An Intelligent Search Method for Query Optimization by Semijoins

HYUCK YOO, STUDENT MEMBER, IEEE, AND STÉPHANE LAFORTUNE, MEMBER, IEEE

Abstract—Query optimization strategies based on the reduction of the referenced relations by means of semijoins have received considerable attention. The limitations of such strategies have to do with computational efficiency (very large search space of semijoin reduction sequences), optimality of the solution (when heuristics are used), and generality of the class of queries allowed (e.g., simple queries, chain queries, tree queries). We consider the problem of finding an optimal semijoin sequence that fully reduces a given tree query. We present a new method that "intelligently" navigates the space of all semijoin sequences and returns an optimal solution. We report on experiments that show that our method performs very efficiently; on average, less than five percent of the search space is searched before an optimal solution is found. Other advantages of the method are: 1) ease of implementation, 2) generality of the cost model considered, and 3) ability to handle tree queries with arbitrary target lists.

Index Terms—Distributed databases, heuristic search, optimal semijoin sequences, query optimization, semijoins.

I. INTRODUCTION

The importance of query optimization in centralized and distributed relational database systems is widely recognized. Among the many different strategies that have been proposed for distributed query optimization, those based on the reduction of the referenced relations by means of semijoins have received considerable attention.

Semijoins have been introduced to reduce the cost of expensive joins in distributed database systems. Suppose that the join of relations \( r_1 \) and \( r_2 \) must be computed and suppose that these relations are located at different sites. Before the join is computed, \( r_1 \) and \( r_2 \) may be "reduced" by the semijoins \( r_1 \bowtie r_3 \) and \( r_2 \bowtie r_4 \), the semijoin of \( r_2 \) by \( r_1 \) (or the semijoin from \( r_1 \) to \( r_2 \)), consists of the following steps: 1) the joining column of \( r_1 \) is sent to the site of \( r_2 \), and 2) the tuples of \( r_2 \) are scanned and those that do not match a value of the column of \( r_1 \) sent are deleted (thus, \( r_2 \) is reduced).

An important issue in query optimization based on semijoin reduction is to find good (possibly optimal) sequences of semijoins that (fully) reduce the relations referenced in a given (tree) query before any of the joins are performed. After this so-called reduction phase, the reduced relations can be joined to produce the answer to the query. The problem of finding optimal semijoin reduction sequences (or programs) has been studied extensively in the literature on distributed query processing [5]. We present a new method for solving this problem. Our method uses heuristic search to "intelligently" navigate the space of all semijoin sequences and return an optimal solution.

Heuristic search is a fundamental method in artificial intelligence (AI). Although some attempts have been made to use heuristic search in query optimization (see [11], [27], [24], [31]), none of these attempts addresses the problem of finding optimal semijoin sequences. (Of course, many query optimization methods employ "heuristics" to ease the computational efforts during exhaustive search; however, these heuristics do not necessarily preserve optimality.)

We employ the A* algorithm, which is probably the best known heuristic search technique (see [2], [23], [28]). More precisely, we model the steps of query processing by semijoins in terms of states in a search space, and we develop heuristic information that is needed by the A* algorithm in order to proceed with the search. This information satisfies an "admissibility" condition, thus guaranteeing that an optimal path is found between the initial and final states in the search space. Our experiments show that this method performs very efficiently; on average, less than five percent of the search space is searched before an optimal solution is found. Other advantages of the method are: 1) ease of implementation, 2) generality of the cost model considered, and 3) ability to handle tree queries with arbitrary target lists.

The organization of this paper is as follows. In Section II, we briefly review some past work related to semijoins and we clarify our motivation. In Section III, we formulate the problem and describe the (general) cost model used. Section IV explains how to navigate the search space. The proof that the heuristic search preserves optimality is given, and a pruning mechanism is described. Section V provides an illustrative example based on Sections III and IV. In Section VI, we present some experimental results. Section VII discusses how to fully reduce an arbitrary number of relations, rather than all the relations in the query. Finally, Section VIII concludes the paper. (A list of symbols is included in the Appendix.)

Manuscript received October 7, 1988; revised April 6, 1989. This work was supported in part by the National Science Foundation under Grant ECS-870767.

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 8930117.

1This review is by no means complete. For more details, readers may refer to [15], [33], and [18].
II. Previous Work

The first algorithm using semijoins for distributed query processing was developed by Wong\textsuperscript{2} [29]. It was used in SDD-1 and further refined in [3]. This SDD-1 algorithm is basically a hill-climbing strategy that tries to apply as many "beneficial" semijoins as possible. Theoretical aspects of semijoins were first studied in [4]. Simple queries, a subset of tree queries, were studied in [14]. Because there are only two strategies that can be optimal for such queries, the optimal strategy can be found easily. Their algorithm for general queries was improved in [1]. In that algorithm, a separate semijoin program is constructed for each joining attribute. However, no attempt was made to integrate all the derived programs. Combining the ideas of beneficial semijoins [3] and processing on each joining attribute [14], an enhanced heuristic algorithm was proposed in [6]. The fact that the reduction of one joining attribute may reduce the other attributes in the same relation was taken into account by considering the order in which the joining attributes are processed.

Given a sequence of semijoins, algorithms to identify relations not needed to produce the answer, to delete redundant semijoins, and to replace semijoins by better ones were proposed in [32]. Semijoin strategies in bus systems where broadcasting is supported and local processing is taken into account were studied in [10]. Semijoin reduction in star computer networks was considered in [17]. Remote semijoins were introduced in [26] for fragmented two-way joins. In [16], for each pair of relations, the semijoin and/or join sequence having the minimum cost is determined among all possible cases.

The methods mentioned so far achieve suboptimal solutions. On the other hand, there have been many efforts to derive optimal semijoin sequences. This problem is shown to be NP-hard in [13], [10]. The focus has been on tree queries because semijoins may work badly for cyclic queries, as proved in [4]. Methods based on dynamic programming to get an optimal semijoin sequence for chain and tree queries were studied in [8] and [9]. Dynamic programming was also used in [19] for the more general case of interleavings of joins and semijoins. These methods have a high computational complexity which limits their applicability. The special case of star queries was considered in [7] where optimal semijoin sequences are obtained by identifying properties and adding conditions on sizes of relations. Properties of optimal semijoin sequences were also used in [34] in conjunction with dynamic programming.

In summary, approaches to obtain an optimal semijoin sequence can roughly be categorized into two types. One is to enumerate all possible sequences. This approach is only good when queries reference a small number of relations because the search space gets large quickly as the number of relations increases. In fact, this rapid growth of the search space limits the usefulness of dynamic programming. The other approach is to restrict the generality, and thus reduce the size of the search space, by imposing various constraints on the problem to be solved.

