

Maximum k -Plex Finding: Choices of Pruning Techniques Matter! [Experiment, Analysis & Benchmark]

Akhlaque Ahmad, Da Yan, Xiao Chen, Lyuheng Yuan, Qin Zhang, Saugat Adhikari

Indiana University Bloomington

{akahmad,yanda,xc56,lyyuan,qzhangcs,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, Seasaw, 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 for different values of k and different datasets. For example, among the two competitive branching methods, the partitioning-based one by DiseMKP performs the best (up to $10\times$ faster) when k is small, but for larger values of k , an alternative pivot-based branching method by Maple can find MkP up to thousands of times faster, so sticking to one strategy for all scenarios can be catastrophic. Through extensive experiments, we obtain important new insights on how to configure the set of pruning rules to avoid performance pitfalls. We also study problem variants such as finding all the MkPs and finding the densest MkP (i.e., with the most edges), and effective algorithm parallelization. Our source code is released 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 [?] and social communities [?]. One classic notion of cohesive subgraph is *clique* which requires every pair of distinct vertices to be connected by an edge (see Figure ??(a)). However, in real graphs, communities rarely appear in the form of cliques due to various reasons such as the existence of data noise [?]. As a relaxed clique model, k -plex was first introduced in [?], which is a graph where every vertex v is adjacent to all but at most k vertices (including v itself). Figure ??(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 ??(c)). It has found extensive applications in the analysis of social networks [?

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

], especially in the community detection [?]. However, mining k -plex structures is NP-hard [?], 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 later). 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 improve the performance on some graphs and k value, but can lead to catastrophic performance in other cases (not tested in the experiments of the respective papers). Even worse, there is even a work [?] whose implementation is totally different from what was proposed in the paper.

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 bounding, and (3) reduction. To be self-contained, we provide the proofs of all techniques in our full technical report [?] 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 on a variety of graphs and k values (rather than only on a subset of favorable scenarios to report). We also provide algorithms for variants of MkP that find all the MkPs or the MkP with the most edges, as well as parallel version of all our algorithms.

Through extensive experiments, we find that different pruning techniques can have drastically different performance impacts for different k and datasets. For example, among the two competitive branching methods, the partitioning-based one by [?] performs the best (up to $10\times$ faster) when k is small, but for larger values of k , an alternative pivot-based branching method [?] finds MkP up to thousands of times faster, so we cannot always stick to one strategy.

We uncover many important new insights on how to configure the set of pruning rules to avoid performance pitfalls. In particular, the AI community has actively pursued the upper-bound-based techniques to prune an entire branch of unpromising subgraph search space. However, those techniques are mostly useful instead as a branching method pioneered by DiseMKP [?] when k is small, but can backfire a lot when k becomes large. In the latter case, the

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

pivot-based branching method of Maple [?] works the best, but this branching approach is what we discover from their code implementation, which is totally different from what their paper originally proposes. Moreover, the concrete upper-bounding strategy choice does not impact the performance much, even when applying the cost model of Seesaw [?] to adaptively choose among the strategies.

On the other hand, the DB community mainly focuses on designing reduction rules to reduce the size of candidate sets for subgraph expansion during exploration. The latest algorithm kPlexT [?] (SIGMOD’24) proposes a new branching method critical in the proof of their improved worst case time complexity, but empirically we find it not competitive to those proposed by the AI community. In contrast, even recently, kPlexT still manages to find a new reduction rule that continues to significantly improve the search performance. In contrast, its prior version kPlexS [?] was the winner at that time (just two years ago) thanks to an incremental reduction technique called CTCP; but CTCP was found to be only worthwhile at the top-level subgraph exploration, but backfires if it is further applied with the lower-level branches due to the overheads of doing CTCP.

The main contributions of this paper is organized as follows:

- We conduct a timely and thorough literature review of nine recent MkP algorithms and summarize their pruning techniques into three categories. We also design a unified algorithmic framework, called Unified MkP (abbr. U-MkP), which can integrate all these techniques to facilitate benchmarking.
- We conduct extensive experiments to uncover many important new insights on the performance impacts of various pruning techniques, and provide recommendations on how to choose among these techniques to avoid performance pitfalls.
- We also provide algorithms for variants of MkP that find all MkPs or the MkP with the most edges, to avoid missing important dense structures due to returning only one MkP.
- We parallelize U-MkP using a task-based approach with timeout mechanism for load balancing to scale up almost ideally.

The rest of this paper is organized as follows. Section ?? introduces our notations and the branch-and-bound framework adopted by MkP algorithms for subgraph exploration. Then, Section ?? overviews our U-MkP framework and introduces the types of pruning techniques with a focus on upper bounding ones. Subsequently, Section ?? summarizes the various branching methods, and Section ?? summarizes the various reduction methods. Finally, we discuss the MkP variants and parallelization in Section ??, report our comprehensive empirical studies in Section ??, review the related works in Section ??, and conclude in Section ??.

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 $d_{\overline{N}_G}(v) = |\overline{N}_G(v)|$. 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 $G[P]$ 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.

