Scientific Research and Essays Vol. 7(5), pp. 560-572, 9 February, 2012
Available online at http://www.academicjournals.org/SRE
DOI: 10.5897/SRE11.491
ISSN 1992-2248 ©2012 Academic Journals

Full Length Research Paper

ESU-GOO: The join order algorithm for optimizing small join queries

Areerat Trongratsameethong* and Jarernsri L. Mitrpanont

Faculty of Information and Communication Technology, Mahidol University, Bangkok, Thailand.

Accepted 30 January, 2012

The near exhaustive search algorithm named ESU-GOO was proposed to optimize small join queries. It optimizes the join query time which consists of both the time to generate join query results and the time to search the join order solution, whereas methods such as exhaustive search and greedy algorithm optimizes only one of them. The ESU-GOO integrates Greedy Operator Ordering (GOO) to Exhaustive Search with join graph Update (ESU). GOO is applied to produce the initial solution in the polynomial time and its solution was used as the starting route for ESU. ESU was applied to generate a good join order solution. In the experiments, the join graphs with 4 to 12 nodes were simulated on the basis of the table relationship of the TPC-H database benchmark. ESU-GOO was compared with ESU optimizing the time to generate join query results and GOO optimizing the time to search the join order solution. The experiment showed that the execution time of ESU-GOO on average is 7 times faster than ESU and the ESU-GOO finds 65% of optimal join order solutions. Of all the experiments, the time that ESU-GOO, ESU and GOO use the least join query time are 60, 14 and 26%, respectively.

Key words: Database, query optimization, join order optimization algorithm, near exhaustive search algorithm.

INTRODUCTION

A join operation is one of the most time consuming operations in query processing and optimization process in Database Management System (DBMS). Therefore, one major problem with a query optimizer is to generate a good join order. There are two major costs in generating join query results: (i) the cost used by the join order algorithm to search a good join order called join order algorithm cost and (ii) the cost to generate join query results according to the *n*-1 join sequences of the join order solution called join operation cost, where *n* is the number of relations in a query. Three groups of join order optimization algorithms have been proposed: dynamic programming (Selinger et al., 1979), randomized algorithm (Yanis and Eugene, 1987; Yanis and Younkyung, 1990; Hongbin and Yiwen, 2007; Najmeh et al., 2010), and greedy algorithm (Fegaras, 1998; Pryscila et al., 2007). The dynamic programming or a near exhaustive search algorithm explores all possible equivalent join order routes and searches for an optimal join order solution. Dynamic programming optimizes the join operation cost but ignores the join order algorithm cost because of the exponential size of search space. The dynamic programming is applicable for small join queries. On the other hand, the randomized algorithm and greedy algorithm are designed to reduce the size of search space so that the join order algorithm cost is optimized, but they cannot guarantee optimality of the join operation cost because not all search space is explored. Thus, the randomized algorithm and greedy algorithm are practical for large join queries.

In this paper, the join order algorithm named ESU-GOO is proposed to improve a whole join query cost which is a summation of join order algorithm cost and join operation cost. The ESU-GOO combines the merits of Exhaustive Search with join graph Update (ESU) algorithm (Areerat and Jarernsri, 2009) and Greedy Operator Ordering (GOO) algorithm (Fegaras, 1998). GOO that is designed to optimize join order algorithm cost is first executed and the join order solution from GOO is used as a base

*Corresponding author. E-mail: g4637948@student.mahidol.ac.th.

solution. ESU that can always find optimal join order solutions is later performed to generate equivalent join order routes starting with the same starting route of the base solution, and the join order route having an estimated minimum join operation cost is selected to be a join order solution. This is to reduce search space size but still preserve the good quality of join order solution. Furthermore, the stopping criterion is designed to speed up the ESU-GOO algorithm as follows. The join operation cost of the base solution is used as an initial threshold in the stopping criterion to eliminate the equivalent join order routes having the cumulative value of join operation costs greater than or equal to the join operation cost of the base solution. Later on, the threshold is replaced with the join operation cost of join order route being generated if its join operation cost is less than the threshold. Consequently, the cost used by the ESU-GOO is reduced and the cost for generating join query results is still preserved. Although, both GOO and ESU are executed in the ESU-GOO, the polynomial time complexity used by the GOO algorithm is negligible compared to the time that will be decreased by the reduction in search space and the stopping criterion implemented in the ESU-GOO algorithm.

MATERIALS AND METHODS

Join graph definition

The weighed graph which is commonly used for modeling shortest-path problems (Kenneth, 2003a) is modified to model the join order optimization problem. In this paper, the initial join graph is generated on the basis of the SQL query which is an input of our join order algorithms. One join graph is used to represent one join order route consisting of *n*-1 join operations, where *n* is the number of relations in a join query. The join graph *G* = (*V*, *S*, *TT*, *E*, *W*) consists of the following parameters:

1. *V*: a set of relations, where $v_i \in V$.
2. *S*: a set of relation sizes, where $s_i \in S$.
3. *TT*: a set of average time for storing one tuple into a relation, where $tt_i \in TT$.
4. *E*: a set of edges connecting v_i with v_j, where $e_{ij} \in E$.
5. *W*: a set of join selectivity's between v_i and v_j assigned to e_{ij}, where $w_{ij} \in W$.

The value of s_i relying on whether there exists a selection condition specified on the relation v_i, the s_i is defined as follows:

$$s_i = \begin{cases} sel_i \cdot |v_i| & \text{if there is a selection condition specified on } v_i \\ |v_i| & \text{otherwise,} \end{cases} \tag{1}$$

Where *sel* is selectivity of relation v_i and $|v_i|$ is cardinality of relation v_i.

The sel_i is the ratio of the number of tuples satisfying the selection condition of v_i to $|v_i|$. It is estimated by the selectivity factor (Selinger et al., 1979; Lise et al., 2001) or the histogram technique (Gregory, 1984). The tt_i is an average time for storing one tuple into v_i. The e_{ij} edge connecting v_i with v_j indicates that a join predicate between v_i and v_j is specified in the join query and their join selectivity, w_{ij}, can be estimated by Equation 2. The w_{ij} is the ratio of the number of tuples satisfying the join predicate between v_i and v_j to the multiplication of $|v_i|$ and $|v_j|$. The value of w_{ij} is between 0 and 1:

$$w_{ij} = \frac{|v_i \rhd\lhd_{v_i.A\ op\ v_j.B}\ v_j|}{|v_i| \cdot |v_j|},\ i, j \in [1..n] \text{ and } i \neq j, \tag{2}$$

Where $\rhd\lhd$ is a join operator, "$v_i.A$ *op* $v_j.B$" is a join predicate between relations v_i and v_j, *A* and *B* are attributes of relations v_i and v_j, respectively, and *op* is a comparison operator.

