Heuristic Algorithms for Designing a Data Warehouse with SPJ Views *

Spyros Ligoudistianos Timos Sellis Dimitri Theodoratos Yannis Vassiliou

Department of Electrical and Computer Engineering
Computer Science Division
National Technical University of Athens
Zographou 157 73, Athens, Greece
{spyros,timos,dth,yv}@dblab.ece.ntua.gr

Abstract. A Data Warehouse (DW) can be abstractly seen as a set of materialized views defined over relations that are stored in distributed heterogeneous databases. The selection of views for materialization in a DW is thus an important decision problem. The objective is the minimization of the combination of the query evaluation and view maintenance costs. In this paper we expand on our previous work by proposing new heuristic algorithms for the DW design problem. These algorithms are described in terms of a state space search problem, and are guaranteed to deliver an optimal solution by expanding only a small fraction of the states produced by the (original) exhaustive algorithm.

1 Introduction

A Data Warehouse (DW) can be seen as a set of materialized views defined over distributed heterogeneous databases. All the queries posed to the DW are evaluated locally using exclusively the data that are stored in the views. The materialized views have also to be refreshed when changes occur to the data of the sources. The operational cost of a Data Warehouse depends on the cost of these two basic operations: query answering and refreshing. The careful selection of the views to be maintained in the DW may reduce this cost dramatically. For a given set of different source databases and a given set of queries that the DW has to service, there is a number of alternative sets of materialized views that the administrator can choose to maintain. Each of these sets has different refreshment and query answering cost while some of them may require more disk space than the available in the DW. The Data Warehouse design problem is the selection of the set of materialized views with the minimum overall cost that fits into the available space.

Earlier work [8] studies the DW design and provides methods that generate the view selections from the input queries. It models the problem as a state space search problem, and designs algorithms for solving the problem in the case of SPJ relational queries and views.

* Research supported by the European Commission under the ESPRIT Program LTR project "DWQ: Foundations of Data Warehouse Quality"

1.1 Related Work

Many authors in different contexts have addressed the view selection problem. H. Gupta and I.S. Mumick in [2] use an A* algorithm to select the set of views that minimizes the total query-response time and also keeps the total maintenance time less than a certain value. A greedy heuristic is also presented in this work. Both algorithms are based on the theoretical framework developed in [1] using AND/OR view directed acyclic graphs. In [3] a similar problem is considered for selection-join views with indexes. An A* algorithm is also provided as well as rules of thumb, under a number of simplifying assumptions. In [10], Yang, Karlapalem and Li propose heuristic approaches that provide a feasible solution based on merging individual optimal query plans. In a context where views are sets of pointer arrays, Roussopoulos also provides in [7] an A* algorithm that optimizes the query evaluation and view maintenance cost.

1.2 Contribution and Paper Outline

In this paper we study heuristic algorithms for the DW design problem. Based on the model introduced in [8, 9] we introduce a new A* algorithm that delivers the optimal design. This algorithm prunes the state space and provides the optimal solution by expanding only a small fraction of the whole state space. We also present two variations of the heuristic function used in A*, a 'static' and a 'dynamic' heuristic function. The dynamic heuristic function is able to do further pruning of the state space. To demonstrate the superiority of the A* algorithm, we compare it analytically and experimentally with the algorithms introduced in [8].

The rest of the paper is organized as follows. In Section 2 we formally define the DW design problem as a state space search problem providing also the cost formulas. In Section 3 we propose a new A* algorithm that delivers an optimal solution for the DW design problem. Improvements to the A* algorithm are proposed in Section 4. Section 5 presents experimental results. We summarize in Section 6.

2 The DW design problem

We consider a nonempty set of queries $\mathbf{Q}$, defined over a set of source relations $\mathbf{R}$. The DW contains a set of materialized views $\mathbf{V}$ over $\mathbf{R}$ such that every query in $\mathbf{Q}$ can be rewritten completely over $\mathbf{V}$ [4]. Thus, all the queries in $\mathbf{Q}$ can be answered locally at the DW, without accessing the source relations in $\mathbf{R}$. By Q^V, we denote a complete rewriting of the query Q in $\mathbf{Q}$ over $\mathbf{V}$.

