Iterative Forward Search: Combining Local Search with Maintaining Arc Consistency and a Conflict-based Statistics

Tomáš Müller[1], Roman Barták[1] and Hana Rudová[2]

[1] Faculty of Mathematics and Physics, Charles University
Malostranské nám. 2/25, Prague, Czech Republic
{muller|bartak}@ktiml.mff.cuni.cz
[2] Faculty of Informatics, Masaryk University
Botanická 68a, Brno 602 00, Czech Republic
hanka@fi.muni.cz

Abstract. The paper presents an iterative forward search framework for solving constraint satisfaction and optimization problems. This framework combines ideas of local search, namely improving a solution by local steps, with principles of depth-first search, in particular extending a partial feasible assignment towards a solution. Within this framework, we also propose and study a conflict-based statistics and explanation-based arc consistency maintenance. To show the versatility of the proposed framework, the dynamic backtracking algorithm with maintaining arc consistency is presented as a special instance of the iterative forward search framework. The presented techniques are compared on random constraint satisfaction problems and a real-life lecture timetabling problem.

1 Introduction

Many real-life industrial and engineering problems can be modeled as finite constraint satisfaction problems (CSP). A 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. In a complete solution of a CSP, a value is assigned to every variable from the variable's domain, in such a way that every constraint is satisfied. Algorithms for solving CSPs usually fall into one of two main families: systematic search algorithms and local search algorithms.

Most algorithms for solving CSPs search systematically through the possible assignments of values to variables. Such algorithms are guaranteed to find a solution, if one exists, or to prove that the problem has no solution. They start from an empty solution (no variable is assigned) that is extended towards a complete solution satisfying all the constraints in the problem. Backtracking occurs when a dead-end is reached. The biggest problem of such backtrack-based algorithms is that they typically make early mistakes in the search, i.e., a wrong early assignment can cause a whole subtree to be explored with no success. There

are several ways of improving standard chronological backtracking. Look-back enhancements exploit information about the search which has already been performed, e.g., backmarking or backjumping [6]. Look-ahead enhancements exploit information about the remaining search space via filtering techniques (e.g., via maintaining arc consistency described in [1, 2]) or variable and value ordering heuristics [13]. The last group of enhancements is trying to refine the search tree during the search process, e.g., dynamic backtracking [8].

Local search algorithms [13] (e.g., min-conflict [14] or tabu search [7]) perform an incomplete exploration of the search space by repairing an infeasible complete assignment. Unlike systematic search algorithms, local search algorithms move from one complete (but infeasible) assignment to another, typically in a non-deterministic manner, guided by heuristics. In general, local search algorithms are incomplete, they do not guarantee finding a complete solution satisfying all the constraints. However, these algorithms may be far more efficient (wrt. response time) than systematic ones in finding a solution. For optimization problems, they can reach a far better quality in a given time frame.

In this paper, we present an iterative forward search (IFS) framework (based on our earlier work published in [15]) to solve CSPs. This framework is close to local search methods; however, it maintains a partial feasible solution as opposed to the complete conflicting assignment characteristic of local search. Similarly to local search, we process local changes in the solution. We also describe how to extend the presented algorithm with dynamic maintenance of arc consistency and conflict-based statistics. New conflict-based statistics is proposed to improve the quality of the final solution. Conflicts during the search are memorized and their potential repetition is minimized. The presented framework is very close to local search algorithms, but unlike them, it can be easily refined into a dynamic backtracking algorithm by enforcement of several basic rules.

There are several other approaches which try to combine local search methods together with backtracking based algorithms. For example, the decision repair algorithm presented in [11] repeatedly extends a set of assignments (called decisions) satisfying all the constraints, like in backtrack-based algorithms. It performs a local search to repair these assignments when a dead-end is reached (i.e., these decisions become inconsistent). After these decisions are repaired, the construction of the solution continues to the next dead-end. Unlike this approach, our algorithm operates more like the local search method – it does not execute a local search after a dead-end is reached but it applies the same local steps during search. A similar approach is used in the algorithm presented in [18] as well.

One of the primary areas to which we intended to apply the proposed algorithm is a course timetabling problem at Purdue University (USA), described in details in [17]. It is a real-life large-scale problem that includes features of over-constrained as well as optimization problems. The goal is to timetable more than 800 lectures to a limited number of lecture rooms (about 50) and to satisfy as many as possible individual course requests of almost 30,000 students. Moreover, the algorithm should be able to react to additional changes, in particular, the

algorithm should be capable of repairing a modified timetable where some hard constraints are violated by the user changes. IFS satisfies all these requirements and as we will show later, it produces better solutions than existing approaches.

