Задача поиска изоморфного подграфа с использованием GPU

Обзор существующих решений

Выполнила: Гарифуллина Гузель, 371 группа

СПБГУ
2017 г.

Постановка задачи

- Исследовать алгоритмы поиска изоморфного подграфа (subgraph matching), использующие GPU
- Реализовать 2 алгоритма

Проблема поиска изоморфного подграфа

- $G = (V_G, E_G, L_G, l_G)$ - data graph
- $l : V \rightarrow L$ - labeling function
- $q = (V_q , E_q , L_q , l_q)$ - query graph
- Ищем подграф $g = (V_g , E_g)$ графа G: существует биекция $f : V_q \rightarrow V_g$, такая что
  $\forall v \in V_q \quad l_q (v) = l_G (f (v))$
  $\forall e = (u, v) \in E_q , (f (u), f (v)) \in E_g$

Актуальность

- Программный анализ
- Поиск химического соединения
- Анализ социальных сетей

Термины

- $v \in V_G$ называется ***кандидатом*** $u \in V_q$, если $l(u) = l(v)$ и $deg(u) \leq deg(v)$

Методы решения

- Индексация (indexing)
- Обход графа (graph traversal)

- С помощью возвратов (backtracking)
- С помощью комбинирования кандидатов (filtering-and-joining strategy)

Основные эвристические подходы

- Порядок обхода вершин
- Удаление неперспективных кандидатов как можно раньше

Efficient Subgraph Matching Using GPUs

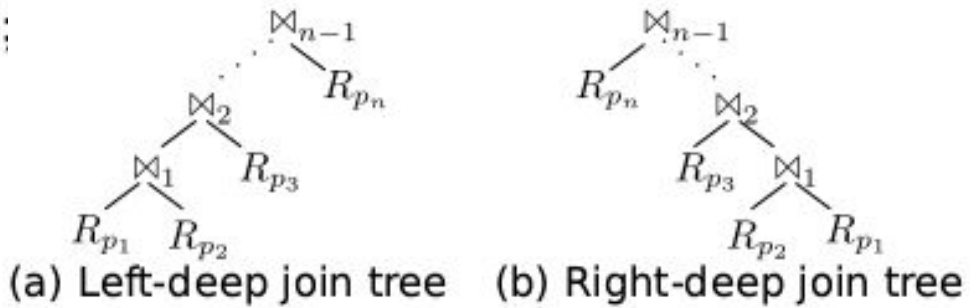
1. Использует STwig алгоритм
2. Только операция join на GPU

STwig алгоритм

1. Декомпозиция запроса
   Минимизировать число STwigs и найти порядок STwig
   <o_1 , o_2 , ... , o_n>, такой что: $\forall$ i >= 2, $\exists$ j < i корень
   STwig o_i является листом o_j
2. Сопоставление графа с STwig (matching STwigs)
   $R_N = \{\{n\} \times T_{l_1} \times T_{l_2} \times \cdots \times T_{l_{|L_1|}}\}$;
3. Комбинирование решений (join)

Особенности реализации join

$R_N = \{\{n\} \times T_{l_1} \times T_{l_2} \times \cdots \times T_{l_{|L_1|}}\}$;



(a) Left-deep join tree (b) Right-deep join tree

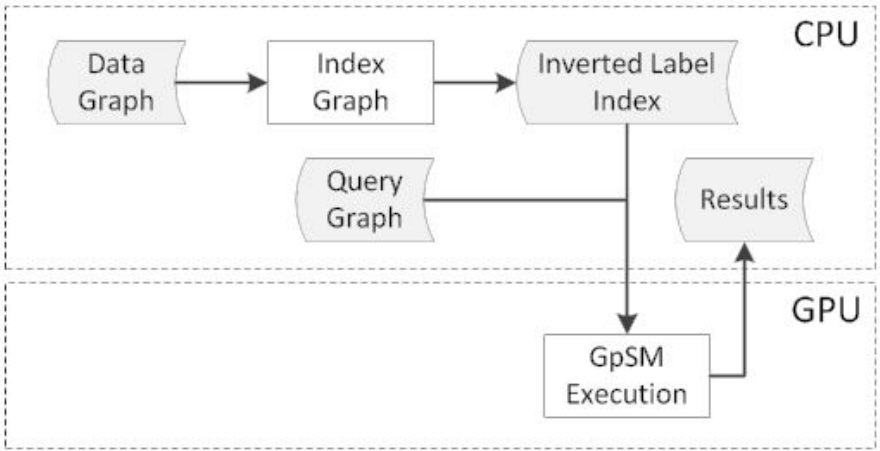
Fast Subgraph Matching on Large Graphs using Graphics Processors (GpSM)

