# Interactive Heuristic Search Algorithm

Tomáš Müller

Charles University
Department of Theoretical Computer Science
Malostranské náměstí 2/25, Praha 1, Czech Republic
muller@kti.mff.cuni.cz

**Abstract.** In this paper we present a hybrid heuristic search algorithm for constraint satisfaction problems. This algorithm was proposed as a mixture of two basic approaches: local search and backtrack based search. One of its major advantages is interactive behaviour in the sense of changing the task during the search. We present the results using the well known n-queens problem.

**Keywords:** Forward Based Search, Local Search, Heuristic Search, N-queens Problem

## Introduction

A finite constraint satisfaction problem (CSP) consists of a set of variables associated with finite domains and a set of constraints restricting the values that the variables can simultaneously take. A solution to a CSP is an assignment of a value from its domain to every variable, in such a way that every constraint is satisfied [1, 4, 6, 10].

In the current personal computing environment, the interactive behaviour of software is a feature requested by many users. Interactivity manifests itself in two directions: the user can observe what is happening inside the system (this is important for debugging but the final users like the animation of the problem solving process as well) and he or she can immediately influence the system.

Naturally, this requires a special interactive solving algorithm that is building the solution step by step and that can present a (sub-) result at anytime. A frequent requirement is to have a sound solution or sub-solution even during the search. In such a solution some variables can still be unassigned, but all constraints over assigned variables have to be satisfied. It means, that there is no violated constraint during the search. The reason can be for example a better visualisation of the sound partial solution than visualisation of a complete inconsistent solution [8].

Moreover, such algorithm can be exposed to a task change, so it should not restart solving from scratch after such a change but the new solution should be built on top of the former solution. By the task change we mean that the user can assign or un-assign a value of the variable manually, he/she can alter some other parameters like a domain of a variable, introduce a new variable or constraint etc.

It may seem that local search algorithms are best suited for this type of problems but note that typically local search is exploring the space of complete solutions which are not sound and it reduces the number of violations. However, we require a sound (perhaps incomplete) solution to be presented to the user every time. On the other hand, backtracking based search, which is a method of extending sound but incomplete solutions to a complete solution, does not support interactivity in the sense of changing the task during search. Therefore we propose a mixture of both approaches that borrows techniques both from backtracking based search (extending a partial solution to a complete solution and value and variable selection criteria) as well as from local search (tabu list to prevent cycling).

In this paper we describe an interactive algorithm to solve finite CSP problems. For demonstrating qualities of the proposed algorithm we compare it with the minimal conflicts random walk algorithm and the backtracking algorithm on the well-known N-queens problem.

The presented algorithm is a generalization of an ad-hoc algorithm we designed for solving timetabling problems in order to satisfy the need of interactivity. Description of this timetabling algorithm and some achieved results on a real university timetabling problem are presented in [8, 9].

## Basic Concept

Let us first summarise the requirements to such an interactive solving algorithm. The algorithm should provide some (partial) solution at each step. This (partial) solution must be sound (all required constraints are satisfied) and it should be good enough in respect to satisfaction of soft constraints (if used). We insist more on the interactivity of the algorithm. That means two things: the difference between the solutions provided by the algorithm in two consecutive steps should be minimal and the user can interfere with the solution process. The user may change domains of variables, add or remove variables, and adjust or detach any variable. The algorithm must be able to start from such a modified solution, make it sound and continue in extending the partial solution to a complete solution.

There are two basic approaches how to solve problems defined by means of constraints: backtracking based search that extends a partial sound solution to a complete solution and local search that decreases the number of violations in a complete solution. The idea of local search follows our requirement that there is some solution provided at each step and the difference between the solutions in consecutive steps is rather small. However, the provided solutions are not sound. On the other side backtracking based search can provide a partial sound solution at each step but it is not easy to add interactive behaviour to it. In particular, it does not support external changes of the partial solution and the difference between two consecutive solutions can be large after a long backtrack.

