A Systematic search strategy for product configuration

Xie, H.; Henderson, P.; Neelamkavil, J.; Li, J.

Proceedings of the 17th International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems Manufacturing (IEA): 01 January 2004, Ottawa, Canada, pp. 718-727, 2004-01-01

A Systematic search strategy for product configuration

IMTI-XP-216

Xie, H.; Henderson, P.; Neelankavil, J.; Li, J.

A version of this document is published in / Une version de ce document se trouve dans:
Proceedings of the 17th International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems Manufacturing (IEA), Ottawa, Ontario, Jan. 1, 2004, pp. 718-727

A Systematic Search Strategy for Product Configuration

Helen Xie[1], Philip Henderson, Joseph Neelamkavil and Jingxin Li

Integrated Manufacturing Technologies Institute
National Research Council of Canada
800 Collip Circle, London, Ontario, Canada N6G 4X8

Abstract. Constraint satisfaction problem (CSP) paradigm has proven highly successful in product configuration, particularly for build-to-order products, by assigning component types to all components without violating any constraints. For engineer-to-order products, however, product configuration requires assigning design parameters to each component as well. Hence, it often involves numeric variables, n-ary constraints, and constraints over variables that depend on other variables. Thus, an efficient search strategy is needed to address these issues. In this paper, an extension to the CSP, called Dependent CSP, is proposed to accommodate the complex engineer-to-order product configuration and the search strategy. In the Dependent CSP, variables are categorized as independent variables and dependent variables so that, search space can be reduced by eliminating dependent variables. Backjumping search strategy is employed to search for a solution as effective as possible. An updating mechanism is designed to avoid repetitive and unnecessary variable updating and constraint evaluation. Several variable ordering heuristics are assessed and the most effective ones are chosen for solution implementation. By applying these strategies, we can achieve a very efficient search algorithm for product configuration. The algorithm has been applied in a product configuration problem – an elevator system design – and a configuration solution can be obtained in a matter of seconds.

Keywords: Constraint satisfaction, product configuration, numeric variables, n-ary constraints, dependent variables, backjumping, variable ordering

1. Introduction

The product configuration is an enabling technology in achieving mass customization to meet the challenges of global competition and customer satisfaction. It is intended to aid manufacturing companies to configure customized products quickly and efficiently by automating the configuration process as much as possible. The constraint satisfaction problem (CSP) paradigm has proven highly successful for product configuration. It provides several advantages [11] in terms of problem representation, algorithms, and result evaluation. First, the domain knowledge of product configuration can be represented in a declarative form, which makes the configuration problem easy to define and maintain. Second, the search strategies are

[1] Corresponding author. *E-mail address: helen.xie@nrc.gc.ca*

generic and domain independent. By experimenting with different search strategies, efficient search algorithms can be identified for a certain type of product configuration problem. Third, given an existing configuration, its accuracy can easily be verified by checking the consistency of constraints. Finally, CSP algorithms easily allow the generation of multiple alternative solutions for preference and optimization purposes.

A typical constraint satisfaction problem (CSP) is defined by a set of variables, $X = \{x_1,\ldots, x_n\}$, and a set of constraints, C, over these variables. An associated domain, D_i, contains possible values for x_i. A constraint $c\ (x_i,\ldots, x_j) \in C$ specifies a subset of the Cartesian product $D_i \times \ldots \times D_j$ indicating the variable assignments that are compatible with each other. A *solution* to a CSP is a complete *assignment* of values to the variables such that all constraints are simultaneously satisfied [7]. In a product configuration framework, component types are represented as variables with discrete and finite domains, and compatibilities of various components to form a valid configuration are represented as constraints [9]. For example, an elevator door is available in four models: SSSO, 2SSO, SSCO, and 2SCO, and an elevator platform can be chosen from three models: 2.5B, 4B, and 6B. The constraints for compatible models are represented as tuples: {SSSO, 2.5B}, {2SSO, 4B}, {SSCO, 4B}, {2SCO, 6B}. A valid configuration is constructed by choosing a door model and a platform model such that the combination appears in one of the tuples.

