A new approach to join reordering in query optimization

Mostafa Pilehvar

A Thesis

In

The Department

Of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

March 2005

Your file Votre référence
ISBN: 0-494-04207-9
Our file Notre référence
ISBN: 0-494-04207-9

Abstract

A new approach to join reordering in query optimization

Mostafa Pilehvar

A major task in query optimization is finding an optimal or near-optimal order to perform join operations. Most optimizers perform this join reordering using an exhaustive search algorithm with a dynamic programming technique. This algorithm always gives the optimal join ordering, but is very expensive. An alternative *greedy* approach is very efficient, but is not guaranteed to produce the optimal order.

In this thesis, we introduce a new algorithm called AO^* to find the optimal join ordering for a given set of relations. Our algorithm is based on a well-known problem reduction technique of the same name proposed in the context of decomposable production systems. In our algorithm, we build an AND/OR graph from the relations in the query which represents all possible join orders, and an optimal join ordering is an optimal-cost path in this graph.

The AO^* algorithm is guaranteed to produce the optimal join ordering, as is dynamic programming. While the worst-case performance cost of AO^* is comparable to that of dynamic programming, in most instances, it is far more efficient. We compared the performance of AO^* with that of dynamic programming and the greedy approach on star-, chain-, circular-, and clique-shaped queries. Our results show that for the first three types of queries, AO^* outperforms dynamic programming dramatically. Another finding is that certain properties of the optimal join order have an impact on the performance of AO^*.

Dedication

To my wife Sepideh and my kids Fara and Radbod.

Acknowledgements

At the beginning of my thesis I would like to thank all those people who made this thesis possible and an enjoyable experience for me. First of all I wish to express my sincere gratitude to Lata Narayanan and Gosta Grahne, who guided this work and helped whenever I was in need. I think their presence in Concordia University was the best thing that could happen to me and my thesis.

Sunil Rottoo implemented all three algorithms studied in this thesis. I am grateful to him for his efforts, without which this thesis would not be completed.

I am grateful to the members of the University for their support, especially to Halina Monkiewicz.

Finally, I would like to express my deepest gratitude for the constant support, understanding that I received from my wife Sepideh during the past years.

Contents

Tables

Figures

Abbreviations

σ	Selection of a tuple in a relation.
π	Projection of a table on (an) attribute(s).
$\bowtie$	The binary operation of join.
$R(X,Y)$	R is a relation with attribute sets of X and Y.
$B(R)$	The number of blocks needed to hold all the tuples of relation R.
$T(R)$	The number of tuples of relation R.
$V(R,a)$	The number of tuples with different values for attribute a in relation R.

Chapter 1

Introduction

The importance of query optimization in relational databases is widely recognized. Given a query, there are many plans that a database management system (DBMS) can follow to answer it. All plans will have the same final output but vary in costs, i.e., the amount of time and/or resources that they need to reach the final answer. Finding the plan with the least cost is the goal of *query optimization*. The performance of database applications can be improved significantly by using query optimization.

The flow of a query through a DBMS is shown in Figure 1.

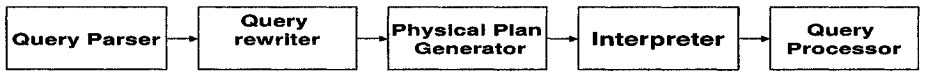


Figure 1.1: Major parts of a query processor.

The above mentioned system modules have the following functionalities:

- *Query parser:* In this stage the validity of the query is checked and a *parse tree* which represents the query and its structure is constructed.
- *Query rewriter:* In this stage the constructed parse tree is converted to an initial query plan, which is an algebraic expression representing the query. It is then transformed to an equivalent plan called the *logical query plan*, which is expected to be executed much faster.
- *Physical plan generator:* In this stage the output of previous stage is transformed to a *query physical plan*. The result of this stage is usually called the *access plan*.

  Query rewriting and *Physical plan generation* together comprise the process of *query optimization*.
- *Interpreter:* In this stage the output of the previous stage, the *access plan*, is transformed into calls to the next stage, the *query processor*.
- *Query Processor:* In this stage the query is finally executed.

Queries in a DBMS are posed either by interactive users or are embedded in general purpose programming languages. An interactive query goes through the entire path shown in Fig. 1, while an embedded query goes through the first three stages only once, when the original program is first compiled. That is why these stages are called the *query compilation* step. The code generated by the interpreter is stored in a database and is called and executed by the last stage, the *query processor*, whenever control reaches that point during run time. Thus, unless something changes, the process of query optimization is not repeated. So in the query optimization stage, we should choose a query plan that takes the least resources, and to do so we should select, amongst different algebraically equivalent forms, the one which has the most efficient algorithm to answer the query, and for every operation in that plan the algorithm that leads to the less expensive implementation [24].

A query optimizer typically translates non-procedural queries into procedural plans for execution by constructing many different alternative and equivalent plans, estimating their execution costs and choosing the plan with the least expensive estimated plan. The baseline approach, *exhaustive search*, is to consider all combinations of choices of the set of access plans. Increasing this set, while it increases the compilation time of the query, improves the possibility of finding a better plan but does not necessarily guarantee it. Since the number of alternative access plans for a query grows very fast, query optimization becomes an expensive process. A major task in the design of a query optimizer is to ensure that the set of access plans contains enough efficient plans.

The major decision that the optimizer should make is the order of different tables to join in the query. Usually join is a 2-way operation, so the optimizer must create plans that achieve n-way joins as a sequence of 2-way operations. When there are more than a few tables involved in the query, the number of those tables dominates the number of possible alternative access plans. The number of possible alternative access plans in a query with n tables is $n!$.

1.1 Joins and Join Trees

The *join* operation, denoted by $\bowtie$, is used to combine *related tuples* from two relations into single tuples. This operation is very important for any relational database with more than a single relation, because it allows the processing of relationships among relations [22][25]. It is defined as follows:

join: $R \bowtie_{t.A\theta u.B} S = \{\langle t,u \rangle$: $t \in R$ and $u \in S$ and t.A θ u.B $\}$

where t is a tuple in relation R, u is a tuple in relation S and these two tuples satisfy the join condition θ. Also t and u should have the same domain. The join condition θ is one of the comparison operators in the set $\{=, \leq, \leq, \geq, \geq, \neq\}$. This type of join is called

Theta Join. The most common join involves a join condition with the equality operator. Such a join is called an *Equijoin*. Notice that in the result of an *Equijoin* we always have one or more pairs of attributes that have identical values in every tuple. Because one of each pair of attributes with identical values is superfluous, a new operation called *Natural Join* was created to get rid of the second(superfluous) attribute in an equijoin condition [20][7].

An important property of the join operation is *associativity*, meaning:

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

Typically an algebraic query is represented by a *query tree* whose leaf nodes are database relations and non-leaf nodes are algebraic operators. An intermediate node indicates the application of the corresponding operator on the relations constructed by its children. The result of such an operation then is sent further up. Therefore, the edges of the tree represent the query itself. In a complicated query, the number of all possible query trees will be enormous. When we have the join of two relations, we will conventionally select the one with smaller size as the left argument. Algorithms used to join relations like, one-pass, nested-loop, and indexed, work best when the left argument is the smaller one. When we have only two relations, there are only two possible alternatives for the join tree. When the join involves more than two relations the number of possible alternative join trees increases very fast. Based on the shape and location of relations in the join tree we can have three different and distinguishable types of trees[27].

- a *left-deep* tree is a binary tree whose right children are always leaves.
- a *right-deep* tree is a binary tree whose left children are always leaves.
- a *bushy* tree is neither left-deep nor right-deep.

These different types of join trees are depicted in Figure 1.2.

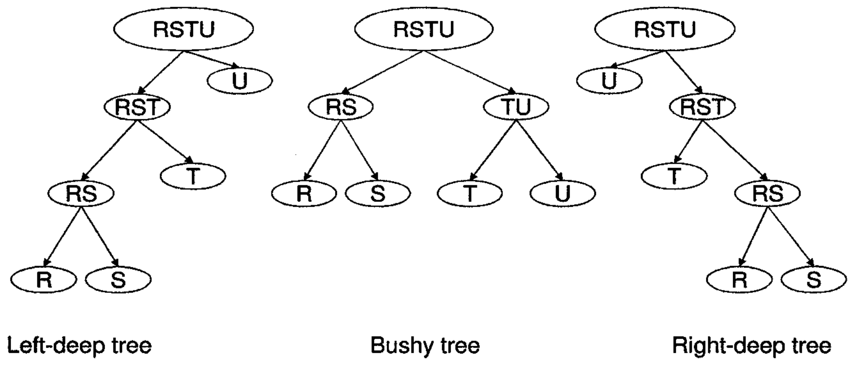


Figure 1.2: Three types of join trees for the join $R \bowtie S \bowtie T \bowtie W$, where the left-deep tree corresponds to $(((R \bowtie S) \bowtie T) \bowtie U)$, the bushy tree corresponds to $((R \bowtie S) \bowtie (T \bowtie U))$ and the right-deep tree corresponds to $(((U \bowtie (T \bowtie (R \bowtie S)))$

To reduce the size of the space that the search strategy has to explore, some alternative access plans or query trees that are likely to be sub-optimal should be eliminated by the application of several useful heuristics. One of them is to consider only left-deep trees as possible join orders. There are two advantages to using such a heuristic:

- With a given number of leaves, the number of left-deep trees is much smaller than the number of all trees. Therefore, the search space will decrease significantly.

- Query plans based on left-deep trees interact efficiently with common join algorithms like, one-pass and nested-loop joins.

The main disadvantage of using such a heuristic is that it is quite possible that the optimal plan is a bushy tree and by using this heuristic we would not find the optimal plan.

1.2 Cost Estimation

As already said, given a query there are many logically equivalent algebraic expressions and for each of these expressions, there are many ways to implement them as operators. Apart from the complexity of enumeration of the search space, the question of making the decision of choosing the operator with least consumption of resources, remains. Resources may be CPU time, I/O cost, memory, communication bandwith, or a combination of these. In this thesis, we use the total size of all intermediate relations needed to achieve an operation as the cost of the operation. Therefore, given an operator tree of a query, finding an accurate and efficient evaluation of cost is of high importance. Obviously we cannot know these costs exactly without executing the plan, and we surely don't want to execute more than one plan for a query. Thus we are forced to estimate the cost of a plan without executing it. Preliminary to our discussion of physical plan enumeration, is a consideration of how to estimate costs of such plans accurately. Such estimates are based on parameters of the data that must be either computed exactly from the data or estimated by a process based on a solid framework.

The basic estimation framework is derived from the System-R approach[5]:

- Collect statistical summaries of data that has been stored.
- Given an operator and the statistical summary of its input data streams, determine the:
    - Statistical summary of the output data stream such as the size of an intermediate relation or the number of attributes with different values.
    - Estimated cost of executing the operation e.g join operation.

Given values for these parameters, we may make a number of reasonable estimates of relation sizes that can be used to predict the cost of a complete physical plan. Since our enumeration is based on queries only using join operation, here we briefly explain our size estimation assumptions for the join operation.

1.2.1 Estimation of the Size of Join

We will consider here only the natural join, that is, we consider the join $R(X, Y) \bowtie S(Y, Z)$, but initially we assume that Y is a single attribute, while X and Z can be any set of attributes. Following [10], we denote the size of a relation R by $T(R)$ and the number of different possible values for attribute a in relation R by $V(R, a)$. The problem is that there exist different possibilities depending on the distribution of Y values in R and S. For example two relations could have disjoint sets of Y-values. In this case the join is empty and the size of $R \bowtie S$ could be zero. Alternatively, Y might be the key in S and foreign key in R. In this case, $T(R \bowtie S) = T(R)$. As another example, all the tuples of R and S could have the same Y-value, in which case $T(R \bowtie S) = T(R)T(S)$.

To focus on the most common situations, we will use these two simplifying assumptions[10]:

- *Containment of Value Sets.* If Y is an attribute appearing in several relations, then each relation chooses its values from the front of a fixed list of values $y_1, y_2, y_3, ...$ and has all the values in that prefix. As a consequence, if R and S are two relations with an attribute Y, and $V(R,Y) \leq V(S,Y)$, then every Y-value of R will be a Y-value of S.

- *Preservation of Value Sets.* If we join a relation R with another relation, then an attribute A that is not a join attribute does not lose values from its set of possible values. More precisely, if A is an attribute of R but not of S, then $V(R \bowtie S, A) = V(R, A)$

Under these assumptions, we can estimate the size of $R(X,Y) \bowtie S(Y,Z)$ as :

- $T(R \bowtie S) = T(R)T(S)/max(V(R,Y), V(S,Y))$

If Y represents several attributes in the join, the formula above can be generalized. For example, suppose we want to join $R(x, y_1, y_2) \bowtie S(y_1, y_2, z)$. Then

- $T(R \bowtie S) = T(R)T(S)/max(V(R,y_1), V(S,y_1))max(V(R,y_2), V(S,y_2))$

Finally, in the general case of the join $R_1 \bowtie R_2 \bowtie ... \bowtie R_n$, the size of the join can be estimated using the following rule:

- Start with the product of the number of tuples in each relation. Then, for each attribute A appearing at least twice, divide by all but the least of the $V(R,A)$'s.

1.3 Organization of the Thesis

Some of the existing query optimizers are introduced in the next chapter. These can be categorized as top-down or bottom-up. In the top-down category, the Volcano algorithm is introduced. In the bottom-up category, the Branch-and-Bound Plan, System R and Starburst are briefly introduced. The dynamic programming and greedy algorithms are explained in detail. In the third chapter, the AO^* algorithm is first introduced for general AND/OR graphs and then our adaptation for join optimization is described. In the fourth chapter, our experimental results based on running AO^* on four types of queries are given. The last chapter contains our conclusions and suggestions for future work.

1.4 Contribution of this Thesis

In this thesis, we propose a new algorithm called AO^* for the well-known problem of join reordering in the query optimization field. AO^* is guaranteed to produce an optimal join ordering. We studied various factors that affect the performance of this algorithm, including the shape of the query, the distribution of the sizes of relations and the value sets, and the shape of the final solution. To capture the effect of the shape of the final solution, we propose a new metric, called cost index, for a join tree. We compared the performance of our algorithm with that of the previously known dynamic programming and greedy algorithms.

Chapter 2

Existing Query Optimizers

There are two broad approaches to explore the space of possible and equivalent access plans [5]:

- *Top-down:* In this approach, we start working on the tree of the logical query plan from the root to leaf nodes. For each possible implementation of the operation at root, we take into account all the possible ways to execute it and then evaluate and compute the cost of each combination, choosing the one with least cost.

- *Bottom-up:* In this approach, we compute every sub-expression in the logical query plan tree, and recursively combining all the possible combinations, compute the least expensive one to produce the root.

We will consider all the existing work in the area of query optimization under these two broad categories. We present a set of relations against which we may run some examples to show how the different algorithms perform. The reference relations are shown in Table 2.1.

2.1 TOP-DOWN

We briefly explain VOLCANO, the major enumerator in this category, which is based on the EXODUS algorithm [17].

S	R	U	W
$T(S) = 1000$	$T(R) = 100$	$T(U) = 100$	$T(W) = 1000$
$V(S,a) = 100$			$V(W,a) = 1000$
$V(S,b) = 100$	$V(R,b) = 100$		
	$V(R,c) = 10$	V(U,c)=10	
		$V(U,d) = 100$	$V(W,d) = 100$

Table 2.1: Reference relations used to illustrate various algorithms.

2.1.1 Volcano

The Volcano optimizer generator is based on three fundamental design decisions:

1- Query processing, both optimization and execution, are presumed to be based on algebraic techniques.

2- Rules are used to specify the data model and its properties. In this case the only needed rule is $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

3- Rules are transformed by an optimizer generator into source code in a standard programming language(C) to be compiled and linked with the other DBMS modules.

The Volcano Optimizer Generator follows the generator-transformation-based approach (first demonstrated by EXODUS [9]) where properties of the data model are input to the optimizer generator. Logical operator declarations, algorithm declarations, logical transformation rules, and implementation rules are contained in the model specification. This file is parsed by the optimizer generator and C code is generated.

In addition to the model input file, the optimizer implementer supplies functions to determine the cost of using each algorithm, algorithm applicability, property derivation functions, and property comparison functions. This code is compiled and linked with the compiled generated code and the compiled data model-independent code to produce an optimizer.

The generated optimizer accepts as input a logical algebra expression(query) and physical property(or properties) pair. There are three principal assumptions made about inputs to the generator and generated optimizers. First, it is assumed a logical expression to be optimized is syntactically and semantically correct. The second assumption is that the transformation rule set is *sound*. The third assumption is that the query to be optimized can be expressed as a tree of logical operators and associated arguments, each operator having one or more inputs and a single output, and that this query can be optimized using algebraic, cost-based techniques, i.e., algebraic equivalence rules and cost information about the implementation of the algebraic operators.

A query passed to the optimizer is expressed as a tree of logical operators and their associated arguments, i.e., an expression in the logical algebra. The logical operator set is declared in the model input file and compiled into the optimizer. Along with the query, the optimizer is also passed a set of physical properties that the query execution plan found by the optimizer should deliver. The output of the optimizer is expressed as a tree of algorithms, i.e., an expression in the physical algebra, that delivers the requested physical properties.

Optimization consists of mapping a logical algebra expression into a relatively low cost, equivalent physical algebra expression. In another words, the optimizer reorders operators and selects implementation algorithm for the operators. Rules that express the equivalence of logical algebra expressions, e.g., commutativity and associativity, are represented in the model input file by *transformation rules*. The possible mappings of logical operators to the algorithms that implement them are expressed using *implementation rules*. Additional information about rules, e.g., applicability, can be specified by attaching *condition code* to the rules to be invoked after pattern matching has succeeded [17].

2.2 BOTTOM-UP

In this category we review these algorithms, Branch-and-Bound Plan, Selinger-Style Optimization, Starburst, Dynamic Programming.

2.2.1 Branch-and-Bound Plan

In this algorithm, we begin the enumeration by using heuristics to find a good physical plan for the entire logical plan with a cost of, e.g., C. While considering all the subqueries, we will discard all those with costs greater than C. Meanwhile, if we find a plan with cost less than C, we will replace C with this new value.

An important advantage of this algorithm is that we can stop our search whenever we find a plan with a reasonable cost, ignoring any possible better plan taking into account the required time for the search [28].

