Chapter 6: Search Strategies (N-Queens)

Helmut Simonis

email: `helmut.simonis@insight-centre.org`
homepage: `http://http://insight-centre.org/`

Insight SFI Centre for Data Analytics
School of Computer Science and Information Technology
University College Cork
Ireland

CRT-AI CP Week 2025

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

As example for the different techniques we use the N-Queens puzzle, a standard problem in Artificial Intelligence and Constraint Programming. The model we use has rather weak constraint propagation, there is not that much information that is deduced from the constraints, so that we rely more on search in solving this problem. The puzzle has many solutions, indeed too many to easily enumerate beyond sizes 10 or 12, and only a limited number of symmetries. A big advantage is that we can easily scale the problem size to create a number of related problem instances, which helps us to understand problems of stability of the methods considered.

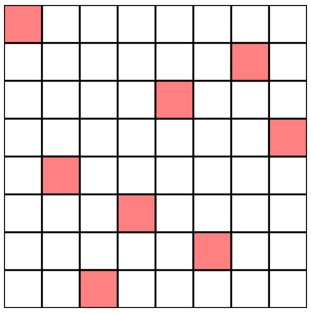
1 Problem

Let's look at the problem in a bit more detail.

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 $N \times N$.

Figure 1: Solution for board size 8×8



The 8-Queens puzzle is the problem of placing eight queens on a chess board so that no queen attacks another. A queen attacks all cells in horizontal, vertical or diagonal direction. The original puzzle is for an 8×8 chess board. Obviously, we can easily generalize this problem to an arbitary board size $N \times N$.

The diagram (Figure 1) shows a solution for the 8×8 case, the queens are marked in red. It is easy to see that no queen attacks any other queen in this solution.

2 Program

We will now discuss how to model and solve this problem with ECLiPSe.

2.1 Model

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

There are basically two models for this problem, one based on cells, the other on columns (or rows).

In the first model we introduce 0/1 integer variables for all cells, they describe if a cell contains a queen or not. Linear constraints on the rows, columns and the diagonals enforce that there is at most one queen in each of them. We also have a constraint stating that we must place N queens. This means that we have N^2 variables, and $N + N + 2 * (2N - 1) + 1 = 6N - 1$ constraints.

The alternative to this model is a column (or row) based model, where we have a variable for each column (or row) of the board. This is based on the observation that each column must contain exactly one queen. If a column contains more than one queen, they would attack each other. A column can also not be empty. We need N queens, and we have N columns, each containing atmost one queen. By the *pigeon-hole principle* we know that there can be no empty columns, each must contain exactly one queen. The variable for a column then ranges over the values from 1 to N, and gives the position of the queen in the column.

This choice of model automatically handles the no-attack condition in the vertical direction, so we don't need a special constraint for this. In the horizontal direction we can express the no-attack condition by a binary disequality between any two column variables, i.e. the variables can not have the same values. But this means the variables must be pairwise different, and that is just the definition of the `alldifferent` constraint, which we already encountered in earlier examples. We can therefore express the no-attack condition in horizontal direction by a single `alldifferent` constraint between all variables.

This leaves us with the no-attack condition for the diagonals. They can also be expressed by disequalities, but we have to add an offset between the variables, which is based on their distance $j - i$.

Model

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

s.t.

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

We can write down our model in a concise mathematical form as a set of constraints and use this as the basis for our ECLiPSe program.

