Date / /
Page

1) Query - Single Server Mode.

*) If your query starts from one or a few starting vertices, use this

In this, the cluster elects one server to be master for that query.
All query computation takes place on query master.
Vertex and edge data are copied to the query master as needed.

2) Query - Distributed Mode

*) If your query starts from all or most vertices, use this mode.

In this, the server that received the query becomes the master.
Computation executes on all servers in parallel.
Accumulators are transferred across the cluster.

Date / /
Page

Page Rank:

A page has a high rank if the sum of the ranks of its backlinks is high.
(Backlinks: No. of web pages linking to webpage in focus).

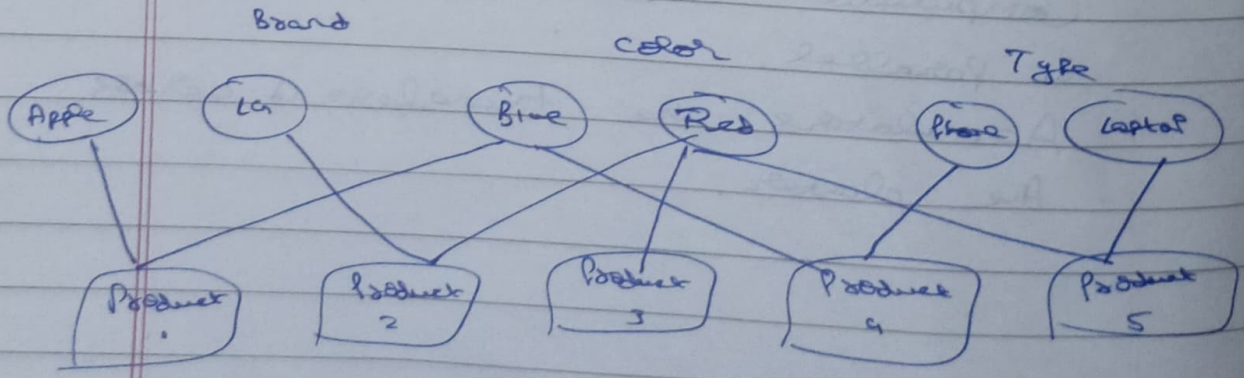
This covers both the case when a page has many backlinks and when a page has a few highly ranked backlinks.

→ Attribute as vertex?

It is beneficial to represent a colour (attribute) as a vertex type if you will frequently need to query for particular values of the property.

This way, the vertices act like an search index.

Ex: All the red products are connected to the Red vertex under Color type.



Date / /

Page

Accumulators

Accumulators are special type of variables that accumulate information about the graph during the traversal.

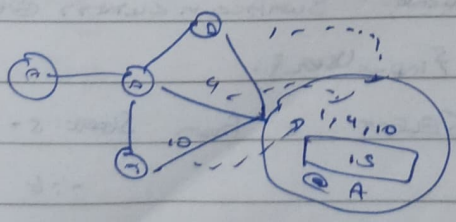
(i) Phase 1: Receives messages, which will (conceptually) be put to a bucket that belongs to the accumulator.

(ii) Phase 2: The accumulator will aggregate the messages it received based on its accumulator type. The aggregated value will become the accumulator's value, and its value can be accessed.

1) Local Accumulators

- Each selected vertex has its own accumulator.
- Local means per vertex. Each vertex does its own processing and considers what it can see / read / write.

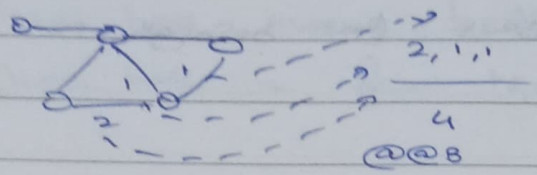
Ex: SumAccum@A;



2) Global Accumulators

- Started in globally visible to all Queries.
- All vertices and edges have access.

Ex: SumAccum @@ B;



Some Accumulators

1) SumAccum <int>
2) MaxAccum <int>
3) MinAccum <int>
4) AvgAccum

Ex. Accum Clause

Q) What is the age distribution of friends that were registered in 2018?

