldoor* [9]; **(6)** Web graph: *web-ClueWeb09* [24]; and **(7)** interaction network: *ia-wiki-Talk* [5]. We roughly categorize them as small and large graphs. The small ones are from DIMACS challenge and are very dense, so solving them is generally harder than the other real-world graphs. The large real-world graphs are carefully chosen so that solving them is sufficiently hard (many very large real graphs are too sparse and can by solved in sub-second).

Table 10: Size of MkP for Various Values of k (OOT = 3 hours)

Dataset	2	3	4	5	6	7	8	9	10	15	20
hamming6-2	32	32	40	48	52	64	64	64	64	64	64
johnson8-4-4	14	18	22	28	30	×	×	42	44	60	70
keller4	15	21	23	28	×	×	×	×	×	×	×
brock200-2	13	16	18	20	×	×	×	×	×	×	×
p-hat500-1	12	14	16	18	×	×	×	×	×	×	×
socfb-Duke14	38	43	45	48	51	54	56	58	60	70	81
ia-wiki-Talk	18	21	23	25	28	30	31	33	35	44	51
soc-buzznet	36	41	46	50	54	57	60	63	65	78	89
soc-LiveMocha	19	22	25	28	30	33	36	38	41	52	60
soc-gowalla	30	31	32	32	34	37	39	41	42	49	56
sc-msdoor	21	21	21	23	27	31	35	35	35	42	45
soc-youtube	20	21	24	26	28	30	31	33	35	43	50
soc-digg	57	63	69	72	75	78	82	84	87	100	109
sc-ldoor	21	21	21	23	27	31	35	35	35	42	45
soc-youtube-snap	20	21	24	26	28	30	32	33	35	43	51
soc-lastfm	18	21	24	27	29	31	34	36	38	47	56
soc-pokec	31	32	32	34	36	39	41	43	45	49	55
socfb-B-anon	27	30	33	35	38	40	43	45	47	57	64
soc-orkut	52	59	63	68	72	77	82	86	89	101	111
socfb-A-anon	28	32	35	37	40	41	44	46	47	54	61
wikipedia-link-en	1099	1101	1103	1105	1107	1109	1112	1115	1117	1129	1134
dbpedia-link	43	47	51	54	55	56	57	58	59	69	79
wikipedia-link-en13	429	430	431	432	432	432	432	432	432	432	432
delicious-ui	9	10	12	14	×	×	×	×	×	×	×
soc-sinaweibo	52	58	62	66	×	×	×	×	×	×	×
web-ClueWeb09	60	63	67	72	×	×	×	×	×	×	×

O COMPLETE RESULTS OF EXPERIMENTS

Due to the page limitation of the main paper, we include all our results in Tables 10–23 on the next few pages.

We set the time limit at 1800 seconds. We use × in our tables to indicate that execution exceeds this time limit. We tested the datasets for 11 different values of k, i.e., 2, 3, 4, 5, 6, 7, 8, 9, 10, 15 and 20. Table 10 shows the MkP sizes of all datasets for all values of k, which we obtain by running U-MkP for 3 hours and report × if it runs out of time (OOT). Note that here the OOT threshold is different from the one used in our experiments (i.e., 1800 seconds).

Table 11 shows the full results of comparison among upper bounding (UB) techniques, where U-MkP does not apply any UB-based pruning. We can see that despite the active development of UB-based techniques for branch pruning by the AI community, none of them are obviously more effective than if they were not used at all. In fact, some techniques can backfire due to their overhead. Therefore, we recommend to apply no bounding technique by default.

We do notice an exception that S-Based is quite effective on the two largest graphs, especially *web-ClueWeb09* when $k = 4, 5$. Moreover, S-Based does not bring much slowdown in most cases, so it could be safe alternative for large graphs (e.g., $|V| = 10^7$) and worth trying out if OOT happens.

Table 11: Comparison of Different Upper Bounding Techniques

Dataset	k = 2				k = 3				k = 4				k = 5			
	U-MkP	S-Bound	R-Bound	SR-Bound	U-MkP	S-Bound	R-Bound	SR-Bound	U-MkP	S-Bound	R-Bound	SR-Bound	U-MkP	S-Bound	R-Bound	SR-Bound
hamming6-2	0.2	0.2	0.2	**0.2**	**35.4**	62.1	54.6	36.1	**36.9**	72.7	58.6	38.1	**3.6**	6.1	5.9	3.7
johnson8-4-4	**0.7**	0.7	0.8	0.8	**5.7**	6.0	6.4	6.7	**29.6**	31.9	34.4	32.8	**2.0**	2.3	2.7	2.1
keller4	25.2	**23.8**	26.5	25.9	**28.9**	29.1	31.3	29.4	**1504.8**	1526.8	1748.0	1726.5	**1120.0**	1125.9	1363.5	1211.3
brock200-2	4.2	**4.1**	4.4	4.2	21.4	**20.8**	22.5	21.0	**71.6**	73.6	81.4	79.7	**715.0**	757.1	771.4	792.9
p-hat500-1	14.4	**13.6**	13.9	14.1	29.6	29.1	31.6	**26.7**	**70.2**	70.7	77.2	72.1	**242.0**	247.1	268.9	244.1
socfb-Duke14	1.2	**1.2**	1.4	1.3	0.8	**0.8**	0.9	0.9	1.4	**1.2**	1.5	1.4	**2.6**	3.1	4.6	2.7
ia-wiki-Talk	0.8	**0.7**	0.8	0.8	0.6	**0.6**	0.6	0.6	**0.4**	0.5	0.5	0.5	**0.7**	0.7	0.7	0.8
soc-buzznet	358.0	358.6	363.9	**349.2**	189.1	**168.0**	187.2	184.0	153.0	**138.2**	154.6	241.2	1771.0	1572.1	**1317.9**	×
soc-LiveMocha	3.3	3.3	3.3	**3.1**	**2.2**	2.3	2.3	2.3	**2.3**	2.3	2.4	2.8	**7.0**	8.2	7.8	9.9
soc-gowalla	0.1	**0.1**	0.1	0.1	0.1	**0.1**	0.1	0.1	0.1	0.1	0.1	**0.1**	**0.1**	0.1	0.1	0.1
sc-msdoor	7.3	7.4	7.5	**7.3**	12.4	**12.3**	12.5	12.3	**25.0**	25.5	29.2	25.3	**40.8**	49.4	56.3	41.5
soc-youtube	**0.2**	0.3	0.3	0.2	**0.3**	0.3	0.3	0.3	0.3	0.3	**0.2**	0.3	0.3	**0.2**	0.3	0.3
soc-digg	121.9	119.3	125.3	**90.0**	49.4	**48.6**	60.3	50.8	**33.2**	35.3	94.9	35.6	**102.4**	218.5	1266.5	×
sc-ldoor	14.1	**14.1**	14.9	14.6	24.6	25.5	25.2	**24.4**	56.6	**48.6**	55.3	51.3	**71.3**	89.7	97.2	71.4
soc-youtube-snap	**0.4**	0.4	0.4	0.4	0.5	0.5	0.5	**0.5**	0.3	0.4	**0.3**	0.4	0.3	0.4	**0.3**	0.4
soc-lastfm	3.6	3.7	3.6	**3.4**	1.9	**1.9**	2.0	2.0	**1.5**	1.5	1.6	1.5	2.2	**2.0**	2.4	2.7
soc-pokec	**5.4**	6.8	5.8	5.7	5.1	5.3	6.0	**5.0**	6.1	6.2	5.8	**5.6**	**4.0**	4.4	5.3	5.0
socfb-B-anon	**7.4**	8.6	8.2	9.9	**7.1**	7.6	8.8	7.6	7.4	8.0	**6.7**	8.4	7.8	8.4	**6.9**	8.4
soc-orkut	96.1	88.0	104.6	**85.4**	79.1	**73.8**	91.1	84.5	**611.5**	645.2	1150.0	644.8	×	×	×	×
socfb-A-anon	**7.8**	8.1	8.9	8.4	**6.5**	6.6	7.7	8.4	**6.2**	7.3	8.4	7.2	7.0	6.1	6.6	**5.9**
wikipedia-link-en	**12.1**	13.3	12.6	12.8	12.0	**11.8**	11.9	11.9	12.9	15.4	13.8	**11.6**	15.3	16.9	**13.4**	15.8
dbpedia-link	76.0	66.1	61.1	**57.4**	**52.2**	59.9	55.4	63.0	50.5	56.2	**50.4**	56.4	49.3	53.7	55.0	**45.3**
wikipedia-link-en13	249.0	231.4	241.6	**230.5**	290.7	233.7	**228.9**	231.1	**249.3**	276.2	249.4	249.7	290.9	259.5	**256.3**	308.5
delicious-ui	62.1	56.9	64.2	**51.9**	99.0	107.7	**97.9**	99.0	**101.2**	112.6	115.9	104.7	**119.3**	138.7	827.9	124.1
soc-sinaweibo	141.2	140.3	157.8	**138.2**	126.3	**124.7**	167.2	129.8	207.3	**161.0**	511.3	×	**1661.7**	×	×	×
web-ClueWeb09	43.8	40.9	**37.1**	42.0	70.5	**63.5**	70.0	73.2	613.2	**330.2**	897.0	1037.4	×	**149.9**	×	×

Dataset	k = 6				k = 7				k = 8				k = 9			
	U-MkP	S-Bound	R-Bound	SR-Bound	U-MkP	S-Bound	R-Bound	SR-Bound	U-MkP	S-Bound	R-Bound	SR-Bound	U-MkP	S-Bound	R-Bound	SR-Bound
hamming6-2	16.9	**12.1**	18.2	17.8	**0.0**	0.0	0.0	0.0	**0.0**	0.0	0.0	0.0	**0.0**	0.0	0.0	0.0
johnson8-4-4	**332.4**	377.8	482.8	350.5	×	×	×	×	×	×	×	×	81.3	131.2	**64.6**	81.8
keller4	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	0.6	**0.6**	0.7	0.6	0.6	**0.6**	0.7	0.6	0.6	**0.6**	0.7	0.6	0.7	**0.7**	0.8	0.7
ia-wiki-Talk	0.4	**0.4**	0.5	0.4	0.5	**0.5**	0.5	0.5	0.9	**0.9**	1.1	1.0	1.4	1.4	1.6	**1.4**
soc-buzznet	123.7	119.1	150.8	**114.2**	**76.1**	77.7	99.2	80.5	75.4	**71.9**	91.2	75.3	106.7	**101.6**	131.7	109.0
soc-LiveMocha	**3.1**	3.1	3.7	3.2	4.1	4.0	4.9	**3.8**	4.1	**3.9**	5.1	4.1	2.3	**2.3**	2.5	2.4
soc-gowalla	0.1	**0.1**	0.1	0.1	**0.1**	0.1	0.1	0.1	0.1	0.1	**0.1**	0.1	0.1	0.1	**0.1**	0.1
sc-msdoor	**28.8**	39.5	38.0	30.1	6.1	6.7	6.2	**5.9**	**2.3**	2.3	2.4	2.4	3.6	3.5	3.5	**3.5**
soc-youtube	0.2	**0.2**	0.2	0.2	0.2	0.2	**0.2**	0.2	**0.3**	0.3	0.3	0.3	0.3	**0.3**	0.3	0.3
soc-digg	**35.4**	36.5	41.3	36.7	**54.5**	57.2	66.6	55.9	**49.1**	50.4	61.2	50.3	**46.7**	50.3	59.9	48.8
sc-ldoor	**50.4**	67.7	63.1	51.0	**11.3**	13.1	11.9	11.7	7.1	7.2	**7.0**	7.3	8.0	**8.0**	8.0	8.4
soc-youtube-snap	0.3	0.3	0.3	**0.3**	0.3	0.3	0.3	**0.3**	0.4	**0.4**	0.4	0.4	0.6	**0.5**	0.6	0.5
soc-lastfm	2.4	**2.4**	2.9	2.4	3.5	**3.3**	4.1	3.4	1.1	**1.1**	1.3	1.1	1.4	**1.3**	1.6	1.4
soc-pokec	4.5	4.0	3.5	**3.4**	**3.4**	4.5	4.0	3.8	**3.2**	3.4	3.3	3.6	8.1	**3.2**	3.2	4.1
socfb-B-anon	8.2	8.1	**6.6**	7.5	6.7	7.5	7.4	**6.2**	**5.6**	6.1	5.9	7.3	6.7	6.9	6.7	**5.5**
soc-orkut	86.9	82.5	87.8	**76.8**	68.8	**56.0**	62.6	60.4	49.9	46.9	57.5	**44.7**	49.7	50.6	49.4	**43.0**
socfb-A-anon	6.9	6.9	**5.7**	5.7	5.3	6.2	6.5	**5.3**	5.7	**5.2**	5.7	5.8	**4.8**	6.1	4.9	5.6
wikipedia-link-en	19.1	25.9	**18.2**	20.1	247.1	458.1	242.7	**238.5**	**69.0**	113.3	70.1	70.5	29.6	42.5	**26.1**	29.6
dbpedia-link	696.0	661.4	777.8	**169.0**	×	×	×	**1541.5**	×	×	×	×	×	×	×	×
wikipedia-link-en13	**245.3**	310.3	248.7	289.0	304.4	289.9	**264.2**	298.2	284.1	295.8	271.9	**268.6**	**255.5**	279.2	294.1	311.9
delicious-ui	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

Dataset	k = 10				k = 15				k = 20			
	U-MkP	S-Bound	R-Bound	SR-Bound	U-MkP	S-Bound	R-Bound	SR-Bound	U-MkP	S-Bound	R-Bound	SR-Bound
hamming6-2	**0.0**	0.0	0.0	0.0	0.0	0.0	**0.0**	0.0	**0.0**	0.0	**0.0**	0.0
johnson8-4-4	**1271.9**	1636.5	1793.4	1347.7	**20.4**	25.4	27.7	21.2	0.0	0.0	**0.0**	0.0
keller4	×	×	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	**0.6**	0.7	0.8	0.7	**2.7**	2.8	3.6	2.9	1.7	**1.7**	2.3	1.8
ia-wiki-Talk	1.5	**1.5**	1.9	1.5	3.6	**3.5**	4.6	3.6	23.7	**23.6**	30.7	23.9
soc-buzznet	150.3	**147.3**	195.9	152.8	520.7	515.3	665.6	**509.3**	701.7	687.7	892.6	**679.5**
soc-LiveMocha	**2.8**	2.9	3.5	2.9	**0.9**	0.9	1.1	0.9	3.2	**3.1**	4.0	3.3
soc-gowalla	0.1	**0.1**	0.1	0.1	0.2	**0.2**	0.2	0.2	0.4	**0.4**	0.5	0.4
sc-msdoor	3.9	3.9	3.7	**3.7**	3.4	**3.1**	3.2	3.4	5.3	6.0	**4.7**	5.2
soc-youtube	**0.3**	0.3	0.3	0.3	**0.9**	0.9	1.2	0.9	2.8	**2.6**	3.8	3.0
soc-digg	**35.1**	40.4	47.3	38.8	7.9	**7.7**	7.9	9.0	**7.4**	8.0	8.2	7.6
sc-ldoor	**8.6**	8.9	8.6	9.0	7.3	7.3	**6.9**	7.1	10.1	10.0	**9.6**	10.4
soc-youtube-snap	0.5	0.5	0.6	**0.5**	1.2	**1.2**	1.5	1.2	4.6	**4.4**	6.3	4.8
soc-lastfm	1.6	**1.6**	2.0	1.6	**5.6**	5.8	7.2	5.7	**21.3**	22.3	26.8	21.4
soc-pokec	3.1	7.9	3.7	**3.0**	3.3	**2.8**	3.3	3.2	**3.4**	4.5	4.5	3.6
socfb-B-anon	7.0	6.8	**5.3**	6.9	6.3	5.3	6.3	**5.2**	6.7	**6.6**	7.1	6.8
soc-orkut	46.1	48.3	47.6	**41.6**	**33.6**	40.2	42.7	41.3	37.2	34.2	**32.9**	35.9
socfb-A-anon	4.9	5.4	**4.6**	4.7	4.8	5.1	**4.4**	4.5	3.7	4.1	**3.7**	4.1
wikipedia-link-en	**12.0**	14.7	12.9	15.0	9.9	10.1	12.6	**9.5**	7.2	**6.9**	7.2	7.5
dbpedia-link	×	×	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	292.3	292.9	**257.7**	286.4	297.2	268.9	292.0	**264.6**	277.6	**220.8**	244.4	228.3
delicious-ui	×	×	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×

Table 12 shows the full results of competitive branching methods. We can see that the baseline binary branching method (where pivot is selected simply based on the degeneracy order) is not competitive and runs OOT most of the time. When k is small, there is no clear winner but S-based method is the most stable and often performs the best for more time-consuming jobs, so we adopt S-based branching when $k \leq 5$ by default. On the other hand, when k is large, the pivoting-based branching method by Maple is a clear winner so we adopt pivoting-based branching by default when $k > 5$.