2.2.2 Dynamic Programming

In this approach, for every subquery we keep only the plan of least cost. Moving upward in the tree, we consider all the possible implementations of each node and choose the one with least cost. In this fashion, we fill in a table of costs and remember the minimum information we need to proceed to a conclusion [3].

This algorithm is the base line to develop other algorithms by applying some helpful heuristics to reduce the size and cost of intermediate relations. Some of those algorithms are described in this chapter. We can also decide either to pick all the orders of joining many relations or consider a subset of them or even use a heuristic to choose one. Based on this decision we may have different results with different costs.

For example, consider the S, R, U, W schema given in Table 2.1 and the requested query $S \bowtie R \bowtie U \bowtie W$.[1] As stated before, we use the sum of sizes of intermediate

[1] The example used here is taken from chapter 7 of [10]. We use it to provide a comparison of different algorithms on a well-known schema.

relations as the cost function. If we apply this algorithm on the reference schema, for singleton sets, the sizes, costs and best plans are shown in Table 2.2. Since singleton relations are base relations and not intermediate relations, their cost is 0.

	S	R	U	W
Size	1000	100	100	1000
Cost	0	0	0	0
Best Plan	S	R	U	W

Table 2.2: The table for singleton sets

Now, consider pairs of relations. The cost for each is 0, since there are still no intermediate relations in a join of two. The sizes of the resulting relations are computed by the previously described formula. The results are summarized in Table 2.3.

	S, R	S, U	S, W	R, U	R, W	U, W
Size	1000	100000	10000	1000	100000	1000
Cost	0	0	0	0	0	0
Best Plan	$S \bowtie R$	$S \bowtie U$	$S \bowtie W$	$R \bowtie U$	$R \bowtie W$	$U \bowtie W$

Table 2.3: The table for pairs of relations

The next step is to consider the join of three relations. The result of the sizes and the costs of each join is presented in Table 2.4.

At this stage we should consider the join of all four relations. The size estimate for this relation is 1000 tuples, so the true cost is essentially in the construction of intermediate relations. We can compute the join of all four in two ways, either picking three relations and join it with the fourth one or picking two relations of two relations to be joined later.

Of course, if we only consider left-deep trees then the second type of plan, being a bushy tree need not to be considered. Table 2.5 summarizes the seven possible ways

	S, R, W	R, S, U	R, W, U	S, W, U
Size	10000	50000	10000	2000
Cost	1000	1000	1000	1000
Best Plan	$(S \bowtie R) \bowtie W$	$(R \bowtie S) \bowtie U$	$(W \bowtie U) \bowtie R$	$(W \bowtie U) \bowtie S$

Table 2.4: The table for triples of relations

to group the joins:

So the optimum plan found by this algorithm would be:

$$((S \bowtie R) \bowtie (U \bowtie W))$$

with the cost of 2000.

2.2.2.1 Complexity

The number of joins needed to be evaluated to find the final solution in dynamic programming is:

$N = \sum_{i=1}^{n}(2^{i-1} \frac{n!}{i!(n-i)!})$ - $2^n + 1$.

In [14], N is shown to be $O(3^n)$, where n is the number of relations to be joined.

2.2.3 Selinger-Style Optimization or System R

This algorithm is based on dynamic programming. In addition to the plan with least cost for each subquery, we keep track of some useful subqueries even with higher costs but interesting results. This optimizer uses heuristics to limit the join sequences which should be evaluated. One of these heuristics is constructing only left-deep joins. Another major heuristic used in this approach is deferring Cartesian products in the join sequence as late as possible. These two heuristics help to make the search strategy much more efficient in most cases. However there are some situations in which the optimal solution is eliminated [21].

Grouping	Cost
$(((S \bowtie R) \bowtie U) \bowtie W)$	11000
$(((S \bowtie R) \bowtie W) \bowtie U)$	11000
$(((U \bowtie W) \bowtie S) \bowtie T)$	11000
$(((U \bowtie W) \bowtie R) \bowtie S)$	11000
$((S \bowtie R) \bowtie (U \bowtie W))$	2000
$((S \bowtie U) \bowtie (R \bowtie W))$	200000
$((S \bowtie W) \bowtie (R \bowtie U))$	11000

Table 2.5: Join grouping and their costs

2.2.4 Starburst

In this approach, for a given query, after parsing the query and storing it in an internal database called the *Query Graph Model (QGM)*, it is sent to the next step called *plan optimization* in which some alternative *query execution plans (QEPs)* and output for executing them with the least estimated costs, are evaluated. The plan optimizer in Starburst consists of two separate and extensible sub-components: the *join enumerator* which enumerates the join orders in which the relations can be joined, and a rule-based *plan generator*, which generates and evaluates the cost of alternative QEPs.

Starburst uses a *generate and filter strategy* for join enumeration. It will generate a superset of *feasible* access plans and then filters *non-feasible* ones by applying some feasibility criteria. Some of these criteria are globally valid and some are optional and therefore can be parameterized for reducing the search space. Among those optional criteria are heuristics deferring Cartesian product and avoiding bushy trees, which can be parameterized either by the user or the system.[8]

The number of joins evaluated for a query depends on two classes of factors: 1) characteristics of the query, such as the number of relations, the number of predicates, and the shape of the query, indicating how the relations are connected by the relations and 2) join feasibility criteria, like whether the bushy joins are allowed or not. Regarding the shape of queries, two distinguishable kind of queries, *linear* and *star* queries, were studied. In the *linear* queries each relation is connected to only two other relations except the first and the last ones which are connected to only one relation. There are some sharper upper bounds for these specific kind of queries [14], which are given below:

- **Complexity of *linear* queries with bushy trees:** Using dynamic programming to optimize a linear query with n relations, and allowing bushy trees, requires evaluating $(n^3 - n)/6$ feasible joins.
- **Complexity of *linear* queries without bushy trees:** Using dynamic programming to optimize a linear query with n relations, and disallowing bushy trees, requires evaluating $(n-1)^2$ feasible joins.
- **Complexity of *star* queries:** Using dynamic programming to optimize a star query with n relations requires evaluating $(n-1)2^{n-2}$ feasible joins.

2.2.5 Greedy algorithm

In order to avoid exhaustive search in dynamic programming, we can choose a *greedy* algorithm in which, when we make a decision on a join order, never reconsider that decision again. One of the greedy algorithms is when we consider only left-deep trees. By using this heuristic we keep the number of intermediate relations as small as possible. The way it works is that, we start by evaluating all the pairs of relations, choosing the pair with smallest estimate of join result. In the next step, we only consider the possibility of join of this pair with the other relations and pick the relation which creates the smallest join. After choosing the next relation, we will once again consider the join of the result with other remaining relations until we obtain the join of all relations [13].

Consider the same schema and the same query, as in section 2.2.2, this time optimized with the Greedy algorithm, using the same cost function. To find the best plan, the first step is as per described in Table 2.2. At that point, we choose, for example, $(S \bowtie R)$ as our first two relations to be joined. In the next step we consider all the possible joins only with this pair, getting $(S \bowtie R \bowtie U)$ and so on till finally we get $(((S \bowtie R) \bowtie U) \bowtie W)$ as the resulting plan, whose cost is 11000. As shown in Table 2.4, the optimal plan is $((S \bowtie R) \bowtie (U \bowtie W))$, whose cost is 2000. Note that the optimal plan is not a left-deep tree, while the greedy algorithm only considers left-deep trees.

This raises the question of whether the greedy algorithm produces an optimal plan in those situations where the optimal plan is a left-deep tree. The answer is unfortunately no, as shown by the example given in Table 2.5 which shows a 9-relation schema.

The optimal ordering of the query found by Dynamic Programming, is:
$((((((((M \bowtie N) \bowtie P) \bowtie Q) \bowtie R) \bowtie S) \bowtie Z) \bowtie U) \bowtie W)$

but the Greedy algorithm finds another left-deep tree as its solution namely:
$((((((((W \bowtie U) \bowtie Z) \bowtie N) \bowtie S) \bowtie R) \bowtie Q) \bowtie M) \bowtie P)$.

Figures 2.1 and 2.2 show the final solution of such a query by Dynamic Programming algorithm and the Greedy algorithm repectively.

In the graphs representing join trees, in each node depicting a join two values are given, the first is the estimated size of intermediate join represented by the node and the second one is the estimated cost of that specific sub-query.

As seen in these two figures, the optimal join order has a cost of 23200 while the suggested join by the Greedy algorithm has a cost of 2300000.

2.2.5.1 Complexity

The number of joins to find the final solution in this algorithm is

$N = (n-1)^2$ which is $O(n^2)$, where n is the number of relations to be joined.

R	S	Z	U	Q
$T(R) = 1000$	$T(S) = 1000$	$T(Z) = 1000$	$T(U) = 1000$	$T(Q) = 1000$
$V(R,a) = 100$				$V(Q,a) = 100$
$V(R,b) = 100$	$V(S,b) = 100$			
	$V(S,c) = 100$	V(Z,c)=100		
		$V(Z,d) = 100$	$V(U,d) = 100$	
			$V(U,e) = 100$	$V(Q,e) = 100$

M	N	P	W
$T(M) = 1000$	$T(N) = 1000$	$T(P) = 1000$	T(W)=1000
$V(M,a) = 100$		$V(P,a) = 100$	
	$V(N,b) = 100$		
		$V(P,c) = 100$	$V(W,c) = 100$
$V(M,d) = 100$			
	$V(N,e) = 100$		$V(W,e) = 100$

Table 2.6: Reference relations showing that the Greedy algorithm does not produce an optimal plan even when the optimal solution is a left-deep tree.

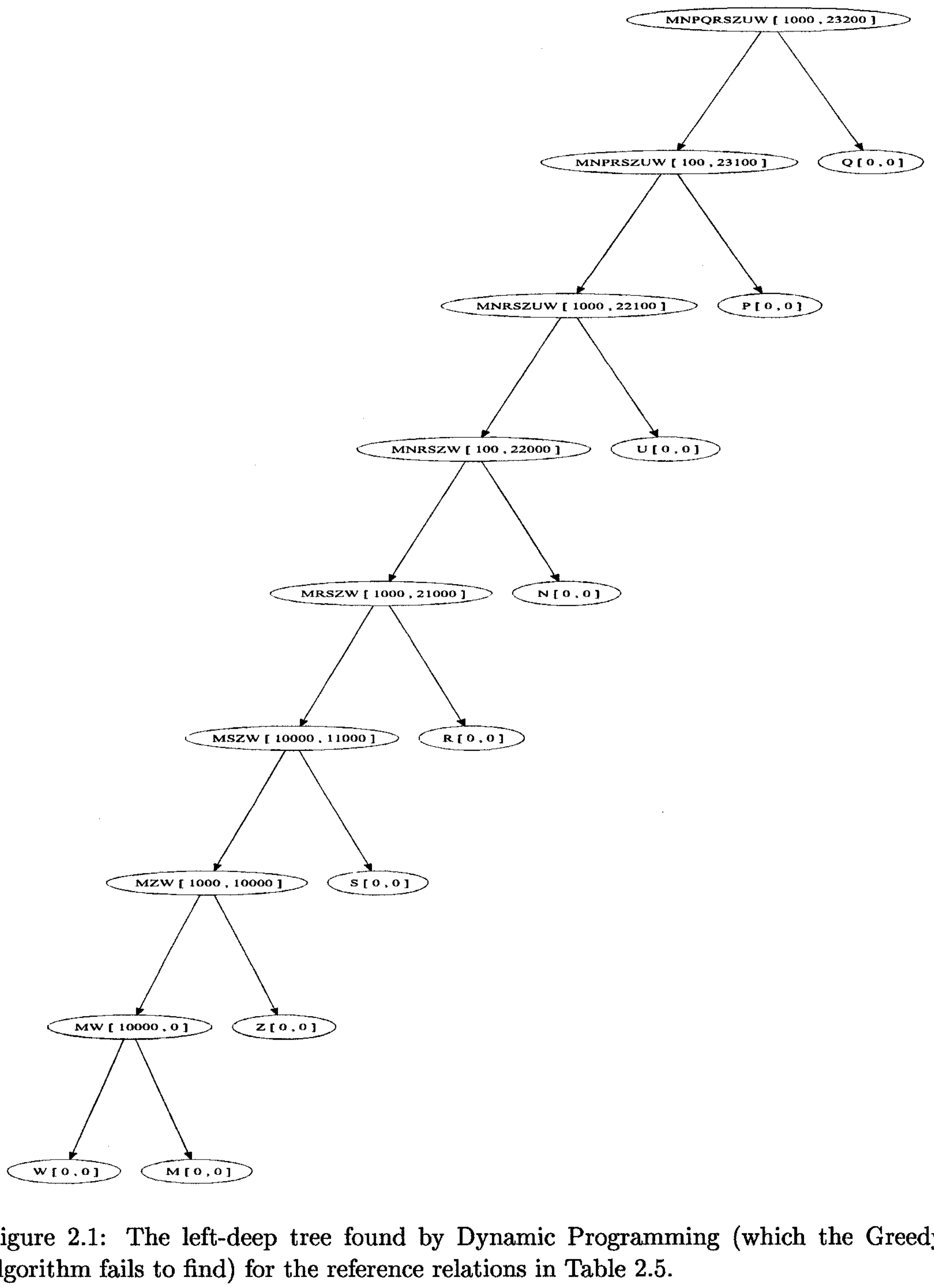


Figure 2.1: The left-deep tree found by Dynamic Programming (which the Greedy algorithm fails to find) for the reference relations in Table 2.5.

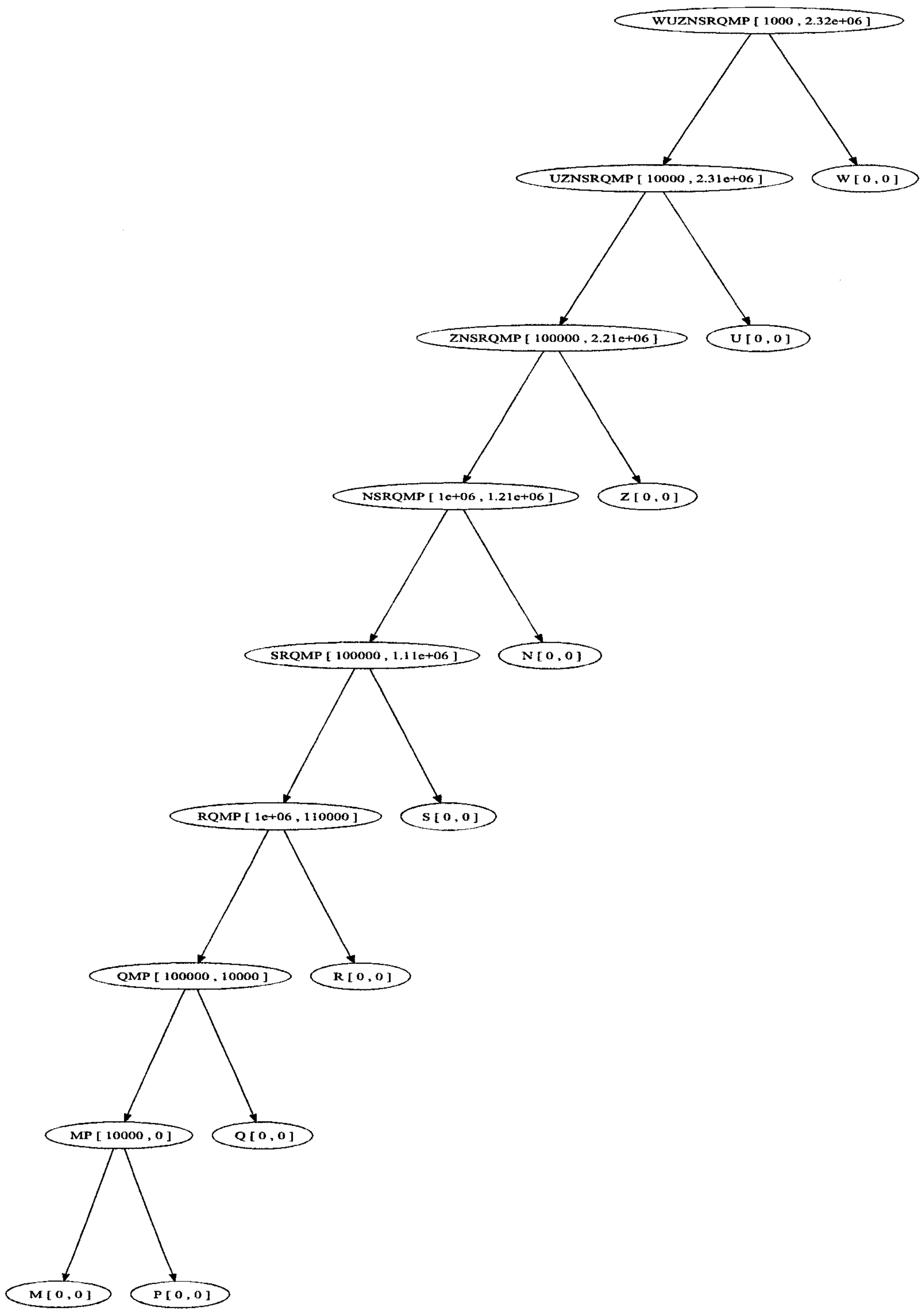


Figure 2.2: The non-optimal solution found by Greedy algorithm for the reference relations in Table 2.5.

Chapter 3

AO^* Algorithm

The term *production system* in Aritifical Inteligence usually refers to systems derived from a computational formalism which was based on string replacement rules. The notion of *decomposable production system* encompasses a technique often called *problem reduction* in AI. The problem reduction idea usually involves replacing a problem *goal* by a set of subgoals such that if the subgoals are solved, the main goal is also solved.

To solve a given problem by decomposition we split the problem into a set of subproblems. Each subproblem is expected to be simpler to solve than the given problem. We then solve the given problem by solving subproblems. A subproblem can, of course, be further decomposed. We proceed this way until we obtain a set of *terminal* problems, that is, problems whose solution we know.

In this chapter, we discuss a well-known problem reduction technique called AO^* [18] and then describe an adaptation of AO^* to solve the problem of join reordering.

3.1 AND/OR Graphs

A *decomposition production system* can be represented by an AND/OR graph as defined in [18]. An AO^* graph is a *hypergraph* in which *hyperarcs* connect a node to its successor nodes. These hyperarcs are called *connectors*. Each *k-connector* is directed from a *parent* node to a set of k *successor* nodes. In Figure 3.1, we show an example of an AND/OR graph. Note that node n_0 has a 1-connector directed to successor n_1

and a 2-connector directed to the set of successors n_2, n_3. For $k > 1$, k-connectors are denoted by a curved line joining the arcs from parent to elements of the successor set.