A) CREATE QUERY GetFriends (vertex<User> inputUser) FOR GRAPH Social {

MapAccum <int, SumAccum <int>> @@ ageMap;

Start = {inputUser};

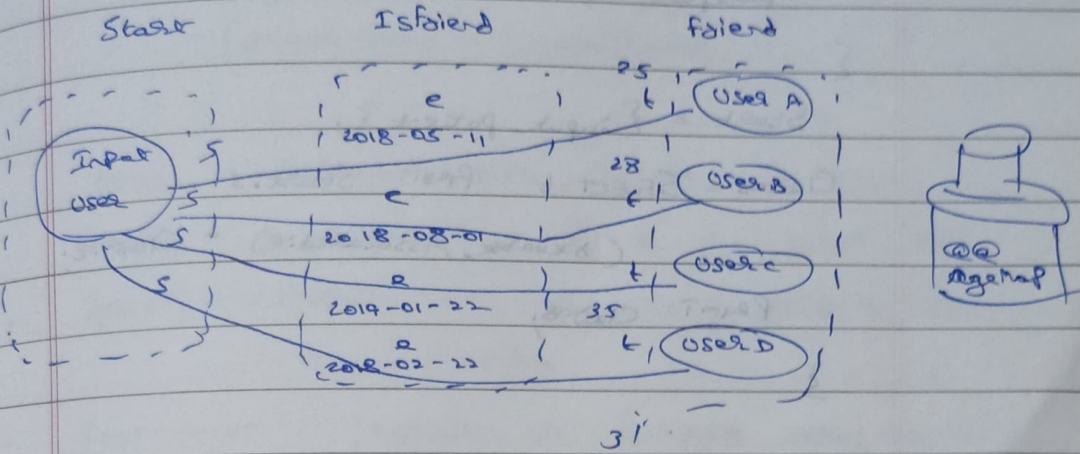
Friends = SELECT t from Start:s - (IsFriend:e) -:t

WHERE e.connectDt BETWEEN to_datetime ("2018-01-01") AND to_datetime ("2019-01-01")

ACCUM @@ageMap += (t.age/10 -> 1);

PRINT @@ ageMap;

}



}

Note: Design the traversal plan to avoid starting from the hub nodes.

→ By default 'As attribute' option is not enabled => ID value is not accessible from the query.

Therefore 'As attribute' needs to be checked when vertex ID values are used in the query logic for non-printing purposes such as string concatenation, value comparison.

Having it switched off is more memory saving.

Date / /
Page

Goal

Q) Find all the claims of a patient

A) CREATE QUERY GetClaims (vertex<User> input_patient) FOR GRAPH Social

{

Start = {input_patient};

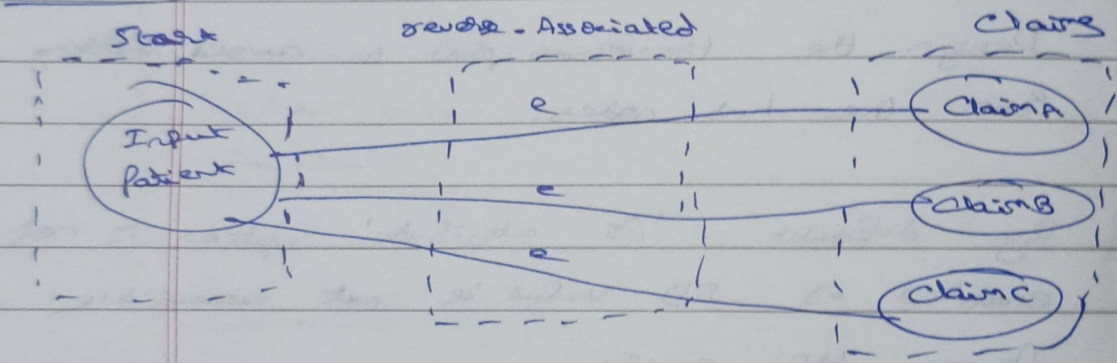
Claims = SELECT t FROM Start:s -

(reverse_Associated:e) - Claim:t;

PRINT Claims;

}

Visualization



Explanation

- Start is a 'vertex set' initialized by the input vertex input_patient.
- From clause finds edges which match the pattern: Source vertex is a Start, edge type is reverse_Associated, and target vertex is restricted to User type.
- s, e, t are aliases.
- Claims is a new vertex set equal to t.