1. Инициализация порядка обхода запроса
2. Фаза фильтрации (filtering phase)
   a. Инициализация кандидатов (initializing candidate vertices)
   b. Улучшение кандидатов (refining candidate vertices)
3. Фаза присоединения (joining phase)

   a. Поиск всех подходящих ребер кандидатов

   b. Комбинирование решений

GpSM для очень больших графов



Towards GPU-Based Common-Sense Reasoning: Using Fast Subgraph Matching (GpSense)

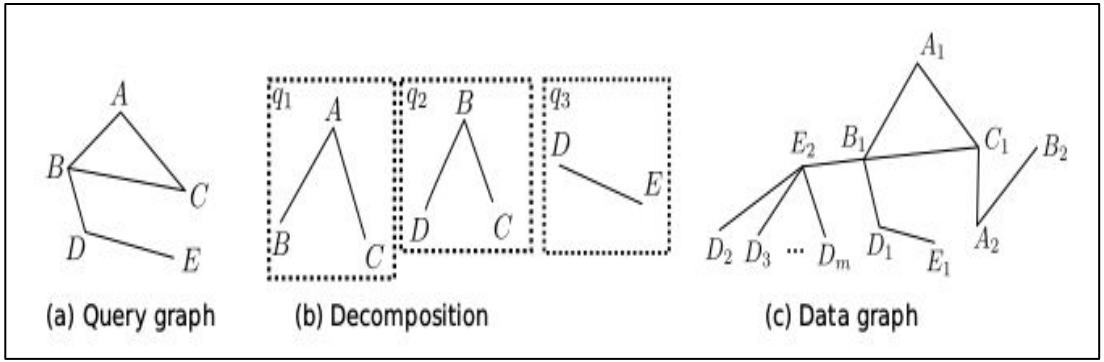
1. Стратегия фильтрации и комбинирования кандидатов (filtering-and-joining strategy)
2. Common-Sense Knowledge графы
   a. ⟨⟨ концепция-отношение-концепция ⟩⟩
   b. Направленный граф и имеет метки на ребрах
   c. Граф-запрос помимо вершин с метками имеет вершины с неизвестными метками
3. Многоуровневое сжатие графа

GPU acceleration of subgraph isomorphism search in large scale graph (GPUSI)

1. Инициализация (запрос-дерево)
2. Исследование региона (region exploration)
3. Определение порядка обхода региона (determine matching order)
4. Поиск изоморфного подграфа (subgraph matching)

Дополнительные материалы

1) Efficient Subgraph Matching Using GPUs



Algorithm 1. Match the First STwig

input : Data graph G, STwig $o_1 = (r_1, L_1)$ where r_1 is the root node and L_1 is the leave label set

output: Result set R

Fill the set S_1 with all nodes with r_1's label;

for *each* n *in* S_1 **do**

  for *each* l_i *in* L_1 **do**

    $T_{l_i} \leftarrow \{m | m \in n.\text{neighbors and } m.\text{label}=l_i\}$;

  $R = R \cup \{\{n\} \times T_{l_1} \times T_{l_2} \times \cdots \times T_{l_{|L_1|}}\}$;

1) Efficient Subgraph Matching Using GPUs

Algorithm 2. Join algorithm for each thread

Input: Probe relation R_{p_1} and hash tables $H_2, H_3, \cdots, H_n$

```
1  begin
       // pos indicates this thread is processing subjoin ⋈_pos
2      pos ← 1;
3      itArray[1] ← none;
4      while true do
5          while itArray[pos] = none and pos ≠ 1 do
6              pos ← pos − 1;
7          while itArray[1] = none do
8              i ← GetNewRecordsOrQuit();
9              set imResult based on R_p1[i];
10             itArray[1] ← H_2.match(imResult);
11         set imResult based on pos and itArray[pos].record();
12         itArray[pos] ← itArray[pos].next();
13         if pos = n − 1 then
14             output imResult as a final result;
15         else
16             itArray[pos + 1] ← H_pos+2.match(imResult);
17             pos ← pos + 1;
```