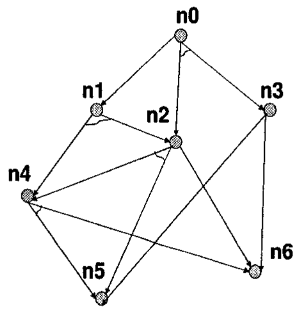
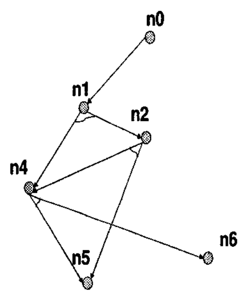


Figure 3.1: AN AND/OR graph.

In the decomposable production system the initial database corresponds to a distinguished node in the graph called the *start node*. The start node has an outgoing connector to a set of successor nodes corresponding to the components of the initial database(if it can be decomposed). Each production rule corresponds to a connector in the implicit graph. The nodes to which such a connector is directed correspond to component databases resulting after rule application and decomposition into components. There is a set of *terminal* nodes in the implicit graph corresponding to databases satisfying the termination condition of the production system. The task of the production system can be regarded as finding a *solution graph* from the start node to the terminal nodes.

Roughly, a solution graph from node n to node set N of an AND/OR graph is analogous to a path in an ordinary graph. It can be obtained by starting with node n and selecting exactly one outgoing connector. From each successor node to which this connector is directed, we continue to select one outgoing connector, and so on, until eventually every successor thus produced is an element of the set N. In Figure 3.2, we show two different solution graphs from node n_0 to n_5,n_6 in the graph of Figure 3.1.



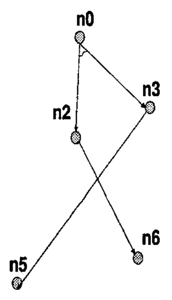


Figure 3.2: Two solution graphs for the graph in Fig 3.1.

We can give a precise recursive definition of a solution graph. The definition assumes that our AND/OR graphs contain no cycles, that is, it assumes that there is no node in the graph having a successor that is also its ancestor. The nodes thus form a partial order which guarantees termination of the recursive procedures we use. We henceforth make this assumption of acyclicity.

Let G' denote a solution graph from node n to a set N of nodes of an AND/ OR graph G. G' is a subgraph of G. Analogous to the use of arc costs in ordinary graphs, it is often useful to assign costs to connectors in AND/OR graphs.(These costs model the costs of rule applications ; again we need to assume that each cost is greater than some small positive number, e.) The connector costs can then be used to calculate the cost of a solution graph. Let the cost of a solution graph from any node n to N be denoted by $k(n, N)$. The cost $k(n, N)$ can be recursively calculated as follows:

If n is an element of N, then $k(n, N) = 0$. Otherwise, n has an outgoing connector to a set of successor nodes $\{n_1, ...n_i\}$ in the solution graph. Let the cost of this connector be c_n. Then, $k(n, N) = c_n + k(n_1, N) + ... + k(n_i, N)$.

We see that the cost of a solution graph, G', from n to N is the cost of the outgoing connector from n (in G') plus the sum of the costs of the solution graphs from the successors of n (in G') to N. This recursive definition is satisfactory because we are assuming acyclic graphs.

Beyond merely finding *any* solution graph from the start node to a set of terminal nodes, we may well want to find one having minimal cost. We call such a solution graph an *optimal* solution graph, and denote the cost of an optimal solution graph from n to a set of terminal nodes by the function $k^*(n)$.

3.2 *AO**: A heuristic search procedure for AND/OR graphs

Since there can be more than one solution graph from a node n, there can be more than one value for the solution graph from n. Let $k^*(n)$ be the cost of the cheapest solution graph from node n, where $k^*(n) \geq 0$; and $k(n)$ be an estimate of $k^*(n)$ based on some heuristic, such that $k(n) \geq 0$.

If $k(n)$ is infinite, then n has been judged to be unsolvable. Suppose that for all nodes n, and for all connectors directed from n, $k(n)$ is at most the cost of the connector from n to the nodes n_1, n_2,...,n_m + $\sum_{i=1}^{m} k(n_i)$. Then the heuristic function k is said to be *monotonic*. It can be shown that if k is monotonic, and if the k value of all terminal nodes is zero, then k is a lower bound on k^*. In other words, $k(n) \leq k^*(n)$, for all nodes n.

The heuristic function k is employed by a search strategy called AO^*. It is an established mathematical result that if k is monotonic, and if k is a lower bound on k^*, then AO^* is admissible. In other words, AO^* is guaranteed to find the cheapest solution graph, provided a solution graph exists.

For completeness, we give here the AO^* algorithm as described in [18]:

```
1 Create a search graph, $G$, consisting solely of the start node, $s$. Associate with
  node $s$ a cost $q(s) = h(s)$. If $s$ is a terminal node, label $s$ *SOLVED*.
2 **until** $s$ is labeled *SOLVED*, **do:**
    3 **begin**
    4 Compute a *partial* solution graph, $G'$, in $G$ by tracing down the *marked*
      connectors in $G$ from $s$.
    5 **select** any nonterminal leaf node, $n$, of $G'$.
    6 Expand node $n$ generating all of its successors and install these in $G$ as
      successors of $n$. For each successor, $n_j$, not already occuring in $G$, associate
      the cost $q(n_j) = h(n_j)$. Label *SOLVED* any of these successors that are
      terminal nodes.
    7 Create a singleton set of nodes, $S$, containing just node $n$.
    8 **until** $S$ is empty, **do:**
        9 **begin**
        10 Remove from $S$ a node $m$ such that $m$ has no descendants in $G$
           occuring in $S$.
        11 Revise the cost $q(m)$ for $m$, as follows:
           for each connector directed from $m$ to a set of nodes $\{n_{1i},...,n_{ki}\}$
           compute $q_i(m) = c_i + q(n_{1i} + ... + q_{ki}$.
           Set $q(m)$ to the minimum over all outgoing connectors of $q_i(m)$ and
           mark the connector through which this minimum is achieved, erasing
           the previous marking if different. If all of the successor nodes through
           this connector are labeled *SOLVED*, then label node $m$ *SOLVED*.
        12 If $m$ has been marked *SOLVED* or if the revised cost of $m$ is different
           than its just previous cost, then add to $S$ all those parents of $m$ such
           that $m$ is one of their successors through a marked connector.
    13 **end**
14 **end**
```

Figure 3.3: The AO^* algorithm [18].

Algorithm AO^* can best be understood as a repetition of the following two major operations. First a top-down operation (steps 4 to 6), to find the best partial solution graph by tracing down through the marked connectors. These marks indicate the current best partial solution graph from each node in the search graph. One of the nonterminal leaf nodes of this best partial solution graph is expanded, and a cost is assigned to its successors.

The second major operation in AO^* is a bottom-up, cost-revising, connector-marking, $SOLVED$-labeling procedure. Starting with the node just expanded, the procedure revises its cost and marks the outgoing connector on the estimated best "path" to terminal nodes. This revised cost estimate is propagated upward in the graph. The revised cost, $k(n)$, is an updated estimate of the cost of an optimal solution graph from n to a set of terminal nodes. Only the ancestors of nodes having their costs revised can possibly have their costs revised, so only these need be considered. Because we are assuming the monotone restriction on k, cost revisions can only be cost increases. Therefore, not all ancestors need have cost revisions, but only those ancestors having best partial solution graphs containing descendants with revised costs.

3.3 Application of AO^* to the join reordering problem

In this section, we describe our adaptation of the AO^* algorithm described in the previous section to solve the problem of join reordering. We start by describing the specific types of AND/OR graphs relevant to our application.

It is straightforward to see that all possible join reordering plans can be represented by an AND/OR graph. For example, Figure 3.4 shows an AND/OR graph representing all possible join orders for performing the join $R \bowtie S \bowtie T$. The root node, labelled RST has three outgoing binary AND connectors. (In fact, all AND connectorsin our application are always 2-connectors, since they always represent the join of two intermediate relations.) These three outgoing connectors represent the three possible ways of obtaining the join $R \bowtie S \bowtie T$ as a binary join of two (possibly intermediate) relations. That is, the AND connector with successor nodes RS and T represents the join order $(R \bowtie S) \bowtie T$, the AND connector with successor nodes RT and S represents the join order $(R \bowtie T) \bowtie S$ and the AND connector with successor nodes TS and R represents the join order $(T \bowtie S) \bowtie R$. For the node RS there is only one possible join order, which is seen by the fact that there is a single outgoing binary AND connector. Terminal nodes (nodes with no outgoing connector) correspond to base relations. A solution graph is an AND/OR tree, with every node having a single outgoing AND connector, and correspond to a specific join-ordering.

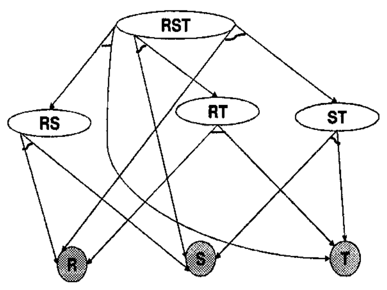


Figure 3.4: An AND/OR graph representing different possible join reorderings of three relations R, S and T.

Next, we present our adaptation of AO^* to deal with the types of AND/OR graphs described above. In the following, a node v is labelled by a set S of relations. It follows from the discussion in the previous paragraph that the two successor nodes corresponding to each 2-connector outgoing from v are labelled by sets S_1 and S_2, which constitute a partition of S. Each node is further labelled by two parameters: *cost* and *size*. The size parameter corresponding to set S of relations is an estimate of the size of the join of all relations in the set S as computed by the formula in Section 1.2.1. The size parameter for each node remains fixed during the operation of the algorithm. The cost parameter is an estimate of query cost. In general, this cost is an underestimate of the true cost. For every node, the cost parameter is initialized to 0, and is revised upwards during the operation of algorithm. When the algorithm terminates, the root node (and all other nodes in the solution graph) have the cost parameter set to the true query cost according to the optimal join order.

The algorithm proceeds as follows. We start with the root node, and expand it to the next level, and choose the best alternative based on current estimates of the costs of various alternatives (i.e. outgoing connectors). This connector is *MARKED*, and we proceed to expand one of the two successor nodes of this connector. If as the result of this expansion, the cost of the current alternative is revised, we propagate this revision upwards to the parent, which evaluates all alternatives again. If it turns out that there is now a better alternative at the parent's level, the current alternative is unmarked, and the new alternative is marked and explored. If the cost is not revised, or if the current alternative is still best, we continue expanding marked non-terminal descendants till all paths are completely explored. We say that a node v is marked when there is a path from the root node to v consisting of only marked connectors. A node is deemed *SOLVED* when it is either a terminal node, or when all marked paths outgoing from that node consist of only solved nodes. The algorithm terminates when the root is solved.

It remains to describe the computation of the cost of a node. The cost of a terminal node (that is, base relation) is 0, and the cost of a non-terminal node is the minimum of the costs of its outgoing connectors. The cost of a connector with successor nodes u and v is $cost(u) + cost(u) + (size(v) + size(u))$. This is because choosing the connector with successor nodes u and v means performing the join of intermediate relations corresponding to u and v, which includes the cost of computing the join of each of those intermediate relations (that is, $cost(u)$ and $cost(v)$) plus the cost of computing the join of these two intermediate relations (that is, $size(u) + size(v)$).

The psuedocode for the initialization function is given in Figure 3.5. All the terminal nodes are labelled solved and their costs and sizes are initialized to 0. All the non-terminal nodes are initially labelled unsolved, their costs and sizes again are initialized to 0. The pseduocode for the main **Solve** procedure is given in Figure 3.6. The algorithm starts by calling the procedure for the node root, **Solve(root)**, where the root is a node labelled by the set of all relations to be joined. Initially the graph G consists only of the root node.

```
initialize_node(S)
    if S is a singleton set,
        $solved(S) \leftarrow true$;
        $size(S) \leftarrow 0$;
        $cost(S) \leftarrow 0$;
    else
        $solved(S) \leftarrow false$;
        $size(S) \leftarrow size\_estimate(S)$;
        $cost(S) \leftarrow 0$;
end of function
```

Figure 3.5: The initialization function.

```
Solve(set S[int size, int cost, bool solved])
    while not solved(S) do:
        $min\_cost \leftarrow \infty$
        for each binary partition $S_1$ and $S_2$ of $S$ do:
            if $S_1$ is not a node in $G$, then:
                initialize_node($S_1$)
            if $S_2$ is not a node in $G$, then:
                initialize_node($S_2$)
            temp $\leftarrow$ $cost(S_1)$ + cost($S_2$) + size($S_1$) + size($S_2$)
            if $temp < min\_cost$, then:
                $min\_cost \leftarrow temp$
                Mark connector($S_1$,$S_2$);
                Mark node $S_1$;
                Mark node $S_2$;
                Unmark any previous marked connector ($S'_1$,$S'_2$) outgoing from
                $S$;
                Unmark node $S'_1$;
                Unmark node $S'_2$;
        if $cost < min\_cost$
            $cost \leftarrow min\_cost$
            if $S$ is not root, then
                exit
        if all marked children of $S$ are solved, then:
            $solved$(S) $\leftarrow$ $true$;
        else for some unsolved marked child $u$ of $S$
            Solve($u$);
end of procedure
```

Figure 3.6: AO^* Pseduocode.

Since our cost function is initialized to 0, and it can easily be seen to satisfy the monotonicity restriction, it follows from the result in [18] that our AO^* algorithm is admissible. It is obvious that in any algorithm, the graph representing the final solution either optimal or non-optimal has a tree shape. The trees representing the final solutions of different algorithms which are shown in this thesis have the following characteristics:

- On each node, there are two values given, the size of the join up to that point and the cost of the query up the point.
- In AO^* graphs, since nodes have binary connectors, to show which connector are combined with which ones, each connector has an identifier number. The connectors which are together have identical numbers. Additionally, the connectors which are MARKED, have an identifier "(M)" in front of their identifier number.

3.4 How AO^* algorithm works on the reference query

In this section, we illustrate the operation of AO^* on the set of relations which was used as our example for the dynamic programming and greedy algorithms in Section 2.1. For ease of reference, we provide the details of the relations here in Table 3.1.

S	R	U	W
$T(S) = 1000$	$T(R) = 100$	$T(U) = 100$	$T(W) = 1000$
$V(S,a) = 100$			$V(W,a) = 1000$
$V(S,b) = 100$	$V(R,b) = 100$		
	$V(R,c) = 10$	V(U,c)=10	
		$V(U,d) = 100$	$V(W,d) = 100$

Table 3.1: Reference relations used to illustrate various algorithms.

In AO^*, we call the algorithm with root node $SRUW$, with cost 0 and size 100. On expanding the node, the minimum cost outgoing connector is found to be $S \bowtie RUW$, which has a cost of $0+0+1000+0 = 1000$. So this connector is marked (drawn bold in the graph), and the resulting AND/OR graph is shown in Figure 3.7. Recall that in the graphs representing join trees, in each node depicting a join two values are given, the first is the estimated size of intermediate join represented by the node and the second one is the estimated cost of that specific sub-query. Now, the *Solve* procedure is called recursively for the node TUW.

In Figure 3.8 the node RUW is expanded to its immediate successors. Among its outgoing connectors, nodes RW and U have the least cost, which is $0+0+1000+0 = 1000$. Consequently the connectors to these nodes are $MARKED$. In this cycle the cost of the intermediate relation RUW is revised to 1000. Since this cost has been revised, we exit the recursive call and return to the while loop at the root node.

In Figure 3.9, the cost of the connector node (RUW,S) will be revised as $0 + 0 + 1000 + 1000 = 2000$ which in turn causes the algorithm to choose a new connector SRW, U as the MARKED connector which has cost 1000. Next, the *Solve* procedure is called recursively for the node SRW, the result of which is shown in Figure 3.10. The best outgoing connector from SRW is (RW,S), which has cost 1000, and causes the cost of the node SRW to change. This causes a return to the while loop at the root node once again.

In the next round, the outgoing connector (RW,SU) from the root node is chosen as the best alternative, since it has the minimum cost of $1000+0+1000+0 = 2000$; as shown in Figure 3.11.

At this point, the *Solve* procedure is called recursively for the node SU. Since there is only one outgoing connector both going to solved (terminal) nodes, the node SU is deemed solved as well. The same holds true for the node RW, and neither procedure call results in a change in cost, which further implies no change in cost for the root. Since both marked children are solved, the root itself is deemed solved and the algorithm terminates. The final solution, as given by a marked path from the root to the leaf nodes is:

$$((R \bowtie W) \bowtie (S \bowtie U))$$

as shown in Figure 3.12.

3.5 Illustration of a best-case for AO^*

As an example of a best-case scenario for our AO^* algorithm, we use a query with 9 relations, given in Table 4.1 (see the next chapter). For the AO^* algorithm to be efficent, the optimization cost should be far less than Dynamic Programming and close to the Greedy algorithm. The number of operations required by Dynamic programming to find the optimal join order for the relation given in Table 4.1 is 9330, whereas AO^* finds the optimal solution using only 284 operations. This number for the Greedy algorithm is 81. The ratio of AO^* to Dynamic Programming is about .03 while this value for Greedy algorithm to AO^* is about .33. The graph showing all nodes created by AO^* is given in Figure 3.12. The shaded nodes are in the solution. As can be seen, very few nodes not part of the solution are created.

There are several factors involved in the cost of the AO^* algorithm which will be discussed in the next chapter. Among them are the shape of the query, the size distribution of relations, and the shape of the final solution.