For engineer-to-order products, however, product configuration also requires assigning design parameters for each component. While the design parameters, including but not limited to component types and component quantities, are represented as variables, design constraints restricting design parameter assignments can be represented as constraints. Since constraints over design parameters may take a wide variety of formats, modeling and solving constraints for engineer-to-order products presents several challenges:

1. Variables may be defined with continuous and numeric domains.
2. Constraints may be n-ary, meaning more than two variables may appear in a constraint.
3. Constraints may be represented as mathematical expressions or computable procedures.
4. A constraint may be defined over variables whose existence depends on the values chosen for other variables. This type of constraint is called activity constraint [8] or conditional constraint [5].
5. More generally, a constraint may be defined over variables whose values cannot be independently assigned, because those variables have dependent relations with other variables outside the constraint. This type of constraint is called dependent constraint.

To address the issue of the conditional constraint, several constraint models have been proposed to extend the original configuration framework, including Dynamic CSP [8], Generative CSP [4][12], Composite CSP [10], and Mixed and Conditional CSP [5]. While the Dynamic CSP, Generative CSP, and Composite CSP models formulate the conditional constraint in such ways that effective search algorithms can be employed, the Conditional CSP model reduce the conditional constraints to a set of standard CSPs by analyzing dependencies between the conditional constraints and applying the conditional constraints in order.

The issues on n-ary constraints over numeric variables or mixed variables (both discrete and numeric variables) are also addressed in Mixed and Conditional CSP [5]. In order to achieve arc-consistency, the n-ary constraint can be decomposed into an equivalent network of ternary constraints. When all but two variables of the n-ary constraint have been instantiated, a binary refine operator can be applied.

In an n-ary dependent constraint, some variables (called dependent variables) may not be independently assigned values from their domains, as they depend on other variables through dependent relations. The dependent relations are often represented by mathematical expressions or computable procedures. In the dependent relations, some variables can be independently assigned values from their domains. Although these variables may not have appeared in the dependent constraint, they are considered as independent variables in the dependent constraint, since these variables have the potential to make the dependent constraint satisfied. In the dependent constraint, a constraint check cannot be done immediately when an independent variable is assigned a value, as its associated dependent variables have to be updated as well. Unlike variables of the conditional constraint which have only two values (existence or non-existence) to choose from, the dependent variables may have numeric domains. Converting the n-ary dependent constraint with numeric variables into a set of typical constraints with tuples may result in exponential increase in parameters and constraints [1]. Hence, a generic constraint model and efficient search strategy are needed for the n-ary dependent constraint.

The main contribution of this paper is to propose a generic constraint model and an efficient search strategy for product configuration with n-ary dependent constraints over numeric variables. Since a dependent constraint can be ultimately represented by a set of independent variables through dependent relations, and dependent variable cannot be independently assigned a value, the search space could be reduced by eliminating dependent variables. As dependent relations among variables are represented by mathematical expressions and computable procedures, their formats only become known later at product constraint modeling time. In order to make a search algorithm as generic as possible, a systematic search strategy is employed. Specifically, backjumping algorithm is chosen because of its success in avoiding thrashing (repeated failure due to the same reason) which is often the leading factor in search efficiency. During a search process, a consistency check is required, whenever an independent variable is assigned a new value. Since a constraint is linked to independent variables through dependent variables, the dependent variables are frequently updated. Thus, the updating of dependent variables becomes a bottleneck in search efficiency. An efficient updating mechanism is designed to avoid unnecessary updating. Moreover, several variable ordering heuristics were assessed and implemented. By applying these strategies, we can achieve a very efficient search algorithm for product configuration. The algorithm has been applied in a product configuration problem – an elevator system design, with excellent results: the search can be done in a matter of seconds.

The remainder of the paper is organized as follows. Section 2 defines the constraint model applicable to n-ary dependent constraints. In Section 3, we present a backjumping search algorithm for product configuration, provide a mechanism for updating dependent variables and constraints, and discuss variable ordering heuristics.

A case study for configuring an elevator system is presented in section 4. Finally, conclusions are given in section 5.

