Explanations and optimization in preference-based configurators

M. Moretti, F. Rossi
Department of Pure and Applied Mathematics,
University of Padova, Italy
mmoretti@studenti.math.unipd.it, frossi@math.unipd.it

E. C. Freuder, C. Likitvivatanavong, R. J. Wallace
Cork Constraint Computation Centre
University College Cork, Cork, Ireland
{e.freuder,chavalit,rjw}@4c.ucc.ie

Abstract

In this extended abstract we consider configuration problems with preferences rather than just hard constraints, and we analyze and discuss the features that such configurators should have. In particular, these configurators should provide explanations for the current state, implications of a future choice, and also information about the quality of future solutions, all with the aim of guiding the user in the process of making the right choices to obtain a good solution.

We then show our implemented system, which, by taking the soft n-queens problem as an example, shows that it is indeed possible, even in this very general context of preference-based configurators, to automatically compute all the information needed for the desired features. This is done by just keeping track of the inferences that are made during the constraint propagation enforcing phases. However, the process of recording the necessary information is more complex and expensive than in the hard case, since in general a sequence of previous choices has to be recorded for each possible future choice, and not just one as in the hard case.

1 Configuration problems

A configurator can be regarded as a system which interacts with a user to help him/her to configure a product. A product can be seen as a set of component types, where each type corresponds to a certain finite number of concrete components, and a set of compatibility constraints among subsets of the component types.

Moreover, during the interaction with the configurator, the user incrementally posts some "personal" constraints (which may be different from user to user) over the component types. Sometimes the component types are not all known in advance, but arise only after the user has posted some of his/her personal constraints.

A configuration of a product is a choice of a concrete component for each component type, such that all the compatibility constraints and personal constraints are satisfied.

A typical example of this scenario is the problem of configuring a PC by choosing all its components. In this case, a compatibility constraint may say that some mother-boards are not compatible with some processors, and a personal constraint could say, for example, that only IBM hard disks are desired.

Another example may concern the choice of a car, via the selection of all its features (engine, chassis, color, ...). In this case, a compatibility constraint could say that only some engines are available for some chassis, and a personal constraint could say that the car should be red, or that the chassis should be convertible.

A third example, where the component types are not known at the beginning, is a PC configuration problem where the user can choose either a combo CD/DVD driver or two single components. In this case, only after the user has said if he wants a combo system or two components, new component types (one or two) will appear.

Desired features for configurators. In the interaction between a configurator and a user, the user usually makes successive choices over the component types, specifying in this way his/her personal constraints, and the configurator should help him/her in several respects, mainly in trying to avoid conflicting situations, where the user choices and the compatibility constraints are not consistent. In fact, such conflicting situations would not lead to any complete product configuration. Examples of desired help from the configurator are:

- the configurator should show the consequences of any possible future user choice: this can help making the next choice in a way that avoids conflicts both at the next step and also at the subsequent steps;
- as soon as a conflict arises, the configurator should show why such a conflict has arisen, and what choice(s) could be retracted to come back to a non-conflicting situation;
- a user choice could make some other choices unfeasible (because of the compatibility constraints): such unfeasible choices should be shown and justified to the user.

Constraint-based configurators. Constraint-based technology is currently used in many configurators to both model and solve configuration problem [7, 1]. In this approach to configuration, component types are represented by variables, with as many values as the concrete components, and both compatibility and personal constraints are represented as constraints over subsets of such variables.

In particular, user choices during the interaction with the configurator are most of the times restricted to be unary constraints, which select a certain value for a variable. However, they could also be more general non-unary constraints.

Whenever a choice is made, the corresponding constraint is added to the compatibility and personal constraints already there, and some constraint propagation notion is enforced, for example arc-consistency [10], to rule out (some of the) future choices which are not compatible with the current choice. While providing justification based on search is difficult, arc-consistency enforcing has been used as a source of guidance for justifications, and it has been exploited to help the users in some of the scenarios mentioned above. For example, in [4], it is shown how AC enforcing can be used to provide both justifications for choice elimination, and also guidance for conflict resolution. Other related approaches to provide justifications and explanations can be found in [5, 11].

