Chapter 6: Search Strategies (N-Queens)

Helmut Simonis

email: h.simonis@4c.ucc.ie
homepage: http://4c.ucc.ie/~hsimonis

Cork Constraint Computation Centre
Computer Science Department
University College Cork
Ireland

ECLiPSe ELearning Overview

Outline

Contents

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

Welcome to chapter 6 of the ECLiPSe ELearning course. This chapter is about search strategies for finite domain constraint programs. We will use the N-Queens puzzle as the main example in this chapter.

I'm Helmut Simonis, I'm working at the Cork Constraint Computation Centre at University College Cork.

We will first talk about our example, the N-Queens puzzle, and describe the problem in a bit more detail. Then we will show how a simple ECLiPSe program can model this problem. We then discuss how a naive search routine based on chronological backtracking and depth-first search solves the puzzle for the original problem size, but has difficulty scaling up for larger board sizes. As a refinement step we can improve the program by using a dynamic variable ordering and the `search` builtin. Finally, we will discuss further improvements to our program which allow us to solve the problem with hundreds of queens quite easily.

The main idea we want to present in this chapter is the importance of the search part in a constraint program. Constraint propagation will take you only so far, we have to combine it with a good enumeration strategy to find solutions quickly.

A dynamic variable ordering can be very helpful in this, it allows us to exploit information from the propagation we have made in order to make better choices for the next assignment step.

We will introduce the concepts of variable and value choice, deciding which variable to assign next and which value to choose for that variable. These are two generic design choices which give us a lot of flexibility when building search routines.

It is quite difficult to find a search strategy which works for all instances of a problem, but the `search` builtin offers many ways of adapting the search to a particular problem, and to try out different strategies with minimal effort. We are in particular interested in making search routines more stable, so that they work in a uniform manner for all problem instances we might wish to solve, and we will discuss a number of ways of achieving this.

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.

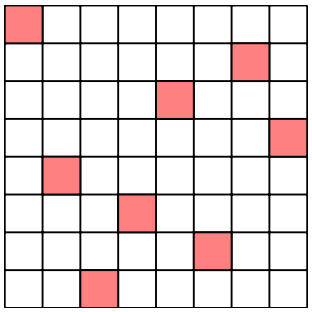
1 Problem

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.

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

This is a rather old problem, in the book by Dudeney from 1917 it is already mentioned as a well-known puzzle, attributed to Nauck of 1850.

On the other hand, it is not a difficult problem, certain solutions for all problem sizes can be constructed systematically. It is therefore not an NP-hard problem, contrary to most other examples in this course.

Even though it is an easy problem, it has long been a standard problem for AI and CP, many papers use it to explain a particular technique or to compare different solution methods. In particular, we want to mention the paper by Haralick and Elliot which introduced the first-fail principle in 1980.

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.

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

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.

2.2 Program (Array version)

To program our model in ECLiPSe, we have a choice between two variants, one using an array of variables, the other a list. Both variants use the same constraints and search, they just represent the variables differently and use different mechanismns to generate the constraints. You can use either style to write your own programs, but it is a good idea to understand both styles, as you may encounter either in programs written by other people.

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]).
```

We start with the version using an array. We define a module for our program, export our top-level predicate `top` with arity 0, and load the `ic` library for the finite domain solver. In `top`, we call the `nqueen` predicate with the problem size. This is were the actual work is done. We first set up an array of size N, then define that the array contains domain variables which range over $1..N$, and state that they must be pairwise different with an `alldifferent` constraint. We then call the predicate `noattack` which will create the disequalities between the variables for the diagonals, and finally call `labeling` to find values for the variables.

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
        )
    ).
```

The `noattack` predicate uses nested `for` loops over indices i and j to set up two disequality constraints between every pair of variables.

2.3 Program (List Version)

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).
```

The list version of the program uses lists and recursion over lists to set up the same constraints. Instead of creating an array, we create a list of size N with the `length` builtin.

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).
```

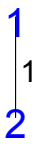
The main difference to the previous variant lies in the `noattack` predicate, which we define recursively for the list variant. `noattack` has two clauses, the first handles the terminal case, when the list is empty. The second calls the `notattack1` predicate with three arguments. The first is the current head of the list, our X_i variable. The second argument is the rest of the list, all variables which we have to constrain against the current X_i. The third argument is the distance between the variables, initially 1. The `noattack1` predicate then recurses over the second argument, the list of variables. For each variable, two disequality constraints are created. Finally, we increment the distance counter in the third argument and continue with the next variable, until we reach the end of the list.