2) Fast Subgraph Matching on Large Graphs using Graphics Processors (GpSM)

Algorithm 1. GpSM(q,g)

Input: query graph q, data graph g
Output: all matches of q in g

```
1 C := InitializeCandidateVertices(q,g);
2 C := RefineCandidateVertices(q,g,C);
3 E := FindCandidateEdges(q,g,C);
4 M := JoinCandidateEdges(q,g,E);
5 return M
```

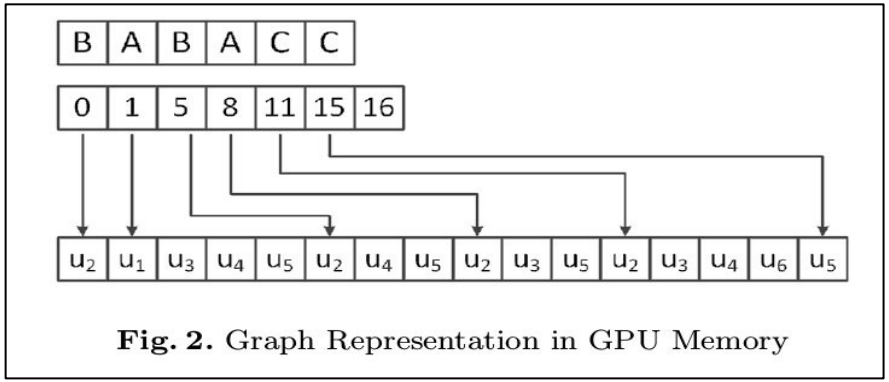


Fig. 2. Graph Representation in GPU Memory

2) Fast Subgraph Matching on Large Graphs using Graphics Processors (GpSM)

