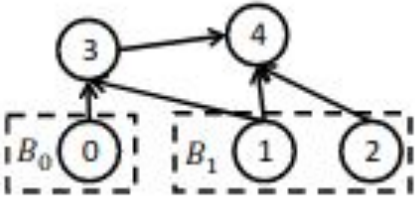
An Efficient Parallel Keyword Search Engine on Knowledge Graphs

Yueji Yang , Divykant Agrawal, H.V. Jagadish, Anthony K. H. Tung, Shuang Wu

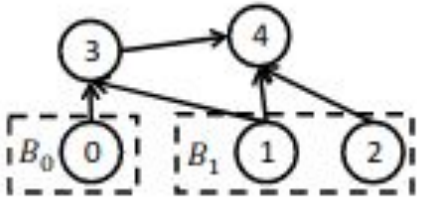
Definitions



Notations	Meaning
$G(V, E)$	bi-directed node-weighted graph G
w_i	the weight of node v_i
a_i	the minimum activation level of v_i
e_{ij}	the directed edge from v_i to v_j
r	a relationship type
R_i	the set of relationship types incident to v_i
r_{ij}	the relationship between v_i and v_j
Q, t_i	a keyword query, a keyword term
T_i	the set of nodes containing keyword t_i
B_i	a BFS instance w.r.t. t_i
h_j^b	the hitting level of v_j w.r.t. B_b
P_i^b	the set of all hitting paths of v_i w.r.t. B_b
C, $d(C)$	Central Graph, the depth of Central Graph
$\overline{A}$	average shortest distances (hops) of graph G
α	a tunable parameter to control a_i
l, l_{max}	BFS level and the max expansion depth (level)
M, m_{ij}	node-keyword matrix, the hitting level w.r.t. v_i and t_j

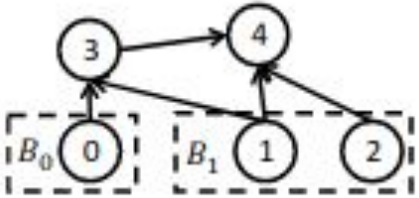
- r_{ij} is the labelled on edge e_{ij}
- BFS instance B_i starts from a set of nodes at the bottom level (0)
- Each node can only be hit once in terms of one BFS instance
- Each keyword t_i corresponds to a BFS instance B_i

Hitting level + path



- Hitting level (h_j^b) -> With BFS instance B_b , the hitting level of node v_j denoted by h_j^b is the first BFS expansion level l where v_j becomes a frontier to expand in B_b
  - E.g. For B_1, v_1 and v_2 has a hitting level of $h_1^1 = h_2^1 = 0$ since they are source nodes that expand at level 0. $h_3^1 = h_4^1 = 1$ because v_3 and v_4 are hit at BFS level 0 and become frontiers at level 1. v_3 will not expand to v_4 in B_1 since v_4 has already been hit
- Hitting path -> Given BFS instance B_b, the hitting path of a node v_j is any expansion path (from source nodes) that hit v_j and make it a frontier in the next expansion level. P_j^b = set of all hitting paths of v_j w.r.t. B_b
  - For B_1, and v_4 , only v_1 -> v_4 and v_2 -> v_4 are hitting paths. v_1-> v_3 -> v_4 is not since it is not an expansion path. There is no expansion from v_3 and v_4.

Central graph



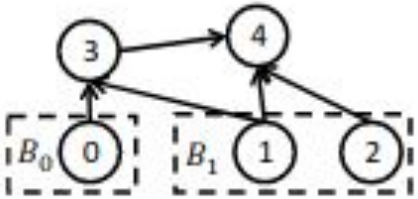
- Central graph -> Given keyword query Q={$t_0 \ldots t_{q-1}$}, for node v_j , if P_j^i != ∅ (w.r.t. T_i (set of all nodes containing the keywords) and B_i (BFS instance)) for every i, then define the central graph centered at v_j as

$$C = \bigcup_{i=0}^{q-1} P_j^i$$

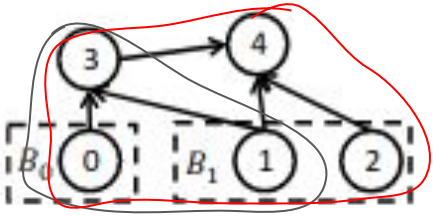
- (union of all hitting paths that center v_j where v_j is the Central Node which includes all keywords)
- The size/depth of the central graph C is the largest hitting level of Central Node v_j with the following equation

$$d(C) = \max_{i \in \{0,1,..,q-1\}} h_j^i$$

Central graph cont.



