# Chapter 6: Search Strategies (N-Queens)

Helmut Simonis

Cork Constraint Computation Centre
Computer Science Department
University College Cork
Ireland

ECLiPSe ELearning Overview





## Licence

# Outline

1. Problem
2. Program
3. Naive Search
4. Improvements





# What we want to introduce

- Importance of search strategy, constraints alone are not enough
- Dynamic variable ordering exploits information from propagation
- Variable and value choice
- Hard to find strategy which works all the time
- `search` builtin, flexible search abstraction
- Different way of improving stability of search routine

# Example Problem

- N-Queens puzzle
- Rather weak constraint propagation
- Many solutions, limited number of symmetries
- Easy to scale problem size

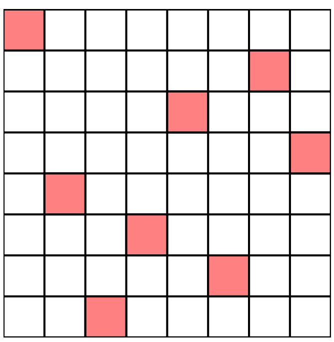




# Problem Definition

**8-Queens**

Place 8 queens on an $8 \times 8$ chessboard so that no queen attacks another. A queen attacks all cells in horizontal, vertical and diagonal direction. Generalizes to boards of size $N \times N$.



Solution for board size $8 \times 8$

# A Bit of History

- This is a rather old puzzle
- Dudeney (1917) cites Nauck (1850) as source
- Certain solutions for all sizes can be constructed, this is not a hard problem
- Long history in AI and CP papers
- Important: Haralick and Elliot (1980) describing the first-fail principle





# Basic Model

- Cell based Model
  - A 0/1 variable for each cell to say if it is occupied or not
  - Constraints on rows, columns and diagonals to enforce no-attack
  - $N^2$ variables, $6N - 2$ constraints
- Column (Row) based Model
  - A 1..N variable for each column, stating position of queen in the column
  - Based on observation that each column must contain exactly one queen
  - $N$ variables, $N^2/2$ binary constraints

# Model

$$\text{assign} \quad [X_1, X_2, ...X_N]$$

s.t.

$$\forall 1 \leq i \leq N: \quad X_i \in 1..N$$
$$\forall 1 \leq i < j \leq N: \quad X_i \neq X_j$$
$$\forall 1 \leq i < j \leq N: \quad X_i \neq X_j + i - j$$
$$\forall 1 \leq i < j \leq N: \quad X_i \neq X_j + j - i$$





# Main Program (Array Version)

```
:-module(array).
:-export(top/0).
:-lib(ic).

top:-
    nqueen(8,Array), writeln(Array).

nqueen(N,Array):-
    dim(Array,[N]),
    Array[1..N] :: 1..N,
    alldifferent(Array[1..N]),
    noattack(Array,N),
    labeling(Array[1..N]).
```

# Generating binary constraints

```
noattack(Array,N):-
    (for(I,1,N-1),
     param(Array,N) do
        (for(J,I+1,N),
         param(Array,I) do
            subscript(Array,[I],Xi),
            subscript(Array,[J],Xj),
            D is I-J,
            Xi #\= Xj+D,
            Xj #\= Xi+D
        )
    ).
```





# Main Program (List Version)

```
:-module(nqueen).
:-export(top/0).
:-lib(ic).

top:-
    nqueen(8,L), writeln(L).

nqueen(N,L):-
    length(L,N),
    L :: 1..N,
    alldifferent(L),
    noattack(L),
    labeling(L).
```

# Generating binary constraints

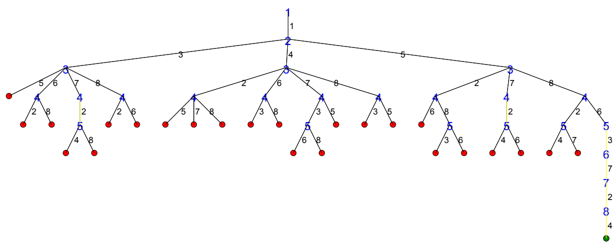
```
noattack([]).
noattack([H|T]):-
    noattack1(H,T,1),
    noattack(T).

noattack1(_,[],_).
noattack1(X,[Y|R],N):-
    X #\= Y+N,
    Y #\= X+N,
    N1 is N+1,
    noattack1(X,R,N1).
```