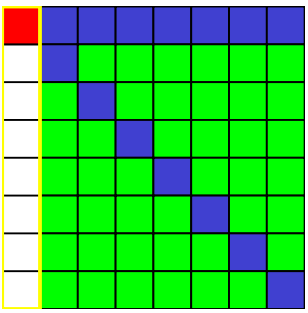
If you are not used to Prolog and recursive definitions, it might take a while to feel comfortable with this style of programming. But it is a very powerful programming concept, which allows you to do certain things easily which are much more complex with conventional programming concepts. In this course, you will see examples of both programming styles.

3 Naive Search

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.

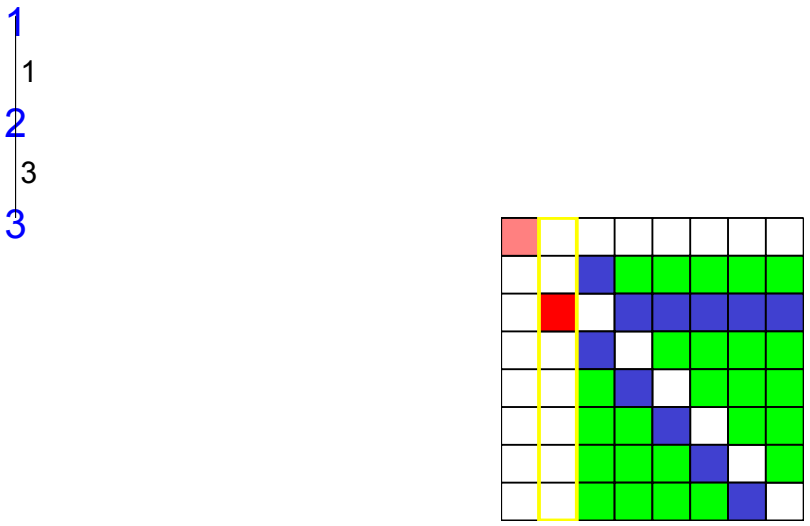
Figure 2: First Assignment





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.

Figure 3: Second Assignment



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.

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.

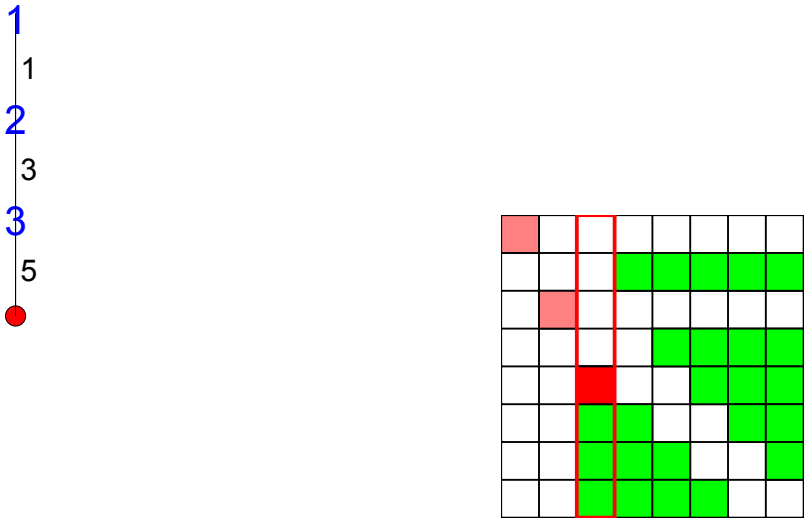
Finally (Figure 5) we find a solution after quite a few backtracking steps. On the right we see the solution found, on the left the search tree explored. Some links in the tree are shown in yellow, this means they are forced assignments made by constraint propagation. If we go back a few steps, we sees 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.

Figure 4: Third Assignment



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.