In this paper, we have attempted to avoid the respective disadvantages of these two approaches. Our method to obtain an optimal sequence of semijoins can be applied to arbitrary types of tree queries without additional constraints and performs efficiently even for queries with "large" search spaces. As in [4], the problem that we address is how to optimally reach a fully reduced database state from an initial state corresponding to the given query. We have tried to develop a method that is independent of particular cost models since the choice of an appropriate cost model is often application-dependent. The assumptions that we make are common in the literature on this problem [5].

III. Problem Representation

A query is defined by a target list and a qualification. A target list consists of those relations whose columns constitute the answer to a given query. Let us call relations in the target list output relations. The tree queries under consideration have qualifications $q$ of the form of conjunctions of equijoin clauses. The qualification $q$ of a query defines a query graph $(V_q, E_q)$ where

$$V_q = \text{set of all original relations referenced by } q$$

$$E_q = \{(r_i, r_j) | i \neq j \text{ and } r_i, r_j \text{ are used in a clause in } q\}.$$  

Without loss of generality, only connected query graphs are considered. Following earlier papers [9], [34], [4], we assume that relations are at different sites and that semijoins are on single domains.

A. States

A database state is defined to be a set of relations, each of which is a subset of the Cartesian product of its domains [4], [9]. When a semijoin is performed on a relation in a database state, the relation is reduced. The database state is then transformed into a new one where the relation is replaced by its reduced version. For example, if $x_0 = \{r_1, r_2, r_3\}$ is the initial database state and a semijoin from $r_2$ to $r_1$ is applied, then the resulting state is $\{r_1 \prec r_2, r_2, r_3\}$. A sequence of semijoins performed on the given initial database state generates a sequence of transformed database states from it. The initial database state is determined from the given query.

In order to formally model the above transformations by semijoin operations, we use a graph representation for the database states. First, for each relation $r_i \in V_q$, the set of derivable relations from $r_i$, denoted $R_i$, is defined to represent the reduced relations obtainable by semijoins to $r_i$:

$$R_i = \{(r_i, \beta) | \beta \subseteq V_q - \{r_i\}\}.$$  

An element $(r_i, \beta)$ of $R_i$ represents a reduced relation where $r_i$ is the reducee and $\beta$ is the set of reducers which was used to reduce $r_i$. For example, $(r_1, \{r_2, r_3\})$ rep-
resists the reduced relation of \( r_1 \) by \( r_2 \) and \( r_3 \), i.e., \( r_1 \alpha r_2 \alpha r_3 \). Note that this notation is not ambiguous because we are dealing with tree queries. For notational convenience, \( r_i = (r_i, \{ \} ) \), \( (r_1, r_3) = (r_1, \{ r_2 \}) \).

**Definition 3.1:** A state \( x \) is a connected graph \((V_x, E_x)\) where

\[
V_x = \{ v_1, v_2, \cdots , v_n \}
\]

where \( v_i \in R_i, i = 1, \cdots , n \)

\[
E_x = \{ (v_i, v_j) | v_i \in R_i, v_j \in R_j, (r_i, r_j) \in E_q \}.
\]

The states resulting from transformations of a state \( x \) by one semijoin are called immediate successors of \( x \). The state \( x \) is called the parent of its immediate successors. Immediate successors are generated by semijoins between vertices in the parent state. If a vertex \( v_1 \) is reduced by \( v_2 \), the reduced vertex is denoted by \( v_1 \alpha v_2 \). If \( v_1 = (r_1, \beta_1) \) and \( v_2 = (r_2, \beta_2) \), then \( v_1 \alpha v_2 = (r_1, \beta_1 \cup \{ r_2 \} \cup \beta_2) \). A fully reduced vertex \( v_i \) is represented by \( v_i^f \), whose set of reducers is \( V_q - \{ r_i \} \).

**Definition 3.2:** Let \( x = (V_x, E_x), v_1 = (r_1, \beta_1), \) and \( v_2 = (r_2, \beta_2) \). A vertex \( v_1 \) is said to be included in \( v_2 \) if \( \{ r_1 \} \cup \beta_1 \subseteq \beta_2 \).

**Definition 3.3:** Given an edge \( e = (v_1, v_2) \in E_x \), the immediate successors of \( e \) in state \( x \) are defined as follows:

\[
SJE(x, e) = \begin{cases} 
\{ x_l, x_r \} & \text{if } v_1, v_2 \text{ are not included in each other,} \\
\{ x_l \} & \text{if } v_1 \text{ is included in } v_2 \text{ but not vice versa,} \\
\{ x_r \} & \text{if } v_2 \text{ is included in } v_1 \text{ but not vice versa,} \\
\emptyset & \text{otherwise}
\end{cases}
\]

where \( x_l = (V_l, E_l), V_l = V_x - \{ v_1 \} \cup \{ v_1 \alpha v_2 \}, E_l = E_x - \{ (v_i, v_j) \cup \{ (v_1 \alpha v_2) \}, \text{and } x_r = (V_r, E_r), V_r = V_x - \{ v_2 \} \cup \{ v_2 \alpha v_1 \}, E_r = E_x - \{ (v_i, v_j) \cup \{ v_2 \alpha v_1) \}. \) The successors \( x_l \) and \( x_r \) are called left and right successors of \( x \), respectively.

The set of immediate successors of state \( x \) is

\[
SI(x) = \bigcup_{e \in E_x} SJE(x, e).
\]

**Example:** Consider \( x_0 \) in Fig. 1. Let \( e_1 = (r_1, r_2) \) and \( e = (r_2, r_3) \). Then \( SJE(x_0, e_1) = \{ x_1, x_2 \} \) and \( SJE(x_0, e_2) = \{ x_3, x_4 \} \).

In principle, there can be two immediate successors for each edge. However, if vertex \( v_1 \) is included in vertex \( v_2 \), the immediate successor by semijoin from \( v_1 \) to \( v_2 \) is the same as its parent, and thus it is not generated. Note that Definition 3.3 is consistent, i.e., the immediate successors of a state are also states. When state \( x \) is generated by the semijoin from \( v_1 \) to \( v_2 \) or from \( v_2 \) to \( v_1 \), the edge \( e = (v_1, v_2) \) is called an involved edge and we say that the semijoin involves edge \( e \). Once all the immediate successors of state \( x \) are generated, \( x \) is said to be expanded.

![Fig. 1. States.](image-url)

The initial state is the given query graph, and from it successors are generated by applying Definition 3.3 until the goal state is reached. The goal state is the state whose set of reducers is \( V_q \). The search space consists of only those that are successors of the initial state, in accordance with the expansion procedure in Definition 3.3.

**B. Cost Function**

Since semijoins have most often been used under the assumption that communication costs are dominant, the cost of a semijoin from relation \( r_2 \) to relation \( r_1 \) has usually been modeled as a linear function of the size of the joining attribute of \( r_2 \) to be moved. This assumption seems to be valid for low-speed networks like ARPANET. However, as high-speed networks have become widely available, many have argued that local processing costs, such as disk access and CPU time, are no longer negligible in these environments (see, e.g., [21], [22]).