This default setting works well in most cases but we do notice some exceptions: (1) the advantage of pivot-based branching may occur at or before $k = 5$, such as on *soc-buzznet*, *soc-digg* and *soc-orkut*, so is worth trying as an alternative when k is small but close to 5; (2) the advantage of S-based branching goes way beyond $k = 5$ on *dbpedia-link*, so can be an alternative to try when OOT happens with pivot-based branching; (3) R-based branching is the most effective on *brock200-2* and *p-hat500-1* but they are small (and dense) synthetic graphs, and for real graphs, it only significantly beats the default configuration on *delicious-ui* when $k = 6, 7$, so can be an alternative to try when OOT happens with pivot-based branching. In summary, we should begin with the default configuration, but when OOT happens with pivot-based branching, we can try S-based and R-based branching as alternatives.

Table 12: Comparison of Different Branching Methods

Dataset	k = 2					k = 3					k = 4					k = 5				
	S-Br	R-Br	SR-Br	Pivot-Br	Binary-Br	S-Br	R-Br	SR-Br	Pivot-Br	Binary-Br	S-Br	R-Br	SR-Br	Pivot-Br	Binary-Br	S-Br	R-Br	SR-Br	Pivot-Br	Binary-Br
hamming6-2	0.2	0.2	0.2	0.2	**0.1**	35.4	42.8	36.0	**35.3**	35.6	36.9	46.1	38.4	**36.8**	×	**3.6**	4.7	3.8	3.7	89.9
johnson8-4-4	**0.7**	1.1	0.8	1.0	1.5	**5.7**	18.9	8.1	10.3	25.2	**29.6**	173.4	43.4	81.6	398.0	**2.0**	37.5	4.5	12.9	153.8
keller4	25.2	49.9	**17.7**	69.8	×	**28.9**	87.9	31.3	274.0	×	**1504.8**	×	×	×	×	**1120.0**	×	×	×	×
brock200-2	4.2	3.2	**2.7**	8.5	×	21.4	**4.4**	18.6	51.7	×	71.6	**13.6**	87.2	153.4	×	715.0	**178.3**	569.2	1394.8	×
p-hat500-1	14.4	14.4	**9.1**	33.2	×	29.6	**12.7**	16.2	105.8	×	70.2	**14.1**	47.3	181.1	×	242.0	**55.3**	252.0	666.8	×
socfb-Duke14	**1.2**	1.9	2.1	1.8	1.9	**0.8**	1.0	1.7	0.8	171.0	1.4	1.3	33.2	**0.8**	×	2.6	1.6	346.7	**0.7**	×
ia-wiki-Talk	0.8	1.1	**0.6**	1.3	1.2	**0.6**	0.8	0.6	0.8	×	**0.4**	0.6	0.5	0.5	0.7	0.7	0.8	1.0	**0.5**	1.8
soc-buzznet	**358.0**	933.6	389.1	1365.3	×	**189.1**	589.5	770.2	456.0	×	**153.0**	315.8	×	206.7	×	1771.0	664.1	×	**180.0**	×
soc-LiveMocha	3.3	5.6	**2.7**	6.6	×	**2.2**	3.5	2.6	3.2	×	**2.3**	3.6	4.0	2.6	×	7.0	8.3	12.2	**3.0**	×
soc-gowalla	0.1	0.1	0.1	**0.1**	0.1	0.1	0.1	**0.1**	0.1	0.1	0.1	0.1	0.1	**0.1**	0.1	0.1	0.1	0.1	**0.1**	0.1
sc-msdoor	7.3	7.4	7.5	**7.0**	7.1	12.4	14.3	13.4	11.7	**11.4**	**25.0**	39.6	31.3	26.8	25.8	**40.8**	252.9	67.9	164.4	153.8
soc-youtube	**0.2**	0.3	0.3	0.3	0.3	**0.3**	0.3	0.3	0.3	0.4	0.3	0.3	0.3	0.2	**0.2**	0.3	0.3	0.3	**0.2**	4.5
soc-digg	**121.9**	226.5	204.9	304.0	276.1	**49.4**	162.5	1373.8	80.2	×	**33.2**	545.9	×	43.9	×	102.4	×	×	**43.0**	×
sc-ldoor	14.1	14.7	14.4	**13.6**	13.6	24.6	26.4	26.3	24.2	**23.1**	56.6	89.0	60.8	51.3	**51.0**	**71.3**	441.4	123.7	283.3	287.2
soc-youtube-snap	0.4	0.4	0.4	**0.4**	0.4	0.5	0.5	**0.5**	0.5	0.6	0.3	0.3	0.3	**0.3**	0.4	0.3	0.4	0.3	**0.3**	0.4
soc-lastfm	3.6	4.6	**2.4**	7.5	6.8	**1.9**	2.9	2.1	3.4	×	**1.5**	2.3	2.0	2.2	×	2.2	**2.1**	4.5	3.2	×
soc-pokec	**5.4**	5.6	5.7	6.6	7.3	5.1	6.5	**4.9**	6.4	6.2	6.1	5.9	6.2	**5.7**	5.7	4.0	4.0	**4.0**	4.6	4.4
socfb-B-anon	**7.4**	10.1	8.9	7.4	8.9	**7.1**	8.1	8.7	7.8	9.5	7.4	9.3	7.6	**6.8**	12.8	**7.8**	8.7	9.1	8.0	220.3
soc-orkut	96.1	135.1	179.0	96.0	**86.1**	79.1	103.0	511.1	**77.3**	×	611.5	983.2	×	**76.8**	×	×	×	×	**75.1**	×
socfb-A-anon	**7.8**	9.7	9.7	9.0	9.3	**6.5**	7.7	6.7	7.5	6.6	**6.2**	6.5	7.6	7.6	7.6	7.0	7.6	**6.0**	6.2	7.5
wikipedia-link-en	12.1	12.5	11.8	**11.7**	12.8	12.0	**11.9**	14.3	13.4	12.8	12.9	14.5	**11.6**	13.9	12.4	15.3	15.1	**13.8**	15.2	19.4
dbpedia-link	76.0	64.8	**54.8**	233.4	×	52.2	64.5	**48.2**	404.0	×	**50.5**	53.6	56.2	141.5	×	**49.3**	49.8	53.3	151.5	×
wikipedia-link-en13	249.0	260.3	288.5	**240.5**	246.9	290.7	277.7	269.8	247.3	**234.6**	249.3	281.5	235.4	231.9	**226.7**	290.9	274.1	254.7	**224.1**	281.5
delicious-ui	62.1	57.8	63.9	**51.9**	64.8	99.0	**77.1**	94.1	161.7	×	**101.2**	162.8	104.1	160.3	×	**119.3**	191.1	120.7	188.8	×
soc-sinaweibo	**141.2**	1036.8	1805.6	306.3	×	**126.3**	1397.6	×	651.5	×	**207.3**	×	×	1211.9	×	**1661.7**	×	×	×	×
web-ClueWeb09	**43.8**	167.5	155.0	192.0	392.6	**70.5**	×	×	×	×	**613.2**	×	×	×	×	×	×	×	×	×

Dataset	k = 6					k = 7					k = 8					k = 9				
	S-Br	R-Br	SR-Br	Pivot-Br	Binary-Br	S-Br	R-Br	SR-Br	Pivot-Br	Binary-Br	S-Br	R-Br	SR-Br	Pivot-Br	Binary-Br	S-Br	R-Br	SR-Br	Pivot-Br	Binary-Br
hamming6-2	**16.7**	21.5	17.9	16.9	×	0.0	**0.0**	0.0	**0.0**	0.0	0.0	0.0	0.0	0.0	**0.0**	0.0	0.0	0.0	**0.0**	0.0
johnson8-4-4	**143.5**	800.4	181.6	332.4	×	×	**1575.1**	×	×	×	×	×	×	×	×	114.9	127.4	**76.9**	81.3	×
keller4	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
brock200-2	×	**37.3**	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	**236.8**	972.0	×	×	×	**992.3**	1692.0	×	×	×	**315.7**	×	×	×	×	**830.9**	×	×	×
socfb-Duke14	7.2	1.9	852.4	**0.6**	×	18.2	2.4	×	**0.6**	×	656.9	6.2	×	**0.6**	×	×	16.3	×	**0.7**	×
ia-wiki-Talk	0.7	0.5	1.5	**0.4**	8.4	2.4	0.9	9.5	**0.5**	3.3	44.7	2.8	22.8	**0.9**	×	193.4	5.4	67.4	**1.4**	×
soc-buzznet	×	655.7	×	**123.7**	×	×	735.3	×	**76.1**	×	×	1438.2	×	**75.4**	×	×	×	×	**106.7**	×
soc-LiveMocha	32.0	13.3	35.6	**3.1**	×	159.4	34.3	115.7	**4.1**	×	492.7	62.2	246.0	**4.1**	×	440.3	20.5	131.9	**2.3**	×
soc-gowalla	0.1	0.1	0.2	**0.1**	0.1	0.2	0.2	0.2	**0.1**	0.2	**0.1**	0.1	0.1	0.1	0.1	0.1	**0.1**	0.1	0.1	0.1
sc-msdoor	**14.5**	51.0	23.3	28.8	36.9	6.1	**5.9**	6.9	6.1	6.0	2.3	2.4	**2.3**	2.3	2.6	3.6	3.4	**3.3**	3.6	3.4
soc-youtube	0.2	0.2	0.2	**0.2**	0.6	0.4	0.3	0.5	**0.2**	0.3	1.1	0.4	1.0	**0.3**	7.2	0.9	0.4	2.3	**0.3**	×
soc-digg	156.4	×	×	**35.4**	×	×	×	×	**54.5**	×	1296.2	×	×	**49.1**	×	×	×	×	**46.7**	×
sc-ldoor	**27.4**	86.4	40.1	50.4	62.2	11.4	11.6	14.1	11.3	**11.2**	**6.9**	7.0	7.2	7.1	7.4	8.0	8.3	7.9	8.0	**7.8**
soc-youtube-snap	**0.3**	0.4	0.3	0.3	0.4	0.4	0.3	0.7	**0.3**	0.4	1.4	0.8	1.1	**0.4**	×	11.1	0.8	5.8	**0.6**	×
soc-lastfm	**2.3**	2.7	17.5	2.4	×	7.8	5.4	14.8	**3.5**	×	3.4	2.0	45.4	**1.1**	×	12.3	3.2	24.2	**1.4**	×
soc-pokec	4.4	4.1	4.8	4.5	**3.9**	3.4	3.4	**3.3**	3.4	4.5	**3.0**	3.5	3.9	3.2	4.2	3.5	**3.3**	4.4	8.1	8.5
socfb-B-anon	7.0	**7.0**	9.8	8.2	8.9	7.5	**6.6**	7.0	6.7	×	5.8	8.2	9.7	**5.6**	×	6.6	7.5	**6.0**	6.7	12.1
soc-orkut	×	×	×	**86.9**	×	×	×	×	**68.8**	×	×	×	×	**49.9**	×	×	×	×	**49.7**	×
socfb-A-anon	7.1	6.8	**5.7**	6.9	6.5	5.7	6.9	5.5	**5.3**	6.4	6.4	**5.5**	6.3	5.7	7.3	**4.8**	5.8	5.8	4.8	5.5
wikipedia-link-en	**16.6**	17.0	17.7	19.1	27.6	245.8	**245.1**	256.6	247.1	581.7	**67.5**	68.3	69.4	69.0	239.0	**28.2**	29.7	33.5	29.6	113.4
dbpedia-link	52.1	72.0	**49.8**	696.0	×	**45.3**	236.3	48.3	×	×	**51.3**	×	53.1	×	×	**46.7**	×	214.4	×	×
wikipedia-link-en13	292.6	**239.6**	250.6	245.3	278.5	319.4	269.2	262.1	304.4	**260.2**	302.8	304.6	297.7	284.1	**272.0**	274.3	286.9	287.7	255.5	**244.6**
delicious-ui	1140.4	**272.9**	929.7	×	×	1248.4	**224.4**	935.0	×	×	1320.7	987.1	**971.4**	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

Dataset	k = 10					k = 15					k = 20				
	S-Br	R-Br	SR-Br	Pivot-Br	Binary-Br	S-Br	R-Br	SR-Br	Pivot-Br	Binary-Br	S-Br	R-Br	SR-Br	Pivot-Br	Binary-Br
hamming6-2	0.0	0.0	**0.0**	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**	0.0	0.0	0.0	0.0	**0.0**
johnson8-4-4	×	×	×	**1271.9**	×	111.6	191.5	166.4	**20.4**	393.9	0.0	0.0	0.0	0.0	**0.0**
keller4	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	**413.0**	×	×	×	×	**1262.6**	×	×	×	×	×	×	×	×
socfb-Duke14	×	52.0	×	**0.6**	×	×	×	×	**2.7**	×	×	×	×	**1.7**	×
ia-wiki-Talk	436.6	4.3	101.0	**1.5**	×	×	53.7	×	**3.6**	×	×	×	×	**23.7**	×
soc-buzznet	×	×	×	**150.3**	×	×	×	×	**520.7**	×	×	×	×	**701.7**	×
soc-LiveMocha	×	16.9	388.6	**2.8**	×	×	2.8	233.8	**0.9**	×	×	23.6	×	**3.2**	×
soc-gowalla	0.4	0.5	0.5	**0.1**	0.2	21.1	4.0	25.3	**0.2**	528.7	×	25.7	×	**0.4**	×
sc-msdoor	3.9	3.8	4.0	3.9	**3.8**	3.3	**3.1**	3.4	3.4	3.1	49.0	29.1	16.6	**5.3**	65.6
soc-youtube	3.0	0.5	2.5	**0.3**	×	×	4.4	512.3	**0.9**	×	×	1076.7	×	**2.8**	×
soc-digg	1580.5	×	×	**35.1**	×	8.1	184.9	**7.8**	7.9	40.5	13.4	×	×	**7.4**	15.0
sc-ldoor	9.0	**8.1**	8.8	8.6	8.4	7.5	7.1	7.5	7.3	**6.9**	10.3	10.3	10.7	10.1	**9.6**
soc-youtube-snap	21.5	1.0	9.9	**0.5**	×	×	15.8	837.4	**1.2**	×	×	428.8	×	**4.6**	×
soc-lastfm	54.2	6.8	68.4	**1.6**	×	×	58.0	×	**5.6**	×	×	999.3	×	**21.3**	×
soc-pokec	3.4	3.2	4.0	**3.1**	3.9	**2.8**	3.0	3.1	3.3	3.5	4.3	17.1	4.4	**3.4**	3.6
socfb-B-anon	6.4	7.8	6.2	7.0	**6.2**	6.4	8.1	**5.8**	6.3	6.0	8.2	18.5	8.1	**6.7**	79.5
soc-orkut	×	×	×	**46.1**	×	44.1	×	1772.4	**33.6**	40.4	38.5	37.6	**35.6**	37.2	38.8
socfb-A-anon	**4.7**	5.5	4.9	4.9	5.5	5.3	12.7	5.1	**4.8**	5.3	×	223.1	1223.8	**3.7**	×
wikipedia-link-en	12.4	12.6	13.5	**12.0**	50.5	11.1	11.1	10.9	**9.9**	12.0	7.0	8.0	**6.7**	7.2	8.6
dbpedia-link	**81.1**	×	×	×	×	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	330.6	283.1	**265.1**	292.3	287.2	269.2	**240.6**	272.6	297.2	281.7	247.7	**227.2**	283.3	277.6	237.2
delicious-ui	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

Table 13 compares our default configuration of U-MkP with the version with **Two-Hop** being disabled. We can see that disabling **Two-Hop** is disastrous and cause most experiments to run OOT, so it should always be enabled.

