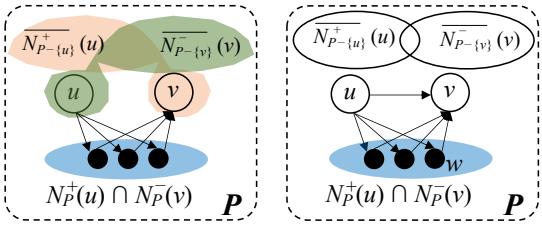


(a) $(u, v) \notin E$ (b) $(u, v) \in E$

Figure 7: Second-Order Pruning

A Proof of Theorem 3.1

Proof. For any $v \in P'$, we have $v \in P$ and since P is a (k_1, k_2)-plex, we have $\overline{d^+_P}(v) = \left|\overline{N^+_P}(v)\right| \le k_1$ and $\overline{d^-_P}(v) = \left|\overline{N^-_P}(v)\right| \le k_2$. Since $\overline{N^+_{P'}}(u) \subseteq \overline{N^+_P}(u)$ and $\overline{N^-_{P'}}(u) \subseteq \overline{N^-_P}(u)$, we have $\overline{d^+_{P'}}(v) = \left|\overline{N^+_{P'}}(v)\right| \le k_1$ and $\overline{d^-_{P'}}(v) = \left|\overline{N^-_{P'}}(v)\right| \le k_2$, so P' is also a (k_1, k_2)-plex. □

B Proof of Theorem 4.2

Proof. This can be seen from Figure 7. Since P is a (k_1, k_2)-plex, we have $\overline{d^+_{P-\{u\}}}(u) \le k_1 - 1$ (since $\overline{d^+_P}(u) \le k_1$) and $\overline{d^-_{P-\{v\}}}(v) \le k_2 - 1$ (since $\overline{d^-_P}(u) \le k_2$).

In Case (a) where $(u, v) \notin E$, any vertex $w \in P$ can only fall in the following 3 scenarios: (1) $w \in \overline{N^+_{P-\{u\}}}(u)$ (which contains v), (2) $w \in \overline{N^-_{P-\{v\}}}(v)$ (which contains u), and (3) $w \in N^+_P(u) \cap N^-_P(v)$. Note that w may be in both (1) and (2). So we have:

$$\begin{aligned} |P| &= |N^+_P(u) \cap N^-_P(v)| + |\overline{N^+_{P-\{u\}}}(u) \cup \overline{N^-_{P-\{v\}}}(v)| \\ &\le |N^+_P(u) \cap N^-_P(v)| + |\overline{N^+_{P-\{u\}}}(u)| + |\overline{N^-_{P-\{v\}}}(v)| \\ &\le |N^+_P(u) \cap N^-_P(v)| + (k_1 - 1) + (k_2 - 1) \\ &= |N^+_P(u) \cap N^-_P(v)| + k_1 + k_2 - 2, \end{aligned}$$

so $|N^+_P(u) \cap N^-_P(v)| \ge |P| - k_1 - k_2 + 2 \ge q - k_1 - k_2 + 2$.

In Case (b) 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) $w \in \overline{N^+_{P-\{u\}}}(u) \cup \overline{N^-_{P-\{v\}}}(v)$, so:

$$\begin{aligned} |P| &= 2 + |N^+_P(u) \cap N^-_P(v)| + |\overline{N^+_{P-\{u\}}}(u) \cup \overline{N^-_{P-\{v\}}}(v)| \\ &\le 2 + |N^+_P(u) \cap N^-_P(v)| + |\overline{N^+_{P-\{u\}}}(u)| + |\overline{N^-_{P-\{v\}}}(v)| \\ &\le 2 + |N^+_P(u) \cap N^-_P(v)| + (k_1 - 1) + (k_2 - 1) \\ &= |N^+_P(u) \cap N^-_P(v)| + k_1 + k_2, \end{aligned}$$

so $|N^+_P(u) \cap N^-_P(v)| \ge |P| - k_1 - k_2 \ge q - k_1 - k_2$. □

C Counter Computation

Initial Counter Computation. Figure 8 shows how we compute the initial values of the counters. Specifically, consider an arbitrary (starting) edge (v_1, v_2), then we can obtain a triangle $\Delta v_1 v_2 v_3$ with four possible edge orientations (a)–(d) as shown on the left of Figure 8. Note that the three acyclic configurations are isomorphic: each triangle of type (b) (resp. (c)) has a one-to-one mapping to