Consider a *DW configuration* $\mathbf{C} =< \mathbf{V}, \mathbf{Q}^V >$ [8, 9]. We define:

$E(\mathbf{Q}^V)$: The sum of the evaluation cost of each query rewriting Q_i^V in $\mathbf{Q}^V$ multiplied by the frequency of the assosiate input query Q_i,

$M(\mathbf{V})$: The sum of the view maintenance cost of each view in $\mathbf{V}$,

$S(\mathbf{V})$: The sum of the space needed for all views in $\mathbf{V}$,

$T(\mathbf{C})$: The operational cost of $\mathbf{C}$ where:

$$T(\mathbf{C}) = cE(\mathbf{Q}^V) + M(\mathbf{V})$$

The parameter c indicates the relative importance of the query evaluation cost and view maintenance cost.

The DW design problem can then be stated as follows [8]:

Input: A set of source relations $\mathbf{R}$. A set of queries $\mathbf{Q}$ over $\mathbf{R}$. The cost functions E, M, T. The space t available in the DW for storing the views. A parameter c.

Output: A DW configuration $\mathbf{C} =< \mathbf{V}, \mathbf{Q}^V >$ such that $S(\mathbf{V}) \leq t$ and $T(\mathbf{C})$ is minimal.

In this paper we investigate the DW design problem in the case of selection-projection-join conjunctive queries without self-joins. The relation attributes take their values from domains of integer values. Atomic formulas are of the form x *op* $y + c$ or x *op* c, where x, y are attribute variables, c is a constant, and *op* is one of the comparison operators $=, <, >, \leq, \geq$ but not $\neq$. A formula F implies a formula F' if both involve the same attributes and F is more restrictive. (For example $A = B$ implies $A \leq B + 10$). Atoms involving attributes from only one relation are called *selection atoms*, while those involving attributes from two relations are called *join atoms*.

2.1 Multiquery graphs

A set of views $\mathbf{V}$ can be represented by a *multiquery graph*. A multiquery graph allows the compact representation of multiple views. For a set of views $\mathbf{V}$, the corresponding multiquery graph, $\mathbf{G}^V$, is a node and edge labeled multigraph. The nodes of the graph correspond to the base relations of the views. The label of a node R_i in $\mathbf{G}^V$ is the set containing the attributes of the corresponding relation that are projected in each view of $\mathbf{V}$. For every selection atom p of the definition of a view V, involving attributes of R_i there is a loop on R_i in $\mathbf{G}^V$ labeled as $V : p$. For every join atom p of the definition of a view V, involving attributes of R_i and R_j there is an edge between R_i and R_j in $\mathbf{G}^V$ labeled as $V : p$. The complete definition of the multiple query graph appears in [8].

2.2 Transformation Rules

In [8] we defined the following five transformation rules that can be applied to a DW configuration.

Edge Removal: A new configuration is produced by eliminating an edge labeled by the atom p from the query graph of view V, and the addition of an associated condition to the queries that are defined over V.

Attribute Removal: If there are atoms of the form $A = B$ and A, B are attributes of a view V, we eliminate A from the projected attributes of V.

View break: Let V be a view and N_1, N_2 two sets of nodes labeled by V in $\mathbf{G}^V$ such that: (a) $N_1 \not\subseteq N2, N2 \not\subseteq N1$, (b)$N1 \cup N2$ is the set of all the nodes

labeled by V in $\mathbf{G}^V$ and (c) there is no edge labeled by V between the nodes in $N_1 - N_2$ and $N_2 - N_1$. In this case this rule replaces V by two views, V_1 defined over N_1 and V_2 defined over N_2. All the queries defined over V are modified to be defined over $V_1 \times V_2$.

View Merging: A merging of two views V_1 and V_2 can take place if every condition of V_1 (V_2) implies or is implied by a condition of V_2 (V_1). In the new configuration, V_1 and V_2 are replaced by a view V which is defined over the same source relations and comprises all the implied predicates. All the queries defined over V_1 or V_2 are modified appropriately in order to be defined over V.

Attribute Transfer: Suppose there are atoms of the form $A = c$, where A is an attribute of a view V, we eliminate A from the projected attributes V. All the queries defined over V are modified appropriately.

