Appeared in *Proceedings of the 4th International Conference on Soft Computing—IIZUKA '96*, volume 2, pages 955–958, Iizuka, Japan, October 1996.

DynaFLIP++: a knowledge interpreter that matches dynamic domain knowledge with static fuzzy constraints

Wolfgang Slany

Christian Doppler Laboratory for Expert Systems
Technical University of Vienna (E184/2)
Paniglgasse 16, A-1040 Wien, Austria, Europe
Phone: +43–1–58801–6123 Fax: +43–1–5055304
mailto:wsi@dbai.tuwien.ac.at
http://www.dbai.tuwien.ac.at/staff/slany.html

Keywords: Fuzzy Modeling, Dynamical Fuzzy Systems, Gradual Rules as a Tool for Qualitative Modeling, Applications to Industry.

Abstract

Fuzzy constraints must be fine-tuned for a certain task. However, such fine-tuning cannot be done for all possible instantiations of a problem, and thus is generally done on protypical constraints. These templates are applied during run-time to a certain actual situation, i.e., dynamically adapted and applied only where necessary. This paper describes issues arising from the existance of such dynamic constraints and how they are handled in the ∗FLIP++ optimization libraries. Examples from a steelmaking scheduling application are given.

1 Introduction

Real-world decision making is optimizing under vague constraints of different importance, often using uncertain data, where compromises between antagonistic criteria are allowed. Typical application areas include scheduling, design, configuration, planning, and classification.

To address such real-world decision making problems, we have designed and partly implemented a family of C++ libraries: *FLIP++ (pronounce: StarFlipPlusPlus; this refers to the fact that *FLIP++ stands as a regular expression for all names of the individual sub-libraries, and all of the latter are based on FLIP++, which is a C++ library). ∗FLIP++ is composed of the following layered sub-libraries:

- FLIP++: the basic fuzzy logic inference processor library.
- ConFLIP++: the static fuzzy constraint library.
- **DynaFLIP++**: the dynamic fuzzy constraint generation library described in the present paper.
- DomFLIP++: the domain knowledge representation library.
- OptiFLIP++: the heuristic optimizing library; several heuristics have so far been implemented and tested, namely:
  - a tabu list min-conflicts repair based hill climbing heuristic,
  - a min-conflicts repair based iterative deepening heuristic,
  - a min-conflicts repair based random search hill climbing heuristic, and
  - a min-conflicts repair based genetic algorithm heuristic.
- CheckFLIP++: the knowledge-change consistency checker library that also allows to fine-tune the configuration parameters of a problem.
- InterFLIP++: the userinterface for all other libraries.
- DocuFLIP++: the on-line documentation available separately for end-users, knowledge-engineers, and programmers, and accessible via the World-Wide-Web as HTML documents.

The functionality provided by ConFLIP++ has been described by the author in great detail in [1] and [2]. The other libraries have partly been described in the same literature. In the present paper, we further develop those ideas to focus on the role played by DynaFLIP++.

2 Dynamic constraints

The DynaFLIP++ library interacts with ConFLIP++ which itself is needed to formulate static fuzzy constraints and then operate on them. DynaFLIP++ wraps additional functionality around these fuzzy constraints and reaches this functionality up to DomFLIP++ where the actual domain knowledge is processed and optimization is performed. This additional functionality has become necessary because of factors encountered in many different real-world optimization problems, of which we nevertheless only became aware after trying to really handle two different problems, namely steel production scheduling and classroom roostering. Since special consideration is given to the reusability aspect of all libraries, we found that static fuzzy constraints are good to model the aspects associated with partial satisfaction, compromising, and relative importance of constraints. However, other aspects require dynamic generation of constraints from template constraints. For instance, the actual number and kind of constraints often depend upon the current instantiation of the problem that is to be optimized. With each search step in the optimization process, the structure of this instantiation may change. Therefore, the structure of the associated constraint tree has also to change. Considering scheduling, for example, it is possible that a job with an associated delivery date is exchanged with another job that has no associated delivery date, therefore this constraint has not to be evaluated for the second schedule.

Another aspect related to dynamic constraints is that it is normally not useful to tune each constraint separately. Instead, a static constraint is tuned for a selected reference value, and DynaFLIP++ then uses this static constraint to generate a dynamic constraint adapted to the actual situation. Again, looking at the delivery date example, this implies that the template is a static constraint that is tuned around the value zero, with an appropriate fuzzy distribution around it. DynaFLIP++ then specializes this to an actual time in the scheduling horizon, for instance 10 a.m. on July 25, 1995. This dynamic adaptation is mostly harmless for simple constraints such as delivery dates, but becomes more tricky when complex constraints are involved. A more complex example, also taken from the steelmaking domain, would be the duration of the tundish life expectancy. The problem is that the attributes of a definite number of jobs must be aggregated, in this case by adding their durations, without knowing at the time when the static constraint is defined how many jobs will have to be eventually aggregated. Their number can only be determined dynamically at optimization time by checking certain criteria of the actual schedule instantiation. These aggregation operators are similar to those found in spreadsheet software that process a range of values. Additionally, the range in DynaFLIP++ depends on conditions that can be checked only at execution time.

