Relational Algebraic Graph Algorithms

Hyu Chan Park
Department of Computer Engineering, Korea Maritime and Ocean University, Korea

Summary
This paper formalizes new graph representation and graph algorithms based on the well-developed relational database theory. In this formalization, graphs are represented in the form of relations which can be visualized as relational tables. Each vertex and edge of a graph is represented as a tuple in the tables. Graph algorithms are also defined in terms of relational algebraic operations such as projection, selection, and join. They can be implemented with the database language, SQL. This database implementation has many advantages compared with traditional approaches. Very large amount of graphs, for example, can be efficiently managed and concurrently shared among users by virtue of the capability of databases.

Key words:
Graph algorithm, Relational Algebra, Database

1. Introduction

Graphs are most powerful methodology to solve the real world problems. A graph, G, consists of two sets: a finite, nonempty set of vertices, and a finite, possibly empty set of edges. V(G) and E(G) represent the sets of vertices and edges of G, respectively. We may write G = (V, E) to represent a graph [1].

While several representations for graphs are possible, adjacency matrix, adjacency list, and adjacency multilist are most commonly used in practical applications [1]. Although the representations have been successfully applied to most of applications, they have some limitations because of their in-memory property. First of all, the lifetime of graph objects are all transient, but not persistent. The representation of graphs are effective only during the program run-time and vanishes when the program terminates. Next, some applications require concurrent access to graphs by several users. In such environment, interactions of concurrent updates may result in inconsistent graphs. Finally, the size of graphs may be limited by the size of memory. In such environment, part of graphs must be repeatedly saved to and retrieved from files.

To cope with the limitations, this paper proposes a relational formalization of graph representation and algorithms. In the formalism, graphs are represented as relations based on the concept of the relational data model [2, 3]. These relations, in turn, can be visualized in the form of relational tables and then saved in a relational database. Graph algorithms are defined in terms of relational algebraic operations such as projection, selection, and join. These algorithms, also in turn, implemented with relational database languages such as SQL [4, 5]. This database implementation has many advantages compared with traditional approaches. Very large amount of graphs, for example, can be efficiently managed and concurrently shared among users by virtue of the capability of databases.

2. Relational Algebra

Cartesian product of domains D1, D2, ⋯, Dn, written D1 × D2 × ⋯ × Dn, is a set of n-tuples <v1, v2, ⋯, vn> such that v1 is in D1, v2 is in D2, and so on. Relation is any subset of the cartesian product of one or more domains, and each element of the relation is called tuple. Relation can be represented as a table where each row is tuple and each column has a distinct name called attribute. Each attribute has an associated domain. A relation R with a set of attributes A = {A1, A2, ⋯, An} is denoted by R[A] or R[A1, A2, ⋯, An]. Let t be a tuple in R[A]. Then the part of t corresponding to a set of attributes $X \subseteq A$ is denoted by t[X] [5].

There is a family of operations usually associated with relations. They can be coded by using algebraic notations, called relational algebra. Fundamental operations in relational algebra are projection(π), selection(σ), union($\cup$), difference(-), and cartesian product(×). In addition to the five fundamental operations, there are some other useful operations, such as intersection($\cap$), natural join($\bowtie$), theta join($\bowtie_\theta$), and aggregation(G), that can be defined in terms of the fundamental operations [5]. Some of the operations are explained in more detail below.

1) Projection (π): projection of a relation R, denoted $\pi_X(R)$, chooses a subset of the columns. Let R be a relation on a set of attributes A = {A1, A2, ⋯, An} and X is a subset of A. Then projection $\pi_X(R)$ is obtained by dropping columns with attributes not in the set X and removing duplicate tuples in what remains.

$$\pi_X(R) = \{t[X] \mid t \in R\}$$

Generalized-projection operation extends the projection operation by allowing arithmetic functions to be used in the projection list.

$$\pi_{F1, F2, \cdots, Fn}(R)$$

Manuscript received January 5, 2017
Manuscript revised January 20, 2017

where, each of F1, F2, ⋯, Fn are arithmetic expressions involving constants and attributes in the schema of R [5].