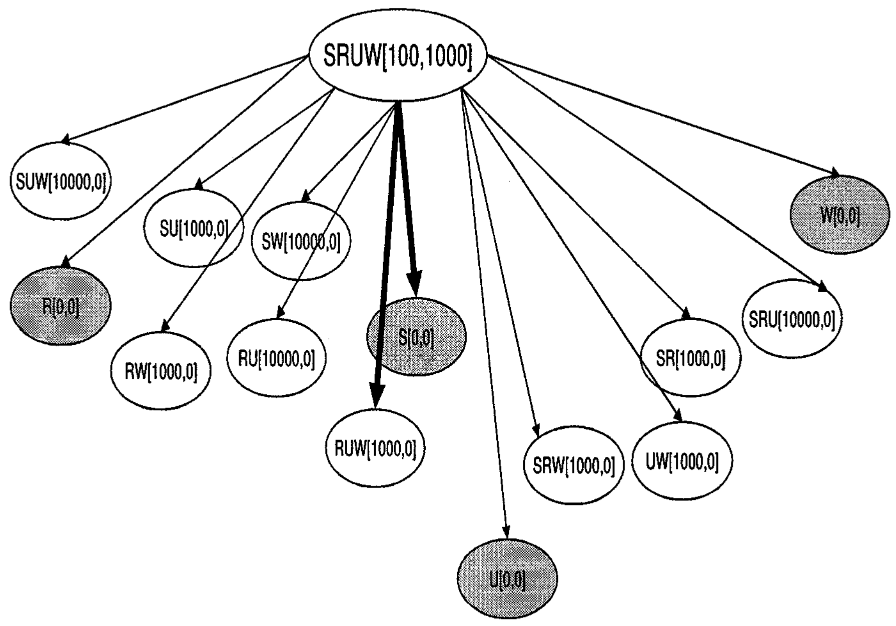


Figure 3.7: The first cycle of running AO^*algorithm on the test query.

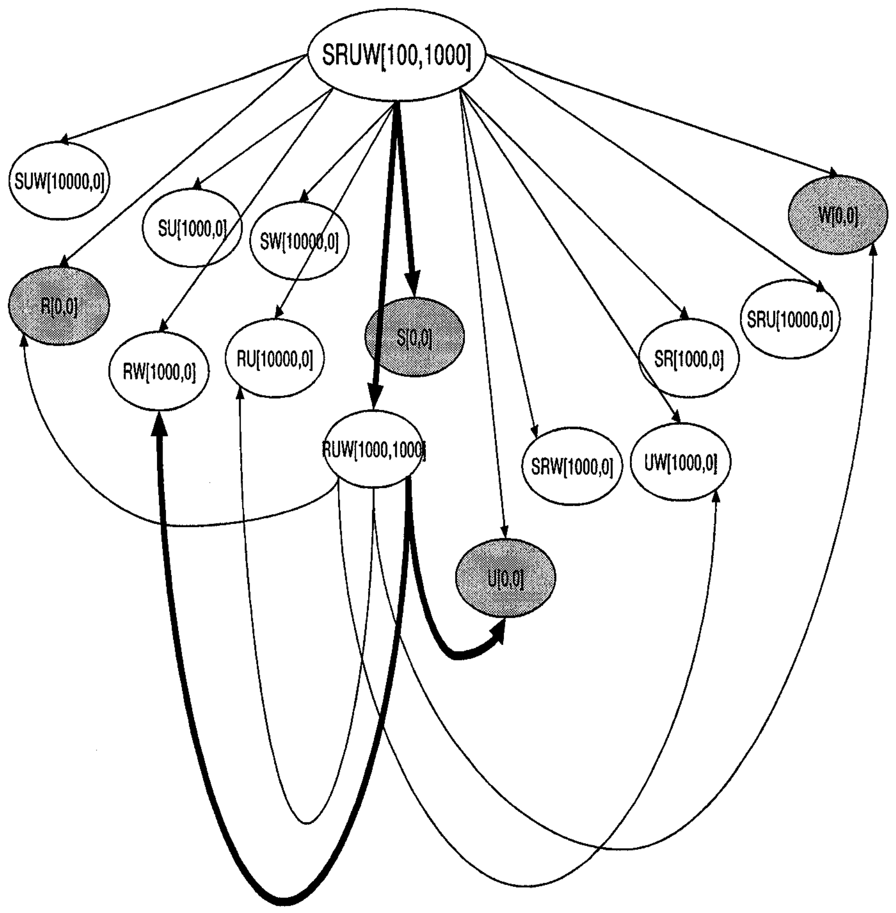


Figure 3.8: The second cycle of running AO^*algorithm on the test query.

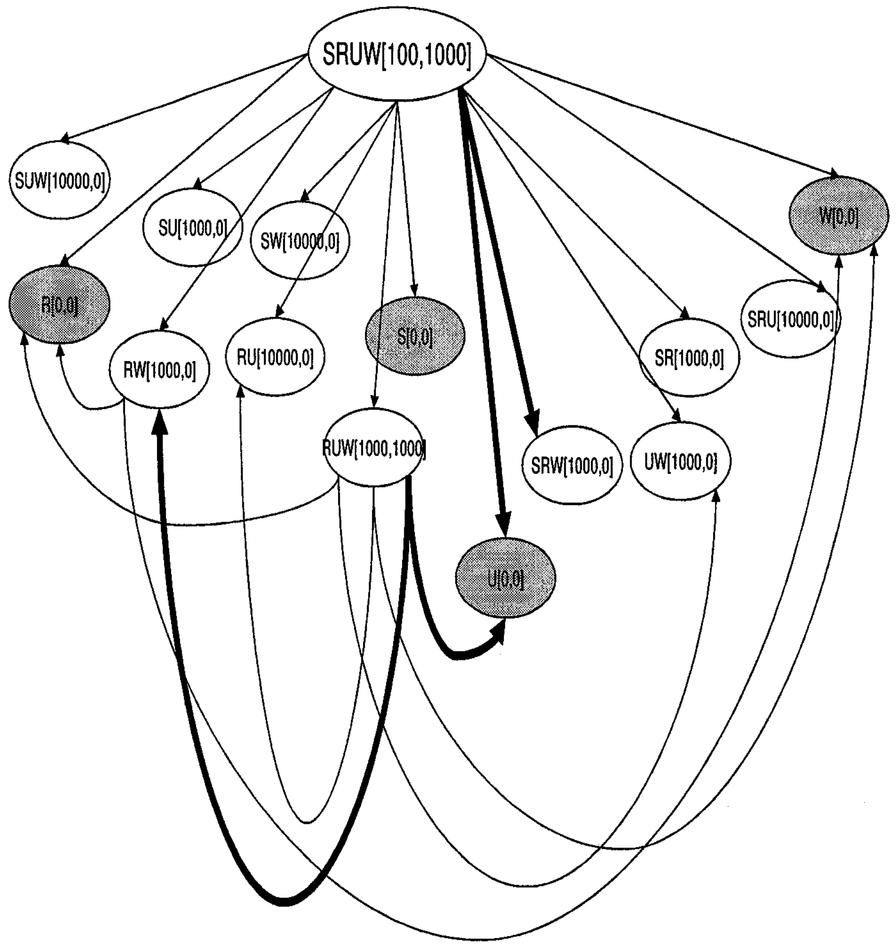


Figure 3.9: The third cycle of running AO^*algorithm on the test query.

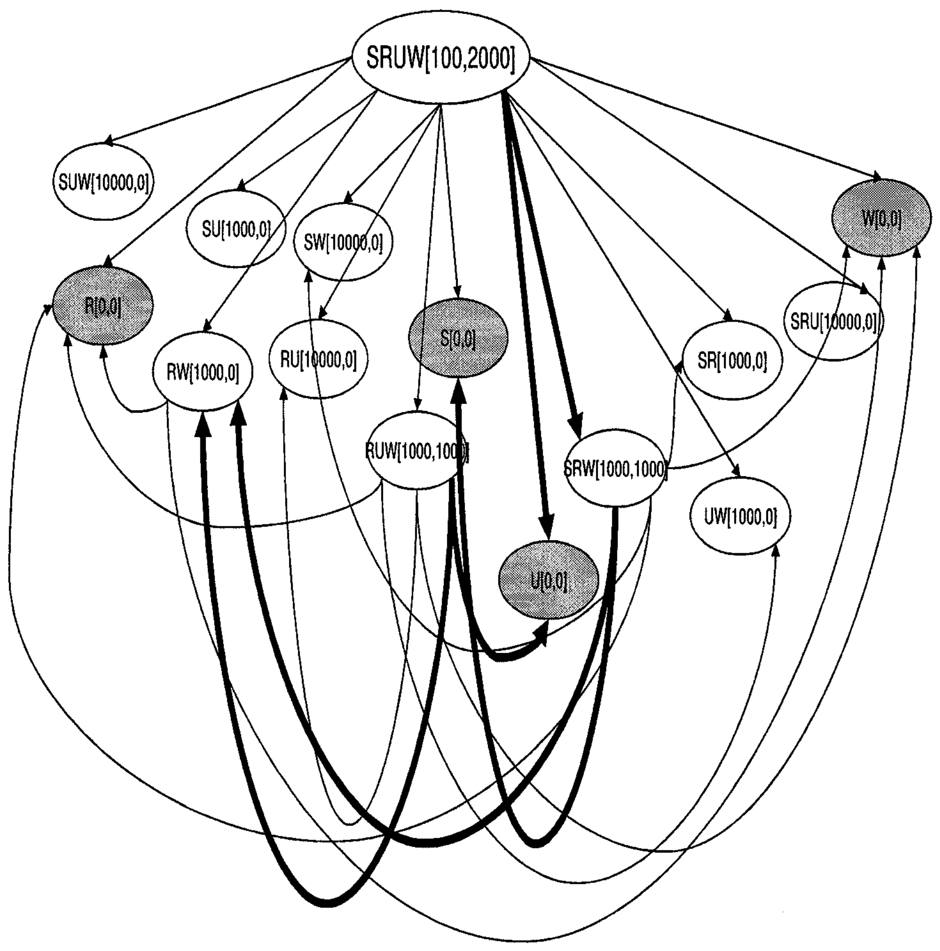


Figure 3.10: The fourth cycle of running AO^*algorithm on the test query.

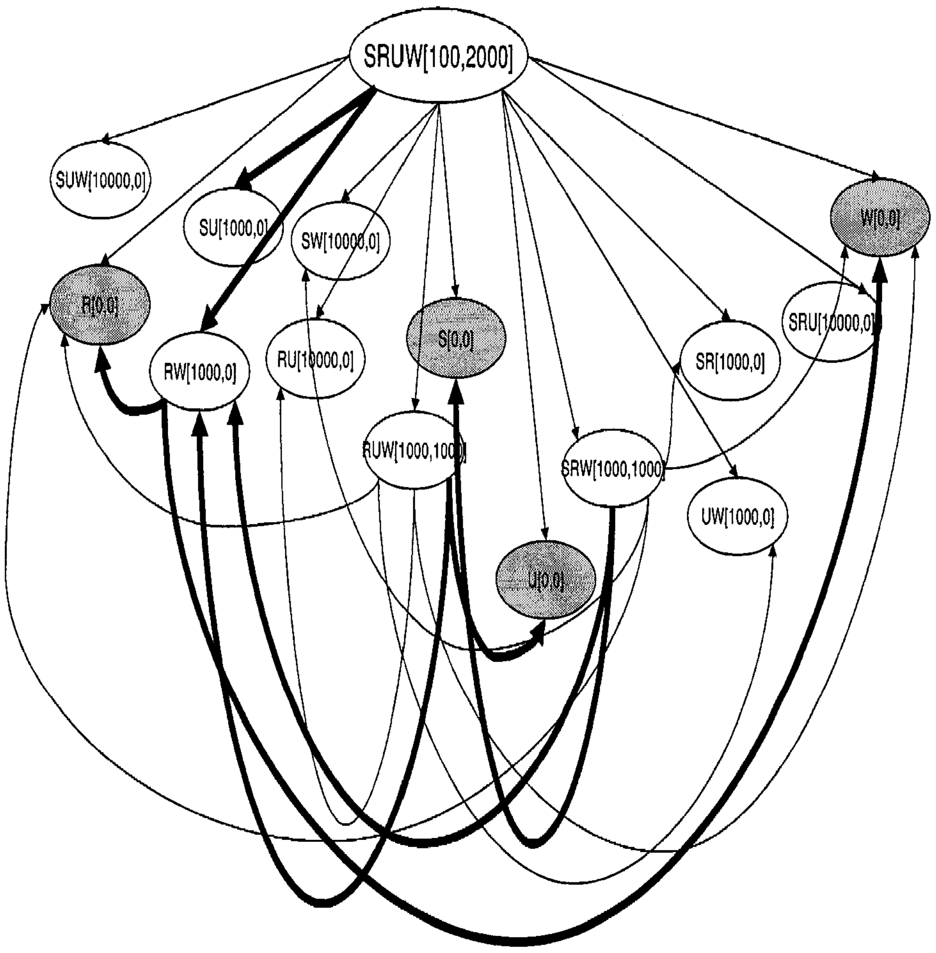


Figure 3.11: The fifth cycle of running AO^*algorithm on the test query.

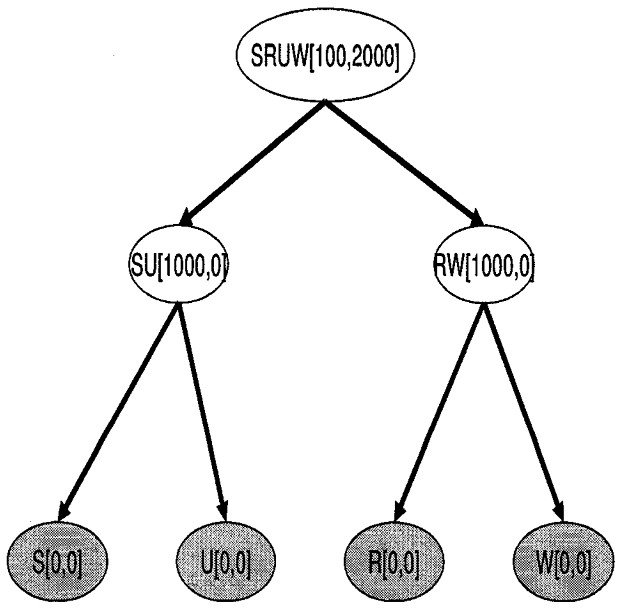


Figure 3.12: The result after running AO^*algorithm on the test query.

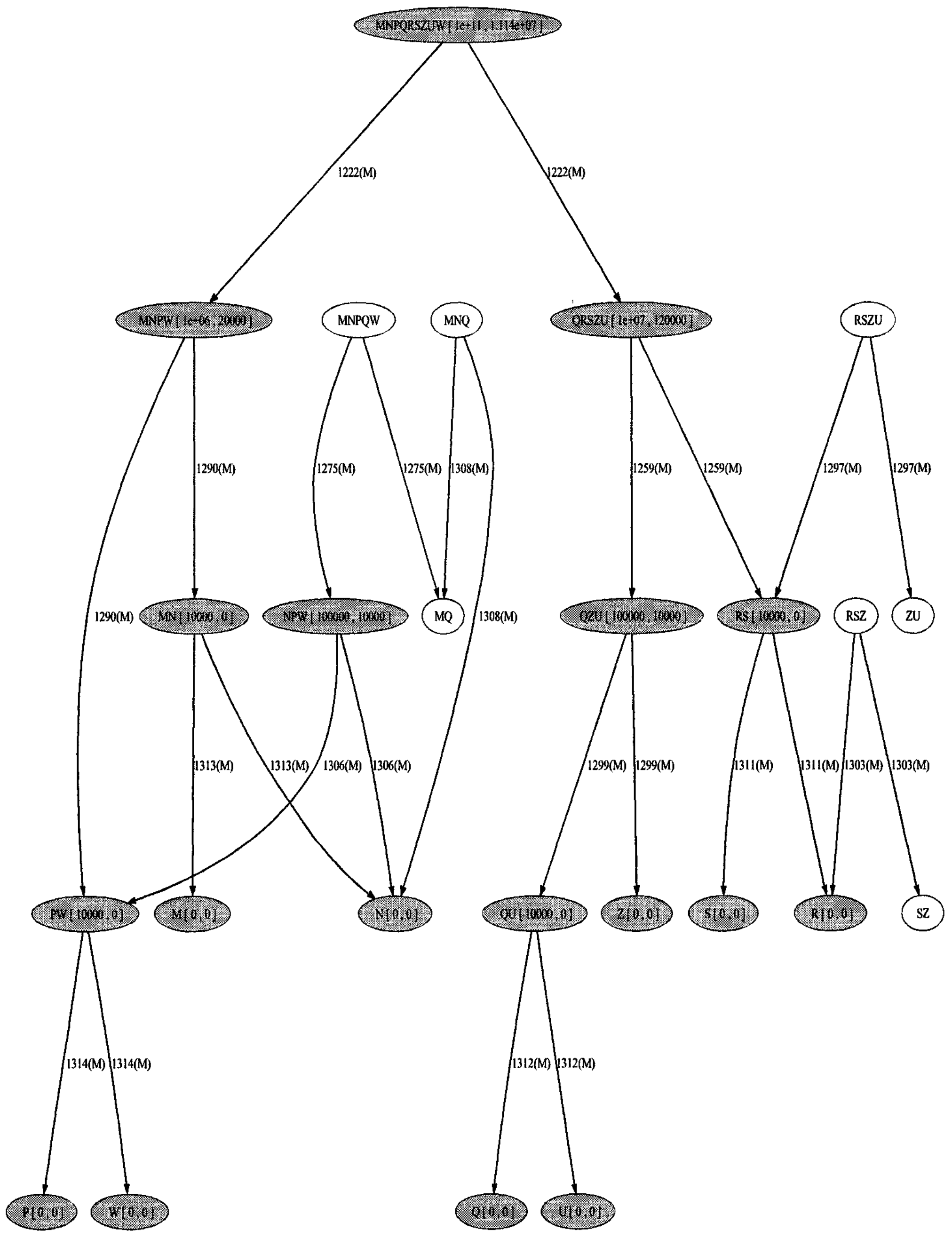


Figure 3.13: The best case example of AO^* algorithm, the AO^* optimization cost is 284 vs the Dynamic Programming optimization cost of 9330.

3.6 Illustration of a bad case for AO^*

To illustrate an example where AO^* has high optimization cost, a query with the same number of relations is used. The reference relations are shown in Table 4.4 (see the next chapter). For AO^* algorithm, a bad scenario is when its optimization cost is close to that of Dynamic Programming. In this example the number of operations to find the optimal answer was 8756 while this number for Dynamic Programming is 9330. The ratio is about .94. The graph including all created nodes for this example is given in Figure 3.14. As can be seen, there are several nodes created which are not part of the eventual optimal solution.

In the next chapter, we will visit some of the factors affecting the performance of the AO^* algorithm. The main factor differentiating the example given in this section, which is a bad case for AO^* and the one given in section 3.5, which is a good case for AO^*, is the shape of the query.

