Efficient Subgraph Matching on Billion Node Graphs

Matteo Pagliari 854353
Nicolas Tagliabue 853097

Introduction

In computer science, graphs represent a very important and vastly used Data structure.

Example of very large graphs:

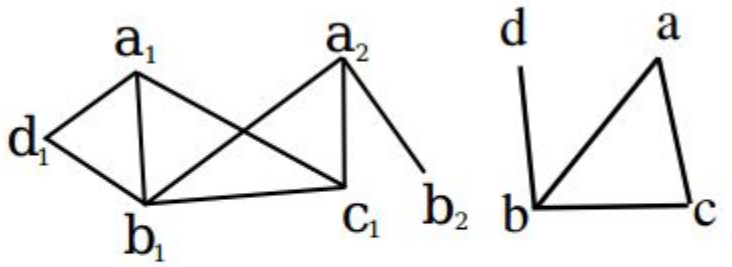
- Facebook (800 million vertex)
- Protein-Protein interaction
- Genome sequencing

Introduction

DEFINITION 1 (SUBGRAPH QUERY). *We denote a subgraph query as* $q=(V_q, E_q, T_q)$, *where* $T_q: V \rightarrow \Sigma^*$ *represents the label constraint for each vertex in* V_q.

DEFINITION 2 (THE PROBLEM OF SUBGRAPH MATCHING). *For a graph* G *and a subgraph query* q, *the goal of subgraph matching is to find every subgraph* $g = (V_g, E_g)$ *in* G *such that there exists a bijection* $f : V_q \rightarrow V_g$ *that satisfies* $\forall v \in V_q, T_q(v) = T_G(f(v))$ *and* $\forall e = (u, v) \in E_q, (f(u), f(v)) \in E_g$, *where* $T_G(f(v))$ *represents the label of the vertex* $f(v)$ *in* G.

Example



(a) Data graph G (b) A query q

Figure 1: The Example for Subgraph Matching

Results: (a1,b1,c1,d1) and (a2,b1,c1,d1)

Aim of this algorithm

1. Propose an algorithm suitable to use with very large graphs that uses no indexes except a simple string index to map text labels to vertex IDs
2. A query optimization strategies for query partitioning to avoid join operations
3. Little memory footprint

Exploration vs Joins

Join operations for subgraph matching

With the join operations the query graph is divided into small queries.

Each small queries produces a result that has to be joined together in order to obtain the final result.

This produces a lot of intermediate result that are not useful, creating a lot of overheads in terms of computing power.

Exploration for subgraph matching

Exploiting the label-vertex IDs index, starting for the first node we find the associated label.

Then starting from the node that we have just found, we try to reach the second node and so on…

This allow us to avoid doing most of the joins for each intermediate result.

The approach presented

STwig

Query decomposition in STwig.

What is a STwig ?

It is a two level tree structure defined as this: $q=(r,L)$

MatchSTwig(q) algorithm

Algorithm 1 MatchSTwig(q) where $q = (r, L)$

$S_r \leftarrow$ `Index.getID`(r)
$R \leftarrow \varnothing$
for each n in S_r **do**
  $c \leftarrow$ `Cloud.Load`(n)
  for each l_i in L **do**
    $S_{l_i} \leftarrow \{m | m \in$ c.children and `Index.hasLabel`$(m, l)\}$
  $R = R \cup \{\{n\} \times S_{l_1} \times S_{l_2} \times \cdots \times S_{l_k}\}$
Return R

Subgraph matching

Using STwig as the unit of query processing, our method performs subgraph matching in three steps:

1. **Query Decomposition and STwig Ordering.**

Query decomposition

The objective function is to minimize the number of the components (STwigs)

We have to find a set of STwigs S, such that every edge of the query graph G belongs to one and only one STwig of S.

If S is called STwig cover of G, the problem is to find the minimum STwig cover of G. This problem is polynomial equivalent to the *minimum vertex cover problem* which is NP-hard.

STwig Order Selection

There are 2 main rules:

1. A root node of a STwigs has to be connected with the previous STwigs.
2. Favor generation of STwigs with higher selectivity in order to reduce the size of intermediate results. Selectivity is determined by two factors:
    a. the popularity of the root label (the more popular, the less is selective)
    b. the more child node the root has, the higher is selective

f-value of a vertex: $f(v)=\frac{deg(v)}{freq(v.label)}$

STwig-Order-Selection(q) algorithm

Algorithm 2 STwig-Order-Selection(q)

```
1: S = ∅
2: 𝕋 = ∅
3: while q has more edges do
4:    if S = ∅ then
5:       pick an edge (v, u) such that f(u) + f(v) is the largest
6:    else
7:       pick an edge (v, u) such that v ∈ S and f(u) + f(v) is
         the largest
8:    T_v ← the STwig rooted at v
9:    add T_v to 𝕋
10:   S ← S∪ neighbor(v)
11:   remove edges in T_v from q
12:   if deg(u) > 0 then
13:      T_u ← the STwig rooted at u
14:      append T_u to 𝕋
15:      remove all edges in T_u from q
16:      S ← S∪ neighbor(u)
17:   remove u, v and all nodes with degree 0 from S
18: return 𝕋
```

Time complexity is **$O(n^2 \log(n))$**

n is the number of nodes

Subgraph matching

Using STwig as the unit of query processing, our method performs subgraph matching in three steps:

1. Query Decomposition and STwig Ordering.
2. **Exploration**

Exploration - 1

After we have generated an ordered list of STwigs we process them one by one.