2.3 The DW design problem as a state-space search problem

The DW design problem is formulated as a state space search problem. A *state* s is a DW configuration $\mathbf{C} =< \mathbf{V}, \mathbf{Q}^V >$. In particular, the state $\mathbf{C} =< \mathbf{G}^Q, \mathbf{Q}^Q >$ that represents the complete materialization of the input queries, is called the *initial state* s_0. There is a transition $T(s, s')$ from state s to state s', iff s' can be obtained by applying any of the five transformation rules to s. It can be shown that by the application of the above transformation rules we can get all possible DW configurations [8]. With every state s we associate, through the function $T(\mathbf{C})$, the operational cost of $\mathbf{C}$. Also, the space needed for materializing the views in $\mathbf{V}$ is given by $S(\mathbf{V})$. We can solve the DW design problem by examining all the states that are produced iteratively from s_0 and report the one with the minimum value for the function $T(\mathbf{C})$ that satisfies the constraint $S(\mathbf{V}) \leq t$.

It was evident that the number of all produced states of the state space is too large. An algorithm that solves the DW design problem by searching the state space within an acceptable time has to prune the state space and examine only a limited fraction of the states. Given a transition $T(s, s')$, the operational cost $T(\mathbf{C}')$ and the space $S(\mathbf{V}')$ of s' are greater, equal or less than the corresponding $T(\mathbf{C})$ and $S(\mathbf{V})$ of s. Hence any algorithm that wishes to guarantee the optimality of the solution it delivers, it needs to examine every feasible state of the state-space.

In order to provide an algorithm that will be able to deliver an optimal solution for the DW design problem but at the same time will prune down the size of the state-space, we proceed as follows to alter the way states are created.

Consider the states $s_1 =< \mathbf{G}^{Q_1}, \mathbf{Q}_1^{Q_1} >, \ldots, s_n =< \mathbf{G}^{Q_n}, \mathbf{Q}_n^{Q_n} >$, where $\mathbf{Q}_1 = \{Q_1\}, \ldots, \mathbf{Q}_n = \{Q_n\}$. Let $S_i = \{s_i^1, \ldots, s_i^{k_i}\}$ denote the set of all the feasible states created from state s_i, $i = 1, \ldots, n$, by applying to $< \mathbf{G}^{Q_i}, \mathbf{Q}_i^{Q_i} >$ the transformation rules: edge removal, attribute removal, view break and attribute transfer. It is not hard to see that view merging cannot be applied to $< \mathbf{G}^{Q_i}, \mathbf{Q}_i^{Q_i} >$ or any of the other state produced from S_i.

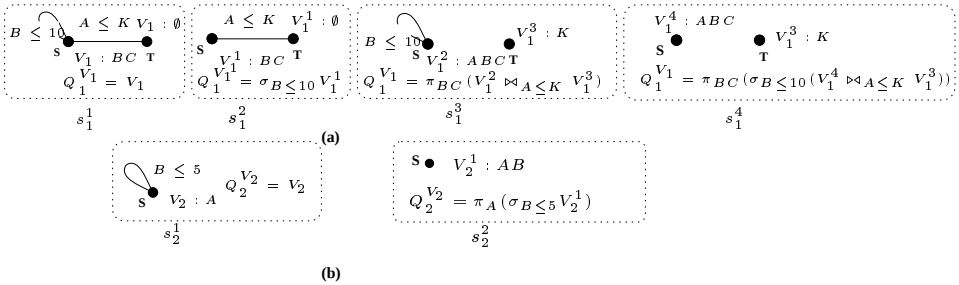


Fig. 1. The states of the set S_1 (a) and S_2 (b)

Example 1. Consider the queries $Q_1 = \pi_{BC}(\sigma_{B\leq 10}(S) \bowtie_{A\leq K} T)$, and $Q_2 = \pi_A(\sigma_{B\leq 5}(S))$. Figure 1(a) shows the elements of the set S_1 that we can get from Q_1 while Figure 1(b) shows the elements of S_2 that we can get from Q_2.