3 Naive Search

Nqueens Models

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

ECLiPSe N-Queens Model

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

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

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

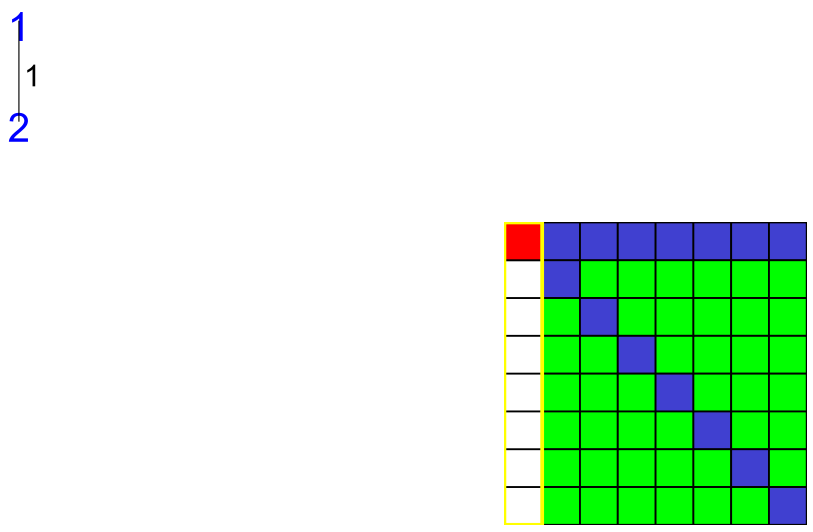
Choco-solver N-Queens Program

```
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

When it comes to search, it does not matter which variant of the program we use. They create the same constraints, and assign the variables in the same order from X_1 to X_N. We will now use some visualization to see how the search routine finds its first solution.

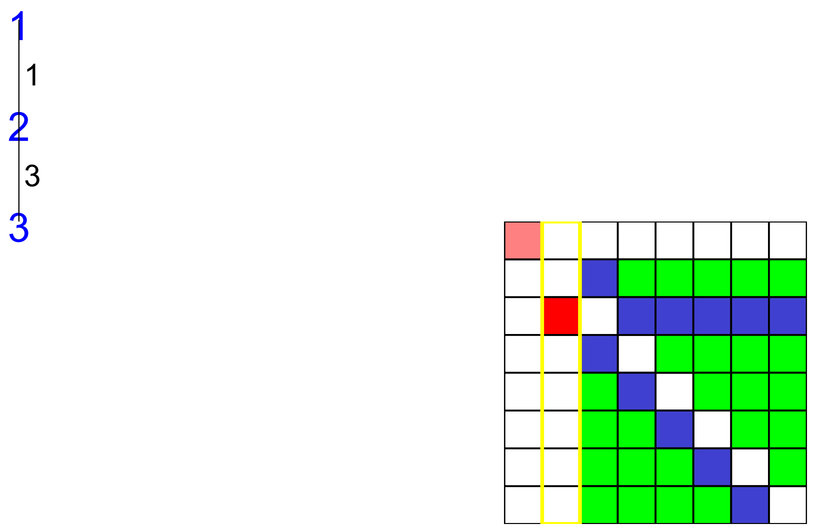
Figure 2: First Assignment



We see (in Figure 2) the search tree on the left, and the current partial assignment on the right. As the first assignment we have fixed X_1 to value 1. This is shown as a red rectangle on the board on the right, indicating the value selected. The currently assigned variable is represented by a yellow outline. The constraint propagation after the assignment has removed values from the other variables. The removed values are shown in blue, the remaining values in the domains are shown as green.

In the second assignment step (Figure 3), we assign a value for variable X_2. We use value 3, since the values 1 and 2 have been removed from the domain by the previous assignment.

Figure 3: Second Assignment



In the next step (Figure 4), we encounter a problem. When we try to assign value 5 to variable X_3, we detect a failure in constraint propagation. This is why the variable is outlined in red on our board, it is also shown by a red leaf node in the search tree. After the failure, we backtrack to the last open choice, this is variable X_3, and try the next available value. In this way we continue our search, stopping whenever we detect a failure and continuing until all variables are assigned or no more choices are left.