2. A Constraint Model for Product Configuration

As a search algorithm should be generic to any product configuration, a constraint model is necessary for defining product configuration. Here, product configuration can be represented by a Dependent Constraint Satisfaction Problem as follows:
Definition. A Dependent Constraint Satisfaction Problem is defined as <X, D, R, C>, where

- $X = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables,
- Each $x_i \in X_{in}$ can take its value from a finite domain D_i, where $D_i \in D$, $X_{in} \subseteq X$ is a set of independent variables,
- Each variable $x_j \in X_{de}$ depends on its dependent relation $r_j \in R$ to its ancestors $X_a \subseteq X$, $X_{de} \subseteq X$ is a set of dependent variables. $X=(X_{in} \cup X_{de})$, and $(X_{in} \cap X_{de})=0$.
- A set of constraints C restricts the combination of values that variables can take.
- A solution to a Dependent CSP is an assignment of a value from its domain to every variable from X, in such a way that every constraint from C and every dependent relation from R are satisfied.

Variables are modifiable during a search process to satisfy all the constraints, so that a solution can be found. According to the way variables can be modified, they can be classified as an independent variable (IV) or a dependent variable (DV). The independent variable is a variable that may be directly modified by a search algorithm. Its value can independently be assigned within its domain. The dependent variable depends on a dependent relation. It can only be derived from existing variables. Its value ultimately depends on independent variables and cannot be independently modified. An ancestor of a dependent variable is a variable whose value determines (at least in part) the value of the dependent variable. Direct ancestors can be any combination of IVs and DVs, but ultimately a dependent variable is defined by IVs. There are advantages to separating dependent variables from independent variables. First, only independent variables define the search space, so the number of possible combinations is dramatically reduced. Second, when a dependent variable is used by multiple constraints, it needs to be computed only once. These characteristics help to improve the efficiency of the search algorithm.

Variables can have numeric or non-numeric domains. Examples of numeric variables are choices relating to dimension or weight, while non-numeric variables can be the model of a part. Numeric variables are given a range of possible numeric values specified by a minimum, maximum, and an interval to be used from one value to the next, while non-numeric variables each have a list of possible values. The sequence of the values in their domain will determine the order in which values are tried in a search process.

Dependent relations are represented by mathematical expression or computable procedures, such as formulas, tables, etc. They specify design relations among independent variables and dependent variables or among dependent variables. An example of a dependent relation is shown as follows: *Counterweight Plate Weight =*

0.2816T(D(BG-2)-3.5(D-5)-6(D-7)), where *BG* is the *Distance Counterweight Between Guiderails* (an independent variable), *D* is the *Counterweight Plate Depth* (an independent variable), *T* is the *Counterweight Plate Thickness* (a constant), *and Counterweight Plate Weight* is a dependent variable .

Constraints specify the restrictions that must be satisfied for a solution. The restrictions may represent a logical requirement, physical requirement, compatibility among parts, safety regulations, or any other design requirement that may be required. A constraint may be extensionally represented as tuples, or intensionally described by mathematical expressions or computable procedures that indicate a valid or invalid assignment for consistency check. The difference between constraints and dependent relations is that constraints specify a limit, while dependent relations result in a value. For example, a constraint can be stated as follows: the *Platform Width* must be at least 60 inches. A constraint may apply to any number of variables, including any combination of IVs and DVs. However, since dependent variables are deterministically defined by independent variables, the constraint ultimately depends solely on independent variables. The number of independent variables that affect a constraint is called the constraint's arity. Sometimes, a independent variable may not appear in a constraint explicitly, since it may affect the constraint through dependent variables. Nevertheless, the relevant independent variables can still be identified by searching the ancestors of a constraint's dependent variables. The relationship between constraints and independent variables is many-to-many, meaning that a constraint may depend on multiple independent variables, and an independent variable may affect multiple constraints.

3. A Search Strategy

Once product configuration has been formulated as a constraint satisfaction problem, a solution can be found using search algorithms. Since they are represented by mathematical expressions or computable procedures, dependent constraints and their corresponding dependent relations can take a wide variety of formats. Hence, they are not known during algorithm design time. To provide a generic search algorithm for solving product configuration problems, we use systematic search strategies for product configuration.

A systematic search strategy incrementally extends a partial solution towards a complete solution by repeatedly choosing a value for another variable, consistent with the values in the current partial solution [2]. Since it traverses the search space systematically, the advantage is that a solution, if one exists, can eventually be found. Also, the algorithm is general and applicable to any configuration design problems. As previously described, dependent variables can be eliminated from search space. Thus, only independent variables are considered as variables in search algorithms.