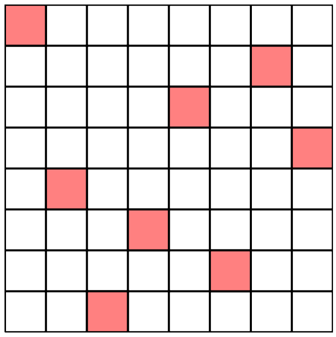


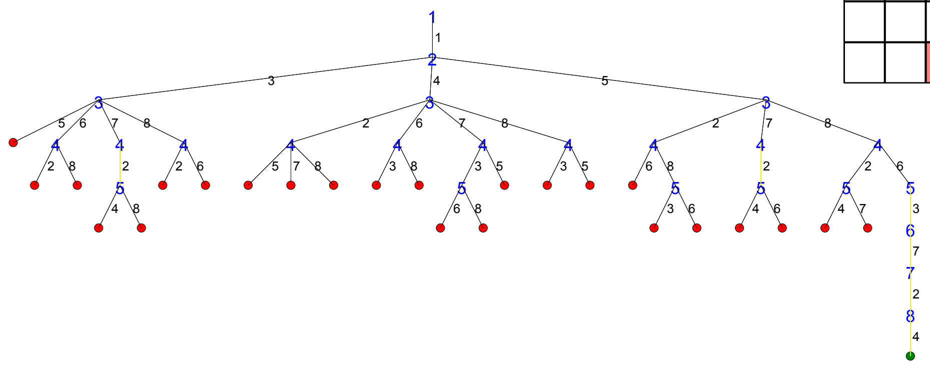


# Default Strategy



# First Solution







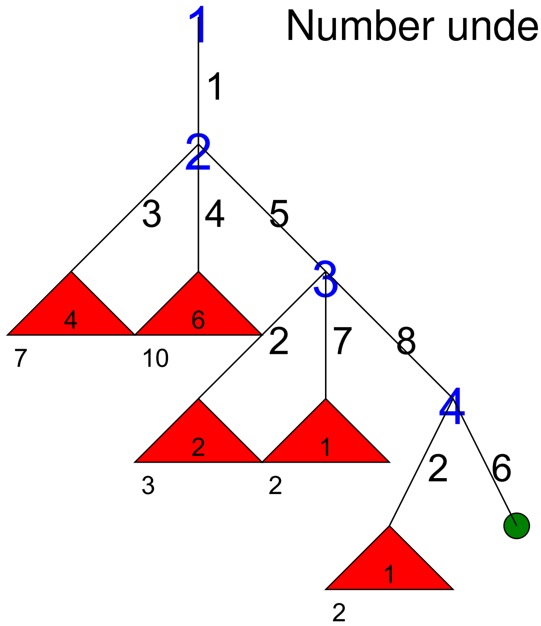


# Observations

- Even for small problem size, tree can become large
- Not interested in all details
- Ignore all automatically fixed variables
- For more compact representation abstract failed sub-trees

# Compact Representation

Number inside triangle: Number of choices
Number under triangle: Number of failures



