Toward automatically learned search heuristics for CSP-encoded configuration problems – results from an initial experimental analysis

Dietmar Jannach
Department of Computer Science, TU Dortmund, Germany
dietmar.jannach@tu-dortmund.de

Abstract

Constraint Programming historically been one of the most important approaches for compactly encoding and solving product configuration problems. Solving complex configuration problems efficiently however often requires the usage of domain-specific search heuristics, which have to be explicitly modeled by domain experts and knowledge engineers. Since this is a time-consuming task, our long term research goal is to develop techniques to automatically learn appropriate search heuristics for a given configuration problem.

Compared to other types of Constraint Satisfaction Problems (CSPs), practical configuration problems have certain specific characteristics. First, often only a few of the variables are used to specify the problem ("inputs"); in addition, the specific user inputs and the corresponding final configurations are not equally distributed in the solution space.

In this paper, we present results of an initial simulation-based experimental analysis, in which we aimed to evaluate if already simple statistics can help to speed up the search process. The first results indicate that already trivial branching statistics can help to improve search efficiency.

1 Introduction

Encoding product configuration problems as Constraint Satisfaction Problems (CSPs) [13] has a comparably long tradition both in research and in industrial practice. Using CSPs and Constraint Programming techniques has various advantages, compared, e.g., to rule-based systems, as CSP encodings are declarative in nature and thus often easier to maintain. Furthermore, a number of extensions to the basic CSP encoding scheme as well as specific solving techniques have been proposed in the past, which are at least partially inspired by the specific characteristics of product configuration problems, see [1], [5], [7] or [11]. Today, there also exists a number efficient free and commercial constraint solvers that can be used to check configurations for consistency or to complete partial configurations given some customer inputs.

However, some larger and complex configuration problems can only be solved efficiently when domain-specific heuristics are used that guide the search process. In [4], for example, Fleischanderl et al. report of a configuration problem in the context of telecommunication switches where the final configuration can comprise thousands of interconnected components. The so-called "Partner Units Problem" is another example of a hard real-world configuration problem, for which recently a heuristic algorithm was proposed which allows the problem to be solved in an efficient way [12].

Such heuristics are however domain-specific or even problem-specific and their identification, formalization and evaluation usually is a time-consuming and manual process. It would therefore be desirable to have domain-independent techniques, which help us to automatically derive appropriate heuristics for a given problem setting. In principle, different approaches to achieve this goal are possible. First, one could try to analytically examine the configuration problem (or constraint network) and its solution space. Alternatively, one could follow a learning-based approach by analyzing a number of past solution searches in order to derive appropriate search heuristics.

In our ongoing research, we will focus on the latter type of systems. The long term goal of this research activity being to develop a set of methods that use a learning-based approach to derive search heuristics for configuration problems. We decided to follow the path of a learning-based approach for several reasons. First, in real-world applications, the configuration reasoning process (e.g., to complete a partial configuration) is initiated several times, so that more and more training data will be naturally available over time and the heuristics can thus be made self-adaptive. Furthermore, in many domains, the actual configurations requested by customers are not equally distributed in the solution space and there might be configurations which are far more popular than others[1]. When using a learning approach, such information, which might only be available once the system is deployed, can be integrated in the heuristics learning process.

In this paper, we report the results of an initial experimental analysis, in which we implemented a basic value ordering heuristic for CSPs, which simply ranks the possible variable values based on the number of times they were suc-

[1] In some complex configuration scenarios, every configuration might be unique; still, some subassemblies are usually similar or identical across different configurations.

Michel Aldanondo and Andreas Falkner, Editors
Proceedings of the 15th International Configuration Workshop
August 29-30, 2013, Vienna, Austria

cessfully chosen in previous configuration runs. Our evaluation is based on a simulation, in which we artificially and randomly generated inputs for a number of benchmark problems from a Constraint Solver competition. The corresponding solutions were used as an input for the "training" phase in which statistics were collected. The benchmark problems were then solved again based on this statistics-based heuristic. To compare the efficiency, the required running times were measured. Our initial results show that significant reductions for some types of problems can be achieved even when a very simple learning strategy is applied.

Overall, we consider our work to be first evidence for the general feasibility of such approaches in the configuration domain and as an initial step toward the development of more advanced learning strategies. Our basic technique can furthermore be used as a baseline in further experiments. Finally we propose an experimental evaluation protocol to evaluate the effectiveness of such learning-based approaches.