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 ?? of our technical report [?].

Moreover, the diameter of k -plexes with a relatively large size (which is usually the case for MkPs) is bounded. Specifically, [?] proved that if a k -plex P satisfies $|P| > 2k - c$, then $G[P]$ is a connected graph with the diameter $\Delta(P) \leq c$ ($c \geq 2$). Most works [?] only consider the case when $c = 2$ by assuming $|P| \geq 2k - 1$:

THEOREM 2.6. Given a k -plex P , if $|P| \geq 2k - 1$, then $\Delta(P) \leq 2$.

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

The assumption $|P| \geq 2k - 1$ is reasonable for a maximum k -plex P that we target, 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, such as one formed by two disjoint $(k - 1)$ -cliques.

Algorithm 1: Basic Branch-and-Bound Search

```

1 function BB_basic( $S, R, g$ )
2   if (reduce_and_prune( $S, R, g$ ) = true) then return
3   for each  $v \in R$  do
4     BB_basic( $S \cup \{v\}, R - \{v\}, g$ )
5      $R \leftarrow R - \{v\}$ 
```

Figure 2: Set-Enumeration Search Tree

Algorithm 2: 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  $d_G(\cdot)$ 
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  $P, ub$ 

```

Subgraph Exploration by Branch and Bound. All existing MkP algorithms follow (but are variants of) the branch-and-bound search framework shown in Algorithm ??, 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 ?? in Section ?. 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 ?? illustrates the search process of Algorithm ?? 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 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 ?? 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 ?? of [?] with Figure ??.

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 [?], which produces a degeneracy ordering of vertices. Appendix ?? of our technical report [?] provides an illustration of this process. Note that $D(G)$ is usually a small value (see Table ??) 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.

Figure 3: Illustration of an Iteration of Algorithm ??

Algorithm 3: Top-Level Branching

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    $P, ub \leftarrow$  find_init( $G$ ) # global variables set by Algo. ??
3   if  $|P| \geq ub$  then return  $P$ 
4   CTCP( $G, \emptyset, |P| + 1 - k, |P| + 1 - 2k$ )
5   Sort  $[v_1, \dots, v_n]$  following degeneracy ordering of  $G$ 
6   for  $i = 1$  to  $|V|$  do
7      $H_1 \leftarrow N(v_i) \cap \{v_{i+1}, \dots, v_n\}$ 
8      $H_2 \leftarrow N(H_1) \cap \{v_{i+1}, \dots, v_n\}$ 
9      $g \leftarrow$  the subgraph of  $G$  induced by  $(\{v_i\} \cup H_1 \cup H_2)$ 
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  $P$ 

```

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 [?] to compute $\langle P, ub \rangle$ while running the peeling algorithm in linear time. Algorithm ?? 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.

Figure ?? illustrates the i^{th} iteration of the while-loop in Algorithm ??, assuming that the degeneracy ordering is given by $[v_1, v_2, \dots, v_n]$. Specifically, after v_i is fetched by Line 4, vertices v_1, \dots, v_{i-1} have already been removed from V by Line 9 in previous iterations, so V in iteration i becomes $V_i = [v_i, \dots, v_n]$. 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 3.1. Any k -plex P with $v \in P$ has $|P| \leq d_G(v) + k$.

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

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.

Top-Level Branching. Recall from Theorem ?? that reasonably large k -plexes ($|P| \geq 2k - 1$) have diameter $\Delta(P) \leq 2$. In other words, for the top-level nodes v_i ($v_i = a, b, c, d$) in Figure ??, the branch $T_{\{v_i\}}$ under it only needs to consider a subgraph of G , denoted by g_i , where vertices are within 2 hops from v_i . It is thus worthwhile

to shrink G to g_i . **Figure 4: Illustration of CTCP.** 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 [?], kPlexS [?], and Maple [?]. 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 > 2 hops away from v , this cost is associated with each expansion of S and so not worthwhile.

Algorithm ?? shows the pseudocode to create top-level search branches $T_{\{v_i\}}$, where we assume there are two globally maintained 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 ?? described above, 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 [?] 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 . 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).

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 $BB(\cdot)$ simply as $BB_basic(\cdot)$ presented in Algorithm ??, and Section ?? 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 ??) 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 $\{v_1, \dots, v_{i-1}\}$ have already been removed in previous iterations.

Core-Truss Co-Pruning (CTCP). kPlexS [?] 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 ?? in our technical report [?]). Figure ?? shows an illustration where vertices and edges below thresholds are alternately pruned. kPlexS [?] shows that the reduced graph by CTCP is guaranteed to be no larger than that computed by BnB [?], Maplex [?] and KpLeX [?] and was then the most efficient MkP algorithm.

