Centre for
Data Analytics

Insight

Chapter 6: Search Strategies (N-Queens)

Helmut Simonis

ACP Winterschool 2024

Licence

Acknowledgments

This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant number 12/RC/2289-P2 at Insight the SFI Research Centre for Data Analytics at UCC, which is co-funded under the European Regional Development Fund.
A version of this material was developed as part of the ECLiPSe ELearning course:
`https://eclipseclp.org/ELearning/index.html`.
Support from Cisco Systems and the Silicon Valley Community Foundation is gratefully acknowledged.

What we want to introduce

- Importance of search strategy, constraints alone are not enough
- Two schools of thought
  - Black-box solver, solver decides by itself
  - Human control over process
- Dynamic variable ordering exploits information from propagation
- Variable and value choice
- Hard to find strategy which works all the time
- Different way of improving stability of search routine

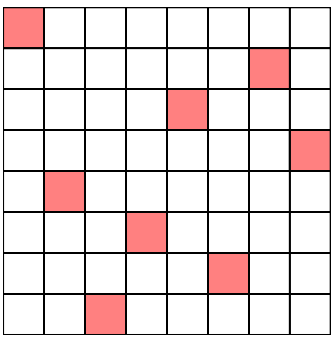
Example Problem

- 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 8×8 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 8×8

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

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

s.t.

$$\begin{aligned}
\forall 1 \le i \le N: &\quad X_i \in 1..N \\
\forall 1 \le i < j \le N: &\quad X_i \neq X_j \\
\forall 1 \le i < j \le N: &\quad X_i + j \neq X_j + i \\
\forall 1 \le i < j \le N: &\quad X_i + i \neq X_j + j
\end{aligned}$$

Nqueens Models

- ECLiPSe ▸ Show
- MiniZinc ▸ Show
- NumberJack ▸ Show
- CPMpy ▸ Show
- Choco-solver ▸ Show

ECLiPSe N-Queens Model

```
:- lib(lists).
:- lib(ic).
:- lib(ic_search).

top:-
    queens(8,Board),
    search(Board, 0, input_order, indomain, complete).

queens(N, Board) :-
    length(Board, N),
    Board :: 1..N,
    ( fromto(Board, [Q1|Cols], Cols, []) do
        ( foreach(Q2, Cols), param(Q1), count(Dist,1,_) do
            noattack(Q1, Q2, Dist)
        )
    ).

noattack(Q1,Q2,Dist) :-
    Q2 #\= Q1,
    Q2 - Q1 #\= Dist,
    Q1 - Q2 #\= Dist.
```

▸ Continue

MiniZinc N-Queens Model

```
int: n=8;
array[1..n] of var 1..n: queens;
constraint
    forall(i, j in 1..n where i < j) (
         queens[i] != queens[j] /\
         queens[i] + i != queens[j] + j /\
         queens[i] - i != queens[j] - j
    )
 ;
solve :: int_search(
        queens,
        input_order,
        indomain_min)
        satisfy;
```

▸ Continue

NumberJack N-Queens Model

```
from Numberjack import *

def get_model(N):
    queens = VarArray(N, N)
    model = Model(
        AllDiff(queens),
        AllDiff([queens[i] + i for i in range(N)]),
        AllDiff([queens[i] - i for i in range(N)])
    )
    return queens, model

def solve(param):
    queens, model = get_model(param['N'])
    solver = model.load(param['solver'])
    solver.setHeuristic(param['heuristic'], param['value'])
    solver.setVerbosity(param['verbose'])
    solver.setTimeLimit(param['tcutoff'])
    solver.solve()
```

▸ Continue

CPMpy N-Queens Model

```
def nqueens_naive(n=8):
    queens = IntVar(1,n, shape=n)

    model = Model()
    for i in range(n):
        for j in range(i):
            model += [queens[i] != queens[j],
                  queens[i] + i != queens[j] + j,
                  queens[i] - i != queens[j] - j,
                  ]
```