2) Selection (σ): selection of relation R, denoted σF(R), is a subset of tuples of R that satisfy the formula F.

$$\sigma F(R) = \{t \mid F(t) \wedge t \in R\}$$

3) Natural join (⋈): natural join of two relations R and S, denoted R⋈S, combines two relations on their common attributes. Let R[XW] and S[YW] be two relations where X, Y and W are disjoint sets of attributes.

$$R \bowtie S = \{t[XYW] \mid t[XW] \in R \wedge t[YW] \in S\}$$

4) Theta join ($\bowtie_\theta$): theta join of two relations R[X] and T[Z], denoted $R \bowtie_\theta T$, is the subset of tuples of cartesian product R×T that satisfy the formula θ.

$$R \bowtie_\theta T = \{t \mid \theta(t) \wedge t[X] \in R \wedge t[Z] \in T\} = \sigma_\theta(R \times T)$$

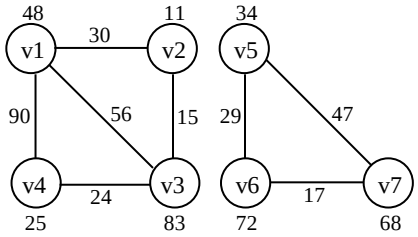
5) Aggregation (G): the general form of aggregation operation G is as follows.

$$_{G1,G2,\cdots,Gn}\mathrm{G}_{F1(A1),\,F2(A2),\cdots,\,Fm(Am)}(R)$$

where, G1, G2, ⋯, Gn constitute a list of attributes on which to group, each Fi is an aggregate function, and each Ai is an attribute name. The meaning of the operation is defined as follows. The tuples in R are partitioned into groups such that all tuples in a group have the same values for G1, G2, ⋯, Gn, and tuples in different groups have different values for G1, G2, ⋯, Gn. Thus the groups can be identified by the values of attributes G1, G2, ⋯, Gn. For each group (g1, g2, ⋯, gn), the result has a tuple (g1, g2, ⋯, gn, a1, a2, ⋯, am) where, for each i, ai is the result of applying the aggregate function Fi on the multiset of values for attribute Ai in the group [5].

3. Relational Algebraic Graph Algorithms

This section proposes a graph representation based on the relational data model, and then proposes graph algorithms based on the relational algebra. In the relational graph representation, vertices of a graph are represented as the vertex relation V[vid, v_attr], and edges are represented as the edge relation E[eid, vid1, vid2, e_ttr].
These relations, in turn, can be visualized in the form of tables. The graph in Fig. 1(a), for example, is visualized as the tables in Fig. 1(b). The column 'vid' of the vertex table 'V' represents the identifier of each vertex. The column 'eid' of the edge table 'E' represents the identifier of each edge, and the column 'vid1' and 'vid2' represent vertices. The pair (v1, v2), where v1 is a value of 'vid1' and v2 is a value of 'vid2', represents an edge connecting the vertex v1 and v2. The pair (v1, v2) is unordered for undirected graphs, and the pair <v1, v2> is ordered for directed graphs. We may consider other columns such as 'v_attr' for the attribute of vertices and 'e_attr' for the attribute of edges. They depend on the characteristics of graphs to be modeled for a given problem.



(a) graph

V

vid	v_attr
v1	48
v2	11
v3	83
v4	25
v5	34
v6	72
v7	68

E

eid	vid1	vid2	e_attr
e1	v1	v2	30
e2	v2	v3	15
e3	v3	v4	24
e4	v4	v1	90
e5	v1	v3	56
e6	v5	v6	29
e7	v6	v7	17
e8	v7	v5	47

(b) relational tables

Fig. 1 Graph and its relational tables

Graph operations and algorithms may depend on the representation of graphs. Under the relational representation of graphs, these are defined in terms of relational algebraic operations such as projection, selection, and selection. We only consider the vertex identifier 'vid' of the vertex table 'V', and 'vid1' and 'vid2' of the edge table 'E' in the following algorithms. The core of algorithms may be similar in spite of extra columns.
There may be a variety of operations to retrieve information from graphs. Some basic ones are defined in the simple relational algebra as the followings.