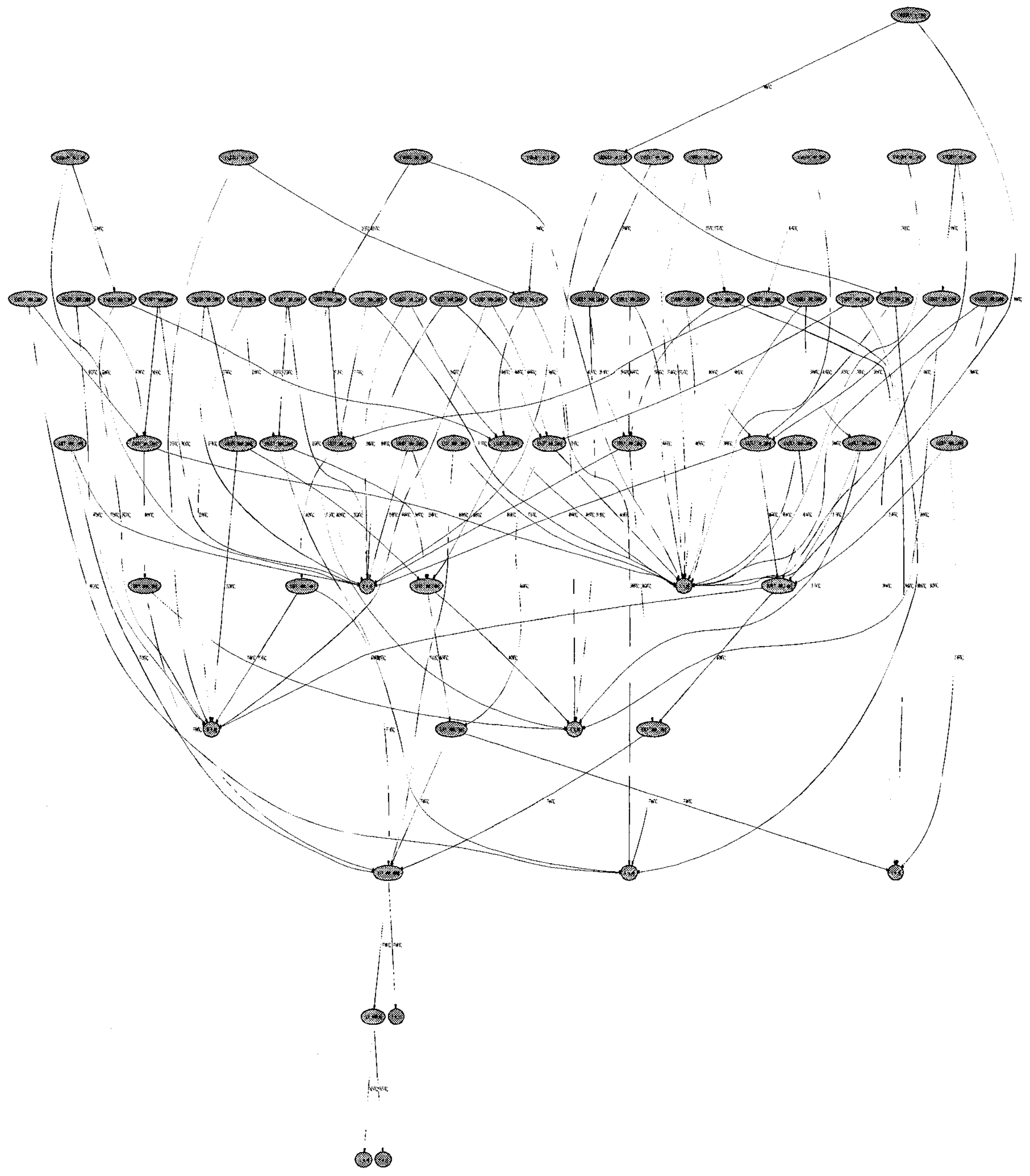


Figure 3.14: The worst case example of AO^* algorithm, AO^* optimization cost 8756 vs Dynamic Programming optimization cost of 9330.

Chapter 4

Experiments

In this chapter, we describe the experiments we undertook to understand the performance and behavior of AO^*, particularly in relation to the dynamic programming and Greedy algorithms. We also present the result of running the AO^* algorithm on four different types of queries. Additionally, we present the experimental results showing their performance on some benchmark queries.

4.1 Shape of Query

As mentioned previously, the cost of optimization of a query by dynamic programming and the Greedy algorithm does not depend on the shape of the query and is only dependent on the number of relations in the query. This does not appear to be true for AO^*. In this thesis, we studied the performance of AO^* on four different types of queries: star, chain, circular and clique queries. We examined the best and worst-case scenarios for all three algorithms, the effect of the shape of the query and the shape of the result on their performance.

4.1.1 Star Query

In a *Star* query, there is a subset of attributes common to all relations in the query. In Figure 4.1 this type of query is illustrated. The graph corresponding to the query is a star.

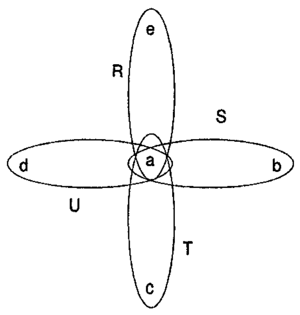


Figure 4.1: Star-shaped query. The relations R, S, T, U have attribute a in common.

4.1.2 Chain Query

In a *Chain* query, all except two queries have common attributes with exactly two other relations but the first and last relations have attributes in common with only one other query. Each attribute is common to at most two relations. This type of query is illustrated in Figure 4.2. The graph corresponding to the query is a chain.

4.1.3 Circular Query

In a *Circular* query, each relation has common attributes with exactly two other relations and each attribute is common to at most two relations. Figure 4.3 illustrates this type of query. The graph corresponding to the query is a cycle.

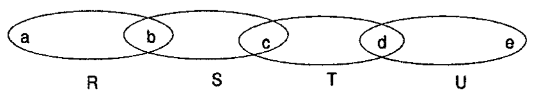


Figure 4.2: Chain-shaped query. The relations R and S have attribute b, S and T attribute c and T and U attribute d in common.

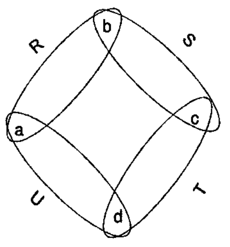


Figure 4.3: Circular-shaped query. The relations R and S have attribute b, S and T have attribute c, T and U have attribute d and U and R have attribute a in common.

4.1.4 Clique Query

In a *Clique* query, every pair of relations has a unique subset of common attributes. This type of query is illustrated in Figure 4.4. The graph corresponding to the query is a clique.

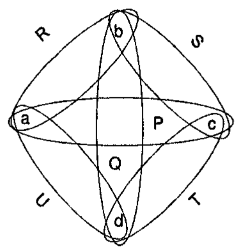


Figure 4.4: Clique-shaped query. Every relation has at least an attribute in common with other relations.

4.2 Left-deep vs Bushy trees

While examining the performance of the AO^* algorithm, it was noticed that the shape of the final optimal solution appears to have an effect on the number of joins to be evaluated to find that optimal solution, or in other words the cost of the query optimization. The factors affecting the performance are listed below [4][29]:

- The "bushyness" of the final solution has a good effect on the cost of the query optimization done by the AO^* algorithm. The bushier the final solution is, the less is the cost incurred by the AO^* algorithm to find it.

- The balance of the internal nodes in left and right subtrees of a "bushy node" has a good effect on the cost of query optimization. The more balanced the final solution is, the less the cost of query optimization.

- The height of the "bushy nodes" has also a good effect on the cost of query optimization. The higher the bushy nodes are located, the less is the cost of query optimization.

In order to study the effect of the shape of the final optimal solution on the performance of algorithms, we propose the *cost_index(g)* as a single factor that includes all the above factors. A node is called a *bushy node* if it has two children, both of which are internal nodes. If v is a bushy node in a solution graph with n nodes:

- *bushy-index* ($X(v)$) of each non-leaf node v is equal to the number of the internal nodes of its children minus one. The bushy index of a graph G, $X(G)$ is equal to the summation of the bushy indices of all its nodes. So:
  $X(G) = \sum_{v \in G} X(v)$

  This value for a solution graph G of a query with n relations can be seen to satisfy:

  * if n is even $X(G) \leq (n-1)(n-3)/4$.
  * if n is odd $X(G) \leq ((n-2)/2)^2$.

- The *balance index* ($Y(v)$) of the node v is the difference of the numbers of the internal nodes in the left and right subtrees of a bushy node v. The balance index of a graph G, $Y(G)$ is equal to summation of the balance indices of all its nodes. So:
  $Y(G) = \sum_{v \in G} Y(v)$

For a solution tree with n relations, $Y(G) \leq (n-4)$.

- The *height* $H(v))$ of a node v is the length of the longest simple path from v to a leaf node. The height of a graph, $H(G)$, is the height of its highest bushy node and the maximum value of the height of a graph with n relations, is $(n-2)$.

Finally, *cost_index* of a graph G is given by:

$$\text{cost index}(G) = (X(G) + H(G) - Y(G))$$

4.3 Illustration of algorithms on four example relations

In this section, we illustrate the operation of AO^*, Dynamic programming and Greedy algorithms on sample relations of all four types discussed in Section 4.1. The reference relations we use (for each type of query) are given in Tables 4.1 to 4.4, the join order given by each algorithm is given in Tables 4.5 and 4.6 and Table 4.7 compares the optimization cost of all 3 algorithms.

4.3.1 Star query

We use the sample relation given in Table 4.1 and present the results produced by all three algorithms. Figure 4.5 gives the result given by Dynamic programming and AO^* and Figure 4.6 gives the result of Greedy.

As shown in Figure 4.5, the optimal solution for this query is a bushy balanced tree with cost of 11140000 while Figure 4.6 presents the output of Greedy algorithm for the same query with a cost of 1000 times more expensive in a left-deep tree solution graph. On the other hand, the optimization costs of Dynamic Programming, AO^* and Greedy algorithms are presented in Table 4.5.

R	S	Z	U	Q
$T(R) = 1000$	$T(S) = 1000$	$T(Z) = 1000$	$T(U) = 1000$	$T(Q) = 1000$
$V(R,a) = 100$	$V(S,a) = 100$	$V(Z,a) = 100$	$V(U,a) = 100$	$T(Q,a) = 100$
$V(R,b) = 100$				
		V(Z,c)=100		
			$V(U,d) = 100$	
				$V(Q,e) = 100$
	$V(S,f) = 100$			

M	N	P	W
$T(M) = 1000$	$T(N) = 1000$	$T(P) = 1000$	T(W)=1000
$V(M,a) = 100$	$V(N,a) = 100$	$V(P,a) = 100$	$V(W,a)$
$V(M,g) = 100$			
	$V(N,h) = 100$		
		$V(P,l) = 100$	
			$V(W,t) = 100$

Table 4.1: Reference relations for a star query.

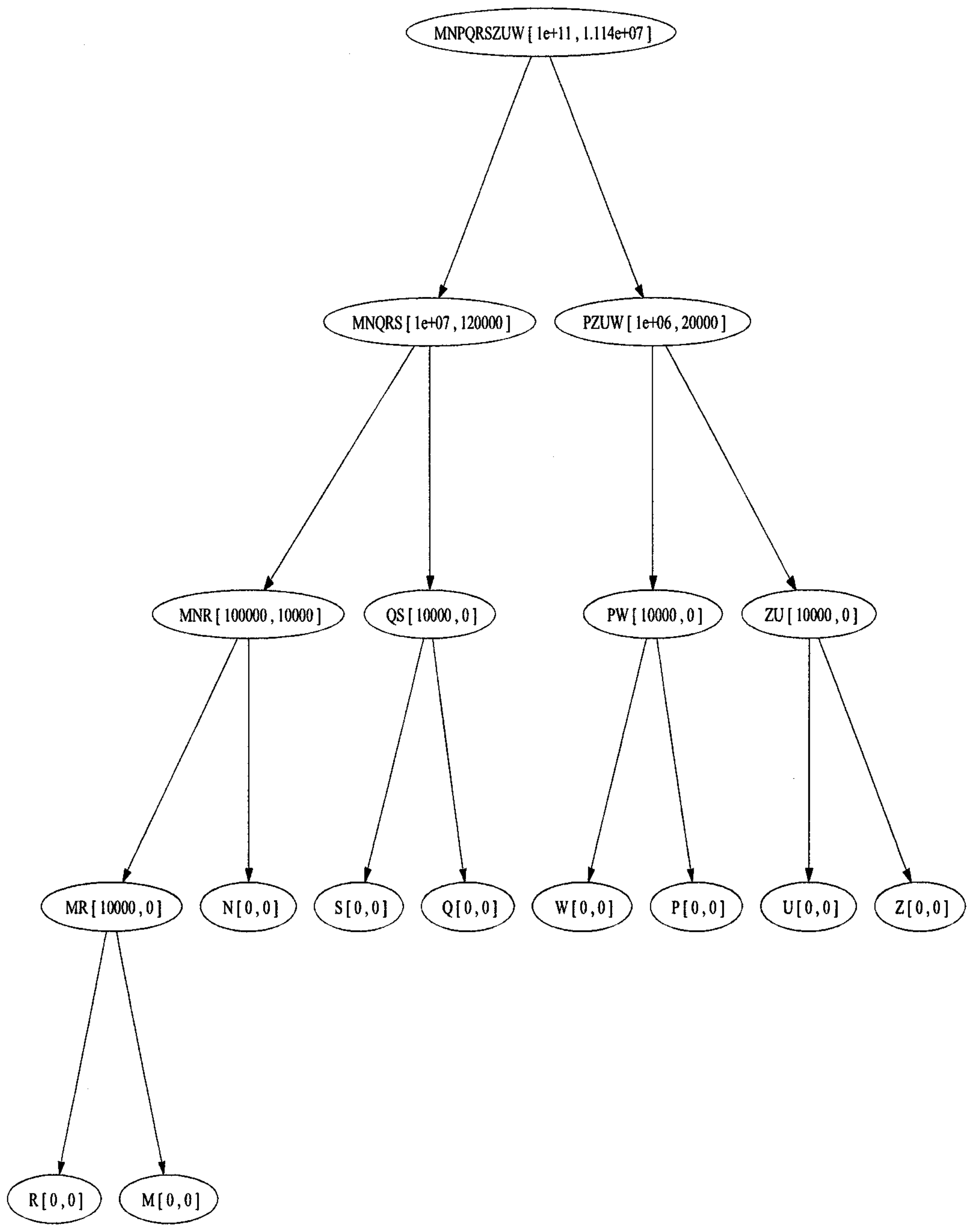


Figure 4.5: The solution given by Dynamic Programming and AO^* on the star query given in Table 4.2 with query cost of 11140000.

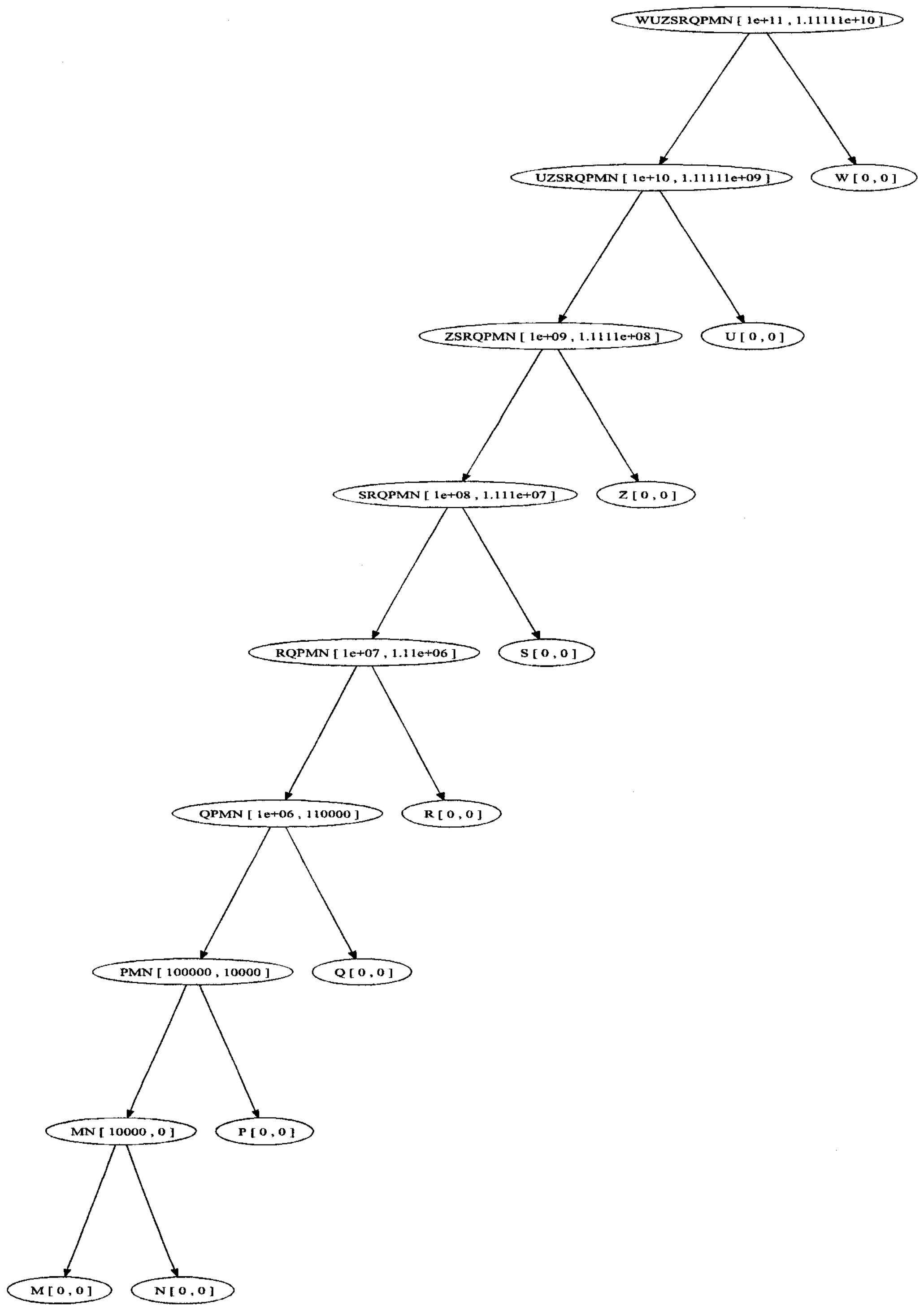


Figure 4.6: The solution given by Greedy algorithm on the star query given in Table 4.2 with query cost of 11111100000.

