Correspondence

Fast Search Algorithms for the N-Queens Problem

Rok Sosič and Jun Gu

Abstract—**The n-queens problem is a classical search problem in the artificial intelligence (AI) area. In recent years, this problem has found many useful, practical applications including two-dimensional (2-D) VLSI routing and testing, maximum full range communication, and parallel optical computing. In this paper we present two new algorithms, called Queen Search 2 (QS2) and Queen Search 3 ($QS3$). $QS2$ and $QS3$ are probabilistic local search algorithms, based on a gradient-based heuristic. These algorithms, running in almost linear time, are capable of finding a solution for extremely large size n-queens problems. For example, $QS3$ can find a solution for 500 000 queens in approximately 1.5 min.**

I. Introduction

The n-queens problem is to place n queens on an n by n chessboard so that no two queens attack each other. This classical combinatorial search problem has traditionally been used as a popular testbed to explore new AI search strategies and algorithms. In recent years, this problem has found many practical scientific and engineering applications including two-dimensional (2-D) VLSI routing and testing, maximum full range communication, and parallel optical computing.

Solutions to the n-queens problem, i.e., to formulate a nonconflicting pattern of n objects within an n by n square, represent a solution to the maximum coverage problem. The maximum coverage problem is that, from any horizontal direction, any vertical direction, or any diagonal direction, all n objects can be accessed without conflict. Since n is the maximum number of objects ("queens") that can be placed within an n by n square, solutions of the n-queens problem thus achieve the maximum coverage with n objects.

In this paper we present two new, probabilistic, local search algorithms based on a gradient-based heuristic. These algorithms run in almost linear time and are capable of finding a solution for extremely large size n-queens problems. For example, on a *NeXT* personal computer, our $QS3$ algorithm can find a solution for 500 000 queens in approximately 1.5 min.

In Section II, we briefly review the prior art for the n-queens problem. An application example of the n-queens problem to achieve maximum parallelism in optical computing is illustrated in Section III. New algorithms, *Queen Search 2 (QS2)* and *Queen Search 3 (QS3)*, and their search technique for the n-queens problem are described in Section IV. We present the run-time measurements of the *QS2* and *QS3* algorithms in Section V. The conclusions are given in Section VI.

Manuscript received October 21, 1990; revised April 21, 1991. This research has been supported in part by the University of Utah research fellowships, in part by the Research Council of Slovenia, in part by ACM/IEEE academic scholarship awards.

R. Socič is with the Department of Computer Science, University of Utah, Salt Lake City, UT 84112.

J. Gu is with the Department of Electrical Engineering, University of Calgary, Calgary, AB T2N 1N4, Canada.

IEEE Log Number 9102942.

II. Prior Art

In a little known work, published in 1918, Ahrens [1] described an analytical solution to the general n-queens problem. Similar solutions are presented in [2], [5], [10]. Current analytical solutions have an inherent limitation in that they generate only a very restricted class of solutions. There is a large class of solutions to the n-queens problem not covered by analytical solutions. Since, in general, this limitation does not apply to search—based algorithms, it is desirable to investigate alternatives to analytical solutions.

One method for solving the n-queens problem that systematically generates all possible solutions is known as *backtracking search* [3], [6], [8], [15]. Sosič and Gu gave a nonbacktracking, fast *Queen Search 1 (QS1)* algorithm for the n-queens problem [11]–[13]. In [4], *QS1* is compared with backtracking search algorithms and is found to be considerably faster. Although the worst case performance of backtracking search is exponential in time, some analyses show that under certain conditions the expected average complexity of backtracking may not be exponential [9], [14]. Recently, other fast search algorithms for the n-queens problem have been reported in [6], [7].

In Section II, we present two new, probabilistic, local search algorithms that are derived from *QS1*. They are based on a gradient-based heuristic. The algorithms, *QS2* and *QS3*, run in almost linear time, do not use backtracking, and are capable of finding a solution for extremely large size n-queens problems within a reasonably short time period.

