IMPROVED A* ALGORITHM FOR QUERY OPTIMIZATION

Amit Goyal
Ashish Thakral
G.K. Sharma
Indian Institute of Information Technology and Management, Gwalior.
Morena Link Road, Gwalior, India.
E-mail: amitgoyal@iiitm.ac.in

KEYWORDS
Query Processing, A* algorithm

ABSTRACT

Exponential growth in number of possible strategies with the increase in number of relations in a query has been identified as a major problem in the field of query optimization of relational databases. Present database systems use exhaustive search to find the best possible strategy. But as the size of a query grows, exhaustive search method itself becomes quite expensive. Other algorithms like A* algorithm, Simulated Annealing etc. have been suggested as a solution. However, all these algorithms fail to produce the best results; necessarily required for query execution. We did some modifications to the A* algorithm to produce a randomized form of the algorithm and compared it with the original A* algorithm and exhaustive search. The comparison results have shown improved A* algorithm to be almost equivalent in output quality along with a colossal decrease in search space in comparison to exhaustive search method.

I. INTRODUCTION

The process of query optimization in relational databases is considered to be an expensive job when it comes to queries involving large number of relations. The number of possible ways to execute a query increases exponentially as the number of relations increases in the query. Finding the best way in reasonable time is absolutely compulsory in a database system since an improper strategy could lead to increase in actual execution time of the query. As the complexity of databases increase, it becomes necessary for the future query optimizers to adopt a low cost (in terms of time) algorithm instead of the traditional exhaustive search methods.

Previous attempts to solve the problem revolve around various search strategies like deterministic, randomized and heuristic. Many algorithms have been proposed in the literature which takes advantage of one or other of the search strategies. Simulated Annealing (SA), Iterative Improvement (II), Two Phase Optimization (2PO) and Genetic Algorithms [D.E. Goldberg 1989] form a class of generic randomized optimization algorithms that have been applied to query optimization. Genetic randomized algorithms simulate a biological phenomenon. Simulated Annealing (SA) performs a continuous random walk accepting downhill moves always and uphill moves with some probability, trying to avoid being caught in a high cost local minimum [Y. Ionnidis E. Wong 1987; S. kikpatrick, Gelatt and Vecchi]. Iterative Improvement (II) [A. Swami 1989] performs a large number of local optimizations. Each one starts at a random node and repeatedly accepts random downhill moves until it reaches a local minimum. The Two Phase Optimization (2PO) [Y. Ionnidis and Y. Kang 1987] algorithm is a combination of II and SA.

One major heuristic algorithm proposed for query optimization is A star (A*) algorithm. This algorithm is useful for queries with few relations [4]. It normally gets stuck with some local minima if the numbers of relations are substantially increased, producing an output sub standard to the exhaustive search. Heuristic algorithms have helped in reducing the time of optimization process at the cost of quality of output.

In this paper, we make certain improvements to the original A* algorithm by taking advantage of the fact that execution of original A* algorithm creates a linked list of promising nodes i.e., the nodes that are most probably leading to the best path. The improved A* algorithm, when used for query optimization, gives output comparable to exhaustive search in minimal amount of search space. Since execution time of a particular algorithm will depend on the search space it requires during execution. Furthermore, query optimization results in the formation of Join processing tree, we have to apply all the possible operators to a node in the tree to create its successors. Node creation in itself is a time consuming process. Thus, we also give a comparative study of the total number of nodes required by exhaustive search, original A* algorithm and our modified A* algorithm for a particular query optimization problem.

This paper is organized as follows: In Section 2, we discuss the basic features of randomized algorithms and the working of original A* algorithm. Our modified version of A* algorithm is introduced in Section 3. Section 4 defines the problem specific parameters to be used while applying our improved A* algorithm. The experimental results, definition of parameters and analysis of results are dealt in Section 5. Section 6 concludes the paper.