In this paper, one of our aims is to develop a query optimization method that is independent of cost models. Since a semijoin operation takes two relations as its operands, we simply say that the cost of a semijoin from vertex \( v_2 \) to vertex \( v_1 \) in a state is \( \text{cost}(v_2, v_1) \). The function \( \text{cost} \) is application-dependent and can be anything deemed appropriate. The only property that we require is the monotonicity of that function. In other words, we assume that the following inequality holds: \( \text{cost}(v_2, v_1) \geq \text{cost}(v_2, v_4) \) whenever \( v_2 = (r_2, \beta_2), v_1 = (r_1, \beta_1) \), \( v_3 = (r_2, \beta_2), v_4 = (r_1, \beta_1) \). This assumption can be justified easily. It clearly holds if only communication costs are considered, i.e., if \( \text{cost}(v_2, v_1) \)

\[
\begin{align*}
\text{cost}(r_1, r_2) & \geq \text{cost}(r_1, r_3) \\
\text{cost}(r_2, r_3) & \geq \text{cost}(r_2, r_4)
\end{align*}
\]

Thus, the overall cost of \( r_1 \) includes the cost of \( r_2 \) and \( r_3 \) and the cost of \( r_4 \).
is the cost of sending \( v_2 \) to the site of \( v_1 \). (Such cost functions are typical in the literature on semijoins.)

The cost of a transformation from a state to one of its immediate successors is the cost of the semijoin done in the transformation. The cost of a sequence of transformations is the sum of the individual transformation costs. The notation \( \text{cost}(x_1, x_2) \) represents the cost of this (these) transformation(s) if there is a trajectory from \( x_1 \) to \( x_2 \) in the search space, i.e., if \( x_2 \) is a successor of \( x_1 \). When there is more than one trajectory from \( x_1 \) to \( x_2 \), the search method only maintains the cheapest trajectory found so far between these two states, whose value will be denoted \( \text{cost}(x_1, x_2) \).

IV. Search

In the previous section, we have modeled a sequence of semijoins as a consecutive expansion of states. We now discuss how to navigate the state (or search) space efficiently in order to reach the goal state; in other words, how to find the optimal sequence of semijoins that fully reduces the initial state (in the terminology of [8], "optimal semijoin program").

In search problems, efficiency and optimality often appear to be contradictory. A conventional way of obtaining efficiency is to use a hill-climbing algorithm. Most heuristic approaches to our problem belong to this category, including the early work of Wong. On the other hand, a well-known algorithm that guarantees optimality is dynamic programming. This algorithm will find an optimal solution, but it requires expanding all the states in the search space.

In this paper, we use the A* algorithm [12]. The A* algorithm is an enhanced version of the Branch and Bound algorithm that makes use of the dynamic programming principle [28]. The Branch and Bound algorithm was developed and used mostly in operations research. The A* algorithm is popular in AI. Whereas the A* algorithm was developed originally as a graph searching algorithm, the Branch and Bound algorithm is based on the splitting principle of the solution space, i.e., it proceeds by repeatedly partitioning (also called branching) subsets of potential solutions into smaller subsets [20]. In that context, in order to avoid the inefficiency of repetitious computations, it is required (so, assumed) that if a subset has been eliminated from consideration, subsequent partitioning of other subsets will not generate any member of the eliminated set [25]. Thus, if one applies the Branch and Bound method to graph searching problems, this assumption means that no verification will be made as to whether newly generated states already exist or not. In many cases, this fact will result in a performance almost as bad as exhaustive search. This observation is well illustrated in [28].

In our problem, since the same states can be generated again and again because of the syntactic properties of semijoins, the A* algorithm is clearly more appropriate.

A. Searching with A*

In the A* algorithm, the search is controlled by a heuristic function \( f \). The state \( x \) chosen for expansion (i.e., whose immediate successors will be generated) is the one which has the smallest value \( f(x) \) among all generated states that have not been expanded so far. The purpose of \( f \) is to evaluate states in terms of their goodness in linking the initial state to the goal state. The search stops when the state chosen for expansion is the goal state. The function \( f \) considers two components: the cost of reaching \( x \) from the initial state, and the cost of reaching the goal state from \( x \). Accordingly, \( f(x) \) is defined by

\[
f(x) = g(x) + h(x)
\]

where \( g(x) \) estimates the minimum cost of a trajectory from the initial state to \( x \), and \( h(x) \) estimates the minimum cost from \( x \) to the goal state. Thus, the value \( f(x) \) estimates the minimum cost of a solution trajectory passing through \( x \). The function \( h \) represents the heuristic information that will determine the power of the algorithm.

Since understanding the A* algorithm is essential for the rest of the paper, we briefly present it. (For a complete description, the reader is referred to [25], [23].)

1. **Step 1:** Put the initial state \( x_0 \) into a list called OPEN and set the value of \( f(x_0) \) to 0. Make an empty list called CLOSED.

2. **Step 2:** Choose the state with the smallest value of \( f \) in OPEN. If it is the goal state, stop.

3. **Step 3:** Expand the chosen state and put it into CLOSED.

4. **Step 4:** For each state \( x \) generated in Step 3: a) calculate \( g(x) \), b) check whether it already exists in OPEN. If yes, keep the smaller value of \( g \). Otherwise, c) check whether \( x \) already exists in CLOSED. If yes, keep the smaller value of \( g \) and propagate the change. Otherwise, d) calculate \( h(x) \) and add the state \( x \) to OPEN. e) go to Step 2.

In order for A* to achieve our objectives of optimality and efficiency, its evaluation function must satisfy two conditions called admissibility and consistency.

The admissibility condition says that for any \( x, h(x) \leq h^*(x) \) where \( h^*(x) \) denotes the cost of an optimal trajectory from \( x \) to the goal state. It is well known that if the admissibility condition is satisfied, then A* will find an optimal solution (with respect to the given cost model) [12].

The function \( g(x) \) is calculated by the cost from the initial state to state \( x \) along the cheapest trajectory found so far in the search space, which is the sum of the costs of the semijoin operations done on this trajectory up to \( x \). In general, this \( g \) tends to overestimate the minimal cost to reach \( x \); its value is adjusted if a better trajectory is found (Step 4 above). This adjustment is done by comparing \( g \) values when an existing state is generated again and by keeping the smaller value. Thus, it may happen that a better \( g(x) \) is found after \( x \) has been expanded [Step
If other words, in order to fully reduce following observation: there will be at least one semijoin from $v_1$ to $v_2$ before any semijoin from $v_2$ to $v_1$ is done is

$$
\text{cost}(v_1 \propto T_1, v_2 \propto T_2).
$$

Proof: Before any semijoins from $v_1$ to $v_2$ and from $v_2$ to $v_1$ are done, $v_1$ can be reduced to $v_1' = v_1 \propto T_1$, and $v_2$ can be reduced to $v_2' = v_2 \propto T_2$. Then the semijoin from $v_1'$ and $v_2'$ has the minimum cost among possible future semijoins from $v_1$ to $v_2$ because of the monotonicity of the cost function.