1) Number of vertices: $\pi_{count(*)}V$

2) Number of edges: $\pi_{count(*)}E$

3) Vertices adjacent with a vertex v: $\pi_{vid2}(\sigma_{vid1=v(E)}) \cup \pi_{vid1}(\sigma_{vid2=v}(E))$

4) Vertices adjacent with a vertex set S: $\pi_{vid2}(\sigma_{vid1\in S}(E)) \cup \pi_{vid1}(\sigma_{vid2\in S}(E))$

5) Vertices adjacent to a vertex v for digraphs: $\pi_{vid1}(\sigma_{vid2=v}(E))$

6) Vertices adjacent from a vertex v for digraphs: $\pi_{vid2}(\sigma_{vid1=v}(E))$

7) Edges incident with a vertex v: $\sigma_{vid1=v \vee vid2=v}(E)$

8) Degree of a vertex v: $\pi_{count(*)}(\sigma_{vid1=v \vee vid2=v}(E))$

Although there are many graph operations which result in new graphs [6], some of such operations can be defined in the relational algebra as the followings.

9) Complement of a graph G has V(G) as its vertex set, but two vertices are adjacent in if and only if they are not adjacent in G.

V' = V
E' = (V × V) - E

10) Union G = G1 ∪ G2.

V = V1 ∪ V2
E = E1 ∪ E2

11) Join G = G1 + G2 consists of G1 ∪ G2 and all edges joining V1 with V2.

V = V1 ∪ V2
E = E1 ∪ E2 ∪ (V1 × V2)

12) Product G = G1 × G2. Consider any two vertices u = (u1, u2) and v = (v1, v2) in V = V1 × V2. Then u and v are adjacent in G1 × G2 whenever [u1 = v1 and u2 adj v2] or [u2 = v2 and u1 adj v1].

V[vid1, vid2] = V1 × V2
/ where, a tuple <vid1, vid2> represents a vertex */*
E[vid11, vid12, vid21, vid22] = V × V
/ where, a tuple <vid11, vid12, vid21, vid22> represents an edge connecting the vertex <vid11, vid12> and <vid21, vid22> */*
E = $((\sigma_{vid11=vid21}E) \bowtie_{vid12=vid1 \wedge vid22=vid2} E2)$ */* [u1 = v1 and u2 adj v2] */*
∪ $((\sigma_{vid12=vid22}E) \bowtie_{vid11=vid1 \wedge vid21=vid2} E1)$ */* [u2 = v2 and u1 adj v1] */*

13) Composition G = G1[G2] also has V = V1 × V2 as its vertex set, and u = (u1, u2) is adjacent with v = (v1, v2) whenever [u1 *adj* v1] or [u1 = v1 and u2 *adj* v2].

V[vid1, vid2] = V1 × V2
E[vid11, vid12, vid21, vid22] = V × V
E = $(E \bowtie_{vid11=vid1 \wedge vid21=vid2} E1)$ */* [u1 adj v1] */*
∪ $((\sigma_{vid11=vid21}E) \bowtie_{vid12=vid1 \wedge vid22=vid2} E2)$ */* [u1 = v1 and u2 adj v2] */*

The followings are fundamental graph algorithms defined with relational algebraic operations and C-like programming structures.