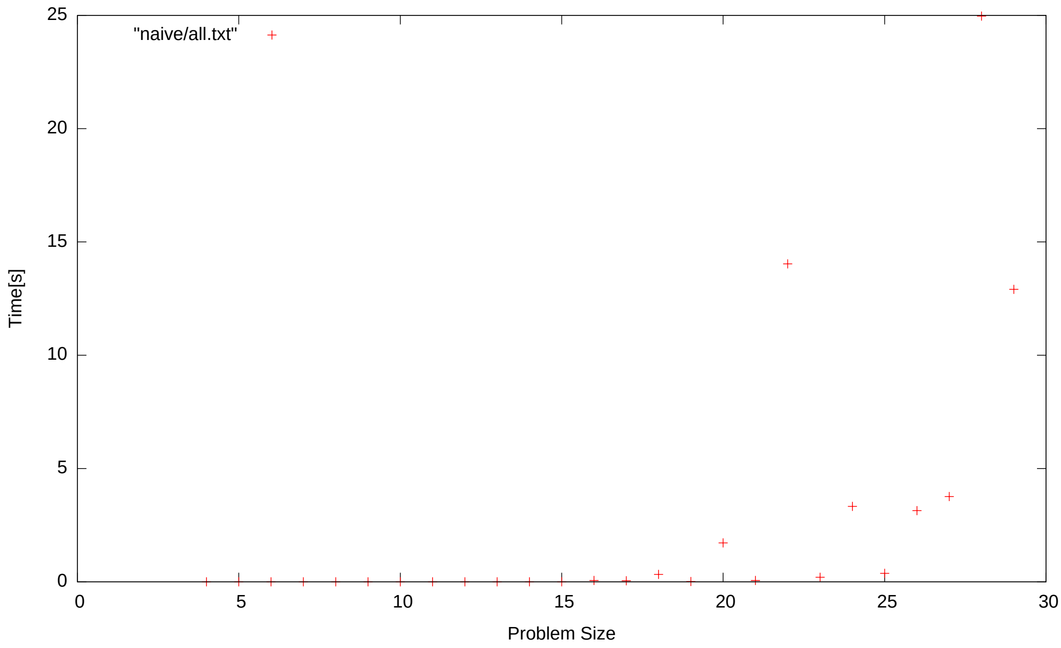





# Exploring other board sizes

- How stable is the model?
- Try all sizes from 4 to 100
- Timeout of 100 seconds

# Naive Stategy, Problem Sizes 4-100








# Observations

- Time very reasonable up to size 20
- Sizes 20-30 times very variable
- Not just linked to problem size
- No size greater than 30 solved within timeout

# Possible Improvements

- Better constraint reasoning
  - Remodelling problem with 3 `alldifferent` constraints
  - Global reasoning as described before
  - Not explored here
- Better control of search
  - Static vs. dynamic variable ordering
  - Better value choice
  - Not using complete depth-first chronological backtracking





# Static vs. Dynamic Variable Ordering

- Heuristic Static Ordering
  - Sort variables before search based on heuristic
  - Most important decisions
  - Smallest initial domain
- Dynamic variable ordering
  - Use information from constraint propagation
  - Different orders in different parts of search tree
  - Use all information available

# First Fail strategy

- Dynamic variable ordering
- At each step, select variable with smallest domain
- Idea: If there is a solution, better chance of finding it
- Idea: If there is no solution, smaller number of alternatives
- Needs tie-breaking method





# Caveat

- First fail in many constraint systems have slightly different tie breakers
- Hard to compare result across platforms
- Best to compare search trees, i.e. variable choices in all branches of tree

# Modification of Program

```
:-module(nqueen).
:-export(top/0).
:-lib(ic).

top:-
    nqueen(8,L), writeln(L).

nqueen(N,L):-
    length(L,N),
    L :: 1..N,
    alldifferent(L),
    noattack(L),
    search(L,0,first_fail,indomain,complete,[]).
```





# The `search` Predicate

- Packaged search library in ic constraint solver
- Provides many different alternative search methods
- Just select a combination of keywords
- Extensible by user

## `search` Parameters

`search(L,0,first_fail,indomain,complete,[])`

1. List of variables (or terms, covered later)
2. 0 for list of variables
3. Variable choice, e.g. `first_fail`, `input_order`
4. Value choice, e.g. `indomain`
5. Tree search method, e.g. `complete`
6. Optional argument (or empty) list





## Variable Choice

- Determines the order in which variables are assigned
- `input_order` assign variables in static order given
- `first_fail` select variable with smallest domain first
- `most_constrained` like first_fail, tie break based on number of constraints in which variable occurs
- Others, including programmed selection

# Value Choice

- Determines the order in which values are tested for selected variables
- `indomain` Start with smallest value, on backtracking try next larger value
- `indomain_max` Start with largest value
- `indomain_middle` Start with value closest to middle of domain
- `indomain_random` Choose values in random order