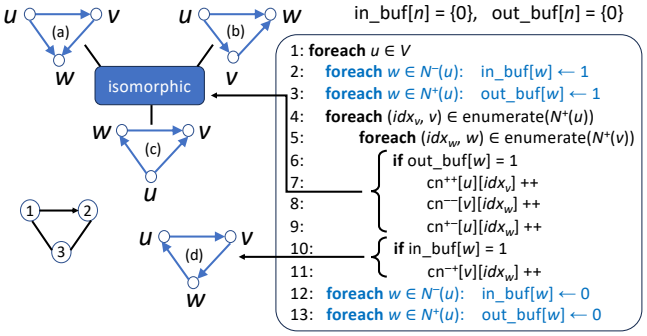


Figure 8: Initial Counter Computation

a triangle of type (a), so we just need to enumerate triangles of type (a) to avoid redundancy.

Figure 8 shows this enumeration where $v \in N^+(u)$ (Line 4) and $w \in N^+(v)$ (Line 5), and Line 6 checks if $w \in N^+(u)$. Note that for each vertex u, we save its counters, such as $cn^{++}(u, :)$, also sparsely to be space-efficient. For each position idx in the adjacency list $N^+(u)$ such that $N^+(u)[idx] = v$, we save $cn^{++}[u][idx] = cn^{++}(u, v)$.

If Line 6 finds that $w \in N^+(u)$, we increment cn^{++}, cn^{--}, and cn^{+-} for the triangle's three edges properly. To avoid repeated binary search when checking if $w \in N^+(u)$ in Line 6, we use a bitmap out_bf of size $|V|$ that is initialized as all zeros. When we process a vertex $u \in V$ in Line 1, we first set the bits of all the out-neighbors in $N^+(u)$ to be 1 in Line 3, so that Line 6 can be implemented as a simple bit check. These bits are cleared in Line 13 before we continue to process the next vertex $u \in V$ in Line 1.

The cyclic triangle of type (d) increments cn^{-+}. In Line 10, if we find $w \in N^-(u)$, we increment $cn^{-+}(v, w)$ only since $cn^{-+}(w, u)$ (resp. $cn^{-+}(u, v)$) will be incremented when we enumerate from starting vertex v (resp. w) in Line 1. □

Incremental Counter Maintenance. Let us consider edge peeling first. When we remove an edge (u, v), (1) we decrement $d^+(u)$ and $d^-(v)$. (2) For each $w \in N^+(u) \cap N^-(v)$, we decrement $cn^{++}(u, w)$ and $cn^{--}(w, v)$ (see Figure 7(b) to get the intuition). Similarly, (3) for each $w \in N^+(u) \cap N^+(v)$, we decrement $cn^{+-}(u, w)$ and $cn^{--}(v, w)$; (4) for each $w \in N^-(u) \cap N^+(v)$, we decrement $cn^{-+}(w, u)$ and $cn^{-+}(v, w)$; (5) for each $w \in N^-(u) \cap N^-(v)$, we decrement $cn^{++}(w, u)$ and $cn^{+-}(w, v)$.

Whenever the decrement of a common-neighbor counter for an edge e causes its value to be below the threshold stipulated by Theorems 4.2–4.5, e is added to an edge queue Q_e for further peeling. Vertices whose degrees get decremented below the thresholds stipulated by Theorem 4.1 are also added to a vertex queue Q_v. When edge peeling completes (i.e., $Q_e = \emptyset$), we conduct vertex peeling starting from the vertices in Q_v. The removal of each vertex w in Q_v will also incrementally decrement the degrees (resp. common-neighbor counters) of w's neighbors (resp. neighbor-pairs), for which we omit the details. Once vertex peeling completes, new edges may have been added to Q_e so we start another round of edge peeling. We alternate between edge peeling and vertex peeling until both Q_e and Q_v are empty. This process is similar to the CTCP algorithm (Algorithm 3) of [20] in the k-plex context. □

Note that the above process of shrinking G by alternating edge peeling and vertex peeling is executed on top of the $(q-k_1, q-k_2)$-core of G which is initially computed by a vertex peeling algorithm, since the latter is more efficient in pruning unpromising vertices as a preprocessing.

