A Complex Benchmark for Logic Programming and Deductive Databases, or Who Can Beat the N-Queens?

Werner Kießling

Technische Universität München, Institut für Informatik
Orleansstrasse 34, D-8000 München 80, Germany
wk@informatik.tu-muenchen.de

Abstract

The N-queens problem with its long history and inherent complexity is a challenging benchmark target. We present our solution and performance results, hoping that this will stimulate a sort of benchmark competition for tough problems.

N-Queens poetry: [Fur55]
They are queens
Reckless
Blasting forth
Insatiable
They need more
Ever more
Dimension
to conquer
When they stop
Panting
They rest
Like drowsy cows

1 Introduction

A major claim of logic programming systems (LPS) and deductive database systems (DDBS) is that they narrow down the infamous impedance mismatch of current SQL-technology, implying that complex problems can be solved better by LPS or DDBS. However, up to now the proposed benchmarks do not sufficiently cover the vast performance spectrum (for special aspects see e.g. [KRS90]). We'd like to improve this situation by posing the following advanced benchmark issue: Given a complex problem, (a) name an algorithm of your choice. (b) pick your favorite LPS or DDBS to compile, optimize and run your choice of (a).

The *N-queens problem* is to determine *all different solutions* in which N queens can be placed on an N-by-N chessboard without any queens attacking each other. According to [RZ92] this problem has a long history. It was first mentioned in the "Berliner Schachgesellschaft" of 1848 and in the sequel attracted the attention of a number of famous mathematicians, including Gauss, Lucas and Polya. Quoting [RZ92], "it is of current interest mostly as a benchmark problem for backtracking algorithms. Yet there appears to be no algorithm whose complexity is known to be better than the brute-force approach". If only a single solution is looked for, very

sophisticated algorithms are known (see [HE80], [vHD87]). For the all-solutions problem which we consider, a first non-trivial upper bound on the number Q(N) of solutions is proved in [RZ92]: Q(N) can be computed in time $O(f(N) * 8^N)$ and space $O(N^2 * 8^N)$. Thus we are faced with exponential complexity. Such programs can be expressed easily in $Datalog^{neg+func}$ ([NT89]).

2 N-Queens Solution

Our algorithm:

Let the rows and columns of an $N * N$ chessboard be numbered $0, 1, ..., N-1$, let $R_i \in \{0, \ldots, N-1\}$ denote the row number of the queen in column i, $0 \leq i \leq N-1$, and let $M \in \{2, \ldots, N\}$.

1. $safe(R_{i+1}, [R_{i+2}, \ldots, R_{i+M}], M)$ iff M queens, put on any M consecutive columns on the board starting in column $i+1$, don't attack each other.

2. For $M \geq 2$ we conclude:

   If $safe(R_{i+1}, [R_{i+2}, \ldots, R_{i+M}], M)$
   and $safe(R_i, [R_{i+1}, \ldots, R_{i+M-1}], M)$
   and the queens on positions R_i and R_{i+M} don't attack each other,
   then $safe(R_i, [R_{i+1}, \ldots, R_{i+M}], M+1)$.

In figure 1, from $safe(3, [0, 2], 3)$ and $safe(1, [3, 0], 3)$ for $i = 0$, and from the fact that the queens on columns 0 and 3 do not attack each other, we conclude $safe(1, [3, 0, 2], 4)$.

Besides, as a standard software engineering technique, a programmer might wish to step-wisely refine above program in the spirit of abstract data types as follows: Consider the case N=8. Instead of representing the queens' positions on the chessboard by a list of row numbers $[r0, \ldots, r7]$, a much more efficient representation would be as a single number in the octal system. For instance, $[1, 3, 5, 7]$ can be represented as octal number 7531. Then the list-processing function *cons*[e|l] can be re-defined in "C" as:

```
int cons(e, l)
int e, l;
{ return(e + l * 010); }
```

The list-processing functions `last` and `butlast` can be re-defined accordingly. Now our algorithm can be expressed in *LDL*-notation [NT89] as shown in figure 4 of appendix A.1.

	0	1	2	3
0			Q	
1	Q			
2				Q
3		Q		

Figure 1: Construction of safe queen positions (N = 4).