To satisfy the needs of an interactive solving algorithm we propose to mix features of these two approaches. Our algorithm uses forward based search that extends a partial sound solution to a complete solution. The next solution is derived from the previous one by assigning some variable and un-assigning conflicting variables from the partial solution. Every step can be guided by a heuristic. If the user changes somehow the partial solution, the algorithm first makes it sound by un-assigning the violated variables and then it continues from the given solution.

## Algorithm

We propose an interactive algorithm that works in iterations. The algorithm uses two basic data structures: a set of variables that are not set (assigned) and a partial sound solution. At each iteration step, the algorithm tries to improve the current partial solution towards a complete one. The algorithm starts with an empty partial solution (all variables are unbound), i.e. all the variables are in the list of un-assigned variables. Then it goes repeatedly from one partial sound solution to another sound solution until all the variables are set or the maximum number of iterations is reached.

The user may interrupt the algorithm after an arbitrary iteration and he or she can acquire a solution (latest or the best ever found) where perhaps not all the variables are set but all the constraints on the assigned variables are satisfied. During this interruption, the user may assign manually some of the un-assigned variables, weaken some constraints or make other changes (add new variables etc.) and then let the solver continue.

Let's have a look at what is going on in an iteration step. First, the algorithm selects an un-assigned variable and chooses its value. This selection of a value from the selected variable's domain can be made by evaluating each value from the domain via a heuristic function operating over the current partial solution. Finally, even if the best value is selected (whatever the best means), it may cause some conflicts with already assigned variables. Such conflicting variables are removed from the solution (become unassigned) and they are put back to the list of non-assigned variables.

```
procedure solve(unassigned, solution, max_iter)
  // unassigned is a list of un-assigned variables
  // solution is a partial solution (empty at the beginning)
  //     e.g. a list (variable, assigned value)
      iterations=0;
      while unassigned non empty & iterations<max_iter
            & non user interruption do
              iterations ++;
              variable = selectVariable(unassigned, solution);
              unassigned -= variable;
              value = selectValue(variable, solution);
              unassigned += assign(solution, variable, value)
                 // assign the variable and return violated variables
      end while;
      return solution;
end solve
```

**Fig. 1.** A kernel of the presented interactive algorithm

The above algorithm schema is parameterised by two functions: the variable selection and the value selection. Note that value/variable selection functions can be found both in backtracking based search as well as in local search algorithms and there are some general techniques used to define these functions. Our view of these functions is somewhere in-between backtracking and local search. We are selecting a non-assigned variable and during this selection we can use information about the previous values of the variable. Remember that the variable might have been removed from the solution during some previous iteration step. Moreover, because of removing variables from the partial instantiation we may need some mechanism to prevent cycles.

## Variable Selection Criterion

As mentioned above, the presented algorithm requires a function that selects an unassigned variable to be assigned in the current iteration step. This problem is equivalent to what is a variable selection criterion in constraint programming. There are several guidelines how to select a variable. In local search the variable participating in the largest number of violations is usually selected first. In backtracking based algorithms, the first-fail principle is often used, i.e., a variable whose instantiation is most complicated is selected first. This could be the variable involved in the largest set of constraints or the variable with the smallest domain etc. [1, 2, 5, 6, 7, 10]

It is possible to select the worst variable among all unassigned variables but due to complexity of computing the heuristic value, this could be rather expensive in some cases. Therefore we can select a subset of unassigned variables randomly (in timetabling algorithm [9] we use a probability of selection 0.2) and then choose the worst variable from this subset. The results will not be much worse and we can select the variable much faster.

## Value Selection Criterion

After selecting the variable we need to find a value to be assigned. This problem is usually called “value selection” in constraint programming. Typically, the most useful advice is to select the best-fit value [1, 2, 5, 6, 7, 10]. So, we are looking for a value, which is most preferred for the variable and also which causes the least trouble. It means that we need to find a value with minimal potential future conflicts with other variables. Note that we are not using constraint propagation explicitly in our algorithm, the power of constraint propagation is hidden in the value selection (it roughly corresponds to a forward checking method).