2 Preferences in configurators and constraints

In this paper we follow the same approach as in [4], in the sense that we use constraint-based technology in the context of configuration problems and we provide explanation and conflict-resolution help via AC enforcing. However, we do that in the more general context of preference-based configurators. In fact, the configurator scenario we have described in the previous section assumes that both the compatibility constraints and the user constraints are hard, that is, they are either satisfied or violated.

In general, however, in many real-life applications, both these kinds of constraints may instead be just preferences, thus allowing for more than two degrees of satisfaction. For example, in a car configuration problem, a user may prefer red cars, but may also not want to completely rule out the other colors. Thus red will get a higher preference with respect to the other colors. In a PC configuration problem, some combinations of mother-boards and processors may be preferrable to others.

Soft constraints. Preferences, or *soft constraints* as we may call them, can easily be modelled as classical constraints where each combination of values for the variables is associated to an element, which is usually interpreted as the preference for that combination, or its importance level [2]. Note that variable domains can be seen as unary constraints, thus it is also possible to associate preferences to single variable values. The elements which are used as preference levels have to be ordered, in order to know which elements represent higher (or lower) preferences.

Given a set of soft constraints, a complete variable assignment inherits its preference level by combining the preference levels it selects in the constraints. Solutions are therefore ranked by their preference level, and thus soft constraint problems are constraint optimization problems, where one usually looks for an optimal solution, that is, a solution which has the highest preference level (according to the ordering of the preference levels).

For example, a typical soft constraint scenario, that we will use extensively in this paper, is the fuzzy constraint framework [8, 9, 6, 3], where preferences are real numbers between 0 and 1, with a higher number denoting a higher preference, and are combined via the minimum operator. Thus the preference value of a solution is the minimum among the preference values selected by this solution on the constraints, and the best solutions are those with maximum preference value.

Another example is the one where one wants to minimize the sum of the costs (which are natural numbers). Note that even classical hard constraints could be seen as soft ones, where there are just two levels of preference (allowed and not-allowed), with allowed better than not-allowed, and with logical AND as the combination operator.

Notice that more complex structures and orderings can be used in this context, to provide a more sophisticated way of comparing solutions. For example, one may want to both maximize the minimum preference and also to minimize the sum of the costs. In this way, solutions will be partially ordered according to these two criteria.

Search and AC enforcing in soft constraint problems. While searching for a solution can be done via a backtracking search, interleaved with constraint propagation, searching for an optimal solution requires a branch-and-bound approach[1], which again can be interleaved with constraint propagation. In fact, the presence of soft constraints, in some cases like the fuzzy one, does not preclude the use of constraint propagation algorithms.

However, there is a fundamental difference beetween constraint propagation in the hard case and in the soft one. Let us consider AC enforcing to make it simple, and also because we will use AC in this paper: while in the hard case AC enforcing eliminates values from variable domains, in the soft case it decreases their preference value.

In fact, in the hard case AC enforcing performs the following change over the domain D_i of variable x_i: $D_i := \{d \in D_i$ such that, for any other variable x_j, there exists $d_j \in D_j$ with $\langle d_i, d_j \rangle$ satisfying the constraint c_{ij} between x_i and $x_j\}$. Instead, soft AC enforcing over fuzzy constraints performs the following: for each $d_i \in D_i$, consider its preference value $p(d_i)$; then $p(d_i) := max_{x_j \in V, d_j \in D_j} min(p(d_i), p(d_j), p(\langle d_i, d_j \rangle)\}$. It is easy to see that this is just an apparent difference, because also value elimination can be seen as preference decrease: from allowed to not-allowed.

Summarizing, after enforcing AC in the soft scenario, we have an equivalent problem with possibly lower preferences in the variable domains.