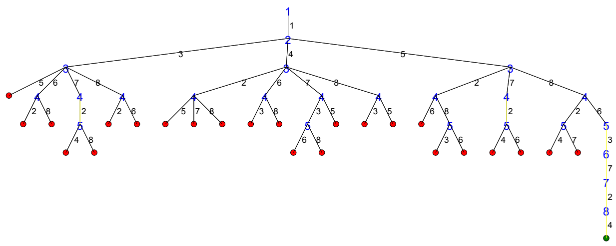
▸ Continue

Choco-solver N-Queens Program

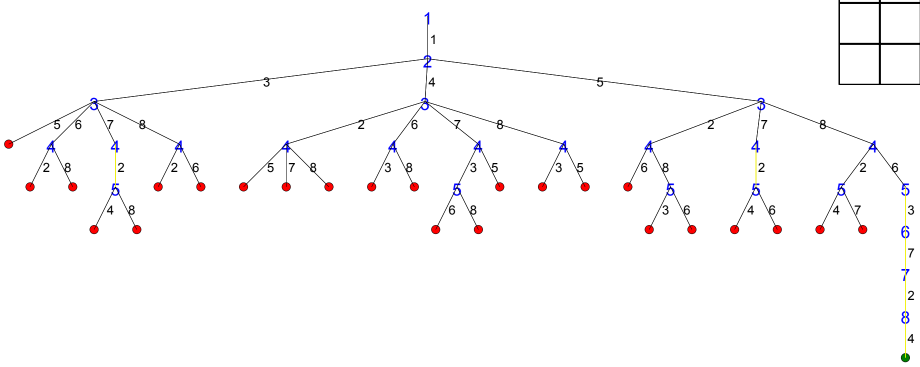
```
int n = 8;
Model model = new Model(n + "-queens problem");
IntVar[] vars = new IntVar[n];
for(int q = 0; q < n; q++){
    vars[q] = model.intVar("Q_"+q, 1, n);
}
for(int i  = 0; i < n-1; i++){
    for(int j = i + 1; j < n; j++){
        model.arithm(vars[i], "!=",vars[j]).post();
        model.arithm(vars[i], "!=", vars[j], "-", j - i).post();
        model.arithm(vars[i], "!=", vars[j], "+", j - i).post();
    }
}
Solution solution = model.getSolver().findSolution();
if(solution != null){
    System.out.println(solution.toString());
}
```