Let $h_c(x)$ be the minimum cost of future semijoins involving $e$ before reaching the goal state from $x$.

**Lemma 4.1:** If vertex $v_2$ is not potentially fully reduced, the minimum cost among possible future semijoins from $v_1$ to $v_2$ before any semijoin from $v_2$ to $v_1$ is done is

$$
\text{cost}(v_1 \propto T_1, v_2 \propto T_2).
$$

Proof: Before any semijoins from $v_1$ to $v_2$ and from $v_2$ to $v_1$ are done, $v_1$ can be reduced to $v_1' = v_1 \propto T_1$, and $v_2$ can be reduced to $v_2' = v_2 \propto T_2$. Then the semijoin from $v_1'$ and $v_2'$ has the minimum cost among possible future semijoins from $v_1$ to $v_2$ because of the monotonicity of the cost function.

Let $h_c(x)$ be the minimum cost of future semijoins involving $e$ before reaching the goal state from $x$.

**Lemma 4.2:** $h_c(x)$ is evaluated as follows.

1) $v_1$ and $v_2$ are not potentially fully reduced:

$$
\begin{align*}
&h_c(x) = \min \left\{ \text{cost}(v_1 \propto T_1, v_2 \propto T_2), \right. \\
&\left. \quad + \text{cost}(v_1^*, v_1 \propto T_1), \right. \\
&\left. \quad + \text{cost}(v_2^*, v_2 \propto T_2) \right\}.
\end{align*}
$$

2) $v_1$ is potentially fully reduced and $v_2$ is not:

$$
\begin{align*}
&h_c(x) = \text{cost}(v_1^*, v_2 \propto T_2).
\end{align*}
$$

3) $v_2$ is potentially fully reduced and $v_1$ is not:

$$
\begin{align*}
&h_c(x) = \text{cost}(v_1^*, v_1 \propto T_2).
\end{align*}
$$

4) Both $v_1$ and $v_2$ are potentially fully reduced:

$$
\begin{align*}
&h_c(x) = 0.
\end{align*}
$$

Proof:

Case 1: From the above observation, there will be at least two semijoins involving $e$ if $v_1$ and $v_2$ are not potentially fully reduced. Clearly, $h_c(x)$ is achieved when only two semijoins are done. The expression of $h_c(x)$ then follows from Lemma 4.1.

Case 2: Once a vertex $v_1$ is potentially fully reduced, it need not be reduced further by any semijoin from $v_2$ before $e$ is fully reduced. The semijoin of minimum cost from $v_1$ will be from $v_1^*$ to $v_2 \propto T$, by Lemma 4.1.

Case 3: Similar to Case 2.

Case 4: Since $v_1$ and $v_2$ can be fully reduced without semijoins involving edge $e$, then clearly $h_c(x) = 0$.

**Theorem 4.1:** The estimate function defined as $h(x) = \Sigma_{e \in E} h_c(x)$ is a lower bound of $h^*(x)$.

Proof: Decompose $h^*(x)$ into a sum $\Sigma_{e \in E} h_c^*(x)$ where $h_c^*(x)$ is the sum of the costs of the semijoins involving edge $e$ in the optimal trajectory from $x$ to the goal state. Then from Lemma 4.2, we must have $h_c(x) \leq h_c^*(x)$ for all $e \in E_x$.

**C. Consistency**

Having proved that the $h$ function of Lemma 4.2 and Theorem 4.1 is admissible, we now prove that the consistency condition is also satisfied.
Lemma 4.3: Let $x_2$ be an immediate successor of $x_1$. Suppose that the transformation from $x_1$ to $x_2$ involves a semijoin from $v_2$ to $v_1$. Then $h(x_1) \leq cost(v_2, v_1) + h(x_2)$.

Proof: Let $e = (v_1, v_2)$ be the involved edge. The values of $h_e(x_i)$ for $e' \in E_n - \{e\}$ are the same as those in $E_2$ because the smallest future reductions of vertices in those edges have not changed. Thus, it suffices to show that $h_e(x_1) \leq cost(v_2, v_1) + h_e(x_2)$. We have to proceed by exhaustive enumeration of all the possibilities for $h_e(x_1)$ and $h_e(x_2)$.

First, suppose that $v_2$ is not fully reduced in $x_1$ and $x_2$. There are two cases to consider for $v_1$ in $x_2$; it is potentially fully reduced or it is not.

Case 1: $v_1$ is not potentially fully reduced in $x_2$.

Recall Lemma 4.2, part 1). There are two possibilities for $h_e(x_1)$.

1) $h_e(x_1) = h_e(x_1)$ + $h_e(x_1)$ where $h_e(x_1) = cost(v_1, v_1 \alpha T_1)$ and $h_e(x_1) = cost(v_1, v_1 \alpha T_1)$. Then $h_e(x_1) = cost(v_1, v_1 \alpha T_1)$ and $h_e(x_1) = cost(v_1, v_1 \alpha T_1)$.

2) $h_e(x_1) = h_e(x_1)$ + $h_e(x_1)$ where $h_e(x_1) = cost(v_1, v_1 \alpha T_1)$ and $h_e(x_1) = cost(v_1, v_1 \alpha T_1)$. Then $h_e(x_1) = cost(v_1, v_1 \alpha T_1)$ and $h_e(x_1) = cost(v_1, v_1 \alpha T_1)$.

Let 1) and 3) hold. Then $h_e(x_1) \leq h_e(x_1)$ and $h_e(x_1) \leq cost(v_1, v_1)$ by the monotonicity of the cost function. Thus, $h_e(x_1) \leq cost(v_1, v_1) + h_e(x_2)$.

1) and 4) hold. Then $h_e(x_1) = h_e(x_1)$ and $h_e(x_1) \leq cost(v_1, v_1)$. Thus, $h_e(x_1) \leq cost(v_1, v_1) + h_e(x_2)$. If 2) holds for $h_e(x_1)$, this implies that 2) is less than 1). Since 1) has just been proved to be less than or equal to $cost(v_1, v_1) + h_e(x_2)$, 2) will also be. So $h_e(x_1) \leq cost(v_1, v_1) + h_e(x_2)$.

Case 2: $v_1$ is potentially fully reduced in $x_2$.