D Proof of Theorem 5.1

Proof. Consider any two vertices u, v in a (k_1, k_2)-plex P where $k_1, k_2 \le \frac{q+1}{2}$, we show that they cannot be more than 2 hops apart (i.e., cannot fall out of the 6 cases in Figure 9).

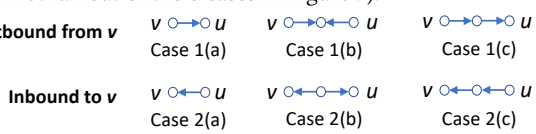


Figure 9: Cases for Two-Hop Diameter Upper Bound

Without loss of generality, we only consider the path from v to u where the first edge is outbound from v, i.e., Cases 1(a)–(c). Cases 2(a)–(c) are symmetric and can be similarly proved.

If v directly points to u, we are done since Case 1(a) occurs. Now assume that edge $(v, u) \notin E$, and we show that:

- **Case (I):** $(\mathbf{u}, \mathbf{v}) \notin \mathbf{E}$, then both Case 1(b) and Case 1(c) should be satisfied. **(i) We first prove Case 1(b).** Note that $u \notin N^+(v)$ and $v \notin N^+(u)$. Since $k_1 \le \frac{q+1}{2} \le \frac{|P|+1}{2}$, u and v each points to at least $|P| - k_1 \ge \frac{|P|-1}{2}$ other vertices in P, so they must share an out-neighbor; otherwise, there exist $2 \cdot \frac{|P|-1}{2} = |P| - 1$ vertices other than u and v, leading to a contradiction since there will be at least $(|P|+1)$ vertices in P when adding u and v. **(ii) We next prove Case 1(c).** Note that $v \notin N^-(u)$ and $u \notin N^+(v)$. Since $k_1 \le \frac{q+1}{2} \le \frac{|P|+1}{2}$, v points to at least $|P| - k_1 \ge \frac{|P|-1}{2}$ other vertices in P; also since $k_2 \le \frac{q+1}{2} \le \frac{|P|+1}{2}$, u is pointed to by at least $|P| - k_2 \ge \frac{|P|-1}{2}$ other vertices in P. So, $N^+(v)$ and $N^-(u)$ must intersect as illustrated by Figure 9 Case 1(c); otherwise, there will be $(|P|+1)$ vertices in P when adding u and v.
- **Case (II):** $(\mathbf{u}, \mathbf{v}) \in \mathbf{E}$, then Case 1(c) should be satisfied. The proof is the same as (ii) above. Note that we cannot guarantee Case 1(b) anymore, since $v \in N^+(u)$, i.e., v can be one of the at least $|P| - k_1 \ge \frac{|P|-1}{2}$ neighbors of u, invalidating the prove for (i) above.

Symmetrically, consider the path from v to u where the first edge is inbound to v, i.e., Cases 2(a)–(c). If u directly points to v, we are done since Case 2(a) occurs. If edge $(u, v) \notin E$:

- **Case (III): edge (v, u) does not exist in** G, then both Case 2(b) and Case 2(c) should be satisfied. The proof is symmetric to Case (I) above and thus omitted.
- **Case (IV): edge (v, u) exists in** G, then Case 2(c) should be satisfied. The proof is symmetric to Case (II) above.

Putting the above discussions together, we obtain the following 4 cases, for each of which we explain how to exclude an impossible candidate u from $ext(S)$ given a vertex $v \in S$, where $ext(S)$ denotes the set of candidate vertices that can extend S into a valid (k_1, k_2)-plex.