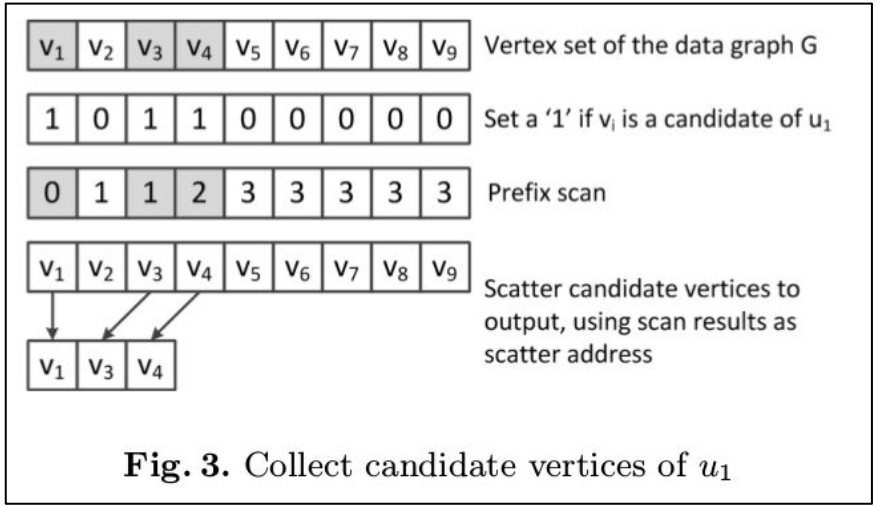


Fig. 3. Collect candidate vertices of u_1

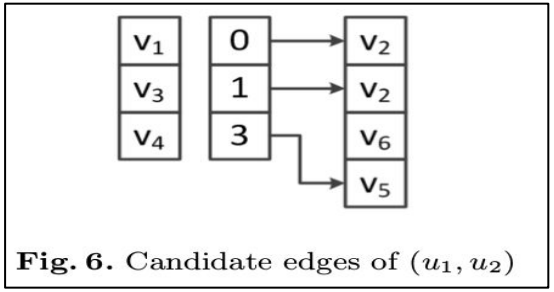
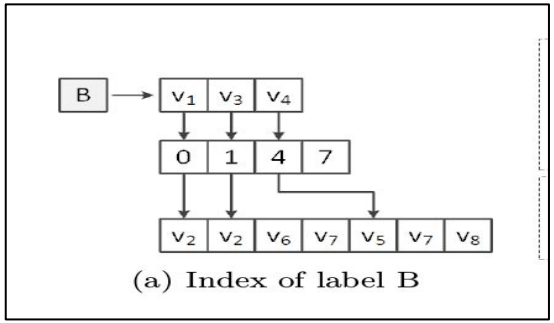
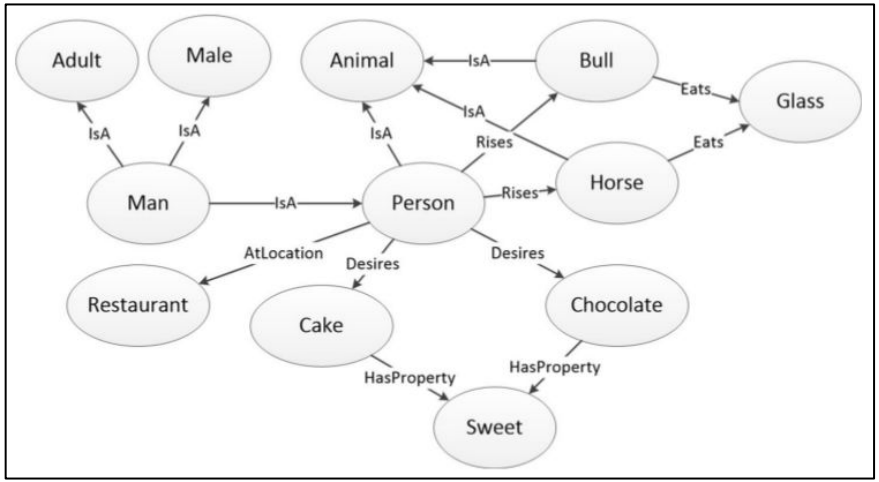


Fig. 6. Candidate edges of (u_1, u_2)



(a) Index of label B

3) Towards GPU-Based Common-Sense Reasoning: Using Fast Subgraph Matching

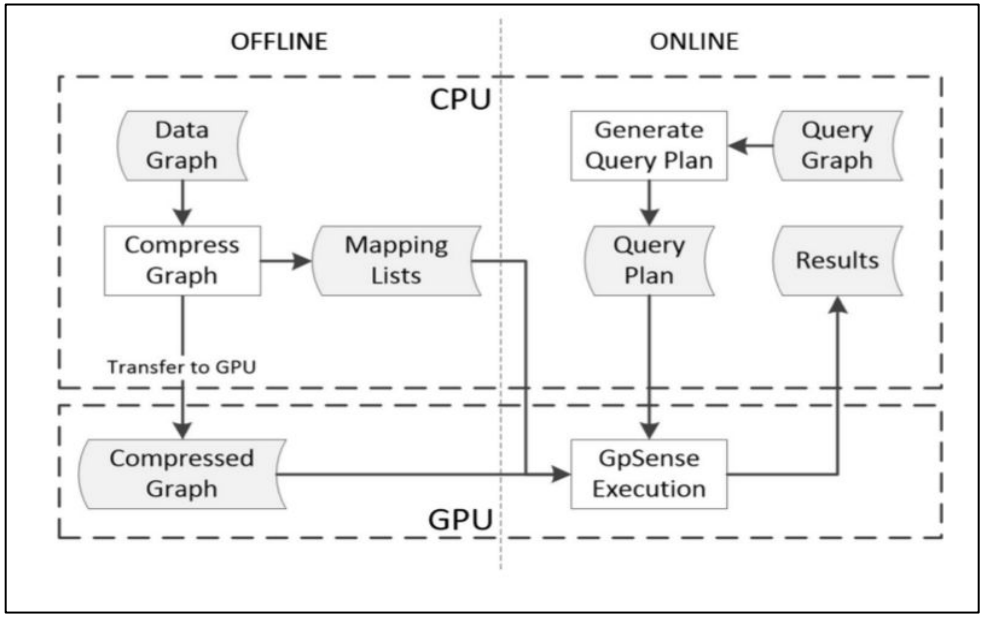


Algorithm 1: GPUSubgraphMatching (q(V, E, L), g(V', E', L'))

Input: query graph q, data graph g
Output: all matches of q in g

```
1  P := generate_query_plan(q, g);
2  forall the node u ∈ P do
3      if u is not filtered then
4          c_set(u) := identify_node_candidates(u, g);
5      c_array(u) := collect_edge_candidates(c_set(u));
6      c_set := filter_neighbor_candidates(c_array(u), q, g);
7  refine_node_candidates(c_set, q, g);
8  forall the edge e (u,v) ∈ E do
9      EC(e) := collect_edge_candidates(e, c_set, q, g);
10 M := combine_edge_candidates(EC, q, g);
11 return M
```