The same possibilities for $h_e(x_1)$ have to be considered. The estimate $h_e(x_1) = cost(v_1, v_1 \alpha T_1)$ by Lemma 4.2, part 2). Let 1) hold for $h_e(x_1)$. Then $h_e(x_1) = h_e(x_1)$ and $h_e(x_1) \leq cost(v_1, v_1)$ by monotonicity so that $h_e(x_1) \leq cost(v_1, v_1) + h_e(x_2)$. If 2) is chosen for $h_e(x_1)$, we use the same argument as in Case 1.

For the cases when $v_1$ is fully reduced in $x_1$, we can prove similarly. ■

Theorem 4.2: Let $x_2$ be a successor of $x_1$. Then $h(x_1) \leq cost(x_1, x_2) + h(x_2)$.

Proof: The proof follows by repeatedly applying Lemma 4.3 and by using the definition of $cost(x_1, x_2)$.

D. State Containment

We now present a pruning mechanism to delete states during the search. The purpose of this pruning is to enhance the efficiency of the A* algorithm by immediately deleting states that we know will never be on an optimal trajectory. Thus, this pruning will preserve the optimality of the search.

Let $v_1 = (r_1, \beta_1)$ and $v_2 = (r_1, \beta_2)$. A vertex $v_1$ is more reduced than $v_2$, denoted $v_1 < v_2$, if $\beta_2 \subset \beta_1$. A state $x_1$ is said to be contained in a state $x_2$, denoted $x_1 \subset x_2$, if $x_1 \neq x_2$ and for any vertex $v'_1$ in $x_1$, there exists a vertex $v'_2$ in $x_2$ such that $v'_1 \leq v'_2$.

Lemma 4.4: If $x_1 \subset x_2$, then for any $y_2 \in SJ(x_2)$, there exists $y_1 \in SJ(x_1) \cup \{x_1\}$ such that $y_1 = y_2$ or $y_1 \subset y_2$ (denoted $y_1 \leq y_2$).

Proof: The proof is by contradiction. Suppose that for some $y_2 \in SJ(x_2)$ there is no $y_1 \in SJ(x_1) \cup \{x_1\}$ such that $y_1 \leq y_2$. With a slight abuse of notation, let $e$ be an edge in $x_1$ and $x_2$ with $e = (v_1, v_2) \in E_n$ and $e = (u_1, u_2) \in E_n$. Then $v_1 \leq u_1$ and $v_2 \leq u_2$ since $x_1 \subset x_2$.

First, suppose that $SJ(x_2, e) = \{y_2, y_3\}$ where $y_2, y_3$ are left and right successors of $x_2$ as defined in Definition 3.3. There are four cases for the immediate successors of $x_1$.

1) $SJ(x_1, e) = \{y_1, y_2\}$: The contradiction assumption means that $y_1 \not\leq y_2$ or $y_2 \not\leq y_1$. But since $v_1 \leq u_1$ and $v_2 \leq u_2$, $v_1 \leq u_1 \not\leq u_2$ and $v_2 \leq u_2 \not\leq u_1$. Because $v_1 \parallel v_2$ is in $y_1$ and $u_1 \parallel u_2$ in $y_2$, $y_1 \not\leq y_2$. Similarly, $y_1 \not\leq y_2$.

2) $SJ(x_1, e) = \{y_2\}$: From the contradiction assumption, $y_1 \not\leq y_2$ or $y_2 \not\leq y_1$, but $y_1 \not\leq y_2$. Then $y_1 \not\leq y_2$. Let $y_1 = (y_1, \beta_1)$, $y_2 = (y_2, \beta_2)$. Then $y_1 \not\leq y_2$.

3) $SJ(x_1, e) = \{y_2\}$: Argument similar to Case 2.

4) $SJ(x_1, e) = \emptyset$: Parent state $x_1 \not\leq y_2$ and $x_1 \not\leq y_2$, by the same argument as in Case 2.

In summary, the assumption is contradicted for all cases of immediate successors of $x_1$. When the immediate successor of $x_2$ is either $y_2$ or $y_2$, this lemma can be proved similarly. ■

Corollary 4.1: Let $y_1$ and $y_2$ be as defined in Lemma 4.4. Then $cost(x_1, y_1) \leq cost(x_2, y_2)$.

Proof: Since $x_1 \subset x_2$ and $y_1 \not\leq y_2$, the monotonicity of the cost function proves the corollary. ■

Let $v'_1$ be a vertex in $x_1$, and let $v'_1$ be the set of $v'_1$ which are the neighboring vertices of $v'_1$ in $x_1$. When $x_1 \subset x_2$, denote $v'_1$ to be the set of $v'_1$ in $x_2$ corresponding to $v'_1$, i.e., for each $v'_1$ in $v'_1$, the corresponding element $v'_2$ satisfies $v'_1 \leq v'_2$.

We say that state $x_1$ is strictly contained in state $x_2$, denoted $x_1 \subset x_2$, if $x_1 \subset x_2$ and for all $v'_1$ in $x_1$, $v'_1$ includes

Note that “contained” and “included” (Section III) are two different concepts; containment relates two states, while inclusion relates two vertices.
(as defined in Section III) all the vertices in $V_I^I$. It is easily verified that if $x_1 \subseteq x_2$, there exists a trajectory from $x_2$ to $x_1$ in the search space.

**Theorem 4.3:** a) If $x_1 \subseteq x_2$ and $g(x_1) \leq g^*(x_2)$, then $x_2$ can be deleted from the search space. b) If $x_1 \supseteq x_2$ and $g(x_1) \leq g(x_2)$, then $x_2$ can be deleted from the search space.

**Proof:** From Lemma 4.4, there exists $y_1$ of $x_1$ for every immediate successor $y_2$ of $x_2$ when $x_1 \subseteq x_2$. From Corollary 4.1, the transformation from $x_2$ to $y_2$ has a cost greater than or equal to that of the transformation from $x_1$ to $y_1$. If $y_1 = y_2$, state $y_2$ will be eliminated because it has a larger cost. If $y_1 \subseteq y_2$, we need to compare their own immediate successors.

Apply Lemma 4.4 to the immediate successors of $y_1$ and $y_2$. It says that for any $z_2 \in S(y_2)$, there is $z_1 \in S(y_1) \cup \{y_1\}$ such that $z_1 \subseteq z_2$. By Corollary 4.1, $cost(y_1, z_1) \leq cost(y_2, z_2)$. If $z_1 = z_2$, state $z_2$ will be eliminated because it has a larger cost than $z_1$. If $z_1 \subseteq z_2$, again we have to compare their own immediate successors. By repeatedly applying the lemma and the corollary in this way, we end up with the goal state. This proves a).

To prove b), we observe that since there is a trajectory from $x_2$ to $x_1$ when $x_1 \subseteq x_2$, future updates that improve $g(x_2)$ will also improve $g(x_1)$. Thus, if $g(x_1) \leq g(x_2)$ holds, then this inequality is preserved even when $g(x_2)$ is improved. Combining this observation with the above proof of a) proves b).