The paper is organized as follows. In the next section, we will describe the iterative search algorithm formally. Special subsections will be devoted to extensions of IFS, namely conflict-based statistics and maintaining dynamic arc consistency. After that, we will show how dynamic backtracking with maintaining arc consistency [10] can be rewritten as a special instance of IFS. A short summary of the implementation together with the experimental results for random CSPs and a real-life timetabling problem will conclude the paper.

2 Iterative Forward Search Algorithm

The iterative forward search algorithm, that we propose here, is based on ideas of local search methods [13]. However, in contrast to classical local search techniques, it operates over feasible, though not necessarily complete solutions. In such a solution, some variables can be left unassigned. Still all hard constraints on assigned variables must be satisfied. Similarly to backtracking based algorithms, this means that there are no violations of hard constraints.

Working with feasible incomplete solutions has several advantages compared to the complete infeasible assignments that usually occur in local search techniques. For example, when the solver is not able to find a complete solution, an incomplete (but feasible) one can be returned, e.g., a solution with the least number of unassigned variables found. Moreover, because of the iterative character of the search, the algorithm can easily start, stop, or continue from any feasible solution, either complete or incomplete.

The search processeds iteratively (see Fig. 1 for algorithm). During each step,

```
procedure SOLVE(initial)            // initial solution is the parameter
   iteration = 0;                   // iteration counter
   current = initial;               // current solution
   best = initial;                  // best solution
   while canContinue(current, iteration) do
         iteration = iteration + 1;
         variable = selectVariable(current);
         value = selectValue(current, variable);
         UNASSIGN(current, CONFLICTING_VARIABLES(current, variable, value));
         ASSIGN(current, variable, value);
         if better(current, best) then best = current
   end while
   return best
end procedure
```

Fig. 1. Pseudo-code of the search algorithm.

an unassigned or assigned variable is initially selected. Typically an unassigned variable is chosen like in backtracking-based search. An assigned variable may be selected when all variables are assigned but the solution is not good enough (for

example, when there are still many violations of soft constraints). Once a variable is selected, a value from its domain is chosen for assignment. Even if the best value is selected (whatever 'best' means), its assignment to the selected variable may cause some hard conflicts with already assigned variables. Such conflicting variables are removed from the solution and become unassigned. Finally, the selected value is assigned to the selected variable.

The algorithm attempts to move from one (partial) feasible solution to another via repetitive assignment of a selected value to a selected variable. During this search, the feasibility of all hard constraints in each iteration step is enforced by unassigning the conflicting variables. The search is terminated when the requested solution is found or when there is a timeout, expressed e.g., as a maximal number of iterations or available time being reached. The best solution found is then returned.

The above algorithm schema is parameterized by several functions, namely

- the termination condition (function *canContinue*),
- the solution comparator (function *better*),
- the variable selection (function *selectVariable*) and
- the value selection (function *selectValue*).

Termination Condition. The termination condition determines when the algorithm should finish. For example, the solver should terminate when the maximal number of iterations or some other given timeout value is reached. Moreover, it can stop the search process when the current solution is good enough, e.g., all variables are assigned and/or some other solution parameters are in the required ranges. For example, the solver can stop when all variables are assigned and less than 10% of the soft constraints are violated. Termination of the process by the user can also be a part of the termination condition.

Solution Comparator. The solution comparator compares two solutions: the current solution and the best solution found. This comparison can be based on several criteria. For example, it can lexicographically order solutions according to the number of unassigned variables (smaller number is better) and the number of violated soft constraints.

Variable Selection. As mentioned above, the presented algorithm requires a function that selects a variable to be (re)assigned during the current iteration step. This problem is equivalent to a variable selection criterion in constraint programming. There are several guidelines for selecting a variable [5]. 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.

We can split the variable selection criterion into two cases. If some variables remain unassigned, the "worst" variable among them is selected, i.e., first-fail principle is applied.

The second case occurs when all variables are assigned. Because the algorithm does not need to stop when a complete feasible solution is found, the variable selection criterion for such case has to be considered as well. Here all variables are assigned but the solution is not good enough, e.g., in the sense of violated soft constraints. We choose a variable whose change of a value can introduce the best improvement of the solution. It may, for example, be a variable whose value violates the highest number of soft constraints.

Value Selection. After a variable is selected, we need to find a value to be assigned to the variable. This problem is usually called "value selection" in constraint programming [5]. Typically, the most useful advice is to select the best-fit value. So, we are looking for a value which is the most preferred for the variable and which causes the least trouble as well. This means that we need to find a value with the minimal potential for future conflicts with other variables. For example, a value which violates the smallest number of soft constraints can be selected among those values with the smallest number of hard conflicts.