Desired features for preference-based configurators. It is now clear that configurators with soft constraints should help users not only to avoid conflicts or to make the next choice in a way that a smaller number of successive choice are eliminated, but also to get to an optimal (or good enough) solution. More

[1] Or also a local search, but we will not consider it in this paper.

precisely, when the user is about to make a new choice for a component type, the configurator should show the consequences of such a choice in terms of conflicts generated, elimination of successive choices, and also quality of the solutions. In this way, the user can make a choice which leads to no conflict, and which presents a good compromise between choice elimination and solution quality.

As for the hard case, all this can be done via the help of the soft AC enforcing algorithm, which is able to remember what caused a certain preference decrease. Thus, when a user wants to know why a certain value has a certain preference, which is lower than the initial one, the configurator is able to justify that, and the justification is just a sequence of previous user choices.

Note that the configurator must remember a sequence of previous choices, and not just one, as in the hard case, because here preferences can be lowered at several stages. For example, from 1 to 0.7 because of a choice C1, then from 0.7 to 0.4 because of another choice C2. Thus, when a user wants to know why we have a current preference of 0.4, we must give him/her both C1 and C2 (in the correct sequence) as the reason. In fact, this is the whole history of the causes that produced the current preference. It is crucial to record the whole history, and not just the last preference change, since we must also be able to retract some steps and go back to previous states (see later).

Summarizing, the questions that a preference-based configurator must be able to answer are at least the following ones:

- Why has this configuration (or this component) such a preference level?
- Why did this sequence of choices lead to a conflict? What should I do to remove this conflict?
- What effect will this choice have on the future choices and on the quality of the solutions? In particular, are there choices that will lead to a conflict?

While the first two questions refer to a situation in which a user tries to understand the consequences of what has been done already, the last one tries to prevent undesired consequences by guiding the choices of the user towards good solutions and no conflict.

3 Our system

We developed a system which shows the above mentioned features for preference-based configurators in the context of a variant of the n-queens problem where fuzzy preferences have been added. The system is available at the URL
`http://www.studenti.math.unipd.it/~mmoretti`.

N-queens with preferences. In the n-queens problem, n queens must be placed on a n by n board in such a way that no queen can attack each other. Recall that a queen attacks any other queen which is on the same row, column, or diagonal.

This problem is only apparently far from the configuration scenario we have described above. In fact, each queen can be seen as a component type, and the concrete components for a type are the positions for the queen. Moreover, the attack rules for the queens are the compatibility constraints, and positioning a queen in a certain cell means choosing a certain concrete component for a component type. A solution to the n-queens problem is therefore a complete configuration where, for each component type, we have chosen a concrete component, in such a way that the compatibility constraints are satisfied.

Notice that, in this system, all the component types (that is, the rows) are known at the beginning. Moreover, the user choices (that is, the queen positionings) are just unary constraints.

In this paper we consider a variant of this classical n-queens problem to model preference-based configurators. In particular, we associate a preference to each cell of the board, and also to each pair of positions within an attack constraint. We use preference levels between 0 and 1, and we aim at maximizing the minimum preference, as in the fuzzy framework [8].

As usual, we have one variable for each row, with the n cells of the row as possible values, and thus constraints hold between two positions on the same column and on the same diagonal. Notice that this implies that there is no constraint between two positions on the same row.

Preference setting. In our system, the user can select any cell and he/she can assign to it the desired preference level (the default one is 1). For the constraints, we allow the user to give to each constraint a level of importance (always between 0 and 1, the default value is 1), from which we automatically obtain the preferences to be assigned to all the pairs of values (we recall that constraints are all binary in the n-queens problem). More precisely, the preference level of a pair will depend both on the level of importance of the constraint (the higher the importance level is, the smaller the preference is for pairs under attack), and on the distance between the elements of the pair (elements under attack which are farther apart are more tolerated by giving them a higher preference value). Figure 1 shows the graphical interface of our system, with the list of possible preferences for cell (4,5), and, on the right, the buttons for setting the importance level of the constraints (”Diagonal ur-bl”, which stands for the constraint over the diagonal from upper right to bottom left, ”Diagonal br-ul”, and ”Vertical”).

After the preference setting phase, the user can run an initial soft AC enforcing via the ”initial AC” button, which, as mentioned above, may possibly decrease the preferences of the cells.

