Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 4, Issue. 5, May 2015, pg.33 – 38

RESEARCH ARTICLE

CONSTRAINT SATISFACTION PROBLEM: A CASE STUDY

Jyoti[1], Tarun Dalal[2]

[1] M.Tech student, CBS Group Of Institutions, Jhajjar, Haryana, India

[2] Assistant Professor, CBS Group Of Institutions, Jhajjar, Haryana, India

[1] jyotigulia06@gmail.com, [2] tarundalal88@gmail.com

Abstract

Constraint programming (CP) is a powerful paradigm for solving combinatorial problems. CP was born as a multidisciplinary research area that embeds techniques and notions coming from many other areas, among which artificial intelligence, computer science, databases, programming languages, and operations research play an important role. It requires a value, selected from a given finite domain, to be assigned to each variable in the problem, so that all constraints relating the variables are satisfied. Researchers in artificial intelligence (AI) usually adopt a constraint satisfaction approach as their preferred method when tackling such problems. The aim of this paper is to give brief description of constraint satisfaction problems (csps). We start by defining CSPs, its variants, and describing the basic techniques for solving them.

Keywords**:** constraint satisfaction problem, domain set, csp variants, search techniques, problem complexity, algorithms, etc.

1. Introduction

A large number of problems in AI and other areas of computer science can be viewed as special cases of the constraint-satisfaction problem. Some examples are machine vision, belief maintenance, scheduling, temporal reasoning, graph problems, floor plan design, the planning of genetic experiments, and the satisfiability problem. A number of different approaches have been developed for solving these problems. A constraint satisfaction problem (CSP) requires a value, selected from a given finite domain, to be assigned to each variable in the problem, so that all constraints relating the variables are satisfied. Many combinatorial problems in operational research, such as scheduling and timetabling, can be formulated as CSPs. Researchers in artificial intelligence (AI) usually adopt a constraint satisfaction approach as their preferred method when tackling such problems

II. Constraint Satisfaction Problems (CSPs)

CSPs are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. CSPs are the subject of intense research in both artificial intelligence and operations research, since the regularity in their formulation provides a common basis to analyze and solve problems of many unrelated families. CSPs often exhibit high complexity, , requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. The boolean satisfiability problem (SAT), the Satisfiability Modulo Theories (SMT) and answer set programming (ASP) can be roughly thought of as certain forms of the constraint satisfaction problem.
Examples of simple problems that can be modelled as a constraint satisfaction problem

- Eight Queens Puzzle
- Map Coloring Problem
- Sudoku,Futoshiki,Kakuro (Cross Sums), Numbrix,Hidato and many other logic puzzles.

A. Problem Definition

Formally, a constraint satisfaction problem is defined as a triple $\langle X, D, C\rangle$, where

$X = \{X_1, \dots, X_n\}$ is a set of variables,
$D = \{D_1, \dots, D_n\}$ is a set of the respective domains of values, and
$C = \{C_1, \dots, C_m\}$ is a set of constraints.

Each variable X_i can take on the values in the nonempty domain D_i. Every constraint $C_j \in C$ is in turn a pair $\langle t_j, R_j\rangle$, where $t_j \subset X$ is a subset of k variables and R_j is an k-ary relation on the corresponding subset of domains D_j. An *evaluation* of the variables is a function from a subset of variables to a particular set of values in the corresponding subset of domains. An evaluation v satisfies a constraint $\langle t_j, R_j\rangle$ if the values assigned to the variables t_j satisfies the relation R_j.
An evaluation is consistent if it does not violate any of the constraints. An evaluation is complete if it includes all variables. An evaluation is a solution if it is consistent and complete; such an evaluation is said to solve the constraint satisfaction problem.

B. Constraints

The constraints of a CSP are usually represented by an expression involving the affected variables, e.g. x1 =! x2, 2x1 = 10x2 + x3 and x1 x2<x3.
Formally, a constraint Cijk ....between the variable xi, xj, xk is any subset of the possible combinations of values of xi, xj, xk..i.e. Cijk...is subset of D1* D2* D3*... The subset specifies the combinations of values which the constraint allows. For example, if variable x has the domain{2, 3} and variable y has the domain {1; 2} then any subset of {(1; 1);(1; 2); (2; 1);(2; 2);(3; 1);(3, 2)} is a valid constraint between x and y. The constraint x= y is equivalent to the subset {(1,1), (2,2)}.Although the constraints of real problems are not represented this way in practice, the definition does emphasize that constraints need not correspond to simple expressions, and, in particular, they need not be linear inequalities or equations (although they can be). A constraint can affect any number of variables from 1 to n, where n is the number of variables in the problem. The number of affected variables is the arity of the constraint.