2.1 Conflict-based Statistics

Conflict-based statistics were successfully applied in earlier works [6, 11]. In our approach, the conflict-based statistics works as an advice in the value selection criterion. It helps to avoid repetitive, unsuitable assignments of the same value to a variable by memorizing conflicts caused by this assignment in the past. In contrast to the *weighting-conflict* heuristics proposed in [11], conflict assignments are memorized together with the causal assignment which impacted them. Also, we propose the presented statistics to be unlimited, to prevent short-term as well as long-term cycles.

The main idea behind conflict-based statistics is to memorize conflicts and prohibit their potential repetition. When a value v_0 is assigned to a variable V_0, hard conflicts with previously assigned variables (e.g., $V_1 = v_1$, $V_2 = v_2$, ... $V_m = v_m$) can occur. These variables V_1,...V_m have to be unassigned before the value v_0 is assigned to the variable V_0. These unassignments, together with the reason for their unassignment (e.g., assignment $V_0 = v_0$), and a counter tracking how many times such an event occurred in the past, is stored in memory.

Later, if a variable is selected for an assignment again, the stored information about repetition of past hard conflicts can be taken into account, e.g., in the value selection heuristics. For example, if the variable V_0 is selected for an assignment again, we can weight the number of hard conflicts created in the past for each possible value of the variable. In the above example, the existing assignment $V_1 = v_1$ can prohibit the selection of the value v_0 for the variable V_0 if there is again a conflict with the assignment $V_1 = v_1$.

Conflict-based statistics is a data structure that memorizes hard conflicts which have occurred during the search together with their frequency (e.g., that assignment $V_0 = v_0$ caused c_1 times an unassignment of $V_1 = v_1$, c_2 times of $V_2 = v_2$ ... and c_m times of $V_m = v_m$). More precisely, it is an array

$$Stat[V_a = v_a, V_b \neq v_b] = c_{\mathrm{ab}},$$

saying that the assignment $V_a = v_a$ caused c_{ab} times unassignment of $V_b = v_b$ in the past. Note that in case of n-ary constraints (where $n > 2$), this does not imply that the assignments $V_a = v_a$ and $V_b = v_b$ cannot be used together. The proposed conflict-based statistics does not actually work with any constraint, it only memorizes the unassignments together with the assignment which caused them. Let us consider a variable V_a selected by `selectVariable` function, a value v_a selected by `selectValue`. Once an assignment $V_b = v_b$ is selected by CONFLICTING_VARIABLES to be unassigned, the array $Stat[V_a = v_a, V_b \neq v_b]$ is incremented by one.

The data structure is implemented as a hash table, storing information for conflict-based statistics. A counter is maintained for the tuple $A = a$ and $B \neq b$. This counter is increased when the value a was assigned to the variable A and b needed to be unassigned from B. The example of this structure

$$A = a \Rightarrow \begin{cases} 3 \times B \neq b \\ 4 \times B \neq c \\ 2 \times C \neq a \\ 120 \times D \neq a \end{cases}$$

expresses that variable B lost its assignment b three times and its assignment c four times, variable C lost its assignment a two times and D lost its assignment a 120 times, all because of later assignments of value a to variable A. This structure is being used in the value selection heuristics to evaluate existing conflicts with the assigned variables. For example, if there is a variable A selected and if the value a is in conflict with an assignment $B = b$, we know that a similar problem has already occurred $3\times$ in the past, and the conflict $A = a$ is weighted with this number.

For example, a *min-conflict* value selection criterion, which selects a value with the minimal number of conflicts with the existing assignments, can be easily adapted to a *weighted min-conflict* criterion. The value with the smallest sum of the number of conflicts multiplied by their frequencies is selected.

Stated in another way, the weighted min-conflict approach helps the value selection heuristics to select a value that might cause more conflicts than another value, but these conflicts occurred less frequently, and therefore they have a lower weighted sum. This can considerably help the search to get out of a local minimum.

We plan to study the following extensions of the conflict-based statistics:

- If a variable is selected for an assignment, the above presented structure can also tell how many potential conflicts a value can cause in the future. In the above example, we already know that four times a later assignment of $A = a$ caused that value c was unassigned from B. We can try to minimize such future conflicts by selecting a different value of the variable B while A is still unbound.
- The memorized conflicts can be aged according to how far they have occurred in the past. For example, a conflict which occurred 1000 iterations ago can have half the weight of a conflict which occurred during the last iteration or it can be forgotten at all.

Let us study the space complexity of the above data structure for storing conflict-based statistics. At a maximum, there could be a counter for each pair of possible assignments $V_a = v_a$ and $V_b = v_b$, where $V_a \neq V_b$ and there is a constraint between variables V_a and V_b which can prohibit concurrent assignments $V_a = v_a$ and $V_b = v_b$. However, note that each increment of a counter in the statistics means an unassignment of an assigned variable. Therefore each counter $Stat[V_a = v_a, V_b \neq v_b] = n$ in the statistics means that there was an assignment $V_b = v_b$ which was unassigned n times when v_a was assigned to V_a. Together with the fact, that there is only one assignment done in each iteration, the following invariant will be always true during the search: The total sum of all counters in the statistics plus the current number of assigned variables is equal to the number of processed iterations. Therefore, if the above described hash table (which is empty at the beginning and does not contain empty counters) is used, the total number of all counters in it will never exceed the number of iterations processed so far.