We adopt the efficient implementation of CTCP($G, Q_v, lb_changed, \tau_v, \tau_e$) from kPlexS [?], 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

Algorithm 4: 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  $g_{union} = g[S \cup R]$  is a  $k$ -plex then
5     if  $|S| + |R| > |P|$  then  $P \leftarrow S \cup R$ 
6     return true
7   if partition( $S, R, g$ ) =  $\emptyset$  then return true # Seesaw UB
8   return false

```

Figure 5: Reported Algorithm Dominance Relationships

$lb_changed = true$, 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 ?? 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 $BB(\cdot)$ 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 ?? that $BB(\cdot)$ first calls $reduce_and_prune(\cdot)$ over the instance (S, R, g) for further reduction before extending S with vertices from R , the pseudocode of which is shown in Algorithm ?. Specifically, Line 1 first checks if the current P already reaches the maximum possible size ub , and if so, it returns *true* to notify $BB(\cdot)$ to terminate its subgraph exploration (see Line 2 of Algorithm ??). Right after the call of $BB(\cdot)$ in Line 10 of Algorithm ??, 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 ??) in Line 3 to reduce the candidate size of R . Before extending S with vertices in R (e.g., Lines 3–5 in Algorithm ??), we first conduct a look-ahead pruning in Line 4 to see if the entire $g_{union} = g[S \cup R]$ is a k -plex, and if so, Line 6 returns *true* so that Line 2 of Algorithm ?? 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 $BB(\cdot)$ terminate its exploration of T_S . The function $partition(\cdot)$ (to be introduced in Section ??) 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 ??, including kPlexT [?], Maple [?], Seesaw [?], DiseMKP [?], kPlexS [?], KpLeX [?], Maplex [?], BnB [?] and BS [?]. Table ?? 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

Table 1: MkP Algorithms and Their Pruning Techniques

Figure 6: Illustration of Partition-Based Upper Bounding

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 [?] and kPlexS [?] are from the DB community, actually the same group. Figure ?? shows the performance dominance relationships reported by the respective papers to their compared algorithms.

We have covered two reduction rules “Two-Hop” and “CTCP” since they are essential for describing our U-MkP framework, and we will introduce the other reduction rules in Section ?. As for branching rules, we will introduce them in Section ?.

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

Also, Table ? 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 ? to enable effective pruning (e.g., by CTCP in Line 4 of Algorithm ?), 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 [?].

4 BRANCHING METHODS

This section reviews two competitive branching methods, one using **partition-based upper bounding**, and the other using **pivoting**, both significantly beat a baseline binary branching method. For the former, we show that although the latest cost-model-driven Seesaw upper bound [?] was originally proposed only for entire-branch pruning (see Line 7 of Algorithm ?), it can actually be adapted for branching. For the latter, we actually reverse engineered this technique from the code of Maple [?], which unfortunately, implements something totally different from what their paper described so the branching technique was not documented anywhere before.

4.1 Partition-Based Upper Bounding

The partitioning-based upper bounding technique was proposed and improved by series of works: Maplex [?], KpLeX [?], DiseMKP [?], Seesaw [?]. Given (S, R) , 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, \dots, 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 ?? 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 ??, 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 ?? of our technical report [?]. We call this method as **R-based strategy**, since it computes independent sets directly from R without referring to S ’s content.

Seesaw [?] 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 ?.

We actually maintain $\delta_S(v)$ 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 ?? of [?] 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 [?]). Moreover, for each $\langle I_i, ub_i \rangle$ obtained where I_i is a maximal independent set, Seesaw [?] further relaxes I_i to include additional vertices from R without increasing the value of ub_i (see Lemmas 2–3 of [?]). 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 $\text{get_R_part}(\cdot)$ is specified by Algorithm 2 of [?].

S-Based Upper Bounding. KpLeX [?] 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 ??(a) where $S = \{a, b, c\}$ and vertices $d-i$ 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 [?] 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 ?? 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

ex-partition.pdf

Figure 7: An Example Graph for S-Based Partitioning

Figure 8: Set-Enumeration Search Tree (Horizontal Flip)

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 $\text{get_S_part}(\cdot)$ simply chooses the next partition among those non-empty π_i 's in Π , and the detailed algorithm is specified in Appendix ?? of our technical report [?].

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

$$\begin{aligned} \langle I_i^R, ub_i^R \rangle &\leftarrow \text{get_R_part}(R, S, g), \\ \langle I_i^S, ub_i^S \rangle &\leftarrow \text{get_S_part}(R, S, g, \Pi), \end{aligned}$$

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 [?]. Specifically, let us consider the horizontal flip of the set-enumeration tree in Figure ??, which we show in Figure ??. We can observe that when we scan the candidates from right to left till reaching a candidate $v \in R$, the set-enumeration

Figure 9: Illustration of Branching Set Computing

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 $S = \{v\}$ in Figure ??, 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 ?? 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 ?? again, assume that $R' = \{d, c\}$ for $S = \{v\}$, 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 ?? to construct R' (and hence $B = R - R'$). As Figure ?? illustrates, our goal is to find R' as a subset of R that is as large as possible so that $UB_S(R') \leq |P|$. 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 ?? to find $\langle I_i, ub_i \rangle$ one at a time and expand R' with I_i . Since we require $UB_S(R') \leq |P|$, 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 $\beta \geq ub_i$ (during S-based partition selection as shown in Figure ??, we also do not consider those candidates π_i with $ub_i > \beta$). Finally, as Figure ?? 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 ?? in Appendix ?? of our technical report [?] shows the pseudocode of $\text{partition}(S, R, g)$ 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 ?? allows additional branch pruning as follows: assume that we have processed the top-level branches $T_{\{v_1\}}, T_{\{v_2\}}, \dots, T_{\{v_{i-1}\}}$ 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 $T_{\{v_{i+1}\}}$.

Algorithm 5: 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( $S, R, g, \sigma_{seed}$ )
3   if (reduce_and_prune( $S, R, g$ ) = true) then return
4    $B \leftarrow \text{partition}(S, R, g)$ 
5    $R' \leftarrow R \setminus B$ 
6   Sort  $B = [v_1, \dots, v_n]$  following degeneracy ordering of  $g$ 
7   for  $i = |B|$  to 1 do
8     if ( $\sigma_{seed} = \text{true}$ ) then  $\sigma_{up} \leftarrow \text{false}$ 
9     if ( $\sigma_{up} = \text{true}$ ) then return
10     $\tau_{old} \leftarrow |P|$ 
11    BB_part( $S \cup \{v_i\}, R', g, \text{false}$ )
12    if  $|P| > \tau_{old}$  then  $\sigma_{up} \leftarrow \text{true}$ 
13     $R' \leftarrow R' \cup \{v_i\}$ 
```