II. ORIGINAL A* ALGORITHM

For optimizing a query, various possible paths to execute the query are considered and each path can be thought of as a strategy to get the final result. Each strategy forms a Join processing tree. The various Join processing trees can be grouped and represented in the form of a tree which we have named as - *strategy tree*. The units constituting a strategy are known as states i.e., nodes of the *strategy tree*. One can estimate the total cost of each strategy or a path in the *strategy tree*. Less is the cost of a particular path; less will be the execution time of a query – the ultimate aim of query optimization. Randomized algorithms move in random directions among the possible paths in search of a better path. Simulated Annealing, 2PO are examples of few randomized algorithms.

TABLE 1
List of Data Structure, Variables and Functions

open, opentemp	Nodes that have been generated and the heuristic function applied to them but which have yet not been examined (i.e. had their successors generated). It is actually a priority queue in which the elements with the highest priority are those whose cost is lowest.
node0	Root node of the strategy tree.
node(x).cost	The cost involved in reaching the node(x) from its parent node.
goal node	Leaf nodes of the strategy tree.
length(open)	Total number of nodes present in linked list *open*.
min(open)	Returns the node with lowest cost from all the nodes present in *open*.
totalcost(node(x),node(y))	The cost involved in reaching from node(x) to node(y).
generateallthesuccesor(node(x))	Applies all the possible transformation rules and generates the successors of node(x), information about each child is also added to node(x).
add(x,y)	Concatenates the nodes present in linked list x to y i.e. finally x contains nodes of x as well as y while y gets empty.
remove(x,y)	Deletes the node(x) from the linked list y.

The original A* algorithm can be explained as follows. Each state in the query optimization can be considered to be a node in the *strategy tree*. Each node contains, in addition to a description of the problem state it represents, an indication of the cost it takes to reach from its parent to the node.

The list of data structures, variables and functions used in original A* algorithm and improved A* algorithm are given in Table 1.The original A* algorithm is shown in Figure 1. The basic steps involved in original A* algorithm are [Rich and Knight et al. 1983]:-

1. Start with *open* containing just the initial state.
2. Until a goal is reached or there are no nodes left in *open* do:
   (a) Pick the best node in *open*.
   (b) Generate its successors.
   (c) For each successor do:
       (i) Evaluate it and add it to *open*.

```
Procedure A*(node0,open) {       //uses linked list open
                                      //and node0 as root
    open ← node0
    prnode ← node0
    //starting node of the graph is node0
    costinit ← node0.cost
    While((prnode!= goal node)or(length (open)!= 0))
    {
          x ← min(open)
// x is assigned the lowest cost node available in open
          remove(x, open)
          prnode ← x
          costinit ← totalcost(node0,prnode)
          generateallthesuccessors(prnode)
          for each successor {
             Evaluate it, add it to open and record its
             parent
          }
    }
return costinit
}
```

Figure1. Original A* Algorithm

This algorithm proves good for low depth trees but when the depth increases it gets stuck with some local minima giving a poor result.

III. IMPROVED A* ALGORITHM

Original A* algorithm is considered to be successful with smaller queries. It gets stuck in some local minima when the number of relations in the query is large or the depth of a *strategy tree* becomes more than the normal depth. The presence of local minima deviates original A* from the best path. The algorithm generates a linked list during its execution. The linked list contains those nodes that have been considered for the best paths but their children were not generated. A particular node in the linked list is not considered because it may be a costly node but it is quite possible that the subsequent nodes generated from this particular node may be of cheaper cost.

Since original A* algorithm considers only the local costs rather than the global costs and results in some substandard results, we may assume that it has erroneously moved in the wrong direction. Further examination of nodes, existing in the linked list, by generating their children or the subsequent nodes could provide greater information regarding the cost and may lead to the best strategy for query optimization.

Our improved A* algorithm utilizes this additional information to find out the best strategy for query optimization. Improved A* algorithm uses two linked lists instead of one used in original A* algorithm. The use of variables, functions and data structures can be referred from Table1. The pseudo code for algorithm is given in Figure 2. The algorithm can be explained as follows:

1. Firstly, original A* algorithm is executed and the cost of the best path calculated by it is stored as *Total_Cost*. The algorithm also generates the linked list *open*.