Another difficulty arises because DynaFLIP++ contains absolutely no knowledge about the domain. Using DomFLIP++, optimization structures are defined together with their associated constraints, using the knowledge-engineering features of DynaFLIP++. Additionally, relations between attributes of items in these optimization structures and the static constraints must be defined using a special rule-based language that is interpreted at evaluation time by DynaFLIP++. A rule, slightly simplified but taken from the actual steelmaking domain, illustrates this mechanism:

CONDITION:
$\langle \mathrm{Job}_i, \mathrm{Job}_{i+1} \rangle \in \mathrm{CC}_3 \wedge$
quality separation $\notin \langle \mathrm{Job}_i, \mathrm{Job}_{i+1} \rangle \wedge$
tundish change $\notin \langle \mathrm{Job}_i, \mathrm{Job}_{i+1} \rangle \wedge$
CC setup $\notin \langle \mathrm{Job}_i, \mathrm{Job}_{i+1} \rangle$
CONSTRAINT:
Chemical_Compatibility_$\mathrm{CC}_3 \langle \mathrm{Job}_i, \mathrm{Job}_{i+1} \rangle$

Here, 'Chemical_Compatibility_$\mathrm{CC}_3 \langle \mathrm{Job}_i, \mathrm{Job}_{i+1} \rangle$' stands as a macro that links several attribute values of Job_i and Job_{i+1} to linguistic variables defined in the pre-tuned static constraints. For the actual implementation, the whole definition of this macro must be specified. A sketch of the definition, where only the formulas involving chemical element carbon are detailed is presented in Table 1, thus giving an idea of what kind of dynamic adaptations must be computed. The actual compatibility encompasses 12 more chemical alloying elements, the degassing procedure in the secondary metallurgy aggregates, as well as the casting format between the jobs.

3 Evaluating dynamic constraints

To evaluate a given instantiation of a soft constraint satisfaction problem, a decision function aggregating

Table 1: Sketch of macro definition for Chemical_Compatibility_$CC_3\langle \mathrm{Job}_i,\mathrm{Job}_{i+1}\rangle$, restricted to carbon.

$$
\begin{array}{l}
\mathrm{Job}_i.\mathrm{C.high} \overset{\mathrm{soft}}{<} \mathrm{alloy\_limit.C} \quad \overset{\mathrm{soft}}{\wedge} \quad \mathrm{Job}_{i+1}.\mathrm{C.high} \overset{\mathrm{soft}}{<} \mathrm{alloy\_limit.C} \quad \overset{\mathrm{soft}}{\vee} \\
\mathrm{Job}_{i+1}.\mathrm{C.high} \leq \mathrm{Job}_i.\mathrm{C.high} \quad \overset{\mathrm{soft}}{\wedge} \quad \mathrm{Job}_i.\mathrm{C.low} \leq \mathrm{Job}_{i+1}.\mathrm{C.low} \quad \overset{\mathrm{soft}}{\vee} \\
\mathrm{Job}_i.\mathrm{C.high} \leq \mathrm{Job}_{i+1}.\mathrm{C.high} \quad \overset{\mathrm{soft}}{\wedge} \quad \mathrm{Job}_{i+1}.\mathrm{C.low} \leq \mathrm{Job}_i.\mathrm{C.low} \quad \overset{\mathrm{soft}}{\vee} \\
\mathrm{overlapping.C}\langle \mathrm{Job}_i,\mathrm{Job}_{i+1}\rangle \overset{\mathrm{soft}}{\geq} \mathrm{pair\_limit.C} \\
\quad \big(\text{where} \quad \mathrm{overlapping.C}\langle \mathrm{Job}_i,\mathrm{Job}_{i+1}\rangle := \\
\qquad\qquad \max(0, \min(\mathrm{Job}_i.\mathrm{C.high},\mathrm{Job}_{i+1}.\mathrm{C.high}) - \max(\mathrm{Job}_i.\mathrm{C.low},\mathrm{Job}_{i+1}.\mathrm{C.low}))\big) \\
\qquad\qquad \vdots \qquad \text{similar for other chemical elements.}
\end{array}
$$