Consider now two configurations $< \mathbf{V}_1, \mathbf{Q}_1^{V_1} >$ and $< \mathbf{V}_2, \mathbf{Q}_2^{V_2} >$. By combining these configurations we can create a new configuration $< \mathbf{V}, \mathbf{Q}^V >$ as follows: $\mathbf{V} = \mathbf{V}_1 \cup \mathbf{V}_2$, $\mathbf{Q}^V = \mathbf{Q}_1^{V_1} \cup \mathbf{Q}_2^{V_2}$. The nodes of the multiquery graph of the new configuration are the nodes of the union of the original configurations. For each edge of the two original multiquery graphs, an identical edge is added to the multiquery graph of the new configuration. The same happens for each node label. The new multiquery graph expresses collectively all the views and the query rewritings of the two original configurations.

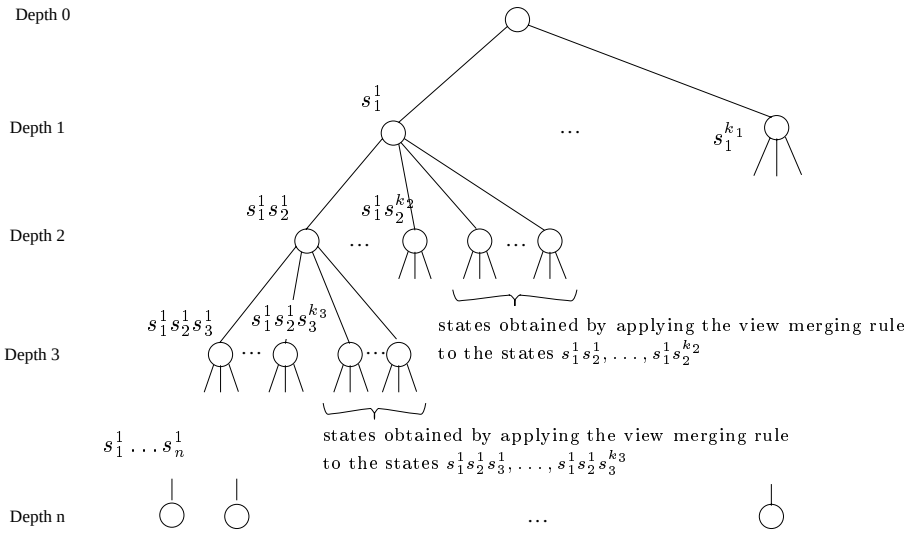


Fig. 2. The tree of the combined states

Combinations between states are defined similarly to combinations between configurations. By performing the combination between states we can create the

tree of combined states of the Figure 2. This tree is defined as follows: The nodes of the tree are states and combined states. The root node is a state with no views or query rewritings ($< \emptyset, \emptyset >$). The children of the root node are the states $s_1^1, \ldots, s_1^{k_1}$. These nodes are at *depth* 1. The children of a node s which are expanded at *depth* d, $d > 1$, are the combined states resulting by combining s with each one of the states $s_d^1, \ldots, s_d^{k_d}$ plus all the states resulting by applying the *view merging* transformation rule to these combined states (not necessarily once). The leaves of the tree (nodes at *depth* n) are the states of the state space of the DW design problem as this was formulated earlier. The tree of the combined states of the example 1 is shown in Figure 3.