- Central graphs contain 3 conditions
  - Must contain all hitting paths from all input keywords and thus connect each keyword
  - For one keyword, a central graph contains all its hitting paths to the central node (allows multi-paths for every keyword). These hitting paths are the “shortest” in terms of hitting levels of the central node.
  - Depth of central path is bounded by the max hitting level of the respective central node
- Example of central graph
  - Above, there are 2 central graphs. One centered at v_3 with depth 1, covering hitting paths v_0->v_3 and v_1->v_3. The other is centered at v_4 with depth 2, covering hitting paths v_0->v_3->v_4, v_1->v_4 and v_2->v_4.



top-(k,d) Central Graphs

- Given keyword query Q and number k, find all central graphs with a depth no larger than d such that d is the smallest possible value to obtain at least k central graphs
  - k = number of central graphs to obtain
  - d = max depth of central graphs

Minimum activation level

- Use minimum activation level to weigh graph to generate meaningful answers
- Minimum activation level (denote as a_i for v_i) lower bounds the hitting level of a vertex. It makes vertices active for exploration.
- Summary node = node with large number of same labelled edges and small number of different edges
- Degree of summary = node to which extent tends to be a summary node (use it for weighing graphs, shown next slide)

Calculate minimum activation level

- Let R_i denote set of in-edge labels incident to v_i, for r in R_i, let $\bar{r}$ denote the number of in-edges of label r pointing to v_i .
- Weight calculated below using degree of summary of a node v_i:

$$w_i = \frac{\sum_{r \in R_i} \bar{r} \log_2 (1 + \bar{r})}{\sum_{r \in R_i} \bar{r}}$$

- Can normalize w_i and denote this as w_i' . Use the min/max weight w from all nodes $v_{0...n}$ in V.

$$w_i' = \frac{w_i - min(w)}{max(w) - min(w)}$$

Calculating minimum activation level - cont.

- Penalty and Reward mapping used to obtain activation level a_i for w_i with tunable parameter $\alpha \in (0, 1)$
  - α allows users to set preference for degree of summary in runtime
  - First compute average distance (hops) between 2 nodes by sampling ($\bar{A}$)
  - Calculate a_i by increasing penalty or decreasing reward using α and w_i from the average distance and round to nearest integer
  - Let $\bar{A}$ denote the average shortest distance between 2 nodes for G

$$Penalty(v_i) = \overline{A} \times \frac{(w_i - \alpha)}{1 - \alpha}, \text{ if } w_i > \alpha$$

$$Reward(v_i) = \overline{A} \times \frac{(\alpha - w_i)}{\alpha}, \text{ if } w_i < \alpha$$

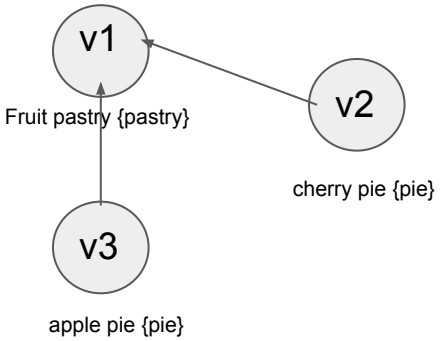
$$a_i = \begin{cases} Rounding(\overline{A} - Reward(v_i)) & w_i < \alpha \\ Rounding(\overline{A}) & w_i = \alpha \\ Rounding(\overline{A} + Penalty(v_i)) & w_i > \alpha \end{cases}$$

Calculating (Ā) - average shortest distance

- Compute the all-pairs shortest path (Floyd-Warshall) and get the average for each shortest path of each vertex
- https://www.tutorialspoint.com/all-pairs-shortest-paths

Two-Stage Parallel Algorithm

- Graph stored in CSR (Compressed Sparse Row Format)
- Bottom-up Search (Initialization for 3 data structures)
  - FIdentifier records 1 for nodes becoming frontiers in the next iteration, or else 0. (size=Θ(|V |))
  - CIdentifier records 1 for nodes identified as central nodes, or else 0. (size=Θ(|V |))
  - Initialize a node-keyword matrix M where m_{ij} records the hitting level of v_i for keyword t_j . m_{ij} = 0 if v_i contains keyword otherwise set to ∞. (size=Θ(|V |q) where q=#of keywords). For GPU initialization, initialize M on GPU and transfer back to CPU after search is done.



M	t_1=pastry	t_2=pie
v_1	0	∞
v_2	∞	0
v_3	∞	0

Algorithm 1: Two-stage Parallel Algorithm

```
/* Bottom-up Search to solve top-(k,d) Central
   Graph Prblem.                                   */
1 BFSLevel ← 0;
2 fork(); Initialize B_i for all t_i in Q; join();
3 while not terminate do
4 |   Enqueue frontiers// Only parallelize on GPU
5 |   fork(); Identify Central Nodes; join();
6 |   fork(); Expansion; join();
7 |   BFSLevel++;
/* Top-down Processing                             */
8 fork();Extract, prune and rank every Central Graph;join();
```

Two-Stage Parallel Algorithm - Cont.