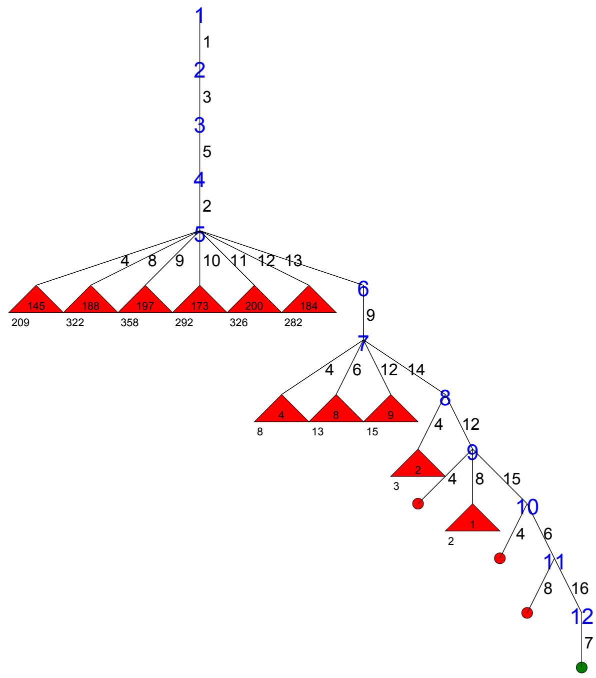


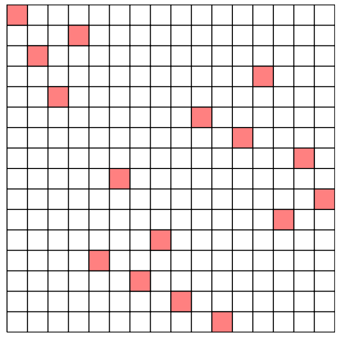


# Comparison

- Board size 16x16
- Naive (Input Order) Strategy
- First Fail variable selection

# Naive (Input Order) Strategy (Size 16)



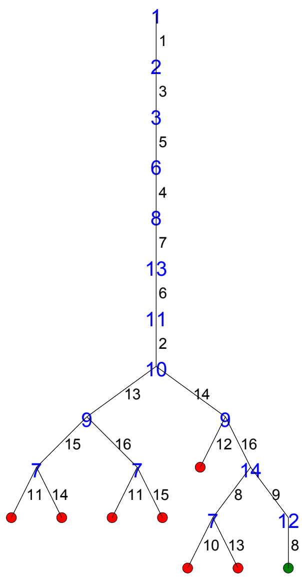



Cork Constraint Computation Centre





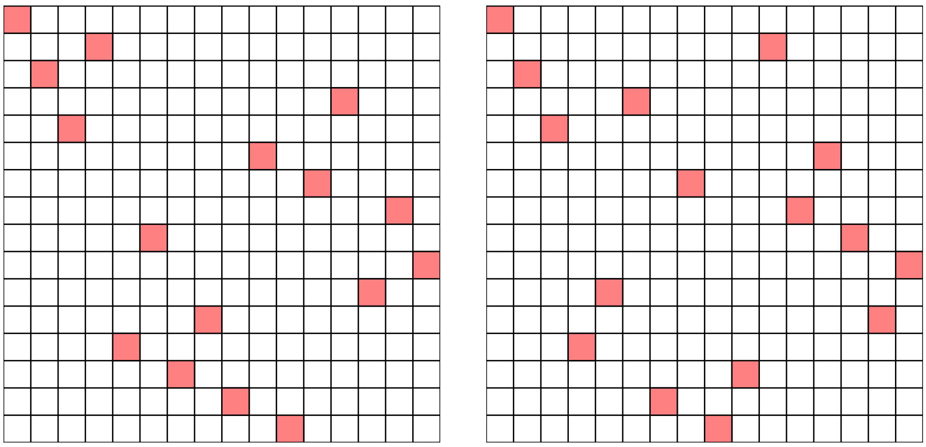
# FirstFail Strategy (Size 16)