Backtracking is a primary algorithm in systematic search. It has two phases: a forward phase in which the next variable is selected and the current partial solution is extended by assigning a consistent value, if one exists for the next variable; and a backward phase in which, when no consistent solution exists for the current variable, attention returns to the previous variable assigned [3]. Backtracking suffers the drawback of thrashing, i.e. repeated failure due to the same reason. The efficiency of backtracking algorithm was improved by backjumping, a proper updating mechanism, and variable ordering in configuration design.

3.1 Backjumping

Backjumping improves on backtracking by analyzing the reasons for a dead-end and jumping back to the appropriate variable. In backtracking, a dead-end is encountered when a consistent value cannot be found for the next variable (i.e. the current partial solution cannot be extended). Instead of just going back to the preceding variable in the ordering, the backjumping algorithm tries to identify the source of failure and prunes a large portion of search space without missing any potential solutions. To help determine an appropriate backtrack point, we discuss the following three situations:

1. *A dead-end variable breaks one unary constraint.* The unary constraint cannot be affected by other variables, so if this is the case, this constraint shall never be satisfied and the CSP is impossible to solve.
2. *A dead-end variable breaks one constraint with n-ary variables.* The broken constraint has one or more variables that can affect it (excluding the dead-end variable). Although any of these variables could be modified, the algorithm should not skip any possible solutions. Thus, the algorithm should jump back to the closest previous variable for this constraint. If the algorithm moves farther back, it may skip a potential solution, and any jumps that do not go back beyond this point will be futile since this constraint will fail again.
3. *A dead-end variable breaks more than one constraint with n-ary variables.* In this case, the values of the variable can not become valid unless all broken constraints are affected. Thus, jumping back to the closest previous variable among all broken constraints is not adequate, since it does not affect the constraints whose variables appear before that variable. In order to ensure that all constraints are affected, the algorithm should jump back to the farthest variable, called the cutoff variable, among the closest variables of all broken constraints. The algorithm cannot jump back farther without facing the risk of missing potential solutions. After jumping back, still, if none of the values are compatible with at least one constraint for the cutoff variable, then the algorithm should jump back to the closest variable among any connected constraints for the new current/dead-end variable.

3.2 Updating mechanism for consistency checks

While choosing appropriate backtrack points could potentially prune a large portion of search space, determining the timing for constraint consistency checks can also improve the efficiency of the search algorithm. It is necessary to have an updating mechanism that identifies which constraints and dependent variables have been affected by the change of an independent variable's value, and updates their status accordingly. The efficiency of the updating mechanism has a major impact on the overall efficiency of the search algorithm, since updating is performed frequently (every time an IV's value is changed).

However, enabling each dependent variable of a constraint to re-compute itself does not guarantee the values will be properly updated, as the constraint and dependent variables need to be updated after their ancestors. In our previous approach [13], a list of dependent variables is stored for each variable (either independent variable or dependent variable) as its direct descendants. Whenever an independent variable is modified, it calls the update procedures of its direct descendants, which in turn call the update procedures of their direct descendants, and so on. In this way, every dependent variable will be properly updated and correctness of the constraint's status is guaranteed. However, this approach may still update a dependent variable more than once. For example, suppose we have (A -> B, C) and (C -> B). An arrow indicates dependency: (A -> B) means that B is a direct descendant of A (or, equivalently, A is a direct ancestor of B). Once A is updated properly, and B's update procedure is called and followed by C's update procedure since B and C are A's direct descendants. However, B is C's direct descendant as well, so B's update procedure shall be called once more, right after C's update procedure. Consequently, B's update procedure was called more than once, because A does not know which of its direct descendants to be updated first.