Explanations for the cell preferences. At any moment, the cells of the board show their current preference level. Figure 2 shows a state of the system, where a queen has already been positioned, in cell (4,5), and where each cell shows its preference level. For example, cell (1,2) has preference level 0.3.

To ask for explanations about the current preference of a certain cell, the user

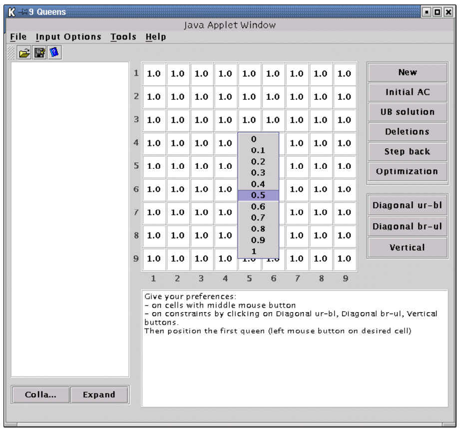

Figure 1: Graphical interface of our system, with the preferences for cell (4,5).

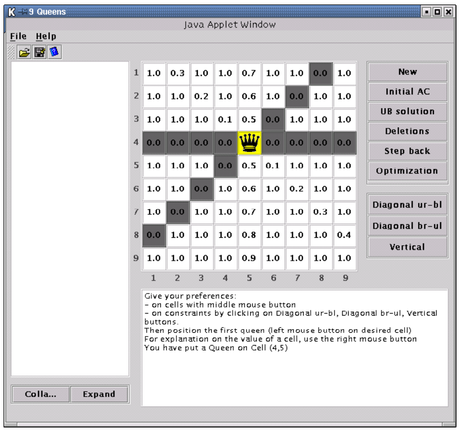

Figure 2: Each cell shows its current preference level.

can click on the cell with the rigth mouse button, and a sequence of past choices (that is, queen positions) will appear in the lower panel of the graphical interface. For example, in Figure 3 we have asked for the explanation of the preference of cell (6,5) (after three queen positionings), and we have got the following list of choices: `Queen(4,5) --> 0.6, Queen(3,2) --> 0.3` (see lower panel of the graphical interface). This means that the current preference for cell (6,5), which is 0.3, has been caused by the positioning of a queen in cell (4,5), which lowered the preference from 1 to 0.6, and by a later positioning of a queen in cell (3,2), which has lowered the preference from 0.6 to 0.3. In the same style as in [4], we can also show the list of explanations as a tree, in the left panel, as shown in Figure 3.

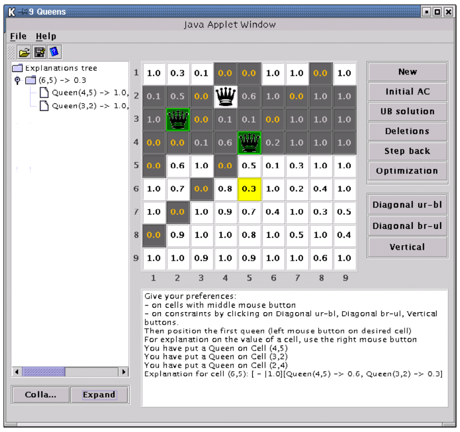

Figure 3: Explanation: list (lower panel) and tree (left panel).

This representation of the explanation for the preference level of a cell may help the user in both understanding why we are in a certain situation, and also in possibly retracting some previous choices, which are shown to be the causes for a certain low preference in some cells.

From the implementation point of view, to keep track of the explanations for each cell, we associate a list of queen positions and preference levels to each variable value (that is, a cell). At the beginning, such lists are empty. Whenever AC is enforced, if a cell, say C, gets a new lower preference, say p, because of a queen positioning, say Qij, the list for C is augmented with a new element, which contains the pair (Qij, p). Thus, at any moment, the list for cell C contains the sequence of queen positionings which are the causes for the preference changes, from the initial level to the current one.