2 Approach and Initial Results

In our analysis, we focus on standard CSPs which are represented by a tuple $< V, D, C >$, where V is a set of variables, D a set of finite domain associated with these variables, and C a set of constraints on the variables. A solution to a CSP comprises an assignment of values to each problem variable in V such that no constraint from C is violated, see, e.g., [13] for a comprehensive discussion of CSPs.

2.1 The role of branching strategies

When systematic tree search is used as a problem solving scheme for CSPs, the choice of the branching strategy, that is, which variable to consider next and which values to try first, can have a significant impact on the required search time. Over the last decades, a number of different and often domain- or problem-independent branching heuristics have therefore been proposed to speed up the search process.

Consider the following example, which shall demonstrate the impact of the branching strategy on the solution efficiency. When searching for one solution for the classical "all-interval series" problem[2] of a given size without any problem specific optimizations, the running times when using the popular Choco[3] constraint solver with different built-in heuristics range from 30ms to 1 minute. Interestingly, the best strategy for this setting seems to be to pick variable values in decreasing order (30ms), which is an order of magnitude faster than the usual "increasing domain" strategy (500ms) or the dynamic "impact-based branching" [9] strategy (800ms). When no specific strategy is explicitly defined, the search can take up to one minute.

In many cases, the question of which heuristic to chose for a certain problem setting cannot be easily decided analytically and requires an experimental analysis or can be explored with a so-called portfolio solver. Such portfolio solvers, which try out different solvers and corresponding solving strategies based on a case base of past solution searches for similar problem instances, have shown to be very successful in CSP Solver competitions [8].

2.2 Proposed baseline and experimental setup

For our experiments, we extended the open source constraint solver Choco and implemented a new CSP value ordering strategy called `MostFrequent`, which picks the next value to test in the search process simply based on the number of occurrences of this value in solutions in previous search runs[4]. When considering, for example, a PC configurator, if "Intel Core i5" was the most frequent choice in previous configuration sessions, the solver would simply try to use this value first when asked to complete a partial configuration. While in reality the choice of the CPU of course depends on the specific requirements of the current customer, our underlying assumption is that not every possible configuration is equally popular. Therefore, if the specific CPU type was compatible with a larger number of popular and frequent configurations, it might be helpful to try this particular value first (as long as it is not already ruled out by other constraints specified in the current session)[5].

In order to evaluate this approach, we conducted experiments in which we measured the running times when using different branching strategies for a number of benchmark problems of the CPAI'08 solver competition[6]. As a baseline in our comparison the typical built-in `IncreasingDomain` default strategy was used. The chosen benchmark problems used in our experiments, see Section 2.3, had to fulfill certain properties. First, they of course had to be solvable. Furthermore, as we had to run a larger number of solution searches, e.g. to factor out random factors, we picked problems for which the solver could determine a solution (or report infeasibility) for a given set of random inputs relatively quickly, i.e., in less than a second[7]. The experiment consisted of two phases.

(A) Statistics collection phase. Depending on the size of the problem, we randomly designated a small number of the problem variables to be input variables. When the CSP for example contained 50 variables with an average domain size of 20, we, e.g., picked up to 5 variables as inputs, so that we could make sure that several thousand input combinations (and resulting configurations) are possible. Next, we created random input values for these variables, started a solution search and recorded the variable assignments, if a solution was found. In order to simulate that some configurations are more popular than others, we picked the input values using a Gaussian distribution and repeated the process until 300 solutions with the default branching heuristic were found. As we are also interested how the number of training instances effects the statistics-based approach, we defined different measurement points during the simulation run, in which we made

[2] The goal is to find permutations of a given list of numbers that fulfills certain properties, see `http://www.cs.st-andrews.ac.uk/~ianm/CSPLib/prob/prob007/refs.html`

[3] `http://www.emn.fr/z-info/choco-solver/`

[4] Technically, we used Choco's built-in extension mechanism and implemented a new `ValIterator` class.

[5] Note that the general solution space for a given configuration problem is not affected by the different heuristics.

[6] `http://www.cril.univ-artois.fr/CPAI08/`

[7] We furthermore excluded benchmark problems which could not be imported by Choco's current XML import program.