3 Compilation

At a first and even more at a second glance this complex doubly recursive program does not seem to have a chance for competitive performance. We discuss the compilation of our algorithm within the context of the commercial deductive database system DECLARE, which was under development between 1986 and 1990[1]

$Datalog^{neg+func}$ programs are processed by Declare's optimizing rule compiler ORC, which includes the usual optimization repertoire. The target code generated by the ORC is a compact expression

[1] Due to internal company policies no scientific reports were released in the past, except for a brief overview given in [KG90].

of extended relational algebra (ERA) operations, with a choice of operators like UNION, PRODUCT, DIFFERENCE, SELECT, PROJECT, various join operators JOIN, FCT_JOIN, SEMIJOIN, ANTIJOIN, and several least fixpoint operators (LFP).

The query `?n_queens(Z,8)` can be transformed into the attributed query execution plan (QEP) of figure 5. The (simplified) adornments shown for the LFP-node are the clique attribute, indicating direct recursion, and the terminate attribute equaling '(N = 8)', which may be detected by a preceding static compile-time analysis of monotonicity constraints ([BS89]). In our case the iteration can be stopped as soon as `N = 8` is reached, since attribute N is strictly monotonic in N.

For the code generation phase the library of available LFP-algorithms is searched and the most appropriate will be fetched. Due to the specific nature of the resulting fixpoint equation we can perform an additional optimization here. According to [GKB87], the symbolic differentiation for the doubly-recursive, one-dimensional clique *safe* through the semi-naive Δ-iteration becomes $Aux(safe, \Delta) = safe * \Delta \dot{+} \Delta * \Delta \dot{+} \Delta * safe$, where * denotes the proper relational algebra expression. Observing again the monotonicity in N, it follows that $safe * \Delta$ and $\Delta * safe$ both are empty. Hence we can simplify the differential expression to $Aux - special(safe, \Delta) = \Delta * \Delta$, which is incorporated into the *LPF-special operator*. The executable runtime code is shown in figure 6.

4 Benchmarks

Back in 1987 ([KW*87]) this compiled program executed on an experimental prototype of DECLARE, written in Common-Lisp, on a SUN3-like machine as shown in figure 2. The improved quadratic fixpoint iteration only computes joins for $\Delta * \Delta$ with respective cardinalities as listed in figure 3. Note that at a superficial glance these numbers are in line with the time and space bounds given by [RZ92].

N	# of solutions	seconds
6	4	0.5
8	92	7.1
9	352	29.1
10	724	127.8
11	2680	834.7

Figure 2: N-queens benchmark on a SUN3-like machine.

N	9	10	11
	56	72	90
	234	364	536
	732	1400	2468
	1614	3916	8492
	2292	7552	21362
	2038	9632	37248
	1066	7828	44118
	352	4040	34774
		724	15116
			2680

Figure 3: Resulting join cardinalities for iterative $\Delta * \Delta$ computation.

It is claimed that this is a highly efficient solution to the N-queens and it seems to be unbeaten at the time of writing.

For a first comparison let us report performance figures of some other *linearly* recursive algorithms for the N-queens problem:

The N-queens algorithm of [SS87] runs about 20 times slower with the 1991 version of LDL or Prolog ([Zan91]).
The algorithm in appendix A.2 solved the 8-queens in 8 sec., using external func-

tions, in 13 sec. using lists and running LDL on a VAX II GPS. The latter algorithm implemented in Quintus Prolog was about four times slower ([Zan91]).
The constraint logic programming system CLP(R) of [HJ*91] processed the algorithm in appendix A.3 on a Sun SparcSLC in the following amount of seconds:
N = 6: 0.9; N = 8: 21.6; N = 9: 114.2;
N = 10: 567.2; N = 11: 3187.4.

5 Summary

We hope that this study provides an incentive to generate some competition on complex-problem performance on logic programming or deductive database systems. In this sense we invite everybody to *beat up our N-queens algorithm.* To keep up with modern workstation technology, let's just put down concrete numbers for a Sun Sparc2-based benchmark, say:
$N \leq 8$ in ≤ 1 sec, N = 9 in ≤ 3 sec,
N = 10 in ≤ 13 sec, N = 11 in ≤ 84 sec,
and N beyond 11. Good luck!

References

[BS89] A. Brodsky, Y. Sagiv: *Inference of Monotonicity Constraints in Datalog Programs*, Proc. ACM PODS 1989.

[Fur55] R. de L. Furtado: *N-queens poetry, in: 'The Centre', 1955*, communicated by P. Reintjes in: Logic Programming Newsletter, Vol. 5/3. Aug. 1992, pp. 30.