Notice that the explanation generated for a cell does not depend on the way AC is performed (that is, the order in which AC considers the variables and their values), but only on the sequence of queen positionings.

Queen positioning. To position a queen on a cell, we just have to click on the cell with the left mouse button. This will put a queen in that cell, and will also automatically run AC again after the positioning. Thus the content of the cells, that is, their preference level, may be lowered after the positioning of the new queen. In particular, if the level of preference of a cell gets to 0 (the lowest value, which indicates a complete conflict), the cell is greyed and an x is shown in the cell (see Figure 4).

Once a queen has been positioned, its position can be changed. If we want to move a queen to another cell of the same row, we can just click with the left mouse button on the new position. We can also retract the position of the latest queen that has been put on the board, via the "step back" button on the right. By clicking on this button, the last queen disappears and the last run of AC enforcing is retracted. This means that we go back to the state before the queen positioning.

Some queens can also be positioned by the system, without any input from the user. This happens where there is no other way to position a queen without leading to a conflict. For the cells that have been automatically filled with a queen by the system, the explanation is the set of all other cells of the same row, which have all preference level 0, and which in turn have their explanation lists.

Changing state. In our system, a state of the computation is the current state of the board (queen positions and other cell information), plus the current explanation for each cell, which, as explained above, is a list of previous choices and preference levels. This state can be changed in several ways and for several reasons.

First, we can change the state by positioning a new queen. This means going forward towards a complete solution. Or, we may also go back, because we are not satisfied with the current state. In fact, we may not like the current state because it is a conflict state, that is, some row without a queen has all 0 preference values in its cells. Or also, we may want to change the state because we don't like the preference values of the cells in the current state. In both cases, we may want to change the state by either retracting the last step, or some previous step.

It is easy to see that, in a state change, the most expensive and complex operation is the update of the explanation lists. It is then easy to imagine that, if we want to retract the last choice, the state change is not so difficult nor expensive, since it just involves the deletion of some last elements from some lists. However, if we need to change the position of a queen which is not the last one to have been positioned, then the lists of explanations require careful handling. In fact, such a queen may appear in the middle of an explanation list, and thus its deletion requires the re-computation of everything that is after it in the lists.

In principle, to compute the new state after a queen move, we can truncate each list up to the element containing this queen, if any, and we redo the queen positionings which are contained in the lists in successive positions. In this way, we have the state obtained after all the previous choices, and in the same sequence, except the choice related with the queen that we have moved. After this, we position the queen in the new position.

Note however that is not what is actually done in the implementation, since there are several sources of optimization in recomputing the new lists. However, this may give an idea of the added complexity given by the introduction of the preferences. In fact, in the hard case, there is no need to keep a list of causes for the preference changes, since each cell can be subject to just one change: from allowed to not-allowed. Thus in the hard case we just need to remember at most one queen positioning as the cause for the not-allowed state of the cell. In the soft case, instead, we need to remember all the preference changes, and the queen causing each of them.

Deletions. As mentioned above, knowing the consequences of all future choices may be helpful in understanding which is the most convenient choice to take next. One of such consequences is the number of cells where it will not be possible to put any queen if we make a certain choice now. These are the cells that will get a preference value 0 if we position a queen in a certain position.

More precisely, for every cell without a queen, say C, we can compute the number of other cells which will be deleted if we put a queen in cell C. This is done by making a copy of the current state, by adding a queen in cell C to such a copy of the state, by running soft AC, and by counting the number of cells that, in the resulting state, get a 0 preference.

By clicking on the button "deletions" on the right, each cell C shows the number of deleted cells if we put a queen in C. Figure 4 shows the board after we have clicked on the deletion button. For example, we can see that cell (3,3) contains the number 10. This means that, by putting a queen in cell (3,3), ten new cells would not be allowed any longer.

This information can guide the user in making the next choice. For example, the user may want to choose the cell which generates the smallest number of deleted cells, hoping to avoid a conflict.

Upper bounds for solutions. When we have preferences, solutions are ranked by their preference level. Thus the aim is to obtain an optimal, or good enough, solution. The configurator should therefore guide the user towards good solutions.