In this paper, the initial join graph *G* is an input of join order algorithm. Figure 1 is an example of SQL query that is transformed into the initial join graph *G* displayed in Figure 2. The initial join graph *G* consists of 4 nodes and 4 edges. Each node represents relation v_i. The number without parenthesis represents relation size, s_i. The number with parenthesis represents average time for storing one tuple into a relation, tt_i. The line connecting v_i with v_j is an e_{ij} edge. The number on each edge represents join selectivity, w_{ij}, between relations v_i and v_j.

Cost parameters of the join graph *G*

To optimize each join query, the join order algorithm generates equivalent join order routes and the join order route having the estimated minimum cost for performing *n*-1 join operations is selected as a join order solution. The equivalent join order routes are the routes producing the same join query results but the costs for performing their *n*-1 join sequences may be different.

In this paper, the cost for performing *n*-1 join operations of the join order route *r* is called a join operation cost, T_J^r, and is estimated by Equation 3:

$$T_J^r = \sum_{k=1}^{n-1} jt_{ij}^k,\ i, j \in [1..n] \text{ and } i \neq j, \tag{3}$$

Where jt_{ij}^k is the join cost for performing a join operation at the join sequence *k* of a join order route *r*.

The join operation ($v_i \rhd\lhd v_j$) at any join sequence *k* is performed as follows:

1. The join predicate between relations v_i and v_j is evaluated.
2. The tuples of relations v_i and v_j satisfying the join predicate are merged and stored in terms of intermediate results.

As a result, the jt_{ij} cost for performing a join operation is the summation of cost for evaluating a join predicate between relations v_i and v_j and cost for storing the intermediate results as illustrated in Equation 4:

$$jt_{ij} = [\max(s_i, s_j) \cdot b \cdot \beta] + [(s_i \cdot s_j \cdot w_{ij}) \cdot (tt_i + tt_j)], \tag{4}$$

Where *b* is the average number of tuples in each found bucket and β is the average time for one evaluation.

The first cost, the cost for evaluating the join predicate, is estimated on the basis of the hash join algorithm (Yu and Meng, 1998) which is efficient and applicable to an equi-join. It should be noted that all join predicates in our experiments are equi-joins. In the hash join algorithm, all tuples of the larger size relation are examined and evaluated with every tuple in the matched bucket.

The second cost, the cost for storing intermediate results, relies on the number of tuples of intermediate results satisfying the join

```
SELECT   *
  FROM   LINEITEM as L, PART as P,
         PARTSUPP as PS, SUPPLIER as S
WHERE    L.L_PARTKEY = P.P_PARTKEY and
         L.L_SUPPKEY = S.S_SUPPKEY and
         P.P_PARTKEY = PS.PS_PARTKEY and
         PS.PS_SUPPKEY = S.S_SUPPKEY and
         L.L_SHIPDATE >= date '1995-09-01' and
         L.L_SHIPDATE < date '1995-10-01' and
         S.S_ACCTBAL < 8000 and
         PS.PS_SUPPLYCOST > 950;
```

Figure 1. An example of SQL query.

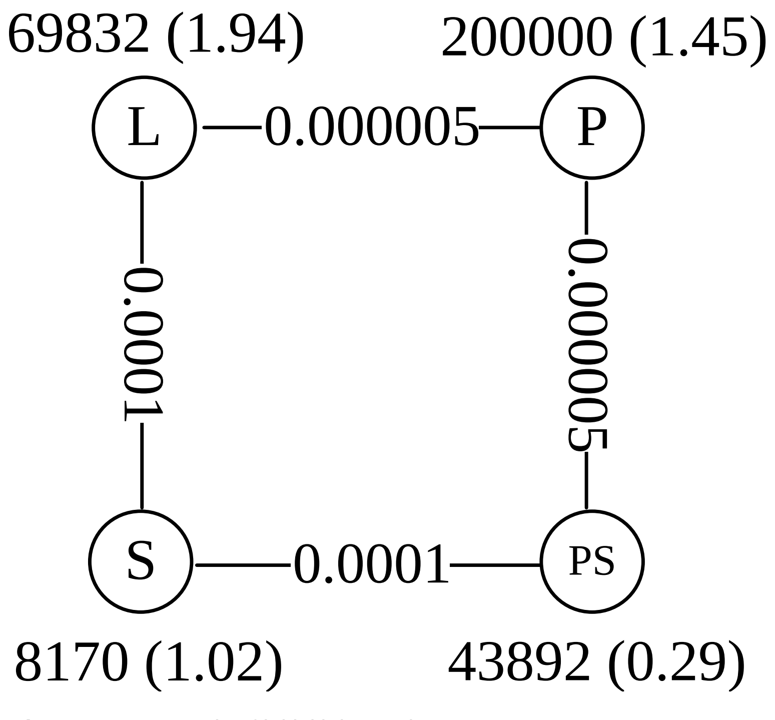

Figure 2. An example of initial join graph *G*.

predicate estimated by the $s_i \cdot s_j \cdot w_{ij}$. Each tuple of intermediate results is generated from merging the tuple of relation v_i to the tuple of relation v_j. Assume that the average time for storing one tuple into v_i and v_j is represented as tt_i and tt_j, respectively. Therefore, the average time for storing one tuple into the intermediate results is approximated as $tt_i + tt_j$.

Join graph updating

The join order optimization problem differs from other shortest-path problems. Cost parameters for calculating join operation cost are changed after two relations are joined at each join sequence. The join graph *G* is then updated after two relations are joined, $v_i \rhd\lhd v_j$, to reflect the change of three following cost parameters as