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

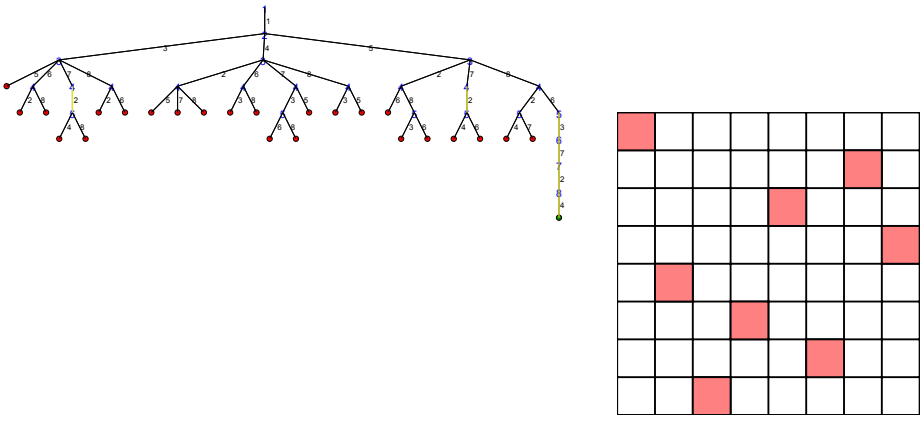


Figure 6: Compact Tree Presentation

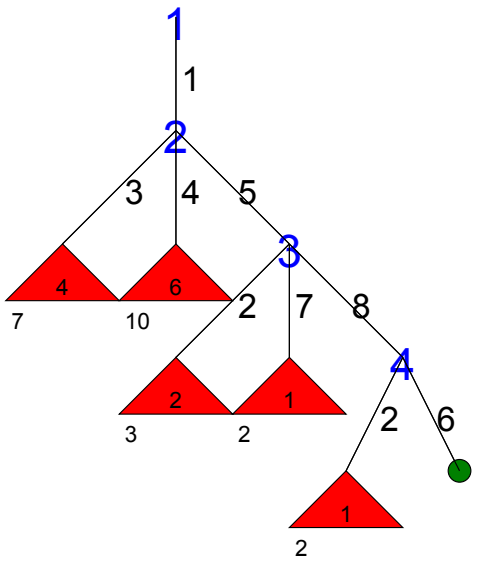
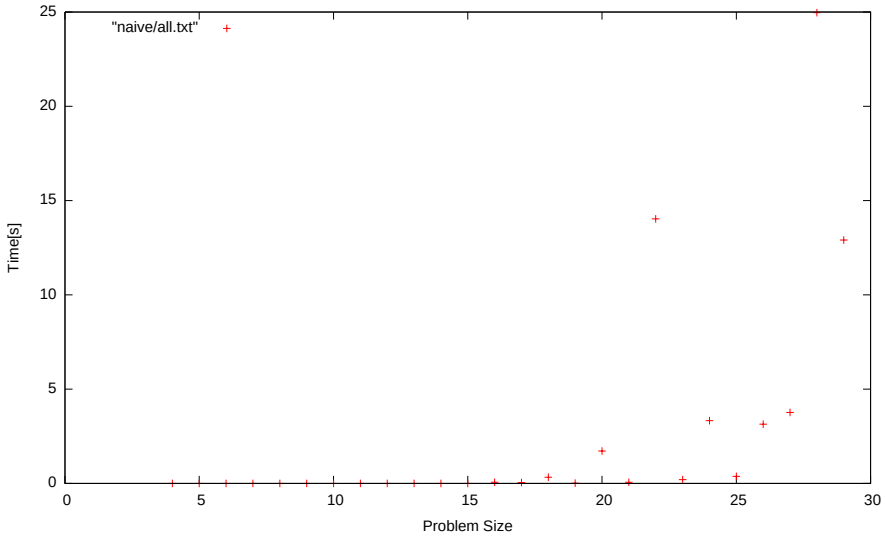


Figure 7: Naive Stategy, Problem Sizes 4-100



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
  - Not explored here
- 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.

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

It is important to note that different constraint systems use quite different tie breaking methods for their first fail strategy. This makes it very hard to compare two tools on examples which use this strategy, unless you know exactly what is happening inside the solver.

This is where the search tree visualization is very handy: It shows us exactly what is going on, and we can compare two search trees quite easily.