To remove cycling, it is possible to randomise the value selection procedure. For example, it is possible to select five best values for the variable and then choose one of them randomly. Or, it is possible to select a set of values so that the heuristic evaluation for the worst value in this group is maximally twice as large as the heuristic evaluation of the best value (when lower value means better value). Again, the value is selected randomly from this group. This second rule inhibits randomness if there is a single very good value.

### How To Escape A Cycle?

In the previous two sections about the value and variable selection we proposed some mechanisms how to avoid cycling, but we can do more. In our timetabling algorithm [9] we proposed a technique based on a tabu list [1, 3, 10].

Tabu list is a FIFO (first in first out) structure of pairs (variable, value) with a fixed length. When a value V is assigned to the variable X, a new pair (X,V) is added to the end of the tabu list and the first pair is removed from the tabu list. Now, we can avoid a repeated selection of the same value by applying a tabu rule which says: if the pair (X,V) is already in the tabu list then do not select the value V for X again (until the pair disappears from the tabu list). The tabu rule prevents short cycles (a cycle length corresponds to the length of the tabu list) and it can be broken only via so called aspiration criterion. If the aspiration criterion is satisfied for a pair (X,V) then V can be assigned to X even if the pair (X,V) is in the tabu list.

### Soft Constraints

Sometimes, the CSP problem is generalized using hard and soft constraints [6, 10]. In the solution, all the hard constraints have to be completely satisfiedwhile the soft constraints (preferences) do not need to be satisfied. We only require the number of violated soft constraints to be minimal (or e.g. weighted sum of violated constraints). Moreover, the user typically does not need an optimal solution (with the minimal number of violated constraints). A close to optimal solution is sufficient.

Soft constraints can also be used in our proposed algorithm. The accomplishment of soft constraints can be introduced into the solver via variable and value selection criteria. We can for example determine a quality of some value according to how many soft constraints are violated.

## N-queens Problem

In this section we will present the efficiency of the described forward search algorithm on the N-queens problem. We will compare the achieved results with another local search and backtrack based algorithm. All these algorithms were implemented in Java and all presented results were measured on Intel Pentium III 1GHz, 256MB RAM, Windows 2000, JDK 1.4.0. .

The N-queens problem is to place $n$ queens on a $n \times n$ chessboard so that no queen is under a direct attack from any other one. This problem can be made more difficult to solve by setting some prohibited fields on chessboard (N-queens problem with holes). None of the queens can be placed on these fields.

At first, we need to model the N-queens problem as a CSP problem:

- variables $\{Q_1, Q_2, \cdots Q_n\}$ represent the queens,
- domains $Q_i \in \{1,2,\cdots n\} \quad \forall i \in \{1,2,\cdots n\}$,

where equality $Q_i = j$ determines the *i*-th queen is placed on the *i*-th column and *j*-th row (note, that each queen strictly determines the column where it is placed),

- constraints

  $Q_i \neq Q_j \quad \forall i, j \in \{1,2,\cdots n\},\ i \neq j$ … condition for rows,

  $|Q_i - Q_j| \neq |i - j| \quad \forall i, j \in \{1,2,\cdots n\}, i \neq j$ … diagonals,

  $Q_i \neq j \quad \forall (i, j) \in H$ … prohibited fields (holes),

  where *H* is the list of holes – pairs (*column*, *row*).

Prohibited fields constraint can be directly propagated to the queens' domains.

## Presented Forward Search Algorithm

As described above, our algorithm will work in iterations. In each iteration step, it selects a queen, which is not placed (equals to a selection of a column with no queen). Next, it selects a location for the queen (a row, where the queen will be placed) and finally it removes all queens from the chessboard, which conflict with the currently placed queen. This process is repeated until all queens are placed.