14) Breadth first search algorithm [1] is defined as the following relational algebraic algorithm.

```
Algorithm breadth_first_search(v)
{
  S[vid] = {v}; /* starting vertex */
  print S; Visited[vid] = S;
  while (V - Visited != ∅) { /* exists not visited vertices */
    S = π_vid2(σ_vid1∈S(E)) ∪ π_vid1(σ_vid2∈S(E)) /* next vertices */
    S = S - Visited; /* vertices not yet visited */
    print S; Visited = Visited ∪ S;
  }
}
```

15) Minimum cost spanning tree is defined as the following, where the Kruskal's algorithm is used [7].

```
Algorithm minimum_cost_spanning_tree()
{ /* Emst[vid1, vid2, ecost] contains the edges of the minimum spanning tree */
  Emst[vid1, vid2, ecost] = {}
  Group[gid, vid] = π_vid,vid V /* each vertex is in different group, group is used for cycle-test */
  while ((π_count(*)Emst < π_count(*)V - 1) && π_count(*)E > 0) {
    <v1, v2, mincost> = (π_min(ecost)E) ⋈ E /* least cost edge */
    E = E - {<v1, v2, mincost>} /* delete it from the edge table */
    <g1> = π_gid(σ_vid=v1 Group) /* group of v1 */
    <g2> = π_gid(σ_vid=v2 Group) /* group of v2 */
    if (<g1> != <g2>) { /* if the least cost edge does not create a cycle */
      Emst = Emst ∪ {<v1, v2, mincost>}
      /* change the group of v2 with the group of v1 */
      Group = Group - {<g2, v2>} ∪ {<g1, v2>}
    }
    if (π_count(*)Emst < π_count(*)V - 1)
      print("No spanning tree");
  }
}
```

16) Single source all destinations shortest path for digraphs is defined as the following, where the Dijstra's algorithm is adapted [1].

```
Algorithm shortest_path(v)
{ /* D[vid, cost] contains the shortest distance to every vertex */
```

$n = \pi_{count(*)}(V)$ /* *number of vertices* */
$D[vid, cost] = \pi_{vid,\infty}(\sigma_{vid!=v}V) \cup \{<v, 0>\}$ /* *initial distance* ∞ */
$F[vid] = \{\}$ /* *found vertices* */
for (i = 0; i < n-1; i++) {
$<mincost> = \pi_{min(cost)}(D - (D \bowtie F))$ /* *smallest not yet checked* */
$<minvid> = \pi_{vid}(\sigma_{cost=mincost}D)$
$Dnew[vid, cost] = \pi_{vid2,ecost+mincost}(\sigma_{vid1=minvid} E)$ /* *new distances* */
$D = {}_{vid}G_{min(cost)}(D \cup Dnew)$ /* *select lower distance* */
$F = F \cup \{<minvid>\}$
}
}

4. Implementation of Graph Algorithms

In general, relational formalization of graphs and algorithms can be easily implemented on top of relational database system. The relations representing a graph, which can be visualized in the form of tables, are saved in and retrieved from a relational database. The relational algebraic algorithms on the graph, which is defined in terms of relational algebraic operations, can be implemented with the database language, SQL [4]. For example, the vertices adjacent on a vertex v, defined in $\pi_{vid2}(\sigma_{vid1=v}(E)) \cup \pi_{vid1}(\sigma_{vid2=v}(E))$, can be implemented as the following simple SQL statement.

```
select vid2 from E where vid1 = v
union
select vid1 from E where vid2 = v;
```