- Enqueueing frontiers
  - At the beginning of each iteration (a new expansion level), extract and enqueue nodes into the frontier queue by examining the flags in FIdentifier which was modified in the last iteration or initialization phase. Enqueue nodes with FIdentifier=1 into the frontier queue.
  - On GPU, parallelize the process of checking FIdentifier and enqueueing nodes with lock. (Too expensive on CPU, so don't do on CPU)
  - A node becomes a frontier as long as it is in any BFS instance (i.e. different BFS instances share a frontier)
  - After enqueuing frontiers at every iteration, set all FIdentifier to 0 in parallel on GPU

Algorithm 1: Two-stage Parallel Algorithm

```
   /* Bottom-up Search to solve top-(k,d) Central
      Graph Prblem.                                  */
1  BFSLevel ← 0;
2  fork(); Initialize B_i for all t_i in Q; join();
3  while not terminate do
4  |   Enqueue frontiers// Only parallelize on GPU
5  |   fork(); Identify Central Nodes; join();
6  |   fork(); Expansion; join();
7  |   BFSLevel++;
   /* Top-down Processing                            */
8  fork();Extract, prune and rank every Central Graph;join();
```

Two-Stage Parallel Algorithm - Cont.

- Identifying central nodes
  - Using M, can identify where v_i is a central node by checking m_{ij} for each keyword t_j. Only need to look at frontiers.
  - Assuming max depth=3 and frontiers contain $\{v_1,v_2,v_3,v_4\}$, v_1 and v_4 are central nodes, but v_2 is not since it has not been hit by all keywords. V_3 is a central node but since it has a hit level of 4 with respect to t_1 which exceeds the max depth, we won't generate a central graph on v_3. We will generate 2 central graphs where v_1 and v_4 at this point.

M	t_1	t_2	t_3
v_1	3	2	1
v_2	∞	2	1
v_3	4	1	1
v_4	1	1	1
v_5	∞	∞	1
v_6	2	0	∞

Two-Stage Parallel Algorithm - Cont.

Algorithm 1: Two-stage Parallel Algorithm

```
   /* Bottom-up Search to solve top-(k,d) Central
      Graph Prblem.                                  */
1  BFSLevel ← 0;
2  fork(); Initialize B_i for all t_i in Q; join();
3  while not terminate do
4  |   Enqueue frontiers// Only parallelize on GPU
5  |   fork(); Identify Central Nodes; join();
6  |   fork(); Expansion; join();
7  |   BFSLevel++;
   /* Top-down Processing                            */
8  fork();Extract, prune and rank every Central Graph;join();
```

- Alg 1: Two-stage parallel algorithm
- /* Bottom up search */
- Int FIdentifier[|V|] = set keyword nodes to 1, else 0
- Int CIdentifier[|V|] = zeros(|V|) //zeros since we can call IdentifyCentralNodes(...) using M
- Int M[|V|][|q|] = set 0 if keyword contained, else infinity //initialize on GPU, transfer back to CPU when search is done
- Int NodeWeights[|V|] = CalcWeights(...) //gets degree of summary
- Int Alpha = …
- Int k = …
- twoStageParallelAlg() :
    - Int BFSLevel = 0
    - fork(); Int BFSInstance[][] = instances for each keyword, add all keyword nodes to each instance //size(|V|,|q|) join();
    - While not terminate do
        - Enqueue frontiers of nodes with FIdentifier = 1 //parallelize on GPU with lock
        - fork(); CIdentifier = IdentifyCentralNodes(M,FIdentifier); //sets CIdentifier as 1 for nodes where FIdentifier=1 join();
        - set FIdentifer[|V|] to all 0 in parallel on GPU
        - fork(); M, FIdentifier = Expansion(G,M,frontiers,NodeWeights,BFSLevel,alpha); join(); //Alg 2
        - BFSLevel++;
    - /* Top-down Processing */
    - Node CentralNodes[] = getCentralNodes(CIdentifier) //add all nodes where CIdentifier = 1
    - fork(); TopDownProcessing(G,M,CentralNodes,NodeWeights,Alpha,k); join(); //Alg 3

Two-Stage Parallel Algorithm - Cont.

