BREADTH-FIRST SEARCH IN THE EIGHT QUEENS PROBLEM

C K Yuen and M D Feng, DISCS, NUS, Kent Ridge, Singapore 0511

Abstract

The Eight Queens Problem is used to illustrate some different approaches to recursive programming and parallel processing.

The Eight Queens Problem is a familiar example in programming and algorithm textbooks. Usually, the given solution applies depth-first search with backtracking, and stops after finding one solution. When we attempted to reformulate the problem by applying breadth-first search, in order to find all solutions without backtracking, we were only able to find one published example, written in a Miranda-like functional language[1].

In the present article, we show the Lisp solution for the problem, and discuss some programming issues arising from the effort. Usually, the backtracking program is used as an illustration of recursion: A tree is being searched, and if one branch leads to an unsatisfactory queen placement, then we move back to a higher node and try another branch. The breadth-first algorithm is iterative rather than recursive: Nodes at the same depth are examined one by one, and those found to be satisfactory are extended to the next level, while unsatisfactory ones are removed. The same node by node search is then performed at the next level. It is also clear that a suitable data structure, with dynamic storage management to extend wanted nodes and remove unwanted ones, must be available to support the iteration, whereas in recursion dynamic storage is achieved through the varying depth of the stack with procedure entries and exits. However, in our Lisp program we will use recursion to perform iteration.

It is known that there are altogether 92 solutions, and one example is

```
Q - - - - - - -
- - - - Q - - -
- - - - - - - Q
- - - - - Q - -
- - Q - - - - -
- - - - - - Q -
- Q - - - - - -
- - - Q - - - -
```

To write a Lisp program, we must first define a list structure to represent the queen placements. Since there can be only one queen per column, we need just 8 numbers, each showing the row number of the queen in a column, e.g., the above position is represented by the list (1 7 5 8 2 4 6 3). At the start of the program, we take a list with just one number, (1), representing a board with a queen in column 1, row 1; we can add another queen to rows 3 to 8 of column 2, producing the placement lists (1 3), (1 4), ... (1 8). Then for each of these lists, we try to add a queen to column 3, etc. The same is also done with the initial placement lists (2), (3),...,(8).

Thus, we start with a solution list of eight initial placement lists ((1) (2) ... (8)). The function Init recurses seven times to add new queens to columns K, K = 2 to 8. For each member of the solution list, Extend recurses 8 times to try to add a queen to row L of column K, L = 1 to 8, while AddL looks at the queens in the previous columns I, I = 1 to K-1, to see if any two queens check each other. If no check is detected, it places L in position K of the list Row, and the extended list is added to the new solution list for future processing. If an existing queen is found to be on the same row or diagonal, the new solution list is returned unchanged. The row numbers J of previous queens are taken from the list Row, which is itself taken from Soln, the list of all solutions still under consideration. We have deliberately written the code in a very terse way - it could be made a little easier to read by judiciously using more LETblocks. Also, no effort has been made to take advantage of board symmetry or minimize space usage. Dynamic scoping is assumed so that functions need not be shown as nested, though in actual fact the default of our Lisp compiler is lexical scoping.

```
(DEFUN AddL (I J Tail)
      (COND ((= I K) (CONS (CONCAT Row (LIST L)) NewSoln))
            ((= J L) NewSoln)
            ((= (- K I) (ABS (- L J))) NewSoln)
            (T (AddL (+ I 1) (CAR Tail) (CDR Tail)))))

(DEFUN Extend (Soln Row L NewSoln)
      (COND ((> L 8)
             (COND ((NULL Soln) NewSoln)
                   (T (Extend (CDR Soln) (CAR Soln) 1 NewSoln))))
            (T (Extend Soln Row (+ 1 L) (AddL 1 (CAR Row) (CDR Row))))))

(DEFUN Init (Soln K)
      (COND ((> K 8) Soln)
            (T (Init (Extend (CDR Soln) (CAR Soln) 1 NIL) (+ 1 K)))))

(Init '((1)(2)(3)(4)(5)(6)(7)(8)) 2)
```

During the execution of the program, the size of the solution list first increases, from 8 to 42 after processing K=3, then to approximately 150,350,550 for K=4,5,6, and thereafter decreases rapidly as placements are found to lead to check.