Date / /
Page

bilt 10/10 on Student Notebooks

d) resultSet = SELECT vSet
FROM (edgeSet | vertexSet)
[WhereClause] [accumClause]
[postAccumClause] [havingClause]
[orderClause] [limitClause];

- FROM: Select active vertices & edges
- WHERE: conditionally filter the active sets.
- ACCUM: iterate on edge set; compute with accumulators
- POST-ACCUM: iterate on vertex sets; compute with accumulators
- HAVING: Conditionally filter the result set
- ORDER BY: sort
- LIMIT: max number of items
- SELECT: result from source & target set

→ What is within 'PRINT' is returned as JSON.

result =
Ex: SELECT s FROM Start:s;

Note: TigerGraph supports full ACID transactions with sequential consistency.

→ Main ways to interact with the Graph solutions are: REST endpoints, GraphStudio and GSQL Shell.

Date / /
Page

Data loading:

If you have defined a graph schema, you can load data into the graph store. Loading data is a 2-step process:

1) Define a loading job, in which you use data loading statements to specify how values in the data sources are extracted, transferred and loaded to their destinations.
2) Run the loading job. Supply parameters if any.

Note: GSQL code must be entered as a single line (unless the multi-line mode is used) within gsql shell

(1) ls:
To check existing databases

(2) DROP ALL:
Delete all database data, schema and all related definitions

Note: You can also run GSQL commands from a Linux shell by using 'gsql' followed by the 'command' enclosed in single quotes.
Ex: gsql 'ls'

Date / /
Page

The first step in creating a GSQL graph is to define its schema. GSQL provides DDL commands for the same.

(3) Creating a vertex type

Ex: CREATE VERTEX person (
PRIMARY_ID name STRING,
name STRING, age INT,
gender STRING, state STRING
)

(4) Create an Edge Type

It can be directed or undirected.

Ex: CREATE UNDIRECTED EDGE friendship
(FROM person, TO person, connect_day DATETIME)
: Property

(5) Creating a Graph

Here, we list the vertex types and edge types that we want to include in this graph.

Ex: CREATE GRAPH social (person, friendship)
: name of graph

(6) SQ2 command to define a Loading Job

Ex:

```
USE GRAPH social
BEGIN
CREATE LOADING JOB load_social
  FOR GRAPH social {

DEFINE FILENAME file1 = "/home/tigergraph/person.csv";
DEFINE FILENAME file2 = "friendship.csv";

LOAD file1 TO VERTEX person VALUES
  ($"name", $"name", $"age", $"gender",
  $"State"), USING header = "true",
    separator = ",";

LOAD file2 TO EDGE friendship VALUES
  ($0, $1, $2) USING header = "true",
  separator = ",";
  }
END
```

For Primary id

for name attribute

Source column Position

(7) Run the Loading Job

```
>>> RUN LOADING JOB load_social
```

Date / /
Page

→ Parameterized queries let you traverse the graph from one vertex set to an adjacent set of vertices, again and again, performing computations along the way, with built-in parallel execution and handy aggregation operations. You can even call one query from another.

Ex: Simple 1-Hop Query

>>> USE GRAPH Social
SET syntax_version = "v2"

CREATE QUERY hello (VERTEX<person> p) {
  Start = {p};
  Result = SELECT tgt
    FROM Start:s -(friendship:e)
      -person:tgt;
  PRINT Result;
}

undirected edge

alias name

// Result will return the target vertex set

You can Paste this query in a file with .gsql extension, Say, hello.gsql
To run this, use >> @hello.gsql
Query is now in the Catalog.

Now, we have to install this:

>> INSTALL QUERY hello

The installation will generate machine instructions and a REST endpoint and helps with faster query execution when executed multiple times.

Running the query:

>>> RUN QUERY hello("rom")

You can run queries without installing too (Anonymous queries)

Example 2 : hello2.gsql

>>> USE GRAPH Social

CREATE QUERY hello2 (VERTEX<person> p) {

OrAccum @visited = false;

AvgAccum @@avgAge;

Start = {p};

FirstNeighbors = SELECT tgt

FROM Start:s - (friendship:e) - person:tgt

ACCUM tgt.@visited += true,

s.@visited += true.

)

SecondNeighbors = SELECT tgt
FROM FirstNeighbors -(:e)- :tgt
WHERE tgt.@visited == false
POST-ACCUM @@avgAge += tgt.age;

PRINT SecondNeighbors;
PRINT @@avgAge;
}

INSTALL QUERY hello2
RUN QUERY hello2("Tom")

→ The above query finds the average age of all neighbors 2-hops away from parameterized input person.

→ To declare a local accumulator, we prefix an identifier name with a single "@" symbol.

→ The identifier for a global accumulator begins with two "@@"s.