To see this, consider Figure ?? 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 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 ?? shows the partition-based branch-and-bound algorithm to be called by Algorithm ?? with $\sigma_{seed} = \text{true}$. 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-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 ?? (BB_part(.)) to compute the branching set, the partition(.) function needs to first initialize $\Pi = \cup_{v \in S} \pi_v$ where for each $v \in S$, $\pi_v = R - N(v)$.

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 ?? 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 ?? of [?] with Figure ?. 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.

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 [?] tested large values for k), but only adopted by S-based methods KpLeX [?] and DiseMKP [?] with the scheme not clearly explained, so later works such as kPlexT [?] 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 $UB_S \leq |P|$ in Line 7 of Algorithm ??, we evaluate if $B = \emptyset$ in Line 7, which is equivalent.

4.3 Binary Branching

As Table ?? 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. Interestingly, kPlexT [?] 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 [?], as shown in Algorithm ??, can significantly improve performance, and is often many orders of magnitude faster for branching for large k values. Interestingly, Algorithm ?? is reverse-engineered from Maple's code, and their paper [?] 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 ??) 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 $\delta_S(v_i) = k - |\bar{N}_S(v_i)|$), reducing the pruning power of BB_part(.) (Algorithm ??). We recommend BB(.) to choose BB_part(.) when $k \leq 5$ and BB_pivot(.) otherwise (see Appendix ?? of [?]).

We now explain Algorithm ??, 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