3) Towards GPU-Based Common-Sense Reasoning: Using Fast Subgraph Matching



4) GPU acceleration of subgraph isomorphism search in large scale graph

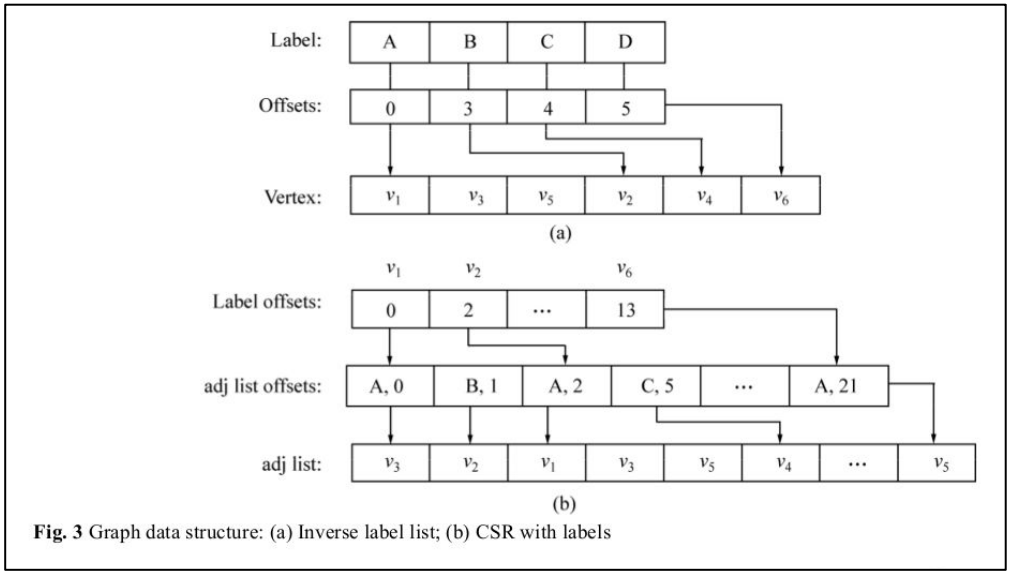


Fig. 3 Graph data structure: (a) Inverse label list; (b) CSR with labels

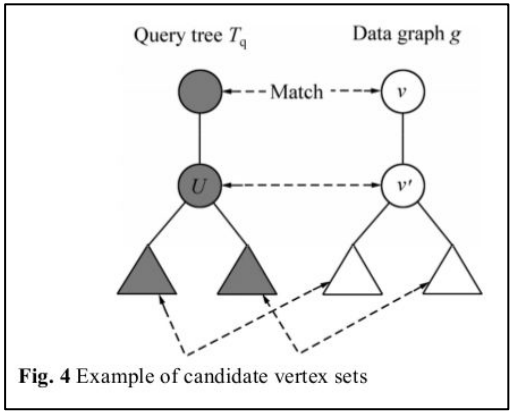


Fig. 4 Example of candidate vertex sets

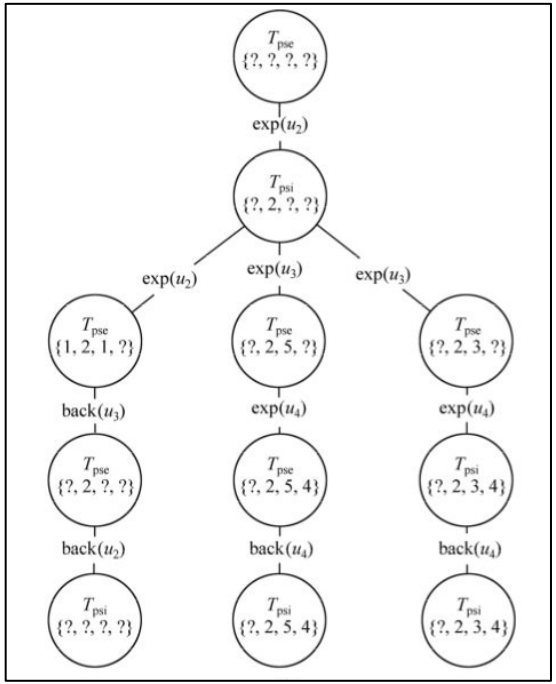
Термины

- Partial subgraph embedding ***(Gpse)*** - это структура данных, состоящая из $|V_q|$ пар $<v_q, v_g>$ ($v_g = map(v_q)$), где вершине v_q сопоставляется v_g
- Набор вершин-кандидатов (candidate vertex set) или CVS(u, v) - набор вершин, содержащий вершины графа G, которые соответствуют u и являются дочерними вершинами v в дереве построенном DFS обходом и каждое поддерево u соответствует соответствующему поддереву v в DFS дереве