→ The += operator within an ACCUM clause means that for each edge matching the FROM clause pattern, we accumulate the right-hand-side expression (1 in this case) to the left-hand accumulator (tgt.@visited and s.@visited)

→ If Edge type is omitted, (like in (A)), it is interpreted as ALL types.

* cURL is a command-line tool for transferring data using various network protocols. It stands for client URL.

Date / /
Page

→ POST_ACCUM traverses the vertex sets instead of the edge sets, guaranteeing that we do not double-count any vertices.

→ >>> SELECT count() FROM person

→ >>> SELECT count() FROM person-(Friendship)->person
edge pattern

→ >>> SELECT * FROM person WHERE primary_id == "Tom"

→ >>> SELECT * FROM person-(Friendship)->person WHERE from_id == "Tom"

#) PATTERN MATCHING TUTORIAL

→ A graph pattern is a traversal trace on the graph schema. A pattern can be a linear trace or non-linear (tree, circle, etc.)

Ex: Person-(Friendship)-Person-(Friendship)-Person

Or, use *2 to denote the two consecutive FriendshipEdges.

Person-(Friendship*2)-Person

• Pattern matching is the process of finding subgraphs in a data graph that conform to a given query pattern.

Date / /
Page

bill 10/10

Pattern matching queries support nested queries.

Pattern example:

Person:p - ((LIKES >|e HAS_CREATOR):e)
- Message:m

Indicates traversing 'HAS_CREATOR' edge of outgoing 'LIKES' edge in same query expression

'|' is alternation separator

EDGES

→ '_' is a wildcard meaning any edge type.

→ The '()' empty parens means any edge, directed or undirected.

VERTICES

→ Use '_' or 'ANY' or omit the vertex type to indicate any vertex type. If you omit the vertex type, you must provide a vertex alias.

Examples:

1) SELECT t FROM Person:s - (IS_LOCATED_IN >:e) - City:t

2) SELECT t FROM (Post|Comment):s - (IS_LOCATED_IN >:e) - Country:t

3) SELECT s FROM :s - (IS_LOCATED_IN >:e) - Country:t

4) SELECT t FROM - (IS_LOCATED_IN >:e) - Country:t

Date / /
Page

Examples of 1-Hop patterns

1) FROM X:x - (E2>:e2) - Y:y

Right directed edge e2 binds to the source of E2; y binds to the target of E2.

2) FROM X:x - (_:e) - Y:y

Any undirected edge between a member of x and a member of Y.

3) FROM X:x -(_>:e) - Y:y

Any right directed edge with source in x and target in y.

4) FROM X:x -((<_|_):e) - Y:y

Any left directed or any undirected

5) FROM X:x - ((E1|E2>|<E3):e) -Y:y

Any one of the three edge patterns

6) FROM X:x - (): Y:y

- any edge (directed or undirected)
- Same as (<_|_>|_)

Date / /
Page

→ To run overpowered queries without restriction:
INTERPRET QUERY (s) FOR GRAPH
graphstore SYNTAX v2 { <query body> }

→ Increasing Query timeout threshold:
SET query_timeout = 60000
// unit is 1 millisecond

(Q) Find persons who know the person named "Victor Akinwande" and display top 3 things such persons.

A) USE GRAPH graph_name

INTERPRET QUERY (s) SYNTAX v2 {
FOR p IN SELECT p
FROM persons = (traverse) - Person: p
WHERE p.first_name == "Victor" AND
p.last_name == "Akinwande"
ORDER BY p.birthday ASC
LIMIT 3;

PRINT persons[p.id, p.first_name, p.last_name,
p.birthday];

}

Date / /
Page

Q) Find total number of comments and posts liked by Viktor. A Person can reach comments or Posts via a directed edge LIKES.

A) USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {
SumAccum<int> @comment_cnt = 0;
SumAccum<int> @post_cnt = 0;

Result = SELECT s
FROM Person:s -(Likes>)- :tgt
WHERE s.first_name == "Viktor" AND
s.lastname == "Akhiezer"
ACCUM CASE WHEN tgt.type == "Comment" THEN
s.@comment_cnt += 1
WHEN tgt.type == "Post" THEN
s.@post_cnt += 1
END;

PRINT Result[Result.@comment_cnt,
Result.@post_cnt];
}

...

→ From Person:tgt - (<Has_Creator|Likes>) - (Comment|Post)
:src

Here, HAS_CREATOR edge type starts from

Comment / Post, and LIKES edge type starts from Person.