Finally (Figure 5) we find a solution after quite a few backtracking steps. On the right we see the solution found, on the left the search tree explored. Some links in the tree are shown in yellow, this means they are forced assignments made by constraint propagation. If we go back a few steps, we sees that as soon as variable X_4 is assigned to 6, the remaining variables are assigned by propagation, so we don't have to choose values for those variables.

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

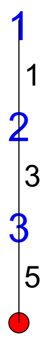
What we can see is that even for small problem sizes, the search tree can become quite large. We are normally not interested in all details of the propagation and the search, so we will tell the system to ignore things like automatically fixed variables.

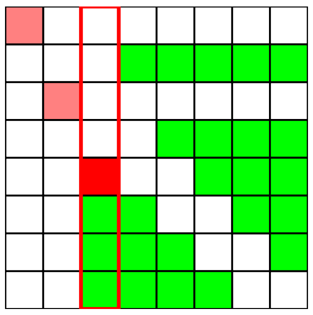
Sometimes we also want to have a shorter representation of the search, a tree in the form shown here may become too wide to fit on our display. In that case, we can choose a more compact representation, which compresses all failed subtrees into little red triangles.

We get a view like this (Figure 6). We have our variable X_1, which we assign to value 1, we then choose variable X_2, for which we first try value 3. This leads to a dead subtree, which does not contain any solutions. This is marked by a triangle, the number inside gives the number of interior nodes (here 4) and the number below gives the number of failures in the subtree (here 7).

Eventually we come to a solution in the green solution node at the bottom, after having assigned four variables.

Figure 4: Third Assignment





Exploring other board sizes

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