Algorithm 6: Pivoting-Based Branch and Bound

Input: P, B are global variables, v_i is the top-level vertex
 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  $S = \emptyset$  or  $S.top() \in S$  or  $S.top() \notin R$  then
4     if  $\exists v \in R, s.t. (v, v_i) \notin E(g)$  and  $(\delta_S(v) = 1$ 
       or  $\delta_S(v_i) = 1$  or  $d_g(v) + k \leq |P| + 1)$  then
5        $S \leftarrow \{v\}$ 
6     else
7        $v_p = A$  vertex with minimum  $d_g(\cdot)$  in  $R$ 
8        $S \leftarrow R - N_g(v_p)$ 
9       Sort  $S$  in descending order of  $d_g(\cdot)$ 
10   $v \leftarrow S.pop()$ 
11  BB_pivot( $S \cup \{v\}, R - \{v\}, g$ )
12  Remove  $v$  from  $g$ 
13   $S.clear()$ 
14  BB_pivot( $S, R, g$ )
```

Figure 10: Illustration of Pivot Selection

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 ??(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 (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 ?? of [?]) 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 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 $d_g(\cdot)$, and pushes the candidates in R to S so that they are popped in Line 10 in non-decreasing order of $d_g(\cdot)$ (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 S since as Figure ??(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 ??, reduction rules in Line 2 may move some candidates directly from R to S , leading to $S.top() \in S$; and may prune some candidates already added to S , leading to $S.top() \notin R$, 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 S will be popped for binary branching.

5 REDUCTION METHODS

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

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 $d_S(v)$ and $d_g(v)$ 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 ?? of [?] for the details on incremental degree maintenance.

Section ?? discussed **TwoHop** and **CTCP**. We will now discuss the remaining ones: **RR1**–**RR3**, **BR1**–**BR2**, and **UBR1**–**UBR2**. Note that while **kPlexS** [?] 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** [?].

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 $\delta_S(\cdot)$. 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** [?], and **UBR2** by **kPlexT** [?], 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 $d_g(\cdot)$. Specifically, **BR1** is first proposed by **kPlexS** [?] 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** [?] based on conditions related to $d_g(\cdot)$ 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 only update the current largest k -plex P if a newly found one has more vertices, or the same number of vertices but more edges.

The Parallel Algorithm. Our MkP framework can be easily parallelized with a task-based model [?] that has been extensively

applied in compute-intensive graph search and mining problems [??]. 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 ?. 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 [??] 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 [?] 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 ?? of Algorithm ? removes v_i when $T_{\{v_i\}}$ is processed, and Line ?? 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 ??–?? 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 ??) when constructing g_i .

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. We also provide important insights on how to configure the set of pruning rules to avoid performance pitfalls. 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 20 datasets as summarized in Table ??, where d_{max} and d_{avg} indicate the maximum degree and average degree, respectively; and D is the degeneracy. These datasets span a wide range of graph sizes and categories, and a detailed description can be found in Appendix ?? of [?]. 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 large real graphs are too sparse and can be solved in sub-second).

Table 2: Datasets

We set the time limit as 1800 s (s = seconds). We use \times 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 ?? in our technical report [?] for the MkP sizes of all datasets for all values of k .

Choice of Upper-Bounding (UB) Techniques: Recall that Section ?? 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 all have at most a marginal benefit, and in some settings, the cost of upper bound computation itself is quite high. For example, the running time on *soc-digg* when $k = 5$ is **102.7** s, 367.2 s, 220.1 s, and OOT for no UB pruning, S-Based, R-Based, and SR-Based UB pruning, respectively. For $k = 7$, the winner is S-Based UB pruning with **56.1** s, but with no UB pruning, it completes in 57.0 s which is close. The full results are shown in Table ?? of our technical report [?], based on which we recommend to apply no bounding technique by default.

Choice of Branching Method: Section ?? presented two competitive branching methods: partition-based and pivoting-based, where partition-based methods are based on the above-mentioned 3 UB techniques. Table ?? shows a comparison of these branching methods when $k = 2, 3, 8$ and 10, and the full results are shown in Table ?? of our technical report [?]. 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 ?? shows, when $k = 3$, it took 29.2 s on *keller4* when applying S-based branching but 274.5 s ($\sim 10\times$ 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 0.7 s ($> 2570\times$ faster) when applying pivoting-based branching.

Finally, as we shall show in Tables ??–??, kPlexT [?], 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: Section ?? presented various reduction rules applied by various MkP algorithms. We tested the efficiency of those rules by disabling each rule and observing the performance difference. Among the reduction rules, conditions of **RR1–RR3** are

Table 3: Comparison of Bra

very efficient to check and essential to good performance, so they are always used by any MkP algorithm and we enable them by default.

Reduction rules **BR1** and **BR2** aim to add some vertices directly to S . We tested their performance in our study, but to our surprise, they merely make any difference for almost all datasets and values of k . This is likely because the condition seldom holds to allow pruning, but since the conditions of both rules are efficient to check, there is no significant overhead incurred, so U-MkP enables them by default. The results on effect of BR1 and BR2 are in Table ?? of [?].