The presented approach can be successfully used in other search algorithms as well. For example, in the local search, we can memorize the assignment $V_x = v_x$, which was selected to be changed (re-assigned). A reason for such selection can be retrieved and memorized together with the selected assignment $V_x = v_x$ as well. Note that typically an assignment which is in a conflict with some other assignments is selected.

Furthermore, the presented conflict-based statistics can be used not only inside the presented algorithm. Its constructed 'implications' together with the information about frequency of their occurrences can be easily used by users or by some add-on deductive engine to identify inconsistencies[3] and/or hard parts of the input problem. The user can then modify the input requirements in order to eliminate the found problems and let the solver continue to search with such a modified input problem.

2.2 Maintaining Arc Consistency Using Explanations

Because the presented algorithm works with partial feasible solutions, it can be easily extended to dynamically maintain arc consistency during the search. This can be done by using well known dynamic arc consistency (MAC) algorithms (e.g., by AC|DC algorithm published in [16] or DnAC6 published in [4]) which are widely used in Dynamic CSPs [12].

Moreover, since the only constraints describing assignments (constraint *Variable=value*) can be added and removed during the search, approaches based on explanations [9, 10] can be used as well. In this section, we present how these explanations, which are traditionally used in systematic search algorithms, can be used in our iterative forward search approach in order to maintain arc consistency.

An explanation, $V_i \neq v_i \Leftarrow (V_1 = v_1 \& V_2 = v_2 ... \& V_j = v_j)$ describes that the value v_i cannot be assigned to the variable V_i since it is in a conflict with

[3] Actually, this feature allows to discover all inconsistent inputs during solving the Purdue University timetabling problem [17].

the existing assignments $V_1 = v_1$, $V_2 = v_2$, ... $V_j = v_j$. This means that there is no complete feasible assignment containing assignments $V_1 = v_1$, $V_2 = v_2$, ... $V_j = v_j$ together with the assignment $V_i = v_i$ (these equalities form a no-good set [9]).

During the arc consistency maintenance, when a value is deleted from a variable's domain, the reason (forming an explanation) can be computed and attached to the deleted value. Once a variable (say V_x with the assigned value v_x) is unassigned during the search, all deleted values which contain a pair $V_x = v_x$ in their explanations need to be recomputed. Such value can be either still inconsistent with the current (partial) solution (a different explanation is attached to it in this case) or it can be returned back to its variable's domain. Arc consistency is maintained after each iteration step, i.e., the selected assignment is propagated into the not yet assigned variables. When a value v_x is assigned to a variable V_x, an explanation $V_x \neq v'_x \Leftarrow V_x = v_x$ is attached to all values v'_x of the variable V_x, different from v_x.

In the case of forward checking, computing explanations is rather easy. A value v_x is deleted from the domain of the variable V_x only if there is a constraint which prohibits the assignment $V_x = v_x$ because of the existing assignments (e.g., $V_y = v_y$,..$V_z = v_z$). An explanation for the deletion of this value v_x is then $V_x \neq v_x \Leftarrow (V_y = v_y \& ... V_z = v_z)$, where $V_y = v_y \& ... V_z = v_z$ are assignments contained in the prohibiting constraint. In case of arc consistency, a value v_x is deleted from the domain of the variable V_x if there is a constraint which does not permit the assignment $V_x = v_x$ with other possible assignments of the other variables in the constraint. This means that there is no support value (or combination of values) for the value v_x of the variable V_x in the constraint. An explanation is then a union of explanations of all possible support values for the assignment $V_x = v_x$ of this constraint which were deleted. The reason is that if one of this support values is returned to its variable's domain, this value v_x may be returned as well (i.e., the reason for its deletion has vanished, a new reason needs to be computed).

As for the implementation, the above described algorithm schema (see Fig. 1) can remain as it is, we only need to enforce arc consistency of the initial solution and to extend UNASSIGN and ASSIGN methods. Procedure ASSIGN(solution, variable, value) should enforce arc consistency of the solution with the selected assignment *variable=value* and the procedure UNASSIGN(solution, variable, value) should 'undo' the assignment *variable=value*. It means that explanations of all values which were deleted and which contain assignment *variable = value* in their explanations needs to be recomputed. This can be done via returning all these values into their variables' domains followed by arc consistency maintenance over their variables.