Algorithm 1: Two-stage Parallel Algorithm

```
   /* Bottom-up Search to solve top-(k,d) Central
      Graph Prblem.                                  */
1  BFSLevel ← 0;
2  fork(); Initialize B_i for all t_i in Q; join();
3  while not terminate do
4  |   Enqueue frontiers// Only parallelize on GPU
5  |   fork(); Identify Central Nodes; join();
6  |   fork(); Expansion; join();
7  |   BFSLevel++;
   /* Top-down Processing                            */
8  fork();Extract, prune and rank every Central Graph;join();
```

- Alg 1: Two-stage parallel algorithm
- /* Bottom up search */
- Int FIdentifier[|V|] = set keyword nodes to 1, else 0
- Int frontiers[|V|] //our queue (initalize to all 0's)
- Int CIdentifier[|V|] = zeros(|V|) //zeros since we can call IdentifyCentralNodes(...) using M
- Int M[|V|][|q|] = set 0 if keyword contained, else infinity //initialize on GPU, transfer back to CPU when search is done
- Int NodeWeights[|V|] = CalcWeights(...) //gets degree of summary
- Int activation[|V|] = CalcActivation(NodeWeights,alpha)
- Int Alpha = …
- Int k = …
- String keywords[|q|] //array of all keywords in the query
- twoStageParallelAlg() :
    - Int BFSLevel = 0
    - While not terminate do
        - Enqueue frontiers of nodes with FIdentifier = 1 (copy FIdentifier to frontiers)
        - #pragma omp for
        - set FIdentifer[|V|] to all 0 in parallel
        - CIdentifier = IdentifyCentralNodes(M,frontiers,keywords); //sets CIdentifier as 1 for nodes but only check when FIdentifier=1 use #pragma omp for inside function instead of fork
        - M, FIdentifier = Expansion(G,M,frontiers,FIdentifier, CIdentifier, activation,BFSLevel,alpha,keywords); //Alg 2
        - BFSLevel++;
    - /* Top-down Processing */
    - TopDownProcessing(G,M,CIdentifier,,NodeWeights,Alpha,k); //Alg 3

Expansion Procedure

- Skip frontiers that are marked as central nodes
- If the frontier's activation level is greater than l, keep it in the frontier and move on
- For each BFS instance, skip the frontier if hit level > l
- Add neighbor of the the frontier to the next frontier if it does not have a keyword and the activation level is l or lesser
- Goal: Perform expansion on the graph to get the hit levels for M and identify central nodes

Algorithm 2: Expansion Procedure

Input : *data graph G, M, frontiers, node weights, BFS expansion level l, α*
Output: *modified M and FIdentifier*

```
/* CPU threads parallel level                                   */
foreach frontier f do
    if CIdentifier[v_f] = 1 then
        continue;
    calculate a_f from w_f and α;
    if a_f > l then
        FIdentifier[v_f] ← 1;
        continue;
    /* GPU warps parallel level                                 */
    foreach BFS instance B_i do
        h^i_f ← m_fi;
        if h^i_f > l then
            continue;
        /* GPU threads parallel level                           */
        foreach neighbor v_n of v_f do
            h^i_n ← m_ni;
            if h^i_n ≠ ∞ then                  Have visited / contains a keyword, skip
                continue;
            if v_n is not a keyword node then  Keyword nodes are marked with a keyword
                calculate a_n from w_n and α;
                if a_n > l + 1 then
                    FIdentifier[v_f] ← 1;
                    continue;
            m^i_n ← l + 1;// Set hitting level in M
            FIdentifier[v_n] ← 1;
return;
```

Expansion Procedure - Parallelism

- On the GPU, let a warp handle one BFS instance of a frontier v_f. The threads within a warp handle different neighbours of v_f
  - Warp = group of threads active simultaneously in SIMD model
- Use coarse grain parallelism (higher computation, lower communication)
- Let CPU threads handle different frontiers with a dynamic scheduling which means that once a thread finishes a frontier, it looks for another
- Parallelism and dynamic scheduling done on OpenMP

Algorithm 2: Expansion Procedure

Input : *data graph G, M, frontiers, node weights, BFS expansion level l, α*
Output: *modified M and FIdentifier*

```
/* CPU threads parallel level                                   */
foreach frontier f do
    if CIdentifier[v_f] = 1 then
        continue;
    calculate a_f from w_f and α;
    if a_f > l then
        FIdentifier[v_f] ← 1;
        continue;
    /* GPU warps parallel level                                 */
    foreach BFS instance B_i do
        h_f^i ← m_fi;
        if h_f^i > l then
            continue;
        /* GPU threads parallel level                           */
        foreach neighbor v_n of v_f do
            h_n^i ← m_ni;
            if h_n^i ≠ ∞ then                  Have visited / contains a keyword, skip
                continue;
            if v_n is not a keyword node then  Keyword nodes are marked with a keyword
                calculate a_n from w_n and α;
                if a_n > l + 1 then
                    FIdentifier[v_f] ← 1;
                    continue;
            m_n^i ← l + 1;// Set hitting level in M
            FIdentifier[v_n] ← 1;
return;
```