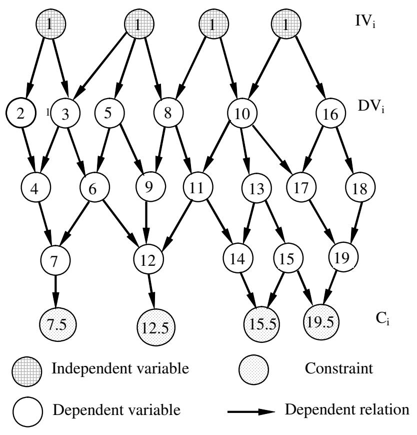

Fig. 1. A directed acyclic graph of independent variables, dependent variables and constraints

In our current approach, the dependencies among variables and constraints are considered as arcs in a directed graph, where variables and constraints are nodes and there is an arc from every variable to each one of its descendants (Fig. 1). In the directed graph, independent variables do not have any ancestors, and constraints do not have any descendants. Dependent variables can only be derived from independent variables or other dependent variables that in turn are eventually derived from independent variables. This directed graph can be shown to be acyclic, implying that a topological ordering exists. A topological ordering of the directed acyclic graph provides an updating order which guarantees that each variable and constraint's status are correct and need to be computed only once. Multiple topological orderings are valid, but the recommended ordering is formed by minimizing the value given to constraints and dependent variables, so that constraints can be evaluated as early as possible.

As previously described, constraints are often indirectly linked to independent variables through dependent variables in configuration design. Also a dependent variable may depend on one or more independent variables. Thus, if any of the independent variable ancestors for a dependent variable is not instantiated, the dependent variable cannot be used as an authentic source for evaluating associated constraints. The evaluation of a constraint has to wait until the last independent variable ancestor is instantiated. Using the criteria for determining proper timing for consistency checks, unnecessary repeated updating can be avoided.

3.3 Variable Ordering

The performance of the backjumping algorithm can also be improved by choosing the order of variable instantiation [6]. In the algorithm, variable ordering is used as a pre-processing technique. A fixed order is determined by heuristic approaches prior to starting of the search. Several heuristics have been analyzed for selecting variable order. One consideration is the variable's degree, a number of variables that are connected with it. The maximum degree variable is instantiated first. If variables are tied in the first heuristic, then a variable with the fewest domain values would be chosen as a secondary heuristic. However, the success of these heuristics is not independent of the specific product configuration problem; hence, the search algorithm may have to try several orderings before finding a good variable ordering.

4. A Case Study and Experimental Results

To exam the efficiency of the constraint model and search algorithm described above, we have tested a configuration design problem—configuring elevator systems [14]. The configuration process begins with a list of customer requirements, such as elevator car capacity and speed, and building dimensions. To configure an elevator system, one must assign a set of variables that satisfies both customer requirements and design constraints. In product configuration problems, not all variables are compatible, and certain combinations may not meet functional or safety regulations. The algorithm has to modify variables until it achieves a valid configuration.

In order for the search algorithm to find a valid solution for the elevator design, it is necessary to generate associated product definitions in the constraint model. There are 241 variables in the elevator system. Among these, there are 32 independent variables (such as platform model and counterweight buffer quantity), and 184 dependent variables (such as counterweight quantity and hoist cable quantity), and 25 input variables (such as car capacity and car speed). Input variables capture customer requirements and are considered as fixed values upon entering the system. There are also 50 constraints that establish criteria for functional and safety regulations, which guide the search algorithm to find a valid solution. In addition to variables and constraints, there are also dependent relations, such as mathematical expression or tables, between dependent variables and independent variables. These relations define how the dependent variables are derived from independent variables.

The backjumping search algorithm was implemented in Java using IBM VisualAge for Java 4.0. It solved almost all elevator configuration problems that we tried in less than 15 seconds. The only problematic case is when car capacity and car speed inputs are set to their maximum possible values. For this scenario, the best variable ordering (we could find) took 50 seconds, whereas the automated variable ordering never completed the search. A series of tests was performed on an Intel Pentium 4 CPU, 1.8GHz, and 1G RAM running on Windows 2000. The results below show that the algorithm works quite well with an automated variable ordering. Note that these results are the slowest test cases found for the given car capacity and car speed. For instance, other test cases with car capacity at 4000lbs and car speed set to 400 feet per minute found solutions in 10-15 seconds.