Using the presented explanations-based approach gives us more flexibility than dynamic arc consistency algorithms (e.g., AC|DC, DnAC, ...) where the value selection function can choose only among the values in the current domain of the variable, i.e., among the values that are not pruned by arc consistency. The values which were deleted via MAC can be selected as well (actually they are

only marked as no-good values in the implementation). If a deleted variable is selected, it can become feasible by repeatedly unassigning a selected value from its explanation until the value is returned to the selected variable's domain. This cannot be done as easily as in the case of dynamic arc consistency algorithms, since we do not know the cause of deletion of a deleted value. For instance, there are several possibilities how to treat a case when there is a variable with an empty domain (i.e., all its values were deleted via MAC). We discuss two of them below, see Sec. 4. Note that we might want to compute the largest feasible solution (in the number of assigned variables) in case of over-constrained problem.

3 IFS as Dynamic Backtracking with MAC

In this section, we describe how the presented iterative forward search framework can be used for modeling of dynamic backtracking (DB) search with the arc consistency maintenance (MAC). In some sense, the presented IFS algorithm with MAC can be seen as an extension of DB with MAC, e.g., described in [10], towards the local search based methods.

Dynamic backtracking with MAC can come out of the above presented IFS with MAC via the following modifications and/or restrictions:

- Variable selection function *selectVariable* always returns an unassigned variable. If there are one or more variables with empty domains, one of them is returned in the variable selection function.
- Value selection function *selectValue* always returns a value from the selected variable's domain (i.e., not-deleted value), if there is no such value, it returns **null**.
- When all the variables are assigned the solver terminates and returns the found solution (termination condition function *canContinue*). In case of **branch&bound** technique an existence of a complete solution should lower the bound so that a conflict arises, which leads to some unassignments.
- If the selected value is **null** (which means that the selected variable has an empty domain), a union of all assignments which prohibits all the values of the selected variable (a union of assignments of all values' explanations) is computed. The last made assignment of them is selected (each variable can memorize an iteration number, when it was assigned for the last time). This assignment has to be unassigned, all other assignments from the computed union are taken as an explanation for this unassignment. If the computed explanation is empty (e.g., $V_x \neq v_x \Leftarrow \varnothing$), the value can be permanently removed from its variable's domain because it can never be a part of a complete solution. If the computed union is empty, there is no complete solution and the algorithm returns **fail**.
- If a value v_x is assigned to a variable V_x, an explanation $V_x \neq v'_x \Leftarrow V_x = v_x$ is attached to all values from the domain of the variable V_x different from v_x. Note that in contrast to the IFS MAC algorithm described in the previous section, the already deleted values from the variable's V_x domain leave its original explanation (it is not changed to $V_x \neq v'_x \Leftarrow V_x = v_x$). This can be

done, because the last assigned variable from the variables which prohibits values from a variable with an empty domain is always unassigned.

Like in the above presented IFS MAC algorithm, arc consistency maintenance and its undo is called automatically after each assignment and unassignment, respectively.

4 Experiments

The above described algorithm together with its presented extensions has been implemented in Java. It contains a general implementation of the iterative search algorithm. The general solver operates over abstract variables and values with a selection of available extensions, basic general heuristics, solution comparators, and termination functions. It may be customized to fit a particular problem (e.g., as it has been extended for Purdue University timetabling, see Sec. 4.2) by implementing variable and value definitions, adding hard and soft constraints, and extending the parametric functions of the algorithm. The results presented here were computed on 1GHz Pentium IIIm PC running Windows 2000, with 512 MB RAM and J2SDK 1.4.2.

Because we attempt to solve large scale problems, maintaining arc consistency is based on AC3 algorithm (e.g., see [19]). In the Purdue University timetabling problem we have almost 830 variables (there is a variable for each course) with the total number of more than 200,000 values (there is a value for each location of a course in the timetable, including a selection of time(s), room and instructor). Furthermore, nearly every two variables are related by some constraint, e.g., typically there is at least one room they can both use. Due to the memory reasons, this prohibits any consistency method which is based on memorizing values for each pair of values or for each pair of value and variable.

In the following experiments we compare several mutations of the above presented algorithm and its improvements. For all these variants, an unassigned variable is selected randomly and the value selection is based on min-conflict strategy. This means that a value is randomly selected among the values whose assignment will cause the minimal number of conflicts with the existing assignments. The search is terminated when a complete solution is found or when the given time limit is reached. As for the solution comparator, a solution with the highest number of assigned variables is always selected. The compared algorithms are:

- **IFS MCRW** ... min-conflict selection of values with 2% random walk[4]
- **IFS TABU** ... tabu list of the length 20 is used to avoid cycling[5]
- **IFS ConfStat** ... min-conflict value selection where conflicts are weighted according to the conflict-based statistics (as described in Sec. 2.1)

[4] With the given probability, a value is selected randomly from all values of the selected variable's domain.