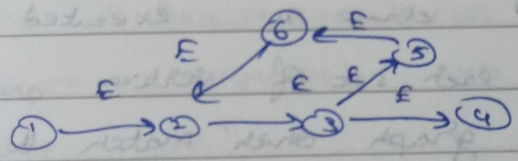
→ Repeating 1-Hop Pattern:

(i) FROM X:x - (E*2..) - Y:y
At least chain of 2 E edges

(ii) FROM X:x - (E>*..3) - Y:y
Chain of between 0 and 3 E edges

Note: Only the shortest matching occurrences are selected.

Ex:



POS 1 - (E>*) - 4,

- 1 → 2 → 3 → 4 ✓
- 1 → 2 → 3 → 5 → 6 → 2 → 3 → 4 ✗

Q) "People who bought this product also bought this other product"

A) FROM This_Product:p - (<Bought:b1)
- Customer:c - (Bought>:b2) - Product:p2
WHERE p2 != p

Date / /
Page

2-Hop pattern

FROM X:x - (E1:e1) - Y:y - (E2:e2) - Z:z

(OR)

FROM X:x - (E1.E2>) - Z:z

↓

Other intermediate vertex set is not required. Vertex should not be only of a specific type. '.' means any vertex in b/w.

Note: If a pattern has a Kleene star (*) to repeat an edge), GSQL pattern matching selects only the shortest paths which match the pattern.

→ ACCUM clause is executed once (in parallel) for each set of vertices and edges in the graph which match the pattern and constraints given in the FROM and WHERE clauses.

→ Use POST-ACCUM clause to perform a second, different round of computations on the results of your pattern matching.

Note.

→ The ACCUM clause executes for each full path that matches the pattern in the FROM clause.

In contrast, the POST-ACCUM clause executes for each vertex in one vertex set.

Date / /
Page

→ 1 POST-ACCUM clause can use only 1 alias.

Q) Find Viktor Alvarez's liked messages whose author's last names begin with S. Find these authors also count.

A) USE GRAPH ldbc_snb

INTERPRET QUERY () SYNTAX v2 {

SumAccum<int> @@cnt;

F = SELECT t
FROM :s - (Likes>:e) - :msg - (Has_Creator>) - :t

WHERE s.first_name == "Viktor" AND
s.last_name == "Alvarez" AND t.last_name
LIKE "S%";

Always = SELECT p
FROM Person:p - (Study_At>) - :u -
(<Study_At) - F:f
WHERE s != p
Per (p)
POST-ACCUM @@cnt += 1;

PRINT @@cnt;

}

→ The PER clause specifies a list of vertex aliases, which are used to group the rows in the match table, one group per distinct value of the alias or of the alias list.

→ Conjunctive Pattern Matching

Logical AND of 2 or more patterns that can be naturally joined on a common alias.

Ex. SELECT FROM

Person: p - (KNOWS) - :tgt,
Post: s - (<LIKES) - :tgt;

→ Data Modification

Pattern Matching GSQL supports Insert, Update and Delete operations.

Data modification happens in 2 ways using GSQL:

1) Top level. The statement does not need to within any other statement.
2) Within a SELECT query statement. The FROM-WHERE clause defines a match table, and the data modification is performed based on the vertex and edge information in the match table.

The GSQL Specifications calls those within-SELECT Statements DML-Sub Statements.

- Data Modification in interpreted mode is not available yet.
- SELECT queries with data modification may only have one POST-Accum clause.

INSERT

Q) Create a Person vertex, whose name is Tiger Woods. Next, find Viktors favorite 2012 Posts authors, whose last name is prefixed with S. Finally, insert knows edges connecting Tiger Woods with Viktor's favorite authors.

A) CREATE QUERY insert_edge_and_vertex() SYNTAX v2 {

AA) INSERT INTO Person VALUES (10000000, "Tiger", "Woods", "M", -, -, -, -, -);

R= SELECT t
FROM Person:s -(Likes>)- :msg -(Has_Creator)- Person:t
WHERE s.firstName == "Viktor" AND s.lastName == "Antiev"
AND t.lastName LIKE "S%" AND year(msg.creation_date) == 2012

PEP (5, 4)
ACCUM

BB) INSERT INTO Knows VALUES (10000000, 6,
to_datetime("2020-06-01"));

PRINT R [R.id, R.first_name, R.last_name];
}

'AA' is called a top-level statement
'BB' is DML-sub INSERT statement.