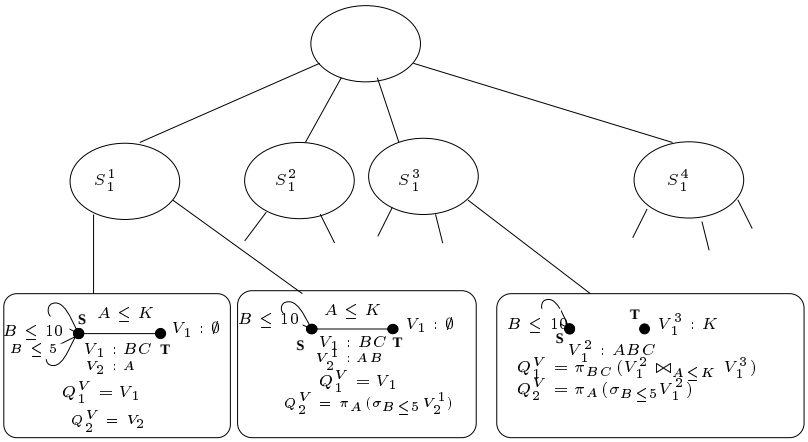


Fig. 3. The tree of the combined states for the sets S_1, S_2 of the Example 1

The operational cost function T and the space function S can be defined at each node of the tree of combined states. At each node, T and S express the operational cost and the space of the corresponding configuration. By induction we can prove that given a node n_1 at *depth* d_1 and a node n_2 at *depth* d_2 where $d_1 < d_2$ and n_1 is an ancestor of n_2 then $T(n_1) \leq T(n_2)$ and $S(n_1) \leq S(n_2)$. This is true under the assumption that we consider no multi query optimization and every view is maintained separately without using auxiliary views [9]. The fact that the cost and the space function monotonically increase while we visit the nodes of the tree from the root to the leaves, allows the design of algorithms that find the optimal DW configuration by exploring only a small fraction of the state space. The following Branch and Bound algorithm is such an algorithm.

Branch and Bound algorithm: The algorithm generates and examines the tree in a depth-first manner. Initially it sets $c = \infty$. When it finds a leaf node state s that satisfies the space constraint and has cost $T(s) \leq c$ it keeps s as s_{opt} ans sets $c = T(s)$. The generation of the tree is discontinued below a node if this node does not satisfy the space constraint or its cost exceeds c. When no more nodes can be generated it returns the s_{opt} as the optimal DW configuration.

3 A* Algorithm

We present an A* algorithm [6] that searches for the optimal solution in the tree of the combined states. The new algorithm prunes down the expanded tree more effectively than the Branch and Bound algorithm because at each step it uses also an estimation of the cost of the remaining nodes. The A* algorithm introduces two functions, $g(s)$ and $h(s)$ on states. The value of $g(s)$ expresses the cost of the state s and is defined as the total operational cost of the associated configuration $\mathbf{C}$ ($g(s) = T(C)$). The value of $h(s)$ expresses an estimation of the additional cost that will be incurred in getting from s to a final state. $h(s)$ is *admissible* if it always returns a value lower than the actual cost to a final state. If $h(s)$ is admissible then the A* algorithm that searches the tree of the combined states is guaranteed to find a final state (leaf node) s_{opt} such that the operational cost of s_{opt} is minimal among all final states [6]. In order to define $h(s)$, we introduce the function $l(s_i^j)$ for each $s_i^j \in S_i$, the set of feasible states created from $< \mathbf{G}^{Q_i}, \mathbf{Q}_i^{Q_i} >$. This function expresses a lower bound of the estimated cost that will be added to a combined state s', in case s' is produced by the combination of s_i^j with a third state s. The value of the function $l(s_i^j)$ is the operational cost of the associated configuration $< \mathbf{V}_j, \mathbf{Q}_i^{V_j} >$ minus the view maintenance cost of the views that may contribute to a view merging.

$$l(s_i^j) = T(< \mathbf{V}_j, \mathbf{Q}_i^{V_j} >) - \sum_k M(V_j^k),\ V_j^k \textit{ contributes to a view merging}$$

We can also define $L(S_i)$ as the minimum $l(s_i^j)$ for each $s_i^j \in S_i$:

$$L(S_i) \ = \ \min_j [l(s_i^j)],\ s_i^j \in S_i$$

For a state s at depth i the heuristic function $h(s)$ is defined as:

$$h(s) \ = \ \sum_{j=i+1}^{n} L(S_j)$$

Proposition 1. *For every leaf node s_l which is successor of the node s, $T(s_l) \geq g(s) + h(s)$ holds.*

The proof of the Proposition 1 is presented in [5].

A* Algorithm: The A* algorithm proceeds as follows: First it initializes $c = \infty$, constructs $S_1, \ldots, S_n$ and begins the tree traversal from the root node. When the algorithm visits a node it expands all its children. It computes the function $g(s) + h(s)$ for each one of the generated nodes and also the space function $S(s)$. Then, it continues to generate the tree starting from the state which has the lowest cost $g(s) + h(s)$. The generation of the tree is discontinued below a node if this node does not satisfy the space constraint or when $g(s) + h(s)$ exceeds c. When the algorithm finds a leaf node state s_l that satisfies the space constraint and has cost $T(s) < c$, it keeps s_l as s_{opt} and sets $c = T(s)$. When no more nodes can be generated, the algorithm returns the s_{opt} as the optimal DW configuration.