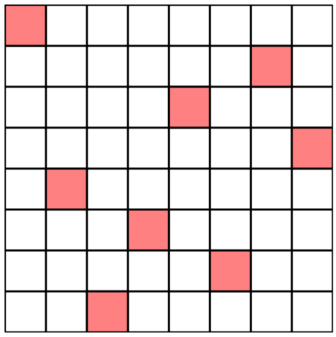
▸ Continue

Default Strategy



First Solution



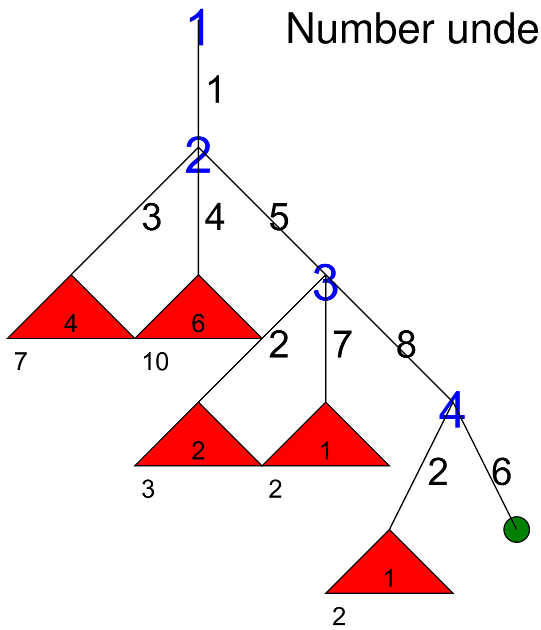


Observations

- 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

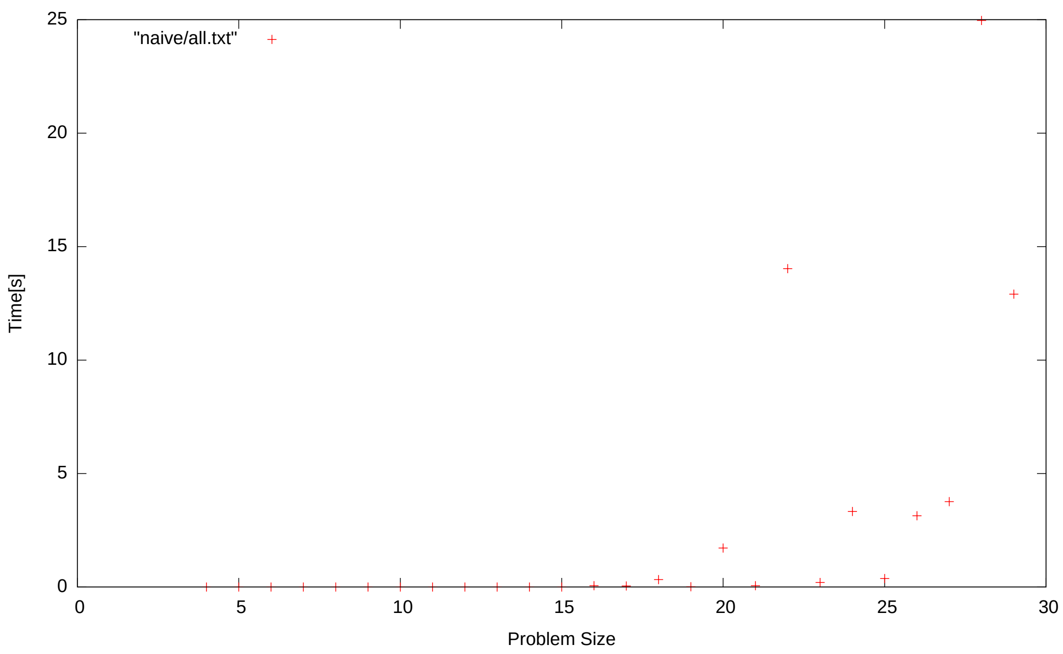
Number inside triangle: Number of choices
Number under triangle: Number of failures



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
- 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

Search Stategy Choices

- Minizinc ▸ Show
- Choco-solver ▸ Show

Modified MiniZinc Program

```
int: n=8;
array[1..n] of var 1..n: queens;
constraint
    forall(i, j in 1..n where i < j) (
         queens[i] != queens[j] /\
         queens[i] + i != queens[j] + j /\
         queens[i] - i != queens[j] - j
    )
 ;
solve :: int_search(
        queens,
        first_fail,
        indomain_min)
        satisfy;
```

Variable Choice (MiniZinc)

- Determines the order in which variables are assigned
- `input_order` assign variables in static order given
- `smallest` assign variable with smallest value in domain first
- `first_fail` select variable with smallest domain first
- `dom_w_deg` consider ratio of domain size and failure count
- Others, including programmed selection for specific solvers

Value Choice (MiniZinc)

- Determines the order in which values are tested for selected variables
- `indomain_min` Start with smallest value, on backtracking try next larger value
- `indomain_median` Start with value closest to middle of domain
- `indomain_random` Choose values in random order
- `indomain_split` Split domain into two intervals

▸ Continue

Modified Choco-solver Model

```
int n = 8;

Model model = new Model(n + "-queens problem");
IntVar[] vars = model.intVarArray("Q", n, 1, n, false);
IntVar[] diag1 = IntStream.range(0, n).
                           mapToObj(i -> vars[i].sub(i).intVar()).
                           toArray(IntVar[]::new);
IntVar[] diag2 = IntStream.range(0, n).
                           mapToObj(i -> vars[i].add(i).intVar()).
                           toArray(IntVar[]::new);

model.post(
    model.allDifferent(vars),
    model.allDifferent(diag1),
    model.allDifferent(diag2)
);

Solver solver = model.getSolver();
solver.showStatistics();
solver.setSearch(Search.domOverWDegSearch(vars));
Solution solution = solver.findSolution();

if (solution != null) {
    System.out.println(solution.toString());
}
```