This can be done by estimating the quality of the solutions which contain a certain queen. More precisely, for each cell, say C, we can compute an upper bound of the preference level of all solutions that contain a queen in cell C. This can be done, for example, by making a copy of the current state, positioning a queen in cell C, enforcing soft AC, and then taking the minimum value among all preferences in the resulting state. Since, by making more choices, the preference values can only decrease, and since the preference of a solution is the minimal preference among all those in the final state, this value is certainly an upper

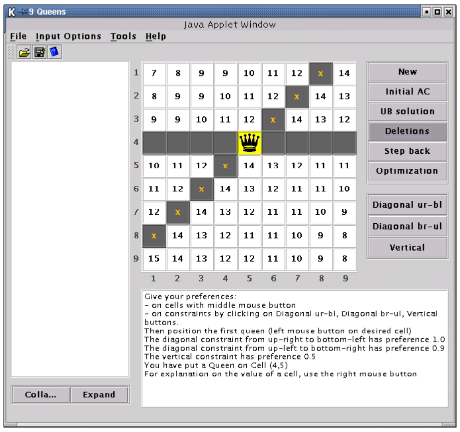

Figure 4: Deletions: each cell shows the number of cells that would be deleted if a queen is positioned in this cell.

bound to the value of all solutions that contain a queen in cell C (and all the previously positioned queens).

Figure 5 shows the upper bounds computed for each cell, which can be seen by clicking on the "UB solution" button on the right. For example, cell (8,9) shows a value of 0.4. This means that, by putting a queen in this cell, we cannot hope to obtain solutions with preference values higher than 0.4.

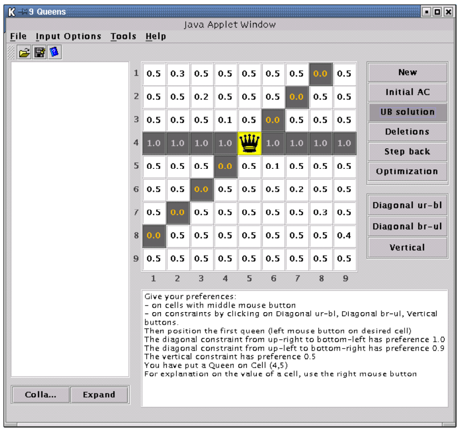

Figure 5: Solution upper bounds: each cell shows an upper bound to the quality of the solutions which are compatible with a queen in this cell.

Thus users will tend to choose cells with a higher upper bound, hoping to get better solutions. However, it is predictable that the user will also have to reach a trade-off between the hope of better solutions and the risk of conflict (which is somewhat described by the number of deletions for that cell).

Getting an optimal solution. Once the user has positioned all the queens, he/she has obtained a complete configuration with a certain preference level (given, we recall, by the minimum of all the preferences in all constraints and domains), which is shown in the lower panel. Figure 6 shows a complete solution with preference level 0.5.

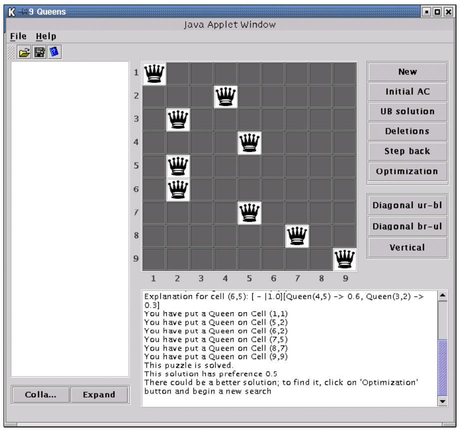

Figure 6: A complete solution, with preference level 0.5.

But this configuration is not necessarily an optimal one. If the user is satisfied with the preference level of the obtained solution, he may stop the interaction with the configurator. However, the configurator should help him/her also in the case he/she wants to get a better solution. In our system, this is simply done as follows:

- the configurator checks (in a possibly incomplete way) whether the obtained solution is optimal or not;

- if the configurator cannot conclude that we have an optimal solution, it proposes the user to start again with an empty board and the initial preferences; however, it first lowers to 0 all those preferences which are lower than or equal to the preference of the last solution found. In this way, if a new solution will be found, it will certainly be better than the last one.