2. A random number in the range 0 and 1 is generated and a probability function is decided upon. The chances of the algorithm to execute the following code is decided by the initial function value We continuously decrement the initial value according to the decided function to make the algorithm reach to a definite end. If the random number is within the range of probabilities, we continue executing the following code otherwise the algorithm quits.

   (a) The node with the lowest cost is selected from the linked list *open.*

   (b) Calculate the total cost from the root to the selected node. If the cost is greater than the *Total_Cost* calculated by original A* algorithm in Step 1, the selected node is discarded and removed from the linked list *open*, and algorithm moves to Step 2 else the algorithm moves to Step 2(c).

   (c) Apply the original A* algorithm to the selected node in Step 2(b) with an additional condition that if at any step the cost goes higher than the *Total_Cost* the algorithm stops and the probability value is reduced by .1 and the corresponding node is removed from *open*; the algorithm moves to Step 2 else the algorithm continues. The subsequent nodes generated by original A* for the selected node are stored in linked list *opentemp* instead of *open.*

   (d) Once the Step 2(c) is finished, the selected node is removed from the linked list *open*

   (e) If the algorithm succeeds in reaching to the goal state, the probability is reduced by.25 and the nodes in the linked list *opentemp* are shifted to *open,* else *opentemp* is made empty and the algorithm again starts from Step 2

```
Segment1
    Procedure IMA*{
    Procedure OA*(node0,open){ //using linked list
                                // open and node0 as root
        open ← node0
        prnode ← node0
        costinit ← node0.cost
        Total_Cost := 0
        while ((prnode != goal node) and (Total_Cost
    <= costinit))or (length (open)  != 0) {
                x ← min(open)
     // x is assigned the lowest cost node available in
    open
                remove(x, open)
                prnode ← x
                costinit ← totalcost(node0,prnode)
               generateallthesuccessors(prnode)
                for each successor {
                  Evaluate it, add it to open and record
    its
                  parent
                }
            }
    }
    y ← min(open)
Segment2
    rand← random between 0 and 1
    probability ←.9 + depth/500
    // probability function defined and initialized; depth
    //is equal to height of tree
    while rand <= probability {
        prnode ←  min(open)
        remove(y,open)
        Total_Cost ← totalcost(node0, prnode)
        if Total_Cost <= costinit{
          procedure A*(prnode,opentemp)
         }
        if (prnode = goal node)  {
            add (open,opentemp)
            probability ← probability - .25
    //probability function value decreased by a high
    value
        }
       else{
            probability ← probability - .1 //probability
    //function decreased by a low value after each cycle
             }
            rand← random between 0 and 1
      }
    return Total_Cost
    }
```

. Figure2. Improved A* Algorithm

.

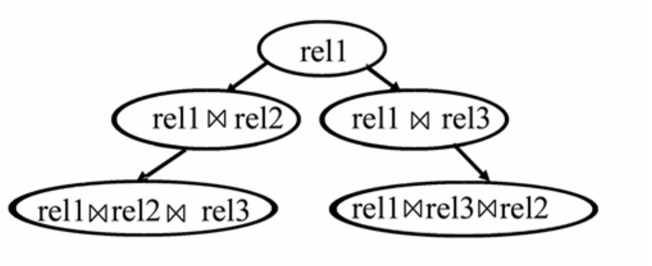
In this algorithm, firstly, original A* is executed. Here, we get the promising nodes accumulated in the linked list *open* generated during the execution. With some probability, node with the lowest cost is picked from the list. Total cost to reach from root to that node is calculated and if found smaller than the *Total_Cost* as calculated by original A*, the selected node is given a chance to prove its capability. Original A* algorithm is applied with the selected node as the starting node. The probability of picking a node from *open* reduces in each cycle.

