Tesseract: Distributed, General Graph Pattern Mining on Evolving Graphs

Authors: : Laurent Bindschaedler, Jasmina Malicevic, et al

6.827 Paper Presentation

Presenter: Edmund Williams

April 2022

Table of Contents

Problem Definition

- Pattern: A desired connected subgraph with constraints to the number and connectivity of vertices allowed in the subgraph. The specification can also include rules to what labels each vertex can have.
- Match: A subgraph that meets the constraints given by the pattern.
- Problem: Given a graph G, find all subgraphs of G that are isomorphic to the desired pattern.
- Naive Solution: Enumerate through all possible subgraphs and test for ismorphism.
- Evolving Graphs: Most graphs are not static, it is undesirable to have recompute the this problem for only small changes to the input graph. Is there a way to solve this in an online/streaming fashion?

Problem Definition

- Pattern: A desired connected subgraph with constraints to the number and connectivity of vertices allowed in the subgraph. The specification can also include rules to what labels each vertex can have.
- Match: A subgraph that meets the constraints given by the pattern.
- Problem: Given a graph G, find all subgraphs of G that are isomorphic to the desired pattern.
- Naive Solution: Enumerate through all possible subgraphs and test for ismorphism.
- Evolving Graphs: Most graphs are not static, it is undesirable to have recompute the this problem for only small changes to the input graph. Is there a way to solve this in an online/streaming fashion?

Problem Definition

- Pattern: A desired connected subgraph with constraints to the number and connectivity of vertices allowed in the subgraph. The specification can also include rules to what labels each vertex can have.
- Match: A subgraph that meets the constraints given by the pattern.
- Problem: Given a graph G, find all subgraphs of G that are isomorphic to the desired pattern.
- Naive Solution: Enumerate through all possible subgraphs and test for ismorphism.
- Evolving Graphs: Most graphs are not static, it is undesirable to have recompute the this problem for only small changes to the input graph. Is there a way to solve this in an online/streaming fashion?

Problem Definition

- Pattern: A desired connected subgraph with constraints to the number and connectivity of vertices allowed in the subgraph. The specification can also include rules to what labels each vertex can have.
- Match: A subgraph that meets the constraints given by the pattern.
- Problem: Given a graph G, find all subgraphs of G that are isomorphic to the desired pattern.
- Naive Solution: Enumerate through all possible subgraphs and test for ismorphism.
- Evolving Graphs: Most graphs are not static, it is undesirable to have recompute the this problem for only small changes to the input graph. Is there a way to solve this in an online/streaming fashion?

Problem Definition

- Pattern: A desired connected subgraph with constraints to the number and connectivity of vertices allowed in the subgraph. The specification can also include rules to what labels each vertex can have.
- Match: A subgraph that meets the constraints given by the pattern.
- Problem: Given a graph G, find all subgraphs of G that are isomorphic to the desired pattern.
- Naive Solution: Enumerate through all possible subgraphs and test for ismorphism.
- Evolving Graphs: Most graphs are not static, it is undesirable to have recompute the this problem for only small changes to the input graph. Is there a way to solve this in an online/streaming fashion?

Problem Difficulties

- Exponential number of subgraphs: enumerating subgraphs is exponential in nature, is there an effective pruning method to know what branch of subgraphs truly need to be checked for a matches?
- Duplicate Matches: A subgraph

Competitors

- Fractal: Most performant architecture for general pattern mining for distributed graphs[1]
- Delta-Bigjoin: Only existing architecture for distributed graphs that can handle evolution, however it doesn't support general patterns[2]

[1] https://dl.acm.org/doi/10.1145/3299869.3319875

[2] https://arxiv.org/abs/1802.03760

Goal

- Make an architecture that can handle evolving graphs AND general patterns on a distributed graph.

Table of Contents

Big Parts

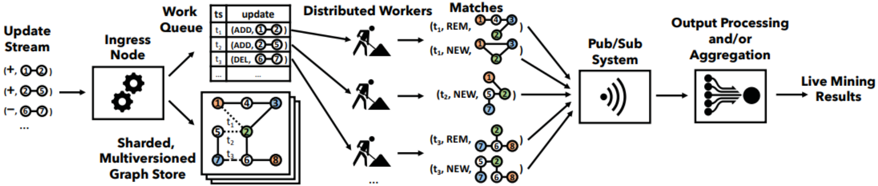

Figure 2. Tesseract architecture and graph keyword search example from Figure 1.

- Ingress Node: Takes in stream of graph updates. With that stream places tasks on Worker Queue and asks Graph Store to update the graph
- Pattern Mining Engine: This is essential part of what the paper implements
- Output Processing: Takes output streams from the workers and uses a publish/subscribe framework to pass this information to receiving clients.

Motivation/Intuition

Objective

- Incremental Updates. What work do we need to do to account for one change in the graph?
- Can we search strictly around the area of change?

Filter function

- Given a subgraph and pattern, is it possible to add more vertices/edges to the subgraph to find a new subgraph that will match the given pattern?
- Search from a starting vertex and incrementally add more vertices. If at any point the subgraph fails the filter then prune that branch of searching.

Motivation/Intuition

Objective

- Incremental Updates. What work do we need to do to account for one change in the graph?
- Can we search strictly around the area of change?