Table 13: Comparison of the Default Configuration with the Variant Without Two-Hop Pruning

Dataset	$k = 2$		$k = 3$		$k = 4$		$k = 5$		$k = 6$		$k = 7$	
	w/ TwoHop	w/o TwoHop	w/ TwoHop	w/o TwoHop	w/ TwoHop	w/o TwoHop	w/ TwoHop	w/o TwoHop	w/ TwoHop	w/o TwoHop	w/ TwoHop	w/o TwoHop
hamming6-2	**0.2**	0.5	**35.4**	189.5	**36.9**	492.7	**3.6**	61.9	**16.9**	291.5	**0.0**	0.0
johnson8-4-4	**0.7**	2.7	**5.7**	46.8	**29.6**	500.1	**2.0**	281.7	**332.4**	×	×	×
keller4	**25.2**	197.3	**28.9**	×	**1504.8**	×	**1120.0**	×	×	×	×	×
brock200-2	**4.2**	9.0	**21.4**	140.0	**71.6**	554.2	**715.0**	×	×	×	×	×
p-hat500-1	**14.4**	24.9	**29.6**	313.6	**70.2**	×	**242.0**	×	×	×	×	×
socfb-Duke14	**1.2**	4.7	**0.8**	7.0	**1.4**	16.6	**2.6**	33.2	**0.6**	99.0	**0.6**	364.7
ia-wiki-Talk	**0.8**	4.8	**0.6**	11.4	**0.4**	11.3	**0.7**	13.0	**0.4**	19.6	**0.5**	154.0
soc-buzznet	**358.0**	×	**189.1**	×	**153.0**	×	**1771.0**	×	**123.7**	×	**76.1**	×
soc-LiveMocha	**3.3**	18.0	**2.2**	18.2	**2.3**	42.4	**7.0**	138.6	**3.1**	×	**4.1**	×
soc-gowalla	**0.1**	0.1	**0.1**	0.1	**0.1**	0.2	**0.1**	1.2	**0.1**	1.8	**0.1**	6.4
sc-msdoor	**7.3**	×	**12.4**	×	**25.0**	×	**40.8**	×	**28.8**	×	**6.1**	×
soc-youtube	**0.2**	0.8	**0.3**	3.6	**0.3**	3.6	**0.3**	1.2	**0.2**	2.3	**0.2**	13.4
soc-digg	**121.9**	×	**49.4**	×	**33.2**	×	**102.4**	×	**35.4**	×	**54.5**	×
sc-ldoor	**14.1**	×	**24.6**	×	**56.6**	×	**71.3**	×	**50.4**	×	**11.3**	×
soc-youtube-snap	**0.4**	2.6	**0.5**	6.4	**0.3**	3.4	**0.3**	3.5	**0.3**	5.8	**0.3**	24.5
soc-lastfm	**3.6**	34.5	**1.9**	44.5	**1.5**	51.1	**2.2**	80.6	**2.4**	189.3	**3.5**	×
soc-pokec	**5.4**	×	5.1	**4.5**	6.1	**4.8**	4.0	**3.7**	4.5	**4.1**	**3.4**	3.5
socfb-B-anon	**7.4**	11.8	**7.1**	14.1	**7.4**	11.9	**7.8**	24.1	**8.2**	17.3	**6.7**	20.1
soc-orkut	**96.1**	×	**79.1**	×	**611.5**	×	×	×	**86.9**	×	**68.8**	×
socfb-A-anon	**7.8**	11.0	6.5	**5.4**	6.2	**4.9**	7.0	**3.9**	6.9	**4.1**	5.3	**4.1**
wikipedia-link-en	**12.1**	19.5	**12.0**	92.6	**12.9**	433.9	**15.3**	×	**19.1**	×	**247.1**	×
dbpedia-link	**76.0**	254.5	**52.2**	×	**50.5**	×	**49.3**	×	**696.0**	×	×	×
wikipedia-link-en13	249.0	**240.0**	**290.7**	321.6	**249.3**	372.1	**290.9**	444.4	**245.3**	539.7	**304.4**	×
delicious-ui	**62.1**	450.3	**99.0**	×	**101.2**	×	**119.3**	×	×	×	×	×
soc-sinaweibo	**141.2**	×	**126.3**	×	**207.3**	×	**1661.7**	×	×	×	×	×
web-ClueWeb09	**43.8**	×	**70.5**	×	**613.2**	×	×	×	×	×	×	×

Dataset	$k = 8$		$k = 9$		$k = 10$		$k = 15$		$k = 20$	
	w/ TwoHop	w/o TwoHop	w/ TwoHop	w/o TwoHop	w/ TwoHop	w/o TwoHop	w/ TwoHop	w/o TwoHop	w/ TwoHop	w/o TwoHop
hamming6-2	0.0	**0.0**	0.0	**0.0**	0.0	**0.0**	0.0	**0.0**	0.0	**0.0**
johnson8-4-4	×	×	**81.3**	×	**1271.9**	×	**20.4**	116.4	0.0	**0.0**
keller4	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	**0.6**	×	**0.7**	×	**0.6**	×	**2.7**	×	**1.7**	×
ia-wiki-Talk	**0.9**	×	**1.4**	×	**1.5**	×	**3.6**	×	**23.7**	×
soc-buzznet	**75.4**	×	**106.7**	×	**150.3**	×	**520.7**	×	**701.7**	×
soc-LiveMocha	**4.1**	×	**2.3**	×	**2.8**	×	**0.9**	×	**3.2**	×
soc-gowalla	**0.1**	52.1	**0.1**	204.6	**0.1**	×	**0.2**	×	**0.4**	×
sc-msdoor	**2.3**	×	**3.6**	×	**3.9**	×	**3.4**	×	**5.3**	×
soc-youtube	**0.3**	154.0	**0.3**	557.2	**0.3**	×	**0.9**	×	**2.8**	×
soc-digg	**49.1**	×	**46.7**	×	**35.1**	×	**7.9**	×	**7.4**	×
sc-ldoor	**7.1**	×	**8.0**	×	**8.6**	×	**7.3**	×	**10.1**	×
soc-youtube-snap	**0.4**	407.9	**0.6**	×	**0.5**	×	**1.2**	×	**4.6**	×
soc-lastfm	**1.1**	×	**1.4**	×	**1.6**	×	**5.6**	×	**21.3**	×
soc-pokec	**3.2**	3.6	8.1	**3.7**	**3.1**	7.3	**3.3**	×	**3.4**	×
socfb-B-anon	**5.6**	49.0	**6.7**	14.5	**7.0**	44.0	**6.3**	×	**6.7**	×
soc-orkut	**49.9**	×	**49.7**	×	**46.1**	×	**33.6**	×	**37.2**	×
socfb-A-anon	5.7	**4.3**	4.8	**3.7**	4.9	**3.6**	**4.8**	×	**3.7**	×
wikipedia-link-en	**69.0**	×	**29.6**	×	**12.0**	×	9.9	**7.4**	**7.2**	7.4
dbpedia-link	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	**284.1**	×	**255.5**	×	**292.3**	×	**297.2**	×	**277.6**	×
delicious-ui	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×

Table 14 compares our default configuration of U-MkP with the version with top-level **CTCP** being disabled. We can see that disabling **CTCP** at the top level does not cause much slowdown, indicating that its pruning effect is mostly covered also by other techniques. However, since top-level CTCP has a low overhead and is slightly beneficial in most cases, we still in enable it by default.

Table 14: Comparison of the Default Configuration with the Variant Without Top-Level CTCP Pruning

Dataset	$k=2$		$k=3$		$k=4$		$k=5$		$k=6$		$k=7$	
	w/ CTCP^T	w/o CTCP^T	w/ CTCP^T	w/o CTCP^T	w/ CTCP^T	w/o CTCP^T	w/ CTCP^T	w/o CTCP^T	w/ CTCP^T	w/o CTCP^T	w/ CTCP^T	w/o CTCP^T
hamming6-2	**0.2**	0.2	**35.4**	40.4	**36.9**	42.5	**3.6**	4.4	**16.9**	18.3	0.0	**0.0**
johnson8-4-4	**0.7**	0.8	**5.7**	6.2	**29.6**	32.0	**2.0**	2.3	**332.4**	396.4	×	×
keller4	**25.2**	27.2	**28.9**	31.7	**1504.8**	1585.2	**1120.0**	1263.8	×	×	×	×
brock200-2	**4.2**	4.5	**21.4**	22.7	**71.6**	80.5	**715.0**	809.9	×	×	×	×
p-hat500-1	**14.4**	14.7	**29.6**	32.0	**70.2**	77.9	**242.0**	265.4	×	×	×	×
socfb-Duke14	1.2	**1.2**	**0.8**	0.9	**1.4**	1.4	**2.6**	3.5	**0.6**	0.7	**0.6**	0.6
ia-wiki-Talk	**0.8**	0.8	**0.6**	0.6	0.4	**0.4**	**0.7**	0.7	**0.4**	0.4	**0.5**	0.5
soc-buzznet	**358.0**	373.5	**189.1**	201.5	**153.0**	174.6	1771.0	**1749.0**	**123.7**	139.0	**76.1**	91.4
soc-LiveMocha	**3.3**	3.4	**2.2**	2.4	**2.3**	2.6	**7.0**	8.1	**3.1**	3.4	**4.1**	4.4
soc-gowalla	0.1	**0.1**	**0.1**	0.1	**0.1**	0.1	**0.1**	0.1	0.1	**0.1**	**0.1**	0.1
sc-msdoor	7.3	**7.1**	**12.4**	12.4	**25.0**	26.4	**40.8**	44.8	**28.8**	32.4	**6.1**	6.3
soc-youtube	**0.2**	0.3	**0.3**	0.3	**0.3**	0.3	**0.3**	0.3	**0.2**	0.2	**0.2**	0.2
soc-digg	**121.9**	127.7	**49.4**	55.4	**33.2**	36.6	**102.4**	126.6	**35.4**	38.7	**54.5**	59.9
sc-ldoor	**14.1**	14.4	**24.6**	25.7	56.6	**50.6**	**71.3**	77.9	**50.4**	55.4	**11.3**	11.4
soc-youtube-snap	**0.4**	0.4	**0.5**	0.5	**0.3**	0.4	0.3	**0.3**	**0.3**	0.3	**0.3**	0.4
soc-lastfm	**3.6**	3.8	**1.9**	2.1	1.5	**1.5**	**2.2**	2.4	**2.4**	2.6	**3.5**	3.7
soc-pokec	**5.4**	7.9	**5.1**	5.2	6.1	**4.9**	**4.0**	5.0	4.5	**4.5**	**3.4**	4.5
socfb-B-anon	**7.4**	9.2	**7.1**	7.6	**7.4**	9.3	7.8	**7.0**	8.2	**6.9**	**6.7**	7.8
soc-orkut	96.1	**95.5**	**79.1**	87.4	**611.5**	757.0	×	×	86.9	**86.4**	68.8	**58.2**
socfb-A-anon	**7.8**	9.5	**6.5**	8.7	**6.2**	8.0	7.0	**6.9**	**6.9**	7.2	**5.3**	7.1
wikipedia-link-en	12.1	**11.7**	**12.0**	12.5	12.9	**11.9**	15.3	**15.1**	**19.1**	19.2	247.1	**221.3**
dbpedia-link	76.0	**68.4**	**52.2**	58.1	**50.5**	54.4	**49.3**	53.9	**696.0**	793.5	×	×
wikipedia-link-en13	**249.0**	266.7	290.7	**261.8**	249.3	**241.7**	290.9	**290.2**	245.3	**241.8**	304.4	**292.1**
delicious-ui	62.1	**52.8**	**99.0**	118.4	**101.2**	117.0	**119.3**	141.3	×	×	×	×
soc-sinaweibo	141.2	**131.4**	**126.3**	133.1	**207.3**	238.3	**1661.7**	×	×	×	×	×
web-ClueWeb09	**43.8**	46.5	**70.5**	73.0	**613.2**	706.8	×	×	×	×	×	×

Dataset	$k=8$		$k=9$		$k=10$		$k=15$		$k=20$	
	w/ CTCP^T	w/o CTCP^T	w/ CTCP^T	w/o CTCP^T	w/ CTCP^T	w/o CTCP^T	w/ CTCP^T	w/o CTCP^T	w/ CTCP^T	w/o CTCP^T
hamming6-2	0.0	**0.0**	**0.0**	0.0	0.0	**0.0**	0.0	**0.0**	0.0	**0.0**
johnson8-4-4	×	×	**81.3**	107.8	**1271.9**	1671.4	**20.4**	24.2	0.0	**0.0**
keller4	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	**0.6**	0.7	**0.7**	0.7	**0.6**	0.7	**2.7**	3.3	**1.7**	2.0
ia-wiki-Talk	**0.9**	1.0	**1.4**	1.5	**1.5**	1.7	**3.6**	4.1	**23.7**	27.4
soc-buzznet	**75.4**	83.9	**106.7**	122.0	**150.3**	176.9	**520.7**	597.5	**701.7**	827.0
soc-LiveMocha	**4.1**	4.7	**2.3**	2.5	**2.8**	3.3	**0.9**	1.0	**3.2**	3.7
soc-gowalla	0.1	**0.1**	**0.1**	0.1	**0.1**	0.1	0.2	**0.2**	**0.4**	0.4
sc-msdoor	2.3	**2.3**	3.6	**3.4**	3.9	**3.8**	3.4	**3.0**	**5.3**	5.7
soc-youtube	**0.3**	0.3	**0.3**	0.3	**0.3**	0.3	**0.9**	1.0	**2.8**	3.3
soc-digg	**49.1**	55.6	**46.7**	54.7	**35.1**	42.2	**7.9**	9.2	**7.4**	7.8
sc-ldoor	**7.1**	7.1	8.0	**7.7**	8.6	**8.1**	7.3	**6.8**	10.1	**9.8**
soc-youtube-snap	**0.4**	0.4	0.6	**0.5**	**0.5**	0.6	**1.2**	1.2	**4.6**	5.3
soc-lastfm	**1.1**	1.2	**1.4**	1.5	**1.6**	1.7	**5.6**	6.4	**21.3**	24.6
soc-pokec	3.2	**3.1**	8.1	**3.6**	**3.1**	3.4	3.3	**2.8**	**3.4**	3.7
socfb-B-anon	**5.6**	7.4	6.7	**5.5**	7.0	**5.5**	6.3	**5.9**	**6.7**	7.9
soc-orkut	49.9	**44.3**	49.7	**47.5**	**46.1**	50.2	**33.6**	35.8	**37.2**	39.3
socfb-A-anon	**5.7**	6.2	**4.8**	6.4	4.9	**4.8**	4.8	**4.8**	**3.7**	4.2
wikipedia-link-en	69.0	**62.2**	29.6	**26.4**	**12.0**	14.1	**9.9**	10.7	**7.2**	7.8
dbpedia-link	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	284.1	**270.5**	**255.5**	292.8	**292.3**	304.0	**297.2**	298.5	277.6	**247.5**
delicious-ui	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×

However, we observed that applying CTCP inside the BB(.) procedure (i.e., in reduce_and_prune(.)), as kPlexS [34] does, is expensive. Table 15 shows that, when enabling CTCP inside BB(.), the execution time was increased by one to two orders of magnitude for many datasets. Thus, U-MkP disables it in BB(.) by default.