Now, to complete the description of the algorithm, we need to define the variable and the value selection criteria:

- The queen, with the maximum number of holes in the queen's column is selected (first-fail principle). When there are more than one such queens the queen with the maximum sum of conflicts in all possible positions (values) from them is selected. If there are still more than one such queens, one of them is selected randomly.
- The queen is placed to the location, where the number of conflict queens is minimal (best-fit principle). If there are more than one such place, one of them is selected randomly.

Tabu search or another feature to prevent cycling is not used – it's not necessary. Number of conflicts on the chessboard fields (equals to how many queens "see" the field) is updated when a queen is placed or removed from the chessboard. Selection of the queen from some subset of not placed queens is not used – all the not placed queens are assumed when selecting the queen to be assigned next.

## Minimal Conflicts Random Walk Algorithm

Minimal Conflicts (or Min-Conflicts) algorithm is one of the typical local (neighbour) search algorithms and it is highly efficient on the N-queens problem. It also works in iterations. In each iteration step the algorithm firstly chooses randomly a conflict variable, i.e., a variable that is involved in an unsatisfied constraint, and than picks a value, which minimises the number of violated constraints.

Because the pure min-conflicts algorithm cannot go beyond a local-minimum, some noise strategies were introduced into it. Among them, the random-walk strategy becomes one of the most popular. For a given conflicting variable, the random-walk strategy picks randomly a value with probability *p*, and applies the min-conflicts heuristic with probability *1-p*.

So, one of the not placed queens is selected randomly at first. For the selection of its new location we use the same strategy as in our presented algorithm:

- The queen is placed to the location, where the number of conflict queens is minimal. If there are more than one such places, one of them is selected randomly.

A random location for the queen is selected with probability 2%.

Note, that the major difference between this algorithm and the above presented one can be seen in not removing conflict queens from the chessboard at the end of each iteration step.

This algorithm starts from an inconsistent solution, where all queens are placed. So, we can either select location for all queens randomly (marked as random start in figures below) or we can use the above heuristics for choosing the location for each queen step by step (heuristic start).

### Backtrack Based Search Algorithm

The last algorithm we used for comparison is the backtrack-based search. Queens are ordered dynamically according to the heuristics described above (queen with maximum number of holes and conflicts), full the look-ahead technique is used.

## Results

In this section we present some comparisons of presented algorithm with min-conflicts random walk and backtracking algorithm on the above described N-queens problem (and N-queens problem with holes - prohibited fields).

The time and the number of iterations needed for solving the N-queens problem (without holes) is given in the tables below (figures 2 and 3). It is not surprising that the local search algorithms are much more efficient than backtrack based search on this problem. So, let's focus only on the differences between the presented forward search algorithm and the minimal conflict random search algorithm. As we can see on figure 2, the presented algorithm is nearly as fast as the min-conflicts algorithm which starts from a chessboard with heuristically placed queens.

| **number of queens** | **presented algorithm** | **min-conflicts random start** | **min-conflicts heuristic start** | **backtracking** |
|---:|---:|---:|---:|---:|
| **100** | 8 ms | 6 ms | 8 ms | 113 ms |
| **500** | 126 ms | 182 ms | 90 ms | 2236 ms |
| **1000** | 558 ms | 1009 ms | 480 ms | 25422 ms |
| **2000** | 2 375 ms | 4 558 ms | 2 097 ms | --- |
| **5000** | 16 031 ms | 31 369 ms | 13 305 ms | --- |

**Fig. 2.** Time to solve the N-queens problem (no holes); mean value from 5 measurements

| number of queens | presented algorithm | min-conflicts random start | min-conflicts heuristic start |
|---|---|---|---|
| **100** | 120 | 127 | 73 |
| **500** | 528 | 421 | 37 |
| **1000** | 1017 | 729 | 61 |
| **2000** | 2022 | 1403 | 102 |
| **5000** | 5026 | 3266 | 70 |

**Fig. 3.** Number of iterations needed to solve the N-queens problem (no holes) mean value from 5 measurements

Comparison of the number of iterations needed to solve the N-queens problem is shown in figure 3. Note, that the proposed forward search algorithm starts from the chessboard, where no queen is placed, so it needs at least as many iterations as the number of queens on the chessboard. This dependence is also shown in figure 4.