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,[]).
```

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.

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

The `search` predicate is a packaged search which is provided in the ic library, which provides many different combinations of search methods in a simple, accessible interface. You can test out many different search variants just be exchanging a few keywords, but it is also extensible by the user, we can integrate our own methods quite easily.

There are search methods which can not be easily programmed with the `search` builtin, and we will encounter some of them later in this course, but for many, if not most, situations, it provides all the flexibility we need.

`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

What we have done in our program is to give the `search` builtin a list of variables as its first argument, in that case the second argument must be zero. We can also pass a list of terms as first argument, in that case the second argument says which argument of the term contains the decision variable.

The third argument is the variable selection method, we now use `first_fail`, our naive method was selecting the variables in `input_order`. The fourth argument is the value choice, we use `indomain`, which enumerates the values starting with the smallest value in the domain.

The fifth argument gives the overall search method, we state that we want to use a `complete` search, which explores all alternatives with depth-first search and chronological backtracking. The last argument is used to pass additional options, and we don't use this at the moment.

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

The variable choice determines the order in which the variables are explored. `input_order` uses the order in which the variables are given. `first_fail` selects the variable with the smallest domain, using the position in the list as tie breaking. `most_constrained` selects the variable with the smallest domain, and uses the number of constraints in which the variable occurs as tie breaking. This can be very helpful for problems where the constraint structure is irregular, and some variables occur in many more constraints than others. It is a good idea to select such variables early on, as their assignment will restrict many other variables.

There are quite a few other selection methods, which you will find in the manual, and it is also possible to program your own selection method.