One interesting question is “How stable is this model?”, can we use it to solve the problem for different board sizes. To make this experiment, we try out all sizes from 4 to 100, with a timeout of 100 seconds. Board sizes 2 and 3 do not allow solutions, you may want to check this yourself on a piece of paper.

In Figure 7 we have a plot of the solutions we obtained. The board size is on the x-axis, time required on the y-axis. We see that initially, we find solutions quite rapidly, up to size 20. For larger sizes, the times vary a lot, for size 25 we use less than a second, but for size 28 we need 25 seconds. More importantly, we don’t find any solutions for sizes greater than 30 within the timeout limit of 100 seconds.

This problem is not just linked to problem size. We would expect the program to take longer for a bigger board, but this is not a simple monotonic function. Some problem sizes require much more time to solve than others. What can we do about this?

Figure 5: First Solution

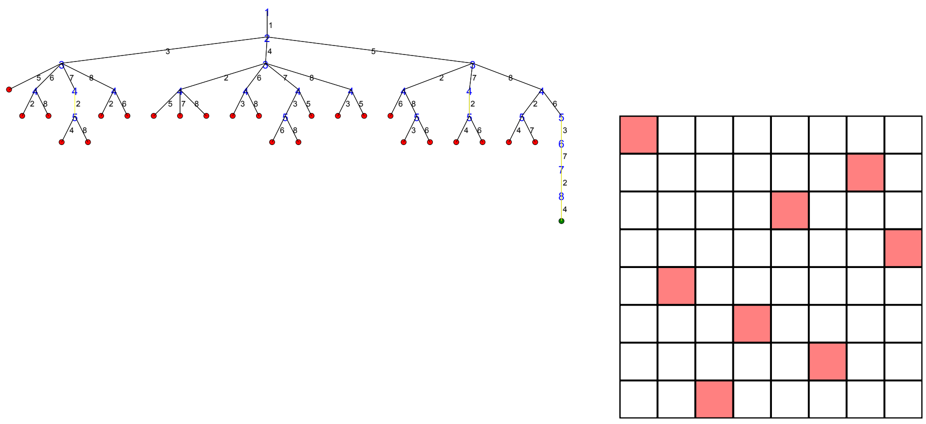


Figure 6: Compact Tree Presentation

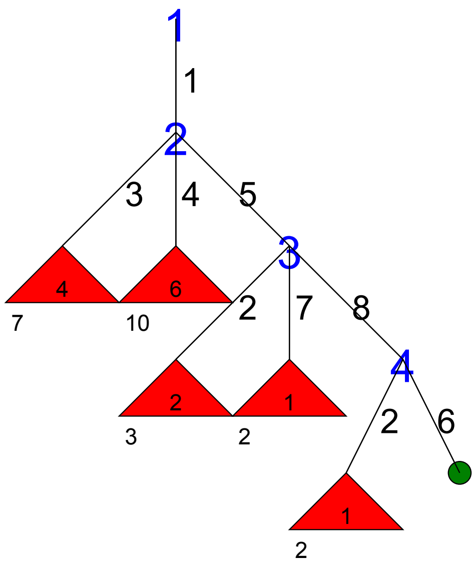
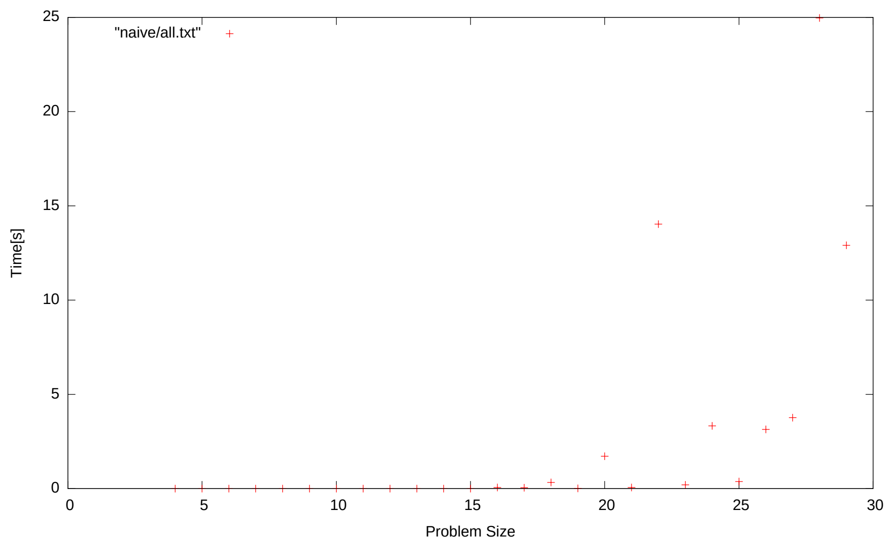


Figure 7: Naive Stategy, Problem Sizes 4-100



4 Improvements

4.1 Dynamic Variable Choice

Can we do better than our naive, straighforward program?

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

There are two ways of improving the situation. One would use better constraint reasoning, so that we detect failures earlier and do not have to explore large, dead subtrees. A way to do this is to remodel our problem with three `alldifferent` constraints, and to use the stronger consistency methods that we have explored in the previous chapter. This is not working that well for this problem, and we do not explore this here.

A more promising alternative is to improve our control of the search, introducing a dynamic instead of a static variable ordering. We can also use a better value choice, selecting a more promising value for the selected variable. Finally, we can replace our complete, chronological backtracking search with a more appropriate method.