4.3.2 Chain query

We use the sample relation given in Table 4.2 and present the results produced by all three algorithms. Figure 4.7 gives the result given by Dynamic programming and AO^* algorithm while Figure 4.8 gives the result of Greedy.

R	S	Z	U	Q
$T(R) = 1000$	$T(S) = 1000$	$T(Z) = 1000$	$T(U) = 1000$	$T(Q) = 1000$
$V(R,a) = 100$				
$V(R,b) = 100$	$V(S,b) = 100$			
	$V(S,c) = 100$	V(Z,c)=100		
		$V(Z,d) = 100$	$V(U,d) = 100$	
			$V(U,e) = 100$	$V(Q,e) = 100$
				$V(Q,f) = 100$

M	N	P	W
$T(M) = 1000$	$T(N) = 1000$	$T(P) = 1000$	T(W)=1000
$V(M,f) = 100$			
$V(M,g) = 100$	$V(N,g) = 100$	$V(P,h) = 100$	
	$V(N,h)$	$V(P,k) = 100$	$V(W,k) = 100$
			$V(W,l) = 100$

Table 4.2: Reference relations for a chain query.

As is shown in Figure 4.7, the optimal solution graph is a bushy tree with the query cost of 11140000. It is a balanced bushy tree. Figure 4.8 presents a *left-deep* tree as the output of Greedy for the same query. The join cost for such a query is given as 11111100000, which is 1000 times more expensive.

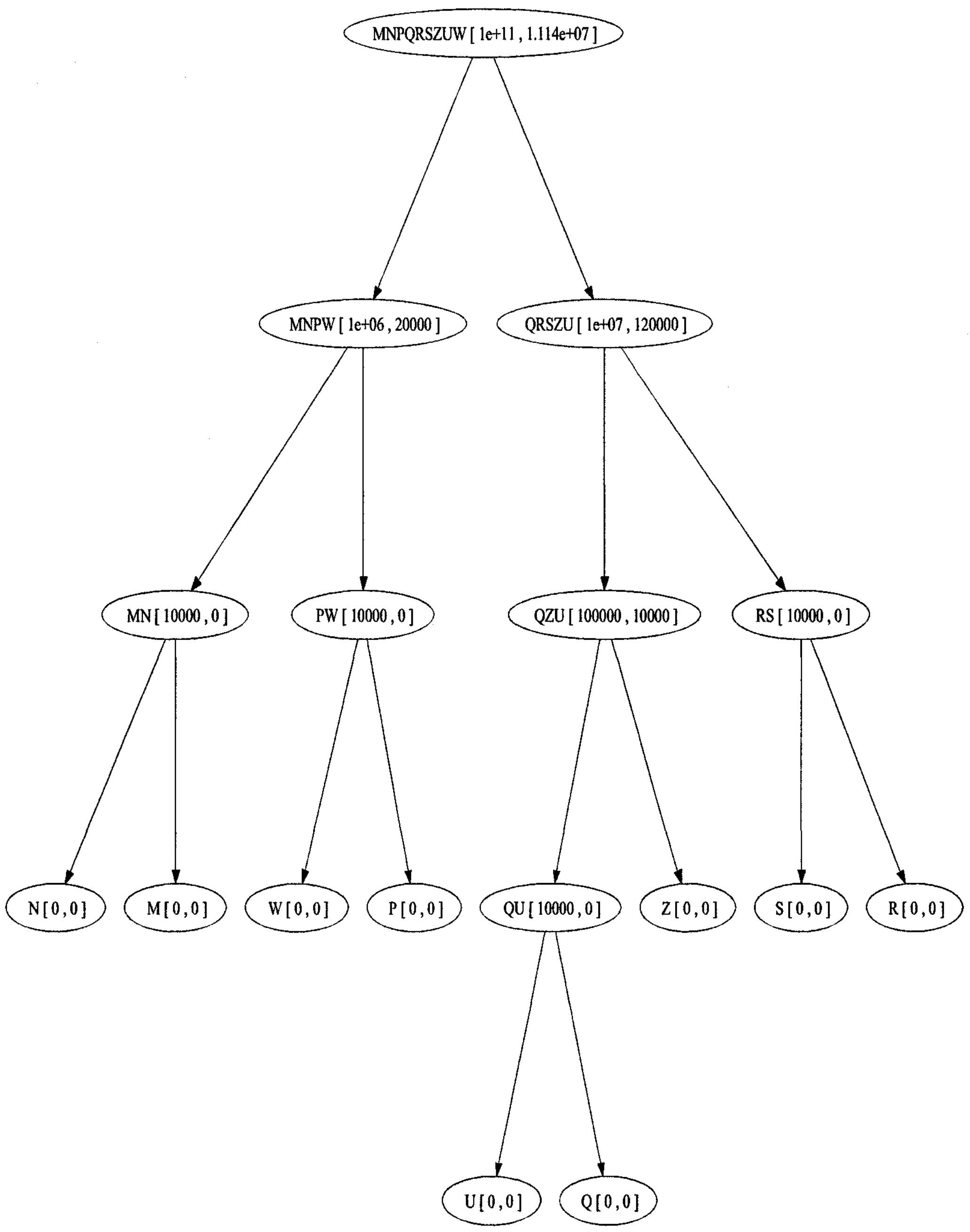


Figure 4.7: The solution given by Dynamic Programming and AO^* on the chain query given in Table 4.1, with query cost of 11140000.

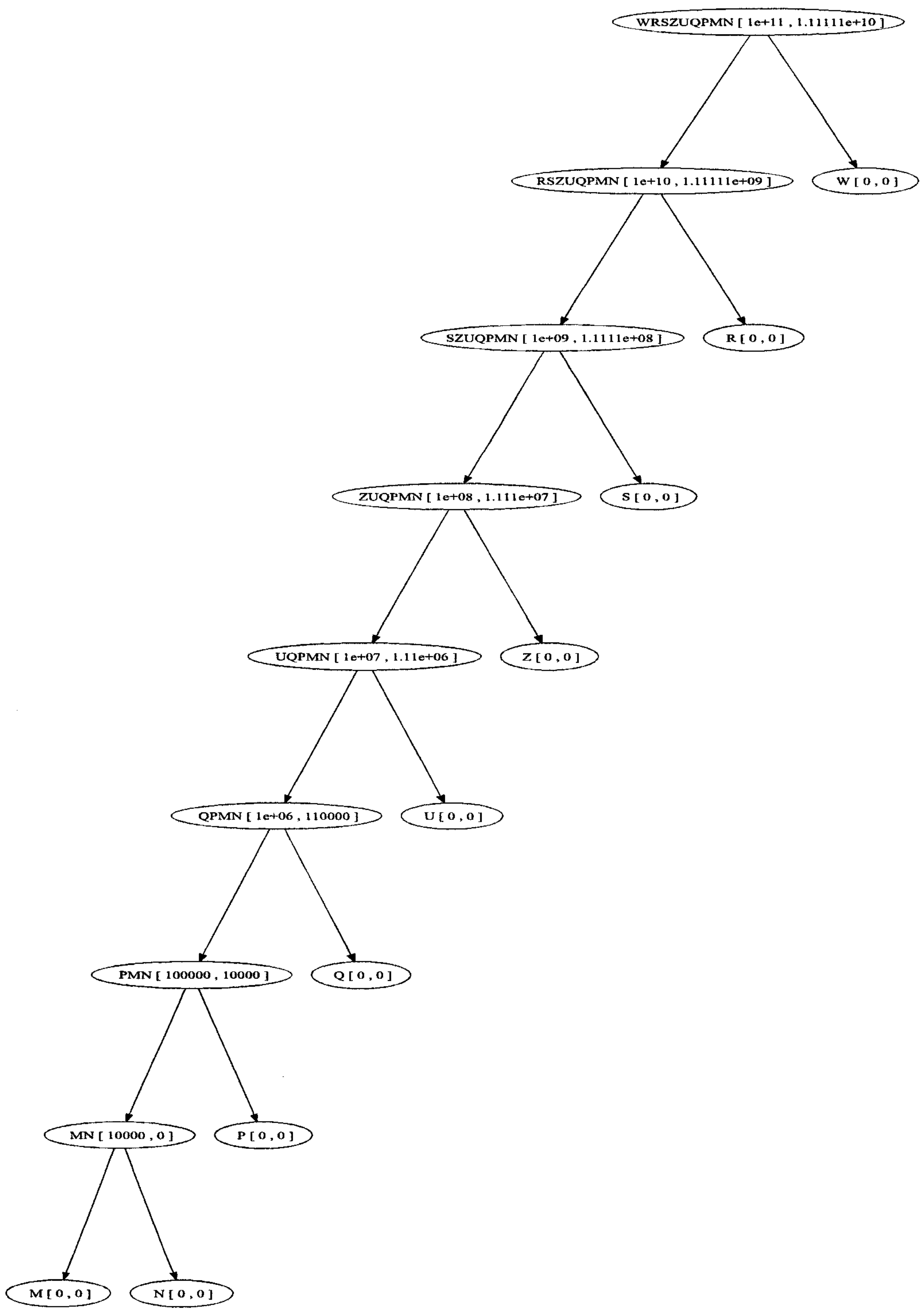


Figure 4.8: The solution given by Greedy algorithm on the chain query given in Table 4.1, with query cost of 11111100000.

4.3.3 Circular query

We use the sample relation given in Table 4.3 and present the result produced by all three algorithms. Figure 4.9 gives the result given by Dynamic programming and AO^* and Figure 4.10 gives the result of Greedy.

R	S	Z	U	Q
$T(R) = 1000$	$T(S) = 1000$	$T(Z) = 1000$	$T(U) = 1000$	$T(Q) = 1000$
$V(R,a) = 100$ $V(R,b) = 100$	$V(S,b) = 100$ $V(S,c) = 100$	V(Z,c)=100 $V(Z,d) = 100$	$V(U,d) = 100$ $V(U,e) = 100$	$V(Q,e) = 100$ $V(Q,f) = 100$

M	N	P	W
$T(M) = 1000$	$T(N) = 1000$	$T(P) = 1000$	T(W)=1000
$V(M,f) = 100$ $V(M,g) = 100$	$V(N,g) = 100$ $V(N,h) = 100$	$V(P,h) = 100$	$V(W,a) = 100$ $V(W,k) = 100$

Table 4.3: Reference relations for a circular query.

As seen in Figure 4.9, the cost for the balanced bushy tree solution graph is 11140000 while Figure 4.10 shows a left-deep tree as the output of Greedy to the same query with a query cost of 11111100000, which is 1000 times more expensive.

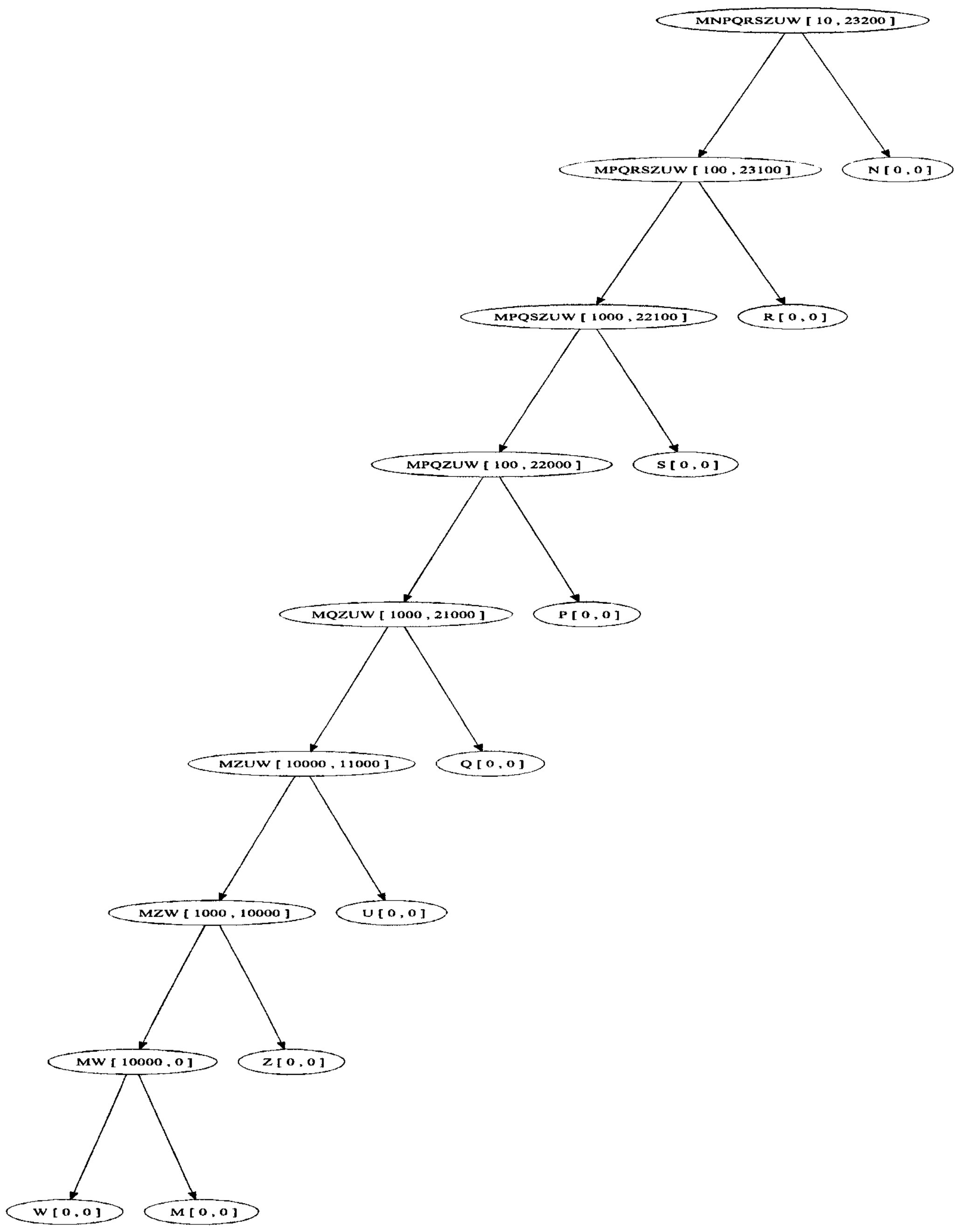


Figure 4.9: The solution given by Dynamic Programming and AO^* on the circular query given in Table 4.3, with query cost of 11140000.

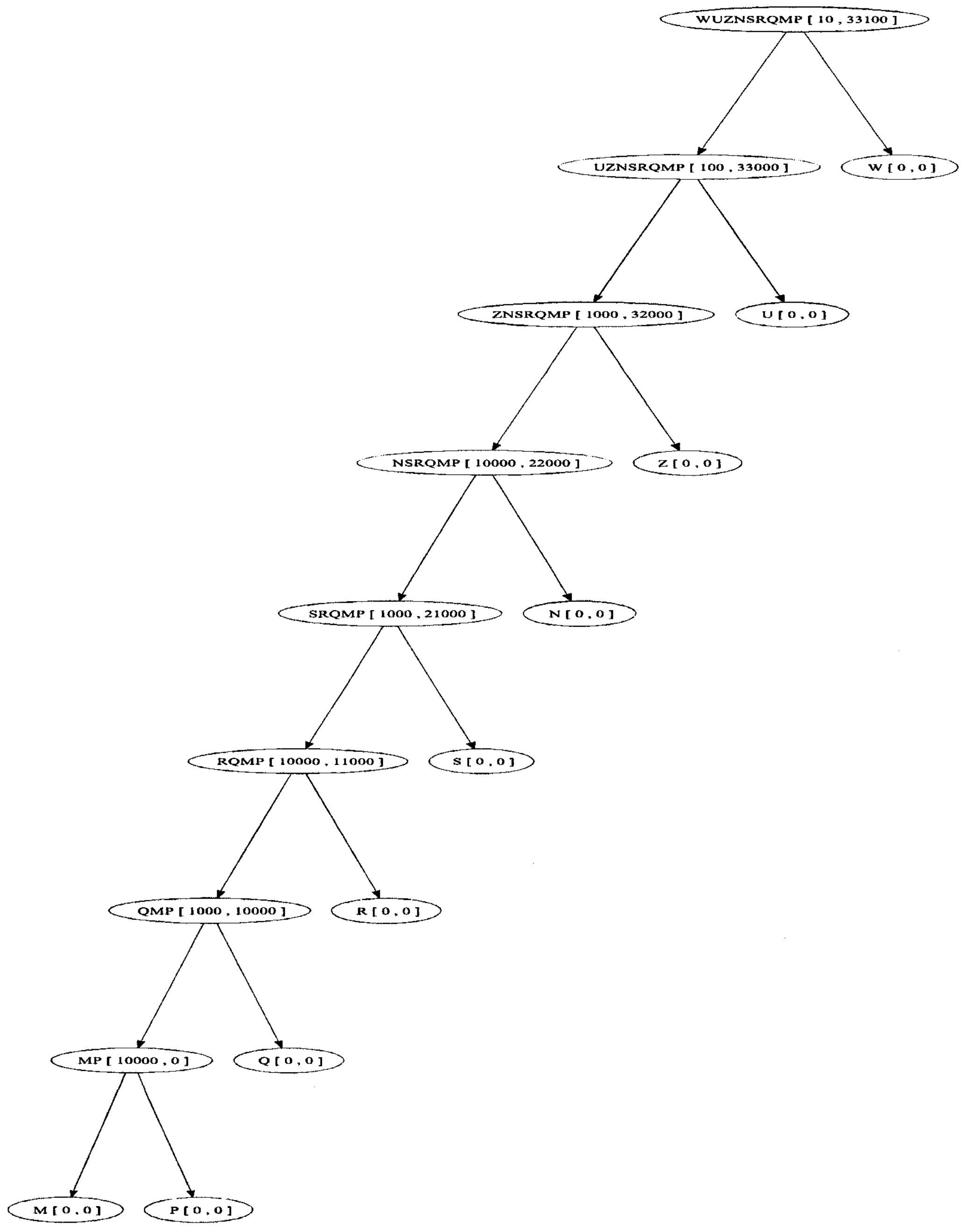


Figure 4.10: The solution given by Greedy algorithm on the circular query given in Table 4.3, with query cost of 11111100000.

4.3.4 Clique query

We use the sample relation in Table 4.4 and present the results produced by all three algorithms. Figure 4.11 gives the result given by Dynamic programming and AO^* and Figure 4.12 gives the result of Greedy.