UPDATE

Top-level UPDATE statements not supported yet.

Vertex attributes can only be updated in POST-ACCUM clause, and edge attributes can only be updated in the ACCUM clause

To perform within-SELECT updates, the FROM pattern can only be a single hop, fixed length pattern.

Date / /
Page

8) For all Knows edges that connect Tiger Woods and his friends whose last_name begins with "S", update the edge creation_date to "2020-10-01". Also, for the person vertex (Tiger Woods) update the vertex's creation_date and language he speaks.

A) CREATE QUERY update_Knows_ts () SYNTAX v2 {

R = SELECT p FROM Person:p
WHERE p.first_name == "Tiger" AND p.last_name == "Woods"
POST-ACCUM
// update simple base type attribute
p.creation_date = to_datetime("2020-6-1");
// update collection-type attribute
p.Speaks = ("english", "golf");

R = SELECT t
FROM Person:s - (Knows:e) - :t
WHERE s.first_name == "Tiger" and
s.last_name == "Woods"

ACCUM e.creation_date =
to_datetime("2020-10-01");

}

Date / /
Page

DELETE

- Edges can only be deleted in the Accum clause.
- For best performance, vertices should be deleted in the Post-Accum clause.

Q) Delete vertex Tiger Woods and its knows edges.

A) CREATE QUERY delete_edge_and_vertex()
SYNTAX v2 {

```
R = SELECT t
    FROM Person:s -(knows:e)- Person:t
WHERE s.first_name == "Tiger" AND s.last_name == "Woods"
    Accum
        // delete edges
        DELETE (e)
    Post-Accum DELETE (s); // delete the vertex

PRINT R[R.id, R.first_name, R.last_name];
}
```

→ For multi-line statements in GSQL shell, use the keywords BEGIN
...
END.

Date / /
Page

CREATE VERTEX Syntax

CREATE VERTEX Vertex_Type_Name "("
primary_id_name type
[","attribute_name type [DEFAULT default_value]] * ")"
[WITH [STATS = "none" | "outdegree_by_edgetype"]
[primary_id_as_attribute = "true"]]

Eg: CREATE VERTEX Movie (PRIMARY_ID id UINT, name STRING, year UINT)
WITH primary_id_as_attribute = "true"

→ GSQL creates a hash index on primary id with O(1) time complexity.

(or)

CREATE VERTEX Movie (id UINT PRIMARY KEY, name STRING, year UINT)

Composite keys

→ CREATE VERTEX Movie (id UINT, title STRING, year UINT, PRIMARY KEY (title, year, id))

→ ## Creating Edges:

CREATE DIRECTED_EDGE Member_Of
(FROM Person, TO Org | FROM Org, TO Org,
joined DATETIME) ... [WITH REVERSE_EDGE = "rev_name"]

It can be UNDIRECTED also

→ Creating a graph with all vertex & edge types

Ex: CREATE GRAPH graph_name(*)

→ altering vertex attribute:

Ex: ALTER VERTEX Company ADD ATTRIBUTE (industry STRING, marketCap DOUBLE)

For above,
you have to create schema job
and Run the schema job

★ Centrality Algorithms:

Centrality algorithms find the important vertices in a graph based on their connections with other vertices.

A centrality algorithm usually assigns scores to vertices or edges based on how close they are to the center of connection-based activity.

The definition of closeness and what constitutes the center of connection-based activity depend on the algorithm.

Ex: Degree centrality

Date / /
Page

bilt 10/10 Student Notebooks

* Classification Algorithms

They assign a label (class name) to a vertex based on it satisfying the established conditions for membership in that class.

This differs from * community (or clustering) algorithms in which there are no pre-established conditions for membership.

* Node Embedding Algorithms

Node embeddings are vector representations of properties of vertices in a graph. These vectors can be used for machine learning.

* Pathfinding Algorithms

It finds the best path (s) between two vertices or among a set of vertices according to a particular rule.

* Similarity Algorithms

Similarity Algorithms assign a score to a pair of vertices based on how similar they are. This usually involves seeing if they have the same or similar neighborhoods. This section also includes distance algorithm, as distance is the inverse of similarity.

Date / /
Page

#) Topological Link Prediction Algorithms

These algorithms determine the closeness score of a pair of vertices. This score can then be used to make predictions about the relationship between the two vertices in the pair.

Ex: Adamic Adair, Common Neighbors