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

We can improve the search routine by either sorting the variables before the search is started based on a heuristic, static ordering or by using a dynamic variable ordering during search, or indeed by a combination of both methods.

For a static ordering, the idea is to make important decisions early in the search, i.e. to choose those variables which have the biggest impact on the search. By making these choices early, we reduce the problem size as quickly as possible. We can also try to assign the variables with small domains first, so that we have fewer alternative branches to explore at the top of the tree. The drawback of such a static heuristic is that we make rather uninformed choices. We can only use the information available at the beginning of the program.

It is usually much better to make the variable selection choice dynamically, this way we can use the current state of the propagation to guide our choice. This can also mean that we use a different ordering in one part of the search tree from the one in another part of the tree. This way we use all information available at every step.

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

One such dynamic selection method is the first fail strategy. At each step, we select the variable which has the smallest domain. There are typically two justifications given for this:

- If there are very few solutions, we have a better chance of finding it by branching on few alternatives.
- If there are no solutions, then we have fewer alternatives to explore, and this should require less time than choosing a variable with a large domain, and exploring many alternatives.

This first fail principle needs a tie-breaking method, to decide what to do if several variables have the same, smallest domain size. In ECLiPSe, we typically use the first variable which has the smallest domain.

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

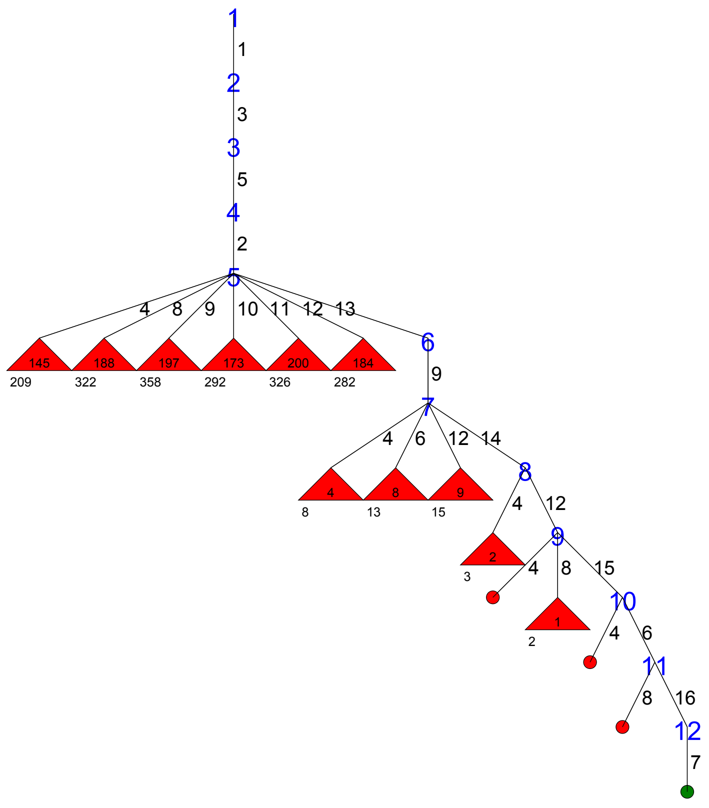
What changes are required to use the first fail princple in our example problem? We have to replace the `labeling` routine with a more general method, the `search` builtin of the ic library, where we give the list of variables to assign, and then a whole set of parameters.

Comparison

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

If we want to see how these methods compare, we should use a bigger problem size than the 8×8 board. For demonstration purposes, we use the 16×16 board. We start with the naive method, `input_order` and `indomain` and compare it to the `first_fail` strategy.

Figure 8: Naive (Input Order) Strategy (Size 16) Search Tree



This (Figure 8) is the search tree required for the naive method. This does not look too bad, until you check the size of the failed sub-trees. For variable X_5 we explore 6 failed alternatives with hundreds of failures each, before we make the correct choice.

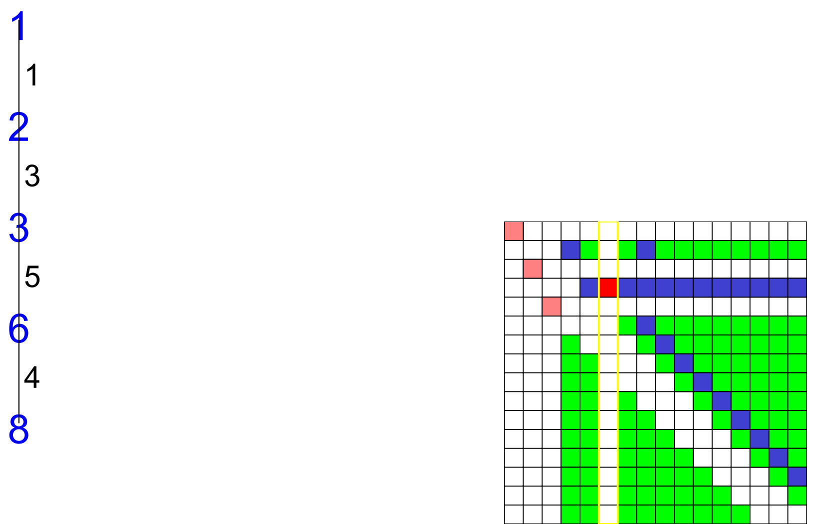
This (Figure 9) is the fourth assignment step in our modified program, and we can see that instead of variable X_4 the program selects X_6 for assignment. From there, the search differs dramatically from the naive method.

If we look at the complete tree for the first fail strategy (Figure 10), we see it is much smaller, requiring only 7 failures before finding a solution.

Also note that the variable order differs in the branches of the tree; on the left side, variable 7 is selected just before the failure, on the right side variable 14 is selected at that level.

It is also important to note that the solutions obtained by the two programs are different, as shown in figure 11. We might be just lucky finding a solution earlier with first fail than the naive method.