**Example:** Consider $x_4$ and $x_1$ in Fig. 1. From the definition of containment, $x_1 \subseteq x_4$. If $g(x_1) = 320$ and $g(x_4) = 400$ (the calculation is done in the next section), then $x_4$ can be pruned without loss of optimality. This proves a).

**V. ILLUSTRATIVE EXAMPLE**

Let us consider a distributed database with relations $r_1$, $r_2$, and $r_3$. Suppose that the given query is represented as $x_0$ in Fig. 1. Let the joining attribute between $r_1$ and $r_2$ be $a_{11}$, and that between $r_2$ and $r_3$ be $a_{12}$. As in [14], [1], the size $|a_{ij}|$ of attribute $a_{ij}$ of relation $r_i$ is defined as the number of bytes of the projected attribute, and the selectivity $p_{ij}$ of $a_{ij}$ is defined as the number of different values occurring in the attribute divided by the number of all possible values of the attribute. When there is a semijoin from $r_2$ on attribute $a_{12}$ to $v_I$ on attribute $a_{1k}$, the parameters of $v_I$ are changed as follows:

$$|a_{1k}| \leftarrow |a_{1k}| \cdot p_{2j}$$

$$p_{1k} \leftarrow p_{1k} \cdot p_{2j}.$$  

The size and selectivity of each attribute used in this example are given in Table I. For simplicity, only communication costs are considered in this example. These costs are given by the sizes of the attributes moved. The states in Fig. 1 are those generated in this example (they do not represent all the state space).

The first step is to generate successors according to Definition 3.3. There are four immediate successors of the initial state: $x_1, x_2, x_3, x_4$. Successors $x_1, x_2, x_3, x_4$ and $x_4$ are by semijoins from $r_2$ to $r_1$, from $r_3$ to $r_2$, from $r_5$ to $r_2$, and from $r_5$ to $r_1$, respectively. The costs of the above states are $g(x_1) = cost(r_1, r_2) = 300$, $g(x_2) = cost(r_2, r_3) = 100$, $g(x_3) = cost(r_3, r_2) = 200$, $g(x_4) = cost(r_2, r_5) = 400$.

The estimate $h(x_1)$ of successor $x_1$ is the sum of $h_v$ of edges $(r_1, r_2)$ and $(r_2, r_3)$. Let $e_1 = (v_11, v_12) = ((r_1, r_2), v_21)$ and $e_2 = (v_21, v_22) = (r_2, r_3)$. By Definition 4.2, the future smallest reductions are $v_11 = (r_1, r_2)$, $v_12 = (r_2, r_3)$, $v_21 = (r_2, r_1)$, and $v_22 = r_3$. According to Lemma 4.2, since vertices $v_11, v_12, v_21, v_22$ are not potentially fully reduced,

$$h_{r_1}(x_1) = \min \left\{ cost(v_11, v_12) + cost(v_12, v_11), \right\}$$

$$= \min \left\{ 100 \cdot 0.6 + 300 \cdot 0.7 \cdot 0.3, 300 \cdot 0.3 \right\}$$

$$= 108 + 284 = 392.$$  

Thus, $h(x_1) = h_{a_11}(x_1) + h_{r_1}(x_1) = 108 + 284 = 392$. Similarly,

$$h(x_2) = 300 \cdot 0.7 \cdot 0.3 + \min \left\{ 280 + 200 \cdot 0.7 \right\}$$

$$= 63 + 200 + 84 = 347.$$  

$$h(x_3) = \min \left\{ 100 + 300 \cdot 0.3 \cdot 0.7, 300 \cdot 0.3 \right\}$$

$$= 90 + 18 + 84 = 192.$$  

$$h(x_4) = \min \left\{ 100 + 300 \cdot 0.3 \cdot 0.7, 300 \cdot 0.3 \right\}$$

$$= 100 \cdot 0.3 \cdot 0.6.$$  

TABLE I

<table>
<thead>
<tr>
<th>Relations</th>
<th>$a_{11}$ Size</th>
<th>$p_{11}$</th>
<th>$a_{12}$ Size</th>
<th>$p_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>100</td>
<td>0.7</td>
<td>400</td>
<td>0.4</td>
</tr>
<tr>
<td>$r_2$</td>
<td>300</td>
<td>0.6</td>
<td>200</td>
<td>0.3</td>
</tr>
<tr>
<td>$r_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
+ min \{ 400 \cdot 0.7 + 200 \cdot 0.4 \cdot 0.7, 200 \\
\cdot 0.4 + 400 \cdot 0.7 \cdot 0.3 \}
= 90 + 18 + 80 + 84 = 272.

Since the value of f for state \(x_3\) is the smallest (200 + 192 = 392), state \(x_3\) is chosen for expansion. It has three immediate successors: \(x_5, x_6, x_7\). The cost to state \(x_5\) is \(g(x_3) + 300 \cdot 0.3 = 290\). The estimate \(h(x_5)\) is 100 \cdot 0.3 \cdot 0.6 + 400 \cdot 0.7 \cdot 0.3 = 18 + 84 = 102\). The cost to state \(x_6\) is \(g(x_3) + 100 = 300\). The estimate \(h(x_6)\) is \(300 \cdot 0.3 \cdot 0.7 + 400 \cdot 0.3 \cdot 0.7 = 63 + 84 = 147\). The cost to state \(x_7\) is \(g(x_3) + 400 \cdot 0.3 = 320\). When \(x_7\) is generated, by state containment pruning, \(x_4\) can be removed from the OPEN list. The estimate \(h(x_7)\) is \(\min \{ 100 + 300 \cdot 0.7 \cdot 0.3, 300 \cdot 0.3 + 100 \cdot 0.3 \cdot 0.6 \\} + 400 \cdot 0.7 \cdot 0.3 = \min \{ 100 + 63, 90 + 18 \} + 84 = 192\).

Since state \(x_5\) has f value of 392, it is chosen for expansion. The successors of \(x_5\) are: \(x_8, x_9, x_{10}\). The cost to state \(x_8\) is \(g(x_5) + 100 \cdot 0.3 \cdot 0.6 = 290 + 18 = 308\). The estimate \(h(x_8)\) is \(400 \cdot 0.7 \cdot 0.3 = 84\). The cost to state \(x_9\) is \(g(x_5) + 400 \cdot 0.3 = 290 + 120 = 410\). The estimate \(h(x_9)\) is \(100 \cdot 0.3 \cdot 0.6 + 400 \cdot 0.3 \cdot 0.7 = 18 + 84 = 102\).