Table 15: Comparison of the Default Configuration with the Variant Without Recursive CTCP Pruning

Dataset	$k=2$		$k=3$		$k=4$		$k=5$		$k=6$		$k=7$	
	w/ CTCP^R	w/o CTCP^R	w/ CTCP^R	w/o CTCP^R	w/ CTCP^R	w/o CTCP^R	w/ CTCP^R	w/o CTCP^R	w/ CTCP^R	w/o CTCP^R	w/ CTCP^R	w/o CTCP^R
hamming6-2	1.2	**0.2**	235.3	**35.4**	228.8	**36.9**	19.2	**3.6**	72.4	**16.9**	**0.0**	0.0
johnson8-4-4	5.9	**0.7**	42.8	**5.7**	397.8	**29.6**	644.4	**2.0**	1603.8	**332.4**	×	×
keller4	382.3	**25.2**	1042.3	**28.9**	×	**1504.8**	×	**1120.0**	×	×	×	×
brock200-2	74.0	**4.2**	533.1	**21.4**	×	**71.6**	×	**715.0**	×	×	×	×
p-hat500-1	429.7	**14.4**	679.6	**29.6**	1546.3	**70.2**	×	**242.0**	×	×	×	×
socfb-Duke14	13.8	**1.2**	6.8	**0.8**	5.3	**1.4**	8.3	**2.6**	1.8	**0.6**	1.4	**0.6**
ia-wiki-Talk	7.2	**0.8**	4.9	**0.6**	2.6	**0.4**	2.4	**0.7**	0.9	**0.4**	0.9	**0.5**
soc-buzznet	×	**358.0**	×	**189.1**	×	**153.0**	×	**1771.0**	×	**123.7**	1763.1	**76.1**
soc-LiveMocha	54.5	**3.3**	20.9	**2.2**	16.2	**2.3**	34.4	**7.0**	14.6	**3.1**	16.2	**4.1**
soc-gowalla	**0.1**	0.1	0.1	**0.1**	0.1	**0.1**	0.1	**0.1**	**0.1**	0.1	0.1	**0.1**
sc-msdoor	12.8	**7.3**	28.0	**12.4**	82.7	**25.0**	152.6	**40.8**	82.6	**28.8**	8.2	**6.1**
soc-youtube-snap	0.3	**0.2**	0.4	**0.3**	**0.2**	0.3	0.3	**0.3**	**0.2**	0.2	0.2	**0.2**
soc-digg	×	**121.9**	×	**49.4**	939.0	**33.2**	791.8	**102.4**	306.3	**35.4**	285.8	**54.5**
sc-ldoor	21.7	**14.1**	54.2	**24.6**	153.9	**56.6**	245.5	**71.3**	135.8	**50.4**	13.9	**11.3**
soc-youtube	0.4	**0.4**	0.7	**0.5**	**0.3**	0.3	0.4	**0.3**	**0.3**	0.3	**0.3**	0.3
soc-lastfm	53.2	**3.6**	17.6	**1.9**	9.2	**1.5**	9.9	**2.2**	6.2	**2.4**	6.7	**3.5**
soc-pokec	6.7	**5.4**	**4.9**	5.1	**4.8**	6.1	4.4	**4.0**	**3.5**	4.5	**3.3**	3.4
socfb-B-anon	8.4	**7.4**	9.7	**7.1**	8.6	**7.4**	**6.7**	7.8	**8.0**	8.2	7.6	**6.7**
soc-orkut	398.1	**96.1**	287.9	**79.1**	×	**611.5**	×	×	239.9	**86.9**	105.7	**68.8**
socfb-A-anon	9.0	**7.8**	7.9	**6.5**	6.9	**6.2**	7.2	**7.0**	**5.8**	6.9	5.5	**5.3**
wikipedia-link-en	216.5	**12.1**	264.8	**12.0**	411.3	**12.9**	796.1	**15.3**	×	**19.1**	×	**247.1**
dbpedia-link	874.4	**76.0**	488.0	**52.2**	463.3	**50.5**	442.4	**49.3**	×	**696.0**	×	×
wikipedia-link-en13	259.7	**249.0**	**260.0**	290.7	**225.8**	249.3	**238.2**	290.9	**229.2**	245.3	**253.3**	304.4
delicious-ui	**61.7**	62.1	231.3	**99.0**	243.5	**101.2**	×	**119.3**	×	×	×	×
soc-sinaweibo	1225.5	**141.2**	1172.6	**126.3**	1044.3	**207.3**	×	**1661.7**	×	×	×	×
web-ClueWeb09	346.4	**43.8**	628.6	**70.5**	×	**613.2**	×	×	×	×	×	×

Dataset	$k=8$		$k=9$		$k=10$		$k=15$		$k=20$	
	w/ CTCP^R	w/o CTCP^R	w/ CTCP^R	w/o CTCP^R	w/ CTCP^R	w/o CTCP^R	w/ CTCP^R	w/o CTCP^R	w/ CTCP^R	w/o CTCP^R
hamming6-2	**0.0**	0.0	**0.0**	0.0	0.0	**0.0**	**0.0**	0.0	0.0	**0.0**
johnson8-4-4	×	×	297.7	**81.3**	×	**1271.9**	41.2	**20.4**	**0.0**	0.0
keller4	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	1.4	**0.6**	1.5	**0.7**	1.7	**0.6**	2.8	**2.7**	2.0	**1.7**
ia-wiki-Talk	2.1	**0.9**	2.6	**1.4**	2.9	**1.5**	8.6	**3.6**	97.0	**23.7**
soc-buzznet	1464.5	**75.4**	1363.6	**106.7**	×	**150.3**	×	**520.7**	×	**701.7**
soc-LiveMocha	15.9	**4.1**	7.5	**2.3**	7.5	**2.8**	1.7	**0.9**	5.1	**3.2**
soc-gowalla	**0.1**	0.1	**0.1**	0.1	0.1	**0.1**	0.2	**0.2**	2.0	**0.4**
sc-msdoor	2.5	**2.3**	**3.4**	3.6	3.9	**3.9**	**3.1**	3.4	**4.7**	5.3
soc-youtube-snap	0.3	**0.3**	0.3	**0.3**	0.3	**0.3**	1.6	**0.9**	18.7	**2.8**
soc-digg	76.0	**49.1**	99.2	**46.7**	122.3	**35.1**	8.5	**7.9**	8.1	**7.4**
sc-ldoor	**7.1**	7.1	**7.4**	8.0	**8.2**	8.6	**6.4**	7.3	**9.6**	10.1
soc-youtube	0.4	**0.4**	**0.5**	0.6	0.6	**0.5**	3.5	**1.2**	25.0	**4.6**
soc-lastfm	1.9	**1.1**	1.8	**1.4**	2.6	**1.6**	10.5	**5.6**	**20.6**	21.3
soc-pokec	3.5	**3.2**	**3.6**	8.1	4.0	**3.1**	**2.9**	3.3	4.6	**3.4**
socfb-B-anon	7.6	**5.6**	**5.5**	6.7	**6.1**	7.0	6.3	**6.3**	6.8	**6.7**
soc-orkut	**49.5**	49.9	54.5	**49.7**	47.6	**46.1**	34.6	**33.6**	38.6	**37.2**
socfb-A-anon	**5.6**	5.7	6.2	**4.8**	5.9	**4.9**	5.3	**4.8**	4.3	**3.7**
wikipedia-link-en	×	**69.0**	×	**29.6**	303.2	**12.0**	18.0	**9.9**	7.3	**7.2**
dbpedia-link	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	299.1	**284.1**	266.5	**255.5**	324.2	**292.3**	**259.1**	297.2	**271.7**	277.6
delicious-ui	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×

Tables 16–18 report the effect of **RR1**–**RR3**, and we can see that they can significantly speed up computation, so we enable them by default. Note that the conditions of RR1–RR3 are very efficient to check, so they incur negligible overhead.

Table 16: Comparison of the Default Configuration with the Variant Without RR1 Pruning

Dataset	$k=2$		$k=3$		$k=4$		$k=5$		$k=6$		$k=7$	
	w/ RR1	w/o RR1	w/ RR1	w/o RR1	w/ RR1	w/o RR1	w/ RR1	w/o RR1	w/ RR1	w/o RR1	w/ RR1	w/o RR1
hamming6-2	**0.2**	0.3	**35.4**	267.3	**36.9**	×	**3.6**	×	**16.9**	×	**0.0**	0.0
johnson8-4-4	**0.7**	4.1	**5.7**	19.5	**29.6**	65.1	**2.0**	2.3	**332.4**	380.4	×	×
keller4	**25.2**	83.8	**28.9**	83.1	**1504.8**	×	**1120.0**	×	×	×	×	×
brock200-2	**4.2**	11.0	**21.4**	60.9	**71.6**	×	**715.0**	×	×	×	×	×
p-hat500-1	**14.4**	23.1	**29.6**	37.2	**70.2**	100.9	**242.0**	529.1	×	×	×	×
socfb-Duke14	**1.2**	6.3	**0.8**	2.5	**1.4**	153.4	**2.6**	×	**0.6**	0.6	**0.6**	0.6
ia-wiki-Talk	0.8	**0.6**	**0.6**	0.6	**0.4**	0.5	**0.7**	1.4	**0.4**	0.5	**0.5**	0.5
soc-buzznet	**358.0**	387.3	**189.1**	×	**153.0**	×	**1771.0**	×	**123.7**	128.0	**76.1**	83.0
soc-LiveMocha	3.3	**2.7**	**2.2**	2.4	**2.3**	3.2	**7.0**	39.7	**3.1**	3.3	**4.1**	4.2
soc-gowalla	0.1	**0.1**	0.1	**0.1**	**0.1**	0.1	**0.1**	0.1	**0.1**	0.1	**0.1**	0.1
sc-msdoor	**7.3**	8.7	**12.4**	60.1	**25.0**	×	**40.8**	×	**28.8**	172.7	**6.1**	6.8
soc-youtube	**0.2**	0.3	**0.3**	0.3	**0.3**	0.3	0.3	**0.3**	**0.2**	0.2	**0.2**	0.3
soc-digg	**121.9**	×	**49.4**	118.1	**33.2**	196.2	**102.4**	×	**35.4**	38.6	**54.5**	58.7
sc-ldoor	**14.1**	15.9	**24.6**	79.1	**56.6**	×	**71.3**	×	**50.4**	252.4	**11.3**	12.7
soc-youtube-snap	**0.4**	0.4	0.5	**0.4**	**0.3**	0.4	**0.3**	0.4	0.3	**0.3**	**0.3**	0.4
soc-lastfm	3.6	**2.7**	**1.9**	2.0	**1.5**	1.6	**2.2**	3.3	**2.4**	2.6	**3.5**	3.6
soc-pokec	**5.4**	6.9	**5.1**	5.2	6.1	**6.0**	**4.0**	6.0	**4.5**	4.6	**3.4**	4.3
socfb-B-anon	**7.4**	7.7	**7.1**	8.1	**7.4**	7.9	**7.8**	8.3	**8.2**	8.5	**6.7**	8.0
soc-orkut	**96.1**	×	**79.1**	210.5	**611.5**	×	×	×	**86.9**	91.3	68.8	**66.0**
socfb-A-anon	**7.8**	10.7	**6.5**	8.7	**6.2**	7.7	**7.0**	7.4	**6.9**	7.8	**5.3**	5.6
wikipedia-link-en	**12.1**	13.8	**12.0**	13.2	12.9	**11.7**	15.3	**14.8**	**19.1**	19.7	**247.1**	254.4
dbpedia-link	76.0	**67.2**	**52.2**	63.3	50.5	**48.8**	**49.3**	50.7	**696.0**	×	×	×
wikipedia-link-en13	**249.0**	282.1	290.7	**279.7**	249.3	**236.4**	**290.9**	293.9	**245.3**	279.1	304.4	**276.7**
delicious-ui	**62.1**	63.0	99.0	**96.1**	**101.2**	116.3	**119.3**	×	×	×	×	×
soc-sinaweibo	**141.2**	×	**126.3**	603.4	**207.3**	×	**1661.7**	×	×	×	×	×
web-ClueWeb09	**43.8**	×	**70.5**	×	**613.2**	×	×	×	×	×	×	×

Dataset	$k=8$		$k=9$		$k=10$		$k=15$		$k=20$	
	w/ RR1	w/o RR1	w/ RR1	w/o RR1	w/ RR1	w/o RR1	w/ RR1	w/o RR1	w/ RR1	w/o RR1
hamming6-2	**0.0**	0.0	**0.0**	0.0	**0.0**	0.0	**0.0**	0.0	**0.0**	0.0
johnson8-4-4	×	×	**81.3**	119.1	**1271.9**	×	**20.4**	22.8	0.0	**0.0**
keller4	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	**0.6**	0.6	**0.7**	0.7	**0.6**	0.7	**2.7**	3.2	**1.7**	2.0
ia-wiki-Talk	**0.9**	1.0	**1.4**	1.4	**1.5**	1.6	**3.6**	4.0	**23.7**	27.0
soc-buzznet	**75.4**	78.6	**106.7**	114.0	**150.3**	183.6	**520.7**	593.3	**701.7**	×
soc-LiveMocha	**4.1**	4.4	**2.3**	2.4	**2.8**	3.1	**0.9**	1.0	**3.2**	3.6
soc-gowalla	**0.1**	0.1	0.1	**0.1**	0.1	**0.1**	**0.2**	0.2	**0.4**	0.4
sc-msdoor	2.3	**2.3**	**3.6**	3.6	**3.9**	3.9	3.4	**3.3**	**5.3**	6.0
soc-youtube	**0.3**	0.3	**0.3**	0.3	**0.3**	0.3	**0.9**	2.0	**2.8**	3.3
soc-digg	**49.1**	54.0	**46.7**	59.3	**35.1**	40.6	**7.9**	8.1	**7.4**	9.5
sc-ldoor	7.1	**7.0**	**8.0**	8.1	**8.6**	8.7	7.3	**7.2**	**10.1**	10.3
soc-youtube-snap	**0.4**	0.4	0.6	**0.5**	**0.5**	0.6	**1.2**	1.3	**4.6**	5.3
soc-lastfm	**1.1**	1.2	**1.4**	1.4	**1.6**	1.7	**5.6**	6.3	**21.3**	24.2
soc-pokec	**3.2**	3.8	8.1	**4.8**	**3.1**	4.7	3.3	**2.9**	**3.4**	4.6
socfb-B-anon	**5.6**	8.2	6.7	**6.1**	7.0	**6.0**	6.3	**5.1**	**6.7**	8.9
soc-orkut	49.9	**43.7**	49.7	**42.8**	**46.1**	53.0	**33.6**	39.8	37.2	**32.7**
socfb-A-anon	**5.7**	6.3	**4.8**	6.0	**4.9**	5.6	**4.8**	5.4	**3.7**	3.9
wikipedia-link-en	**69.0**	80.2	**29.6**	40.0	**12.0**	13.9	9.9	**9.9**	7.2	**7.2**
dbpedia-link	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	284.1	**273.2**	**255.5**	281.2	**292.3**	328.1	297.2	**245.7**	**277.6**	278.7
delicious-ui	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×

Table 17: Comparison of the Default Configuration with the Variant Without RR2 Pruning

Dataset	k = 2		k = 3		k = 4		k = 5		k = 6		k = 7	
	w/ RR2	w/o RR2	w/ RR2	w/o RR2	w/ RR2	w/o RR2	w/ RR2	w/o RR2	w/ RR2	w/o RR2	w/ RR2	w/o RR2
hamming6-2	**0.2**	43.7	**35.4**	×	**36.9**	×	**3.6**	×	**16.9**	209.9	**0.0**	0.0
johnson8-4-4	**0.7**	0.7	**5.7**	6.0	**29.6**	32.0	**2.0**	2.2	**332.4**	440.3	×	×
keller4	**25.2**	25.5	**28.9**	29.4	**1504.8**	×	**1120.0**	×	×	×	×	×
brock200-2	4.2	**4.2**	**21.4**	22.3	**71.6**	76.2	**715.0**	×	×	×	×	×
p-hat500-1	14.4	**13.7**	29.6	**29.6**	**70.2**	73.0	**242.0**	254.7	×	×	×	×
socfb-Duke14	**1.2**	1.5	**0.8**	0.9	**1.4**	1.5	**2.6**	3.0	**0.6**	0.7	**0.6**	0.6
ia-wiki-Talk	0.8	**0.8**	**0.6**	0.6	**0.4**	0.4	**0.7**	0.7	**0.4**	0.5	**0.5**	0.5
soc-buzznet	**358.0**	396.1	**189.1**	×	**153.0**	166.1	**1771.0**	×	**123.7**	129.9	**76.1**	86.7
soc-LiveMocha	3.3	**3.3**	**2.2**	2.2	**2.3**	2.5	**7.0**	7.8	**3.1**	3.4	**4.1**	4.4
soc-gowalla	0.1	**0.1**	**0.1**	0.1	0.1	**0.1**	**0.1**	0.1	**0.1**	0.1	**0.1**	0.1
sc-msdoor	7.3	**7.1**	**12.4**	12.4	**25.0**	26.2	**40.8**	43.6	**28.8**	38.3	**6.1**	6.1
soc-youtube	**0.2**	0.3	**0.3**	0.3	**0.3**	0.3	0.3	**0.3**	**0.2**	0.2	**0.2**	0.3
soc-digg	**121.9**	506.0	**49.4**	×	**33.2**	×	**102.4**	×	**35.4**	×	**54.5**	×
sc-ldoor	14.1	**14.1**	24.6	**24.5**	56.6	**49.9**	**71.3**	75.4	**50.4**	64.9	**11.3**	11.3
soc-youtube-snap	**0.4**	0.4	**0.5**	0.5	**0.3**	0.4	**0.3**	0.4	0.3	**0.3**	**0.3**	0.4
soc-lastfm	**3.6**	3.6	**1.9**	2.0	1.5	**1.5**	**2.2**	2.4	**2.4**	2.6	**3.5**	3.8
soc-pokec	**5.4**	7.9	**5.1**	6.1	**6.1**	6.7	**4.0**	4.2	**4.5**	5.0	**3.4**	4.8
socfb-B-anon	**7.4**	8.6	**7.1**	10.1	**7.4**	9.0	7.8	**7.0**	**8.2**	9.3	6.7	**6.1**
soc-orkut	**96.1**	×	**79.1**	×	**611.5**	×	×	×	**86.9**	92.7	**68.8**	70.6
socfb-A-anon	**7.8**	9.5	**6.5**	7.4	**6.2**	8.9	**7.0**	7.7	6.9	**6.7**	**5.3**	6.4
wikipedia-link-en	**12.1**	14.2	**12.0**	×	**12.9**	×	**15.3**	×	**19.1**	×	**247.1**	×
dbpedia-link	76.0	**73.7**	**52.2**	65.3	**50.5**	52.6	49.3	**46.1**	**696.0**	×	×	×
wikipedia-link-en13	**249.0**	261.0	290.7	**278.6**	249.3	**248.0**	**290.9**	295.8	**245.3**	275.7	304.4	**261.9**
delicious-ui	**62.1**	64.9	**99.0**	114.0	**101.2**	123.3	**119.3**	135.9	×	×	×	×
soc-sinaweibo	**141.2**	×	**126.3**	×	**207.3**	×	**1661.7**	×	×	×	×	×
web-ClueWeb09	**43.8**	×	**70.5**	×	**613.2**	×	×	×	×	×	×	×