4 Improvements to the basic A* Algorithm

Consider two admissible heuristic functions h_1 and h_2. h_1 is said to be more *informed* than h_2 if for every non-final state s, $h_1(s) \geq h_2(s)$. When an A* algorithm is run using h_2, it is guaranteed to expand at least as many nodes as it does with h_1 [6] The definition of h uses functions $L(S_i)$, $l(s_i^j)$ to pre-compute the additional cost from a state s to a final state. $l(s_i^j)$, at each step, excludes from the estimation the maintenance cost of each view that may participate to a view merging, without considering the maintenance cost of the new view that will replace the merged views. Another point is the fact that in some cases the maintenance cost of a view is eliminated even if the merging of this view will generate no successor node. In order to get a more informed heuristic function, we define a new "dynamic" heuristic function h'. The new heuristic function uses the functions L' and l' which are "dynamic" versions of functions the L and l. The functions L', l' are called "dynamic" because they are recomputed at each algorithm iteration. The main advantage of these functions is that they are able to exploit information from the states already expanded, making the new heuristic function h' more informed than h. For each state s at depth d and for each $s_i^j \in S_i$ where $i > d$ the function $l'(s_i^j, s)$ is defined as follows:

$$l'(s_i^j, s) = T(< \mathbf{V}_j, \mathbf{Q}_i^{V_j} >) - \sum_k M(V_j^k) + \sum_k W(V_j^k, s)$$

$$W(V_j^k, s) = \begin{cases} 0 & : \quad if\ \exists\ V \in s,\ V\ can\ be\ merged\ with\ V_j^k \\ \frac{M(V_j^k)}{n_j^k + 1} & : \quad otherwise \end{cases}$$

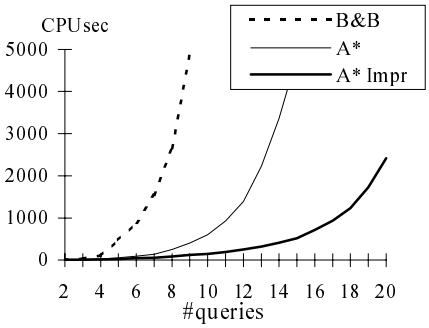
where V_j^k may be contributing to a view merging and n_j^k is the number of views that can be merged with V_j^k and these views are in any of $S_{d+1}, \ldots, S_n$.

The function $L'(S_i, s)$, similarly as the function $L(S_i)$, is defined as the minimum $l'(s_i^j, s)$ for each $s_i^j \in S_i$. The heuristic function h' for a state s at depth i is defined also as the sum of the function $L'(S_j, s)$ for each $j \in [i+1, \ldots, n]$.

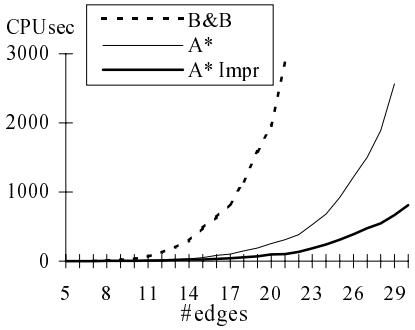
In [5] we prove that the heuristic function h' is admissible. Obviously $h'(s) \geq h(s)$ for every state s of the tree of the combined states, so h' is more informed than h. That means that when the A* algorithm uses h', less nodes are expanded compared to the case where h is used.

5 Experimental Results

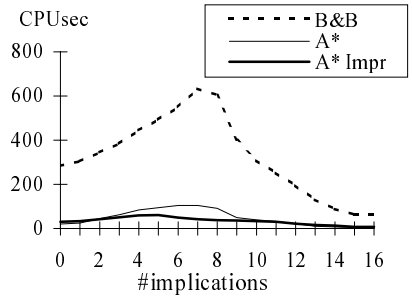
We have performed a sequence of experiments to compare the performance of the Branch and Bound algorithm and the A* algorithm, the latter using the two heuristic functions presented in the previous sections. The algorithms are compared in terms of the following factors: (a) the complexity of the input query set and (b) physical factors. The complexity of the query set is expressed by three parameters: the number of input queries, the number of selection and join edges of all the input queries, and the overlapping of the queries in the input