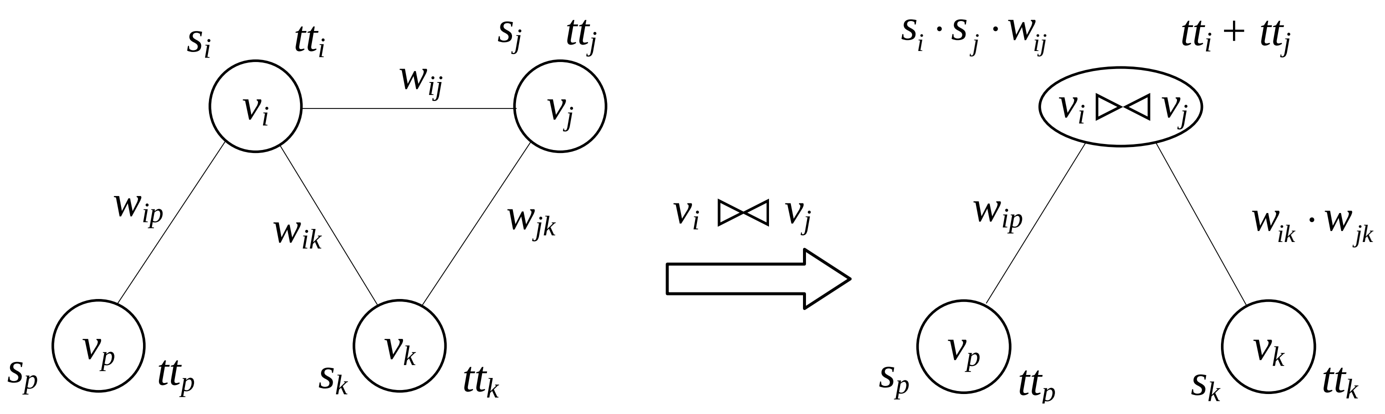

Figure 3. Join graph updating.

displayed in Figure 3:

1. The size of new joined node $v_i \bowtie v_j$ is changed to $s_i \cdot s_j \cdot w_{ij}$.

2. The join selectivities on the edges associated with the $v_i \bowtie v_j$ are updated as follows: for each node v_k connected to both nodes v_i and v_j, the join selectivity between node v_k and joined node $v_i \bowtie v_j$ becomes $w_{ik} \cdot w_{jk}$.

3. The average time for storing one tuple into $v_i \bowtie v_j$ is updated as $tt_i + tt_j$.

It should be noted that the first two cost parameters are updated similarly to the GOO algorithm. The third cost parameter is added to our algorithm because GOO measures cost in terms of intermediate result size but our algorithm measures it in terms of join operation cost.

Join query cost

In this paper, the join query cost, T_Q, includes not only the cost to generate join query results but also the cost used by the join order algorithm. Thus, the join query cost shown in Equation 5 is the summation of two following costs:

1. T_A: join order algorithm cost, the cost used by the join order algorithm for generating equivalent join order routes and searching for the join order solution. The T_A is depicted in Equation 6.
2. T_J: join operation cost of join order solution, the cost for performing *n*-1 join operations of the equivalent join order route providing the estimated minimum join operation cost. The T_J is illustrated in Equation 7:

$$T_Q = T_A + T_J. \quad (5)$$

$$T_A = \sum_{r=1}^{rc} at_r, \quad (6)$$

$$T_J = \min(T_J^1,\ T_J^2,\ ...,T_J^{rc}), \quad (7)$$

Where *rc* is the number of equivalent join order routes generated by the join order algorithm and at_r is the time to generate the equivalent join order route *r*.

ESU-GOO algorithm

The ESU-GOO is a combination of Exhaustive Search with join graph Update (ESU) algorithm and the Algorithm I of Greedy Operator Ordering (GOO) algorithm. The Algorithm I of GOO hereafter is called GOO in short.

GOO is a greedy bottom-up algorithm that is similar to the Kruskal's minimum spanning tree algorithm (Kenneth, 2003b). GOO finds the minimum spanning tree of the join graph, the total size of intermediate results. For each join sequence, the two relations providing the minimum size of intermediate results are searched to be joined. After two relations are joined at each join sequence, GOO updates the join graph to reflect the change of new size and new join selectivity. This directly determines the decision of the join at subsequent steps. Consequently, GOO can generate a good quality order of relational joins in a polynomial time. The number of join order routes generated by GOO is always one. The join order route consists of *n*-1 join sequences, where *n* is the number of relations in a join query. Each join sequence takes at most n^2 iterations to find the two relations providing the minimum size of intermediate results to be joined. As a result, the time complexity of GOO algorithm is $O(n^3)$. The pseudo code of Algorithm I of GOO is given by (Fegaras, 1998).

ESU is an exhaustive search algorithm generating all equivalent join order routes using Depth First Search (DFS) traversal and the join order route providing the estimated minimum size of intermediate results is selected to be a join order solution. ESU also uses the join graph to maintain the equivalent join order routes. One join graph is generated for representing one join order route. The new size and new join selectivity are also updated similarly to the GOO algorithm after two relations are joined at each join sequence. The join order solution obtained by ESU is always the best solution. The time complexity of ESU algorithm is analyzed as follows:

1. The number of join order routes generated by the ESU algorithm is analyzed as follows:
1.1. For the first join sequence, *m* possible join order routes can be generated,
1.2. For the join sequences 2 to *n*-1, there are (*m*-1), (*m*-2), …, 3, and 1 remaining edges that can be generated as the join sequence of equivalent join order routes. The number 3 and 1 are the maximum number of remaining edges at the join sequences *n*-2 and *n*-1, respectively. It should be noted that, the search space is estimated for the worst case, only the edge of two relations selected to be joined, is reduced at each join sequence.
2. As a result, the search space size of ESU is approximated as

$3\prod_{i=0}^{n-4}(m-i)$.

3. Each join order route consists of n-1 join sequences, and each join sequence takes at most n^2 iterations to find the two relations to be joined.
4. Consequently, the time complexity of ESU algorithm is $O(m^n n^3)$, where m and n are the number of edges and nodes, respectively in an initial join graph G.

The ESU-GOO combines the merits of ESU and GOO algorithms. The GOO algorithm is executed to obtain two parameters: the starting route of the join order solution named e_{GOO}, and the join operation cost of join order solution named $T_{J\text{-}GOO}$. The ESU is later executed to generate equivalent join order routes and the join order route having the estimated minimum join operation cost is selected to be a join order solution. The ESU is modified to reduce size of search space as follows:

1. The equivalent join order routes generated by ESU start with the same edge e_{GOO} using DFS traversal so that the search space at the first join sequence is reduced from m to 1. However, our join order solution is guaranteed to be always better than or equal to the join order solution obtained by GOO because the join order solution obtained by GOO is also included in our search space.
2. The DFS used for generating equivalent join order routes and searching for the join order solution is modified to speed up the algorithm as follows:
2.1. The join operation cost of join order solution, T_J, is initially set to $T_{J\text{-}GOO}$.
2.2. During the generation of the join order route, if the cumulative of join operation cost of join order route being generated is greater than or equal to T_J, the join order route is immediately discarded. DFS is not applied to find path beyond this level. The search is then backtracked.
2.3. If the join operation cost of join order route being generated, T_J^r, is less than the T_J then the T_J is replaced with the T_J^r.