- **Case A:** $(v, u) \in E$ **and** $(u, v) \in E$**.** In this case, we always have $u \in ext(S)$.
- **Case B:** $(v, u) \notin E$ **and** $(u, v) \in E$**.** Based on Case (II) above, we have $u \in ext(S)$ only if a path $u \leftarrow w \leftarrow v$ exists in G for some $w \in V$ $(w \ne u, v)$.
- **Case C:** $(v, u) \in E$ **and** $(u, v) \notin E$**.** Based on Case (IV) above, we have $u \in ext(S)$ only if a path $u \rightarrow w \rightarrow v$ exists in G for some $w \in V$ $(w \ne u, v)$.
- **Case D:** $(v, u) \notin E$ **and** $(u, v) \notin E$**.** Based on Case (I) above, we have Condition (C1): $u \in ext(S)$ only if both Case 1(b) and Case 1(c) in Figure 9 are satisfied. Similarly, based on Case (III) above, we have Condition (C2): $u \in ext(S)$ only if both Case 2(b) and Case 2(c) are satisfied. Combining both conditions, $u \in ext(S)$ only if there exist $w_1, w_2, w_3, w_4 \in V - \{u, v\}$ such that $u \leftarrow w_1 \leftarrow v$ and $u \leftarrow w_2 \rightarrow v$ and $u \rightarrow w_3 \leftarrow v$ and $u \rightarrow w_4 \rightarrow v$.

Once we have applied the above rules to prune $ext(S)$ to exclude invalid candidates u, let us abuse the notation to use G again to denote the resulting graph induced by $S \cup ext(S)$ after pruning. Note that we can apply this diameter-based pruning on the pruned G again, since some vertex w in Case B (resp. Case C) could have been pruned by Case C (resp. Case B) in the previous iteration, causing some required paths to disappear, further invalidating more vertices u from $ext(S)$. This pruning can be iteratively run over G.

Based on the above idea, Algorithm 2 computes the set of vertices in $ext(S)$ that are not 2-hop pruned by a vertex $v \in S$. Specifically, Line 1 computes O (resp. I) as the set of v's out-neighbors (resp. in-neighbors) u that belong to Case B (resp. Case C). Then, Line 3 recovers S_O (resp. S_I) as the set of v's all non-pruned out-neighbors (resp. in-neighbors) w in Case B (resp. Case C) with path $v \rightarrow w \rightarrow u$ (resp. $v \leftarrow w \leftarrow u$). Note that $N^{\pm}(v) \subseteq ext(S)$ based on Case A so its vertices cannot be further pruned, so the iterative pruning is contributed by the shrink of sets O and I.

Next, Line 4 prunes away those vertices $u \in O$ (resp. $u \in I$) that cannot find a path $u \rightarrow w$ (resp. $u \leftarrow w$) for some non-pruned $w \in N^-(v)$ (resp. $w \in N^+(v)$), which is based on Case C (resp. Case B). Note that if O or I shrinks in Line 4, Line 5 will trigger another iteration of pruning. When the loop of Lines 2–5 exits, we have O (resp. I) being the remaining vertices $u \in ext(S)$ in Case B (resp. Case C) after iterative pruning. Finally, Line 6 computes the set B of vertices where u satisfies Case D w.r.t. v, and Line 7 unions the 4 disjoint candidate sets that correspond to Cases A–D, respectively, to obtain the final 2-hop pruned $ext(S)$ for a vertex $v \in S$. We denote this set as $\mathbb{B}(v)$, which is returned by Line 7. □

E Complexity Analysis

Given a directed graph $G = (V, E)$, we evaluate the worst-case running time of Algorithm 4. Let $T(m, n)$ be the worst-case running time of function BK2$(G, k_1, k_2, q, P, C, X)$ where $m = |C|$ and $n = |P \cup C|$, and we define $k = max(k_1, k_2)$. Next, we analyze $T(m, n)$ by considering four different cases:

I. When $C = \emptyset$ (Lines 4–6), then

$$T(m, n) = T(0, n) \le c_1, \tag{4}$$

where c_1 is a constant.