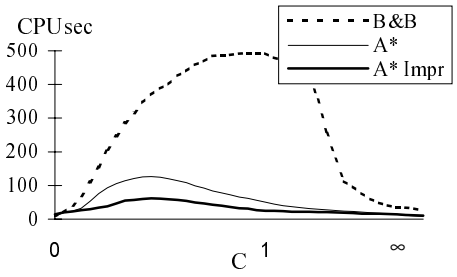
(a) CPU time vs size of input



(b) CPU time vs number of edges



(c) CPU time vs number of implications



(d) CPU time vs physical characteristics

Fig. 4. Experimental Results

query set. The query overlapping is expressed by the total number of implications between the selection or join atoms of different input queries. Finally the physical factors are expressed by the relative importance of the query evaluation and view maintenance cost (factor c in Section 2).

Experiment 1: The number of queries varies. We study the performance of the algorithms when the number of the input queries varies. Figure 4.(a) shows the CPU time needed by each algorithm. The performance of the A* algorithm using the dynamic heuristic function h' is denoted by *A* Impr*. Actually it is this algorithm that is significantly better than the other two.

Experiment 2: The number of edges varies. We study the performance of the algorithms as the number of selection and join edges of the input queries varies. Figure 4.(b) shows the CPU time needed by the Branch and Bound algorithm and the two variations of A* take. In this experiment too the improved A* algorithm is the winner.

Experiment 3: The number of implications varies. We study the performance of the algorithms while varying the number of implications between atoms of different queries. Figure 4.(c) shows the CPU time taken by the algorithms. As the number of implications grows and before it exceeds a certain limit, both

the Branch and Bound and the A* algorithm execution time increases. When the number of implications exceeds this limit, the algorithms become fasters.
Experiment 4: parameter c varies. We run the algorithms while varying the parameter c. Figure 4.(d) reports the CPU time needed by the algorithms. When c is close to 0 or much greater than 1 (the view maintenance cost or the query evaluation cost is important), then all the algorithms perform very efficiently. In the middle interval the algorithms are slower.

6 Summary

In this paper we have studied heuristic algorithms that solve the DW design problem, by extending the work presented in [8]. We have studied the DW design problem as a state space search problem and proposed a new A* algorithm that guarantees to deliver an optimal solution by expanding only a small fraction of the states produced by the (original) exhaustive algorithm and the Branch and Bound algorithm proposed in [8]. We have also studied analytically the behaviour of the A* algorithm and proposed a new improved heuristic function. Finally we implemented all the algorithms and investigated their perfomance with respect to the time required to find a solution.

Interesting extensions of the present work include the following: (a) The use of auxiliary views in the maintenance process of the other views, and (b) The enlargement of the class of queries to include aggregate queries.

References

[1] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *Proc. of the 6th ICDT Conf.*, pages 98–112, 1997.

[2] H. Gupta and I. S. Mumick. Selection of Views to Materialize Under a Maintenance Cost Constraint. In *Proc. of the 7th ICDT Conf.*, pages 453–470, 1999.

[3] W. Labio, D. Quass, and B. Adelberg. Physical Database Design for Data Warehousing. In *Proc. of the 13th Intl. Conf. on Data Engineering*, 1997.

[4] A. Levy, A. O. Mendelson, Y. Sagiv, and D. Srivastava. Answering Queries using Views. In *Proc. of the ACM Symp. on Principles of Database Systems*, pages 95–104, 1995.

[5] S. Ligoudistianos. *Design and Operational Issues of Data Warehouse Systems.* PhD thesis, NTUA, 1999.

[6] N. J. Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann Publ, 1980.

[7] N. Roussopoulos. View Indexing in Relational Databases. *ACM Transactions on Database Systems*, 7(2):258–290, 1982.

[8] D. Theodoratos, S. Ligoudistianos, and T. Sellis. Designing the Global DW with SPJ Queries. *Proc. of the 11th (CAiSE) Conference*, June 1999.

[9] D. Theodoratos and T. Sellis. Data Warehouse Configuration. In *Proc. of the 23rd Intl. Conf. on Very Large Data Bases*, pages 126–135, 1997.

[10] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *Proc. of the 23rd Intl. Conf. on Very Large Data Bases*, pages 136–145, 1997.