[5] Repeated selection of the same pair (*variable*, *value*) is prohibited for the given number of following iterations.

- **IFS MAC** ... arc-consistency maintenance; if there is a variable with an empty domain, a variable which caused a removal of one or more of values is selected and unassigned.[6]
- **IFS MAC+** ... arc-consistency maintenance; the algorithm continues extending the solution even when there is a value with an empty domain. If the selected variable has an empty domain (pruned by MAC) then one of values deleted by MAC is selected (via min-conflic value selection).
- **DBT MAC** ... dynamic backtracking algorithm with arc consistency maintenance (as described in Sec. 3)
- **DBT FC** ... dynamic backtracking algorithm with forward checking

4.1 Random CSP

In this section, we present results achieved on the Random Binary CSP with uniform distribution [3]. A random CSP is defined by a four-tuple (n, d, p_1, p_2), where n denotes the number of variables and d denotes the domain size of each variable, p_1 and p_2 are two probabilities. They are used to generate randomly the binary constraints among the variables. p_1 represents the probability that a constraint exists between two different variables (tightness) and p_2 represents the probability that a pair of values in the domains of two variables connected by a constraint are incompatible (density).

Figure 2 presents the number of assigned variables in percentage to all vari-

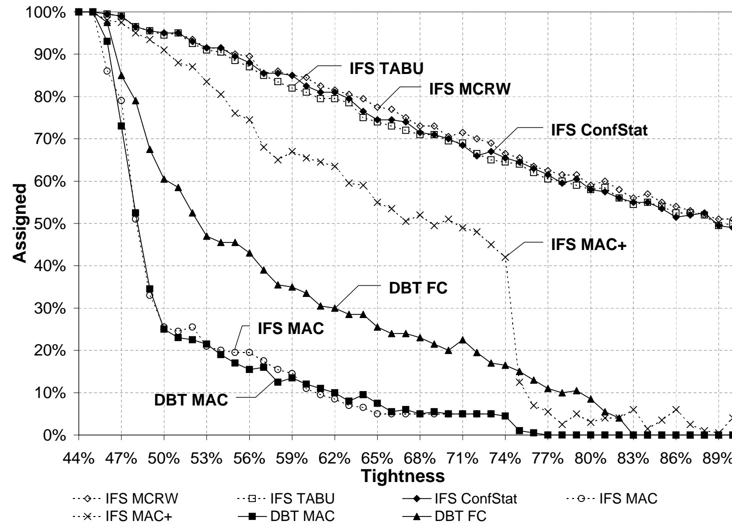

Fig. 2. $CSP(20, 15, 43\%, p_2)$, number of assigned variables (in percentage to all variables), the best achieved solution within 60 seconds, average from 10 runs.

[6] This is done so that a value v_x of such a variable with an empty domain V_x is selected randomly and a randomly selected assignment from the explanation $V_x \neq v_x$ is unassigned.

ables wrt. the probability p_2 representing tightness of the generated problem $CSP(20, 15, 43\%, p_2)$. The average values of the best achieved solutions from 10 runs on different problem instances within the 60 second time limit are presented.

Each of the compared algorithms was able to find a complete solution within the time limit for all the given problems with a tightness under 46%. Achieved results from min-conflict random walk, tabu-list and the presented conflict-based statistics seem to be very similar for this problem. Also, it is not surprising that a usage of consistency maintenance techniques lowers the maximal number of assigned variables, e.g., both dynamic backtracking with MAC and IFS with MAC extend an incomplete solution only when it is arc consistent with all unassigned variables. As we can see from the Figure 2, we can get much better results when we allow the search to continue even if there is a variable with an empty domain.

For the results presented in Figure 3, we turned the random CSP into an op-

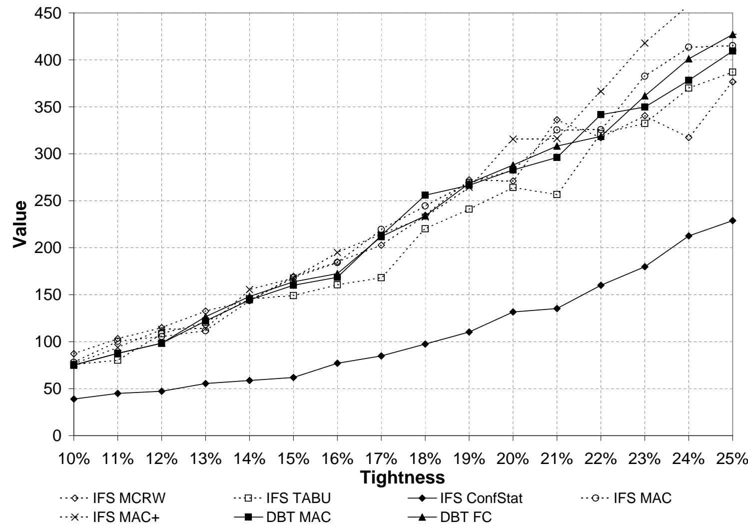

Fig. 3. $minCSP(40, 30, 43\%, p_2)$, the sum of all assigned values of the best solution within 60 seconds wrt. problem tightness, average from 10 runs.