Assuming q1 is the first STwig obtained by the previous step, we process q1 first and we get result G(q1), which are the results of the algorithm matchSTwig(q1).

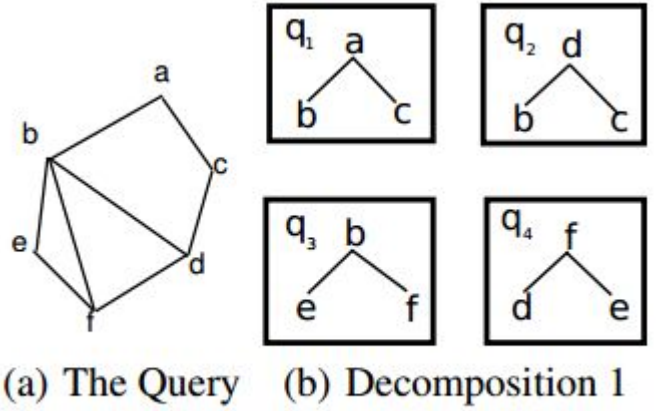
From G(q1) we extract the binding informations which are useful for the next partial queries.

Then we start processing the next query in order.

Exploration - 2

If the next query for example q2, doesn't have its root node bounded to the previous query q1, we will need to retrieve all the noded labeled with the label of the root node (in this case label d). It’s possible that also the child nodes are connected with binding informations.

If the next query for example q3, does have its root node bounded to the previous query q1, we will need to retrieve only the nodes belonging to the binding informations.



In any case we will update the binding information with the newly found bounds

Subgraph matching

Using STwig as the unit of query processing, our method performs subgraph matching in three steps:

1. Query Decomposition and STwig Ordering
2. Exploration
3. **Join**

Distributed subgraph matching

Distributed subgraph matching

The first step, *Query decomposition* and *STwig ordering*: this step is made once for all, at the beginning, and its results are known to all the machines.

The second step, *Exploration*: each machine executes the MatchSTwig algorithm.

The third step, *Join*, is not trivial.

Join - 1

Every machine will collect the results from other machines that are in the load set, and will perform the join operation.

$$R_k(q_i) = \bigcup_{k' \in F_{k,i} \cup \{k\}} G_{k'}(q_i)$$

In the end the results of each machines will be merged together

$$R_k = R_k(q_{i_1}) \bowtie R_k(q_{i_2}) \bowtie \cdots \bowtie R_k(q_{i_n})$$

$$\bigcup_{k \in \text{cluster}} R_k$$

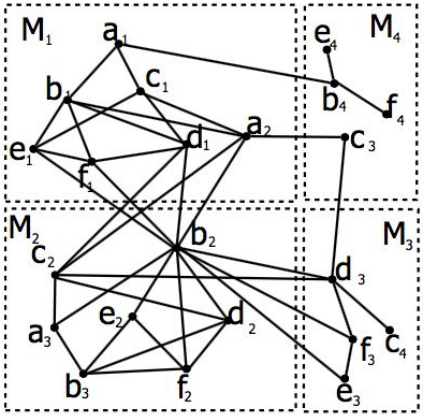
Join - 2

Criteria for choosing Load set Fk,i:

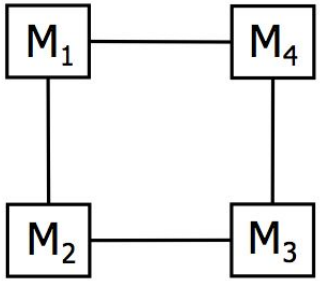
- Rk and Rk’ have to be disjoint to avoid deduplication:
  - We have to choose a Head STwig and set Fk.h=0
  - Data graph is disjointly partitioned among the machines and this guarantees that the local result are disjoint too
- Fk,i small as possible to reduce network communication

The Cluster Graph

In the preprocessing phase, for each pairs of machines, we record all possible pairs of node labels. That is, we associate a pair of labels (A, B) to a pair of machines (i, j) if there exists an edge u -- v such that u and v reside in machine i and j respectively, and u and v are labeled A and B respectively.



(a) G_q



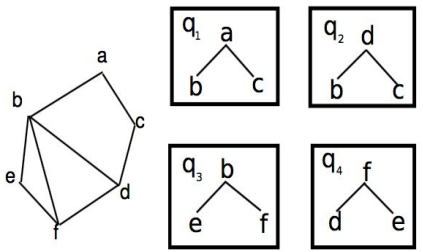
(b) A cluster graph

Load Set

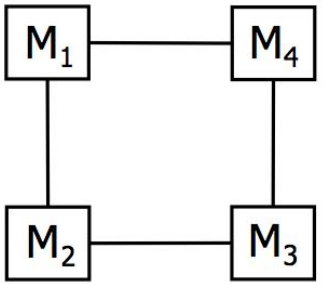
THEOREM 4. *Machine k only needs query results for STwig q_t from machines in the set $F_{k,t}$, which is given by:*

$$F_{k,t} = \{j | D_C(k,j) \leqslant d(r_s, r_t)\}$$

where $d(r_s, r_t)$ denotes the shortest distance in q between r_s and r_t (the root nodes of q_s and q_t).