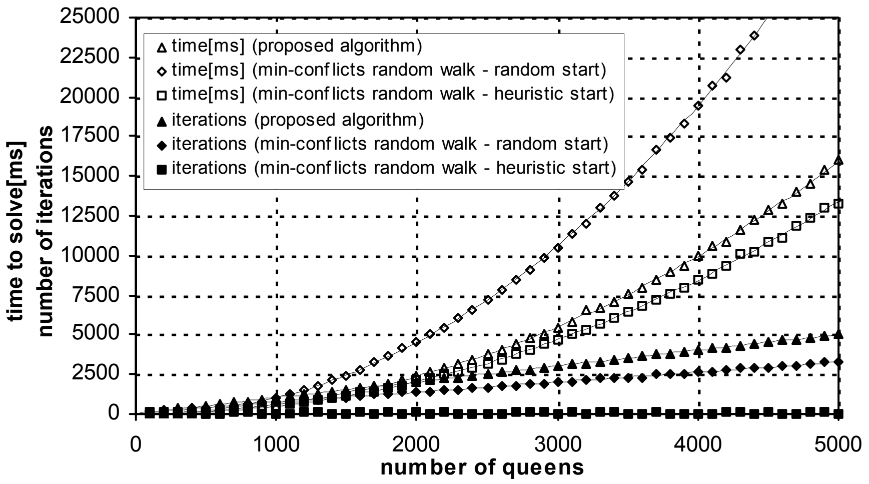

**Fig. 4.** Comparison of the number of iterations and time for the described algorithms mean value from 5 measurements

Very interesting comparison is presented at figure 5. The number of "repair" iterations is compared there. It is the number of iterations needed to solve the given problem decreased by the number of queens (for presented forward search algorithm). In the min-conflicts algorithm, the number of "repair" iterations equals to the number of iterations needed to solve the given problem. Note, that it is the number of iterations, which is needed to convert heuristically selected starting (inconsistent) solution to some consistent solution. It seems that this number does not depend on the number of queens (size of the problem) for both algorithms. Moreover, the diffusion of this number of "repair" iterations is lower for our algorithm (mean value of the number of "repair" iterations for presented algorithm is 25.0, for min-conflicts random walk algorithm it is 64.1).

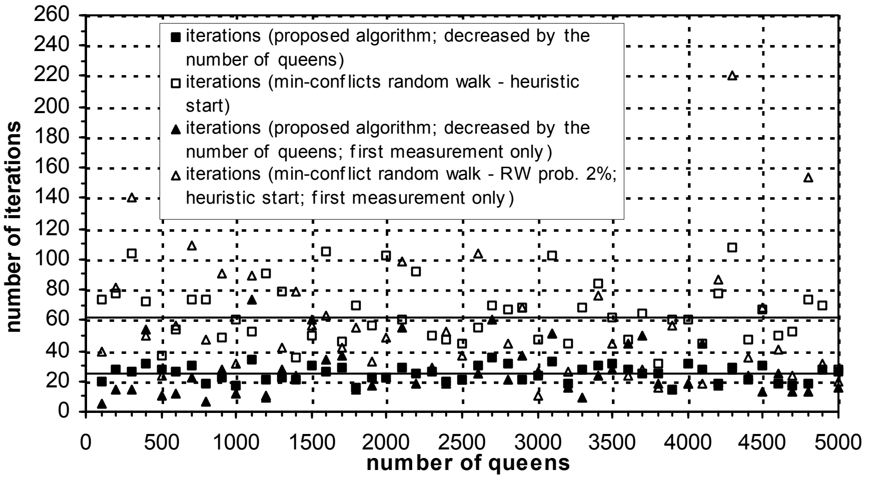

**Fig. 5.** Comparison of the number of "repair" iterations for the described algorithms mean value from 5 measurements, first measurement

Finally, we can compare the time and the number of iterations for the described algorithms when holes (prohibited fields) are introduced into the N-queens problem (see figure 6.).