VariableSelector Choice (Choco-solver)

- Determines the order in which variables are assigned
- `InputOrder` assign variables in static order given
- `Smallest` assign variable with smallest value in domain first
- `FirstFail` select variable with smallest domain first
- `DomOverWDeg` consider ratio of domain size and failure count
- `ActivityBased` dynamic, based on dynamic observed behaviour
- `ImpactBased` dynamic, based on dynamic observed behaviour

IntValueSelector Choice (Choco-solver)

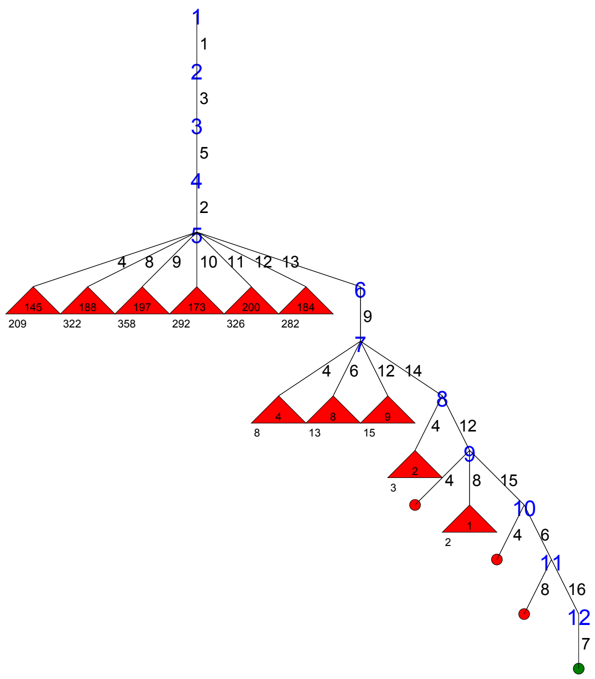
- Determines the order in which values are tested for selected variables
- `IntDomainMin` Start with smallest value, on backtracking try next larger value
- `IntDomainMiddle` Start with value closest to middle of domain
- `IntDomainRandom` Choose values in random order
- `IntDomainRandomBound` Randomly choose between smallest and largest value

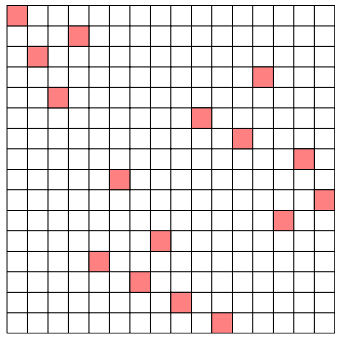
Continue

Comparison

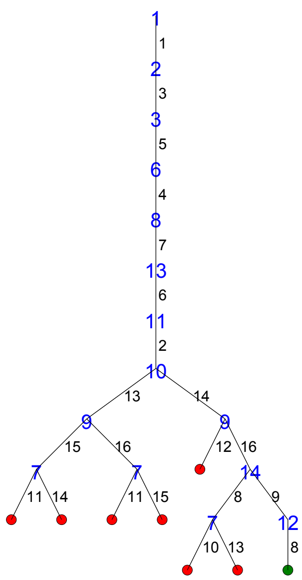
- Board size 16x16
- Naive (Input Order) Strategy
- First Fail variable selection

Naive (Input Order) Strategy (Size 16)