timization problem where we are searching for a complete feasible solution with the smallest total sum of the assigned values. Recall that each variable has d generated values from $0, 1, ...d-1$. For the comparison we used $CSP(40, 30, 43\%, p_2)$ problems with the tightness p_2 taken so that every measured algorithm was able to find a complete solution for each of 10 different generated problems within the given 60 second time limit. The min-conflict value selection criterion was adapted to take the smallest value from the values which cause the minimum number of conflicts. For DBT, the smallest value from a domain of a selected variable was taken as well. As for conflict-based statistics, each value is weighted by itself added to the number of weighted conflicts. For example, value 7 with 4 conflicts has a weight $7 + 4 = 11$. If value 3 has 9 conflicts, so its weight is 12, then the value 7 will be preferred over 3. For all the measured algorithms,

the solver continues even if a complete solution is found until the time limit is reached. A complete solution with the smallest sum of the assigned values is then returned. For this problem, the presented conflict-based statistics was able to give much better results than other compared algorithms. The algorithm is obviously trying to stick much more with the smallest values than the others, but it is able to find a complete solution since the conflict counters are rising during the search. Such behaviour can be very handy for many optimization problems, especially when optimization criteria (expressed either by some optimization function or by soft constraints) go against the hard constraints.

Conclusion. For the tested random constraint satisfaction problems, IFS MAC does not seem to be suitable: it produces very similar results to DBT MAC, but unlike DBT MAC, it can not guarantee finding a complete solution, if one exists, or to prove that the problem has no solution.

Since IFS MAC+ can continue extending a partial solution even when there is a variable with an empty domain, it can produce much better results than IFS MAC in case that the given problem is over-constrained (i.e., there is no complete solution). For some tasks, it can be an interesting compromise between backtrack-based search and local search.

For pure, not-optimization constraint satisfaction problems (e.g., results from Fig. 2), the presented IFS with conflict-based statistics returns very similar results to other tested traditional local search principles preventing cycles (i.e., tabu-list and random walk). However the presented conflict-based statistics can be very useful when optimization criteria are considered.

4.2 Real-Life Application: Purdue University Timetabling

In this section, we present some results achieved on the large lecture timetabling problem from Purdue University. The following tests were performed on the complete Fall 2004 data set[7] which consists of 826 classes (forming 1782 meetings, 4050 half-hours) that must fit into 50 lecture rooms with capacities up to 474 students. In this problem, 89,633 course demands of 29,808 students must be considered.

The timetable maps classes (students, instructors) to meeting locations and times. A primary objective is to minimize the number of potential student course conflicts which occur during this process. Other major constraints on the problem solution are instructor availability and a limited number of rooms with sufficient capacity, specific equipment, and a suitable location. Some of these constraints must be satisfied; others are introduced within an optimization process in order to avoid an over-constrained problem.

Figure 4 presents some results achieved with the presented framework. Average values together with their RMS (root-mean-square) variances of the best achieved solutions from 10 different runs found within 30 minute time limit are presented. *Time* refers to the amount of time required by the solver to find the presented solution.

[7] Similar results were achieved also on a complete data set from Fall 2001, used in our previous work [17].

Test cases	IFS ConfStat	IFS TABU	IFS MCRW
Assigned variables [%]	100.00 ± 0.00	97.67 ± 0.15	98.29 ± 0.16
Time [min]	24.11 ± 4.42	24.17 ± 3.62	24.52 ± 3.83
Student conflicts [%]	1.97 ± 0.06	1.97 ± 0.07	2.05 ± 0.19
Preferred time [%]	85.64 ± 1.57	89.86 ± 0.69	89.63 ± 1.06
Preferred room [%]	50.39 ± 5.34	66.48 ± 3.42	64.84 ± 3.86

Fig. 4. Purdue University Timetable, characteristics for the best achieved solutions within 30 minutes, average values and RMS variances from 10 runs.

The value selection criterion was extended to take into account three optimization criteria. *Student conflicts* give the percentage of unsatisfied requirements for the courses chosen by the students. One student conflict means that there are two courses required by a student that cannot be both attended, e.g., because they overlap in time. *Preferred time* and *preferred room* estimate the satisfaction of time and room preferences respectively. These preferences are given by the instructors individually for each class. In our heuristics, room preferences are considered much less important than time preferences and student conflicts.