III. An Application Example in Optical Computing

One important application of the maximum coverage problem is in optical computing. To achieve high communication bandwidth, a 2-D array of n optical—processing elements must be placed without any overlap. With n computing elements distributed in a nonconflicting pattern, each element can communicate with the outside world in eight directions (i.e., two horizontal directions, two vertical directions, and four diagonal directions) without being obscured by other elements. Fig. 1 shows an example of a maximum—coverage placement of six optical—processing elements, which is one solution to the 6-queens problem.

IV. *QS2* and *QS3*: Efficient Search Algorithms for the *N*-Queens Problem

A. Data Structures

Queen Placement: Let N be the size of the board and let each row contain exactly one queen. When N queens are arranged on the board, their column positions are stored in array $queen$ of length N. The ith queen is placed on the board at row i and column $queen[i]$. Array $queen$ contains a permutation of integers $1, \cdots, N$. This guarantees that no two queens attack each other on the same row or the same column. The problem remains to resolve any collisions among queens that may occur on the diagonal lines.

Collisions on Diagonal Lines: On a square board, there are two types of diagonal lines, diagonal lines with a positive slope and diagonal lines with a negative slope. Let i be a *row index* and j be a *column index*. It is assumed that the index labeling starts with the

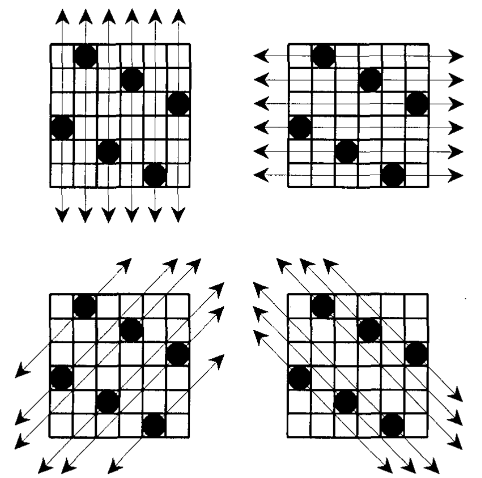

Fig. 1. A nonconflicting placement of six optical processing elements (each optical element can be accessed from any vertical, horizontal, and diagonal direction).

lower left corner. For all squares on any diagonal line with a *negative slope*, the *sum* of both square indexes is constant. For example, on a 4 × 4 board, one diagonal line with a negative slope passes through squares (3,1), (2,2), and (1,3). Similarly, for all squares on any diagonal line with a *positive slope*, the *difference* of both square indexes is constant. Since sums and differences differ for different diagonal lines, they are used to characterize the diagonal lines. A square in row i and column j is on a negative slope diagonal line with index $i+j$ and on a positive slope diagonal line with index $i-j$. On a board of size N, negative slope diagonal lines use indexes $1-N$, $\cdots$, $N-1$ and positive slope diagonal lines use indexes $2, \cdots, 2 * N$.

An array, denoted as dn, keeps track of the number of queens on diagonal lines with negative slope. If there are k queens on the mth diagonal line with negative slope, number k is stored in $dn[m]$. Similarly, array dp keeps track of the number of queens on diagonal lines with positive slopes. The number of collisions on any diagonal line is one less than the number of queens on that line, i.e., $k-1$.

Function $compute\_collisions$ initializes arrays dn and dp. It returns the initial total number of collisions on the diagonal lines that exist in the initial queen placement.

The Attacked Queens: Another array, denoted as $attack$, is maintained to speed up the search process. This array stores row indexes of queens in array $queen$ that are attacked by other queens. Array $attack$ is computed by function $compute\_attacks$. This function returns the number of attacked queens.

B. The QS2 Algorithm

The $QS2$ algorithm is shown in Fig. 2. The main procedure is called $queen\_search2$.

Initialization: At the beginning of the search, an initial random permutation of numbers $1, \cdots, N$ is generated and the resulting numbers (i.e., the column positions) are stored in array $queen$. Arrays dn and dp are initialized and the total number of collisions on diagonal lines is stored in variable $collisions$. Array $attack$ is initialized and the number of attacked queens is stored in $number\_of\_attacks$.