	Problem name	Default	30	50	100	150	200	300	Diff.
1	normalized-renault-mod-0_ext.xml	143,44	37,03	26,20	26,60	28,72	28,70	30,77	-82%
2	normalized-bibd-8-14-7-4-3_glb.xml	11,71	7,65	6,94	7,71	7,64	8,38	6,48	-41%
3	normalized-squares-9-9.xml	53,75	41,88	44,30	41,72	42,40	43,32	44,90	-22%
4	normalized-geo50-20-d4-75-29_ext.xml (less inputs)	431,25	353,07	366,06	327,03	354,16	342,25	352,29	-24%
5	normalized-geo50-20-d4-75-24_ext.xml	95,11	99,64	91,66	87,19	93,27	100,32	99,39	-8%
6	normalized-costasArray-13.xml	53,36	48,33	47,70	47,18	47,13	46,96	46,49	-12%
7	normalized-air05.xml	856,51	847,22	859,04	850,81	854,80	848,71	853,36	-1%
8	normalized-magicSquare-5_glb.xml (GAUSSIAN)	112,59	142,86	149,65	145,17	140,57	147,66	145,05	25%
9	normalized-magicSquare-5_glb.xml (RANDOM)	122,81	154,49	144,64	137,90	141,06	131,88	139,49	7%

Figure 1: Measurements for example problems. Running times are given in milliseconds.

a snapshot of the collected statistics so far. These snapshots were taken after 30, 50, 100, 150, 200 and 300 solutions. As a baseline for the required running times using the default strategy, we calculated the average search time for the 300 solutions.

(B) Measuring the effects. After the training phase, we repeated the experiment with random inputs 300 times. This time we however used the statistics-based value selection strategy and using the training data for each of the snapshots to analyze the effect when different amounts of training data was available. For cases when individual values never appeared in a previous solution, we used the `IncreasingDomain` strategy as a fallback.

Note that we used the same set of input variables in that phase as in the training phase, but did *not* use exactly the same input values. Instead, we again generated them randomly. Otherwise, a simple solution caching technique would have obviously led to the best results.

Since the total time of finding a solution for many problems depends on the specific set of variables used as an input set, we repeated the whole above-described procedure for 5 times, each time with a different set of input variables. Each problem therefore had to be solved successfully $(300 + 300) * 5 = 3000$ times, which explains why we only considered problems which could be solved efficiently.

2.3 Initial results

As a performance indicator, we used the average CPU time needed for the solver to find a solution. We also collected statistics about the standard deviation of the different runs. Figure 1 shows the average running times for 300 runs using the default strategy and the running times for the `MostFrequent` strategy at different training levels.

The first row shows the results of a benchmark car configuration problem from Renault, which in our view is therefore the most relevant of all measurements. The problem has 111 variables and an average domain size of around 5. As inputs, we used 6 variables, which leads to a range of $5^6 = 15,625$ input combinations. The number of possible configurations is actually lower as not all input combinations correspond to feasible product variants. In this case, about one third (about 5,000) of the randomly generated input combinations were feasible.

Using the solver's built-in default value selection heuristic, the problem could on average be solved in 134.44 milliseconds. When using the statistics-based strategy, however, the average running times could be reduced to 26.20ms, which is a reduction of over 80%. Interestingly, this effect could be achieved already after a very small number of training runs. Later on, when more training data is available, the values seem to slightly increase again. As we did not measure if these differences are statistically different so far, the slight increase could however be a random effect.

When using the default branching strategy, the standard deviation for all experiment runs for the Renault problem was around 220ms. With the statistics-based strategy, this value could be reduced to the half of that. However, when compared to the absolute overall running times, the standard deviation is much higher for the statistics-based strategy. This in general means that some problems can be very efficiently solved with the statistics-based approach, while in some cases the statistics-based strategy can also lead the solver to wrong areas of the search space[8]. Overall, however, the average number of required backtracks, which we measured but do not report here for space reasons, could also be reduced to a third using the statistics-based strategy.

The other problems of our analysis shown in Figure 1 have different characteristics and are in particular not configuration problems but other types of general constraint problems. Problem 2 in Figure 1, for example, is an artificially created one and has over 500 variables and 400 constraints. Also in this case, a significant reduction of the running times could be observed. Problems 4 and 5 are quite similar, but in the case of problem 5 we used many more variables as inputs which led to a drastically higher number of possible inputs while at the same time the solution search was more constrained. As a result, the improvements for Problem 5 were very small. For Problem 7, no improvement was observable. Problems 8 and 9 are classical magic square problems with a highly symmetric problem structure that does not correspond to the typical characteristics of configuration problems. In particular for case 9, in which the inputs were chosen from an equal

[8] In all experiments we limited the allowed computation time to avoid effects of extreme outliers. In the Renault example, the time limit was set to 1,000ms. Situations in which the time frame was not sufficient were however very rare.

distribution, nearly no improvement was achieved with the statistics-based strategy, because every variable value has an nearly equal probability to appear in a random solution. For case 8, a slight improvement could be observed because the inputs values were chosen using a Gaussian distribution.