[GKB87] U. Güntzer, W. Kießling, R. Bayer: *On the Evaluation of Recursion in (Deductive) Database Systems by Efficient Differential Fixpoint Iteration.* Proc. 3rd Data Eng. Conf., 1987, pp. 120 - 128.

[HES0] R. Haralick, G. Elliot: *Increasing Tree Search Efficiency for Constraint Satisfaction Problems,* Artificial Intelligence, 14, 1980, pp. 263 - 313.

[HJ*91] N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey, R. Yap: *The CLP(R) Programmer's Manual Version 1.1*, IBM Th. J. Watson Res. Center, Yorktown Heights, Nov. 1991.

[vHD87] P. van Hentenryck, M. Dincbas: *Forward Checking in Logic Programming*, Proc. 4th Int'l Conf. on Logic Programming, Melbourne, May 1987, pp. 229 - 256.

[KG90] W. Kießling, U. Güntzer: *Deduktive Datenbanksysteme auf dem Weg zur Praxis*, Informatik Forschung und Entwicklung, Springer, Vol. 5, 1990, pp. 177 - 187.

[KRS90] D. B. Kemp, K. Ramamohanarao, Z. Somogyi: *Right-left-multilinear Transformations that Maintain Context Information*, techn. rep. 90-2, Univ. of Melbourne, 1990.

[KW*87] W. Kießling, A. Wittkowski, H. Schmidt, W. Strauss, R. Azone: *DECLARE: A Deductive Database Language for Large Knowledge-Based Applications*, MAD Intelligent Systems GmbH, Munich, unpublished internal document, 1987.

[NT89] S.A. Naqvi, S. Tsur: *A Logic Language for Data and Knowledge Bases*, New York: Computer Science Press 1989.

[RZ92] I. Rivin, R. Zabih: *A dynamic programming solution to the n-queens problem*, Information Processing Letters, North-Holland, 41, 1992. pp. 253 - 256.

[SS87] L. Sterling, E. Shapiro: *The Art of Prolog: Advanced Programming Techniques*, 1987.

[Zan91] C. Zaniolo: *Private communication*, 1991.

Appendix

A1: N-queens compilation.

□ Datalog solution for N × N chessboard using external functions:

```
DB-predicates: row(0), ..., row(N-1).

import cons($E: integer, R: integer) => Z: integer
   from C external 'cons.o' as cons($E,$R,Z).
import last($R: integer) => Z: integer
   from C external 'last.o' as last($R,Z).
import butlast($R: integer) => Z: integer
   from C external 'butlast.o' as butlast($R,Z).

safe(R1,R2,2)   <- row(R1), row(R2),
                   R1 <> R2, R1 <> R2 + 1, R1 <> R2 - 1.
safe(R0,Z,N+1)  <- safe(R1,R,N), butlast(R,H), cons(R1,H,Y), N >= 2
                   safe(R0,Y,N), last(R,X),
                   R0 <> X, R0 <> X + N, R0 <> X - N,
                   cons( R1, R, Z ).

n_queens(Z,N)  <- safe(R1,R,N), cons(R1,R,Z).
```

□ Query asking for all solutions for N = 8: ?n_queens(Z,8).

Figure 4: Doubly-recursive N-queens program with external functions.

