Algorithm 4 Ullmann's Subgraph Matching Algorithm

1: **Input:** query graph $G_q = (V_q, E_q)$, data graph $G = (V, E)$
2: generate matching order $\pi = [u_1, u_2, \ldots, u_{|V_q|}]$ $(u_i \in V_q)$
3: ENUMERATE$(\emptyset, 1, \pi)$
4: **procedure** ENUMERATE(S, i, π)
5: $C_S(u_i) \leftarrow$ viable vertex candidates in G to match u_i
6: **for all** $v \in C_S(u_i)$ **do**
7: append S with (u_i, v)
8: **if** $|S| = k$ **then output** S
9: **else** ENUMERATE$(S, i+1, \pi)$
10: pop (u_i, v) from S

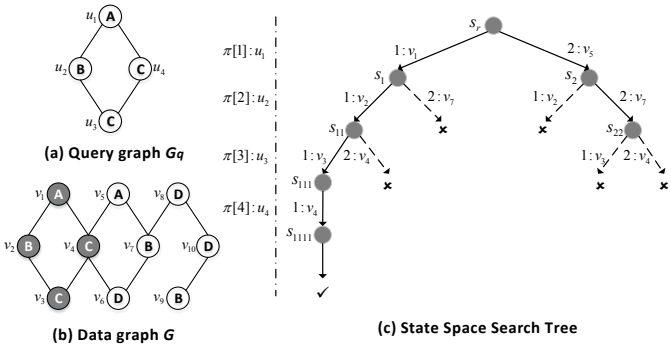

Fig. 9. Illustration of Subgraph Matching

APPENDIX A
SUBGRAPH MATCHING

Subgraph matching (SM) is a representative SF problem, which enumerates all subgraphs of a data graph G that are isomorphic to a query graph G_q. Many SM algorithms [38], [55] have been proposed based on Ullmann's algorithm [41], which orders the query vertices and recursively extends the partial mapping by iterating over candidate data vertices for the next query vertex and pruning infeasible extensions.

We now briefly describe the basic Ullmann's algorithm [41] as shown in Algorithm 4. The algorithm incrementally matches data vertices to query vertices $u_1, \ldots, u_{|V_q|}$, maintaining the partial mapping in S. This search is conducted over a state-space tree as illustrated in Figure 9(c). For instance, the path $S = [(u_1, v_5), (u_2, v_7), (u_3, v_4)]$ fails due to the absence of edge (v_7, v_4) corresponding to query edge (u_2, u_3). Various enhancements have been proposed by various SM algorithms [38], such as optimizing the matching order (Line 2) and deriving tighter candidate sets $C_S(u_i)$ (Line 5).

Note that the state-space tree is also a subgraph enumeration tree over G: a node at Level i corresponds to a subgraph in G matching the query vertex prefix $[u_1, \ldots, u_i]$. Since the programming model of G-thinkerCG is designed based on the subgraph enumeration tree and subgraph-task abstractions, it can implement the CPU-GPU co-processing programs for all SM algorithms by properly adapting their serial algorithm counterparts based on the interface of G-thinkerCG.

APPENDIX B
GPU ALGORITHMS FOR SF PROBLEMS

Except for G^2-AIMD, existing GPU works mainly target a specific SF problem for acceleration, using either BFS- or DFS-based subgraph extension. BFS-based solutions compute one chunk at a time, while DFS-based solutions assign independent subgraph enumeration subtrees to different computing units (thread blocks or warps), each of which maintains its own stack in the global memory for backtracking.

For the application of MCE, [24] is a DFS-based approach that assigns different subtrees to thread blocks, while within a subtree, warps parallelize operations like filtering adjacency lists. The work finds that their GPU implementations only achieved about 1.4–2.25× the performance of a single CPU core. GBK [45] implements the BK algorithm using BFS-based approach but it does not address the issue of subgraph number explosion. Like [24], MCE-GPU [4] also employs a DFS-based approach to implement BK but it implements several effective performance optimizations: (1) binary encoding that represents adjacency lists as bit vectors to allow fast intersections; (2) sub-warp partitioning that divides thread blocks into groups smaller than a warp to resolve the warp over-provisioning issue for small adjacency lists; (3) space-efficient data representations including partial induced subgraphs and a compact representation of the exclusion set X. This tailor-made solution to MCE does achieve remarkable performance, but we find that it runs OOM on many graphs, because technique (3) needs to materialize and maintain an adjacency matrix for each top-level subtree – even though the space cost has been optimized to $O(\Delta + d^2)$ where Δ is the maximum degree and d is the graph degeneracy – which is costly in space. Moreover, all three techniques are orthogonal to the design of G-thinkerCG and can be integrated into our MCE application with some implementation efforts, though we do not implement them currently which makes our MCE program more scalable (since we read adjacency lists from a global graph rather than materialize many adjacency matrices).