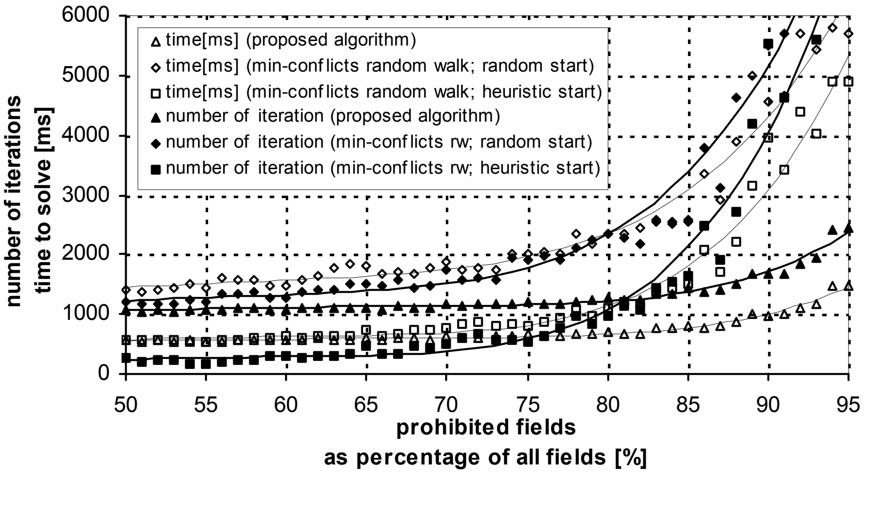

**Fig. 6.** Comparison of the number of iterations and time for the described algorithms; N-queens problem with prohibited fields; mean value from 5 measurements; 1000 queens

The presented algorithm seems to be much more robust when there are more than 80% of all fields on the chessboard prohibited. The input chessboard is generated so, that there always exists a solution.

As it is shown above, the presented algorithm seems to be very efficient on the N-queens problem, even if this problem is tighten by adding prohibited fields (measured up to 95% of all fields). Indeed, the efficiency of the presented algorithm highly depends on the good choice of the variable and value selection criteria.

## Conclusion

We presented a promising algorithm for solving finite constraint satisfaction problems, which combines principles of the local search with other techniques for solving constraint satisfaction problems. The basic motivation was to design a generic algorithm with interactive features for solving university timetabling problems.

The efficiency of the proposed forward search algorithm on the N-queens problem was presented as well. We believe that this algorithm can be successfully applied to many constraint satisfaction problems especially when the interactive behaviour is required or when working with sound incomplete solutions is needed in general.

## References

[1] R. Barták. *On-line Guide to Constraint Programming*, http://kti.mff.cuni.cz/~bartak/constraints, 1998.

[2] D. Clements, J. Crawford, D. Joslin, G. Nemhauser, M. Puttlitz, M. Savelsbergh. *Heuristic optimization: A hybrid AI/OR approach.* In Workshop on Industrial Constraint-Directed Scheduling, 1997.

[3] P. Galinier and J.K. Hao. *Tabu search for maximal constraint satisfaction problems*. In Proceedings of CP'97, G. Smolka ed., LNCS 1330, pp. 196-208, Schloss Hagenberg, Austria, Springer, 1997.

[4] P. Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1998

[5] J.-M. Labat, L. Mynard. *Oscillation, Heuristic Ordering and Pruning in Neighborhood Search*. In Proceedings of CP'97, G. Smolka ed., LNCS 1330, pp. 506-518, Schloss Hagenberg, Austria, Springer, 1997.

[6] K. Marriot, P. J. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998

[7] Z. Michalewicz, D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2000.

[8] T. Müller and R. Barták. *Interactive Timetabling*. In Proceedings of the ERCIM workshop on constraints, Prague, 2001.

[9] T. Müller and R. Barták. *Interactive Timetabling: Concepts, Techniques, and Practical Results*. Submitted to PATAT'02, Gent, 2002.

[10] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993