GOO and ESU used in ESU-GOO are modified to support the cost parameters used for calculating the join operation cost. The pseudo codes of ESU-GOO algorithm and DFS function are illustrated in Figures 4 and 5, respectively. The time complexity of ESU-GOO algorithm is analyzed as follows:

1. The number of join order routes generated by the ESU-GOO algorithm is from two algorithms.
1.1. GOO always generates 1 join order route.
1.2. The ESU starts from the e_{GOO} edge. Thus the search space at the first join sequence is reduced from m to 1 so that the number of equivalent join order routes generated by ESU is reduced to $3\prod_{i=1}^{n-4}(m-i)$.
1.3. Consequently, the search space size of ESU-GOO is $1+[3\prod_{i=1}^{n-4}(m-i)]$ that is approximately $3\prod_{i=1}^{n-4}(m-i)$.
2. Each join order route consists of n-1 join sequences, and each join sequence takes at most n^2 iterations to find two relations to be joined.
3. Consequently, the time complexity of ESU-GOO algorithm is $O(m^n n^3)$.

Although, the time complexity of ESU-GOO algorithm is still $O(m^n n^3)$ that is similar to the time complexity of ESU algorithm in terms of mathematics but reducing the search space from $3\prod_{i=0}^{n-4}(m-i)$ to $3\prod_{i=1}^{n-4}(m-i)$ can reduce a large number of join order algorithm costs.

In this paper, the experimental results of ESU-GOO algorithm are compared to ESU and GOO algorithms. Therefore, the ESU and GOO are also modified to support the cost parameters for calculating the join operation cost. The pseudo code of GOO algorithm used in the experiments is similar to the GOO algorithm used in the ESU-GOO (lines 1 to 7 in Figure 4). It should be noted that the ESU used in the experiments also uses the DFS function of ESU-GOO (Figure 5) to generate equivalent join order routes and search for the join order solution. Thus, the ESU is also a near exhaustive search but the join order solutions obtained by the ESU still preserve the best join order solutions. The pseudo code of ESU algorithm is displayed in Figure 6.

Figure 7 shows the comparison between the search space of ESU, GOO, and ESU-GOO algorithms. The numbers shown in each join graph are as follows: the number without parenthesis represents s_i, the number with parenthesis represents tt_i in microsecond, and the number on each edge represents w_{ij}. The ESU is an exhaustive search algorithm generating all possible equivalent join order routes, that are r_1 to r_{12}, and the join order solution obtained by ESU is optimal. The search space of GOO is always one. The join order solution obtained by GOO is not optimal. The search space of ESU-GOO is 4, r_1 to r_4. The join order solution obtained by ESU-GOO is not optimal but it is better than the solution obtained by GOO.

Experimental set up

All experiments were done on the HP Compaq LE1711 computer with 4 GB memory running Microsoft Windows XP. The join order algorithms used in the experiments were developed by C++. The join graphs were simulated based on the table relationship of the relational TPC-H database benchmark (available on http://www.tpc.org/tpch). This benchmark has 8 relations consisting of business oriented ad-hoc queries and concurrent data modification. The TPC-H schema represents the table relationship of the TPC-H database (available on http://www.tpc.org/tpch/spec/tpch2.14.0.pdf, page 12). The scale factor, SF, determines the size of the TPC-H database is set to 1.

The cardinality of relation, $|v_i|$, and the average time in millisecond for storing one tuple into a relation v_i of the TPC-H database, tt_i, are presented in Table 1. The size of relation, s_i, is randomly selected in the range (0, $|v_i|$]. All join predicates used in our experiments were equi-joins and their join selectivity's, w_{ij}, between $v_i \rhd\lhd v_j$ were computed by equation 2 and displayed in Table 2. The average number of tuples per bucket b was set to 1000. An average time for evaluating one join predicate, β, was estimated as follows:

1. All equi-joins in the TPC-H database were evaluated and their joining times were measured.
2. The joining times were averaged and the averaged time was found to be 12 ns..

The ESU-GOO, ESU, and GOO algorithms were executed and the following were measured: the join order algorithm cost, T_A, the join operation cost of join order solution, T_J, and the number of equivalent join order routes generated by the algorithm for each simulated join graph, rc.
The T_A used for generating equivalent join order routes and seeking for the route having the estimated minimum join operation cost were measured from the algorithm execution time in millisecond.

```
ESU-GOO Algorithm:
Input:
        G      An initial join graph
Outputs:
        JL     The join sequences of join order solution
        $T_J$     Join operation cost of join order solution
Algorithm:

//***  GOO algorithm ***//
(1)    JL and $e_{GOO}$ are emptied. $T_J$ is set to 0.
(2)    for (k = 1; k <= n-1, k++)
(3)        Find the $e_{ij}$ edge providing the minimum $jt_{ij}$ cost, where $i, j \in [1..n]$ and $i \neq j$.
(4)        $e_{ij}$ found in step 3 is added to JL.
(5)        $jt_{ij}$ found in step 3 is added to $T_J$.
(6)        Update join graph G.          // see Section 2
(7)    end for
(8)    $e_{GOO}$ is set to the first join sequence in JL.

//***  ESU algorithm started with $e_{GOO}$ ***//
(8)     Initialization:
            r is set to 1. The join order route r, $J_r$, is emptied.
            The first join sequence of $J_r$ is set to $e_{GOO}$.
            $T_J^r$ is set to the $jt_{ij}$ cost of $e_{GOO}$.
(9)     Update join graph of $J_r$, $G_r$.
(6)     JL, $T_J$ ← Perform DFS of $J_r$.
(7)    return JL, $T_J$.
```

Figure 4. Pseudo code of ESU-GOO algorithm.

```
DFS Function:
Inputs:
 J_r     The join order route r.          G_r     The join graph of J_r.
 $T^r_J$  The join operation cost of J_r
Outputs:
 JL      The join sequences of join order solution
 $T_J$   The join operation cost of join order solution
FUNCTION  DFS:
(1)  Initialization:
         The first child node of DFS is set to the $v_i \bowtie v_j$ of the first join sequence in $J_r$
         The level of DFS, k, is set to 1.
(2)  repeat:
(3)      repeat:
(4)          k is increased by 1.
(5)          DFS is traversed to the k level.
(6)          The $jt_{ij}$ of the edge that the DFS is traversed to is added to $T^r_J$ .
(7)          Update join graph $G_r$.
(8)      until $k = n-1$ or $T^r_J >= T_J$.
(9)      if ($k = n-1$ and $T^r_J < T_J$ ) then {
             $T_J$ is set to $T^r_J$ . JL is set to $J_r$.
         }
(10)     k is decreased by 1 and DFS is backtracked to the k level.
(11)     r is increased by 1. $J_r$ is emptied. $T^r_J$ is set to 0.
(12)     The join sequences 1 to k of $J_{r-1}$ are copied to $J_r$.
(13)     $T^r_J$ is set to the cumulative of join costs at the join sequences 1 to k of $T^{r-1}_J$.
(14) until DFS is backtracked to the last node of the 2nd level.
(15) return $T_J$, JL.
END FUNCTION
```