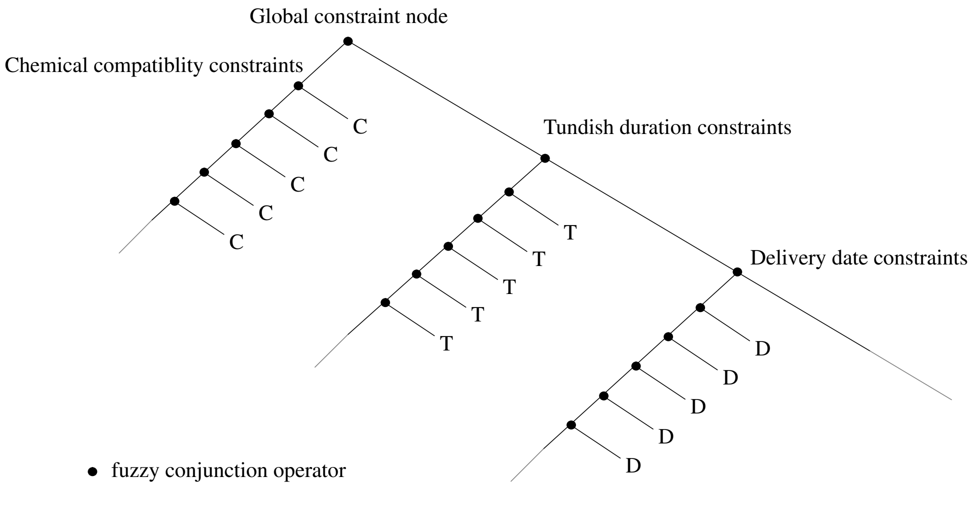

Figure 1: Outline of constraint tree constructed by DynaFLIP++.

all the constraints with their respective priorities, using an appropriate aggregation operator and a corresponding weighting scheme, must be established. In [1, 2], we described possible techniques that help to construct the static part of such a decision function, i.e., the general constraints that will have to be evaluated several times to compute a global evaluation score for one instantiation. Whereas the representation of these general constraints is handled with the ConFLIP++ library, we present in the present paper the DynaFLIP++ library responsible for establishing efficiently a new global constraint representation for a specific instantiation of the problem. This global constraint will result in a highly structured constraint tree for the whole schedule. The constraint evaluation function will return the weighted global satisfaction score based on the current schedule, the value of all open variables, and this global constraint tree. Figure 1 outlines the general structure of this dynamically constructed tree, where the nodes are weighted aggregation operators (in the simplest case conjunctions) and the leafs are ConFLIP++ objects representing individually fine-tuned static constraints, again taken from the steelmaking domain. DynaFLIP++ is able to use most of the framework provided by ConFLIP++ to efficiently compute the evaluation scores for a new schedule.

When scheduling, it is often advisable to introduce an additional measure into the decision function dependent upon whether the current schedule (= instantiation of the constraint satisfaction problem) contains certain difficult jobs. If the scheduling of these jobs is not introduced as a bonus into the decision function, these jobs might never be considered for actual scheduling. There usually exists a non-empty pool of waiting jobs, and only a subset of jobs from the pool can be scheduled immediately. Therefore, the danger is that some difficult jobs will remain in the pool forever unless additional measures are taken. It is clear that this 'difficulty' or 'importance' of a job must increase over the time for which it is still reasonable to 'produce' it, to favorize its eventual scheduling. The easiest way to introduce this 'difficulty' is to formulate a corresponding constraint with an associated priority that will represent these difficult jobs and which will therefore be represented by another branch of a certain constraint type as in Figure 1. Thus, the 'difficulty' of jobs will be one criteria considered when the soft constraint satisfaction problem is optimized. The same applies equally to other soft constraint satisfaction problems such as those encountered in design or planning.

4 Dynamic constraint satisfaction algorithms

To guide the search of the OptiFLIP++ algorithms as discussed in [1], it is necessary to identify the constraint with the worst weighted evaluation, i.e., the severest conflict which can be attacked to minimize conflicts. This can be considered as a side product of evaluating the current instantiation. It corresponds to computing the evaluation using the minimum operator, and more importantly, to remember the constraint involved in the minimal weighted evaluation. This constraint represents the largest conflict for the current instantiation. Often the constraint corresponds to a general feature of the instantiation and cannot be attributed to a specific part of the instantiation. Depending on the repair operators available to the repair based constraint satisfaction algorithms, it can be helpful to find additionally the second largest and third largest conflict. Generally, the search should return the largest conflict being of a type that can be handled by an available repair operator. When DynaFLIP++ has to generate a new dynamic constraint representation for a given instantiation, it computes the individual 'leaf' constraints by calling ConFLIP++ again and again with new variable instantiations on one of the stored reference constraints, and stores the results in an intermediate form that can be used by ConFLIP++ for further aggregation. This ensures a relatively efficient processing of the constraints since the sometimes very large data structure of a static constraint can be reused for all dynamic constraints of its type. At the same time, DynaFLIP++ sorts all the computed intermediate evaluation scores, together with type information, for later selection of 'good' repair operators.

Further efficiency enhancements could be obtained by caching parts of the constraint tree since large areas remain unchanged from one instantiation to the next one.

5 Conclusion

In this paper, we presented problems as well as solutions associated with dynamic fuzzy constraints in real-world decision making problems.

References

[1] Wolfgang Slany. *Fuzzy Scheduling*. PhD thesis, Technical University of Vienna, 1994. URL: ftp://ftp.dbai.tuwien.ac.at/pub/slany/thesis.ps.Z.

[2] Wolfgang Slany. Scheduling as a fuzzy multiple criteria optimization problem. CD-Technical Report 94/62, Christian Doppler Laboratory for Expert Systems, Technical University of Vienna, 1994. A revised version has appeared in *Fuzzy Sets and Systems* 78 (1996). URL: ftp://ftp.dbai.tuwien.ac.at/pub/papers/slany/cd-tr9462.ps.gz.