After these initial steps, search steps are repeated a certain number of times depending on n, the size of the problem. The search process is terminated if a solution is found; otherwise it is repeated for a new random permutation.

```
1.  procedure queen_search2 (var queen : array [1 .. N] of integer);
2.  var
3.     dn : array [2 .. 2 * N] of integer;
4.     dp : array [1 - N .. N - 1] of integer;
5.     attack : array [1 .. N] of integer;
6.     limit,collisions,number_of_attacks,loopcount : integer;
7.     i,j,k : integer;
8.  begin
9.     repeat
10.                    { initialization }
11.       Generate a random permutation of queen[1] to queen[N];
12.       collisions := compute_collisions(queen,dn,dp);
13.       limit := C1 * collisions;
14.       number_of_attacks := compute_attacks(queen,dn,dp,attack);
15.       loopcount := 0;
16.                    { search }
17.       repeat
18.          for k := 1 to number_of_attacks do begin
19.             i := attack[k];
20.             Choose j in [1..N];
21.             if swap_ok(i,j,queen,dn,dp) then begin
22.                perform_swap(i,j,queen,dn,dp,collisions);
23.                if collisions = 0 then return;
24.                if collisions < limit then begin
25.                   limit := C1 * collisions;
26.                   number_of_attacks := compute_attacks(queen,dn,dp,attack);
27.                end;
28.             end;
29.          end;
30.          loopcount := loopcount + number_of_attacks;
31.       until loopcount > C2 * N;
32.
33.    until collisions = 0;
34. end;
```

Fig. 2. $QS2$: An efficient N-queens search algorithm.

The Search Process: During one search step, two queens with indexes i and j are considered. Queen i is selected from queens stored in array $attack$ (line 19 in Fig. 2). Queen j is a randomly selected queen (line 20 in Fig. 2).

The test to see if a swap of two queens will reduce the total number of collisions is performed by function $swap\_ok$. Function $swap\_ok$ returns $true$ if the swap reduces the total number of collisions; otherwise it returns $false$. Since each queen can affect at most two diagonals, only eight diagonals need to be checked: four for two original queen positions and four for two new queen positions.

The actual swap is performed by procedure $perform\_swap$. Procedure $perform\_swap$ updates arrays $queen$, dn, and dp. It also maintains the number of collisions in variable $collisions$. When the number of collisions reaches 0, a solution has been found and procedure $queen\_search2$ is terminated.

The Termination of the Search: Since the algorithm is probabilistic, it can not guarantee a solution for every initial permutation. If a solution is not found after a certain number of search steps, a new permutation is generated and the search process is repeated. During numerous algorithm runs, for problem sizes with N greater than 1000, the algorithm has always found a solution for the first permutation.

Variable $loopcount$ counts the number of tested pairs. When this number exceeds the value of $C2 * N$, a new permutation is generated. Constant $C2$ has been set to 32 in order to maximize algorithm execution speed for small N. Constant $C2$ has no effect on the running time for large n, because a solution is usually found for the first initial permutation.

Maintaining Queens Under Attack: Array $attack$ stores row indexes of queens that are under attack. Variable $number\_of\_attacks$ is the number of queens under attack. In general, elaborate bookkeeping is needed to maintain array $attack$ and variable $number\_of\_attacks$ consistent with the queen placement.

To avoid the cost of bookkeeping, a different strategy is chosen. Array $attack$ and variable $number\_of\_attacks$ are not updated

TABLE I
THE EXECUTION TIME OF THE $QS3$ ALGORITHM ON A NeXT PERSONAL COMPUTER (AVERAGE OF 10 RUNS; TIME UNITS: SECONDS)

Number of Queens n	1000	10 000	100 000	500 000
Queens with Conflict	50	50	80	100
Time of the 1st Run	0.3	1.7	16.4	92.4
Time of the 2nd Run	0.3	1.8	16.4	91.4
Time of the 3rd Run	0.3	1.7	16.5	92.7
Time of the 4th Run	0.4	1.7	16.1	91.6
Time of the 5th Run	0.3	1.7	16.1	94.6
Time of the 6th Run	0.4	1.7	16.3	95.6
Time of the 7th Run	0.3	1.6	16.5	96.3
Time of the 8th Run	0.3	1.7	16.3	94.7
Time of the 9th Run	0.3	1.7	16.7	95.2
Time of the 10th Run	0.4	1.7	16.8	95.0
Avg. Time to Find a Solution	0.33	1.7	16.4	94.0

