```
Algorithm: AlgebraicMaxCliques_Canonical
  Input: Graph G, parameters K, max_iter, tol, stable_iter_threshold,
         rel_thresh
  Output: Set F of maximal cliques (tuples of node ids)
core_num ← CoreNumber(G) ;                                  // O(n + m)
order_pos ← DegeneracyOrderPositions(G) ;                   // O(n + m)
Initialize empty LRU cache: SubgraphCache;
F ← ∅;
foreach v ∈ V do
    N_v ← list(neighbors of v);
    if N_v is empty then
    end
    neighs ← top_k_by_core(N_v, core_num, K) ;              // at most K
     neighbors
    seeds ← [u ∈ neighs | order_pos[u] ≥ order_pos[v]] ;    // canonical
     seeding
    if seeds is empty then
        seeds ← [None] ;                 // still try seed with v alone
    end
    S ← {v} ∪ neighs ;                        // nodes of induced subgraph
    key ← tuple(sorted(S));
    if key ∈ SubgraphCache then
        (A_csr, nodes_list) ← SubgraphCache[key];
    else
        A_csr ← CSR adjacency of G induced on nodes_list = list(key);
        store (A_csr, nodes_list) in SubgraphCache;
    end
    p ← |seeds|; n_s ← |nodes_list|;
    Build X_0 ∈ ℝ^(n_s×p): foreach column j corresponding to seed u do
        if u is None then
            X_0[idx(v), j] = 1;
        else
            X_0[idx(v), j] = 0.5; X_0[idx(u), j] = 0.5;
        end
        normalize column j to sum 1;
    end
    (X_final, iters) ←
     BatchedReplicatorSparseAdaptive(A_csr, X_0, max_iter, tol, stable_iter_threshold);

    for j ← 1 to p do
        if max(X_final[:, j]) = 0 then
        end
        support_idx ← {i | X_final[i, j] ≥ rel_thresh · max(X_final[:, j])};
        candidate ← {nodes_list[i] for i ∈ support_idx};
        if not IsClique(G, candidate) then
        end
        clique ← ExpandToMaximal(G, candidate) ;    // greedily add
         nodes adjacent to all in candidate
        min_node ← argmin_{u∈clique} order_pos[u];
        if min_node ≠ v then
        end
        F.add(tuple(sorted(clique)));
    end
end
return F;
```

Algorithm: BatchedReplicatorSparseAdaptive

Input: Sparse matrix A_{csr} $(s \times s)$, initial matrix X_0 $(s \times p)$, *max_iter*, *tol*, *stable_iter_threshold*

Output: Matrix X, iteration count *iters*

$X \leftarrow X_0$;
prev $\leftarrow$ None;
prev_supports $\leftarrow$ [None] $\times\ p$;
stable_counts $\leftarrow$ zeros(p);
for $t \leftarrow 1$ **to** *max_iter* **do**
    $AX \leftarrow A_{\text{csr}} \cdot X$; `// sparse-dense matmul` $\rightarrow$ `dense` $(s \times p)$
    $X \leftarrow X \odot AX$; `// element-wise multiply` $(s \times p)$
    col_sums $\leftarrow \sum(X, \text{axis} = 0)$;
    set zero columns to zero, divide nonzero columns: $X[:, nz] \leftarrow X[:, nz]/col\_sums[nz]$;
    if *prev* $\neq$ None ***and*** *all columns j satisfy* $\|X[:, j] - prev[:, j]\|_1 < tol$ **then**
        return (X, t);
    end
    compute $supports_j$ = tuple(indices where $X[i, j] \geq \epsilon \cdot \max(X[:, j])$) for each j;
    for $j \leftarrow 1$ **to** p **do**
        if $supports_j = prev\_supports[j]$ **then**
            $stable\_counts[j] \leftarrow stable\_counts[j] + 1$;
        end
        else
            $stable\_counts[j] \leftarrow 0$;
        end
    end
    if *all* $stable\_counts[j] \geq stable\_iter\_threshold$ **then**
        return (X, t);
    end
    prev $\leftarrow X$; *prev_supports* $\leftarrow$ *supports*;
end
return (X, max_iter);