II. When $P \cup C$ is not a (k_1, k_2)-plex and the pivot vertex is in P (Lines 8–17). Here, $p+1$ branches are created each with a smaller candidate set C'. Before each call the sets C' and X' are further shrunk in Lines 15–16 which takes time $O(|V|^2)$. The worst-case running time of this case is, therefore

$$T(m,n) \leq (p+1)c_2|V|^2 + \sum_{i=1}^{p} T(m-i, n-1) + T(m-d, n-d+p), \quad (5)$$

where c_2 is a constant.

III. When $P \cup C$ is a (k_1, k_2)-plex (Lines 19–24), then no further branches are created, and the cost in this case is only incurred by updating X' in Line 22, which is

$$T(m,n) \leq c_2|V|^2 \quad (6)$$

IV. When $P \cup C$ is not a (k_1, k_2)-plex and the pivot vertex is in C (Lines 25–30). The cost in this case is

$$T(m,n) \leq c_2|V|^2 + T(m-1,n) + T(m-1,n-1).$$

In the worst case, no candidate vertex is reduced from $C - \{v_p\}$ in Line 28 so v_p remains the minimum degree vertex in the subsequent recursive call at Line 30 (within which $v_p \in P' = P \cup \{v_p\}$). Hence, by expanding $T(m-1,n)$ using Equation (5) we have:

$$T(m,n) \leq (p+2)c_2|V|^2 + \sum_{i=1}^{p} T(m-1-i, n-1) + T(m-1-d, n-d+p) + T(m-1, n-1). \quad (7)$$

Next, we compute a closed-form upper-bound of $T(m,n)$ which satisfies the Equations (4)–(7), and we can drop the second argument n from $T(m,n)$ since the base case expressions in Equations (4) and (6) are not a function of n. Then, the largest $T(m)$ is upper-bounded by Equations (4) and (7), i.e.

$$T(m) \leq \begin{cases} c_1 & \text{if } m = 0 \\ (p+2)c_2|V|^2 + \sum_{i=0}^{p} T(m-1-i) & \\ \quad + T(m-1-d) & \text{otherwise} \end{cases} \quad (8)$$

Since $p \leq d-1$ and $p < k$, the worst-case is reached when $p = k-1$ and $d = k$. Then, Equation (8) can be rewritten as:

$$T(m) \leq \begin{cases} c_1 & \text{if } m = 0 \\ (k+1)c_2|V|^2 + \sum_{i=0}^{k} T(m-1-i) & \text{otherwise} \end{cases} \quad (9)$$

We show that $T(m) = O(\alpha_k \gamma_k^m - \beta_k)$ where γ_k is largest real root of $x^{k+2} - 2x^{k+1} + 1 = 0$, $\beta_k = \frac{(k+1)c_2|V|^2}{k}$ and $\alpha_k = \frac{(2^k k+1)\beta_k + 2^k c_1}{\gamma_k^{k+1}}$.

Lemma E.1. *Let $H(m) = \alpha_k \gamma_k^m - \beta_k$ where γ_k is largest real root of $x^{k+2} - 2x^{k+1} + 1 = 0$, $\beta_k = \frac{(k+1)c_2|V|^2}{k}$ and $\alpha_k = \frac{(2^k k+1)\beta_k + 2^k c_1}{\gamma_k^{k+1}}$. If $T(m)$ satisfies Equation (9), then $T(m) \leq H(m)$ for $m \geq 0$.*

Proof. We first show $T(m) \leq 2^{m-1}k\beta_k + 2^{m-1}c_1$ by induction on m.

Basis: for $m = 1, T(1) \leq (k+1)c_2|V|^2 + T(0) \leq k\beta_k + c_1$.

Induction: We assume that $T(i) \leq 2^{i-1}k\beta_k + 2^{i-1}c_1$ for all $i = 1, \ldots, m-1$ and prove that it also holds for $i = m$. By Equation (9) we have:

$$\begin{aligned} T(m) &\leq (k+1)c_2|V|^2 + T(m-1) + \cdots + T(0) \\ &= (k+1)c_2|V|^2 + c_1 + \sum_{i=1}^{m-1} T(i) \\ &\leq (k+1)c_2|V|^2 + c_1 + \sum_{i=1}^{m-1} (2^{i-1}k\beta_k + 2^{i-1}c_1) \quad (10) \\ &= k\beta_k(1 + \sum_{i=1}^{m-1} 2^{i-1}) + c_1(\sum_{i=1}^{m-1} 2^{i-1} + 1) \\ &= 2^{m-1}k\beta_k + 2^{m-1}c_1 \end{aligned}$$

Then, we prove that $T(m) \leq H(m)$ for all $m \geq 0$ again by induction on m.