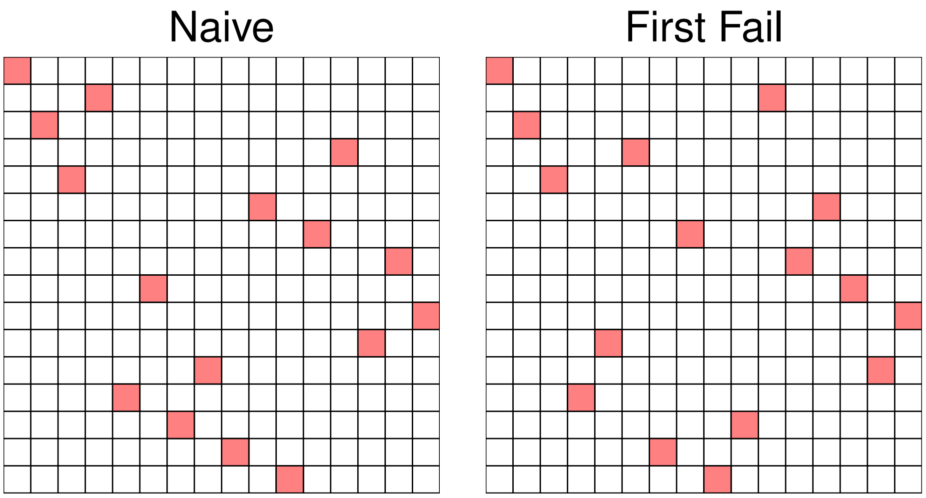




FirstFail Strategy (Size 16)