Cork Constraint Computation Centre

# Comparing Solutions

Naive First Fail

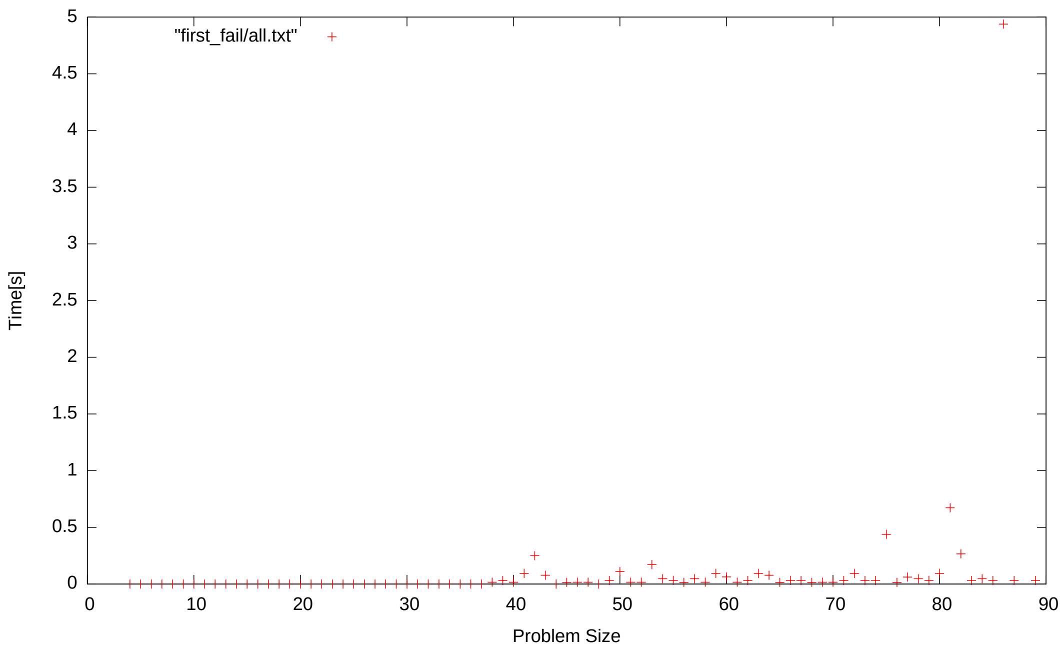


Solutions are different!





# FirstFail, Problem Sizes 4-100



# Observations

- This is much better
- But some sizes are much harder
- Timeout for sizes 88, 91, 93, 97, 98, 99





# Can we do better?

- Improved initial ordering
  - Queens on edges of board are easier to assign
  - Do hard assignment first, keep simple choices for later
  - Begin assignment in middle of board
- Matching value choice
  - Values in the middle of board have higher impact
  - Assign these early at top of search tree
  - Use `indomain_middle` for this

# Modified Program

```
:-module(nqueen).
:-export(top/0).
:-lib(ic).
top:-
    nqueen(16,L),writeln(L).

nqueen(N,L):-
    length(L,N),
    L :: 1..N,
    alldifferent(L),
    noattack(L),
```

**reorder(L,R),**
**search(R,0,first_fail,indomain_middle,complete,[]).**

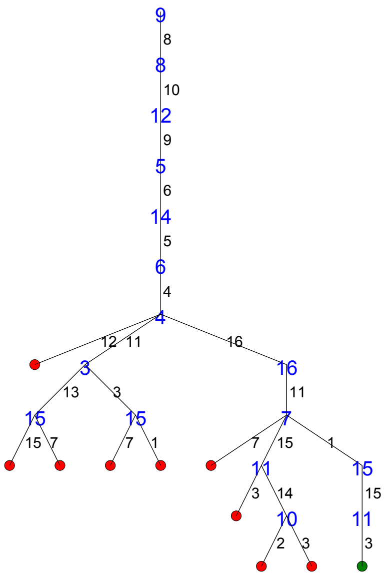




# Reordering Variable List

```
reorder(L,L1):-
    halve(L,L,[],Front,Tail),
    combine(Front,Tail,L1).

halve([],Tail,Front,Front,Tail).
halve([_],Tail,Front,Front,Tail).
halve([_,_|R],[F|T],Front,Fend,Tail):-
    halve(R,T,[F|Front],Fend,Tail).

combine(C,[],C):-!.
combine([],C,C).
combine([A|A1],[B|B1],[B,A|C1]):-
    combine(A1,B1,C1).
```