Basis: for $m = 1, T(1) \leq (k+1)c_2|V|^2 + T(0) \leq k\beta_k + c_1$, and $H(1) = \alpha_k\gamma_k - \beta_k = \frac{(2^k k+1)\beta_k + 2^k c_1}{\gamma_k^k} \geq k\beta_k + c_1$ (since $\gamma_k < 2$ and hence $\gamma_k^k < 2^k$).

Induction: We assume that $T(m) \leq H(m)$ for $m = 1, \ldots, i$ and prove that it also holds for $m = i+1$. First, let us factorize $x^{k+1} - 2x^{k+1} + 1$ as $(x-1)(x^{k+1} - x^k - \cdots - 1)$. Denote $F_k(x) = (x^{k+1} - x^k - \cdots - 1)$. We have $F_k(1) < 0$ and $F_k(2) > 0$, indicating that $F_k(x) = 0$ has a root between $(1,2)$. Combining the fact that γ_k is also the root of $F_k(x) = 0$, thus $\gamma_k^k + \cdots + 1 = \gamma_k^{k+1}$. Multiply both side with γ_k^{i-k}, we have $\gamma_k^i + \cdots + \gamma_k^{i-k} = \gamma_k^{i+1}$. Then the following inequalities hold.

$$\begin{aligned} T(i+1) &\leq T(i) + \cdots + T(i-k) + (k+1)c_2|V|^2 \\ &\leq \alpha_k(\gamma_k^i + \cdots + \gamma_k^{i-k}) - (k+1)\beta_k + k\beta_k \\ &\leq \alpha_k(\gamma_k^i + \cdots + \gamma_k^{i-k}) - \beta_k \quad (11) \\ &= \alpha_k\gamma_k^{i+1} - \beta_k \\ &= H(i+1) \end{aligned}$$

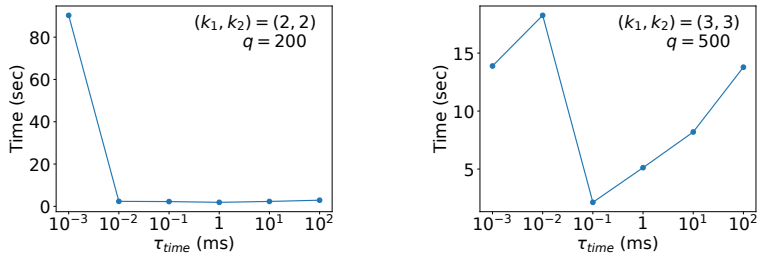
□

Finally, as we call BK2(.) with $m = |V|$ and k is a constant, we get the worst-case running time of our algorithm as given in Theorem 6.1.

F Incremental Set and Degree Maintenance

Instead of creating new sets P', C' and X' as the inputs into each recursive call BK2($G, k_1, k_2, q, P', C', X'$) (see Lines 17, 27 and 30), we propose to reuse the same containers for P, C and X 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 three global containers for P, C and X, respectively, using the data structure shown in Figure 10. Specifically, two arrays with capacity $|V|$ are maintained for each set S: (1) *list*[] which keeps the list of S's elements for scanning and adding new elements; (2) *pos*[] with 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 $O(1)$



(a) uk-2005 (b) webbase-2001

Figure 11: The Running Time of Large Datasets When τ_{time} is Varied.