Dataset	k = 8		k = 9		k = 10		k = 15		k = 20	
	w/ RR2	w/o RR2	w/ RR2	w/o RR2	w/ RR2	w/o RR2	w/ RR2	w/o RR2	w/ RR2	w/o RR2
hamming6-2	**0.0**	0.0	**0.0**	0.0	0.0	**0.0**	0.0	**0.0**	**0.0**	0.0
johnson8-4-4	×	×	**81.3**	112.1	**1271.9**	×	**20.4**	×	0.0	**0.0**
keller4	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	**0.6**	0.6	**0.7**	0.7	**0.6**	0.7	**2.7**	3.4	**1.7**	2.1
ia-wiki-Talk	**0.9**	1.0	**1.4**	1.5	**1.5**	1.7	**3.6**	4.0	**23.7**	27.9
soc-buzznet	**75.4**	80.7	**106.7**	125.5	**150.3**	193.9	**520.7**	×	**701.7**	×
soc-LiveMocha	**4.1**	4.6	**2.3**	2.5	**2.8**	3.3	**0.9**	1.0	**3.2**	3.7
soc-gowalla	0.1	**0.1**	0.1	**0.1**	**0.1**	0.1	**0.2**	0.2	**0.4**	0.4
sc-msdoor	2.3	**2.3**	3.6	**3.4**	3.9	**3.9**	**3.4**	3.4	**5.3**	6.8
soc-youtube	**0.3**	0.3	**0.3**	0.3	**0.3**	0.3	**0.9**	2.0	**2.8**	3.3
soc-digg	**49.1**	59.7	**46.7**	62.6	**35.1**	43.2	**7.9**	8.8	**7.4**	9.4
sc-ldoor	**7.1**	7.1	8.0	**7.9**	**8.6**	8.7	**7.3**	7.3	10.1	**10.1**
soc-youtube-snap	0.4	**0.4**	0.6	**0.5**	**0.5**	0.6	**1.2**	1.4	**4.6**	5.5
soc-lastfm	**1.1**	1.3	**1.4**	1.5	**1.6**	1.7	**5.6**	6.7	**21.3**	25.3
soc-pokec	**3.2**	4.4	8.1	**4.1**	3.1	**3.1**	3.3	**3.0**	**3.4**	4.8
socfb-B-anon	**5.6**	7.8	**6.7**	6.7	7.0	**6.5**	6.3	**5.4**	**6.7**	7.0
soc-orkut	49.9	**48.9**	**49.7**	52.3	**46.1**	49.0	**33.6**	38.9	37.2	**36.9**
socfb-A-anon	**5.7**	6.0	**4.8**	5.3	4.9	**4.6**	**4.8**	5.6	**3.7**	4.5
wikipedia-link-en	**69.0**	×	**29.6**	×	**12.0**	×	**9.9**	10.9	**7.2**	8.1
dbpedia-link	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	**284.1**	323.3	**255.5**	339.1	**292.3**	307.4	297.2	**292.3**	277.6	**234.9**
delicious-ui	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×

Table 18: Comparison of the Default Configuration with the Variant Without RR3 Pruning

Dataset	$k = 2$		$k = 3$		$k = 4$		$k = 5$		$k = 6$		$k = 7$	
	w/ RR3	w/o RR3	w/ RR3	w/o RR3	w/ RR3	w/o RR3	w/ RR3	w/o RR3	w/ RR3	w/o RR3	w/ RR3	w/o RR3
hamming6-2	0.2	**0.1**	35.4	**21.7**	**36.9**	×	**3.6**	60.1	**16.9**	117.6	0.0	**0.0**
johnson8-4-4	**0.7**	×	**5.7**	×	**29.6**	×	**2.0**	10.7	**332.4**	×	×	×
keller4	**25.2**	×	**28.9**	×	**1504.8**	×	**1120.0**	×	×	×	×	×
brock200-2	**4.2**	×	**21.4**	×	**71.6**	×	**715.0**	×	×	×	×	×
p-hat500-1	**14.4**	×	**29.6**	×	**70.2**	×	**242.0**	×	×	×	×	×
socfb-Duke14	**1.2**	1.6	**0.8**	1.6	**1.4**	3.0	**2.6**	8.6	**0.6**	0.8	**0.6**	0.8
ia-wiki-Talk	**0.8**	0.9	**0.6**	0.8	**0.4**	0.6	**0.7**	0.9	**0.4**	0.7	**0.5**	0.7
soc-buzznet	**358.0**	×	**189.1**	×	**153.0**	352.2	**1771.0**	×	**123.7**	233.2	**76.1**	151.6
soc-LiveMocha	**3.3**	×	**2.2**	3.2	**2.3**	3.4	**7.0**	×	**3.1**	4.7	**4.1**	5.9
soc-gowalla	0.1	**0.1**	**0.1**	0.1	0.1	**0.1**	**0.1**	0.1	**0.1**	0.1	**0.1**	0.1
sc-msdoor	**7.3**	7.5	**12.4**	13.2	**25.0**	27.2	**40.8**	51.4	**28.8**	36.2	**6.1**	6.1
soc-youtube	**0.2**	0.3	**0.3**	0.3	**0.3**	0.3	**0.3**	0.4	**0.2**	0.2	**0.2**	0.2
soc-digg	**121.9**	160.3	**49.4**	85.7	**33.2**	106.1	**102.4**	×	**35.4**	×	**54.5**	×
sc-ldoor	**14.1**	14.4	**24.6**	25.7	56.6	**51.8**	**71.3**	92.8	**50.4**	62.3	**11.3**	11.6
soc-youtube-snap	**0.4**	0.4	**0.5**	0.5	**0.3**	0.4	0.3	**0.3**	**0.3**	0.3	**0.3**	0.4
soc-lastfm	**3.6**	4.1	**1.9**	2.9	**1.5**	2.1	**2.2**	3.3	**2.4**	4.8	**3.5**	4.8
soc-pokec	5.4	**5.3**	**5.1**	6.5	6.1	**4.9**	**4.0**	5.7	4.5	**4.2**	**3.4**	4.1
socfb-B-anon	**7.4**	8.3	**7.1**	9.2	**7.4**	7.5	**7.8**	8.0	**8.2**	8.7	**6.7**	6.9
soc-orkut	**96.1**	101.0	**79.1**	124.9	**611.5**	×	×	×	**86.9**	112.5	**68.8**	72.3
socfb-A-anon	**7.8**	9.5	**6.5**	8.5	**6.2**	6.4	**7.0**	7.4	**6.9**	7.2	**5.3**	6.3
wikipedia-link-en	12.1	**12.1**	12.0	**12.0**	**12.9**	14.6	**15.3**	20.6	**19.1**	126.1	**247.1**	×
dbpedia-link	76.0	**75.0**	**52.2**	66.8	**50.5**	65.2	**49.3**	60.1	**696.0**	×	×	×
wikipedia-link-en13	**249.0**	250.6	290.7	**265.7**	**249.3**	307.3	290.9	**235.5**	**245.3**	273.6	**304.4**	329.9
delicious-ui	**62.1**	65.2	**99.0**	139.2	**101.2**	×	**119.3**	×	×	×	×	×
soc-sinaweibo	**141.2**	187.5	**126.3**	242.2	**207.3**	×	**1661.7**	×	×	×	×	×
web-ClueWeb09	**43.8**	60.4	**70.5**	193.6	**613.2**	×	×	×	×	×	×	×

Dataset	$k = 8$		$k = 9$		$k = 10$		$k = 15$		$k = 20$	
	w/ RR3	w/o RR3	w/ RR3	w/o RR3	w/ RR3	w/o RR3	w/ RR3	w/o RR3	w/ RR3	w/o RR3
hamming6-2	0.0	**0.0**	0.0	**0.0**	**0.0**	0.0	**0.0**	0.0	0.0	**0.0**
johnson8-4-4	×	×	**81.3**	×	**1271.9**	×	**20.4**	93.4	0.0	**0.0**
keller4	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	**0.6**	0.8	**0.7**	0.9	**0.6**	1.0	**2.7**	4.7	**1.7**	3.1
ia-wiki-Talk	**0.9**	1.4	**1.4**	2.1	**1.5**	2.4	**3.6**	6.2	**23.7**	43.6
soc-buzznet	**75.4**	137.3	**106.7**	185.9	**150.3**	285.0	**520.7**	×	**701.7**	×
soc-LiveMocha	**4.1**	8.5	**2.3**	3.4	**2.8**	4.6	**0.9**	1.3	**3.2**	5.3
soc-gowalla	0.1	**0.1**	**0.1**	0.1	**0.1**	0.1	**0.2**	0.2	**0.4**	0.8
sc-msdoor	**2.3**	2.4	3.6	**3.3**	3.9	**3.7**	3.4	**3.1**	**5.3**	7.7
soc-youtube	**0.3**	0.3	**0.3**	0.3	**0.3**	0.3	**0.9**	1.6	**2.8**	6.5
soc-digg	**49.1**	82.4	**46.7**	80.2	**35.1**	64.2	**7.9**	9.1	**7.4**	8.3
sc-ldoor	7.1	**7.1**	8.0	**7.7**	8.6	**8.3**	7.3	**6.7**	10.1	**9.7**
soc-youtube-snap	**0.4**	0.4	**0.6**	0.6	**0.5**	0.7	**1.2**	2.1	**4.6**	9.8
soc-lastfm	**1.1**	8.1	**1.4**	1.9	**1.6**	2.3	**5.6**	9.4	**21.3**	38.7
soc-pokec	**3.2**	4.5	8.1	**3.4**	**3.1**	3.4	**3.3**	3.5	**3.4**	3.5
socfb-B-anon	**5.6**	6.9	6.7	**6.6**	**7.0**	7.8	6.3	**6.0**	6.7	**6.7**
soc-orkut	**49.9**	59.1	**49.7**	50.7	**46.1**	53.3	**33.6**	43.6	**37.2**	39.2
socfb-A-anon	5.7	**5.4**	**4.8**	6.1	**4.9**	5.6	4.8	**4.7**	**3.7**	4.4
wikipedia-link-en	**69.0**	×	**29.6**	555.4	**12.0**	24.2	**9.9**	10.7	7.2	**6.5**
dbpedia-link	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	284.1	**283.8**	**255.5**	296.0	**292.3**	300.1	297.2	**274.1**	277.6	**250.0**
delicious-ui	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×

Table 19 reports the effect of BR1 and BR2, and to our surprise, we can see that they merely make any difference for almost all datasets and values of k, except for delicious-ui where a significant speedup is observed. This shows that their conditions seldom hold to allow pruning, but since they are efficient to check, U-MkP enables them by default.

Table 19: Effect of Reduction Rules BR1 and BR2

Dataset	*k* = 2				*k* = 3				*k* = 4				*k* = 5			
	U-M*k*P	w/o BR1	w/o BR2	w/o BR12	U-M*k*P	w/o BR1	w/o BR2	w/o BR12	U-M*k*P	w/o BR1	w/o BR2	w/o BR12	U-M*k*P	w/o BR1	w/o BR2	w/o BR12
hamming6-2	**0.2**	0.2	0.2	0.2	**35.4**	40.2	40.4	37.9	36.9	37.8	42.5	**36.3**	3.6	3.7	4.4	**3.5**
johnson8-4-4	0.7	**0.7**	0.8	0.7	5.7	**5.5**	6.2	5.6	29.6	30.6	32.0	**29.5**	2.0	2.1	2.3	**2.0**
keller4	**25.2**	25.3	27.2	25.3	28.9	29.0	31.7	**26.9**	**1504.8**	1517.1	1585.2	1542.2	**1120.0**	1127.5	1263.8	1146.2
brock200-2	**4.2**	4.2	4.5	4.3	21.4	20.9	22.7	**20.8**	71.6	**70.4**	80.5	71.7	**715.0**	736.8	809.9	742.8
p-hat500-1	14.4	**14.3**	14.7	14.4	29.6	**29.5**	32.0	29.6	70.2	**69.1**	77.9	70.7	**242.0**	242.3	265.4	246.2
socfb-Duke14	1.2	1.2	**1.2**	1.3	0.8	**0.8**	0.9	0.8	1.4	**1.0**	1.4	1.1	2.6	2.7	3.5	**2.4**
ia-wiki-Talk	**0.8**	0.8	0.8	0.8	**0.6**	0.6	0.6	0.6	0.4	**0.4**	0.4	0.4	0.7	0.7	0.7	**0.7**
soc-buzznet	**358.0**	362.5	373.5	360.8	189.1	173.6	201.5	**171.7**	153.0	124.9	174.6	**116.1**	1771.0	1378.6	1749.0	**1353.8**
soc-LiveMocha	3.3	**3.3**	3.4	3.5	2.2	**2.2**	2.4	2.3	2.3	**2.3**	2.6	2.5	7.0	7.0	8.1	**6.8**
soc-gowalla	0.1	0.1	**0.1**	0.1	0.1	**0.1**	0.1	0.1	0.1	**0.1**	0.1	0.1	**0.1**	0.1	0.1	0.1
sc-msdoor	7.3	8.4	**7.1**	8.6	**12.4**	13.8	12.4	14.6	**25.0**	32.9	26.4	32.3	**40.8**	59.8	44.8	59.2
soc-youtube	**0.2**	0.3	0.3	0.3	**0.3**	0.3	0.3	0.3	0.3	**0.2**	0.3	0.3	0.3	**0.3**	0.3	0.3
soc-digg	**121.9**	125.6	127.7	124.0	**49.4**	51.3	55.4	50.3	33.2	**29.6**	36.6	31.9	102.4	132.3	126.6	**100.3**
sc-ldoor	**14.1**	15.9	14.4	15.8	**24.6**	29.1	25.7	28.8	56.6	63.7	**50.6**	60.7	**71.3**	105.3	77.9	105.9
soc-youtube-snap	0.4	**0.4**	0.4	0.4	**0.5**	0.5	0.5	0.5	**0.3**	0.4	0.4	0.4	0.3	**0.3**	0.3	0.4
soc-lastfm	**3.6**	3.6	3.8	3.7	1.9	2.0	2.1	**1.9**	1.5	1.5	1.5	**1.5**	2.2	**2.2**	2.4	2.3
soc-pokec	5.4	**5.1**	7.9	5.5	**5.1**	6.2	5.2	5.8	6.1	5.4	4.9	**4.7**	**4.0**	4.1	5.0	4.7
socfb-B-anon	**7.4**	9.5	9.2	9.2	**7.1**	9.7	7.6	7.3	7.4	8.7	9.3	**7.1**	7.8	7.5	**7.0**	7.5
soc-orkut	96.1	**86.3**	95.5	95.7	79.1	74.7	87.4	**64.1**	611.5	437.6	757.0	**372.0**	×	×	×	×
socfb-A-anon	**7.8**	8.0	9.5	8.1	**6.5**	7.0	8.7	7.9	**6.2**	7.0	8.0	7.2	7.0	**5.8**	6.9	6.0
wikipedia-link-en	12.1	**11.0**	11.7	14.2	**12.0**	16.5	12.5	17.6	12.9	31.6	**11.9**	32.3	15.3	78.3	**15.1**	78.8
dbpedia-link	76.0	65.6	68.4	**63.8**	**52.2**	53.4	58.1	60.1	50.5	**47.9**	54.4	58.5	49.3	53.5	53.9	**46.0**
wikipedia-link-en13	**249.0**	279.2	266.7	273.8	290.7	297.2	**261.8**	272.0	249.3	244.1	**241.7**	272.7	290.9	**249.6**	290.2	293.9
delicious-ui	62.1	60.1	**52.8**	64.9	**99.0**	106.2	118.4	105.7	**101.2**	105.6	117.0	117.8	**119.3**	120.5	141.3	122.4
soc-sinaweibo	141.2	144.2	**131.4**	154.1	126.3	132.0	133.1	**113.6**	207.3	**111.4**	238.3	136.1	1661.7	542.6	×	**519.6**
web-ClueWeb09	**43.8**	193.5	46.5	186.1	**70.5**	144.8	73.0	143.5	**613.2**	1032.7	706.8	1043.0	×	×	×	×