TABLE II
THE EXECUTION TIME OF THE $QS2$ ALGORITHM ON A NeXT PERSONAL COMPUTER (AVERAGE OF 10 RUNS; TIME UNITS: SECONDS)

Number of Queens n	1000	10 000	100 000	500 000
Time of the 1st Run	0.7	7.5	87.2	494.5
Time of the 2nd Run	0.8	7.3	84.2	488.3
Time of the 3rd Run	0.7	6.9	88.3	495.6
Time of the 4th Run	0.7	7.1	86.7	498.1
Time of the 5th Run	0.8	6.9	86.6	488.5
Time of the 6th Run	0.8	7.4	86.3	484.5
Time of the 7th Run	0.7	7.5	86.7	485.8
Time of the 8th Run	0.8	7.6	85.5	489.4
Time of the 9th Run	0.7	7.5	87.2	472.9
Time of the 10th Run	0.7	6.9	88.1	492.7
Avg. Time to Find a Solution	0.74	7.26	86.7	489.0

after every queen swap, but instead only when the number of collisions falls below the value in variable $limit$. Variable $limit$ is initialized to a certain percentage (denoted as constant $C1$) of the number of collisions in variable $collisions$. After the number of collisions becomes less than the value of $limit$, array $attack$ and variable $number\_of\_attacks$ are updated, and the value of $limit$ is reinitialized (lines 24–27 in Fig. 2). This adaptive adjustment of $limit$ has reduced computing cost and increased algorithm efficiency.

The optimal value of $C1$ depends on the detailed implementation of the algorithm. In order to speed up algorithm execution, $C1$ was investigated over a wide spectrum of possible values. In our case, $C1$ is set to 0.45.

C. The QS3 Algorithm

The initial permutation in the $QS2$ algorithm is completely random. It was observed in [12], [13] that a random permutation of n integers generates approximately $n \times 0.53$ collisions on the diagonal lines. For example, a random permutation of 500 000 numbers generates approximately 265 000 collisions among queens. The $QS2$ algorithm can be made more efficient if the collisions in the initial permutation itself can be minimized. This is the basic idea behind the $QS3$ algorithm.

In the $QS3$ algorithm, an initial random permutation is generated such that the number of collisions among queens is minimized. The search part of the algorithm is identical to the $QS2$ algorithm. The position for a new queen to be placed on the board in the next column is randomly generated until a conflict free place is found for this queen. After a certain number of queens have been placed in a conflict free manner the queens in remaining columns are placed randomly regardless of conflicts on diagonal lines. This process of generating the initial permutation does not require backtracking.

The number of queens placed in a conflict free manner needs to be chosen carefully. If this number is too small the $QS3$ algorithm shows no improvement over the $QS2$ algorithm. If this number is too large the initialization either takes too long or it does not terminate. The number of conflict free queens in the initial permutation that we have chosen in our real time runs vary with n. Its value is less than or equal to 100 for n up to 500 000. This number is shown together with n and the real execution time of the $QS3$ algorithm in Table I.

V. MEASUREMENTS AND RESULTS

If the random number generator is initialized with a different value, both algorithms, $QS2$ and $QS3$, produce different solutions to the n-queens problem in different runs. In principle, they are able to produce any solution and thus are not limited to a restricted class of solutions as analytical methods are. This advantage of $QS2$ and $QS3$ is important when a large number of solutions is requested or when solutions must satisfy certain constraints. $QS2$ and $QS3$ can not guarantee to find all solutions to the n-queens problem. However, this is not a problem, since there is strong evidence that the number of solutions to the n-queens problem increases exponentially with n. For example, the total number of solutions for n equals 7 to 12 is: 40, 92, 352, 724, 2680, and 14 032 [1]. The algorithm to find all solutions must be exponential just to print out results. This takes a prohibitive amount of time even for small values of n.

We summarize some experimental data below. Among various algorithm features we have studied, the following observations are of particular interest.