# Start from Middle (Size 16)

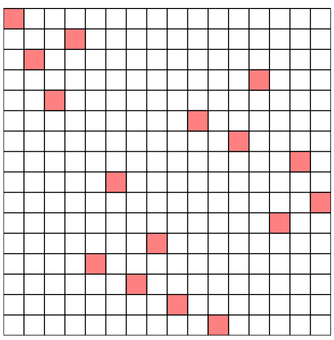




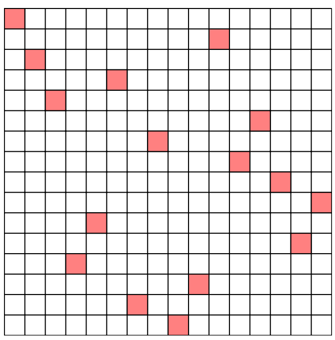



# Comparing Solutions

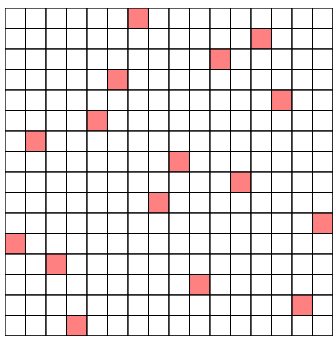
Naive



First Fail

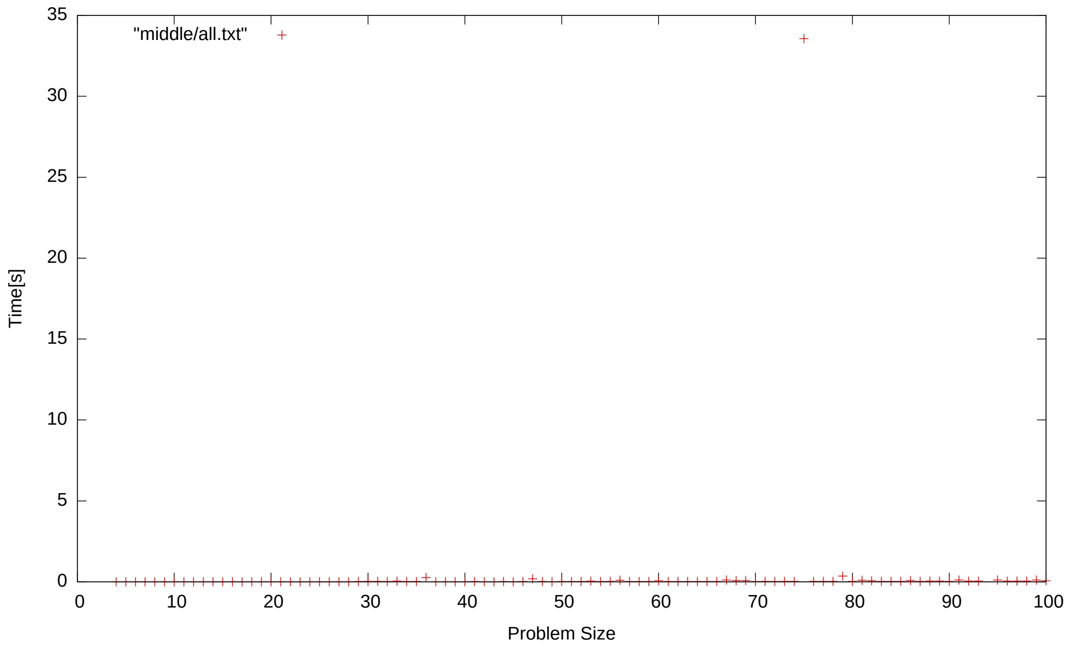


Middle



Again, solutions are different!

# Middle, Problem Sizes 4-100








# Observations

- Not always better than first fail
- For size 16, trees are similar size
- Timeout only for size 94
- But still, one strategy does not work for all problem sizes
- There are ways to resolve this!