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

The value choice determines the order in which values are tested for a selected variable. We have used `indomain` in our examples so far, always starting with the smallest value in the domain, and on backtracking, testing the next larger value, until all choices have been explored. `indomain_max` starts with the largest value in the domain, so that the order in which the values are tested is reversed compared to `indomain`. `indomain_middle` begins with a value in the middle of the domain, and continues alternating from this middle value.

Finally, `indomain_random` uses a random order of the values. This can be very effective by spreading out the choice of possible values, not always testing the same values first. But a random choice makes it much harder to test and debug programs as well.

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.

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.

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

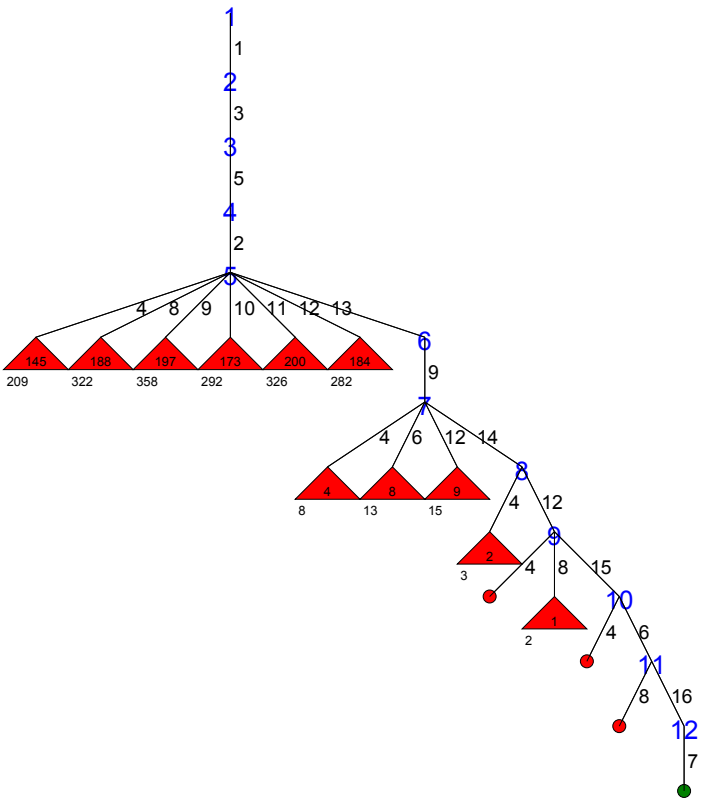


Figure 9: First Fail, Fourth Assignment

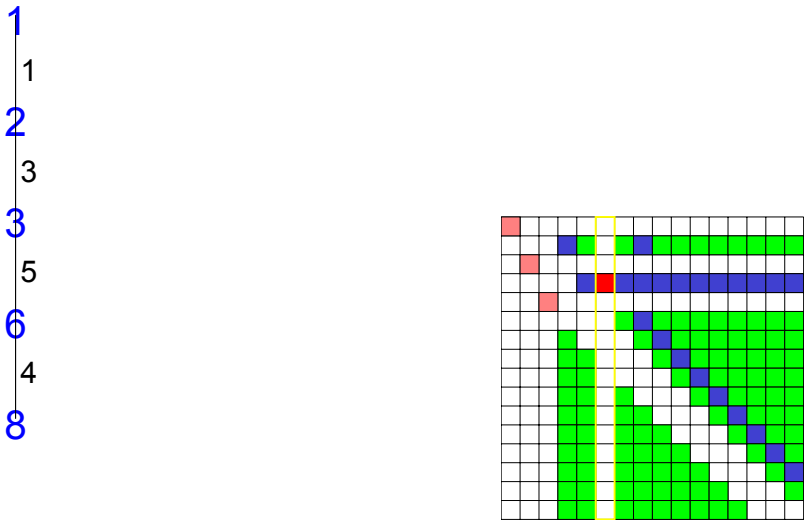


Figure 10: FirstFail Strategy (Size 16)

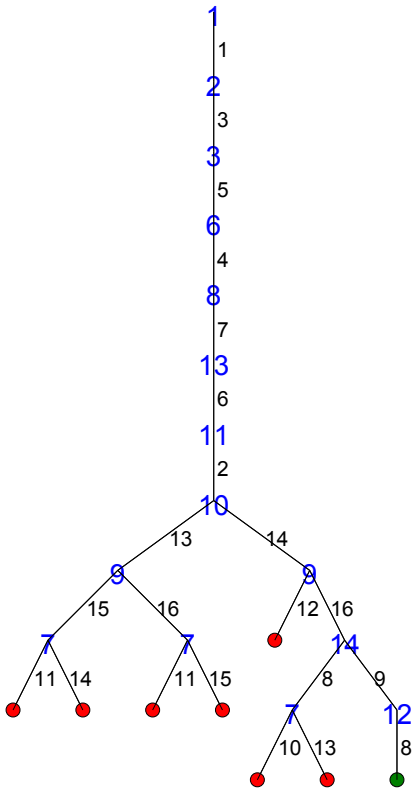


Figure 11: Solutions are different!

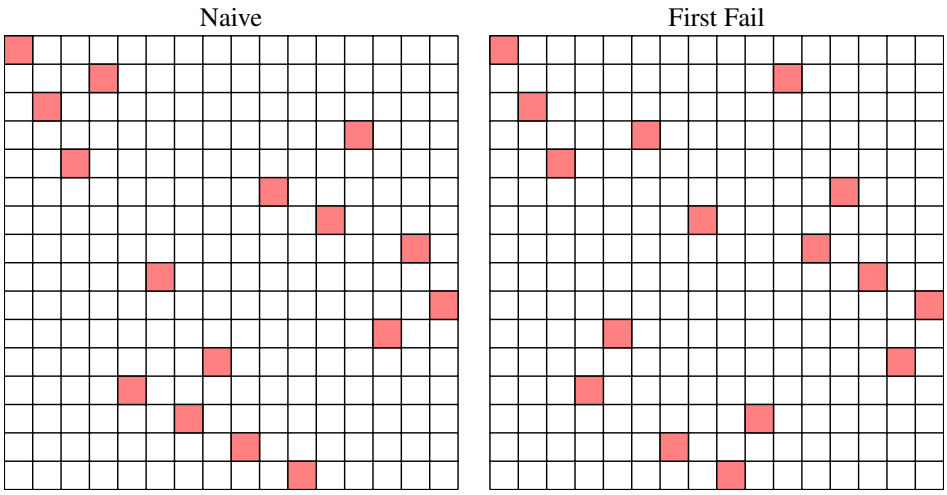
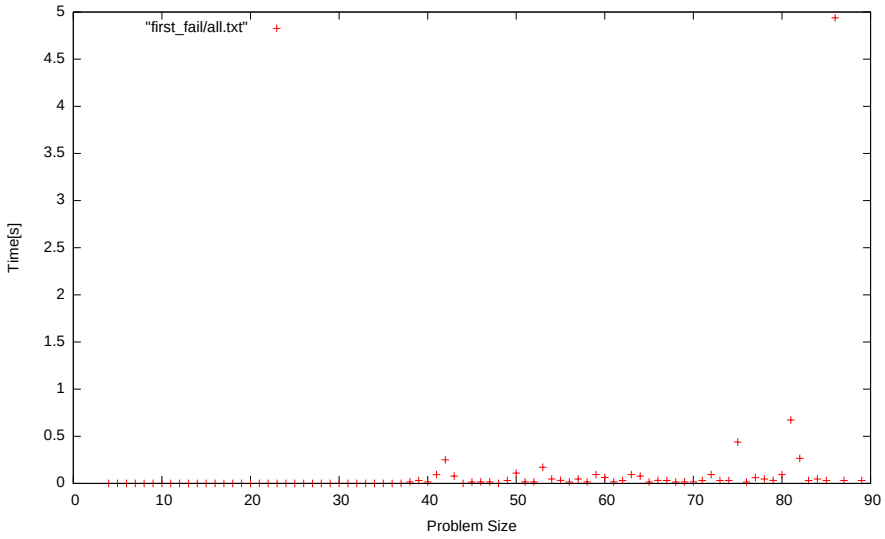


Figure 12: FirstFail, Problem Sizes 4-100



4.2 Improved Heuristics

We will now discuss how we can improve our program even further.

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

One way of doing this is to use an improved initial variable ordering. Before, we started with the variables on the left edge of the board. But they are not the most difficult choices to be made. Placing a queen in a corner has a limited impact on the other variables. If a queen is placed in the middle of the board, then its impact is much larger, it removes more values from the domains of other variables. As we have to assign these variables eventually, it makes sense to assign them at the beginning of the search, reducing the remaining problem size.

We can combine this with a matching value selection strategy, by assigning queens to the center of the board first. Every value must be used eventually, it is a good idea to make the hard choices in the beginning, and leave easier alternatives to the end. The `indomain_middle` method is ideal for this purpose, it starts with values in the middle of the domain.

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,[]).
```