Expansion Procedure - Parallelism

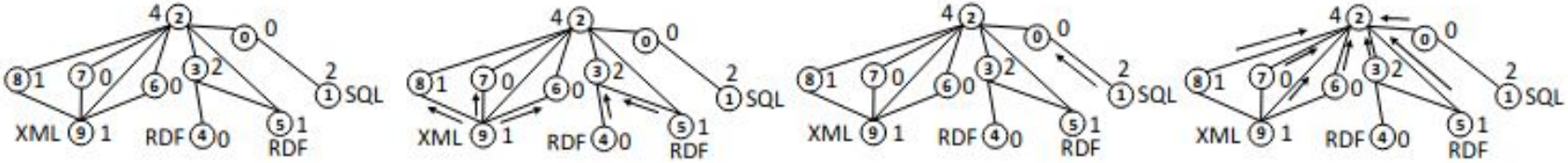
- Expansion(G, M, frontiers, FIdentifer,activation, BFSLevel, alpha,keywords):
  - #pragma omp for collapse(3) schedule(dynamic)
  - For v_f=0….|V|-1
    - If frontiers[v_f] == 1
      - If CIdentifier[v_f] = 1 then
        - Continue;
      - If a_f > l then
        - FIdentifer[v_f] = 1
        - Continue
      - for i=0…q-1 (index that corresponds to each BFS instance B_i for t_i)
        - h_f^i -> m_f^i
        - If h_f^i > l then
          - continue;
        - foreach neighbor v_n of v_f do
          - h_n^i <- m_n^i
          - If h_n^i != ∞ then
            - continue;
          - If !(v_n).contains(keyword[i]) then
            - If a_n > l+1 then
              - FIdentifer[v_f] = 1
              - Continue
          - m_n^i = l+1
          - FIdentifer[v_n] = 1;
  - Return modified M and FIdentifier

Algorithm 2: Expansion Procedure

Input : *data graph G, M, frontiers, node weights, BFS expansion level l, α*
Output: *modified M and FIdentifier*

```
/* CPU threads parallel level                                   */
foreach frontier f do
    if CIdentifier[v_f] = 1 then
        continue;
    calculate a_f from w_f and α;
    if a_f > l then
        FIdentifier[v_f] ← 1;
        continue;
    /* GPU warps parallel level                                 */
    foreach BFS instance B_i do
        h_f^i ← m_fi;
        if h_f^i > l then
            continue;
        /* GPU threads parallel level                           */
        foreach neighbor v_n of v_f do
            h_n^i ← m_ni;
            if h_n^i ≠ ∞ then                  Have visited / contains a keyword, skip
                continue;
            if v_n is not a keyword node then  Keyword nodes are marked with a keyword
                calculate a_n from w_n and α;
                if a_n > l + 1 then
                    FIdentifier[v_f] ← 1;
                    continue;
            m_n^i ← l + 1;// Set hitting level in M
            FIdentifier[v_n] ← 1;
return;
```

Expansion Procedure - Example



(a) BFS expansion level 0 (b) BFS expansion level 1 (c) BFS expansion level 2 (d) BFS expansion level 3

Fig. 4. A running example with values of a_i attached to each node from Fig. 1. Arrows denote the expansion.

Example 4. *Fig. 4 illustrates a running example for nodes in Fig. 1. In Fig. 4a, three BFS instances are initiated at* $\{v_9\}$, $\{v_4, v_5\}$ *and* $\{v_1\}$. *At every BFS level, the expansion is in parallel. Since* $a_4 = 0$, *only* v_4 *is active to expand, but* v_3 *is not active because* $a_3 = 2$. *As a result, there is no expansion. At expansion level 1 shown in Fig. 4b,* v_9, v_4 *and* v_5 *start expansion. Also,* v_3 *is able to accept expansion and it becomes a frontier in level 2 which reaches its minimum activation level. The hitting levels of new frontiers are,* $h_6^0 = h_7^0 = h_8^0 = h_3^1 = 2$. *Finally, at level 3 (Fig. 4d), every node near* v_2 *can expand to it.* v_2 *is identified as a Central Node in the next iteration (not shown) and its depth is 4.*

Top-down processing

- 3 steps
  - 1) Extract respective Central Graphs
  - 2) Apply level-cover strategy to all central graphs to prune redundant nodes based on keyword co-occurrence
  - 3) Select final top-k answers by a scoring function
- Goal: construct central graphs, return top(k,d) graphs based on top k highest ranked results

Algorithm 3: Top-down Processing

Input : $G(V, E)$, M, *identified Central Nodes, node weights,* α, k
Output: *Final top-k answers*