A. Real Execution Time of the QS2 and QS3 Algorithms

The real execution time of the *QS2* and *QS3* algorithms, programmed in C and run on a *NeXT* personal computer (with a 25 MHz Motorola 68030 processor), are illustrated in Tables I and II, respectively. Due to the memory limitation of our computer, the largest problem size we were able to run was 500 000. The *Queens with Conflict* in Table I displays the maximum number of queens with a conflict that may be generated by initialization. Table III gives the running time of our previous $QS1$ algorithm [12], [13]. Compared to the $QS1$ algorithm (see Table III), the execution speed of the $QS3$ is more than 100 times faster for n equal to 500 000.

B. The Probabilistic Behavior of the Algorithms

Tables in this section were obtained from 10 sample algorithm runs. They illustrate the behavior of the $QS2$ and $QS3$ algorithms.

The $QS2$ and $QS3$ algorithms are probabilistic in that, if the algorithm could not find a solution from a given random permutation, a new permutation is generated and the algorithm starts a new search. Table IV shows the probabilistic behavior of $QS2$ regarding algorithm success in finding a solution from an initial random permutation. The behavior of $QS3$ is similar and it is not shown in a separate table. The *solution in the first permutation* represents, among 10 sample algorithm runs, the number of times a solution is found based on an initial (the first) permutation. The *maximum number of permutations* is, within 10 sample algorithm runs, the maximum number of permutations that were required to find a solution in one

TABLE III
THE EXECUTION TIME OF THE $QS1$ ALGORITHM ON A NeXT PERSONAL COMPUTER (AVERAGE OF 10 RUNS; TIME UNITS: SECONDS)

Number of Queens n	1000	10 000	100 000	500 000
Time of the 1st Run	2.1	27.7	1098.4	7500
Time of the 2nd Run	1.9	38.2	1081.2	9065
Time of the 3rd Run	1.8	42.6	997.6	12 617
Time of the 4th Run	3.1	34.9	979.9	11,730
Time of the 5th Run	1.9	34.3	1286.4	9934
Time of the 6th Run	2.4	31.2	992.3	9198
Time of the 7th Run	1.9	41.2	1425.5	9789
Time of the 8th Run	3.3	36.5	1235.4	11 142
Time of the 9th Run	2.3	52.4	1285.7	11 788
Time of the 10th Run	2.1	35.1	1285.4	8300
Avg. Time to Find a Solution	2.3	37.0	1167.0	10 106

TABLE IV
PERMUTATION STATISTICS OF $QS2$[a]

Number of Queens n	Solution in the First Permutation	Maximum Number of Permutations
10	4	26
100	3	5
1000	10	1
10 000	10	1
100 000	10	1
500 000	10	1

[a] (Average of 10 runs)

TABLE V
SEARCH STATISTICS FOR $QS2$[a]

Number of Queens n	Number of Pairs Tested	Number of Swaps Performed
1000	14 742	436
10 000	138 726	4 333
100 000	1 386 974	43 256
500 000	6 981 193	216 407

[a]Average of 10 runs.

program run. It can be seen that the number of permutations required to find a solution is very small and it goes to 1 with increasing n. That is, for large n, the probability that the first permutation will lead to a solution is close to 1. From numerous algorithm runs we observe that the algorithm always found a solution in the first permutation for n greater than 1000.

Table V shows the parts of the $QS2$ algorithm on which the most CPU time was spent. The *number of pairs tested* indicates the total number of calls of function *swap_ok* (line 21 in Fig. 2). The *number of swaps performed* gives the number of times that *swap_ok* returns *true* and an actual swap was performed (lines 22–27 in Fig. 2). Both *number of pairs tested* and *number of swaps performed* increase linearly with increasing n (number of queens). The algorithm is thus linear in the number of performed tests and swaps.

Table VI demonstrates search statistics for $QS3$. Search features of $QS3$, presented in Table VI, are the same as the features of $QS2$ in Table V. Because the initialization in $QS3$ significantly reduces the number of queen collisions, the search part of $QS3$ takes much less time than the search part of $QS2$. This reduction in search time is done at the expense of longer initialization time. Next, we present the behavior of the initialization part of the $QS3$ algorithm.