In order to do this, we have to modify our program, reordering the variables before the assignment. We use a predicate `reorder` to do this, and pass the rearranged list of variables to our search routine, using `indomain_middle` as the value selection method.

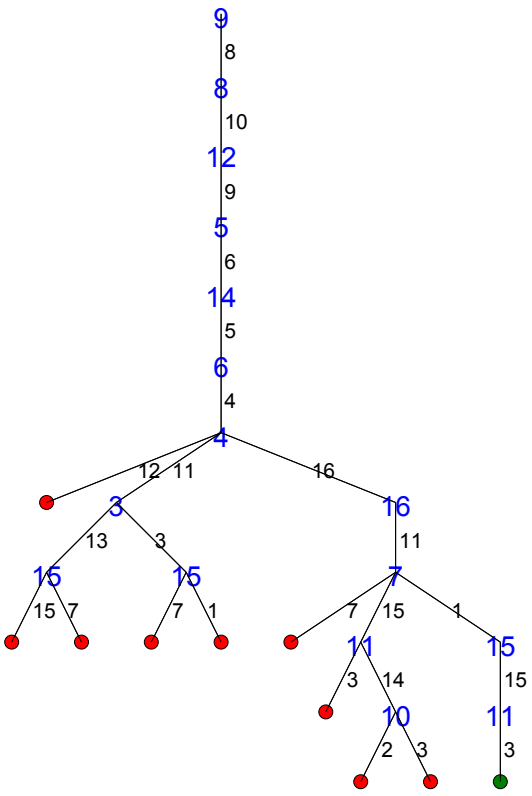
Reordering Variable List

```
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).
```

The code for reordering looks quite complex, it is a good exercise to check what is does exactly. The principle is that variables in the middle of the initial list are now at the beginning, and variables at either end are at the end of the list.

Figure 13: Start from Middle (Size 16)



For size 16, the change does not really pay off. Figure 13 shows the search tree, very slightly larger than before.

And again, we find a different first solution (Figure 14). We really have to check all problem sizes to see if the modification makes a systematic difference.

Figure 15 shows that while the new method is not always faster than the first fail selection, it is very good. We now have a timeout only for problem size 94, and size 75 is the only other instance which requires more than a fraction of a second.

Still, it is disturbing that the program shows this behaviour, and the remaining improvements we will consider are all about making the program more stable, not just faster on some instances.

Figure 14: Again, solutions are different!