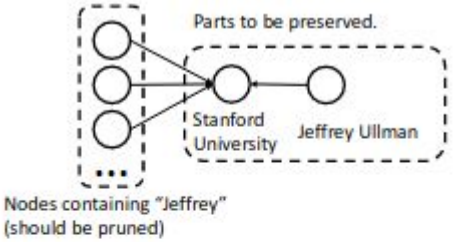
Initialize top-k answer heap T_k;
foreach v_c *in Identified Central Nodes* **do**
  insert v_c to $|Q|$ frontier queues f_1 to $f_{|Q|}$;
  // Loop until all frontier queues are empty.
  while $\exists f \neq \emptyset$ **do**
    $v_f \leftarrow f.next(), f' \leftarrow \emptyset$;
    /* Scan neighbors of v_f */
    foreach $v_n \in N(v_f)$ **do**
      foreach B_i **do**
        if v_f *has keywords* ***and*** $h_f^i = 1 + \max\{a_n, h_n^l\}$ **then**
          Extract v_n, insert v_n into f', if not in f';
        if v_f *has no keywords* ***and*** $h_f^i = 1 + \max\{a_n, h_n^l, a_f - 1\}$ **then**
          Extract v_n, insert v_n into f', if not in f';
    $f \leftarrow f'$;
  /* let C_n denote the Central Graph at v_c */
  Apply level-cover strategy to C_n;
  Insert into T_k, if possible;
return;

Top-down processing cont.

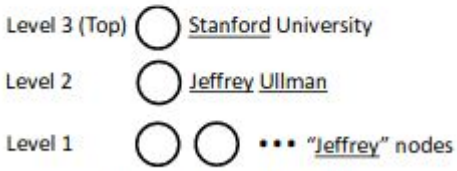
- Parallelism Strategy
  - Let one thread to recover one or more Central Graphs
  - Dynamic scheduling handled by OpenMP
  - Implement top-down process on CPU

Top-down processing cont.

- Level-cover strategy
  - Classify only keyword nodes within a Central Graph into different levels based on the number of keywords they contribute
  - Central Node is always at top level
  - Proceed in a greedy manner starting downwards from top level where nodes contain the most keywords
  - If nodes in one level cover all keywords, prune all nodes in the rest levels along with hitting paths from pruned nodes to Central Nodes
  - Preserve as many keyword nodes as possible since nodes will not lead to pruning of nodes in the same level



(a) Central Graph before pruning



(b) Level classification

Fig. 5. Level cover strategy example.

Example 5. *As shown in Fig. 5, the input keywords are Stanford, Jeffrey and Ullman. After pruning nodes with only one keyword "Jeffrey", we have an answer with only Stanford University and Jeffrey Ullman nodes.*

Top-down processing cont.

- Scoring function
  - Select final top-k answers from pruned top(k,d) Central Graphs using a ranking function to restrict its width

$$S(C) = d(C)^{\lambda} \sum_{v_i \in C} w_i$$

  - C=Central graph
  - λ>=0 (parameter to control effect of CG depth), set λ=0.2 by default
  - Use central graphs of top k highest scores to put into the answer heap

Top-down processing cont.

- Initialize frontier queues for each keyword in a query (|Q| = number of keywords in a query)
- For each frontier, apply theorem V.4 between the frontier and its neighbours to see whether a neighbour can be recovered w.r.t. a keyword t_i
- Get the central graph C_n by having a copy of v_c and adding neighbours to it and other nodes and edges through extraction done on line 9 and 11
- Level-cover strategy applied to C_n on line 13 (need to use the constructed central graph from extraction and M for level cover strategy), choose top-k based on scoring function
- Insert pruned graph from top-level strategy to top-k answer heap (T_k)

Theorem V.4. *Suppose v_i expands to v_j during bottom-up search and the extraction is now at v_j to extract v_i, we have the following heuristics between h_i^l and h_j^l for keyword t_l.*

1) If v_j contains keywords, $h_j^l = 1 + \max\{a_i, h_i^l\}$,

2) If v_j contains no keywords, $h_j^l = 1 + \max\{a_i, h_i^l, a_j - 1\}$.

Algorithm 3: Top-down Processing

```
   Input : G(V, E), M, identified Central Nodes, node weights, α, k
   Output: Final top-k answers
1  Initialize top-k answer heap T_k;
2  foreach v_c in Identified Central Nodes do
3      insert v_c to |Q| frontier queues f_1 to f_|Q|;
       // Loop until all frontier queues are empty.
4      while ∃f ≠ ∅ do
5          v_f ← f.next(), f' ← ∅;
           /* Scan neighbors of v_f                    */
6          foreach v_n ∈ N(v_f) do
7              foreach B_i do
8                  if v_f has keywords and h_f^i = 1 + max{a_n, h_n^l}
                     then
9                      Extract v_n, insert v_n into f', if not in f';
10                 if v_f has no keywords and
                     h_f^i = 1 + max{a_n, h_n^l, a_f − 1} then
11                     Extract v_n, insert v_n into f', if not in f';
12         f ← f';
       /* let C_n denote the Central Graph at v_c  */
13     Apply level-cover strategy to C_n;
14     Insert into T_k, if possible;
15 return;
```

Top-down processing cont.