Figure 9: First Fail, Fourth Assignment



For this problem we don't care which solution we obtain, we only look for a feasible solution, which satisfies all constraints. For other problems, we may be interested in optimizing a cost function. In that case it is important to generate the feasible solutions in a good order, finding solutions with low cost early on.

We again test our program on all instances from 4 to 100. Here (Figure 12) we have the plot of the execution times required. This is much better than before, note the different time scale on the left. We often find solutions nearly instantly, right up to size 100. But there are still some problems which are more difficult to solve. We still don't find solutions for sizes 88, 91, 93, 97, 98, 99 within the 100 second time limit.

Using the dynamic variable selection with the first fail strategy was a very successful idea, but it does not solve the problem completely.

Figure 10: FirstFail Strategy (Size 16)

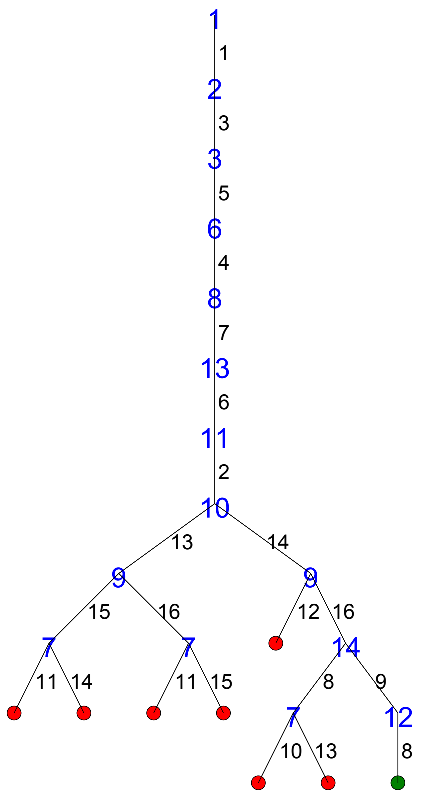


Figure 11: Solutions are different!

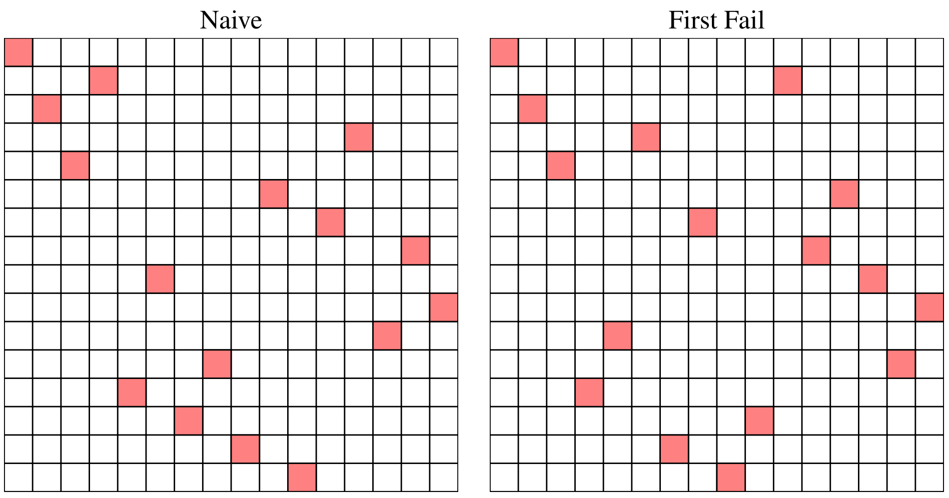
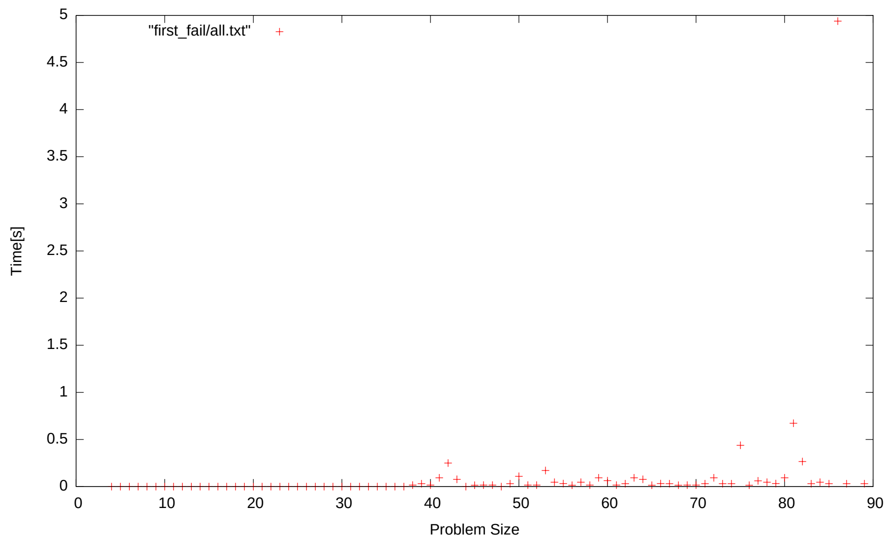


Figure 12: FirstFail, Problem Sizes 4-100



4.2 Weighted Degree

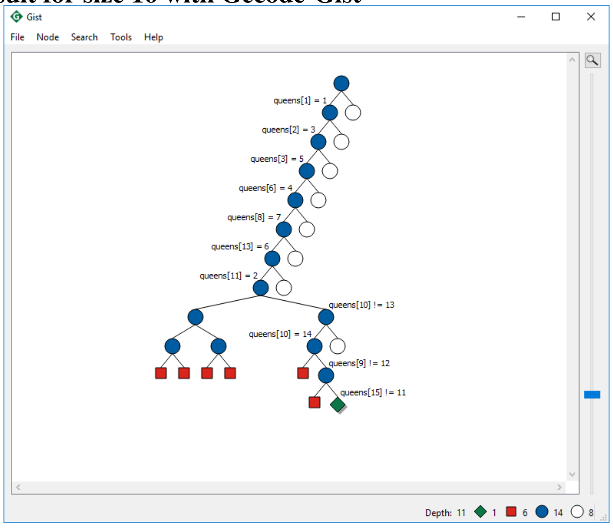
More Reactive Variable Selection

- 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**

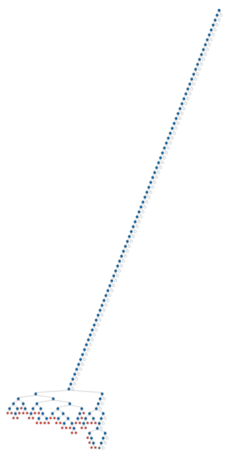
Weighted Degree Variable Selection

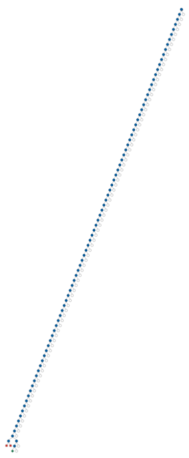
```
int: n=8;
array[1..n] of var 1..n: queens;
constraint
    forall(i, j in 1..n where i < j) (
         queens[i] != queens[j] /\
         queens[i] + i != queens[j] + j /\
         queens[i] - i != queens[j] - j
    )
 ;
solve :: int_search(
        queens,
        dom_w_deg,
        indomain_random)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