TABLE VI
SEARCH STATISTICS FOR $QS3$[a]

Number of Queens n	Number of Pairs Tested	Number of Swaps Performed
1000	5 278	49
10 000	7 085	52
100 000	10 982	81
500 000	13 108	98

[a]Average of 10 runs.

TABLE VII
INITIALIZATION STATISTICS OF $QS3$[a]

Number of Queens n	Minimum	Maximum	Average
10	6	10	8
100	88	99	93
1000	967	997	985
10 000	9949	9989	9978
100 000	99 953	99 992	99 979
500 000	999 944	999 979	999 967

[a]Average of 10 runs.

Table VII shows the number of queens that may be placed without conflicts during the initialization step after $5 \times n$ placement attempts. The table shows the minimum, maximum, and the average number of placed queens in 10 runs. The number of successfully placed queens is very high even for large values of n. For example, on average all queens except the last 33 queens were placed without conflicts for n equal to one million.

To find the rate at which queens are placed, we present the board saturation versus the number of placement attempts (see Fig. 3). If m denotes the number of queens placed without conflicts, then m/n represents the board saturation. The board saturation is 1 when the placement of n queens is a solution to the n-queens problem. Since Fig. 3 shows the behavior for n equal to 100, 1000, and 1 000 000, the number of placement attempts is normalized by n. The lower curve represents the case of n equal to 100, the top curve the case of n equal to 1 000 000, and the middle curve the case of n equal to 1000. It can be observed that all three curves are surprisingly similar. The increase of saturation slows down rapidly at a point close to $3 \times n$. This number, $3 \times n$, is a good estimate for the number of placement attempts that are performed during the initialization part of $QS3$.

Since the number of placement attempts grows linearly with n and the number of search steps grows much slower than n, it follows that the $QS3$ algorithm finds a solution to the n-queens problem in a linear number of steps. A very small nonlinear time component is added to the running time for maintaining array *attack*.

VI. CONCLUSION

Two search algorithms able to find a solution for very large n-queens problems are presented. The algorithms run in almost linear time. Their performance is achieved by applying a gradient-based heuristic within a local search.

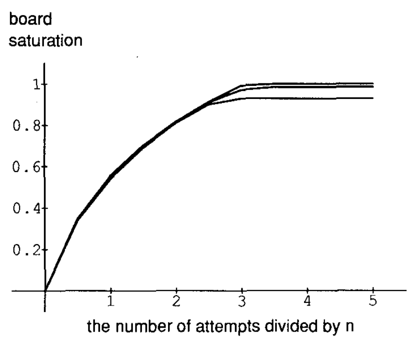

Fig. 3. Board saturation in $QS3$ for n=100, 1000, and 1 000 000.

ACKNOWLEDGMENT

We thank Vladimir Batagelj who pointed out a reference to the earlier work by Ahrens [1]. Bob Johnson, Vipin Kumar, and Tomaž Pisanski provided relevant and instructive comments on various early versions of this paper. The authors would like to thank anonymous referees for their valuable comments and additional references. Jack Mostow pointed out a recent work in [7].