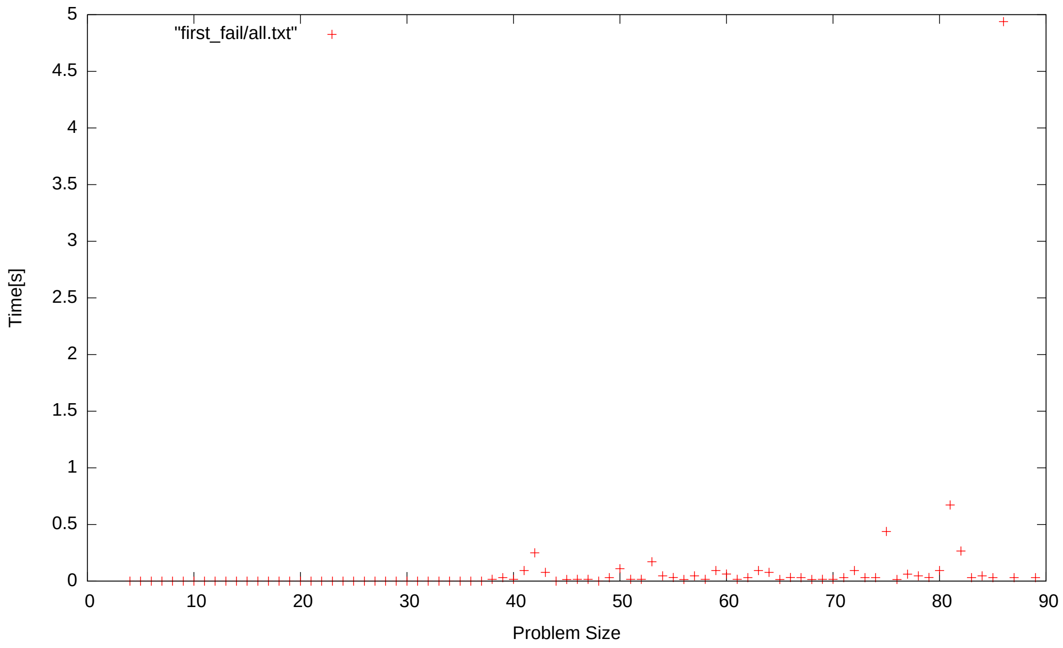


Comparing Solutions



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

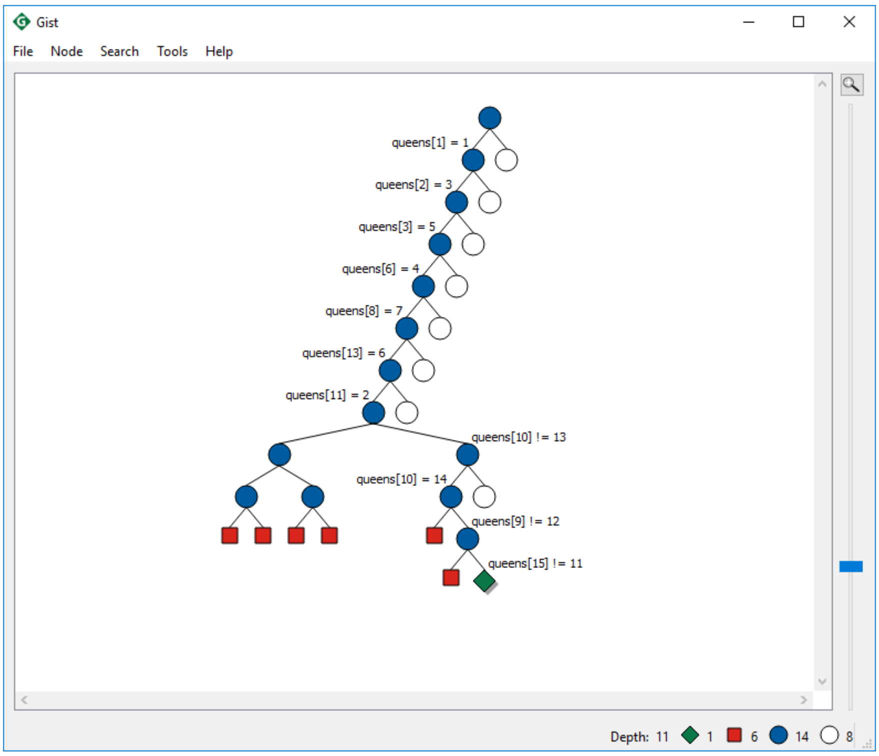
More Reactive Variable Selection

- Domain size is important, but other information is useful as well
- Dom/Weighted Degree: better results in many situations
- Weight Degree: count how often variable has been involved in failure
- Focus on more complicated part of problem
- Changes during search, learns from past performance
- Option **dom_w_deg**

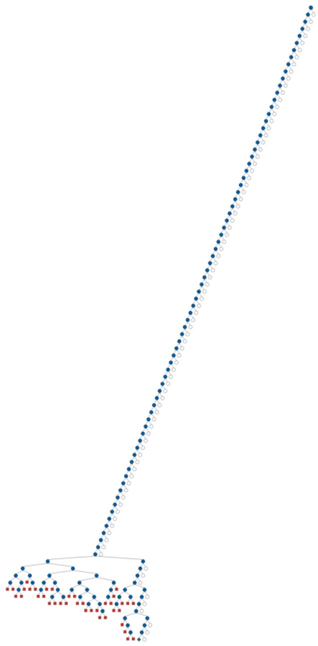
Weighted Degree Variable Selection

```
int: n=8;
array[1..n] of var 1..n: queens;
constraint
    forall(i, j in 1..n where i < j) (
         queens[i] != queens[j] /\
         queens[i] + i != queens[j] + j /\
         queens[i] - i != queens[j] - j
    )
 ;
solve :: int_search(
        queens,
        dom_w_deg,
        indomain_random)
        satisfy;
```