Figure 5. Pseudo code of DFS function.

```
ESU Algorithm:
Input:
G     An initial join graph.
Outputs:
T_J   Join operation cost of join order solution.
JL    The join sequences of join order solution.
Algorithm:
(1)   JL is emptied. $T_J$ is set to 0.
(2)   Find all edges, $\{e_{ij,}^{1}\ e_{ij,}^{2}\ ...,\ e_{ij}^{m}\}$, in G.
(3)   for (k=1; k<=m; k++) {
(4)       r is set to 1.
(5)       The join order route r, $J_r$, is emptied.
(6)       $e_{ij}^{k}$ is added to $J_r$.
(7)       The join cost of $e_{ij}^{k}$ edge, $jt_{ij}^{k}$, is added to the join operation cost of $J_r$, $T_J^r$.
(8)       Update join graph of $J_r$, $G_r$.
(9)       JL, $T_J$ ← Perform DFS of $J_r$.
(10)  }
(11)  return JL, $T_J$.
```

Figure 6. Pseudo code of ESU algorithm.

GOO always generates one join order route so that the *rc* of GOO is always one. Due to the termination criteria implemented in the DFS function of ESU and ESU-GOO, the number of join sequences of some equivalent join order routes may be less than *n*-1. Thus the *rc* of ESU and ESU-GOO were measured from the total number of join sequences generated by the algorithm divided by *n*-1.

Join graph simulation

The join graphs with 4 to 12 nodes were simulated based on the table relationship of the TPC-H database benchmark. The simulated join graphs that form unconnected weight graphs were discarded in our experiments because we only focus on a join operator. The unconnected weight graphs will result in the Cartesian product.

All node combinations were used in our simulation. There are 8 relations in the TPC-H. For the join graphs with n = 4 to 8 nodes, the number of node combinations is ${}^{8}C_{n}$. For the join graph with n = 9 to 12 nodes, the nodes consist of two parts: (i) the set of all 8 relations and (ii) n – 8 relations selected from the 8 available relations. The number of the node combinations for the join graph with n = 9 to 12 nodes is ${}^{8}C_{(n \bmod 8)}$.

The experiment was done with the number of edges, *m*, varying from *n*-1 to max(link), where max(link) is the maximum number of links connecting the simulated relations. The max(link) is searched from the table relationship of the TPC-H database. For each simulated join graph, the relation size, s_i, is randomly selected from (0, $|v_i|$]. The average time for storing one tuple into a relation is set to the tt_i. The join selectivity assigned on each edge is set to the w_{ij}. For the join graphs with n = 9 to 12 nodes, if v_i is connected to v_i then the join selectivity between them will be set to 1.0.

Figure 8 shows one of the possible 70 (${}^{8}C_{4}$) combinations for the simulated join graphs with n = 4 nodes. These four nodes are C, L, N, and S relations. The *m* of this combination is in the range of [*n*-1, max (link), which is (3, 4). Then there are two sets of the simulated join graphs for this combination: (i) the set of join graphs with n = 4 and m = 4 and (ii) the set of join graphs with n = 4 and m = 3. One of the simulated join graphs in the second set that forms the unconnected weight graph is discarded. As a result, four join graphs are simulated in this combination.

The number of connected weight graphs for the simulated join graphs with n = 4 to 8 nodes is small. Thus, each connected weight graph was simulated with different sets of relation sizes as illustrated in Table 3. On the other hand, the number of connected weight graphs for the simulated join graphs with n = 9 to 12 nodes is large. Therefore, the join graphs were randomized from the sets of connected weight graphs with *n* nodes and *m* edges as displayed in Table 4.

RESULTS

The experimental results are shown in Figure 9 as follows. Figure 9 (a) shows percent of minimum join query costs, T_Q, obtained by each algorithm. Figure 9 (b) illustrates the average ratio of T_Q costs of GOO to ESU-GOO and ESU to ESU-GOO. Figure 9 (c) expresses