# Approach 1: Heuristic Portfolios

- Try multiple strategies for the same problem
- With multi-core CPUs, run them in parallel
- Only one needs to be successful for each problem





# Approach 2: Restart with Randomization

- Only spend limited number of backtracks for a search attempt
- When this limit is exceeded, restart at beginning
- Requires randomization to explore new search branch
- Randomize variable choice by random tie break
- Randomize value choice by shuffling values
- Needs strategy when to restart

# Approach 3: Partial Search

- Abandon depth-first, chronological backtracking
- Don't get locked into a failed sub-tree
- A wrong decision at a level is not detected, and we have to explore the complete subtree below to undo that wrong choice
- Explore more of the search tree
- Spend time in promising parts of tree





# Example: Credit Search

- Explore top of tree completely, based on credit
- Start with fixed amount of credit
- Each node consumes one credit unit
- Split remaining credit amongst children
- When credit runs out, start bounded backtrack search
- Each branch can use only $K$ backtracks
- If this limit is exceeded, jump to unexplored top of tree

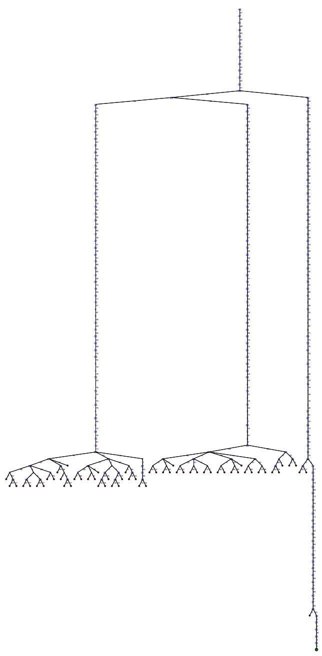
# Credit based search

```
:-module(nqueen).
:-export(top/0).
:-lib(ic).
top:-
    nqueen(8,L),writeln(L).

nqueen(N,L):-
    length(L,N),
    L :: 1..N,
    alldifferent(L),
    noattack(L),
    reorder(L,R),
    search(R,0,first_fail,indomain_middle, credit(N,5), [])
```