Table 1. Backjumping results (with automated variable ordering)

Worst-case time found (seconds)		Car Speed (feet per minute)				
		200	250	300	350	400
Car Capacity (pounds)	2000	1.5	----	----	----	3.4
	3000	----	1.5	----	4.5	6.2
	4000	2.4	----	2.5	----	Forever

A web-based application prototype system was implemented using this algorithm. The system allows the customer to enter requirements, and displays the final configuration results back to the customer through the Web. The Web application was deployed on IBM WebSphere Application Server.

5. Conclusions

Market trends that affect today's competitive environment are changing dramatically. Mass production of identical products - the business model for industries in the past - is no longer viable for many sectors. Customized products offer great market potential to manufacturers in the current climate of global competition and improved customer satisfaction. The complexity of products brings along new demands for configuration technology to cope with search efficiency. However, commercially available configuration systems only support build-to-order type product configuration in which constraints are represented by tuples. In this paper, an extension of the CSP paradigm was presented to cover dependent constraints with mathematical expressions in product configuration. The extension supports n-ary dependent constraints and variables with both discrete and numeric domains. Dependent variables are separated from independent variables to reduce search space. The updating mechanism proposed for dependent variables and constraints ensures correctness while avoiding repeated computations. The search algorithm is based on backjumping, a systematic search strategy. Specific backjumping situations were discussed to cover many-to-many relations between variables and constraints. Several heuristics of variable ordering were also applied for the backjumping search algorithm. The implemented algorithm is capable of solving almost all elevator configuration problems within 15 seconds based on an elevator case study. The test results show that the algorithm works well with a good (automated) variable ordering heuristic. This approach can be easily applied to a wide variety of product configuration problems.

References

1. Bartak, R.: Theory and Practice of Constraint Propagation. Proceedings of CPDC2001 Workshop (invited talk). Gliwice (2001) 7-14
2. Bartak, R.: Constraint Programming: In Pursuit of the Holy Grail. Proceedings of Week of Doctoral Students (WDS99), Part IV. MatFyzPress, Prague (1999) 555-564
3. Dechter, R. and Frost, D.: Backjump-based Backtracking for Constraint Satisfaction Problems. Artificial Intelligence, Vol. 136 (2002) 147-188
4. Fleischanderl, G., Friedrich, G., Haselboeck, A., Schreiner, H. and Stumptner, M.: Configuring Large Systems Using Generative Constraint Satisfaction. IEEE Intelligent Systems, Vol. 13 (1998) 59-68
5. Gelle, E. and Faltings, B.: Solving Mixed and Conditional Constraint Satisfaction Problems. Constraints, Vol. 8, No. 2 (2003) 107-141
6. Kumar, V.: Algorithms for Constraint Satisfaction Problems: A Survey. AI Magazine, Vol. 13, No. 1 (1992) 32-44
7. Miguel, I. and Shen, Q.: Solution Techniques for Constraint Satisfaction Problems: Foundations. Artificial Intelligence Review, Vol. 15 (2001) 243-267
8. Mittal, S. and Falkenhainer, B.: Dynamic Constraint Satisfaction Problems. Proceedings of the 8th National Conference on Artificial Intelligence (1990) 25-32
9. Mittal, S. and Frayman, F.: Towards a Generic Model of Configuration Tasks. Proceedings of the 11th IJCAI, Detroit, MI (1989) 1395-1401
10. Sabin, D. and Freuder, E. C.: Configurations as Composite Constraint Satisfaction. Working Notes, AAAI Fall Symposium on Configuration, Boston (1996) 28-36
11. Stumptner, M.: An Overview of Knowledge-Based Configuration. AI Communications: The European Journal on Artificial Intelligence, Vol. 10, No. 2 (1997) 111-125
12. Stumptner, M. and Haselbock, A.: A Generative Constraint Formalism for Configuration Problems. 3rd Congress Italian Assoc. for AI. Torino, Italy. Lecture Notes in AI, Vol. 729. Springer-Verlag (1993) 302-313
13. Xie, H. and Lau, F.: Towards Engineer-to-order Product Configuration. Proceedings of the ISCA 15th International Conference, Computer Application in Industry and Engineering. San Diego, CA, USA (2002) 180-184
14. Yost, G. R. and Rothenfluh, T. R.: Configuring Elevator Systems. Int. J. Human-Computer Studies, Vol. 44 (1996) 521-568