The CTCP method detailed in Section ?? significantly reduces the input graph G at the beginning of the program. However, we observed that applying CTCP inside the BB(.) procedure (i.e., in `reduce_and_prune(.)`), as kPlexS [?] does, is expensive. Specifically, when enabling CTCP inside BB(.), the execution time was increased by over 10× for many datasets, and even worse, some datasets could not complete within the time limit. Thus, U-MkP disables it in BB(.). The results on effect of CTCP is in Table ?? of [?].

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 CTCP, so they are usually used together if enabled in BB(.). However, since dynamically maintaining these counts is expensive, U-MkP disables UBR1 by default along with CTCP. On the other hand, we observed that **UBR2** is highly effective on many datasets. Table ?? 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 ?? of our technical report [?]. We can see that enabling UBR2 can speed up computing by up to a few orders of magnitude. 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 0.7 s, so the speedup is over 2570×.

Comparison of MkP Algorithms. Recall from the algorithm dominance graph shown in Figure ?? that among existing MkP algorithms, only kPlexT [?], Maple [?] and DiseMKP [?] 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 ?? 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

Table 4: Effect of Reduction Rule UBR2

tab-UBR2-34810.pdf

for time-consuming jobs. Even when U-MkP is not the best, it is often very close to the best. For example, U-MkP is 36.3× faster than DiseMKP (the second best) on *johnson8-4-4* when $k = 5$, and it is only 1.2× 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 often many times slower than U-MkP. Maple does not win except on *soc-orkut* when $k = 5$, showing that its pivot-based branching method is not effective for small k values. Finally, DiseMKP actually wins on three synthetic graphs when $k = 2$ and 3 thanks to its application of S-based branching, but it does not utilize effective reduction rules like UBR2, so shows no performance advantage on real graphs.

Table ?? 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 even when $k = 10$; while even though kPlexT is the latest algorithm that supports large values of k , it is still slower than U-MkP for $k = 15, 20$ due to its suboptimal approach for binary branching. We skip the synthetic datasets in Table ?? since on them, either all algorithms run OOT, or they can finish in sub-second. We provide full results on all datasets for all values of k in Table ?? of our technical report [?].

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

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

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

tab-parallel.pdf

Performance of MkP Variants. Section ?? presents a two-phase approach to compute all MkPs as well as the densest MkP. Table ?? reports the results of running this variant for $k = 5$, 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 *sc-msdoor* has 17,799,765 MkPs so are not selective, many graphs have tens to a couple of hundreds 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 most graphs. For example, on *sc-msdoor*, Phase 1 finds an MkP with 442 edges but

Table 7: Finding All MkPs and Densest MkP



Figure 11: Speedup Ratio of Parallel U-MkP

the densest has 540 edges. The results for all values of k are given in Table ?? of our technical report [?].

Performance of U-MkP Parallelization. We implemented the parallel version of U-MkP based on the description in Section ?. Table ?? reports the running time 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 ?? 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 ?. We can see that our parallelization can achieve a speedup ratio of up to $28.1\times$ with 32 threads. The complete results for all datasets can be found in Table ?? of our technical report [?].

8 RELATED WORK

This paper has surveyed the algorithms for maximum k -plex finding as summarized in Table ?. There are also works for finding maximal k -plexes (i.e., those without a supergraph that is also a k -plex) [? ? ? ? ?], often with a size threshold q to find only those with at least q vertices. Among them, ListPlex [?] 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 [?], but is also efficient and hence adopted by a later work [?]. The authors of ListPlex aim to apply this idea to

the MkP problem by proposing Maple [?], 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 [?].

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 for different values of k and different datasets. Through extensive experiments, we obtained important new insights on the effectiveness of these techniques, and recommended a default configuration of U-MkP that works generally the best.

We note that providing a timely, in-depth summary and empirical study of classical graph problems, which have seen significant recent algorithmic advancements, is crucial to avoid redundant efforts and to motivate further development. It is important for such a study to categorize and unify existing techniques under one framework to facilitate comparison for benchmarking and technique selection. An example is subgraph matching which saw a surge of algorithm development close to Year 2020 [? ? ?], so Sun and Luo [?] conducted an experimental study to summarize the techniques in the previous algorithms under a unified three-step framework for comparison to provide guidelines on how to select the optimal configuration. This work then motivated many more algorithm developments in subgraph matching [? ? ?] that finally lead to another recent experimental study [?]. The goal of our U-MkP framework and timely experimental study is to bring such success to the MkP problem that has recently seen a surge of algorithm development as summarized in Figure ?, to motivate more algorithm development.

Figure 12: Illustration of k -Cores

Figure 13: Second-Order Pruning

A k -CORE DECOMPOSITION

The graph degeneracy can be computed by the process of k -core decomposition [?]. 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. ?? shows the 1-core, 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. ?? since A is in 2-core but not 3-core.

The process of k -core decomposition [?] can be computed by a peeling algorithm that repeatedly removes a vertex with the minimum current degree at a time [?]. For the graph G in Fig. ??, 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.

B PROOF OF THEOREM ??

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. \square

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

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  $I, ub$ 