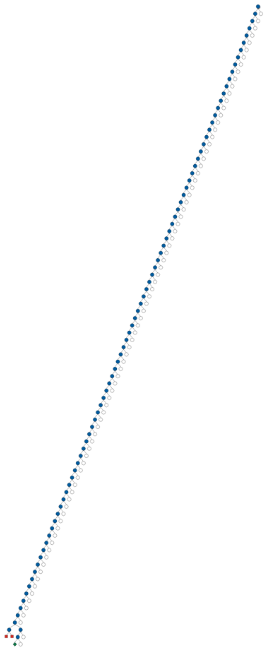

Result for size 16 with Gecode-Gist



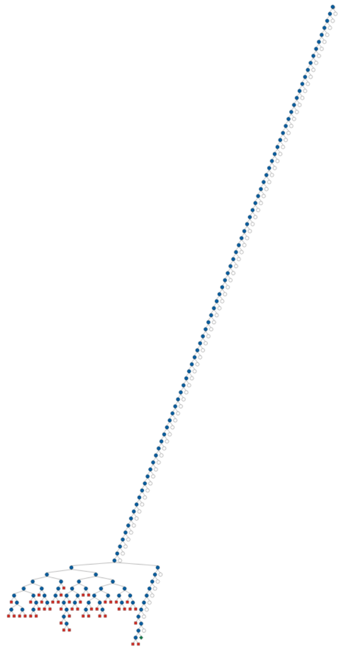
Sample Results for Larger Sizes



Size 93



Size 94



Size 95

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

Random Variable Choice and Restarts

```
int: n=8;
array[1..n] of var 1..n: queens;
constraint
    forall(i, j in 1..n where i < j) (
         queens[i] != queens[j] /\
         queens[i] + i != queens[j] + j /\
         queens[i] - i != queens[j] - j
    )
 ;
solve :: int_search(
        queens,
        dom_w_deg,
        indomain_random)
        :: random_linear(100)
        satisfy;
```

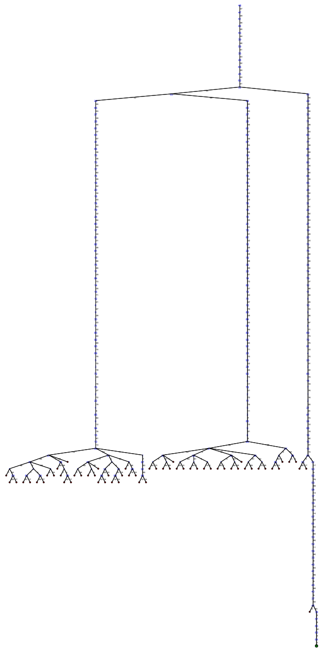
Approach 3: Partial Search

- 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

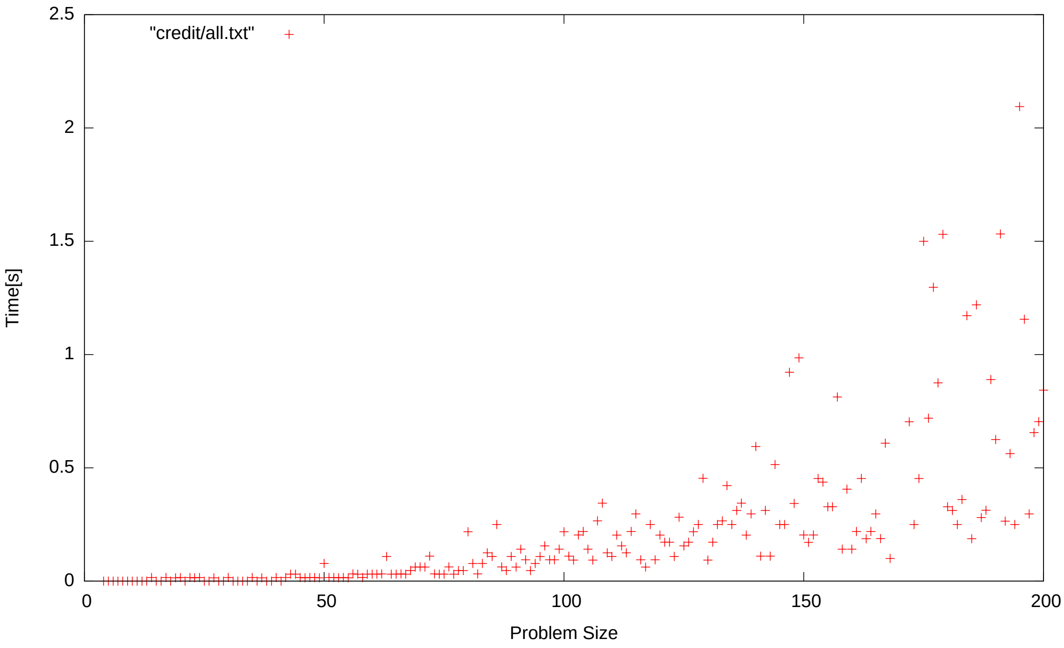
Example: Credit Search

- Not available in all solvers
- 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, Search Tree Problem Size 94



Credit, Problem Sizes 4-200



Points to Remember

- Choice of search can have huge impact on performance
- Dynamic variable selection can lead to large reduction of search space
- Packaged search can do a lot, but programming search adds even more
- Depth-first chronological backtracking not always best choice
- How to control this explosion of search alternatives?