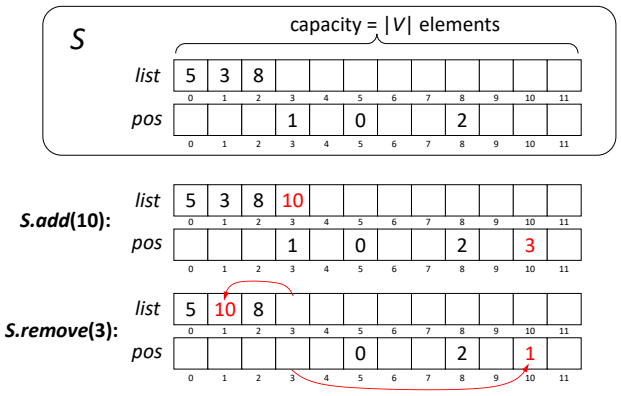


Figure 10: Data Structure for Set Maintenance

time. To illustrate using Figure 10, $S.add(10)$ appends 10 to $list[]$ at position 3, and records $pos[10] = 3$; while $S.remove(3)$ first finds the element position in $list[]$ as $pos[3] = 1$, and then erases it in $list[]$ by moving the last element 10 to $list[1]$, and moving 1 from $pos[3]$ to $pos[10]$. In this way, we can incrementally populate $\langle P, C, X\rangle$ for calling BK2(.) with minimal cost.

For example, for the else-branch in Lines 27–30, we can first conduct $C.remove(v_p)$ and $X.add(v_p)$ before calling BK2(.) at Line 27, and then recover X for use by Line 30 by conducting $X.remove(v_p)$. Line 28 (resp. Line 29) then removes from C (resp. X) those elements v such that $P \cup \{v_p, v\}$ is a not (k_1, k_2)-plex, and let us denote the set of removed elements by C'' (resp. X''). Then, we can conduct $P.add(v_p)$ and call BK(.) with (P, C, X) in Line 30, after which we recover P, C and X for backtracking by conducting $P.remove(v_p)$, $C.add(C'')$ and $X.add(X'')$. Note that C'' and X'' are kept in global buffers whose spaces are initialized at the beginning, and reused throughout the recursive execution.

Finally, we also incrementally maintain (1) $d^+_{P\cup C}(.)$ and $d^-_{P\cup C}(.)$ which are needed in Lines 22, 28 and 29; and (2) $d^+_P(.)$ and $d^-_P(.)$ which are needed in Lines 8, 28 and 29, as we have explained at the end of Section 3. For example, whenever we remove a vertex v from C, we need to decrement $d^-_{P\cup C}(u)$ for all $u \in N^+_{P\cup C}(v)$ and $d^+_{P\cup C}(u)$ for all $u \in N^-_{P\cup C}(v)$.

G Complete Results in Parallel Executions

We implemented the parallel version of Ours based on the description in Section 7. Table 5 reports the (k_1, k_2)-plex enumeration time of parallel Ours on all our datasets in Table 1 when running with 2, 4, 8, 16 and 32 threads, respectively, where we use the default timeout threshold $\tau_{time} = 0.1$ ms is adopted, which is tested to work consistently well on various datasets.

As Table 5 shows, our parallel algorithm is efficient and scales up well with the number of CPU cores on all the datasets.

H Load Balancing

We also conducted experiments to study the impact of τ_{time}. Recall from Section 7 that if a task runs for a period of more than τ_{time}, it will decompose itself so that the decomposed tasks can be processed in parallel. Figure 11 shows the impact of τ_{time} on the job running time, when using 32 threads and varying τ_{time} as 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100 for two representative settings on our large graphs uk-2005 and webbase-2001. We can see that our default setting of $\tau_{time} = 0.1$ ms does work the best in general. Also, Figure 11(a) shows that if we set τ_{time} too small, too many tasks will be created so the task creation time dominates and the running time can be significantly increased, and Figure 11(b) shows that if we set τ_{time} too large, load balancing suffers so the running time can be many times longer.