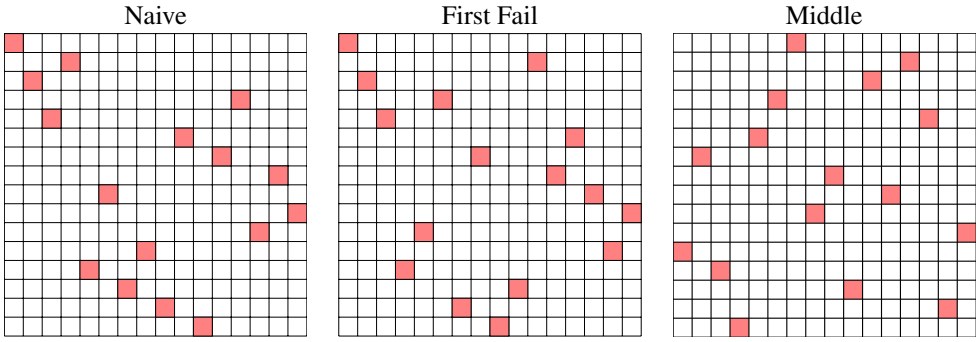
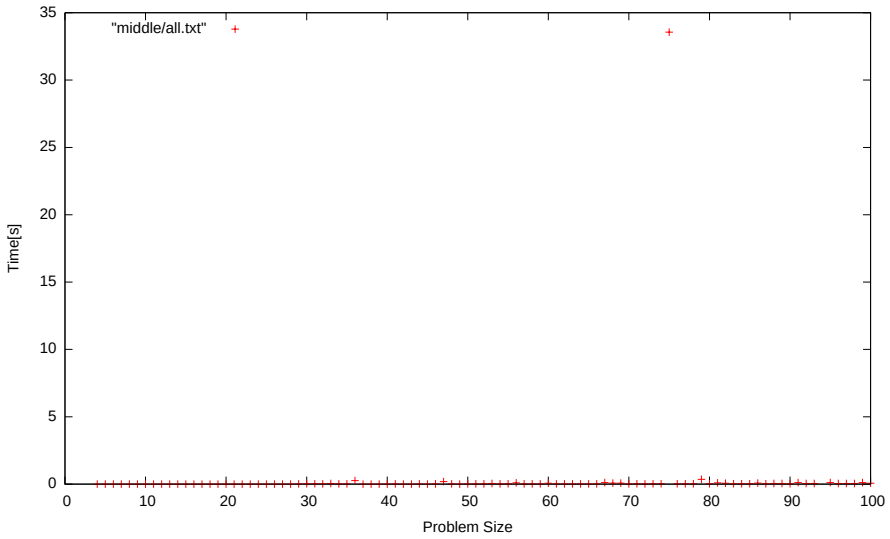


Figure 15: Middle, Problem Sizes 4-100



4.3 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

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

- 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

One example of a partial search scheme is the *credit based search*, which we are going to use on our N-Queens problem. In credit based search, we are given a fixed amount M of credit at the root of the search tree. Each node that we explore consumes one unit of credit, and we split the remaining credit between the children of the node. In this, we might give more credit to promising children, i.e. those which are preferred by a heuristic.