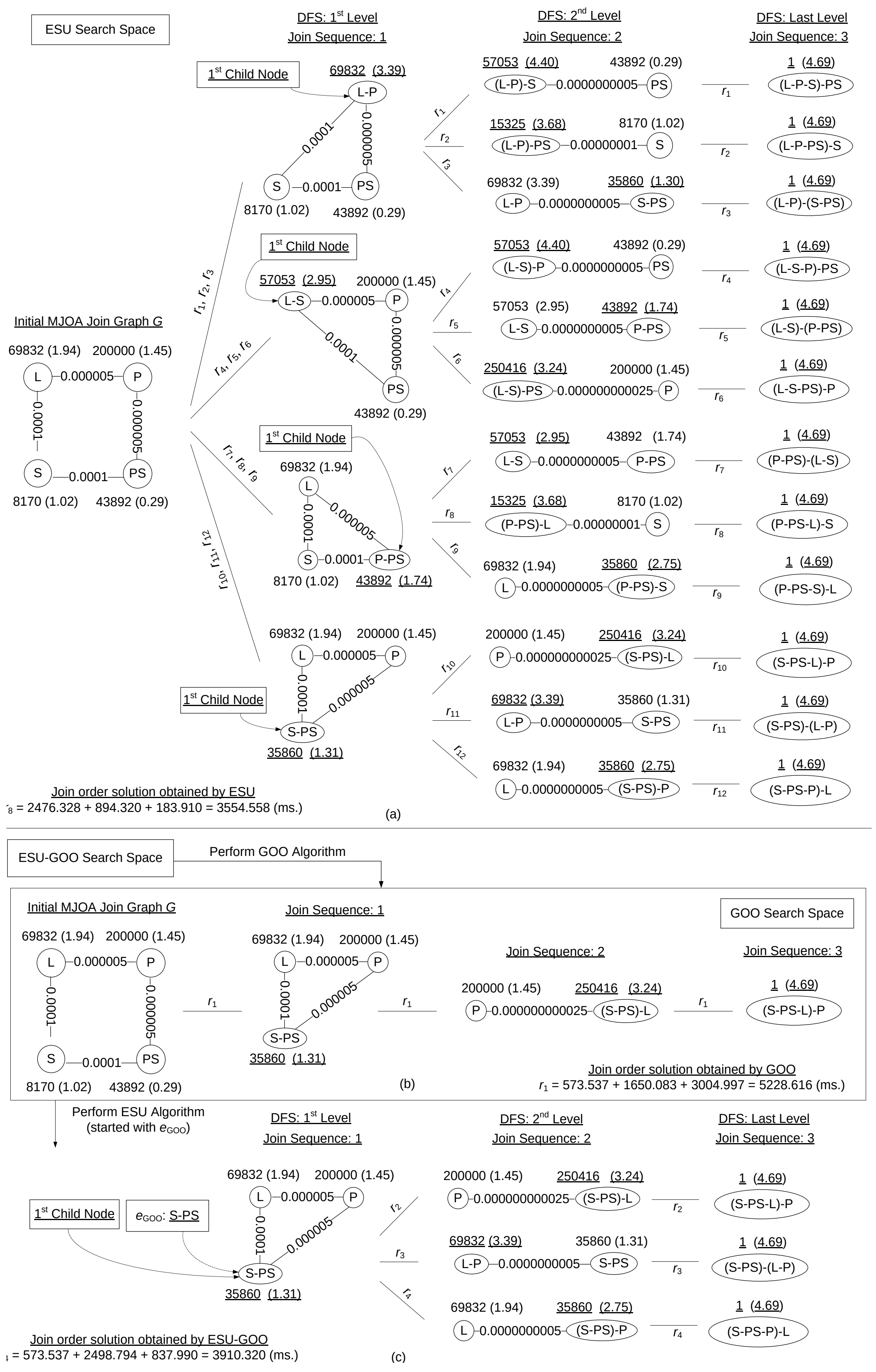
ESU Search Space
DFS: 1st Level
Join Sequence: 1
DFS: 2nd Level
Join Sequence: 2
DFS: Last Level
Join Sequence: 3
1st Child Node
Initial MJOA Join Graph G
69832 (1.94) 200000 (1.45)
8170 (1.02) 43892 (0.29)
r1, r2, r3
r4, r5, r6
r7, r8, r9
r10, r11, r12
Join order solution obtained by ESU
r8 = 2476.328 + 894.320 + 183.910 = 3554.558 (ms.)
(a)
ESU-GOO Search Space
Perform GOO Algorithm
GOO Search Space
Join order solution obtained by GOO
r1 = 573.537 + 1650.083 + 3004.997 = 5228.616 (ms.)
(b)
Perform ESU Algorithm (started with eGOO)
eGOO: S-PS
Join order solution obtained by ESU-GOO
r3 = 573.537 + 2498.794 + 837.990 = 3910.320 (ms.)
(c)

Figure 7. Comparison of search spaces of ESU, GOO, and ESU-GOO.

Table 1. Cardinality of relation and average time for storing one tuple into a relation.

v_i	Alias name of v_i	$\vert v_i \vert$	tt_i (ms.)
CUSTOMER	C	150,000	0.001257
LINEITEM	L	6,001,215	0.001937
NATION	N	25	0.000480
ORDERS	O	1,500,000	0.001241
PART	P	200,000	0.001449
PARTSUPP	PS	800,000	0.000290
REGION	R	5	0.000526
SUPPLIER	S	10,000	0.001016

Table 2. Join selectivity's.

$v_i \bowtie v_j$	w_{ij}	$v_i \bowtie v_j$	w_{ij}	$v_i \bowtie v_j$	w_{ij}
C-N	0.04000000	N-R	0.20000000	PS-S	0.00010000
C-O	0.00000667	N-S	0.04000000	R-N	0.20000000
C-S	0.04000028	O-C	0.00000667	S-C	0.04000028
L-O	0.00000067	O-L	0.00000067	S-L	0.00010000
L-P	0.00000500	P-L	0.00000500	S-N	0.04000000
L-PS	0.00000125	P-PS	0.00000500	S-PS	0.00010000
L-S	0.00010000	PS-L	0.00000125		
N-C	0.04000000	PS-P	0.00000500		

percent of optimal join operation costs, T_J, obtained by each algorithm. Figure 9 (d) displays the average ratio of T_J costs of GOO to ESU-GOO and ESU to ESU-GOO. Figure 9 (e) presents the average time used by each join order algorithm, T_A. Figure 9 (f) shows the average ratio of T_A costs of GOO to ESU-GOO and ESU to ESU-GOO. Table 5 illustrates the average ratio of the number of equivalent join order routes generated by ESU to ESU-GOO.

The x axis in Figure 9a to f consists of two lines: the number of relations is shown at the first line and the number of simulated join graph is displayed at the second line. It should be noted that the ESU and ESU-GOO were very slow in processing the query having n = 13, m >= 20, and n = 13, m >= 26, respectively. As a result, only the join graphs with n = 4 to 12 nodes were simulated and tested in our experiment.

The experiments indicated that reduction in search space in ESU-GOO still preserved the good quality of join order solutions. This resulted in improvement of a whole join query cost, $T_Q = T_A + T_J$, used for each join query.

DISCUSSION

Figure 9b indicates that the percent of T_Q costs obtained by GOO were better than ESU-GOO for the join graphs with n = 4 to 5 nodes. However, the improvement of GOO was negligible, that is, the average of 0.005 and 0.006 millisecond decreases for the join graphs with n = 4 nodes and n = 5 nodes, respectively. The percent of T_Q costs obtained by ESU for the join graphs with n = 4 nodes were also better than ESU-GOO. Nevertheless, the average ratio of T_Q costs obtained by ESU to ESU-GOO for the join graphs with n = 4 nodes was approximately 1 (Figure 9 (b)) that means the T_Q costs obtained by ESU and ESU-GOO were very close.

The experimental results show that although all T_J and T_A costs were optimal for ESU and GOO, respectively, but the whole join query cost, $T_Q = T_J + T_A$, obtained by ESU and GOO were not optimized in most cases. On the other hand, most T_Q costs obtained by ESU-GOO were the smallest among the three, although T_J and T_A costs of ESU-GOO were not optimal.

Consequently, optimizing both cost used by the join order algorithm and cost used for generating join query results in ESU-GOO can improve a whole join query cost, whereas optimizing either cost used by the join order algorithm in greedy algorithm or cost used for generating join query results in exhaustive search cannot guarantee optimality of a whole join query cost.