R	S	Z	U	Q
$T(R) = 1000$	$T(S) = 1000$	$T(Z) = 1000$	$T(U) = 1000$	$T(Q) = 1000$
$V(R, a) = 100$				$V(Q, a) = 100$
$V(R, b) = 100$	$V(S, b) = 100$			
	$V(S, c) = 100$	V(Z,c)=100		
		$V(Z, d) = 100$	$V(U, d) = 100$	
			$V(U, e) = 100$	$V(Q, e) = 100$

M	N	P	W
$T(M) = 1000$	$T(N) = 1000$	$T(P) = 1000$	T(W)=1000
$V(M, a) = 100$		$V(P, a) = 100$	
	$V(N, b) = 100$		
		$V(P, c) = 100$	$V(W, c) = 100$
$V(M, d) = 100$			
	$V(N, e) = 100$		$V(W, e) = 100$

Table 4.4: Reference relations for a clique query.

As shown in Figure 4.11, the optimal solution is a left-deep tree with query cost of 23200, but Greedy outputs another left-deeptree with a query cost which is 100 times more expensive for the same query.

4.3.5 Result of running three algorithms on four reference relations

After running our three algorithms on the four sets of relations described in Tables 4.1 to 4.4, we can summarize the results in Tables 4.5 and 4.6.

As shown in the Tables 4.5 to 4.7, the AO^* algorithm always finds the same optimal solution as dynamic programming does. However the number of operations or the optimization cost is much less than dynamic programming. On the other hand, the

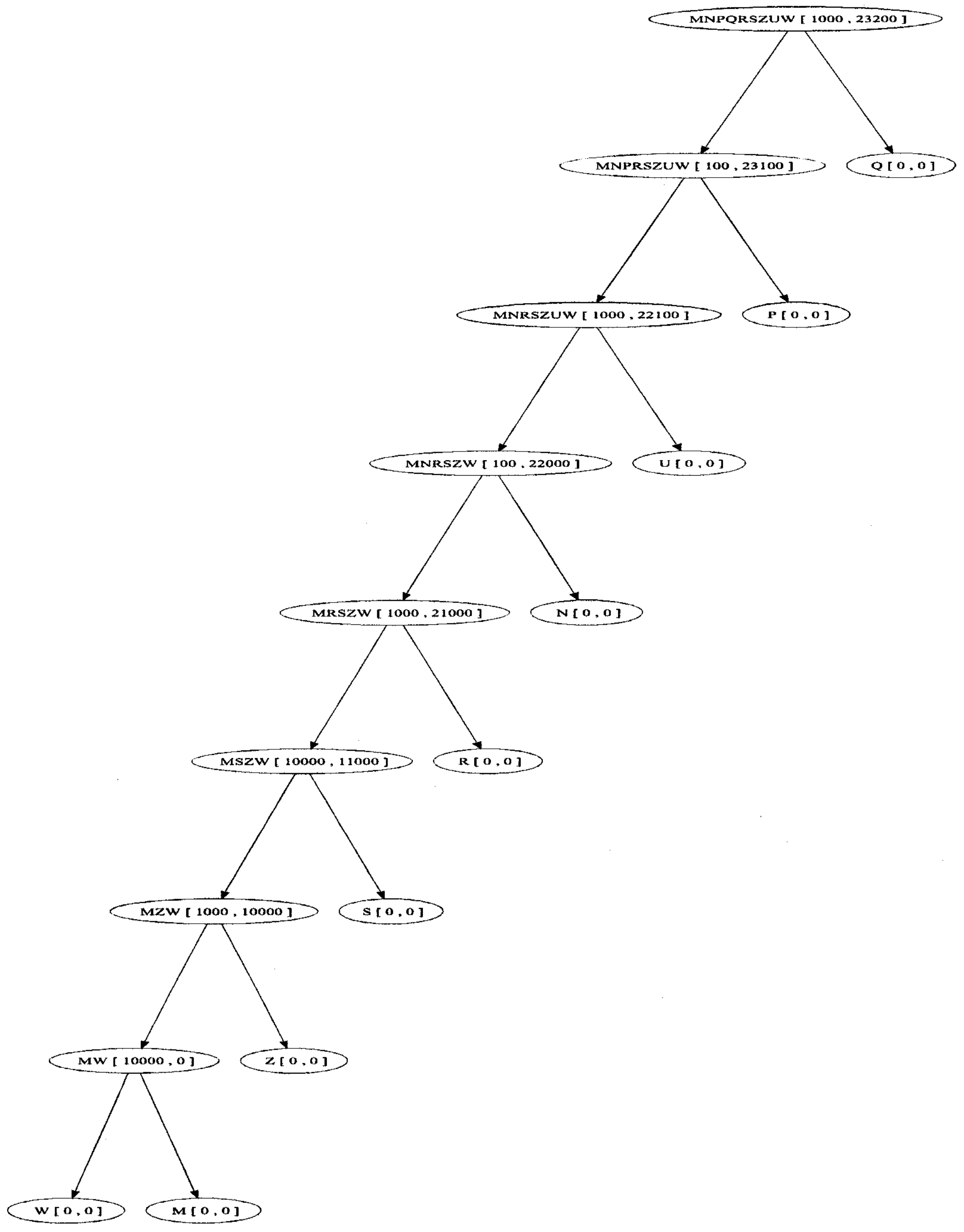


Figure 4.11: The solution given by Dynamic Programming and AO^* on the clique query given in Table 4.4, with query cost of 23200.

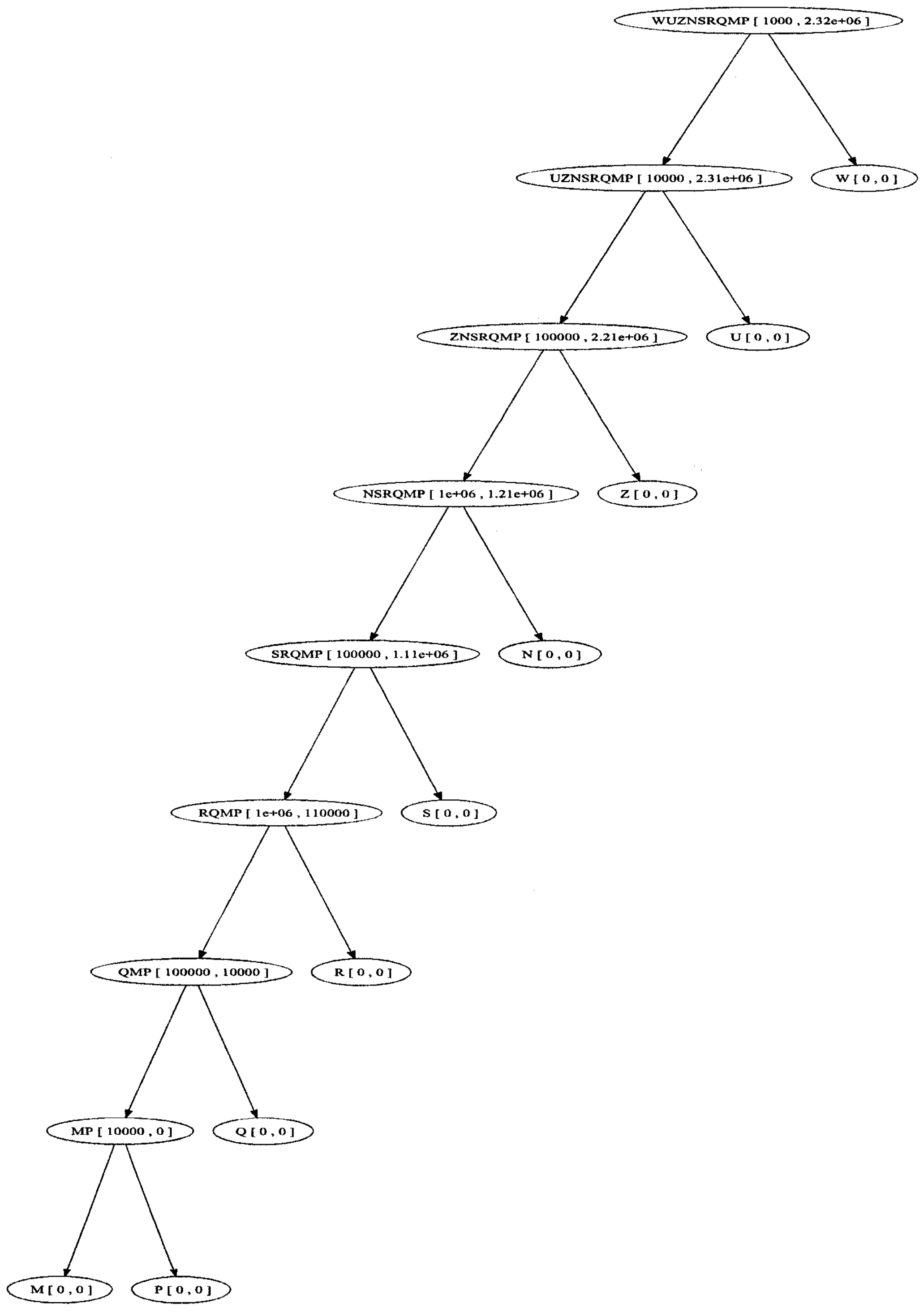


Figure 4.12: The solution given by Greedy algorithm on the clique query given in Table 4.4, with query cost of 2320000.

	Result of the Search	query cost
Chain Q.	$((M \bowtie N) \bowtie (P \bowtie W)) \bowtie ((Q \bowtie Z \bowtie U) \bowtie (R \bowtie S))$	11140000
Star Q.	$(((M \bowtie R) \bowtie N) \bowtie (S \bowtie Q)) \bowtie ((P \bowtie W) \bowtie (Z \bowtie U))$	11140000
Circular Q.	$(((M \bowtie Q) \bowtie U) \bowtie (S \bowtie Z)) \bowtie ((N \bowtie P) \bowtie (R \bowtie W))$	11140000
Clique Q.	$(((((((M \bowtie W) \bowtie Z) \bowtie U) \bowtie Q) \bowtie P) \bowtie S) \bowtie R) \bowtie N)$	23200

Table 4.5: Results of dynamic programming and AO^* algorithm on the reference relations given in Tables 4.1 to 4.4.

query cost produced by the Greedy algorithm is approximately 1000 times the optimal query cost for the first three queries. For the clique query it turns out that the optimal solution is a left-deep tree, but the Greedy algorithm fails to find it, and finds instead a more expensive solution.

4.4 Effect of size of relations (evenly distributed)

To study the effect of the shape of the query on the performance of AO^*, we computed the average cost of query optimization on relations whose sizes were chosen uniformly at random as described below:

- We start with the base relations as given in Tables 4.1 to 4.4.
- The size of each relation was multiplied by different j, where j was an integer chosen randomly from $[10, 100]$ while the attributes of each relation was multiplied by different k, where k was an integer chosen randomly from $[1, 10]$. This test was performed 20 times and the average is presented in results .
- For another set of 20 tests the size of each relation was multiplied by j, where j was an integer chosen randomly from $[100, 1000]$ while the size of each attribute was multiplied by an ineger chosen randomly from $[10, 100]$.
- The last set of 20 tests was performed by multiplying the size of relations by

	Result of the Search	query cost
Chain Q.	$((((((((M \bowtie N) \bowtie P) \bowtie Q) \bowtie U) \bowtie Z) \bowtie S) \bowtie R) \bowtie W)$	11111100000
Star Q.	$((((((((M \bowtie N) \bowtie P) \bowtie Q) \bowtie R) \bowtie S) \bowtie Z) \bowtie U) \bowtie W)$	11111100000
Circular Q.	$((((((((M \bowtie N) \bowtie P) \bowtie Q) \bowtie U) \bowtie Z) \bowtie S) \bowtie R) \bowtie W)$	11111100000
Clique Q.	$((((((((M \bowtie P) \bowtie Q) \bowtie R) \bowtie S) \bowtie N) \bowtie Z) \bowtie U) \bowtie W)$	33100

Table 4.6: Results of Greedy algorithm on the reference relations given in Tables 4.1 to 4.4.

j, where j was an integer chosen from $[1000, 10000]$, while the size of attributes was multiplied by k, where k was an integer chosen randomly from $[100, 1000]$.

- Another set of three tests was performed with the same criteria as 2 to 4 with the exception that the j and k values were chosen unevenly meaning that the j and k values were used for half of the relations chosen randomly.

As seen in Table 4.7, the optimization cost of Greedy and Dynamic programming depend only on the number of relations, and are independent of the shape of the query. However, for AO^*, this factor appears to make a difference.

4.5 Effect of size of relations (unevenly distributed)

As stated before, the optimization cost of Greedy and Dynamic programming again depend only on the number of relations, and are independent of the shape of the query. However, for AO^*, the size of relations does make a difference, specifically for clique shape query the smaller size for relations causes the optimization cost to increase.

4.6 Effect of cost index of solution tree on the optimization cost

As seen in Figures 4.13 through 4.20, there is a correlation between the cost index of the solution tree and the cost of query optimization. The larger the cost index, the less the optimization cost for the query by the AO^* algorithm. It seems that the reason

	Dynamic Programming	AO^* Algorithm	Greedy Algorithm
Chain Q.	9330	304	64
Star Q.	9330	436	64
Circular Q.	9330	496	64
Clique Q.	9330	4224	64

Table 4.7: Optimization costs on the reference relations for the three algorithms.

	$100 \leq size \leq 1000$	$1000 \leq size \leq 10000$	$10000 \leq size \leq 100000$
Chain Q.	304	306	302
Star Q.	484	372	356
Circular Q.	1256	380	346
Clique Q.	6546	4480	2735

Table 4.8: Result of optimization costs for AO^* on the reference relations with unevenly distributed value set.

	$100 \leq size \leq 1000$	$1000 \leq size \leq 10000$	$10000 \leq size \leq 100000$
Chain Q.	284	284	284
Star Q.	305	284	284
Circular Q.	568	298	284
Clique Q.	5864	3756	2686

Table 4.9: Results of optimization costs for AO^* algorithm on the reference relations with evenly distributed value set.

for such a behavior is that the bushier the solution tree, the larger the cost index for that specific solution tree. For our reference sets of relations with 9 relations, the query index for the possible solution graphs would differ between 0 to 13, 0 for a fully left-deep tree and 13 for a fully bushy tree. The bushiness of the solution tree will affect the optimization cost by removing a large number of potential join orders from being evaluated which decreases the optimization cost. It seems that all the discussed factors, like shape, size of relation, and size of value set affect the optimization cost by affecting the shape of the solution tree. A query with a solution tree of bushy shape has less optimization cost by AO^* than a query with solution tree of left-deep shape.

Figures 4.13 to 4.20 show the correlation between *cost index* and optimization cost for all four different query shapes for AO^* algorithm. For each query shape, this correlation is examined in two different evenly and unevenly distributed values of relation size and attribute value set size.

The solutions with higher cost indices have lower optimization costs. It also seems that for *Star, Chain and Circular* queries the *cost index* remains relatively high, resulting in a lower optimization costs but for a *Clique* query the *cost index* remains close to 0 (left-deep tree) resulting in a rather higher optimization cost.

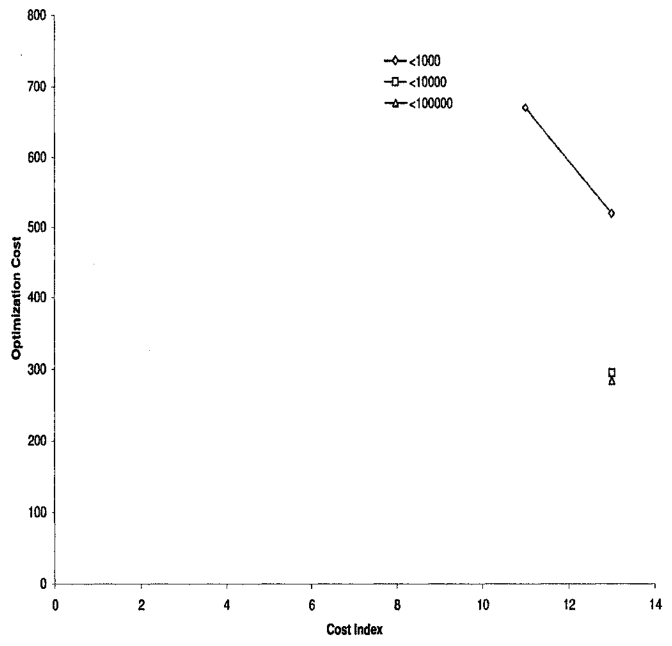


Figure 4.13: Diagram of average of optimization costs vs cost index in a Circular query evenly distributed.

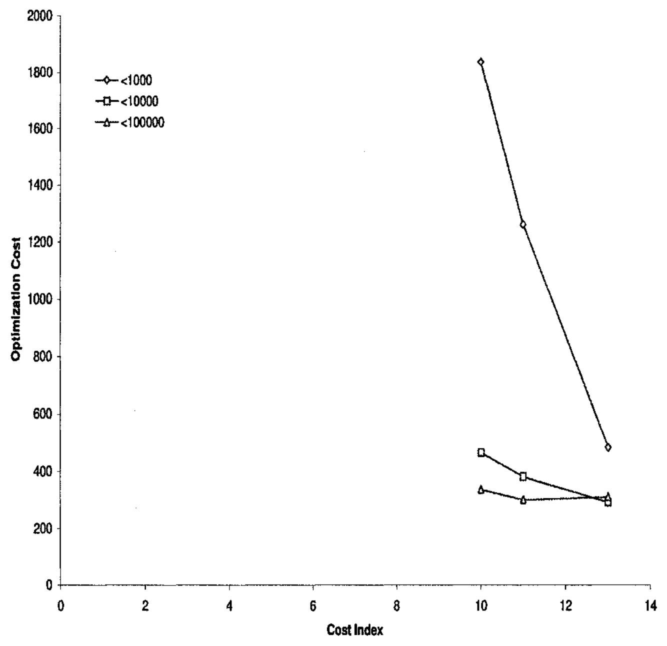


Figure 4.14: Diagram of average of optimization costs vs cost index in a Circular query unevenly distributed.