(a) The Query (b) Decomposition 1



(b) A cluster graph

Head STwig selection

THEOREM 5. *Assume a query q is decomposed as $q_1, \cdots, q_n$ and q_s is selected as the head STwig. The communication cost is*

$$T(s) = \sum_{k \in C} |\{j | D_C(k, j) \leqslant d(s)\}| \qquad (2)$$

where $d(s) = \max_{1 \leqslant i \leqslant n}\{d(r_s, r_i)\}$.

To minimize T(s) we should minimize d(s): we compute d(i) for each qi and we select qs with minimal d(s). The complexity is O(m^3) where m in the number of vertices in the subgraph pattern q

Our implementation

We used Python with Networkx to manage graphs and NumPy for some matrix usage.

Our implementation by the way is **not** parallelized, even if we simulated the presence of multiple machines by making it parametric to the number of machines.

So the graph is splitted among a N number of machines.

Experiments results

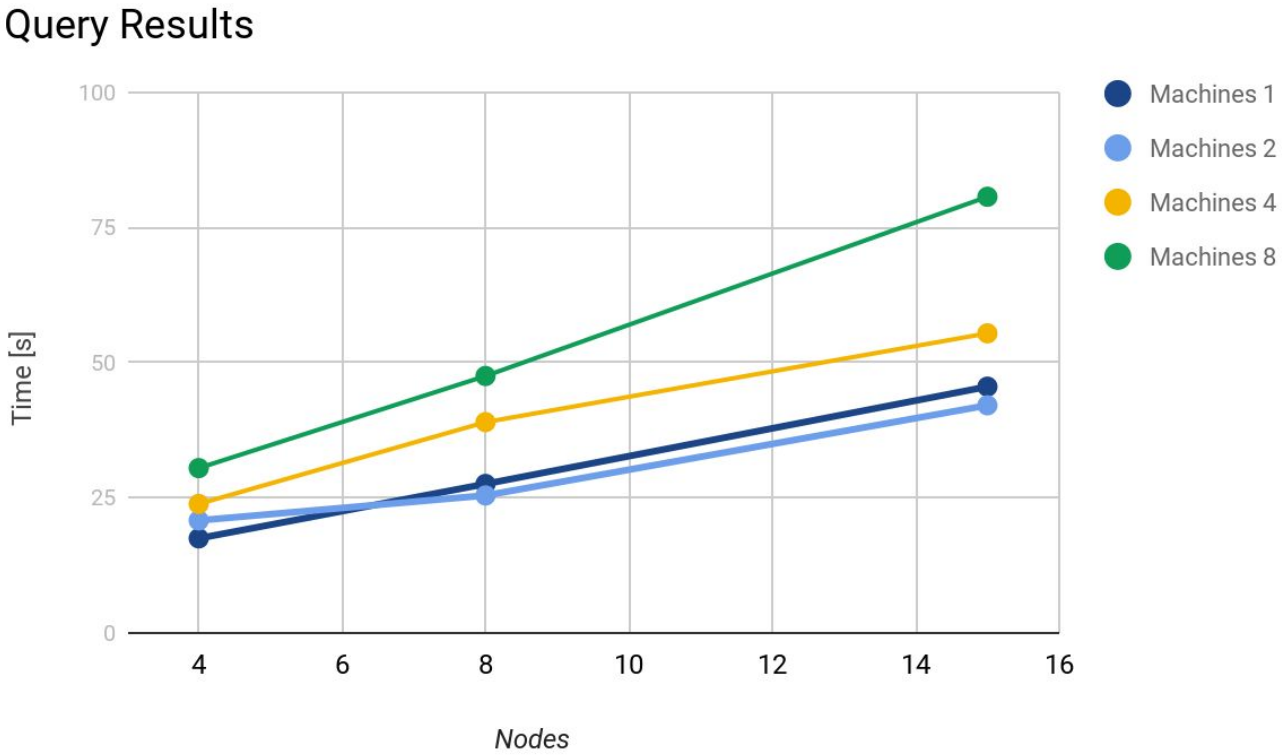
What we used for testing

We used one of the nets used in the paper, WordNet, which was freely available.

Wordnet is a lexical database of English words. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept.

In order to keep things manageable we used a subset of WordNet consisting in approximately **10.000 nodes**, **1.500 edges** and **5 labels.**

Experiment results on WordNet



Future improvements

To be continued...

The single most important thing to be developed in the future to increase performance significantly is, of course, the **parallelization** of the algorithm.

Considering that this whole algorithm has distribution and parallelization in mind it will not be so complicated to do it.