Conclusions

In this paper, we have presented a novel join order

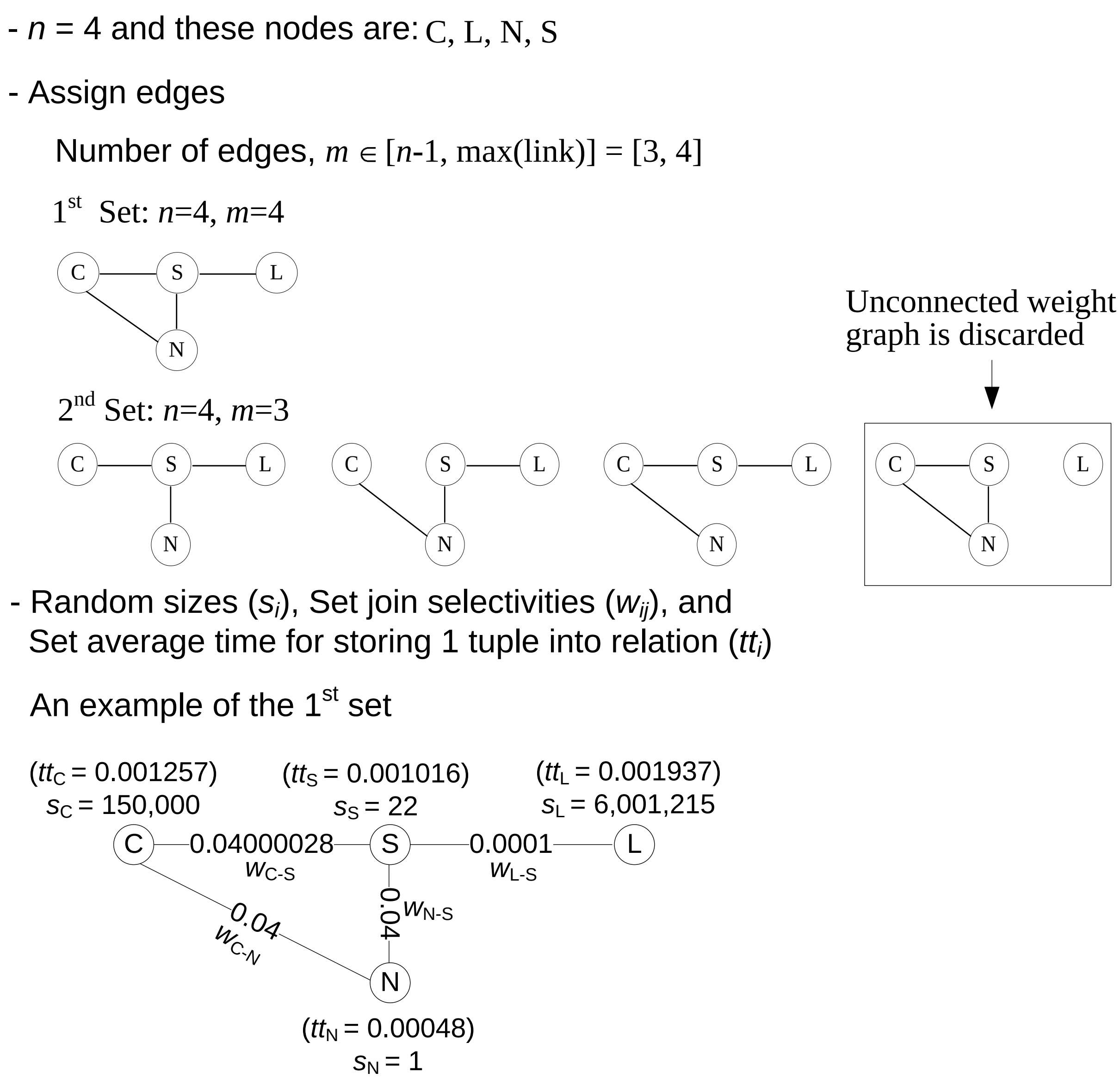
- n = 4 and these nodes are: C, L, N, S
- Assign edges
Number of edges, m ∈ [n-1, max(link)] = [3, 4]
1st Set: n=4, m=4
C S L N
Unconnected weight graph is discarded
2nd Set: n=4, m=3
C S L N
- Random sizes (si), Set join selectivities (wij), and Set average time for storing 1 tuple into relation (tti)
An example of the 1st set
(ttC = 0.001257) sC = 150,000
(ttS = 0.001016) sS = 22
(ttL = 0.001937) sL = 6,001,215
0.04000028 wC-S
0.0001 wL-S
0.04 wN-S
0.04 wC-N
(ttN = 0.00048) sN = 1

Figure 8. An example of one of 70 combinations of the simulated join graphs with n = 4 nodes.

Table 3. The numbers of simulated join graphs with 4 to 8 nodes.

n	No. of connected weight graphs	No. of sets of different relation sizes	Total simulated join graphs
4	64	20	1,280
5	143	10	1,430
6	289	5	1,445
7	380	4	1,520
8	220	5	1,100

algorithm named ESU-GOO to optimize the small join queries. The ESU-GOO is a near exhaustive search designed to reduce the size of search space and still preserve the quality of join order solution. The experimental results reveal that the whole join query cost used by the ESU-GOO is minimized in most cases, whereas most of the whole join query costs used by ESU and GOO are not minimized. The experiments indicate

Table 4. The numbers of simulated join graphs with 9 to 12 nodes.

n	No. of sets of join graphs with *n* nodes and *m* edges, *m* ∈ [*n*-1, max (link)]	No. of random graphs per set	Total of simulated join graphs
9	62	20	1,240
10	305	5	1,525
11	808	2	1,616
12	1,285	1	1,285