Let us compare this with the solution shown in [1], slightly changed here to facilitate comparison:

```
NoCheck Row L = ALL [(J<>L) AND ((K-I)<>ABS(L-J)) | (I,J) <- ZIP
                                                    ([1..#Row], Row)]
                  WHERE K = #Row+1

Queens 1      = [[1][2][3][4][5][6][7][8]]

Queens (K+1)  = [Row++[L] | Row <- Queens K; L <- [1..8]; NoCheck Row L]
```

This says, produce each element of Queens (K+1) by concatenating L, an integer between 1 and 8, to sequence Row, which is taken from the result of Queens K, if NoCheck Row L is true, which requires us to zip through all pairs of values I,J, with I=1 to #Row (length of Row) and J taken from Row, to discover whether two queens at I,J and K,L check each other.

There is little doubt that the Miranda-like solution is easier to understand: the idea of the algorithm is more directly and clearly reflected, whereas in the Lisp program, algorithm knowledge has been embedded in the Lisp syntax, and it is necessary to manage an interplay of Lisp language, algorithm structure and data structure ideas as one reads the program. In our own experience, the number of students and colleagues who can quickly understand the Lisp program is very small. Often, the comment is that Lisp programs are as hard to read as assembly code.

But there is the reverse side of this: In the Miranda-like program, the L is first checked against the content of sequence Row in NoCheck; then, in a separate operation inside Queens, [L] is concatenated to Row. In the Lisp program, checking and placement are integrated: we go through Row cell by cell performing the check with L, and if we reach the end of Row, L is left there as the new tail element and the new Row is CONSed before NewSoln. This integration of processing, together with the lower level of code interpretation, leads to better efficiency. Further, the Lisp program gives an indication of the execution activities during the program's run, allowing us to form an estimate of its complexity, whereas the Miranda-like program's behaviour is dependent upon the actual workings of the compiler and is rather difficult to discern from the program. Again, we have the analogy of Lisp with assembly code.

In fact, we can produce a closer Lisp equivalent to the Miranda-like program as follows:

```
(DEFUN NoCheck (I J Row)
       (COND ((= I K) T)
             ((= J L) NIL)
             ((= (- K I) (ABS (- L J))) NIL)
             (T (NoCheck (+ I 1) (CAR Row) (CDR Row)))))

(DEFUN Extend (Soln Row L)
       (COND ((> L 8)
              (COND ((NULL Soln) NIL)
                    (T (Extend (CDR Soln) (CAR Soln) 1))))
             ((NoCheck 1 (CAR Row) (CDR Row))
              (CONS (CONCAT Row (LIST L)) (Extend Soln Row (+ 1 L))))
             (T (Extend Soln Row (+ 1 L)))))

(DEFUN Init (Soln K)
       (COND ((> K 8) Soln)
             (T (Init (Extend (CDR Soln) (CAR Soln) K 1) (+ 1 K)))))

(Init '((1)(2)(3)(4)(5)(6)(7)(8)) 2)
```

But this runs a little slower than the first Lisp program. At the same time, it is generally considered to be somewhat easier to understand, because of the separation of "finding out what to do" and "doing it": NoCheck says it is OK to add a queen, and Extend does it, but there may be an efficiency penalty to be paid for this kind of separation.

CONCAT is usually a slow operation in Lisp, requiring a fresh traversal of Row after NoCheck or AddL has already traversed it, and every cell of Row must be copied before L is attached to the end: If L is simply attached to the end of Row, we would be changing Row, whereas CONCAT is meant to produce no side effect. If the underlined part is replaced by (CONS L Row), the result continues to be correct, but each placement list would show the rightmost column position at the head instead of the tail. It so happens

that for this particular problem, the reversal does not matter because of the symmetry of the solutions, but of course this would not be generally true - for some algorithms one could keep the result in reverse order and turn it around at the end, while in others this cannot be done because a fixed head-to-tail search sequence is involved. Note that because Row represents the columns in reverse, the index I must start at K-1 and go down to 0, and the other underlined parts of the code must also be changed when CONCAT is replaced by CONS. With these additional changes (see below), the new version executes faster than the previous one and also takes up less space because Row may be re-used to construct several extensions and its space is shared.

It turns out to be quite troublesome to reprogram the problem in Pascal, because of the difficulty of managing the dynamic storage requirement. After experimenting with several list structures using Pascal records, we settled on the use of a 600 x 8 matrix as a circular buffer to store both existing and new board placements. The program took 20 seconds to run on a 286-based UCSD Pascal system, compared to 0.35 second on a INMOS 805 Transputer for the fastest Lisp program.