3 Previous and future works

To the best of our knowledge, limited research has been done so far on the automatic derivation of search heuristics in the area of product configuration. There are, however, approaches in the area of general CSPs that include a learning component in the search process. In the context of our work, we are particularly interested in approaches which aim to learn from previous solution searches or similar problem instances (in contrast to works which try to adapt the search strategy within a single search, often referred to as "online learning", or based on multiple restarts).

In [3], for example, the authors propose an "advice generation" framework for value ordering in CSPs which is based on estimating the likelihood of the existence of at least one solution in the area of the search graph to be explored. Such estimates can be obtained by analyzing a simplified and backtrack-free version of the problem, where the hope is that the number of solutions in the simplified version with less constraints correlates with the solution count for the original problem. Overall, while the general goal of their work is similar to ours, the approach in [3] is not based on past solutions but from an analysis of adapted problem instances, which can also induce significant additional computational costs. In our work, we assume that the solver can learn by collecting information from previous search runs for different users.

In the area of Answer Set Programming, Balduccini in [2] presents an approach for learning branching heuristics from past solution instances. Similar to our work, the proposed DORS framework aims to derive a problem-specific *policy* to guide the solving procedure, i.e. which branch of the search graph should be explored first. While there are differences related to the actual search procedures, the general idea of [2] corresponds to the work presented in this paper. Our future work includes an analysis of how the more advanced but still not very complex approach from [2] can be integrated in the CSP solving process.

Finally, the idea of deriving heuristics from past observations in an automated way can be also found in other application areas. In [10], for example, supervised machine learning techniques are used to at least partially automate the construction of heuristics for the NP-complete problem of instruction scheduling on modern processors. While the specific relation to our work is limited, our future work includes the exploration of techniques such as decision tree learning or classification, see e.g., [6], for the generation of branching heuristics for configuration problems.

4 Summary

Problem-specific search heuristics are a key element for the practical success of many configurator applications. Since the manual definition of such heuristics is time-consuming, our research goal is to develop techniques that help us to learn such heuristics automatically from previous solution searches. In this paper, we have reported results of an initial analysis of the general feasibility of such an approach based on a simulation with small examples and a simple statistics-based branching strategy. Our first results indicate that measurable efficiency improvements can be achieved, when the special characteristics of configuration problems are taken into account.

References

[1] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.

[2] M. Balduccini. Learning and using domain-specific heuristics in ASP solvers. *AI Commun.*, 24(2):147–164, 2011.

[3] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artif. Intell.*, 34(1):1–38, 1987.

[4] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction. *IEEE Intelligent Systems*, 13(4):59–68, 1998.

[5] A. Haselböck. Exploiting interchangeabilities in constraint-satisfaction problems. In *IJCAI'03*, pages 282–289, Chambery, France, 1993.

[6] H. Ingimundardottir and T. P. Runarsson. Supervised learning linear priority dispatch rules for job-shop scheduling. In *Proc. LION 2011*, pages 263–277, Rome, Italy, 2011.

[7] D. Jannach and M. Zanker. Modeling and solving distributed configuration problems: A CSP-based approach. *IEEE TKDE*, 25(3):603–618, 2013.

[8] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Proc. AICS 2008*, 2008.

[9] P. Refalo. Impact-based search strategies for constraint programming. In *Proc. CP 2004*, pages 557–571, Toronto, Canada, 2004.

[10] T. Russell, A. M. Malik, M. Chase, and P. van Beek. Learning heuristics for the superblock instruction scheduling problem. *IEEE Trans. on Knowl. and Data Eng.*, 21(10):1489–1502, 2009.

[11] T. Soininen, E. Gelle, and I. Niemelä. A fixpoint definition of dynamic constraint satisfaction. In *Proc. CP'99*, volume 1713, pages 419–433, Alexandria, Virginia, USA, 1999.

[12] E. Teppan, G. Friedrich, and A. A. Falkner. QuickPup: A heuristic backtracking algorithm for the partner units configuration problem. In *Proc. AAAI/IAAI 2012*, Toronto, Canada, 2012.

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