Table 5: The Running Time of Mining Large Maximal (k_1, k_2)-Plexes in Parallel Execution

Dataset	k_1	k_2	q	#(k_1, k_2)-plexes	Serial	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
bitcoin	2	5	10	144.00	0.06	0.04	0.02	0.01	0.01	0.02
			12	0	0.01	0.00	0.00	0.00	0.00	0.00
	3	5	10	24,261	8.35	4.78	2.46	1.29	0.66	0.36
			12	286	1.05	0.62	0.33	0.17	0.09	0.05
wiki-vote	3	5	10	474	1.85	1.08	0.55	0.29	0.15	0.08
			12	0	0.08	0.05	0.02	0.01	0.01	0.01
	4	5	10	30,222	349.32	197.94	102.45	52.57	26.57	16.44
			12	94	16.63	9.76	4.99	2.61	1.32	0.91
mathoverflow	2	2	10	523,633	28.55	18.48	9.48	4.93	2.50	1.34
			12	150,888	13.35	9.01	4.56	2.38	1.22	0.69
	2	3	10	1,762,917	63.16	47.81	24.41	12.71	6.45	3.52
			12	602,862	32.44	24.92	12.61	6.63	3.36	1.86
as-caida	2	3	10	23,314	7.99	0.34	0.19	0.10	0.08	0.15
			15	185	0.93	0.02	0.01	0.01	0.01	0.02
	3	4	15	17,303	14.87	0.63	0.33	0.18	0.10	0.10
			20	0	0.19	0.01	0.00	0.00	0.00	0.00
epinions	2	3	10	773,095	23.85	18.34	9.48	4.94	2.56	1.41
			12	252,511	10.33	8.06	4.18	2.23	1.19	0.80
	3	4	12	24,599,029	2,053.04	1,174.05	605.69	314.25	158.78	85.62
			15	2,382,084	381.99	229.27	118.32	61.53	31.62	18.13
email-euall	2	3	10	17,438	1.69	1.26	0.65	0.34	0.18	0.19
			12	1442	0.58	0.39	0.21	0.11	0.07	0.12
	3	4	12	196,878	80.58	46.44	23.49	12.18	6.14	3.38
			15	1351	9.19	5.20	2.68	1.40	0.72	0.44
amazon0505	3	5	10	4,696	0.27	0.08	0.04	0.02	0.02	0.01
			12	24	0.14	0.01	0.00	0.00	0.00	0.00
	5	3	10	1,532	0.20	0.04	0.02	0.01	0.01	0.00
			12	1	0.13	0.00	0.00	0.00	0.00	0.00
web-google	2	3	15	997	0.29	0.05	0.02	0.01	0.01	0.00
			20	57	0.21	0.01	0.01	0.00	0.00	0.00
	3	4	15	1,993	0.32	0.06	0.03	0.02	0.01	0.01
			20	61	0.22	0.02	0.01	0.00	0.00	0.00
soc-pokec	2	3	15	17,980	2.11	0.92	0.49	0.25	0.21	0.39
			20	17	0.95	0.03	0.01	0.01	0.01	0.02
	3	3	15	508,884	54.91	32.24	16.52	8.65	4.41	2.41
			20	648	1.66	0.49	0.25	0.13	0.07	0.06
wiki-talk	2	2	15	22,219	287.28	173.50	87.73	44.75	22.50	11.48
			20	0	69.70	38.84	19.89	10.27	5.17	2.67
	2	3	15	165,156	525.13	376.12	191.71	97.82	48.99	25.01
			20	0	129.02	77.30	39.23	20.22	10.16	5.21
arabic-2005	2	2	1000	106,895	3,906.70	1,897.35	1,017.31	527.40	269.04	140.58
	3	3	3000	3,500	111.60	60.20	29.56	15.10	8.69	4.64
uk-2005	2	2	200	117,255	301.09	162.70	84.11	43.34	21.82	11.38
	3	3	400	66,639	168.98	91.51	46.34	23.95	12.11	6.34
it-2004	2	2	1000	15,087	1,170.05	621.80	321.12	166.53	85.40	52.04
	3	3	3000	1,034	77.90	41.35	21.72	10.21	6.40	3.94
webbase-2001	2	2	300	327,882	646.41	320.84	164.46	90.02	53.56	30.75
	3	3	500	88,777	184.44	95.16	52.07	30	18.15	8.75
clue-web	2	2	10	1,215,977	26.05	14.15	7.66	3.82	1.75	1.69
	3	3	20	95,383	52.92	31.76	16.46	8.56	4.42	2.37