4) GPU acceleration of subgraph isomorphism search in large scale graph

Algorithm 2: CPU region exploration

```
1  Explore Region (u, V_m, R){
2  for each unvisited data vertex v in V_adj
3          if deg(u)>deg(v) or NLF(u)=FAIL
4                  then
5                          continue;
6          mark v as visited;
8          append v into CR(u, v_p), where v_p is the
           parent of v in DFS tree;
10         sort child u_c in ascending order of
           |dj(v, L(u_c))|
11         for each child vertex u_c of u then
12                 if explore(u_c, adj(v, L(u_c)), R)=FAIL
                   then
13                   for each visited child query vertex u_cc
                     of u except u_c do
14                           ClearRegion(u_cc, v, R);
15                   remove v from R(u, v_p);
16                   return FAIL;
17           mark v as unvisited;
19 if |R(u, v_p)|=0 then
20         return FAIL;
21 return DONE;
22 }
```



4) GPU acceleration of subgraph isomorphism search in large scale graph

Algorithm 3: Task dispatch of parallel region exploration

```
1  DispatchExploration(u, CVS){
2  for each category CVS(u, c_at, −) of CVS(u, −) in
   parallel
3          if CVS(u, c_at, −) is empty then
4                  return FAIL;
5          for each child u_c of u do
6                  Expand(u_c, CVS(u, c_at, −), CVS);
7                  DispatchExploration(u_c, CVS)
8                  Backward(u_c, CVS(u, c_at, −), CVS);
9                  if CVSp(u, −) is empty then
10                         break;
11         if num of children of u>1 then
12                 ClearRegion(CVS(u, c_at, −));
13 return DONE;
14 }
```

```
15
16 Expand(u_c, CVS(u, c_at, −), R){
17         atomic_enqueue(gpu_emb_queue, u_c,
           CVS(u, c_at, −), R, EXPAND);
18         barrier();
19         if cpu_thread_id=0 then
20                 GPUProcessKernel(gpu_emb_queue);
21         barrier();
22 }
23
24 Backward(u_c, CVS(u, c_at, −), R){
25         atomic_enqueue(gpu_emb_queue, u_c,
           CVS(u, c_at, −), R, BACKWARD);
26         barrier();
27         if thread_id=0 then
28                 GPUProcessKernel(gpu_emb_queue);
29         barrier();
30 }
```

4) GPU acceleration of subgraph isomorphism search in large scale graph

Algorithm 4: GPU region exploration

```
1  GPUProcessKernel(gpu_emb_queue){
2  if laneId of a warp=0 then // distribute task to warps
3        {u_c, CVS(u, c_at, −), R, offset, operation}=
         atomic_warp_dequeue (gpu_emb_queue);
4  if operation = EXPAND then
5           WarpForward(offset);
6  else if operation=BACKWARD then
7           WarpBackward(offset, gpu_emb_queue);
8  else then
9           WarpClear(offset, gpu_emb_queue);
10 }
11
12 WarpExpand(offset, gpu_emb_queue){
13 WarpLoad(v[tid], CVS(u, c_at, −), offset);
14 if (v[tid]=visited){
15          mark the flag of v as 0;
16 }
17 if (deg(u)>deg(v[tid]) or NLF(u, v[tid])=FAIL){
18          mark the flag of v as 0;
19 }
20 while WarpAny(v[tid])=1 do // do if there is a v[tid]
   that is not processed)
21          // vie for control of warp
22          if v[tid] that is not processed then
23                winner=tid;
24          if winner=tid then
25                v_m=v[tid];
26                v[tid]=0;
27          // warp traverse the winner's R(u_c, v[tid])
28          WarpLoad(v_c[tid], adj(u_c, v_m));
29          sort children u_ccs of u_c in ascending order of
            |adj(v, L(u_c))|
30          c_at=the order;
31          WarpStore(CVS(u_c, c_at, v_m), v_c[tid]);
32 }
35 WarpLoad(v[tid], CVS(u, c_at, −), offset);
36 while WarpAny(v[tid])=1 do // do if there is a v[tid]
   that is not processed)
37          if v[tid] that is not processed then
38                winner=tid;
39          if winner=tid then
40                v_w1=v[tid];
41                v[tid]=0;
42          // warp traverse the winner's R(u_c, v[tid])
43          WarpLoad(v_c[tid], CVS(u_c, v_1));
44          If flag of v_c[tid]=matched;
45          WarpStore(CVS(u_c, c_at, v), v_c[tid]);
46          if WarpAll(flag of v_c[tid])=unmatched then
47                if tid=own then
48                      mark the flag of v[tid] as
                        unmatched;
49 }
```