```

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

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 ??.

D PROOF OF THEOREM ??

Theorem ?? says that for any k -plex P containing v , it holds that $P \leq d_G(v) + k$. We show this by contradiction. Assume that $|P| > d_G(v) + k$, then since $v \in P$ and P is a k -plex, we have $d_G(v) \geq |P| - k > d_G(v)$, leading to a contradiction. \square

E BASIC R-BASED PARTITION COMPUTING

Algorithm ?? shows the pseudocode of Maplex [?] 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 2–3 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 [?] rather than this simple version.

F S-BASED PARTITION COMPUTING

Algorithm ?? 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 dis 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).

G COMPUTING SR-BASED BRANCHING SET

Algorithm ?? 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 ??

Algorithm 8: S-Based Partitioning of Candidate Set R

```
1 # Initializing auxiliary structure  $\Pi$ 
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( $S, R, g, \Pi$ )
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  $\pi^*, ub^*$ 
```

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      $\pi^* \leftarrow \emptyset, ub^* \leftarrow 0, dise^* \leftarrow 0$ 
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     $I, ub \leftarrow \text{get\_R\_part}(S, B, g)$ 
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  $B$ 
```

with Figure ??, where moving a vertex from B to R' takes only $O(1)$ time.

Line 2 also initializes the remaining quota β , 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

Figure 14: Dual-Array Data Structure for Set Maintenance

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.

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

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 ??. 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 ??, 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 ??(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 ??(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 $BB(\cdot)$ with minimal cost.

For example, in Line ?? of Algorithm ??, we add vertex v from R to S using the above mentioned method, and then at Line ?? 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 ??, 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.

I INCREMENTAL DEGREE MAINTENANCE

Note that support $\delta_S(\cdot)$ as defined in Definition ?? is frequently needed by the pruning techniques, such as $ub_i = \min\{|\pi_i|, \delta_S(v_i)\}$ in S-based strategy, Line ?? of Algorithm ?? for pivot-based branching, and reduction rules RR1, RR2, UBR1 and UBR2. Also, $d_g(\cdot)$ are also used in many places such as Line ?? of Algorithm ?? for pivot-based

Algorithm 10: Adaptive Branch-and-Bound Method