REFERENCES

[1] W. Ahrens, *Mathematische Unterhaltungen und Spiele*, (in German). Leipzig, Germany: B. G. Teubner, pp. 1918–1921.
[2] B. J. Falkowski and L. Schmitz, "A note on the queens' problem," *Inform. Processing Lett.*, vol. 23, pp. 39–46, July 1986.
[3] J. Gaschnig, *Performance Measurements and Analysis of Certain Search Algorithms.* Ph.D. thesis, Carnegie-Mellon Univ., Dept. Comput. Sci., May 1979.
[4] J. Gu, "Parallel algorithms and architectures for very fast AI search," Tech. Rep. UUCS-TR-88-005, Univ. Utah, Dept. Comput. Sci., July 1988.
[5] E. J. Hoffman, J. C. Loessi, and R. C. Moore, "Constructions for the solution of the m queens problem," *Mathematics Magazine*, pp. 66–72, 1969.
[6] L. V. Kalé, "An almost perfect heuristic for the n nonattacking queens problem," *Inform. Processing Lett.*, vol. 34, pp. 173–178, Apr. 1990.
[7] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird, "Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method," in *Proc. AAAI90*, Aug. 1990, pp. 17–24.
[8] B. A. Nadel, "Representation selection for constraint satisfaction: A case study using n-queens," *IEEE Expert*, Jun. 1990, pp. 16–23.
[9] D. Nicol, "Expected performance of m-solution backtracking," *SIAM J. on Computing*, ,vol. 17, no. 1, pp. 114–127, Feb. 1988.
[10] M. Reichling, "A simplified solution of the n queens' problem," *Inform. Processing Lett.*, vol. 25, pp. 253–255, June 1987.
[11] R. Sosič and J. Gu. Fast, "n-queen search on vax and bobcat machines," AI Project Report, Feb. 1988.
[12] ——, "How to search for million queens," Tech. Rep. UUCS-TR-88-008, Dept. Comput. Sci., Univ. Utah, Feb. 1988 (also available from authors).
[13] ——, "A polynomial time algorithm for the n-queens problem," *SIGART Bull.*, vol. 1, no. 3, pp. 7–11, Oct. 1990.
[14] H. S. Stone and P. Sipala, "The average complexity of depth-first search with backtracking and cutoff," *IBM J. Res. Develop.*, vol. 30, no. 3, pp. 242–258, May 1986.
[15] H. S. Stone and J. M. Stone, "Efficient search techniques—an empirical study of the n-queens problem," *IBM J. Res. Develop.*, vol. 31, no. 4, pp. 464–474, July 1987.

A Fuzzy Query Language for Relational Databases

Yoshikane Takahashi

***Abstract*— A fuzzy query language (FQL) for relational databases is proposed. FQL, constructed as an enhancement of the relational domain calculus, has sufficient capability to represent all four types of the fuzzy statements distinguished by the work of L. A. Zadeh. The idea to construct FQL is in the formulation of the fuzzy matching degrees that assigns the appropriate values in the interval [0, 1] to any combination of fuzzy queries and tuples in relational databases. FQL helps to provide a human-oriented interface with currently widespread relational databases that altogether have stored a vast amount of information. Furthermore, fuzzy expert systems are expected to be provided with the facility to make use of fact data in relational databases through FQL. In addition, FQL is a theoretical basis for systematically developing a higher human-oriented interface with relational databases.**

I. Introduction

One of the most important technologies in relational databases, compared to tree or hierarchical-type databases, has been developed to improve the query language more fitted to nonexpert users. The higher level the query language will support a human-oriented interface, the more complex it must involve imprecise statements. Those imprecise statements used in the nonexpert users' queries to databases with natural languages include many fuzzy statements such as "X is very small." and "Abe is young is not very true." Typical fuzzy statements were distinguished and classified into four types that are represented by the formal language PRUF (possibilistic relational universal fuzzy) constructed by Zadeh [1]. It is required to incorporate these four types fuzzy statements in the relational database query languages in order to improve the human interface. Here, any one of such enhancements in the relational database query languages is generally called a fuzzy query language (FQL in short).

Several FQL's have been investigated in some case studies [2]–[4]. Unfortunately, these studies were directed to the partial technical aspects of FQL as well as to the parts of the practical applications for particular purposes. The systematical development of fuzzy query relational database systems needs a firm theoretical basis for FQL, such as shown in the systematical development of the relational database systems based on the E. F. Codd's relational database theory.

This correspondence proposes a theoretical framework for FQL based on the relational domain calculus that incorporates all four types of the fuzzy statements exhaustively investigated by L. A. Zadeh. With this framework, the fuzzy query relational database systems such as shown in Fig. 1 could be systematically developed and further investigated for higher human-oriented interfaces.

II. Fuzzy Query Types

Zadeh [1] classified natural language imprecise statements into four types from the viewpoint how simple fuzzy statements are modified into complex fuzzy statements where a simple fuzzy statement is represented by a sentence composed of a subject and a predicate in which the predicate is a fuzzy predicate *underlined* in the following examples.

a) X is *small.*

Manuscript received September 26, 1990; revised April 30, 1991.
The author is with the Communications and Information Processing Laboratories, NTT, 1–2356 Take, Yokosuka-shi, Kanagawa-ken, 238–03 Japan.
IEEE Log Number 9102940.