Dataset	*k* = 6				*k* = 7				*k* = 8				*k* = 9			
	U-M*k*P	w/o BR1	w/o BR2	w/o BR12	U-M*k*P	w/o BR1	w/o BR2	w/o BR12	U-M*k*P	w/o BR1	w/o BR2	w/o BR12	U-M*k*P	w/o BR1	w/o BR2	w/o BR12
hamming6-2	17.1	18.6	**16.1**	16.2	0.0	0.0	0.0	**0.0**	0.0	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0
johnson8-4-4	**335.4**	385.5	369.5	367.1	×	×	×	×	×	×	×	×	**85.6**	93.1	100.6	95.7
keller4	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	0.7	0.7	**0.7**	0.7	**0.7**	0.7	0.7	0.7	0.7	0.7	0.7	**0.6**	0.7	0.8	**0.7**	0.7
ia-wiki-Talk	**0.4**	0.4	0.4	0.4	0.5	0.5	**0.5**	0.5	1.0	1.0	1.0	**1.0**	**1.4**	1.5	1.4	1.4
soc-buzznet	**129.8**	132.0	132.0	136.4	85.8	**84.4**	87.2	89.9	78.3	81.5	**77.4**	78.7	112.0	116.8	111.1	**110.6**
soc-LiveMocha	**3.1**	3.3	3.3	3.4	4.3	4.2	**4.2**	4.3	4.5	4.3	4.6	**4.3**	2.5	2.6	2.5	**2.4**
soc-gowalla	0.1	**0.1**	0.1	0.1	0.1	0.1	0.1	**0.1**	0.1	0.1	**0.1**	0.1	**0.1**	0.1	0.1	0.1
sc-msdoor	30.6	**30.0**	31.2	34.6	6.2	**6.0**	6.4	7.3	2.6	**2.4**	2.6	2.6	3.6	**3.5**	3.7	3.5
soc-youtube	0.3	0.3	0.2	**0.2**	0.2	**0.2**	0.3	0.2	0.3	0.3	**0.3**	0.3	0.3	0.3	0.3	**0.3**
soc-digg	**37.2**	39.7	38.8	38.8	57.0	59.5	**56.2**	67.6	52.6	57.4	**51.1**	58.5	**48.4**	55.1	50.0	55.9
sc-ldoor	**47.1**	52.1	54.3	61.8	11.7	11.9	**11.5**	12.4	7.4	7.6	**7.2**	7.6	8.3	8.1	8.4	**7.9**
soc-youtube-snap	0.4	0.3	**0.3**	0.3	0.3	0.3	**0.3**	0.4	0.5	**0.4**	0.4	0.4	**0.5**	0.5	0.5	0.5
soc-lastfm	**2.5**	2.6	2.7	2.6	3.5	**3.5**	3.7	3.5	**1.2**	1.2	1.3	1.2	1.4	1.4	1.4	**1.4**
soc-pokec	**3.7**	4.4	4.6	4.9	4.9	4.6	**3.6**	4.0	4.8	4.3	4.0	**3.9**	4.8	**3.2**	4.5	4.0
socfb-B-anon	8.8	8.8	**8.1**	8.8	9.6	7.3	**6.3**	6.4	8.8	8.3	**6.0**	6.5	**5.9**	7.4	6.6	6.7
soc-orkut	96.5	95.1	88.2	**83.5**	69.5	70.1	**68.6**	69.7	52.4	57.9	47.4	**45.5**	44.3	47.4	53.4	**43.7**
socfb-A-anon	6.7	7.6	**5.9**	7.5	6.1	7.5	7.0	**5.5**	**5.6**	6.3	7.0	6.4	5.4	6.2	6.3	**5.0**
wikipedia-link-en	**20.8**	23.7	22.6	22.3	**248.2**	260.1	256.4	255.6	**72.1**	75.8	74.4	74.8	**31.1**	34.2	33.6	33.6
dbpedia-link	**682.2**	720.2	717.3	723.1	×	×	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	**317.3**	359.6	368.9	367.4	**304.7**	371.2	360.8	378.3	**291.5**	334.5	381.1	372.8	**308.6**	389.5	385.0	383.8
delicious-ui	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

Dataset	*k* = 10				*k* = 15				*k* = 20			
	U-M*k*P	w/o BR1	w/o BR2	w/o BR12	U-M*k*P	w/o BR1	w/o BR2	w/o BR12	U-M*k*P	w/o BR1	w/o BR2	w/o BR12
hamming6-2	**0.0**	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**	**0.0**	0.0	0.0	0.0
johnson8-4-4	**1384.5**	1469.8	1590.4	1497.3	21.6	23.8	23.4	**20.4**	0.0	0.0	0.0	**0.0**
keller4	×	×	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	0.7	0.8	**0.7**	0.7	**2.8**	3.0	3.2	2.9	1.8	1.9	1.9	**1.8**
ia-wiki-Talk	**1.5**	1.6	1.6	1.5	**3.7**	4.1	4.1	3.8	24.6	27.4	27.6	**24.5**
soc-buzznet	**158.5**	166.0	159.9	164.5	543.6	588.6	543.1	**529.4**	729.2	779.7	799.7	**727.1**
soc-LiveMocha	3.0	3.3	3.3	**2.9**	0.9	0.9	**0.9**	1.1	**3.2**	3.5	3.5	3.4
soc-gowalla	0.1	**0.1**	0.1	0.1	0.2	**0.2**	0.2	0.2	**0.4**	0.4	0.4	0.4
sc-msdoor	**4.0**	4.0	4.2	4.1	**3.4**	3.4	3.6	3.4	5.5	5.4	5.9	**4.6**
soc-youtube	0.3	0.3	**0.3**	0.3	0.9	1.0	1.0	**0.9**	**3.1**	3.3	3.3	3.1
soc-digg	39.3	42.4	**36.7**	44.9	**9.2**	9.3	9.3	9.4	8.9	9.2	**7.9**	8.8
sc-ldoor	**8.3**	8.4	8.5	8.9	7.1	7.3	7.7	**6.9**	10.4	**9.7**	10.4	10.4
soc-youtube-snap	0.5	**0.5**	0.5	0.5	1.2	1.2	1.3	**1.1**	**4.3**	5.1	5.3	4.9
soc-lastfm	**1.6**	1.6	1.8	1.7	**6.0**	6.3	6.6	6.1	**22.1**	24.7	24.6	22.7
soc-pokec	4.5	4.1	**4.0**	4.3	**2.8**	3.7	3.5	3.6	4.0	**3.3**	4.5	4.0
socfb-B-anon	7.2	6.9	6.4	**6.3**	6.0	6.4	6.4	**5.3**	**6.7**	8.8	6.9	8.2
soc-orkut	56.5	50.9	**44.2**	47.7	49.1	45.0	43.9	**37.1**	34.0	42.3	41.9	**33.5**
socfb-A-anon	6.3	4.8	**4.6**	6.6	**4.7**	5.4	5.8	5.1	4.9	4.5	4.3	**4.2**
wikipedia-link-en	14.0	**13.7**	17.9	17.7	**12.7**	14.9	15.0	14.3	**8.4**	11.6	11.4	10.7
dbpedia-link	×	×	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	**325.5**	395.6	391.9	381.6	**290.2**	367.6	358.6	318.2	**331.8**	339.2	346.6	341.4
delicious-ui	×	×	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×

Table 20 shows the full results of comparison between our default U-MkP that enables UBR2 and the version that disables it. We can see that enabling UBR2 can speed up computing by up to a few orders of magnitude, and is a clear winner spanning all datasets and all values of k.

Table 20: Effect of Reduction Rule UBR2

Dataset	$k = 2$		$k = 3$		$k = 4$		$k = 5$		$k = 6$		$k = 7$	
	w/ UBR2	w/o UBR2	w/ UBR2	w/o UBR2	w/ UBR2	w/o UBR2	w/ UBR2	w/o UBR2	w/ UBR2	w/o UBR2	w/ UBR2	w/o UBR2
hamming6-2	**0.2**	0.3	**35.4**	677.2	**36.9**	×	**3.6**	498.4	**16.9**	390.0	**0.0**	0.0
johnson8-4-4	**0.7**	1.1	**5.7**	7.1	**29.6**	74.4	**2.0**	28.7	**332.4**	×	×	×
keller4	**25.2**	39.6	**28.9**	43.5	**1504.8**	×	**1120.0**	×	×	×	×	×
brock200-2	**4.2**	6.9	**21.4**	45.8	**71.6**	281.8	**715.0**	×	×	×	×	×
p-hat500-1	**14.4**	23.3	**29.6**	106.1	**70.2**	422.1	**242.0**	×	×	×	×	×
socfb-Duke14	**1.2**	5.3	**0.8**	3.7	**1.4**	14.1	**2.6**	60.5	**0.6**	×	**0.6**	×
ia-wiki-Talk	**0.8**	1.1	**0.6**	1.4	**0.4**	1.4	**0.7**	5.0	**0.4**	×	**0.5**	×
soc-buzznet	**358.0**	1171.8	**189.1**	×	**153.0**	1701.1	**1771.0**	×	**123.7**	×	**76.1**	×
soc-LiveMocha	**3.3**	5.2	**2.2**	6.7	**2.3**	7.5	**7.0**	29.7	**3.1**	×	**4.1**	×
soc-gowalla	0.1	**0.1**	0.1	**0.1**	0.1	**0.1**	**0.1**	0.1	**0.1**	2.7	**0.1**	16.7
sc-msdoor	7.3	**6.9**	**12.4**	12.7	**25.0**	33.7	**40.8**	62.1	**28.8**	433.7	**6.1**	8.5
soc-youtube	**0.2**	0.3	**0.3**	0.3	0.3	**0.2**	**0.3**	0.4	**0.2**	8.1	**0.2**	561.7
soc-digg	**121.9**	478.9	**49.4**	558.3	**33.2**	918.2	**102.4**	×	**35.4**	×	**54.5**	×
sc-ldoor	14.1	**13.7**	**24.6**	25.2	**56.6**	64.6	**71.3**	108.8	**50.4**	669.8	**11.3**	14.9
soc-youtube-snap	**0.4**	0.4	**0.5**	0.6	**0.3**	0.4	**0.3**	0.4	**0.3**	26.9	**0.3**	104.9
soc-lastfm	**3.6**	6.1	**1.9**	7.7	**1.5**	5.3	**2.2**	9.1	**2.4**	×	**3.5**	×
soc-pokec	**5.4**	6.2	**5.1**	6.3	6.1	**5.6**	**4.0**	4.9	**4.5**	4.9	**3.4**	5.8
socfb-B-anon	**7.4**	9.6	**7.1**	8.6	7.4	**6.7**	**7.8**	8.0	**8.2**	×	**6.7**	×
soc-orkut	**96.1**	167.5	**79.1**	809.8	**611.5**	×	×	×	**86.9**	×	**68.8**	×
socfb-A-anon	7.8	**7.4**	6.5	**6.5**	**6.2**	8.1	7.0	**6.0**	6.9	**5.5**	**5.3**	6.2
wikipedia-link-en	**12.1**	12.6	**12.0**	16.1	**12.9**	468.6	**15.3**	×	**19.1**	×	**247.1**	×
dbpedia-link	76.0	**74.4**	**52.2**	68.6	**50.5**	50.8	49.3	**44.4**	**696.0**	×	×	×
wikipedia-link-en13	249.0	**244.9**	290.7	**229.0**	249.3	**217.6**	290.9	**262.6**	245.3	**225.5**	304.4	**284.5**
delicious-ui	62.1	**58.9**	**99.0**	168.2	**101.2**	171.8	**119.3**	434.5	×	×	×	×
soc-sinaweibo	**141.2**	×	**126.3**	×	**207.3**	×	**1661.7**	×	×	×	×	×
web-ClueWeb09	**43.8**	995.6	**70.5**	×	**613.2**	×	×	×	×	×	×	×

Dataset	$k = 8$		$k = 9$		$k = 10$		$k = 15$		$k = 20$	
	w/ UBR2	w/o UBR2	w/ UBR2	w/o UBR2	w/ UBR2	w/o UBR2	w/ UBR2	w/o UBR2	w/ UBR2	w/o UBR2
hamming6-2	**0.0**	0.0	**0.0**	0.0	0.0	**0.0**	0.0	**0.0**	0.0	**0.0**
johnson8-4-4	×	×	**81.3**	×	**1271.9**	×	**20.4**	1409.3	0.0	**0.0**
keller4	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	**0.6**	×	**0.7**	×	**0.6**	×	**2.7**	×	**1.7**	×
ia-wiki-Talk	**0.9**	×	**1.4**	×	**1.5**	×	**3.6**	×	**23.7**	×
soc-buzznet	**75.4**	×	**106.7**	×	**150.3**	×	**520.7**	×	**701.7**	×
soc-LiveMocha	**4.1**	×	**2.3**	×	**2.8**	×	**0.9**	×	**3.2**	×
soc-gowalla	0.1	**0.1**	**0.1**	1.2	**0.1**	908.4	**0.2**	×	**0.4**	×
sc-msdoor	**2.3**	2.5	3.6	**3.4**	3.9	**3.7**	3.4	**3.0**	**5.3**	×
soc-youtube	**0.3**	×	**0.3**	×	**0.3**	×	**0.9**	×	**2.8**	×
soc-digg	**49.1**	×	**46.7**	×	**35.1**	×	**7.9**	×	**7.4**	×
sc-ldoor	**7.1**	7.3	8.0	**7.7**	8.6	**8.4**	7.3	**6.7**	**10.1**	×
soc-youtube-snap	**0.4**	×	**0.6**	×	**0.5**	×	**1.2**	×	**4.6**	×
soc-lastfm	**1.1**	×	**1.4**	×	**1.6**	×	**5.6**	×	**21.3**	×
soc-pokec	**3.2**	3.9	**8.1**	1081.7	**3.1**	8.6	**3.3**	399.2	**3.4**	×
socfb-B-anon	**5.6**	×	**6.7**	×	**7.0**	×	**6.3**	×	**6.7**	×
soc-orkut	**49.9**	×	**49.7**	×	**46.1**	×	**33.6**	×	37.2	**36.4**
socfb-A-anon	5.7	**5.2**	**4.8**	5.2	**4.9**	24.4	**4.8**	×	**3.7**	×
wikipedia-link-en	**69.0**	×	**29.6**	×	**12.0**	×	**9.9**	12.0	**7.2**	7.6
dbpedia-link	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	284.1	**247.2**	255.5	**241.3**	292.3	**284.5**	297.2	**275.6**	277.6	**231.7**
delicious-ui	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×

Table 21 shows the full results of comparison between our default U-MkP and competitive baselines including kPlexT [35], Maple [53] and DiseMKP [44]. We can see that U-MkP is a clear winner.