Then state \(x_8\) is chosen since it has \(f(x_8) = 392\). State \(x_8\) generates state \(x_{10}\) which is the goal state. The cost to state \(x_{10}\) is \(g(x_8) + 400 \cdot 0.3 \cdot 0.7 = 308 + 84 = 392\). Since the f value of state \(x_{10}\) is the smallest, it is chosen for expansion thus ending the search. The optimal semi-join sequence is: \(x_0 \rightarrow x_3 \rightarrow x_5 \rightarrow x_8 \rightarrow x_{10}\).

VI. EXPERIMENTS

We have implemented a prototype of our method in Common Lisp and used it for experiments. Four different kinds of query graphs were tested. They are given in Fig. 2. Each query graph was executed 1000 times with different data. Attribute sizes were between 100 and 1000, and they were generated randomly. Selectivities were chosen randomly between 0 and 1. The cost model chosen was as discussed in the example in the previous section.

Table II indicates the size of the state space for each query graph. This size indicates the number of states to be expanded if dynamic programming is used instead of A*.

Since the searching effort in A* is proportional to the number of expanded states, the standard performance measure is the number of expanded states [25], [23]. Let useless states be states expanded but not on the optimal trajectory found from the initial state to the goal state. Our measure of performance is the ratio of the number of useless states to the size of state space. Let us call this uselessness. This indicates how much effort is wasted in the search space.

Figs. 3–6 present the performance of our method. The horizontal axis represents uselessness multiplied by 100. The vertical axis represents the number of cases which have the number of useless states less than or equal to the corresponding uselessness. Fig. 3 shows the performance of query graph (a). Among 1000 runs, 917 cases had no useless states, 75 had 1 useless state, 8 had 2 useless states, and no case had 3 or more useless states. The performance for query graph (b) is shown in Fig. 4. In all of 1000 cases, an optimal solution was found without exceeding 5 percent uselessness. The performance of query graphs (c) and (d) is given in Figs. 5 and 6. In Fig. 5 [for query graph (c)], all cases had less than 2 percent uselessness; in Fig. 6, [for query graph (d)], all cases had less than 1 percent uselessness.

The results of these experiments show that our method prunes the state space drastically to find an optimal solution very efficiently. The savings in terms of the number of expanded states as compared to dynamic programming is thus considerable. In addition, the results show a tendency for the improvement to increase as the complexity of the queries increases. We believe that this is a very interesting result even though more queries are yet to be tested.

In order to get a more complete evaluation of our heuristic search method, the overhead to calculate the heuristic information was also measured. The run time including the overhead was measured for each of 1000 runs, and then the average run time, denoted \(T_s\), was obtained. The run time of expanding all the states, denoted \(T_{all}\), was also measured. The value of \(T_{all}\) provides an estimate of the performance of dynamic programming since that algo-
The algorithm must expand all the states. (The machines used in the experiments were Apollo DN 4000 workstations.) The run times for the tested query graphs are given in Table III.

For query graph (a), the improvement in run time is about 70 percent; it becomes 98 and 99.9 percent for query graphs (b) and (c); finally, for query graph (d), $T_h$ is negligible compared to $T_{all}$. These results clearly indicate that the extra overhead of computing the $h$ values in our method as compared to other algorithms that must expand all the states is amply compensated by the savings in searching efforts that it permits.

VII. HANDLING ARBITRARY TARGET LISTS

So far, we have tried to fully reduce all the relations referenced by the given query independent of target lists. This section shows how to consider target lists when finding an optimal sequence of semijoins. The underlying idea

---

**TABLE III**

<table>
<thead>
<tr>
<th>Run Time (seconds)</th>
<th>Query (a)</th>
<th>Query (b)</th>
<th>Query (c)</th>
<th>Query (d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_h$</td>
<td>0.08</td>
<td>0.26</td>
<td>1.00</td>
<td>2.17</td>
</tr>
<tr>
<td>$T_{all}$</td>
<td>0.23</td>
<td>10.88</td>
<td>731.40</td>
<td>$\geq43200$ ($\geq12$ h)</td>
</tr>
</tbody>
</table>

---

**Fig. 3.**

**Fig. 4.**

**Fig. 5.**

**Fig. 6.**
is that not all the relations need to be fully reduced for queries whose target lists do not touch all the relations in the initial state.

Few papers on query optimization by semijoins considered arbitrary target lists. In [9] and [8], target lists consisting of only one relation were considered, i.e., only one relation is fully reduced. Such algorithms may not be applied to queries with arbitrary target lists. A "two-phase" approach was taken in [4] and [34] to handle this problem. In [4], semijoins are used to traverse the query graph from the leaves to the root for fully reducing the relation at the root of the tree, and then to move back to leaves in order to reduce the remaining relations in the leaves. Similarly, [34] considered the problem of finding an optimal sequence fully reducing all relations after finding an optimal sequence fully reducing one relation.

It is quite evident that there may be a better sequence of semijoins than that obtained by the two-phase approach. The objective of this section is to describe an "integrated" way of avoiding unnecessary semijoins and of still finding an optimal sequence efficiently, even when target lists are considered. Our approach for the integration is to define the generation of successors and the calculation of the estimate function \( h \) according to the target list, which is different from what we presented in Sections III and IV.

A. One Output Relation

Consider a query which has only one output relation. The initial state is still the given query graph. A goal state will be a state which has the output relation fully reduced. Let vertex \( v \) represent that output relation.

**Definition 7.1:** Let \( e = (v_1, v_2) \). Vertex \( v_1 \) is said to be closer to the output relation than \( v_2 \) if \( v \in T_1(e) \) or \( v_1 = v \).

**Definition 7.2:** For an edge \( e = (v_1, v_2) \in E_s \), the immediate successors of \( e \) in state \( x \) are defined as follows.

\[
SJE(x, e) = \begin{cases} 
\{x_1, x_r\} & \text{if } v_1 \text{ is closer and not included in } v_2, \\
\{x_1\} & \text{if } v_1 \text{ is closer and included in } v_2, \\
\{x_r\} & \text{if } v_2 \text{ is closer and included in } v_1, \\
\varnothing & \text{otherwise} 
\end{cases}
\]

where \( x_1 \) and \( x_r \) are as defined in Section III.

Like Definition 3.3, the purpose of Definition 7.2 is to generate successors for all possible semijoins. The difference is that, since the goal is to fully reduce the output relation \( v \), only semijoins that can be used to reduce \( v \) are allowed here. Whether a semijoin can be used to reduce \( v \) is determined by verifying that its reducer is included in \( v \). Thus, Definition 7.2 is consistent with Definition 3.3, but augmented with the conditions that the reducer is not included in \( v \) and that the reducee is closer to \( v \).

**Lemma 7.1:** The minimum cost \( h_s(x) \) to fully reduce an edge \( e = (v_1, v_2) \in E_s \) is the following.

1) \( v_1, v_2 \) are not potentially fully reduced.