For example, in our system, the configurator checks the optimality of an obtained solution in an incomplete way as follows: it creates a copy of the initial state where all the preferences lower than, or equal to, the solution preference are set to 0, and it checks whether there is a variable whose domain elements all have preference 0. If so, it communicates to the user that the solution is an optimal one. Otherwise, it says that there could be better solutions. In Figure 6, we see a solution which is not optimal, while in Figure 7 we have an optimal solution.

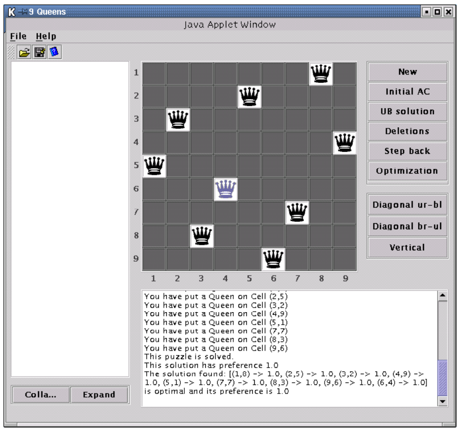

Figure 7: An optimal solution, with preference level 1.

4 Conclusions

We have investigated the scenario of configuration problems with preferences rather than just hard constraints, and we have shown that, even in this very general context, it is possible to build automatically explanations and implications by just keeping track of the inferences that are made during the constraint propagation enforcing. However, the process of recording the necessary information is more complex and expensive than in the hard case, since in general a sequence of previous choices has to be recorded for each possible choice, and not just one as in the hard case.

In particular, for the soft n-queens example considered in this paper, which however contains many general features, we were able to compute information that provide guidance to a user both to make choices that lead to good or optimal solutions, and also to understand the reasons for the current state. In fact:

- current preference values for the cells can be explained by a sequence (or a tree) of previous choices;
- deletion numbers provide a way to predict the negative consequences of a future choice;
- solution upper bounds give an estimate of the goodness of the solutions compatible with a future choice;
- optimization is achieved by iterating the choice process with new preferences, to get better solutions.

References

[1] J. Amilhastre, H. Fargier, P. Marquis. Consistency restoration and explanations in dynamic CSPs – application to configuration. In *Proc. ECAI 2000 workshop on modelling and solving problems with constraints*, 2000.

[2] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2):201–236, March 1997.

[3] D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. IEEE International Conference on Fuzzy Systems*, pages 1131–1136. IEEE, 1993.

[4] E. C. Freuder, C. Likitvivatanavong, R. J. Wallace. Explanation and implication for configuration problems. *IJCAI 2001 workshop on configuration*, pages 31-37, 2001.

[5] N. Jussien, V. Barichard. The palm system: explanation-based constraint programming. In *Proc. CP 2000 workshop on techniques for implementing constraint programming systems*, 2000.

[6] Zs. Ruttkay. Fuzzy constraint satisfaction. In *Proc. 3rd IEEE International Conference on Fuzzy Systems*, pages 1263–1268, 1994.

[7] D. Sabin, R. Weigel. Product configuration frameworks – a survey. *IEEE Intelligent Systems and their Applications. Special issue on configuration*, pages 42-49, 1998.

[8] T. Schiex. Possibilistic constraint satisfaction problems, or "how to handle soft constraints?". In *Proc. 8th Conf. of Uncertainty in AI*, pages 269–275, 1992.

[9] T. Schiex, H. Fargier, and G. Verfaillie. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *Proc. IJCAI95*, pages 631–637. Morgan Kaufmann, 1995.

[10] Edward P. K. Tsang. Foundations of Constraint Satisfaction. Academic Press, 1993.

[11] R. J. Wallace, E. C. Freuder. Explanations for whom? In *Proc. CP 2001 Workshop on User-Interaction in Constraint Satisfaction*, 2001.