Table 21: Execution Times of U-MkP

Dataset	$k=2$				$k=3$				$k=4$				$k=5$			
	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP
hamming6-2	**0.2**	0.8	0.4	6.0	**35.4**	208.8	189.0	261.5	**36.9**	472.5	359.1	134.3	**3.6**	53.2	50.6	34.7
johnson8-4-4	**0.7**	1.7	1.7	1.1	**5.7**	24.0	34.4	6.6	**29.6**	210.9	926.3	239.4	**2.0**	123.0	×	76.1
keller4	25.2	134.6	111.6	**19.7**	**28.9**	2073.1	1251.1	64.0	**1504.8**	×	×	×	**1120.0**	×	×	×
brock200-2	**4.2**	8.1	9.7	5.7	21.4	68.8	278.9	**10.2**	**71.6**	353.1	2388.2	138.8	**715.0**	×	×	×
p-hat500-1	14.4	29.5	35.4	**9.2**	29.6	163.1	×	**9.4**	70.2	×	×	**69.6**	**242.0**	×	×	×
socfb-Duke14	**1.2**	6.4	2.3	129.2	**0.8**	14.4	43.7	×	**1.4**	9.2	30.9	×	**2.6**	21.3	4.8	×
ia-wiki-Talk	0.8	4.1	4.1	**0.7**	**0.6**	2.7	10.1	4.1	**0.4**	0.8	1.3	6.9	**0.7**	0.9	4.1	70.7
soc-buzznet	**358.0**	1577.7	1589.5	×	**189.1**	2744.2	×	×	**153.0**	×	×	×	**1771.0**	×	×	×
soc-LiveMocha	**3.3**	24.6	8.0	8.2	**2.2**	7.8	178.6	19.2	**2.3**	10.5	120.2	120.2	**7.0**	28.1	62.3	×
soc-gowalla	**0.1**	0.1	0.2	1.1	**0.1**	0.1	0.2	0.8	0.1	**0.1**	0.1	2.3	**0.1**	0.3	0.2	8.9
sc-msdoor	7.3	**4.7**	6.5	×	12.4	**7.1**	21.5	×	**25.0**	85.9	120.2	×	**40.8**	160.0	349.1	×
soc-youtube	**0.2**	0.8	0.4	1.1	0.3	**0.3**	0.4	1.3	**0.3**	0.3	1.3	2.2	**0.3**	0.6	0.4	12.9
soc-digg	**121.9**	392.2	331.1	×	**49.4**	×	×	×	**33.2**	252.0	×	×	**102.4**	503.7	×	×
sc-ldoor	14.1	**9.9**	11.7	×	24.6	**14.9**	40.8	×	**56.6**	159.9	221.9	×	**71.3**	298.9	572.2	×
soc-youtube-snap	**0.4**	1.3	0.6	1.6	**0.5**	1.4	0.7	2.2	**0.3**	0.4	1.6	2.8	**0.3**	1.0	0.5	10.6
soc-lastfm	**3.6**	8.1	7.9	4.9	**1.9**	6.2	63.9	17.8	**1.5**	6.4	28.8	24.8	**2.2**	7.5	34.1	11.7
soc-pokec	**5.4**	5.5	18.8	20.1	5.1	**3.8**	33.7	20.1	6.1	**3.6**	15.7	43.7	4.0	**3.5**	29.0	17.1
socfb-B-anon	**7.4**	9.3	21.6	35.8	**7.1**	24.1	45.4	33.3	**7.4**	30.8	23.5	45.5	**7.8**	23.6	56.4	289.0
soc-orkut	**96.1**	505.2	220.0	×	**79.1**	706.9	603.0	×	**611.5**	×	1047.9	×	×	×	**1473.2**	×
socfb-A-anon	7.8	**5.0**	19.8	28.0	6.5	**4.4**	41.7	29.8	6.2	**3.9**	39.4	31.9	7.0	**4.5**	14.9	53.3
wikipedia-link-en	12.1	**10.7**	65.9	×	12.0	**9.9**	560.5	×	**12.9**	17.6	1485.6	×	**15.3**	44.9	×	1520.1
dbpedia-link	**76.0**	198.4	332.2	×	**52.2**	700.5	×	×	**50.5**	1371.4	×	×	**49.3**	×	×	×
wikipedia-link-en13	**249.0**	321.0	×	×	**290.7**	364.6	×	×	**249.3**	470.5	×	×	**290.9**	552.6	×	×
delicious-ui	**62.1**	109.3	189.6	326.9	**99.0**	441.9	215.1	395.3	**101.2**	1368.2	215.2	394.3	**119.3**	×	×	1452.4
soc-sinaweibo	**141.2**	674.8	512.0	×	**126.3**	×	×	×	**207.3**	×	×	×	**1661.7**	×	×	×
web-ClueWeb09	**43.8**	355.3	224.6	×	**70.5**	×	×	×	**613.2**	×	×	×	×	×	×	×

Dataset	$k=6$				$k=7$				$k=8$				$k=9$			
	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP
hamming6-2	**16.9**	216.4	30.3	79.0	0.0	0.0	0.0	**0.0**	0.0	0.0	0.0	**0.0**	0.0	0.0	0.0	**0.0**
johnson8-4-4	**332.4**	×	×	×	×	×	×	×	×	×	×	×	**81.3**	1599.8	×	1996.2
keller4	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	**0.6**	304.6	9.5	×	**0.6**	307.8	6.1	×	**0.6**	×	6.3	×	**0.7**	×	2.3	×
ia-wiki-Talk	**0.4**	2.1	1.8	92.3	**0.5**	4.7	0.6	×	**0.9**	426.3	1.3	×	**1.4**	355.3	1.5	×
soc-buzznet	**123.7**	×	×	×	**76.1**	×	×	×	**75.4**	×	×	×	**106.7**	×	×	×
soc-LiveMocha	**3.1**	139.7	195.1	×	**4.1**	495.9	365.5	×	**4.1**	×	103.8	×	**2.3**	2732.0	19.2	×
soc-gowalla	**0.1**	0.9	0.1	35.8	**0.1**	0.4	0.1	15.1	**0.1**	23.7	0.1	×	**0.1**	21.2	0.4	×
sc-msdoor	**28.8**	123.2	36.3	×	**6.1**	6.1	10.1	×	2.3	2.5	**2.2**	5.7	3.6	**1.6**	7.0	×
soc-youtube	**0.2**	0.2	0.3	9.5	**0.2**	1.1	0.4	23.4	**0.3**	4.1	1.2	89.3	**0.3**	10.0	0.4	192.5
soc-digg	**35.4**	837.6	1002.9	×	**54.5**	1529.0	1683.5	×	**49.1**	1827.6	1280.4	×	**46.7**	×	1353.1	×
sc-ldoor	**50.4**	70.6	66.1	×	**11.3**	29.8	16.3	×	7.1	6.3	**5.3**	42.3	**8.0**	14.0	15.3	×
soc-youtube-snap	**0.3**	0.5	0.5	34.5	**0.3**	1.5	0.5	186.4	**0.4**	4.3	1.5	522.0	**0.6**	22.9	0.6	×
soc-lastfm	**2.4**	85.1	7.1	72.3	**3.5**	168.8	7.0	534.1	**1.1**	131.8	4.2	156.7	1.4	548.1	**1.4**	×
soc-pokec	4.5	**3.4**	28.5	17.3	**3.4**	3.6	10.6	17.0	**3.2**	3.3	13.2	16.3	8.1	**3.1**	13.2	13.3
socfb-B-anon	**8.2**	18.3	19.0	262.6	**6.7**	10.3	16.8	146.4	**5.6**	7.7	41.0	388.7	6.7	**6.3**	36.6	×
soc-orkut	**86.9**	×	×	×	**68.8**	×	1200.9	×	**49.9**	1054.4	312.7	×	**49.7**	530.0	345.0	×
socfb-A-anon	6.9	**3.6**	17.1	241.5	5.3	**4.2**	13.6	277.9	5.7	**4.4**	16.1	803.7	**4.8**	9.8	16.1	×
wikipedia-link-en	**19.1**	164.7	×	×	**247.1**	580.8	×	×	**69.0**	236.1	×	×	**29.6**	100.9	×	30.2
dbpedia-link	**696.0**	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	**245.3**	677.1	×	×	**304.4**	652.2	×	×	**284.1**	806.9	×	×	**255.5**	×	×	×
delicious-ui	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

Dataset	$k=10$				$k=15$				$k=20$			
	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP	U-MkP	kPlexT	Maple	DiseMKP
hamming6-2	0.0	0.0	0.0	**0.0**	0.0	0.0	**0.0**	0.0	0.0	0.0	0.0	**0.0**
johnson8-4-4	**1271.9**	×	×	×	**20.4**	36.3	159.5	401.2	0.0	0.0	0.0	**0.0**
keller4	×	×	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	**0.6**	×	9.0	×	**2.7**	×	18.3	×	**1.7**	×	25.2	×
ia-wiki-Talk	**1.5**	×	2.5	×	**3.6**	×	25.5	×	**23.7**	×	×	×
soc-buzznet	**150.3**	×	×	×	**520.7**	×	×	×	**701.7**	×	×	×
soc-LiveMocha	**2.8**	×	24.9	×	**0.9**	×	1.9	×	**3.2**	×	335.7	×
soc-gowalla	**0.1**	66.5	0.1	×	**0.2**	×	0.6	×	**0.4**	×	4.5	125.0
sc-msdoor	**3.9**	4.9	8.1	×	3.4	**1.4**	2.0	×	5.3	**4.5**	8.1	×
soc-youtube	**0.3**	26.2	0.4	1026.4	**0.9**	×	2.8	×	**2.8**	×	372.2	×
soc-digg	**35.1**	×	608.2	×	**7.9**	×	28.1	×	**7.4**	×	26.5	×
sc-ldoor	8.6	**5.7**	5.7	×	7.3	**3.0**	4.3	×	10.1	**5.2**	6.5	×
soc-youtube-snap	**0.5**	143.8	0.5	×	**1.2**	×	3.6	×	**4.6**	×	1347.6	×
soc-lastfm	1.6	2014.8	**1.3**	2958.2	**5.6**	×	6.7	×	**21.3**	×	824.1	×
soc-pokec	**3.1**	4.1	11.2	16.3	**3.3**	97.9	21.1	×	**3.4**	×	18.8	×
socfb-B-anon	**7.0**	7.3	34.4	497.6	**6.3**	764.8	35.0	×	**6.7**	×	12.4	×
soc-orkut	**46.1**	659.4	195.5	×	**33.6**	519.9	142.5	×	**37.2**	×	98.3	×
socfb-A-anon	4.9	**4.1**	14.5	×	**4.8**	×	13.6	×	**3.7**	×	12.7	×
wikipedia-link-en	**12.0**	179.5	1639.1	39.1	9.9	**8.4**	18.7	101.7	**7.2**	8.9	15.3	14.4
dbpedia-link	×	×	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	**292.3**	×	×	×	**297.2**	×	×	×	**277.6**	×	×	×
delicious-ui	×	×	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×

Section 6 presents a two-phase approach to compute all MkPs as well as the densest MkP. Table 22 reports the results of running this variant for all values of k, including (1) the running time, (2) the number of MkPs found, (3) the number of vertices and the number of edges in the MkP found by Phase 1, and (4) the number of edges of the densest MkP found in Phase 2.

Table 22: Results of All-MkP and Densest-MkP

Dataset	k = 2				k = 3				k = 4				k = 5			
	Time	#MkPs	$\lvert E_{\text{Phase 1}}\rvert$	$\lvert E_{\text{Densest}}\rvert$	Time	#MkPs	$\lvert E_{\text{Phase 1}}\rvert$	$\lvert E_{\text{Densest}}\rvert$	Time	#MkPs	$\lvert E_{\text{Phase 1}}\rvert$	$\lvert E_{\text{Densest}}\rvert$	Time	#MkPs	$\lvert E_{\text{Phase 1}}\rvert$	$\lvert E_{\text{Densest}}\rvert$
hamming6-2	0	8	992	992	44	5077	992	992	105	353	1440	1440	15	105	2064	2064
johnson8-4-4	0	1851	182	182	19	15,168	274	274	198	49,214	398	398	9	226	644	644
keller4	41	72,066	203	203	62	45	378	378	×	×	×	×	×	×	×	×
brock200-2	6	57	146	150	22	1	214	214	655	26	260	266	×	×	×	×
p-hat500-1	14	2	124	124	30	2	158	158	106	11	196	204	601	10	246	248
socfb-Duke14	2	133	1396	1397	1	3	1772	1772	4	39	1926	1932	12	75	2166	2172
ia-wiki-Talk	1	6	298	298	1	2	390	390	1	5	454	454	3	95	530	534
soc-buzznet	417	624	1238	1246	249	247	1590	1602	301	3	1990	1990	×	×	×	×
soc-LiveMocha	5	15	328	330	4	65	432	436	6	20	544	552	15	24	664	676
soc-gowalla	0	4	866	868	0	5	922	924	0	1	980	980	0	40	972	980
sc-msdoor	1	1	420	420	1	2	420	420	1	1	420	420	56	17,799,765	442	540
soc-youtube	0	5	372	372	1	77	400	408	0	2	508	508	0	3	582	588
soc-digg	139	1	3172	3172	64	246	3852	3856	38	1	4602	4602	90	15	4976	4986
sc-ldoor	2	1	420	420	2	1	420	420	2	1	420	420	×	×	×	×
soc-youtube-snap	1	8	372	372	1	147	396	408	1	8	512	512	1	8	584	590
soc-lastfm	6	349	296	298	3	1103	398	398	3	207	502	510	3	2	622	622
soc-pokec	7	2	926	926	8	1	984	984	9	11	972	984	9	9	1056	1058
socfb-B-anon	17	3428	696	696	17	10,475	852	852	15	5416	1020	1020	16	24,378	1132	1140
soc-orkut	168	925	2634	2634	143	8	3372	3374	655	2	3784	3784	×	×	×	×
socfb-A-anon	18	333	742	742	16	692	962	964	16	82	1142	1142	15	2774	1264	1268
wikipedia-link-en	15	2509	1,206,698	1,206,698	15	13,454	1,211,088	1,211,088	19	34,403	1,215,482	1,215,482	31	24,837	1,219,880	1,219,880
dbpedia-link	116	325	1800	1802	104	140	2124	2134	95	40	2490	2490	91	2	2776	2776
wikipedia-link-en13	68	2	183,610	183,610	67	7	184,462	184,462	68	5	185,316	185,316	67	1	186170	186,170
delicious-ui	183	2	64	66	559	522	76	78	783	105	100	106	882	23	110	132
soc-sinaweibo	202	1744	2636	2636	177	28	3252	3256	265	563	3704	3714	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

Dataset	k = 6				k = 7				k = 8				k = 9			
	t (sec)	#MkPs	$\lvert E\rvert_{\text{init}}$	$\lvert E\rvert_{\text{dense}}$	t (sec)	#MkPs	$\lvert E\rvert_{\text{init}}$	$\lvert E\rvert_{\text{dense}}$	t (sec)	#MkPs	$\lvert E\rvert_{\text{init}}$	$\lvert E\rvert_{\text{dense}}$	t (sec)	#MkPs	$\lvert E\rvert_{\text{init}}$	$\lvert E\rvert_{\text{dense}}$
hamming6-2	93	3811	2404	2412	×	×	×	×	×	×	×	×	×	×	×	×
johnson8-4-4	×	×	×	×	×	×	×	×	×	×	×	×	510	229	1386	1386
keller4	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	1	20	2430	2432	1	3	2702	2708	1	7	2880	2890	1	15	3064	3084
ia-wiki-Talk	1	3	652	652	1	2	736	736	3	27	774	786	4	12	848	864
soc-buzznet	131	6	2688	2698	126	61	2992	2992	124	19	3274	3292	126	2	3596	3596
soc-LiveMocha	10	478	760	770	8	46	896	904	5	3	1052	1062	8	404	1164	1178
soc-gowalla	0	118	1022	1030	0	8	1196	1196	0	21	1312	1320	0	2	1446	1446
sc-msdoor	×	×	×	×	408	27,954,741	786	786	3	10,579	994	994	4	10,579	994	994
soc-youtube	0	14	670	670	0	2	754	754	1	609	778	804	1	154	876	894
soc-digg	67	26	5378	5400	121	918	5778	5804	52	5	6362	6372	252	11,618	6662	6680
sc-ldoor	×	×	×	×	555	39,455,805	786	786	7	14,025	994	994	7	14,025	994	994
soc-youtube-snap	1	31	670	672	1	15	760	760	1	2	844	844	1	608	882	898
soc-lastfm	8	9	702	718	19	1443	800	810	7	4	942	942	10	63	1056	1058
soc-pokec	7	38	1164	1164	6	3	1342	1342	4	10	1470	1474	2	16	1608	1610
socfb-B-anon	17	449	1318	1318	22	15,830	1450	1450	11	173	1626	1626	14	846	1770	1772
soc-orkut	256	659	4892	4898	110	834	5554	5568	83	482	6272	6282	73	56	6878	6884
socfb-A-anon	14	8	1448	1448	12	3788	1512	1526	14	44	1728	1732	13	3	1866	1874
wikipedia-link-en	81	33,082	1,224,282	1,224,282	776	7,844,681	1,228,678	1,228,688	250	2,315,395	1,235,266	1,235,296	32	24,444	1,241,918	1,241,918
dbpedia-link	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	68	1	186170	186,170	68	1	186170	186,170	68	1	186170	186,170	69	1	186170	186,170
delicious-ui	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