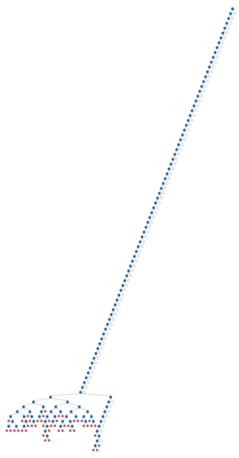
Result for size 16 with Gecode-Gist



Sample Results for Larger Sizes


Size 93


Size 94


Size 95

4.3 Improved Heuristics

4.4 Making Search More Stable

We will now consider some methods to improve the stability of our program.

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

A first idea is to use a portfolio of search strategies, and to run them in parallel. With multi-core CPUs, it is quite possible to use four or more search routines at the same time, and to stop the overall search when one of them has found a solution.

An important point is to select just a few, quite different strategies for the portfolio. We might even be able to select good methods automatically based on the structure of the problem that we are trying to solve. This is an active research area at the moment.

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;
```

A perhaps more resource-friendly method uses restart with randomization. The idea is to allow only a limited number of backtracks for a given search attempt. If that limit is exceeded, then the search should restart at the beginning, perhaps exploring another part of the search tree. If the problem has many solutions, we are likely to find one with such a strategy even if constraint propagation is not very powerful.

In order to work, this requires some form of randomization, otherwise we would explore the same part of the search tree over and over again. We can randomize the variable selection, for example by a random tie break, and/or randomize value selection. We also need to decide how much search to allow for each restart attempt.

We will consider such a randomized search routine later in the course.

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

A third way of improving search behavior is to abandon the depth-first chronological backtracking. The idea is not to get locked into a failed sub-tree, which might contain many, many nodes. This happens when we make a wrong descision at some step, but don't detect this immediately. In an ideal world we would just strengthen our constraint reasoning to detect such failures, but unfortunately this is a hard problem that we can't solve in general. Instead, we are left inside a subtree which has no solution, and we have to spend a lot of time exploring all the remaining alternatives, before we have a chance of finding a solution eventually.

In a partial search, we don't continue in such a case, but spend our time exploring other, perhaps more promising parts of the search tree as well.

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

We can see (in Figure 13) the typical behavior of the credit based search. We start with an initial branch, which at some point starts backtracking. We give up on this branch, jump back to the unexplored part of the credit search and choose another branch there. This may lead to a failure again, but eventually we make a good choice there and find a solution.

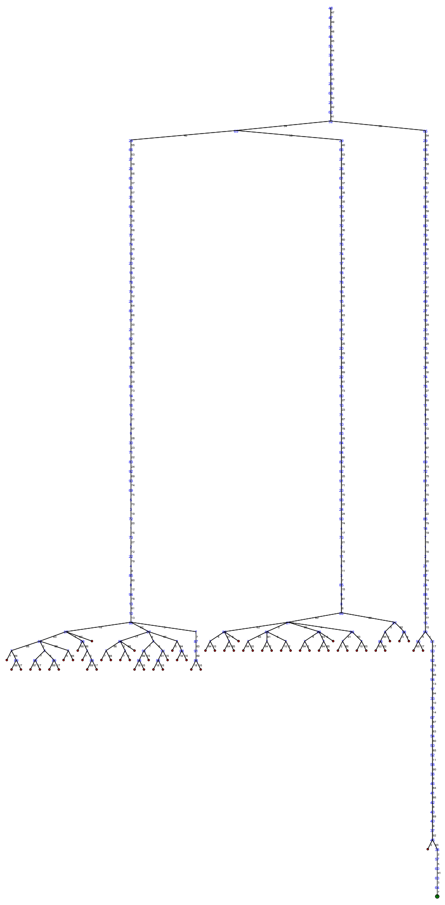
In this example, we have only just started to explore the top part of the tree, there are many unexplored choices left there.

This technique works best if the solutions are not too sparse, and the initial choices really have a big impact on the overall solution obtained. A potential problem is that we might give up too easily with our limited backtracking steps, never finding a solution at all.

We can continue to run the program for larger board sizes, this (Figure 14) shows results for all sizes up to 200. The maximal time needed for any instance is just over 2 seconds.

Nobody really needs to solve the problem with 200 queens or more, so our solution is probably good enough in that sense. But if we really want to solve bigger problem instances, we can use a repair based technique which we will discuss in a later chapter.

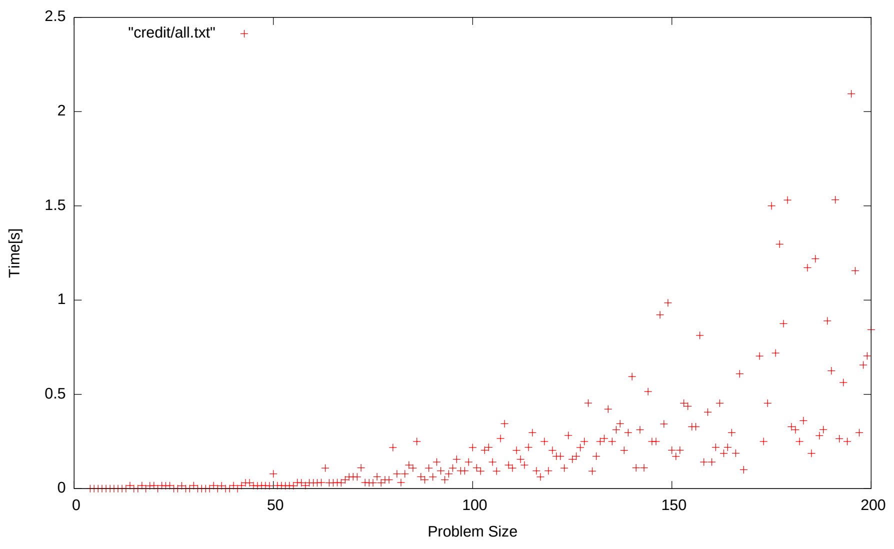
Figure 13: Credit, Search Tree Problem Size 94



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?

Figure 14: Credit, Problem Sizes 4-200


"credit/all.txt" +
Time[s]
Problem Size