For the application of SM, Ullmann's algorithm (recall Algorithm 4) is used to extend partially matched data graph instances into fully matched ones, where each subgraph instance is extended by a warp. Among them, GSI [59] and cuTS [47] are BFS-based and for in-device processing, while PBE [21], VSGM [25] and SGSI [58] explore methods to partition a large input graph so that only one partition needs to be loaded to a GPU for processing at each time. This out-of-core processing technique is orthogonal to the design of our G-thinkerCG, and we can directly integrate VSGM's subgraph k-means-based binning approach to construct graph partitions if the input graph is too large to fit into the device memory of a GPU. More recently, DFS-based approaches emerge such as STMatch [46] and EGSM [39], which assign independent subtrees to warps for backtracking search, each maintaining its own stack in GPU global memory, with load balancing via task splitting and work stealing. While STMatch reports a good performance, EGSM shows that its DFS execution is nearly two orders of

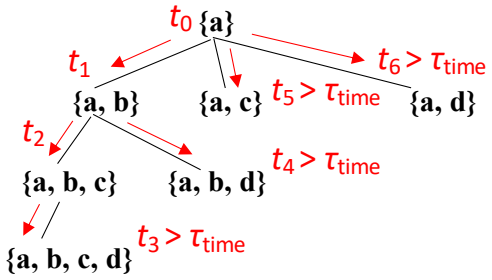

Fig. 10. Timeout Example

magnitude slower than its BFS execution as well as a hybrid BFS-DFS matching strategy, which divides a query graph into groups and extends a group of vertices at a time (BFS), while within each group, subgraph matching is performed in the DFS order. Note that the hybrid BFS-DFS strategy of EGSM is only slightly faster than their BFS strategy, and it is based on query graph partitioning and is quite different from our novel BFS-DFS strategy to be presented in Section III. Last but not the least, GraphPi [36], GraphZero [31] and AutoMine [32] further propose a compilation-based approach to generate efficient subgraph enumeration code with a favorable vertex matching order.

Appendix C
The Algorithm of `atomicSubNonNegative(.)`

To pop a subgraph from the stack $\mathcal{H}$, each thread of a warp calls the `pop_offsets(sglen)` operation shown in Figure 4-❺, which properly decrements `otail` in Line 2, and returns the trackers [`offsets[ot-2], ohead[ot-1]`) for all threads of the warp to read the obtained subgraph from $\mathcal{H}$.`vertices`. Note that since we cannot decrement `otail` to be smaller than 0, we cannot use `atomicSub(.)` since when multiple warps subtract 2 from `otail` consecutively, `otail` can become negative. We solve this issue by calling `atomicSubNonNegative(.)` as shown in Figure 4-❻.

Specifically, Line 1 reads the value of `otail` into variable `old`, which is then read into `assumed` in Line 3. If this value is already 0 (Line 4), `otail` cannot be further decremented so Line 5 returns 0 directly as the original value of `otail`. Otherwise, Line 6 atomically checks if `otail`'s previously retrieved value (i.e., `old`) still equals `assumed`: (1) if so, no other warp has changed `otail` since Line 3, so the decremented value (`assumed` − `dec`) is written to `otail`; (2) while otherwise, `otail` has been updated by another warp, so we cannot update `otail` due to the atomicity requirement of subtraction, and `old` gets the latest value of `otail` in Line 6 and tries again until an atomic subtraction is successfully done (Line 7), after which the old value of `otail` before subtraction is returned in Line 8.

Appendix D
Implementation of Timeout Mechanism

To implement the timeout mechanism for Algorithm 1, we let a task note down its creation time t_0, and in `compute`(c) that runs Algorithm 1, we change Line 5 to check if current time $t_{cur} - t_0 > \tau_{time}$; if so, instead of recursively process the subgraph enumeration subtree, the worker creates a task $t = (R \cup \{v\}, P \cup N(v), X \cup N(v))$ and calls `addTask`(t) to add t to stack $\mathcal{S}$. Figure 10 shows this process for the task with $R = \{a\}$, where timeout occurs before processing $\{a, b, c, d\}$ so it becomes a new task along with $\{a, b, d\}$, $\{a, c\}$ and $\{a, d\}$ during backtracking that are pushed to $\mathcal{S}$.