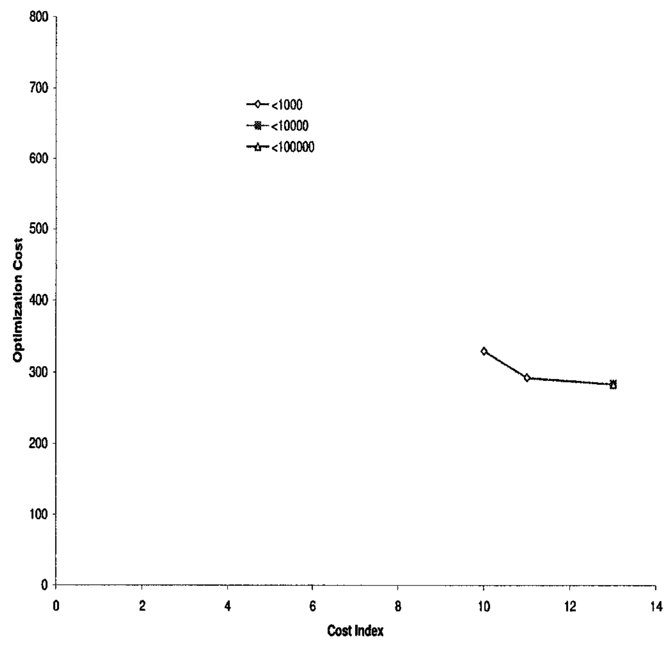


Figure 4.15: Diagram of average of optimization costs vs cost index in a Chain query evenly distributed.

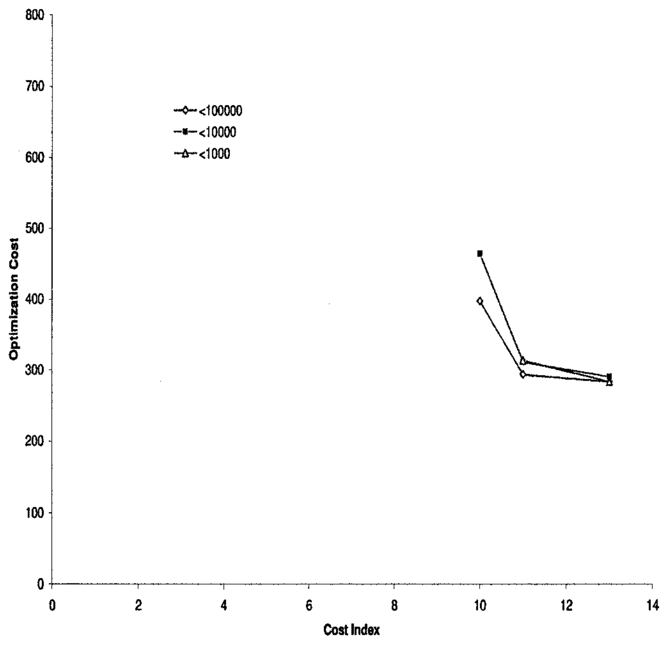


Figure 4.16: Diagram of average of optimization costs vs cost index in a Chain query unevenly distributed.

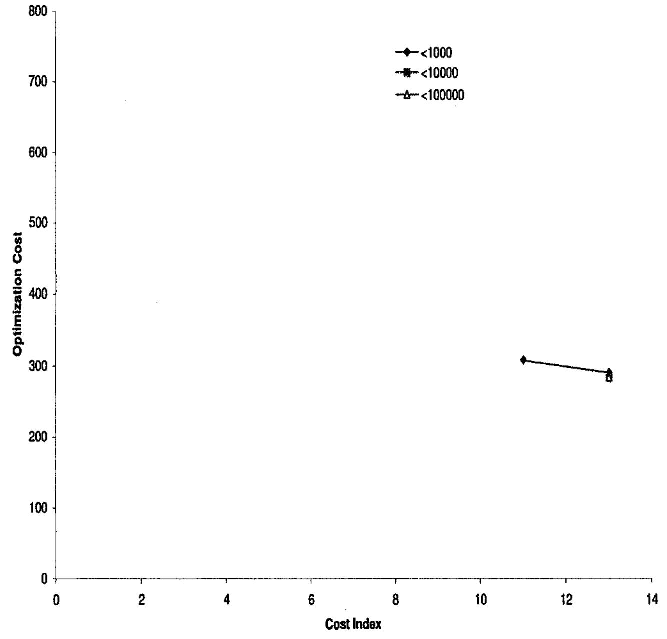


Figure 4.17: Diagram of average of optimization costs vs cost index in a Star query evenly distributed.

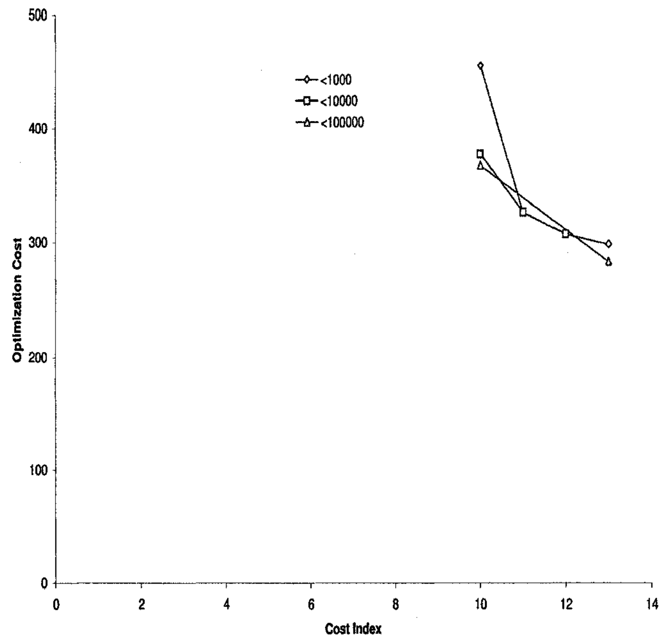


Figure 4.18: Diagram of average of optimization costs vs cost index in a Star query unevenly distributed.

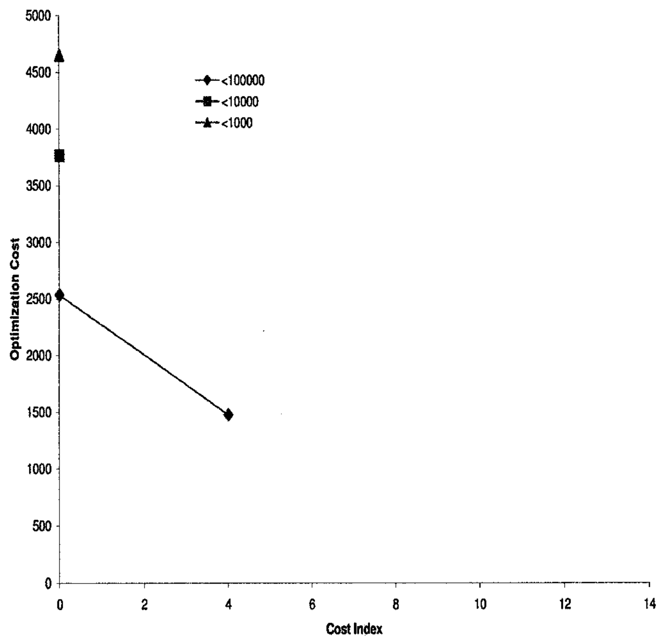


Figure 4.19: Diagram of average of optimization costs vs cost index in a Clique query evenly distributed.

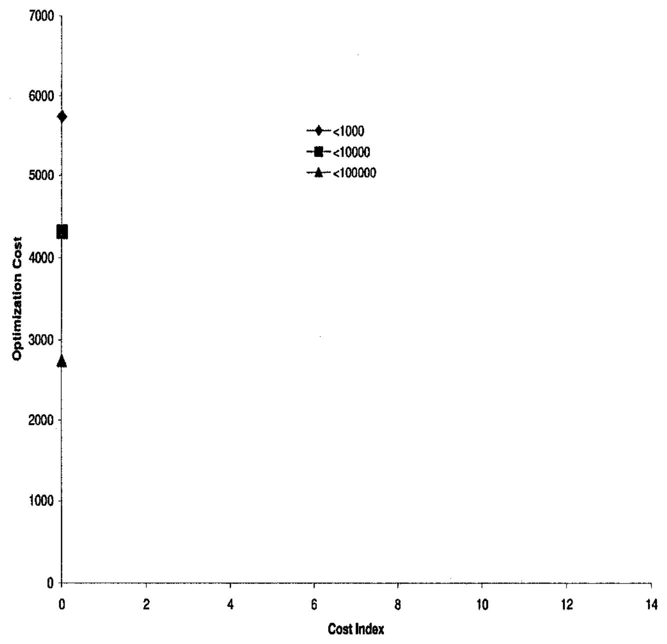


Figure 4.20: Diagram of average of optimization costs vs cost index in a Clique query unevenly distributed.

4.7 Effect of shape of query on the performance of AO^*

To illustrate the effect of the shape of a query on the performance of the AO^* algorithm, we show the average of cost indices of all the experiments run in the previous sections on four different types of queries in Table 4.10. As shown, for *Chain, Circular* and *Star* queries the averages cost indices are around 12 while this value for *Clique* queries is around zero. This is consistent with the results of the previous sections, which show that the optimization cost of star, chain and circular queries are lower than that of clique queries, and that optimization cost is inversely proportional to the cost index of the query.

	Average Cost Index
Chain Query	12.13
Circular Query	11.76
Clique Query	0.27
Star Query	11.82

Table 4.10: The results of averages of cost indices of different query shapes.

4.8 Performance of AO^* Algorithm on Bencmark TPC-D Queries

To show the performance of our AO^* algorithm on commercial schemas, we use the TCP-D benchmark schema [19] and some queries generated based on this schema. You can find the benchmark schema in Figure 4.21 and the relations are presented in Table 4.11.

We have run four queries based on this schema with our AO^* algorithm and the result with the number of operations to find the optimal solution is defined as follows:

- The first query is $(P \bowtie X \bowtie S \bowtie N \bowtie R)$ and the result is:
  $(P \bowtie (X \bowtie (S \bowtie (R \bowtie N))))$

P	X	L	O
$T(P) = 200M$	$T(X) = 800M$	$T(L) = 6B$	$T(O) = 1.5B$
$V(P,a) = 200M$	$V(X,a) = 200M$	$V(L,a) = 200M$	
	$V(X,b) = 10M$	$V(L,b) = 10M$	
		$V(L,e) = 1.5B$	V(O,e)=1.5B
			$V(O,f) = 150M$

S	N	C	R
$T(S) = 10M$	$T(N) = 25$	$T(C) = 150M$	$T(R) = 5$
$V(S,b) = 10M$			
$V(S,c) = 25$	$V(N,c) = 25$	$V(C,c) = 25$	
	$V(N,d) = 5$		$V(R,d) = 5$
		$V(C,f) = 150M$	

Table 4.11: Reference relations for TPC-D benchmark.

The number of operations to find the optimal solution with the AO^* algorithm is 40 while with dynamic programming it would be 90.

- The second query is $(C \bowtie L \bowtie N \bowtie O \bowtie R \bowtie S)$ and the optimal join order is:

  $((C \bowtie O) \bowtie (L \bowtie (S \bowtie (N \bowtie R)))$

  The number of operations to find this query with the AO^* algorithm is 141 while with dynamic programming it would be 301.

- The third query is $(C \bowtie L \bowtie N \bowtie O \bowtie S)$ and the optimal solution is :

  $((C \bowtie O) \bowtie (L \bowtie (N \bowtie S))$

  The number of operations to find the optimal solution with AO^* algorithm is 51 while with dynamic programming it would be 90.

- The fourth query is $(C \bowtie L \bowtie N \bowtie O \bowtie R \bowtie S \bowtie P \bowtie X)$ and the optimal solution is :

  $((((((L \bowtie X) \bowtie S) \bowtie O) \bowtie C) \bowtie P) \bowtie (N \bowtie R))$ The number of operations to find this query with AO^* algorithm is 826 while with dynamic programming it would be 3025.

	Dynamic Programming	AO^* Algorithm
Q1	90	40
Q2	301	141
Q3	90	51
Q4	3025	826

Table 4.12: The optimization costs for AO^* and Dynamic Programming on Benchmark TPC_D queries.

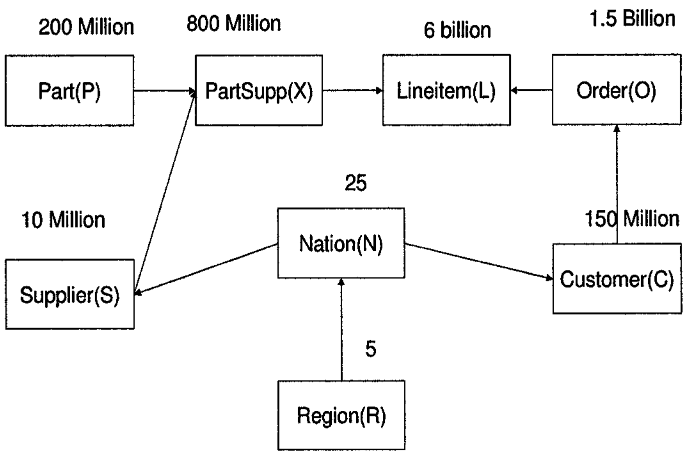


Figure 4.21: The TPC-D Schema.

Chapter 5

Conclusions and Future Work

The problem of join reordering or rewriting is a major task in query optimizers. Two classical techniques for join redordering are dynamic programming and the greedy algorithm. The first algorithm always gives the optimal join ordering, but is very expensive. In contrast, the greedy algorithm is very efficient, but may produce sub-optimal join orders.

In this thesis, we presented a new algorithm called AO^* to find the optimal join ordering for a given set of relations. Our algorithm is based on a well-known problem reduction technique of the same name proposed in the context of decomposable production systems. In our algorithm, we build an AND/OR graph from the relations in the query; the graph represents all possible join orders, and an optimal join ordering is an optimal-cost path in this graph. The algorithm is comprised of a top-down graph growing procedure, as well as a bottom-up cost-revising procedure.

The AO^* algorithm is guaranteed to produce the optimal join ordering, as is dynamic programming. While the worst-case performance cost of AO^* is comparable to that of dynamic programming, in most instances, it is far more efficient. We compared the performance of AO^* with that of dynamic programming and the greedy approach on star-, chain-, circular-, and clique-shaped queries. Our results show that the performance of AO^* is substantially better than dynamic programming for the first three types of queries. While the performance of both dynamic programming and the greedy algorithm are affected only by the number of relations and are independent of the shape of the queries, or the size of the relations or the value sets, it appears that all these factors affect the performance of AO^*.

Remarkably, the shape of the final result also affects the performance of AO^*. When the optimal solution has a high cost index, as defined in this thesis, AO^* appears to converge to it much more rapidly than when it has a relatively low cost index. To understand the reasons for this, as well as the relationship between query shape and cost index would be a fruitful direction of future research. A complete characterization of the worst-case and average-case complexity of AO^* would be also an interesting avenue of research.

Bibliography

[1] M. Kersten A. Pellenkoft, C. Galindo-Legaria. The complexity of transformation-based join enumeration. In VLDB, 1997, pp. 85-97.

[2] A. Gupta A. Swami. Optimization of large join queries. In ACM SIGMOD, 1988, pp. 8-17.

[3] R.E. Bellman. Dynamic Programming. Princeton University Press, 1975.

[4] David Maier Bennet Vance. Rapid bushy join-order optimization with cartesian products. In ACM SIGMOD, 1996, pp. 35-46.

[5] Surajit Chaughuri. An overview of query optimization in relational systems. In ACM SIGMOD, 1998, pp. 34-43.

[6] C. Date. An introduction to Database systems. Addison-Wesley, California, 1986.

[7] C. Date. Relational Database:Selected Writings. Addison-Wesley, California, 1986.

[8] L. Haas et al. Extensible query processing in starburst. In ACM SIGMOD, 1989, pp. 143-160.

[9] G. Graefe and D.J. DeWitt. The exodus optimiser generator. In ACM SIGMOD, 1987, pp. 160-172.

[10] Jeffrey D. Ullman Hector Garcia-Molina and Jennifer Widom. Database Systems: The Complete Book. The Prentice-Hall Company, California, 2002.

[11] T. Imielinski. Advances in Database theory. The Plenum Press, New York, 1984.

[12] Yannis E. Ioannidis. Query optimization. In Database and Information Retrieval, 1997, pp. 1038-1057.

[13] E. Wang K. Youssefi. Query processing in a relational database management system. In ACM SIGMOD, 1997, pp. 223-241.

[14] Guy M. Lohman Kiyoshi Ono. Measuring the complexity of join enumeration in query optimization. In VLDB, 1990, pp. 314-325.

[15] G.M. Lohman L.F. Mackert. Optimizer validation and performance evaluation for local queries. In ACM SIGMOD, 1986, pp. 149-159.

[16] G.M. Lohman. Is query optimization a solved problem? In ACM SIGMOD, 1988, pp. 110-123.

[17] William J. McKenna. Efficient Search Extensible Database Query Optimization: The Volcano Optimizer Generator. PhD thesis, University of Colorado at Boulder, 1993.

[18] Nils J. Nilsson. Principles of Artificial Intelligence. Tioga Publishing Co., Palo Alto, California, 1980.

[19] TPC organization. The tpcd queries. In Introduction to Queries., http://www.tpc.org/tpcd/default.asp.

[20] D.S.Parker P. Atzeni. Assumptions in relational database theory. In ACM Symp. on Principles of Database Systems., 1982, pp. 76-90.

[21] T.G. Price P.G. Selinger, R.A. Lorie. Access path selection in a relational database management system. In ACM SIGMOD, 1979, pp. 170-183.

[22] S.B. Navathe R. Elmasri. Fundamentals of Database Systems. Addison-Wesley, Vancouver, 2000.

[23] A. Levy R. Pottinger. A scalable algorithm for answering queries using views. In VLDB, 2000, pp. 243-261.

[24] J. Gehrke R. Ramakrishnan. Database Management Systems. McGraw Hill, 2000.

[25] V. Vianu S. Abiteboul, R. Hull. Foundations of Databases. Addison-Wesley, 1995.

[26] R. Badrinath T. Imielinski. Querying in highly mobile distributed environments. In VLDB, 1992, pp. 44-54.

[27] Jeffrey D. Ullman. Principles of Database Systems. Computer Science Press, Rockville, 1982.

[28] M.C. Shan W. Du, R. Krishnamurthy. Query optimization in a heterogeneous dbms. In VLDB, 1992, pp. 128-137.

[29] Younkyung Cha Kang Yannis E. Ioannidis. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In ACM SIGMOD, 1991, pp. 168-177.