When the credit runs out, we start a *bounded backtracking search* which uses a maximum of K backtracks. When these backtracks run out, the search gives up on this part of the tree, jumps back to the unexplored top part of the search tree and continues there using up all credit. This means that we perform at most M bounded backtrack searchs with up to K backtrack steps, and at most M choices in the top of the 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),[]).
```

In order to use this credit based search, we only have to replace the `complete` keyword in the `search` predicate with the term `credit(N,5)`, which allows N units of credit and 5 backtracks in each bounded backtracking branch.

The parameter values are not critical, it is commom to use N or N^2 units of credit, and to allow between 5 and 20 extra backtracking steps in each branch.

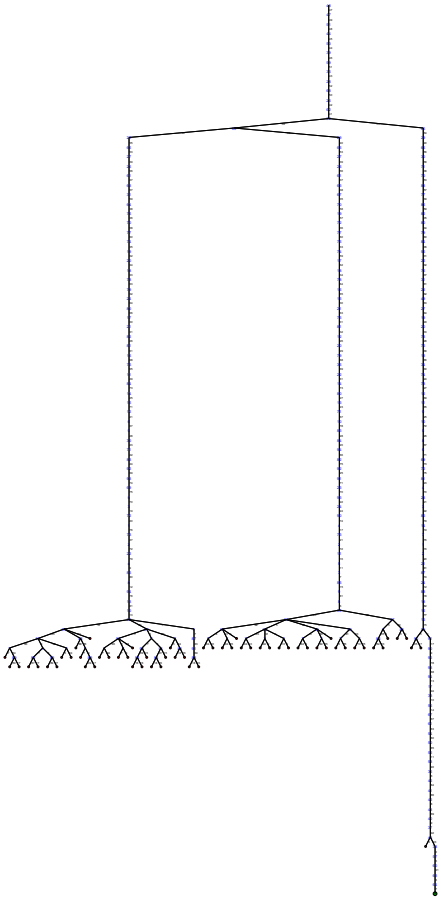
We can see (in Figure 16) the typical behavior of the credit based search. We start with an initial branch, which at some point starts backtracking. We give up on this branch, jump back to the unexplored part of the credit search and choose another branch there. This may lead to a failure again, but 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.

But this does not happen for the N queens problem, we now find solutions for all problem sizes up to 100 in a fraction of a second (Figure 17). We can see that the execution times are clearly layered. For some instances, we find a solution nearly without any backtracking in our first branch, for others we need two, three or multiple branches.

Figure 16: Credit, Search Tree Problem Size 94



We can continue to run the program for larger board sizes, this (Figure 18) 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.

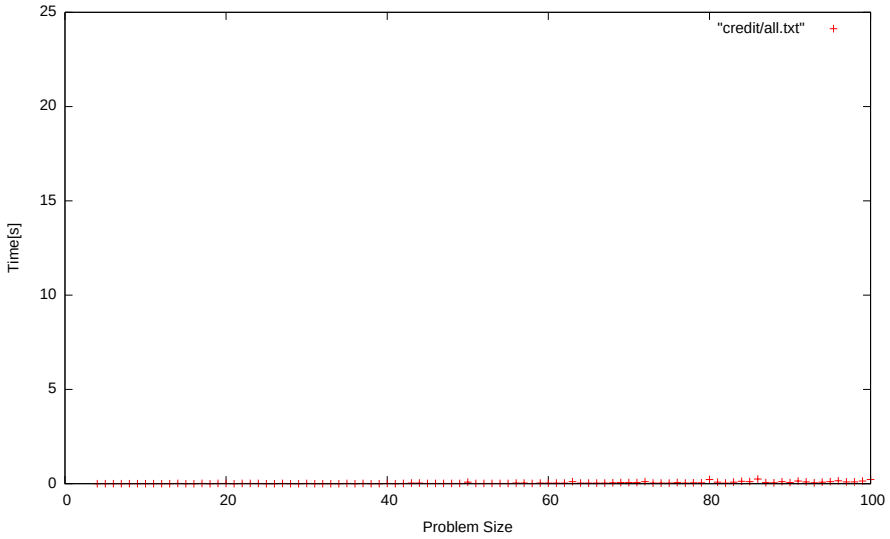
Conclusions

- 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

We have seen that even for a rather simple problem like the N-Queens puzzle the choice of the search method is very important, and can have a huge impact on the performance of a model. A dynamic variable selection method can reduce the search tree significantly, by exploiting information about the current state of the problem.

The `search` builtin provides an interface where we can specify many different search methods quite easily, so that we can experiment with problem instances without too much programming effort. It is important to consider a set of example instances for a problem to really understand the impact of a particular search strategy and check that we are not just lucky to find a solution for one particular data set.

Figure 17: Credit, Problem Sizes 4-100



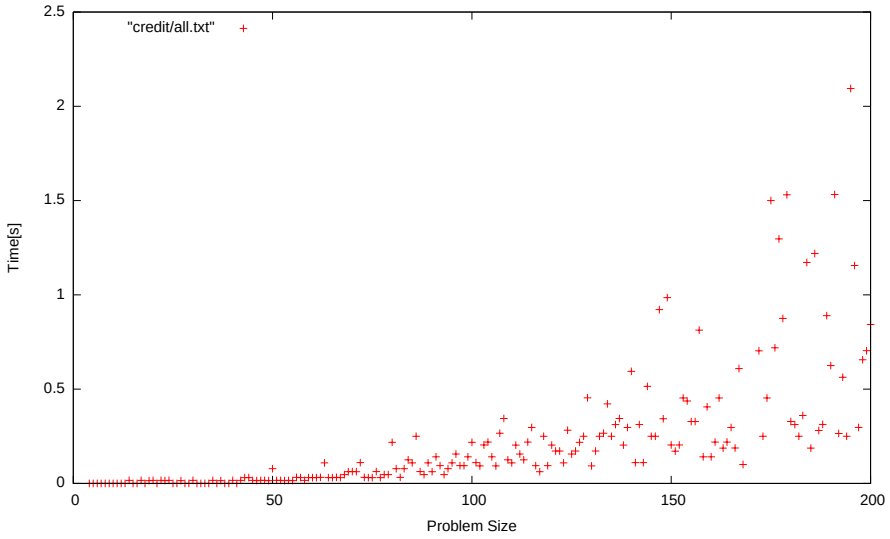
Outlook

- 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

For the N-Queens puzzle, finite domains work reasonably well up to a board size of perhaps 1000, the limiting factor is memory consumption, not execution time. Every assignment changes the domain of every other, unassigned variable. We need to store (*trail*) all these changes, so that we are able to backtrack to each choice point if required. But this uses quadratic memory.

As we said in the beginning, N-Queens is not a hard problem, so results obtained here must be considered carefully. But it provides a good demonstrator for the importance of search in constraint programs, and the variety of results that can be obtained by clever changes in the strategy.

Figure 18: Credit, Problem Sizes 4-200



5 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?