IV. PROBLEM SPECIFIC PARAMETERS

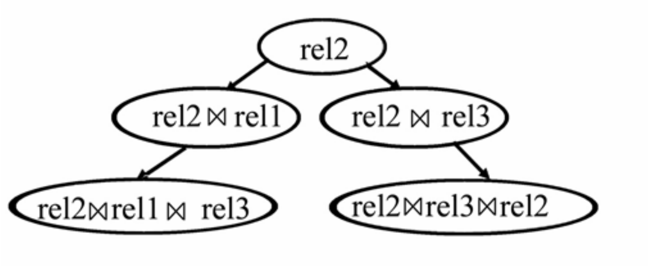
The application of above mentioned algorithm for query optimization involves specification of certain parameters like state space, cost functions and transformation rules.

A. State Space

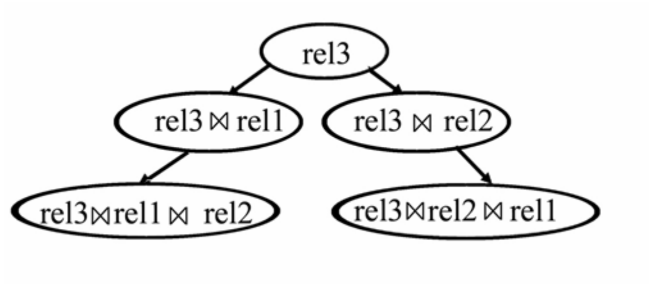
Each state in query optimization corresponds to an access plan (strategy) of the query to be optimized. Using the general rules of optimization, selection and projections are performed first and excluding unnecessary Cartesian products, thus reducing various bad strategies [M. Jarke and G. Koch 1984]. The final problem settles down to finding the best order of relations that gives us the minimum cost. We can represent each strategy in form of a tree known as Join processing tree. All the Join processing trees can be combined and shown in the form of trees; we call it a *strategy tree*. The *strategy tree* is formed in a way such that the leaves represent the final order of relations (present in the query). The nodes represent the intermediate relation order, the results from the nodes are passed down in the tree and the edges indicate the flow of data from top to bottom i.e. from root to leaves. Leaves are the operations producing the final query result. The best path found will just be a strategy or a Join processing tree giving the optimal way to execute the query. An example of *strategy tree*s is shown in figure 3.



3(a)



3(b)



3(c)

Figure 3. Strategy Tree for Query Involving Three Relations

B. Transformation Rules

Children of a node in a *strategy tree* are generated by adding one more relation to the existing set of relations present with the node; for example, if we are having the present node as (rel1 ⋈ rel2 ⋈ rel3) then one of the children can be (rel1 ⋈ rel2 ⋈ rel3 ⋈x), where x can be any other relation except rel1, rel2 and rel3. Finally, when all the relations are exhausted we are able to get all the possible combinations of relations i.e. all the query execution plans. For example in a query involving three relations we get the possible plans as (rel1⋈ rel2 ⋈ rel3), (rel1 ⋈ rel3 ⋈ rel2), (rel2 ⋈ rel1 ⋈ rel3), (rel2 ⋈ rel3 ⋈ rel1) and (rel3 ⋈ rel1 ⋈ rel2), (rel3 ⋈ rel2 ⋈ rel1) which are equivalent in results (except in their costs), since following transformation rules hold [Ramakrisnan Gehrke 2003 et. al.].

(a) A ⋈ B → B ⋈ A
(b) (A ⋈ B) ⋈C ↔ A ⋈ (B ⋈ C)
(c) (A ⋈B) ⋈C → (A⋈C) ⋈B
(d) A ⋈ (B ⋈ C) → B ⋈ (A ⋈C)

The *strategy trees* possible for the example are shown in Figure. 3

C. Cost Functions

Cost means the processing time involved in moving a downward step in a *strategy tree*. The cost function which estimates the costs of nodes in the *strategy tree* can be fixed as per the requirement of a database management system. For example in a select- project-join query the cost function can include the effects of (a) the size of tuple (in case of projections) (b) number of tuples to be processed. This function f(x) can be as

$$f(x) = f(\text{a, b, size of a tuple, number of tuples}) \qquad (1)$$

where a and b are some constants that can be set to some value so as to prefer projection over selection and join or as per the requirements. In a distributed database, the cost function can be the time taken by data to come from different sites [Bernstein, Goodman, Wong, Reve and Rothnie 1981].