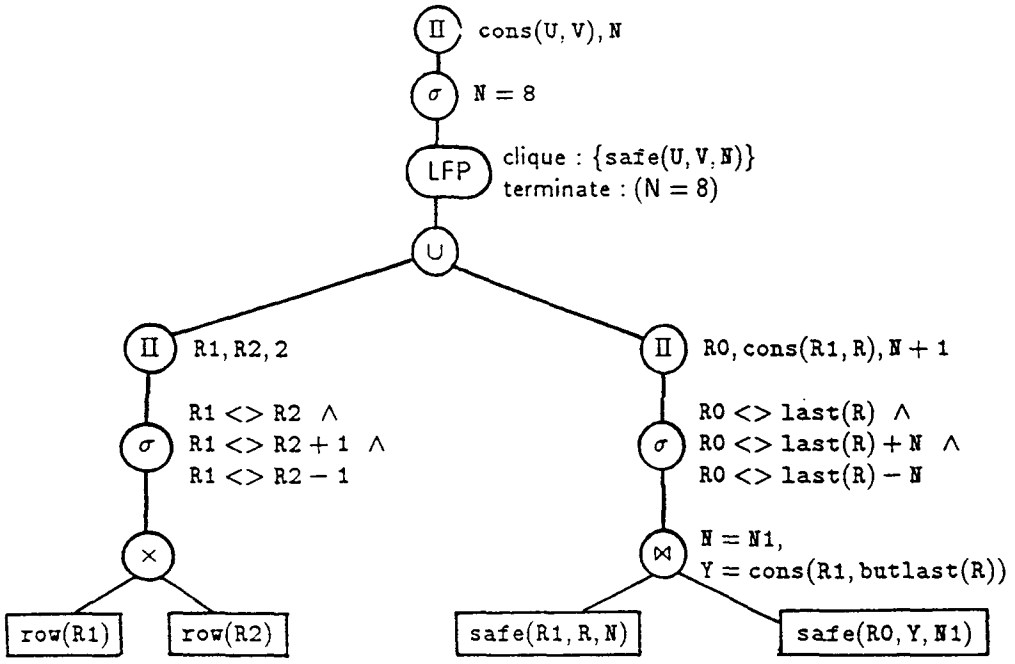

Figure 5: QEP for doubly-recursive N-queens.

```
PROJECT[cons(U,V), N]
  (LFP_special[clique: ( safe(U,V,N) ), terminate: ( N = 8 ) ]
    (UNION
      (PROJECT[R1, R2, 2]
        (SELECT[R1<>R2, R1<>R2+1, R1<>R2-1]
          (PRODUCT( row(R1), row(R2) )))) ,
      PROJECT[R0, cons(R1,R), N+1]
        (SELECT[R0 <> last(R), R0<>last(R)+N, R0<>last(R)-N]
          (FCT_JOIN[ N=N1, Y = cons(R1, butlast(R))]
            ( safe(R1,R,N), safe(R0,Y,N1)))))))
```

Figure 6: Optimized ERA-code for N-queens.

A.2: N-queens by Brijesh Agrawal (communicated by [Zan91]).

```
safeq( [Q1,Q2], 2 ) <-
          row( Q1 ), row( Q2 ), Q1 <> Q2, abs(Q1, Q2, 1).
safeq( [NewQ | Qlist],Col)  <-
          Col > 2, safeq( Qlist, Col-1 ),
          notattack( NewQ, Qlist, Col-1, Col-2 ).

notattack(NewQ, [HeadQ | Q_list], NewCol, HeadCol)  <-
          notattack(NewQ, Q_list, NewCol, HeadCol-1),
          nosweat(NewQ, HeadQ, NewCol, HeadCol).
notattack(NewQ, [Q], NewCol, HeadCol)  <-
          row(NewQ), nosweat(NewQ, Q, NewCol, HeadCol).

nosweat(NewQ, Q, NewCol, HeadCol)  <-
          NewQ <> Q, abs(NewQ, Q, Del), NewCol <> HeadCol + Del.

abs(NewQ, Q, Del)   <-
          if (NewQ > Q then Del = NewQ - Q else Del = Q - NewQ).


qform safeq(Qlist,$Size).
%qform nosweat($A,$B,$C,$D).
%qform notattack(NewQ, $Qlist, $NewCol, $HeadCol).

database({ row(integer) }).
row(0).  ...  row(7).
```

A.3: N-queens by Karlhorst Klotz, Techn. Univ. Muenchen.

```
solve( Size, Vars ) :-
        make_queen_variables( Size, Vars ), constraints( Vars ),
        make_columns( Size, Columns ),
        gen_diff_columns( Vars, Columns ).
```

```
make_queen_variables( 0, [] ).
make_queen_variables( Size, [V|Vars] ) :-
        Size > 0, make_queen_variables( Size-1, Vars ).

make_columns( 0, [] ).
make_columns( Size, [Size|Vars] ) :-
        Size > 0, make_columns( Size-1, Vars ).

constraints( [] ).
constraints( [Q|Queens] ) :-
        constraint( 1, Q, Queens ), constraints( Queens ).

constraint( Diff, Q, [] ).
constraint( Diff, Q, [QQ|Queens] ) :-
        not_equal( Q , QQ ), not_equal( Q + Diff , QQ ),
        not_equal( Q , QQ + Diff ), constraint( Diff+1, Q, Queens ).

not_equal(P,Q) :- abs(P-Q) >= 1.

gen_diff_columns([], _).
gen_diff_columns([H | T], L) :-
        select(H, L, L2), gen_diff_columns(T, L2).
select(H, [H | T], T).
select(H, [H2 | T], [H2 | T2]) :-
        select(H, T, T2).
```