Dataset	k = 10				k = 15				k = 20			
	t (sec)	#MkPs	$\lvert E\rvert_{\text{init}}$	$\lvert E\rvert_{\text{dense}}$	t (sec)	#MkPs	$\lvert E\rvert_{\text{init}}$	$\lvert E\rvert_{\text{dense}}$	t (sec)	#MkPs	$\lvert E\rvert_{\text{init}}$	$\lvert E\rvert_{\text{dense}}$
hamming6-2	×	×	×	×	×	×	×	×	×	×	×	×
johnson8-4-4	×	×	×	×	379	415,293	2736	2740	×	×	×	×
keller4	×	×	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	2	4	3282	3282	10	71	4230	4246	1	8	5464	5468
ia-wiki-Talk	5	27	944	950	11	2	1422	1422	225	5950	1826	1826
soc-buzznet	255	1371	3804	3820	321	2	5320	5320	691	7	6738	6740
soc-LiveMocha	4	9	1342	1350	3	2	2060	2060	4	18	2664	2664
soc-gowalla	0	4	1488	1490	0	747	1942	1958	1	8	2428	2442
sc-msdoor	4	10,579	994	994	3	4655	1232	1232	1,170	93,731,434	1258	1356
soc-youtube	1	32	990	992	2	2	1382	1382	18	102	1768	1790
soc-digg	228	652	7128	7128	9	279	9194	9212	11	8	10,776	10,776
sc-ldoor	7	14,025	994	994	6	4496	1232	1232	1,192	110,544,857	1206	1356
soc-youtube-snap	1	211	984	996	5	121	1398	1406	6	6	1794	1808
soc-lastfm	13	18	1164	1164	82	1948	1652	1682	28	27	2230	2270
soc-pokec	2	3	1746	1746	2	362	2022	2026	2	837	2316	2322
socfb-B-anon	13	1191	1886	1918	10	552	2674	2696	13	4166	3260	3300
soc-orkut	65	2	7332	7332	63	228	9318	9318	65	5	11,072	11,072
socfb-A-anon	12	515	1946	1946	13	1066	2426	2468	9	13	3018	3022
wikipedia-link-en	317	57,370	1,246,352	1,246,352	11	536	1,272,962	1,272,990	11	15	1,284,134	1,284,134
dbpedia-link	×	×	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	68	1	186170	186,170	68	1	186170	186,170	70	1	186170	186,170
delicious-ui	×	×	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×

Table 23 reports the running time of our parallel U-MkP when using 1, 2, 4, 8, 16 and 32 threads, respectively, where we use the default timeout threshold $\tau_{time} = 0.1$ ms for load balancing (which consistently works well). We can see that our parallel U-MkP generally achieves a near-ideal speedup ratio.

Table 23: Execution Times of Our Parallel Algorithm

Dataset	k = 2						k = 3						k = 4						k = 5					
	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32
hamming6-2	0.8	0.4	0.2	0.1	0.1	0.1	96.5	47.3	23.8	13.5	7.1	5.7	40.7	20.7	10.9	6.0	2.8	2.1	4.9	2.4	1.5	0.7	0.5	0.4
johnson8-4-4	0.8	0.4	0.2	0.2	0.1	0.1	9.8	4.9	2.4	1.2	0.8	0.5	81.2	42.4	20.9	11.2	5.9	4.4	3.1	1.5	0.7	0.4	0.4	0.3
keller4	27.6	13.9	6.7	3.4	1.7	0.9	247.0	120.3	60.7	30.7	16.0	6.3	1480.2	1028.2	728.3	564.6	279.3	151.6	1120.2	620.4	287.5	120.3	85.3	43.2
brock200-2	4.3	2.3	1.1	0.6	0.3	0.2	24.3	11.5	5.7	2.9	1.5	0.8	102.0	49.8	25.3	13.8	7.6	3.5	×	×	333.7	156.6	81.4	53.6
p-hat500-1	14.0	7.0	3.5	1.7	0.9	0.5	32.5	15.8	7.9	4.1	2.0	1.1	92.6	46.1	22.3	11.6	5.6	2.8	417.1	213.4	109.2	54.0	27.7	15.0
socfb-Duke14	0.6	0.3	0.2	0.1	0.0	0.0	0.3	0.1	0.1	0.0	0.0	0.0	0.6	0.3	0.2	0.1	0.0	0.1	2.8	1.1	0.6	0.3	0.1	0.2
ia-wiki-Talk	0.5	0.3	0.1	0.1	0.1	0.1	0.4	0.2	0.1	0.0	0.0	0.0	0.3	0.2	0.1	0.0	0.1	0.0	0.7	0.3	0.2	0.1	0.1	0.1
soc-buzznet	301.6	150.5	74.9	36.7	19.1	9.4	145.7	73.3	36.5	18.0	9.2	4.7	143.4	71.1	35.0	17.9	9.0	4.8	1720.7	824.2	421.2	231.4	134.2	81.2
soc-LiveMocha	2.4	1.2	0.6	0.3	0.2	0.1	1.5	0.7	0.4	0.2	0.1	0.1	3.7	1.7	0.8	0.4	0.2	0.2	18.8	10.1	5.5	3.0	1.5	1.5
soc-gowalla	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
sc-msdoor	2.9	1.5	0.8	0.4	0.3	0.3	7.6	3.7	2.0	1.1	0.7	0.7	88.1	48.0	24.4	13.6	9.1	6.3	289.9	140.2	79.3	44.2	28.3	10.3
soc-youtube	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	×	0.1	0.0	0.0	0.0	0.0	0.0
soc-digg	91.6	45.9	22.2	11.5	5.7	2.9	29.5	15.2	7.6	3.7	1.9	1.0	42.5	16.0	6.6	3.0	1.3	0.9	100.2	54.3	27.3	14.9	8.2	4.3
sc-ldoor	5.9	3.0	1.6	0.8	0.5	0.7	24.2	12.3	5.8	3.7	2.8	3.2	48.2	24.3	15.2	8.9	4.3	3.2	72.8	38.2	20.9	10.3	6.7	3.2
soc-youtube-snap	0.1	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.2	0.1	0.0	0.0	0.0	0.0
soc-lastfm	2.5	1.2	0.6	0.3	0.2	0.1	1.1	0.6	0.3	0.1	0.1	0.1	0.8	0.4	0.2	0.1	0.1	0.1	2.3	1.1	0.6	0.3	0.2	0.1
soc-pokec	7.1	5.6	5.9	4.7	4.3	4.3	7.6	5.6	3.5	4.7	4.3	4.3	7.2	5.6	5.9	4.7	4.3	4.3	4.1	3.2	3.9	2.5	2.4	2.4
socfb-B-anon	10.2	5.6	5.9	2.3	1.2	1.8	7.5	6.3	5.4	5.3	5.2	5.2	8.0	7.2	3.4	3.1	3.2	2.9	7.5	6.7	6.2	5.4	5.8	3.4
soc-orkut	106.3	52.3	30.2	20.4	10.8	2.4	80.6	39.2	25.8	20.3	10.2	8.3	663.3	463.0	243.3	156.4	112.2	68.7	×	×	×	×	×	×
socfb-A-anon	9.9	8.1	6.7	7.4	2.3	3.4	10.6	8.1	8.6	8.9	8.2	1.9	8.3	7.9	8.9	3.4	5.2	2.3	9.1	8.2	4.4	5.4	3.3	3.3
wikipedia-link-en	0.7	0.4	0.2	0.1	0.1	0.1	0.7	0.4	0.2	0.1	0.1	0.1	1.0	0.5	0.3	0.2	0.1	0.1	1.5	0.8	0.4	0.2	0.1	0.1
dbpedia-link	10.0	5.2	2.8	1.4	0.7	0.4	6.7	3.4	1.9	1.0	0.5	0.3	5.8	3.0	1.7	0.9	0.5	0.3	5.6	2.8	1.5	0.8	0.4	0.3
wikipedia-link-en13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
delicious-ui	18.8	9.9	5.3	2.7	1.4	0.6	95.7	48.4	25.4	13.3	6.8	2.8	345.2	178.7	94.8	49.5	24.9	10.2	1282.2	834.2	612.1	406.9	205.2	88.4
soc-sinaweibo	32.0	18.5	10.3	5.3	3.6	2.6	30.2	18.1	10.1	5.2	3.7	2.6	60.1	34.1	19.0	8.7	6.2	5.1	1812.2	918.3	345.2	121.8	74.3	45.7
web-ClueWeb09	9.2	5.9	3.2	1.7	1.2	1.0	16.2	8.8	4.8	2.4	1.9	1.6	57.2	29.4	15.8	7.8	5.7	4.8	52.2	28.9	17.4	8.4	4.9	3.7

Dataset	k = 6						k = 7						k = 8						k = 9					
	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32
hamming6-2	17.1	10.1	5.9	2.5	×	2.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
johnson8-4-4	335.4	193.4	91.7	49.8	26.3	17.8	×	×	×	×	×	×	×	×	×	×	×	×	85.6	26.2	3.5	2.6	3.8	2.1
keller4	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	540.7	277.9	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	410.0	207.0	×	×	×	×	×	531.3	×	×	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	0.7	0.1	0.1	0.0	0.0	0.0	0.7	0.1	0.0	0.0	0.0	0.0	0.7	0.1	0.1	0.0	0.0	0.0	0.7	0.1	0.1	0.0	0.0	0.0
ia-wiki-Talk	0.4	0.2	0.1	0.0	0.1	0.1	0.5	0.2	0.1	0.1	0.0	0.1	1.0	0.5	0.2	0.1	0.1	0.1	1.4	0.7	0.4	0.2	0.1	0.1
soc-buzznet	129.8	60.3	29.4	14.7	7.5	3.8	85.8	38.3	18.6	9.5	4.8	2.4	78.3	35.3	17.2	8.8	4.4	2.2	112.0	52.6	25.9	13.1	6.5	3.4
soc-LiveMocha	3.1	2.1	1.0	0.6	0.2	0.1	4.3	2.0	1.0	0.5	0.3	0.2	4.5	2.5	1.2	0.6	0.4	0.2	2.5	1.5	0.8	0.4	0.2	0.2
soc-gowalla	0.1	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0
sc-msdoor	30.6	63.0	29.5	16.6	9.8	7.6	6.2	2.0	1.0	0.5	0.3	0.3	2.6	0.0	0.0	0.0	0.0	0.0	3.6	0.2	0.1	0.1	0.0	0.1
soc-youtube	0.3	0.0	0.0	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.0	0.0	0.3	0.1	0.0	0.0	0.0	0.0	0.3	0.1	0.1	0.0	0.0	0.0
soc-digg	37.2	15.8	8.1	4.1	2.0	1.1	57.0	18.7	9.8	4.9	2.5	1.2	52.6	24.7	14.7	6.2	3.0	0.9	48.4	30.8	12.8	6.2	3.1	1.8
sc-ldoor	47.1	28.2	14.8	8.2	4.2	2.8	11.7	7.1	3.6	2.3	1.5	1.5	7.4	0.1	0.1	0.0	0.1	0.1	8.3	4.0	3.5	2.3	2.2	2.0
soc-youtube-snap	0.4	0.0	0.0	0.0	0.0	0.0	0.3	0.1	0.0	0.0	0.0	0.0	0.5	0.1	0.1	0.0	0.0	0.0	0.5	0.2	0.1	0.0	0.0	0.0
soc-lastfm	2.5	1.2	0.6	0.3	0.2	0.1	3.5	1.7	0.9	0.4	0.2	0.1	1.2	0.5	0.3	0.1	0.1	0.1	1.4	0.7	0.3	0.2	0.1	0.1
soc-pokec	3.7	2.0	1.0	0.0	0.0	0.0	4.9	2.0	1.0	0.0	0.0	0.0	4.8	4.5	3.9	1.2	0.0	0.0	4.8	3.4	2.1	2.8	1.8	1.3
socfb-B-anon	8.8	4.0	6.0	2.0	1.0	0.0	9.6	4.0	0.1	0.1	0.0	0.0	8.8	4.0	3.4	3.4	2.9	0.0	5.9	4.0	2.3	4.9	2.9	1.0
soc-orkut	96.5	16.0	7.7	4.0	2.1	1.1	69.5	49.0	18.3	8.3	4.5	4.5	52.4	34.0	20.0	10.4	4.3	1.5	44.3	28.0	19.8	12.3	8.0	6.0
socfb-A-anon	6.7	4.1	3.2	3.9	2.5	1.2	6.1	3.9	4.2	3.8	3.8	1.5	5.6	3.2	3.8	2.3	1.9	1.0	5.4	2.3	2.4	2.4	2.4	2.1
wikipedia-link-en	9.8	5.1	2.7	1.6	0.9	0.4	367.5	187.7	103.3	51.2	14.7	1.4	81.6	41.4	24.2	12.3	4.7	0.3	29.9	14.8	7.4	2.7	0.3	0.1
dbpedia-link	589.2	371.7	203.1	102.3	51.3	21.4	×	×	×	×	×	294.5	×	×	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
delicious-ui	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	468.9	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×

Dataset	k = 10						k = 15						k = 20					
	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32
hamming6-2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
johnson8-4-4	1395.0	602.8	306.6	171.8	65.2	24.3	25.9	11.2	5.7	2.8	1.6	0.9	0.0	0.0	0.0	0.0	0.0	0.0
keller4	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
brock200-2	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
p-hat500-1	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
socfb-Duke14	2.8	0.2	0.1	0.0	0.0	0.0	1.8	1.6	0.9	0.4	0.3	0.1	1.9	0.9	0.4	0.2	0.1	0.1
ia-wiki-Talk	3.7	1.4	0.7	0.3	0.2	0.1	24.6	2.1	1.0	0.5	0.2	0.2	10.2	12.7	6.1	3.1	1.5	0.8
soc-buzznet	158.0	82.3	43.7	22.2	10.8	5.2	543.0	299.6	148.6	72.7	36.8	18.1	754.2	465.0	221.9	103.9	51.0	22.8
soc-LiveMocha	0.9	1.6	0.8	0.4	0.2	0.2	3.2	1.4	0.7	0.3	0.2	0.1	3.2	2.1	2.2	2.5	1.8	1.6
soc-gowalla	0.2	0.0	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0	14.5	12.7	6.1	3.1	2.4	2.3
sc-msdoor	3.4	0.4	0.2	0.1	0.1	0.1	5.5	0.1	0.1	0.0	0.1	0.1	3.6	20.0	11.7	6.2	5.3	4.3
soc-youtube	0.9	0.1	0.0	0.0	0.0	0.0	3.1	0.6	0.3	0.2	0.1	0.1	6.5	2.7	1.3	0.6	0.3	0.2
soc-digg	9.2	20.3	9.4	4.5	2.5	1.3	8.9	4.7	2.2	1.1	0.6	0.3	2.1	6.1	3.0	1.5	0.8	0.4
sc-ldoor	7.1	5.3	0.4	0.2	0.2	0.2	10.4	4.5	2.3	1.9	1.5	0.1	2.4	1.3	0.7	0.4	0.2	0.2
soc-youtube-snap	1.2	0.2	0.1	0.0	0.0	0.0	4.3	4.0	4.2	2.3	0.2	0.1	20.3	19.2	16.0	10.3	5.3	3.0
soc-lastfm	6.0	4.6	2.3	2.1	1.8	0.1	22.1	3.1	1.5	0.8	0.4	0.2	6.0	5.5	5.8	2.9	1.5	0.7
soc-pokec	2.8	2.8	2.8	2.8	1.8	0.0	4.0	3.5	3.8	2.4	2.9	1.2	48.3	24.0	12.0	8.3	4.5	1.2
socfb-B-anon	6.0	4.2	2.3	2.5	2.1	1.0	6.7	5.3	2.3	1.4	1.8	1.9	4.4	2.2	2.4	2.3	2.4	1.9
soc-orkut	49.1	45.2	24.2	8.9	4.3	2.0	34.0	29.2	25.2	18.2	10.3	5.0	19.4	14.2	10.2	10.2	5.3	2.5
socfb-A-anon	4.7	3.4	2.1	2.1	0.1	0.1	4.9	4.3	4.9	3.4	3.4	2.0	25.8	20.8	21.3	10.3	5.4	2.0
wikipedia-link-en	4.0	2.1	1.1	0.6	0.3	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
dbpedia-link	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
wikipedia-link-en13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
delicious-ui	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
soc-sinaweibo	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
web-ClueWeb09	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×