- TopDownProcessing(G,M,CentralNodes,NodeWeights,Alpha,k):
  - Node* heap[k];
  - Initialize $f_1 \ldots f_q$ //frontiers for each keyword
  - Initalize levels[] // will represent the level classification for each central graph
  - foreach v_c in CentralNodes do
    - foreach $f_1 \ldots f_q$ do
      - f.insert(v_c)
  - NewC = new graph starting at v_c, will be the central graph
  - while there exists a frontier f that is not empty:
    - v_f -> f.next(), f' -> NULL
    - foreach v_n in Neighbours(v_f) do
      - foreach B_i do
        - If v_f has keywords and $h_f^i = 1+\max(a_n, h_n^l)$
          - Connect v_n to NewC, insert v_n into f' if not in f'
        - If v_f has no keywords and $h_f^i = 1+\max(a_n, h_n^l, a_f-1)$
          - Connect v_n to NewC, insert v_n into f' if not in f'
    - f -> f'
  - level = LevelCoverStrategy(NewC,M)
  - levels.insert(level)
  - SortbyScoring(levels) //get score for each levelled central graph and sort by score defined by the scoring function
  - Insert levels into heap[k] if possible //add top k levels into heap[k]

Algorithm 3: Top-down Processing

Input : $G(V, E)$, M, *identified Central Nodes, node weights,* α, k
Output: *Final top-k answers*

1 Initialize top-k answer heap T_k;
2 **foreach** v_c *in Identified Central Nodes* **do**
3 insert v_c to $|Q|$ frontier queues f_1 to $f_{|Q|}$;
  // Loop until all frontier queues are empty.
4 **while** $\exists f \neq \emptyset$ **do**
5 $v_f \leftarrow f.next(), f' \leftarrow \emptyset$;
  /* Scan neighbors of v_f */
6 **foreach** $v_n \in N(v_f)$ **do**
7 **foreach** B_i **do**
8 **if** v_f *has keywords* ***and*** $h_f^i = 1 + \max\{a_n, h_n^l\}$ **then**
9 Extract v_n, insert v_n into f', if not in f';
10 **if** v_f *has no keywords* ***and*** $h_f^i = 1 + \max\{a_n, h_n^l, a_f - 1\}$ **then**
11 Extract v_n, insert v_n into f', if not in f';
12 $f \leftarrow f'$;
  /* let C_n denote the Central Graph at v_c */
13 Apply level-cover strategy to C_n;
14 Insert into T_k, if possible;
15 **return**;

Top-down processing cont.

- TopDownProcessing(G,M,CIdentifier,NodeWeights,Alpha,k,keywords):
  - Node* heap[k];
  - Initalize CentralGraphArray[] // will represent the level classification for each central graph
  - #pragma omp for schedule(dynamic)
  - for i=0…|V|-1
    - If CIdentifier[i]==1 do:
      - Initialize queue f[|q|] //frontiers for each keyword (array of queues of size q)
      - Create central vertex v_c from index i
      - for j=0…q-1 do
        - f[j].push(v_c)
      - NewC = new graph starting at v_c, will be the central graph
      - Bool frontiers_empty = allFrontiersEmpty(f)
      - while frontiers_empty == false:
        - k=0;
        - f2 = f
        - while(f2[k].empty()==true and k<q) //looks for a non-empty frontier
          - k=k+1;
        - if(k>=q) {
          - break;
        - }
        - v_f -> f[k].pop(), f' -> NULL
        - foreach v_n in Neighbours(v_f) do
          - for k_i=0…q-1
            - If v_f.contains(keywords[k_i]) and $h_f^{k\_i}$ = 1+max(a_n, h_n^l)
              - Connect v_n to NewC, push v_n into f' if not in f'
            - If !v_f.contains(keywords[k_i]) and $h_f^{k\_i}$ = 1+max(a_n, h_n^l,a_f-1)
              - Connect v_n to NewC, push v_n into f' if not in f'
        - f[k] = f'
        - frontiers_empty = allFrontiersEmpty(f)
      - prunedCentralGraph = LevelCoverStrategy(newC, keywords)
      - #pragma omp critical {
      - CentralGraphArray.insert(prunedCentralGraph)
      - }
  - SortbyScoring(CentralGraphArray) //get score for each levelled central graph and sort by score defined by the scoring function
  - Insert CentralGraphArray into heap[k] if possible //add top k graphs into heap[k]

Algorithm 3: Top-down Processing

Input : $G(V, E)$, M, *identified Central Nodes, node weights,* α, k
Output: *Final top-k answers*