Another issue we wish to raise concerns parallel processing. Instead of starting one task processing a solution list of 8 placements, we can easily have 8 tasks, each with a one-placement list, and concatenate their results:

```
...
(CONCAT (EXEC Init '((1)) 2) (EXEC Init '((2)) 2) (EXEC Init '((3)) 2)
        (EXEC Init '((4)) 2) (EXEC Init '((5)) 2) (EXEC Init '((6)) 2)
        (EXEC Init '((7)) 2) (Init '((8)) 2))
```

where we have rewritten the one-line main part of the program, using the BaLinda Lisp[2] parallel tasking prefix EXEC.

It is also possible to have a "pipelined" parallel version, with each task responsible for adding queens to one column. The column 2 task executes first, and as soon as it produces one result, the column 3 task starts working, etc. Each iteration of Init, except the last, produces a parallel task. To achieve proper communication, tuples are used to control access to the solution lists. That is, task K obtains Row from tuple K and puts the extended Row into tuple K+1. It is possible for the solution list to be empty not because there no further work to do but because the previous task is too slow. To prevent tuple access in such a situation, there is a Boolean field in the tuple that goes to NIL when the last element of Soln has been taken out. An empty list is appended to the end of the initial Soln. Hence, if the element taken from Soln turns out to be the empty list, the task knows it has completed its work, and so it adds an empty list to the end of Soln in tuple K+1 to mark the end of work for task K+1.

```
(DEFUN AddL (I J Tail)
      (COND ((= I 0) (LET ((New NIL))
                          (IN KK - ? New)
                          (OUT KK T (CONS (CONS L Row) New))))
            ((= J L) NIL)
            ((= (- K I) (ABS (- L J))) NIL)
            (T (AddL (- I 1) (CAR Tail) (CDR Tail)))))

(DEFUN Extend (L Row)
      (COND ((> L 8)
```

```
            (LET ((New NIL)
                  (KK (+ 1 K)))
                 (IN K T ? New)
                 (COND ((NULL (CAR New))
                        (IN KK - ? New)
                        (OUT KK T (CONCAT New (LIST ()))))
                       (T (OUT K (NOT (NULL (CDR New))) (CDR New))
                          (Extend 1 (CAR New)))))
            (T (AddL (- K 1) (CAR Row) (CDR Row))
               (Extend (+ 1 L) Row)))))

(DEFUN Init (K)
       (LET ((KK (+ 1 K))
             (Soln NIL))
            (OUT KK NIL NIL)
            (IN K T ? Soln)
            (OUT K (NOT (NULL (CDR Soln))) (CDR Soln))
            (COND ((< K 8) (EXEC Init KK)
                           (Extend 1 (CAR Soln)))
                  ((= K 8) (Extend 1 (CAR Soln))))))

(OUT 2 T '((1)(2)(3)(4)(5)(6)(7)(8)()))
(Init 2)
(IN 9 T ? Result)
```

Note that a BaLinda Lisp form returns only when all the tasks within it have ended. Hence, when (Init 2) returns, the final result is already available. The reader might like to compare the two programs with [3].

While a depth-first program can also be parallelized by having tasks go down different branches in parallel, the task partitioning is usually not as tidy. The important difference lies in the use of a recursive program structure, into which we can fit parallel tasking only when the program reaches a new node, and the amount of parallel tasking is limited by the number of branches at that node. In our example we have an iterative program working with a recursive data structure, which exists separately from the program structure, and parallelism is introduced either by running the same task on several partitions of the structure (first case), or by partitioning the iterations and then organizing the data structure to achieve the required inter task information transfers (second case). In other words, parallelism is added at a "global" level, instead at "local" levels within a recursive program. We feel this is an interesting theoretical issue worthy of further detailed study.

References

[1] R Bird and P Wadler, Introduction to Functional Programming, Prentice-Hall, 1988, p. 161.

[2] C K Yuen, M D Feng and J J Yee, "BaLinda suite of languages and implementations", Journal of Systems and Software, 1995. (to appear)

[3] W C Athas and C L Seitz, "Multicomputers: message-passing concurrent computers", IEEE Computer, vol. 21, no. 8, p. 20, August 1988.