V. TESTING RESULTS

The following section deals with the initialization values and parameters taken by us while simulating the algorithms.

A. Testing Related Values

In this section, we provide with the values and conditions provided during comparison of original A* algorithm, exhaustive search and the improved A* algorithm. We conducted tests for queries containing 5 to 11 relations. In real situations all the tables will (mostly) have different number of tuples. The number of tuples was given a random integer value from the set [1000, 10000] for each table. For example, a query involving four tables i.e. four relations can have the number of tuples as (2000, 3701, 7700, 9000).

We implemented all the algorithms in MATLAB. For simplicity, we provided the *strategy tree* initially in the form of adjacency matrices. The number of relations were varied, comparison of the cost involved and the number of nodes required in each case (in real situations graph will be created dynamically so less the number of nodes required by some algorithm i.e. less is the search space requirements; less time will be taken by that algorithm) was recorded. We also plotted the comparative graphs for the costs and the number of nodes required by all the three algorithms.

B. Cost Calculation

For simplicity, we assume that the tuples in all the tables are of same length and the tables can have any kind of relationship. Thus, the number of output tuples of rel1 $\bowtie$ rel2 can be any random number between the number of tuples of rel1 and rel2. Moreover, we assume that a join operation is implemented by nested-loop join method, so the total processing cost of a join equals: no. of tuples (rel1) × no. of tuples (rel2), since for each tuple in rel1 a relation has to scan all the tuples in rel2.

C. Probability Function

The probability function has been initialized in such a way that the segment 2 of improved A* algorithm (in comparison to original A*) is executed at least once for queries having relations larger than 6. If we find a better path, the probability is reduced drastically so that we do not get stuck in the segment2. Moreover, our experiment reveals high chances of detecting the best path just after the first execution of segment2. The probability function used in our experiment is

$$p = .9 + n/500 \qquad (2)$$

Where n is the number of relations in a query and p is the probability to enter segment2. The detection of a good path results in decrease of probability value by .25

$$p = p - .25 \qquad (3)$$

else, we decrease probability by .1 after each cycle.

D. Other Assumptions

Other assumptions are as follows:
1. node0 is always taken as rel1 i.e. the *strategy tree* is always assumed to have its root as rel1.
2. All joins are handled in a single way i.e. nested-loop join method.

VI. RESULTS

For queries involving few relations i.e. less than six, there was almost no difference in all the three algorithms in terms of output quality and number of nodes to be visited. The results are compared with respect to the following attributes:

A. Output Quality

By output quality we mean the total cost of the best path found by a certain algorithm. The output quality of the improved A* algorithm is almost equivalent to exhaustive search i.e. most of the times we are able to find the shortest path (since exhaustive search is sure to find the shortest path). When compared with original A*, modified A* gave better performance.

B. Nodes to be Visited

In real situations, since the *strategy tree* will always be generated during the execution process itself; more the number of nodes required to be visited during execution of a certain algorithm, greater will be the search space and more will be the time taken to execute the algorithm, as node creation itself is a time consuming process. Thus, lesser number of nodes visited during execution of an algorithm indicates smaller run time. The results show that the improved A* algorithm requires lesser number of nodes as compared to exhaustive search and this gap increases tremendously as the number of relations increases. On an average, the total search space was reduced just to two percent in comparison to exhaustive search. It was slightly greater than the nodes required by original A* algorithm. With a small increase in search space of about .001 percentages as compared to original A* algorithm, we are able to achieve very good results in output quality.