IFS with conflict-based statistics was able to find a complete solution of a good quality in each run; the first complete solution was found after $6-10$ minutes. Moreover, it was able to significantly improve the first complete solution with approximately 2.3% of violated student requirements and 80% of time preferences satisfied. On the other hand, neither tabu search nor min-conflict random walk were able to find any complete solution within the given 30 minute time limit; at least 17 variables for TABU and 12 variables for MCRW remained unassigned after each run.

Dynamic backtracking with either MAC or FC (not included in Figure 4) was able to assign in average approximately 93% of variables, with almost 3% violated student requirements and only approximately 40% of time preferences satisfied. IFS MAC was able to assign only about 65% of variables. IFS MAC+ assigned about 94% variables, with approximately 2.2% student conflicts and around 75% of time preferences. Consistency was maintained over all hard constraints.

We plan to use MAC+ only over the "additional" constraints, e.g. a precedence constraint between two or more courses or not-overlap constraint between a lecture and its seminars. However, the used data set contains only 203 of such constrains, so there is no significant difference between a solution with and without MAC+ (IFS ConfStat versus IFS ConfStat with MAC+ on additional constraints). Currently we work on solving other Purdue University timetabling problems where the number of these constraints significantly increases.

Conclusion. The general consensus, that local search is more suited for optimization problems than backtrack-based search, is valid also for our large lecture timetabling problem. Unlike the other tested algorithms, the presented conflict-based statistics is capable to produce high quality and stable results. Furthermore, if there is any time available after the first complete solution is found, the solver is also able to gradually improve this solution. We believe that the arc

consistency maintenance can help us solve some complicated situations which can arise in our timetabling problem.

5 Conclusions and Future Work

We have presented a promising iterative forward search framework which is, as we believe, together with the presented improvements, capable of solving various constraint optimization problems. We have presented some results on random CSP and on Purdue University timetabling problem. Our solver is able to construct a demand-driven timetable, which satisfies approximately 98% course requests of students together with about 85% of time preferences.

Our future research will include extensions of the proposed general algorithm together with improvements to the implemented solver. We would like to do an extensive study of the proposed framework and its possible application to other, non timetabling-based problems. As for Purdue University timetabling, we are currently extending the CLP solver [17] with some of the features included here to present a fair comparison. However, the CLP solver was not yet able to find a complete solution in the accomplished preliminary experiments.

Acknowledgements

This work is partially supported by the Czech Science Foundation under the contract No. 201/04/1102 and by Purdue University.

References

[1] C. Bessière and J. C. Régin. Arc consistency for general constraint networks: Preliminary results. In *Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 398–404, Nagoya, Japan, 1997.

[2] C. Bessière and J. C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings IJCAI'01*, pages 309–315, Seattle WA, 2001.

[3] Christian Bessière. Random uniform csp generators, 1996. `http://www.lirmm.fr/~bessiere/generator.html`.

[4] R. Debruyne. Arc-consistency in dynamic csps is no more prohibitive. In *Proceedings of 8th Conference on Tools with Artificial Intelligence (TAI'96)*, pages 299–306, 1996.

[5] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[6] Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188, 2002.

[7] Philippe Galinier and Jin-Kao Hao. Tabu search for maximal constraint satisfaction problems. In *Proceedings 3rd International Conference on Principles and Practice of Constraint Programming*, pages 196–208. Springer-Verlag LNCS 1330, 1997.

[8] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, pages 23–46, 1993.

[9] Narendra Jussien. The versatility of using explanations within constraint programming. In *Habilitation thesis of Universit de Nantes*, France, 2003.

[10] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming*, pages 249–261, 2000.

[11] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1):21–45, 2002.

[12] Narendra Jussien and Gérard Verfaillie. Dynamic constraint solving. In *A tutorial including commented bibliography presented at CP 2003*, Kinsale.

[13] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2000.

[14] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

[15] Tomáš Muller and Roman Barták. Interactive timetabling: Concepts, techniques, and practical results. In *PATAT 2002 — Proceedings of the 4th international conference on the Practice And Theory of Automated Timetabling*, pages 58–72, 2002.

[16] B. Neveu and P. Berlandier. Maintaining arc consistency through constraint retraction. In *Proceedings of the IEEE International Conference on Tools for Artiocial Intelligence (TAI)*, pages 426–431, New Orleans, LA, 1994.

[17] Hana Rudová and Keith Murray. University course timetabling with soft constraints. In *Practice And Theory of Automated Timetabling, Selected Revised Papers*, pages 310–328. Springer-Verlag LNCS 2740, 2003.

[18] Andrea Schaerf. Combining local search and look-ahead for scheduling and constraint satisfaction problems. In *Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1254–1259, Nagoya, Japan, 1997.

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