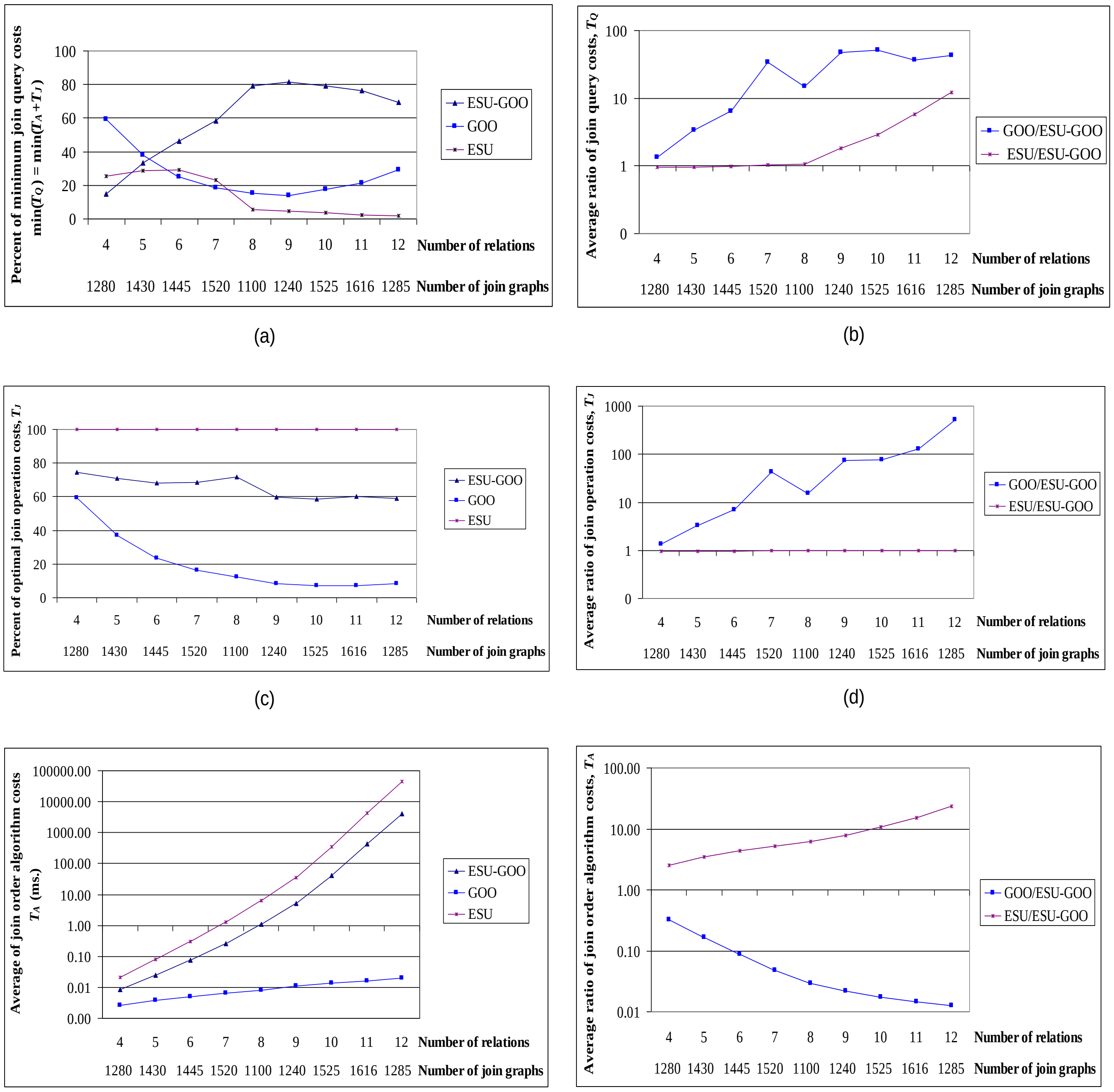

Figure 9. Experimental results.

Table 5. Average ratio of the number of equivalent join order routes (*rc*) generated by ESU to ESU-GOO.

n	Average ratio of *rc* of ESU to ESU-GOO
4	1
5	3
6	4
7	5
8	6
9	7
10	9
11	13
12	18
Average	**7**

that the ESU-GOO is suitable for join queries with less than 13 relations and ESU-GOO is not practical when the number of relations in a query is 13 or more.

ACKNOWLEDGEMENTS

We would like to thank Dr. Supatana Auethaavekiat for her valuable comments and suggestions and Assoc. Prof. Dr. Ruja Pholsward for her valuable assistance in proofreading. We also would like to convey thanks to the Faculty of Information Communication and Technology, Mahidol University for providing all laboratory facilities.

REFERENCES

Areerat T, Jarernsri LM (2009). Exhaustive Greedy Algorithm for Optimizing Intermediate Result Sizes of Join Queries. Proceedings of the 8th IEEE/ACIS International Conference on Computer and Information Science held at Shanghai, China. IEEE Comput. Soc., pp. 463-468.

Fegaras L (1998). A New Heuristic for optimizing large queries. Proceedings of the 9th International Conference on Database and Expert Systems Applications held at Vienna, Austraria. SpringerLink LNCS, pp. 726-735.

Gregory P (1984). Accurate Estimation of the Number of Tuples Satisfying a Condition. Proceeding of the 1984 ACM SIGMOD international conference on Management of data held at Boston, Massachusetts, USA. ACM SIGMOD, pp. 256-276.

Hongbin D, Yiwen L (2007). Genetic Algorithms for Large Join Query Optimization. Proceeding of the 9th annual conference on Genetic and evolutionary computation held at London, England, UK. ACM, pp. 1211-1218.

Kenneth RH (2003). Discrete Mathematics and Its Applications: Fifth Edition. McGRAW-HILL, New York, pp. 593-601.

Kenneth RH (2003). Discrete Mathematics and Its Applications: Fifth Edition. McGRAW-HILL, New York, pp. 691.

Lise G, Ben T, Daphne K (2001). Selectivity Estimation using Probabilistic Models. Proceeding of the 2001 ACM SIGMOD international conference on Management of data held at Santa Barbara, California, USA. ACM SIGMOD, pp. 461-472.

Najmeh D, Hossein S, Homayun M (2010). Optimizing N relations join queries by genetic algorithm. SRE. 5(13):1576-1582.

Pryscila GB, Marcos SS, Fabiano S (2007). Kruskal's Algorithm for Query Tree Optimization. Proceedings of the 11th International Database Engineering and Applications Symposium held at Banff, Alberta, Canada. IEEE Comput. Soc., pp. 296-302.

Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG (1979). Access Path Selection in a Relational Database Management system. Proceeding of the 1979 ACM SIGMOD international conference on Management of data held at Boston, Massachusetts, USA. ACM SIGMOD, pp. 23-34.

Yanis IE, Eugene W (1987). Query Optimization by Simulated Annealing. Proceeding of the 1987 ACM SIGMOD international conference on Management of data held at San Francisco, California, USA. ACM SIGMOD, pp. 9-22.

Yanis IE, Younkyung KC (1990). Randomized Algorithms for Optimizing Large Join Queries. Proceeding of the 1990 ACM SIGMOD international conference on Management of data held at Atlantic City, New Jersey, USA. ACM SIGMOD, pp. 312-321.

Yu CT, Meng W (1998). Principles of Database Query Processing for Advanced Applications. Morgan Kaufmann, CA, pp. 20-26.