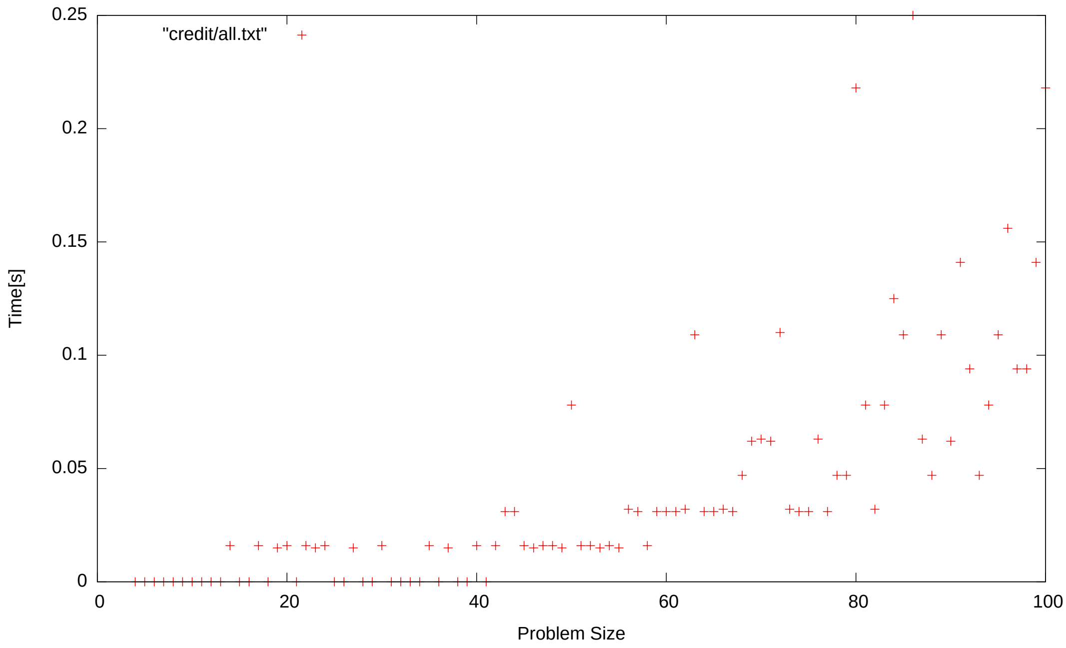




# Credit, Search Tree Problem Size 94



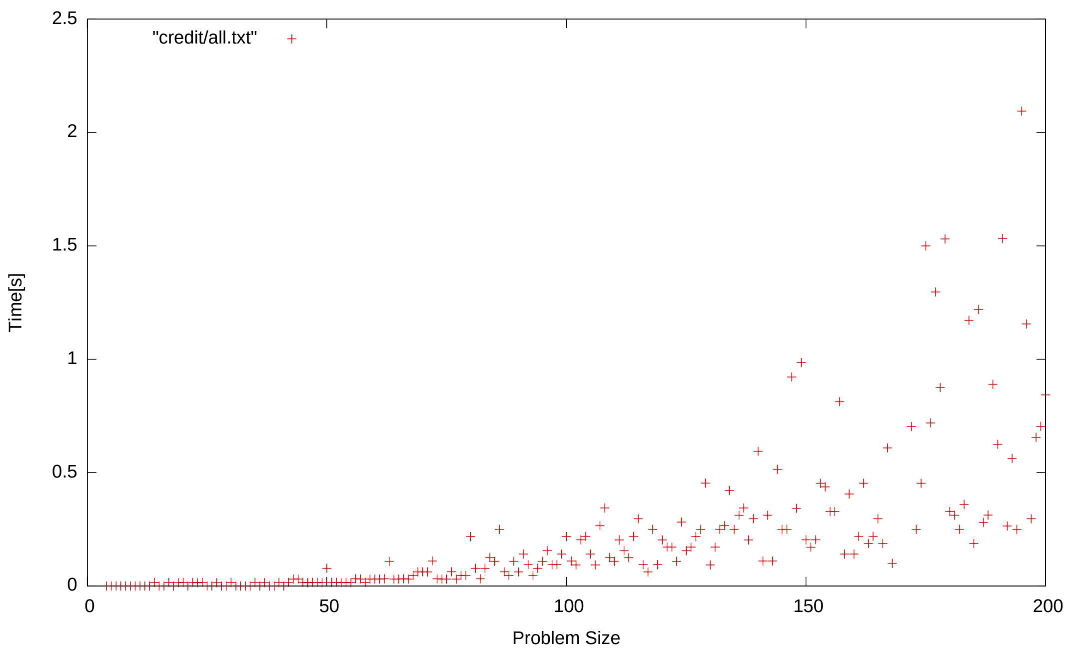
# Credit, Problem Sizes 4-100








# Credit, Problem Sizes 4-200



# Conclusions

- Choice of search can have huge impact on performance
- Dynamic variable selection can lead to large reduction of search space
- `search` builtin provides useful abstraction of search functionality
- Depth-first chronologicial backtracking not always best choice





# Outlook

- Finite domain with good search reasonable for board sizes up to 1000
- Limitation is memory, not execution time
- Memory requirement quadratic as domain changes must be trailed
- Better results possible for repair based methods
- N-Queens not a hard problem, so general conclusions hard to draw

# More Information


- Henry Dudeney.
  *Amusements in Mathematics*.
  Project Gutenberg, 1917.
  `http://www.gutenberg.org/etext/16713`.
- J.L. Lauriere.
  ALICE: A language and a program for solving combinatorial problems.
  *Artificial Intelligence*, 10:29–127, 1978.
- R. Haralick and G. Elliot.
  Increasing tree search efficiency for constraint satisfaction problems.
  *Artificial Intelligence*, 14:263–313, 1980.






# Exercises

1. Write a program for the 0/1 model of the puzzle as described above. Explain the problem with introducing a dynamic variable ordering for this model.
2. It is possible to express the problem with only three `alldifferent` constraints. Can you describe this model?
3. What is the impact of using a more powerful consistency method for the `alldifferent` constraint in our model? How do the search trees differ to our solution? Does it pay off in execution time?
4. Describe precisely what the `reorder` predicate does. You may find it helpful to run the program with instantiated lists of varying length.
5. The credit search takes two parameters, the total amount of credit and the extra number of backtracks allowed after the credit runs out. How does the program behave if you change these parameters? Can you explain this behaviour?