The relational algebraic algorithms with programming language features can be implemented with the embedded SQL, which mixes programming language with SQL statements [5]. Actually, the proposed graph algorithms are implemented on Oracle database with database language such as SQL and embedded SQL. Fig. 2 shows an implementation of the shortest path algorithm.

```
void shortest_path(v)
{
  int i, n;
  int mincost, minvid;

  /* connect to the graph database */
  EXEC SQL CONNECT :username

  /* calculate the number of vertices */
  EXEC SQL select count(*) into :n from V;

  /* create the distance table D and initialize */
  EXEC SQL create table D (vid int, cost int);
  EXEC SQL insert into D values(:v, 0);

  /* create the found table F */
  EXEC SQL create table F (vid int);

  for (i = 0; i < n - 1; i++) {
    /* find smallest distance vertex among not yet found */
    EXEC SQL create table D1 (vid, cost)
          as (select vid, cost from D) minus
            (select vid, cost from D, F where D.vid = F.vid);
    EXEC SQL select min(cost) into :mincost
            from D1;
    EXEC SQL select vid into :minvid from D1
            where cost = :mincost;

    /* recalculate new distances */
    EXEC SQL create table Dnew (vid int, cost int)
          as (select vid2, cost + :mincost
            from E where vid1 = :minvid;

    /* select smaller distance */
    EXEC SQL create table D2 (vid, cost)
          as (select * from D) union
            (select * from Dnew);
    EXEC SQL create table D (vid, cost)
          as (select vid, min(cost) from D2
            group by vid);

    /* add the vertex into the found table F */
    EXEC SQL insert into F values(:minvid);
  }
}
```

Fig. 2 Implementation of shortest path algorithm

The implementation was conducted on several experiments. During the experiments, graphs are generated synthetically according to the parameters, such as number of vertices and edges. Fig. 3 shows an experimental result of the shortest path algorithm, in which the number of vertices is fixed with 1,000. And the number of edges varies from 50,000 to 1,000,000 that is, from 1% of complete graph to 100%. As shown in the figure, the run time has some

overhead for the cases of lower number of edges, but it has more benefit for the cases of higher number of edges. For example, the run time increases from 90 to 140, that is only 1.6 times, even though the number of edges increase from 100,000 to 500,000, that is 5 times. This benefit is due to the power of database.

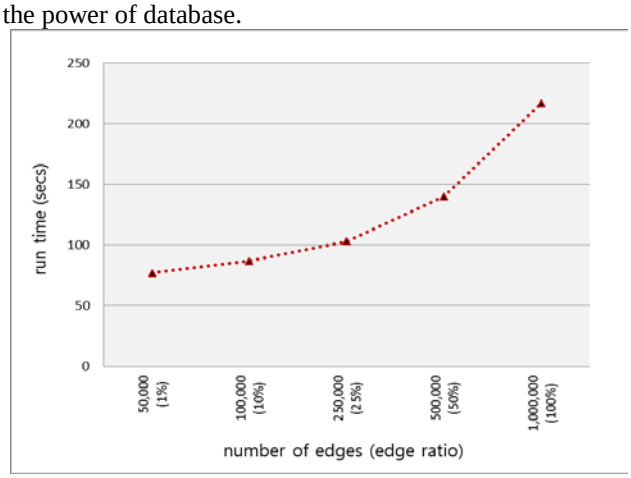


Fig. 3 Experimental result for short path algorithm

5. Conclusions

This paper showed graph algorithms can be modeled and implemented on top of relational database. It is based on the well-developed relational data model and relational algebra. Graph is first modeled in the form of relations of the relational data model. Graph algorithms are then formalized in terms of the relational algebraic operations.
This database formalization has many advantages compared with traditional approaches. Very large amount of graphs, for example, can be efficiently managed and concurrently shared among users by virtue of the capability of databases.

References

[1] E. Horowitz, S. Sahni, S. Anderson-Freed, Fundamentals of Data Structures in C, Computer Science Press, New York, 1993.
[2] E.F. Codd, A relational model of data for large shared data banks, Comm. ACM 13(6) (1970) 377-387.
[3] E.F. Codd, Extending the database relational model to capture model meaing, ACM Trans. Database Systems 4(4) (1979) 397-434.
[4] C.J. Date, A Guide to The SQL Standard, Addison-Wesley, Reading, MA, 1989.
[5] A. Silberschatz, H.F. Korth, S. Sudarshan, Database System Concepts, 3rd ed., McGraw-Hill, New York, 1997.
[6] F. Harary, Graph Theory, Addison-Wesley, Reading, MA, 1972.
[7] J.A. Mchugh, Algorithmic Graph Theory, Prentice-Hall, 1990.