If \( v_1 \) is closer and \( v_2 \) is not included in \( v \),

\[
h_s(x) = \min \left\{ \text{cost}(v_1 \sim T_1, v_2 \sim T) + \text{cost}(v_2 \sim T, v_1 \sim T) \right\}.
\]

If \( v_2 \) is closer and \( v_1 \) is not included in \( v \),

\[
h_s(x) = \min \left\{ \text{cost}(v_1 \sim T_1, v_2 \sim T), \text{cost}(v_2 \sim T, v_1 \sim T) \right\}.
\]

2) \( v_1 \) is not potentially fully reduced, \( v_2 \) is potentially fully reduced, \( v_1 \) is closer, \( v_2 \) is not included in \( v \).

\[
h_s(x) = \text{cost}(v_1 \sim T_1, v_2 \sim T_1).
\]

3) \( v_1 \) is potentially fully reduced, \( v_2 \) is not potentially fully reduced, \( v_2 \) is closer, \( v_1 \) is not included in \( v \).

\[
h_s(x) = \text{cost}(v_1 \sim T_1, v_2 \sim T_1).
\]

4) Otherwise,

\[
h_s(x) = 0.
\]

**Proof:**

*Case 1:* Since \( v_1 \) is closer, \( v_2 \) is not included in \( v \), and \( v_1 \) is not potentially fully reduced, there will be at least a semijoin from \( v_2 \) to \( v_1 \) in order to fully reduce \( v \). Also, since \( v_2 \) is not potentially fully reduced, \( v_2 \) may be reduced by a semijoin from \( v_1 \) before the semijoin from \( v_2 \) to \( v_1 \) is done. Lemma 4.1 gives the minimum cost for those possible semijoins. The other case is also proved similarly.

*Case 2:* In this case, only one semijoin is allowed, namely, a semijoin from \( v_2 \) to \( v_1 \). Again, Lemma 4.1 is used.

*Case 3:* Similar to Case 2.

The estimate function \( h(x) = \sum_{e \in E_s} h_s(x) \) can be proved to be a lower bound as done in Theorem 4.1. The consistency condition can also be proved as before.

B. Multiple Output Relations

Finally, we consider the case where there is more than one output relation. The initial state is the given query graph, and the goal state is the state where all the output relations are fully reduced. Roughly speaking, the generation of successors and the calculation of \( h \) in this general case can be viewed as a combination of the two extreme cases studied so far: fully reduce all relations, fully reduce one relation.
Lemma 7.2: Edges in any path connecting output relations in a state will be fully reduced.

Proof: Take a path connecting output relations $u_1$ and $u_2$. This path is unique because the state is a tree graph. Let $e = (v_1, v_2)$ be one of the edges in that path. Suppose that $v_1$ and $v_2$ are not the output relations. Consider the case where $u_1$ is in $T_i(e)$ and $u_2$ in $T_j(e)$. In reducing $u_2$, vertices of $T_i(e)$ can be used only after they are used to reduce $v_1$ and $v_2$. Similarly, when reducing $u_1$, any vertex of $T_j(e)$ can be used only after they are used to reduce $v_1$ and $v_2$. This implies that $v_1$ and $v_2$ will be fully reduced, and so edge $e$ will be. The proof is similar in the other cases.

Let us call edges in any path connecting output relations full edges; other edges are called half edges. If edge $e$ in state $x$ is a full edge, Definition 3.3 for the generation of successors of $e$ and Lemma 4.2 for the calculation of $h_x(e)$ can be applied because its end vertices will be fully reduced.

If edge $e$ is a half edge, all the output relations are in the left or right subtree of $e$. We can apply Definition 7.2 and Lemma 7.1 to such an edge $e$ as follows. Let $u_1, u_2, \ldots, u_m$ be output relations. Suppose that they are in $T_i$ of half edge $e = (v_1, v_2)$. Then there are unique paths from $v_1$ of each of $u_1, \ldots, u_m$ in state $x$. All the paths have a common vertex $u$ which belongs to each of them and needs to be fully reduced. Note that the vertex $u$ may be an output relation or not. This common vertex $u$ is the one that shall be treated as the "output relation" when applying Definition 7.2 and Lemma 7.1 to $e$. It then follows that the estimate function $h_x(u)$ is still a lower bound because $h_x(u)$ has been proved to be the minimum cost for each edge whether it is a full edge or a half edge. It can also be shown that the consistency condition is satisfied as before.

C. Example

Suppose that the same query as in Section V is given, but that $r_2$ is the only output relation. As in Section V, $x_0$ in Fig. 1 is the initial state, and there are four immediate successors of the initial state: $x_1, x_2, x_3, x_4$. According to Lemma 7.1, however, heuristic function values of states are calculated differently.

For state $x_1$, let $e_1 = (v_{11}, v_{12}) = ((r_1 \not\subset r_2), r_2)$ and $e_2 = (v_{21}, v_{22}) = (r_2, r_3)$. For $e_1$, $v_{11}$ and $v_{12}$ are not potentially fully reduced, $v_{12}$ is closer, and $v_{11}$ is not included in the output relation. For $e_2$, $v_{21}$ and $v_{22}$ are not potentially fully reduced, $v_{21}$ is closer, and $v_{22}$ is not included in the output relation. From Lemma 7.1,

$$h_{x_1}(x_1) = \min \{ \text{cost}(v_{11}^*, v_{12}), \text{cost}(v_{12}, v_{11}) \}$$

$$= \min \{ 60, 90 + 18 \}$$

$$h_{x_2}(x_1) = \min \{ \text{cost}(v_{22}, v_{21}), \text{cost}(v_{21}, v_{22}) \}$$

$$+ \text{cost}(v_{22}, v_{21}) \}$$

$$= \min \{ 200, 280 + 56 \}.$$
ACKNOWLEDGMENT

The authors would like to thank the anonymous referees for their helpful comments and suggestions.

REFERENCES


Hyuck Yoo (S'89) received the B.S. and M.S. degrees in electronic engineering from the Seoul National University, Seoul, Korea, in 1982 and 1984, respectively, and the M.S. degree in computer engineering from the University of Michigan, Ann Arbor, in 1986.

He is currently completing the Ph.D. degree in computer science at the University of Michigan. His research interests include database systems, knowledge-based systems, distributed computing, and computer networks.

Mr. Yoo is a member of Tau Beta Pi and the IEEE Computer Society.

Stéphane Lafortune (S'78-M'86) was born in Montreal, P.Q., Canada, in 1958. He received the B.Eng. degree from l’Ecole Polytechnique de Montreal in 1980, the M.Eng. degree from McGill University, Montreal, in 1982, and the Ph.D. degree from the University of California, Berkeley, in 1986, all in electrical engineering.

Since September 1986 he has been an Assistant Professor of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. His current research interests are in discrete event systems and in database systems.