Aleksnadra Shanina

Ilya Dyskin

CSC 173

**N Queens Scheme Project**

**Backtracking**

This is a greedy algorithm to solve the N Queens problem that works by initializing an empty board at first that later (using tail recursion) gets filled with queens column by column. The queens are placed in a row that does not have conflicts with the filled out part of the board. If there is not a single row in a current column that doesn’t cause conflicts, the algorithm backtracks (hence the name) to a previous column and tries selecting a different queen placement. The process continues until there is not a single empty column. We used CB’s pseudocode (which was borrowed from Russel and Norvig’s textbook) to implement this algorithm in Scheme.

**Pseudocode:**

function RECUR-BACKTRACK(state)

% returns solution or failure

local row, col; % local row, column index variables

if goal(state) then return(state);

endif;

col <- SELECT-UNASSIGNED-COLUMN(state);

foreach row do

if a queen on row causes no attacks, then

state[col] <- row; % update state with queen on that row, col.

result <- RECUR-BACKTRACK(state);

if result != failure then return(result); %success!

state[col] <- -1; % or whatever flag means unassigned;

% failure-- try next row

endif;

endif;

endfor;

return(failure); % can't place queen in this column in current state

endfunc;

**Functions**

Our implementation of backtracking consists of two functions. The front-end function is executed by a user with a size argument, initializes a board vector with a given size, passed it to the back-end function that does the actually queen placements, receives filled vector from it, prints it out in a tabular form with the number steps. The back-end function was written based on the previously mentioned pseudocode in order to allow tail recursion. The main/front-end function looks as follows:

(define backtracking

(lambda (num)

(let ((vec (make-vector num -1)))

(back-internal vec 0)

;(print-vector vec)

(printf "steps: ~a\n" counter)

(set! counter 0))))

**Sample Output**

> (nq-bt 10)

1 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 1 0 0

0 1 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 1 0

0 0 0 0 0 1 0 0 0 0

0 0 1 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 1

0 0 0 1 0 0 0 0 0 0

0 0 0 0 0 0 1 0 0 0

0 0 0 0 1 0 0 0 0 0

194

> (nq-bt 25)

1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0

0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0

0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0

97341

**Testing Time and Number of Queen Placements**

|  |  |  |
| --- | --- | --- |
| Backtracking | | |
| Size | Steps | Real Time (ms) |
| 5 | 5 | 1 |
| 10 | 194 | 1 |
| 15 | 2703 | 25 |
| 20 | 399250 | 5451 |
| 25 | 97341 | 1664 |

**Hill Climbing - Min Conflicts**

A different approach to solving the N Queens problem was using “Min Conflicts”. This approach is known to be much more efficient than the tedious backtracking method. Min Conflicts works by randomly placing the queens on a board, then finding a random column with a conflict and replacing it in a row with the minimum conflicts for that column. We used CB’s (which was borrowed from Russel and Norvig’s textbook) pseudocode and only changed the goal of the function to generate an answer and return a correct board with the number of times it took to move the queens, instead of trying to check if it’s possible to get to a correct board in a given number of steps:

function MIN-CONFLICTS-N-QUEENS(n, maxsteps)

state = initialize(n);

for i = 1 to maxsteps do

if goal(state) then return(state);

endif

col <- a random column that has conflicts;

row <- the row in col where a queen causes smallest number of conflicts;

state[col,0] <- row;

end-for

return(failure)

**Methods**

We approached the implementation of this function by breaking it down into smaller functions. Thus, we have made separate functions for creating a board of a given size with queens place diagonally and one away from each other, checking whether the current state is a goal state (i.e. has no conflicts), updating a vector which marks conflicting columns and selects a random column from the vector, and making a vector containing the amount of conflicts each queen would have in each row of a given random column. Our main function that calls all those methods then looks like this:

(define (min\_conflicts n) ;x for col

(let ((col 0)

(row 0))

(let loop ((state (initialize n)))

(if (goal state) ;if the current state equals its final state (no attacking queens) then we are done

(printf "~s \n ~s \n" state min\_con\_steps) ;otherwise...

(begin (printf "------current state: ~a -------- \n" state)

(set! min\_con\_steps (+ min\_con\_steps 1)) ;updating the number of steps it takes to rearrange this board

(update\_c\_columns state) ;updating the current conflicting columns in the state

(set! col (choose\_col n))

(set! row (choose\_row state n col))

(printf "chose col: ~a and row: ~a \n" col row)

(vector-set! state col row)

(loop state))))))

**Examples of The Results**

The simplest case would be a board size of 4:

> (nq-mc 4)

0 1 0 0

0 0 0 1

1 0 0 0

0 0 1 0

14

You can see that a board of size 5 takes 0 steps to be completed (see the Discussion for an explanation)

> (nq-mc 5)

1 0 0 0 0

0 0 0 1 0

0 1 0 0 0

0 0 0 0 1

0 0 1 0 0

0

**Testing Time and Number of Queen Placements**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Minimum Conflicts | | | | | | | |  |
| Size | Steps | | | Average Steps | Real Time (ms) | | | Average Time |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 85 | 72 | 39 | 196 | 13 | 5 | 3 | 7 |
| 15 | 66 | 261 | 149 | 476 | 10 | 40 | 20 | 23.33333333 |
| 20 | 32 | 58 | 3 | 93 | 9 | 15 | 1 | 8.333333333 |
| 25 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0.333333333 |
| 50 | 7 | 2 | 2 | 11 | 11 | 5 | 3 | 6.333333333 |
| 100 | 354 | 106 | 239 | 699 | 2032 | 581 | 1318 | 1310.333333 |

**Discussion**

Firstly, we made a decision to place the queens in a manner than minimizes number of collisions for odd numbered boards. For example, a board of size 5 would already be placed in a goal state. The board of even size would be then placed this way:

1 0 0 0 0

0 0 1 0 0

0 0 0 0 1

0 1 0 0 0

0 0 0 1 0

(size 5)

The board of even size would be then placed this way:

1 0 0 0

0 0 1 0

0 1 0 0

0 0 0 1

(size 4)

It would be interesting to test whether or not there is a difference in completion time between this placement and for example just any random row.

An important obstacle that we had to overcome when implementing this pseudocode is that for certain board sizes an infinite loop can occur when there are only 2 columns with conflicts left, and the rows keep switching between one another. We decided to solve this problem by setting a rule that there is a 1 in 10 chance that a random row will be chosen as opposed to a minimum conflict one. This however, did increase the number of queen movements for all the cases.

**References:**

Artificial Intelligence. A Modern Approach by Russel and Norvig

http://www.cs.hut.fi/Studies/T-93.210/schemetutorial/node8.html#SECTION00810000000000000000

<http://docs.racket-lang.org/reference/vectors.html>