C. Time of Execution

The execution time of improved A* algorithm was found to be much less than the execution time of exhaustive search and almost equivalent to that of original A* algorithm on the higher side.

VII. EXPLANATION OF THE OBSEVED BEHAVIORS

The output quality of the improved A* algorithm is found to be almost equivalent to the exhaustive search.

This is because of the generation of linked list name *open* containing the most promising nodes (those nodes that were considered in the best path search but there children were not generated) during the execution of original A*. Original A* algorithm fails because at some point of time it gets stuck in a local minima and deviates from the best path, ultimately leading to a substandard result. Improved A* algorithm utilizes this fact and the search for better path starts from the nodes in linked list only. Thus, without much increase in search space, in comparison to original A* algorithm, it is able to find results almost equivalent to exhaustive search. The execution of improved A* is equivalent to 2-3 times execution of original A* on trees of reduced height, resulting in an acceptable execution time.

Moreover, we are decreasing the probability function value (by a large value) only when we are able to find a complete alternate path. This helps to ensure that we are able to find the best possible path; since a complete alternative path is acceptable only when its total cost is lower than the cost of previously found complete paths. However, to make sure that the algorithm reached a definite end, we reduce the probability function value by a small amount in each cycle of the execution.

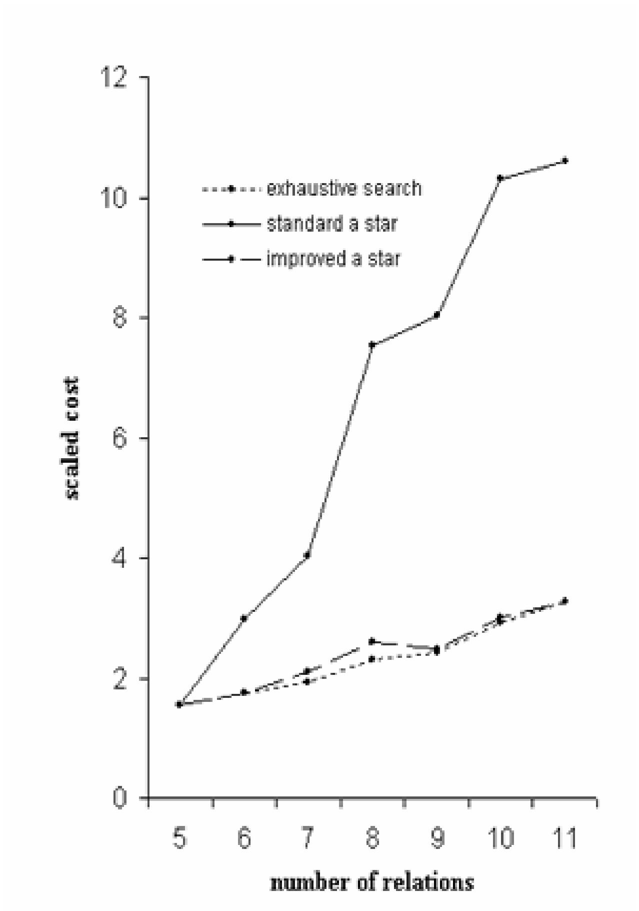


Figure4. Shortest Path Cost Calculated by Each Algorithm

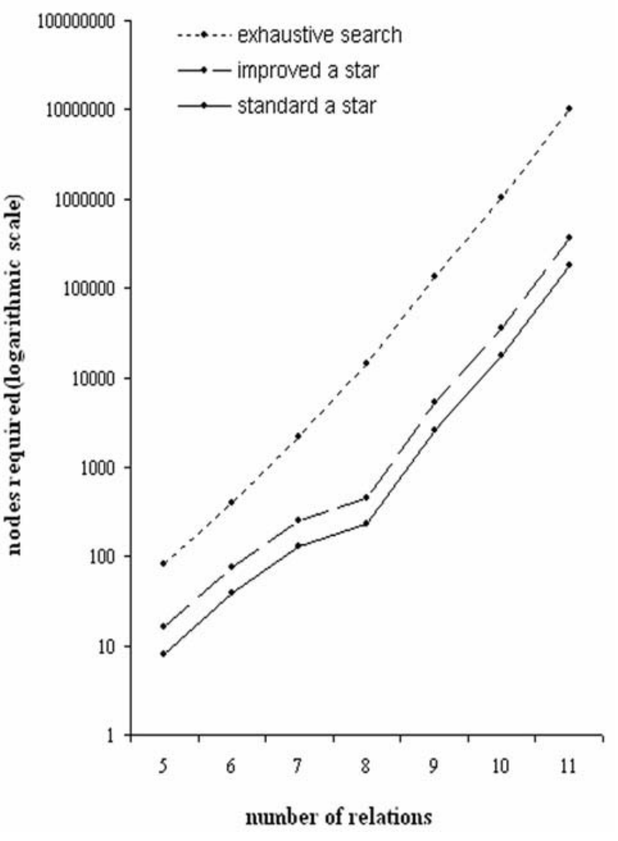


Figure5. Average Number of Nodes Required by Each Algorithm

IX. CONCLUSION

The algorithm can be used for all the queries involving any number of relations. There is always a tradeoff between the time requirements and the output quality which can be controlled by the probability function as per the requirements. The cost function can be modified to incorporate the various features that may affect the cost. Future works can be directed towards finding the optimal probability value and the cost functions. Original A* algorithm is suitable for a query having number of relations up to five. Thus, the probability value can be adjusted in a manner that for queries with relations up to five doesn't need to enter the segment2 of Improved A* algorithm. The algorithm can also be tested to find the shortest path in a tree or a graph.