Filter function

- Given a subgraph and pattern, is it possible to add more vertices/edges to the subgraph to find a new subgraph that will match the given pattern?
- Search from a starting vertex and incrementally add more vertices. If at any point the subgraph fails the filter then prune that branch of searching.

Exploration Pruning

See Chalkboard for walk through of exploration

Duplicate Pruning

Problem: Duplicate matches

- Different paths of exploration can yield the same subgraph.
- Large overhead of computing duplicates for general patterns (cliques is a prime example)

Solution

- Ordering of vertices. Exploring along an edge will be pruned if the outgoing vertex's ordering comes before any vertex within the current subgraph.
- Guarantees only 1 instance of a match will be found as paths of exploration have to following the ordering of vertices and there is only 1 possible ordering of vertices if each vertex has a unique identifier.
- Also gives the benefit of additional pruning.

Duplicate Pruning

Problem: Duplicate matches

- Different paths of exploration can yield the same subgraph.
- Large overhead of computing duplicates for general patterns (cliques is a prime example)

Solution

- Ordering of vertices. Exploring along an edge will be pruned if the outgoing vertex's ordering comes before any vertex within the current subgraph.
- Guarantees only 1 instance of a match will be found as paths of exploration have to following the ordering of vertices and there is only 1 possible ordering of vertices if each vertex has a unique identifier.
- Also gives the benefit of additional pruning.

CanExpand Algorithm

- Assures the graph considered only includes updates during or before the current snapshot
- Incorporates vertex ordering to decide if subgraph should be expanded with vertex v

Algorithm 3: CAN_EXPAND

input : G data graph snapshot at timestamp ts
input : s subgraph
input : ts update timestamp
input : v new vertex to expand s

```
foreach edge (v, u) in G with u ∈ s do
    if TIMESTAMP(v, u) == ts and (v, u) < (s[0], s[1])
      then return false
found ← IS_NEIGHBOR(G, v, s[0]) or
  IS_NEIGHBOR(G, v, s[1])
foreach u in s[2 :] do
    // s[2:] excludes the update endpoints
    if not found and IS_NEIGHBOR(G, v, u) then
        found ← true
    else if found and u > v then return false
return true
```

Explore Algorithm

Algorithm 2: The EXPLORE Algorithm

input : G data graph snapshot at timestamp ts
input : ts update timestamp
input : s subgraph (initialized to the edge update)
input : c_{pre} continue pre-update (initialized to true)
input : c_{post} continue post-update (initialized to true)

function EXPLORE($G, ts, s, c_{pre}, c_{post}$) **is**
  foreach *neighbor* v *of* s *in* G **do**
    if CAN_EXPAND(G, ts, s, v) **then**
      $s' \leftarrow$ EXPAND(G, s, v)
      $(c'_{pre}, c'_{post}) \leftarrow$ DETECT_CHANGES($G, ts, s', c_{pre}, c_{post}$)
      if c'_{pre} **or** c'_{post} **then**
        EXPLORE($G, ts, s', c'_{pre}, c'_{post}$)

function DETECT_CHANGES($G, ts, s', c'_{pre}, c'_{post}$) **is**
  $s'_{pre} \leftarrow$ SUBGRAPH_AT_PREVIOUS_SNAPSHOT(G, s')
  if c'_{pre} ***and*** filter(s'_{pre}) **then**
    if IS_CONNECTED(s'_{pre}) **and** match(s'_{pre}) **then**
      EMIT (ts, REM, s'_{pre})
  else $c'_{pre} \leftarrow$ **false**
  if c'_{post} ***and*** filter(s') **then**
    if IS_CONNECTED(s') **and** match(s') **then**
      EMIT (ts, NEW, s')
  else $c'_{post} \leftarrow$ **false**
  return (c'_{pre}, c'_{post})

- Nuance to c_{pre}, c_{post}, this algorithm is search for matches before the update and after the update.
- Since there shouldn't be much change to the graph these searches should have similar explorations.
- To increase performance these two searches are run together (the various helper functions may not be cheap to redundantly compute)

Example Use Case

- User specified functions to interface with Tesseract
- Filter function is used to determine if the current subgraph doesn't meet criteria for making a potential match given the addition of more vertices/edges.
- Match function is used to specify what pattern to be mined for in the graph.

Algorithm 1: Examples of graph mining applications

```
algorithm graph_keyword_search
    function filter(s)
        return len(s) <= MAX and
          num_orange(s) <= 1 and num_green(s) <= 1
          and num_blue(s) <= 1
    function match(s)
        if num_green(s) != 1 or num_orange(s) != 1
          or num_blue(s) != 1 then return false
        foreach vertex v in s if color(v) == white do
            if IS_CONNECTED(s \ v) then return false
        return true
algorithm clique_mining
    function filter(s)
        return len(s) <= MAX and
        num_edges(s) == len(s)*(len(s)-1)/2
    function match(s)
        return true
```

Sharding/Snapshotting

- Gives timesteps to the evolution of the graph and partitions the stream of updates into discreet chunks
- Frequency of snapshotting is a tunable parameter, if too frequent there is an overhead of producing snapshots for each timestamp, if infrequent the memory volume of a set updates will no longer fit in cache.

Table of Contents

Evaluations

- Tested against Fractal to prove effectiveness of incrimental updating
- Tested against Delta-Bigjoin, a similar evolving graph framework, to compare runtime performance