4) GPU acceleration of subgraph isomorphism search in large scale graph

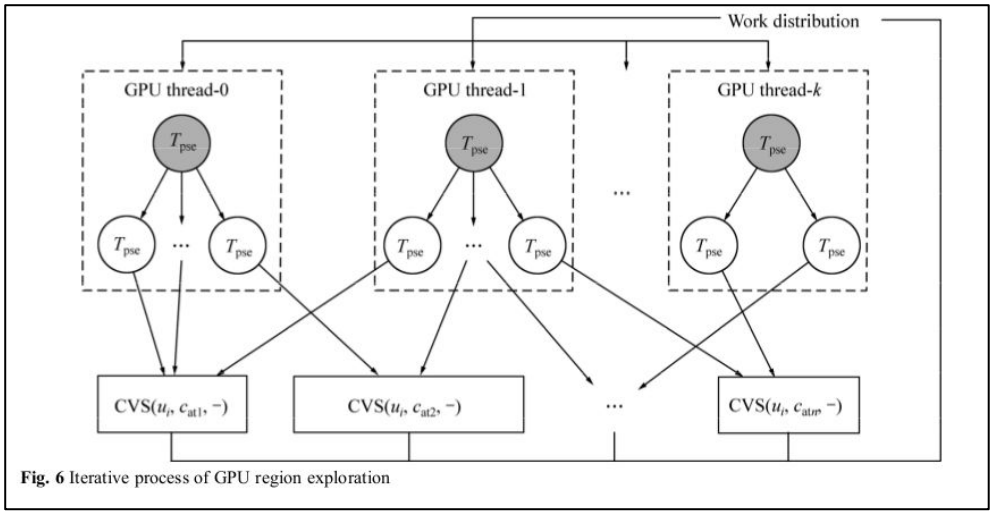


Fig. 6 Iterative process of GPU region exploration

4) GPU acceleration of subgraph isomorphism search in large scale graph

Algorithm 5: CPU graph matching algorithm

```
1  SubgraphMatching(){
2  for each u in query graph in the matching order obtained do
3          expandEmbedding(u, q, R, emList);
4          if outputEmlist is empty then
5                  return FAIL;
6  return emList;
7  }
8
9  expandEmbedding(u, q, R, emList){
10 for each partial subgraph embedding G_pse dequeued from InputEmbList
11         matched=true;
12         for each vertex v in R(u, v_p) do; //v_p is the parent of v
13           for each non-tree between u and the matched query vetex u_m
14                 if IsJoin(v_m, v)=false then; //the vertex v_m of G_psi matching with u_m
15                         matched=false;
16           if matched=true do
17                 new G_psi=generateEmbedding(G_psi, v);
18                 enqueue new G_psi into OutputEmList;
20 return outputEmlist;
21 }
```

Algorithm 6: GPU subgraph matching

```
1  WarpExpandEmbedding(){
2  if laneId of a warp=0 then // distribute task to warp
3          offset=atomic_warp_dequeue(InputEmbList);
4
5  WarpLoad(embedding[tid], offset, InputEmbList);
6  WarpLoad(v[tid], CVS(u, v_p in embedding)); // v_p is the parent of v
7
8  for each non-tree between u and the matched query vetex u_m do
9          WarpLoad(neighbor of v_m with the label L(u));
10         matched=false;
11         for each neighhbor n_vm of v_m do
12                 if n_vm=v[tid] then
13                         matched=true;
14         if matched=false then
15                 break;
16 if matched=true then
17         new G_psi=generateEmbedding(G_psi, v);
18         enqueue new G_psi into OutputEmbList;
19 }
```