REFERENCES

A. Swami. "Optimization of large join queries: Combining heuristics and combinatorial techniques". In Proc. *ACM-SIGMOD Conference on the Management of Data, pages* 367{376, Portland, OR, June 1989.

B. E. Goldberg. "Genetic Algorithms in Search, Optimization, and Machine Learning". Addison-Wesley, Reading, MA, 1989.

Elaine Rich and Levin Knight 2002. *Artificial Intelligence* TATA McGRAW-HILL, New Delhi.

H. Yoo and S. Lafortune. "An intelligent search method for query optimization by semijoins." *IEEE Trans. on Knowledge and Data Engineering,* 1(2):226{237, June 1989}.

M. Jarke and J. Koch. "Query optimization in database systems". *ACM Computing Surveys*, 16(2):111{152, June 1984}.

P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie. "Query processing in a system for distributed databases (SDD-1)". ACM TODS, 6(4):602{625, December 1981}.

Ramakrishnan and Gehrke 2003 *Database Management Systems* TATA McGRAW-HILL, New Delhi.

S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. Science, 220(4598):671{680, May 1983}.

Y. Ionnidis and Y. Kang. "Randomized algorithms for optimizing large join queries". In Proc. *ACM-SIGMOD Conference* on Management of Data Pages 312{321. Atlantic City, NJ, May 1990.

Y. Ionnidis and E. Wong. "Query optimization by simulated annealing". In Proc. *ACM-SIGMOD Conference* on the Management of Data, Pages 9{22, San Francisco, CA, May 1987}.

AUTHOR BIOGRAPHIES

AMIT GOYAL was born in Jodhpur, India and is doing his graduation in the field of Information Technology from Indian Institute of Information Technology and Management, Gwalior, India. His e-mail is amitgoyal@iiitm.ac.in

ASHISH THAKRAL was born in Shamli, India and is doing his graduation in the field of Information Technology from Indian Institute of Information Technology and Management, Gwalior, India. His e-mail is asthakral@iiitm.ac.in

G.K. Sharma was born in Ghaziabad, India and is a professor at Indian Institute of Information Technology and Management, Gwalior, India in the department of Information Technology His e-mail is gksharma@iiitm.ac.in