1 Initialize top-k answer heap T_k;
2 **foreach** v_c *in Identified Central Nodes* **do**
3 insert v_c to $|Q|$ frontier queues f_1 to $f_{|Q|}$;
// Loop until all frontier queues are empty.
4 **while** $\exists f \neq \emptyset$ **do**
5 $v_f \leftarrow f.next(), f' \leftarrow \emptyset$;
/* Scan neighbors of v_f */
6 **foreach** $v_n \in N(v_f)$ **do**
7 **foreach** B_i **do**
8 **if** v_f *has keywords* ***and*** $h_f^i = 1 + \max\{a_n, h_n^l\}$ **then**
9 Extract v_n, insert v_n into f', if not in f';
10 **if** v_f *has no keywords* ***and*** $h_f^i = 1 + \max\{a_n, h_n^l, a_f - 1\}$ **then**
11 Extract v_n, insert v_n into f', if not in f';
12 $f \leftarrow f'$;
/* let C_n denote the Central Graph at v_c */
13 Apply level-cover strategy to C_n;
14 Insert into T_k, if possible;
15 **return**;

Level cover strategy algorithm

- LevelCoverStrategy(newC, keywords):
  - min = GetMinNumberOfKeywordsToPreserve(newC, keywords) (see slide 28 for details)
  - root = newC.root() //central node (copy of the original graph newC)
  - prunedGraph = root //newly constructed pruned graph
  - Initialize frontier
  - frontier.push(root)
  - added[|V|] = initalize all to false
  - added[root.index()] = true
  - vector<int> distfromC[|V|] = initialize to all infinity
  - distFromC[root.index()] = 0
  - While frontier is not empty
    - curr = frontier.pop();
    - node_keywords = getKeywords(curr.label(), keywords) //get all keywords from label of v_n that is part of the keyword query e.g. [k1,k2,k3]
    - number_of_keywords = length(node_keywords)
    - If number_of_keywords >= min
      - JoinShortestPath(curr, distfromC, added, prunedGraph) //join shortest path curr -> … -> (last node not added) onto prunedGraph (see slide 29)
    - For all incoming neighbours v_n of curr:
      - if distFromC[v_n.index()]==infinity
        - distFromC[v_n.index()] = distFromC[curr.index()] + 1
      - If v_n is not in frontier
        - frontier.push(v_n)
  - Return prunedGraph

Level cover strategy algorithm - level classification

- GetMinNumberOfKeywordsToPreserve(G, keywords): //keywords is an array with a list of keywords in the query
  - vector<vector<string>> levelClass[|q|][|V|] //|q| = number of keywords in the query
  - levelClass[|q|-1].push(G.root().label())
  - Initalize frontier
  - Initalize explored[|V|] = {set all to false}
  - explored[G.root().index()] = true
  - frontier.push(G.root())
  - While frontier is not empty
    - curr = frontier.pop()
    - if(curr!=G.root())
      - curr_keywords = getKeywords(curr.label(), keywords) //returns array of all keywords from node label
      - If length(curr_keywords)!=0
        - levelClass[length(curr_keywords)-1].push(curr.label());
    - For all neighbours v_n of curr
      - If explored[v_n] == false
        - explored[v_n] = true
        - frontier.push(v_n)
  - //return min number of keywords that a label must have for its corresponding node to be preserved
  - keywords_to_scan = keywords
  - For i=(|q|-1)…0
    - For j=0…length(levelClass[i])
      - splitLC = levelClass[i][j].split(“”)
      - For k=0…length(keywords_to_scan)
        - If keyword_to_scan[k]==””
          - continue
        - For L=0…length(splitLC)
          - If keywords_to_scan[k] == splitLC[L]
            - Keywords_to_scan[k] = “”
            - Break
    - If all elements in keywords_to_scan are “”
      - Return i+1

Level cover strategy algorithm - Shortest Path

- JoinShortestPath(&last_node, &distfromC, &added, &prunedGraph)
  - Path = last_node (copy from the original graph)
  - frontier = []
  - frontier.push(last_node)
  - notaddedVerticesToJoin = []
  - addedVerticesToJoin = []
  - While frontier is not empty
    - curr = frontier.pop()
    - minCost = distfromC[curr.index()] - 1
    - For all outgoing neighbours v_n from curr //add vertices with min cost to graph
      - If distfromC[V_n.index()] == minCost
        - If added[v_n.index()] == false
          - Join path[curr.index()] -> v_n (copy from the original graph)
          - frontier.push(v_n) if v_n is not in frontier
        - Else If added[v_n.index()] == true
          - notaddedVerticesToJoin.append(curr.index())
          - addedVerticesToJoin.append(v_n.index())
  - For i=0…len(addedVerticesToJoin)-1 //Join the constructed path with the pruned graph
    - Join prunedGraph[addedVerticesToJoin[i]] <- path[notaddedVeticesToJoin[i]]
  - For all nodes n in path
    - added[n.index()] = true

Top-down processing - output

- Assuming that the top-k answer heap (T_k) is an AVL/height balanced tree (since insertion time complexity is O(log2(n)) according to the paper), the output of top-down processing would look something like this for the query `SELECT "Jeffrey", "Ullman", "Stanford"`
- For top-(k,d) central graphs, assume k=4, so the heap has 4 nodes.