```

1 function BB( $S, R, g$ )
2   if  $k \leq 5$  then
3     BB_part( $S, R, g, \text{true}$ )
4   else
5     BB_pivot( $S, R, g$ )

```

branching, and reduction rules RR3, BR1 and BR2 (which needs $\overline{d_g}(v) = |V(g)| - d_g(v)$). Therefore, we maintain $d_g(\cdot)$ and $\delta_S(\cdot)$ 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 $u \in N_g(v)$.

J ADAPTIVE BRANCH AND BOUND

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

K 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 ??, the proof of which is given in Appendix ??.

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 [?]:

THEOREM K.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 ??:

UBR1.pdf

Figure 15: Upper Bound Illustration of Theorem ??

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 ??:

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 ??: (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

$$|P| = |S| + |f_S(N_R(u) \cap N_R(v))| + |f_S(N_R(u) \cap \overline{N_R}(v))| + |f_S(\overline{N_R}(u) \cap N_R(v))| + |f_S(\overline{N_R}(u) \cap \overline{N_R}(v))|$$

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. \square

UBR2: This is a new upper-bound-based pruning rule proposed by the latest MkP algorithm kPlexT [?], which we find to be particularly effective in pruning R . Specifically, UBR2 first computes

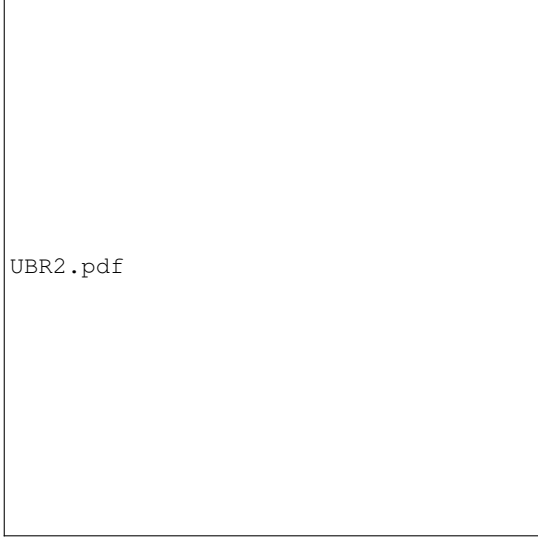


Figure 16: Illustration of UBR2

the following upper established by Theorem ??, which will then be further tightened (to be described soon):

THEOREM K.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:*

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

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

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 $\overline{d_{S_2}}(\cdot)$, and add the vertices one at a time to S' until the missing-edge quota of $\delta_{S'}(S_2)$ is reached (Let $[v_1, \dots, v_{|R'|}]$ be the sequence of vertices in R' sorted in non-decreasing order with respect to $\overline{d_{S_2}}(\cdot)$). That is, when adding the

Figure 17: Illustration of the Proof of BR1

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:

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

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

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

which completes the proof of Theorem ??. \square

To further tighten the bound of Equation (??), we would like to remove some vertices in the list $[v_1, \dots, v_{|R'|}]$ sorted by $\overline{d_{S_2}}(\cdot)$, so fewer vertices in the reduced list can be added to S' , since some vertices with small $\overline{d_{S_2}}(\cdot)$ 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 $\overline{d_{S_2}}(\cdot)$, 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 (??) but over \mathcal{R}_v rather than R' :

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

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

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 $d_S(\cdot)$.

BR1 is first proposed by kPlexS [?] 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:

- $d_g(v) = |V(g)| - 1$: 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.
- $d_g(v) = |V(g)| - 2$: 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 contradiction). 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:

- $u \notin S$ (see Figure ??(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.

- $u \in S$ (see Figure ??(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).

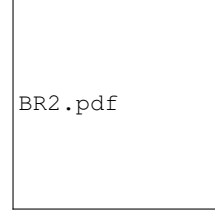


Figure 18: Illustration of the Proof of BR2

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. \square

Note that BR1 cannot be used if our goal is to find all MkPs. In contrast, BR2 is first proposed by BS [?] based on conditions related to $d_g(\cdot)$ 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 ??.

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 $\overline{d_{P \cup \{v\}}}(u) \leq \overline{d_g}(u) \leq k$.

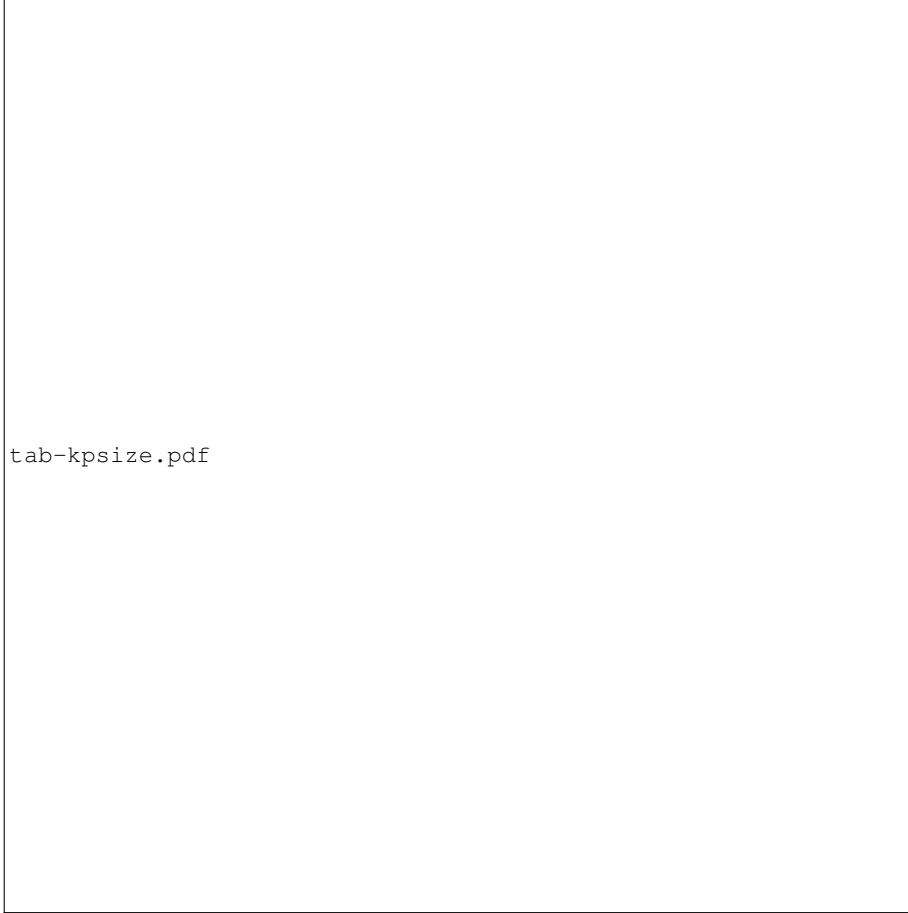
For $u \in P$ that is a neighbor of v , $\overline{d_{P \cup \{v\}}}(u) = \overline{d_P}(u) \leq k$ where the last inequality is because P is a k -plex.

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

L DATASET DESCRIPTION

We use 20 datasets as summarized in Table ??, where d_{max} and d_{avg} indicate the maximum degree and average degree, respectively; and D is the degeneracy. These datasets span a wide range of graph sizes and categories: (1) synthetic graphs from DIMACS: *hamming6-2* [?], *johnson8-4-4* [?], *keller4* [?], *brock200-2* [?] and *p-hat500-1* [?]; (2) social networks: *soc-buzznet* [?], *soc-LiveMocha* [?], *soc-gowalla* [?], *soc-youtube* [?], *soc-digg* [?], *soc-youtube-snap* [?], *soc-lastfm* [?], *soc-pokec* [?] and *soc-orkut* [?]; (3) Facebook networks (in specific): *socfb-Duke14* [?], *socfb-B-anon* [?] and *socfb-A-anon* [?]; (4) scientific computing: *sc-msdoor* [?] and *sc-ldoor* [?]; and (5) interaction network: *ia-wiki-Talk* [?]. We roughly categorize them as small and large graphs. The small ones are from DIMACS challenge and are very dense, so

Table 9: Size of MkP for Various Values of k



tab-kpsize.pdf

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 be solved in sub-second).

M COMPLETE RESULTS OF EXPERIMENTS

Due to space limit of the main paper, we include all our results in Tables ??–?? on the next few pages.

We set the time limit at 1800 seconds. We use \times 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 ?? shows the MkP sizes of all datasets for all values of k .

Table ?? 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, such as SR-based upper bounding on *soc-buzznet* when $k = 3, 4$.

Table ?? shows the full results of competitive branching methods. Table ?? shows the results on the effect of BR1 and BR2, while Table ?? shows the results on the effect of CTCP in recursion.

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

Section ?? presents a two-phase approach to compute all MkPs as well as the densest MkP. Table ?? 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 ?? 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 10: Comparison of Different Bounding Techniques

tab-SR-Bounding.pdf	
---------------------	--

Table 11: Comparison of Different Branching Methods

tab-branching.pdf	
-------------------	--

Table 12: Effect of Reduction Rules BR1 and BR2

tab-BR12.pdf	
--------------	--

Table 13: Effect of CTCP in Recursion

tab-CTCP.pdf	
--------------	--

Table 14: Effect of Reduction Rule UBR2

tab-UBR2.pdf	
--------------	--

Table 15: Execution Times of U-MkP

tab-fullUMKP.pdf	
------------------	--

Table 16: Results of All-MkP and Densest-MkP

appendix-Dense.pdf

Table 17: Execution Times of Our Parallel Algorithm

appendix-tab-parallel.pdf

Temporary page!

L^AT_EX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because L^AT_EX now knows how many pages to expect for this document.