III. Variants Of CSPs

The classic model of Constraint Satisfaction Problem defines a model of static, inflexible constraints. This rigid model is a shortcoming that makes it difficult to represent problems easily Several modifications of the basic CSP definition have been proposed to adapt the model to a wide variety of problems.

A. Dynamics CSPs

Dynamic CSPs(DCPs) are useful when the original formulation of a problem is altered in some way, typically because the set of constraints to consider evolves because of the environment. DCSPs are viewed as a sequence of static CSPs, each one a transformation of the previous one in which variables and constraints can be added (restriction) or removed (relaxation). Information found in the initial formulations of the problem can be used to refine the next ones. The solving method can be classified according to the way in which information is transferred:

- **Oracles:** the solution found to previous CSPs in the sequence are used as heuristics to guide the resolution of the current CSP from scratch.
- **Local Repair:** Each CSP is calculated starting from the partial solution of the previous one and repairing the inconsistent constraints with local search.
- **Constraint recording:** new constraints are defined in each stage of the search to represent the learning of inconsistent group of decisions. Those constraints are carried over the new CSP problems

B. Flexible CSPs

Classic CSPs treat constraints as hard, meaning that they are imperative (each solution must satisfy all them) and inflexible (in the sense that they must be completely satisfied or else they are completely violated). **Flexible CSP**s relax those assumptions, partially relaxing the constraints and allowing the solution to not comply with all them. This is similar to preferences in preference-based planning. Some types of flexible CSPs include:

- **MAX-CSP**, where a number of constraints are allowed to be violated, and the quality of a solution is measured by the number of satisfied constraints.
- **Weighted CSP**, a MAX-CSP in which each violation of a constraint is weighted according to a predefined preference. Thus satisfying constraint with more weight is preferred.
- **Fuzzy CSP** model constraints as fuzzy relations in which the satisfaction of a constraint is a continuous function of its variables' values, going from fully satisfied to fully violated.

C. Decentralized CSPs

In DCSPs each constraint variable is thought of as having a separate geographic location. Strong constraints are placed on information exchange between variables, requiring the use of fully distributed algorithms to solve the constraint satisfaction problem.

IV. Resolution Of CSPs

Constraint satisfaction problems (CSPs) can be solved by systematically exploring the solution space via an uninformed search. Such search algorithms instantiate variables one after the other in such a way that the current partial instantiation is always consistent. If this is not possible, that is, all the possible values for the next variable are in conflict with some earlier assignment, then backtracking takes place. Following, we present some well-known search techniques.

A. Complete Search Algorithms

Most algorithms for solving CSPs search systematically the space of all possible assignments of values to variables. Such algorithms are guaranteed to find a solution, if one exists, or to prove that the problem is insoluble. The disadvantage of these algorithms is that they may take a very long time. The actions of many search algorithms can be described by a search tree. Backtracking Search (BT) A simple algorithm for solving a CSP is backtracking search (B. Backtracking works with an initially empty set of consistently instantiated variables and tries to extend the set to a new variable and a value for that variable. If successful, the process is repeated until 3 all variables are included. If unsuccessful, another value for the most recently added variable is considered. Returning to an earlier variable in this way is called a backtrack. If that variable doesn't have any further values, then the variable is removed from the set, and the algorithm backtracks again. The simplest backtracking algorithm is called chronological backtracking because at a dead-end the algorithm returns to the previous variable in the ordering. In the BT method, as soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial solution violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. Clearly, whenever a partial instantiation violates a constraint, backtracking is able to eliminate a subspace from the Cartesian product of the variable domains.

1). Look-Back Algorithms

Chronological backtracking can suffer from thrashing; the same dead-end can be encountered many times. If Xi is a dead-end, the algorithm will backtrack to Xi−1. Suppose a new value for Xi−1 exists, but there is no constraint between Xi and Xi−1. The same dead-end will be reached at Xi again and again until all values of Xi−1 have been exhausted.

Look-back algorithms try to exploit information from the problem to behave more efficiently in dead-end situations. Like BT, look-back algorithms perform consistency checks backwards (between the current variable and past variables).

Backjumping (BJ) (Gaschnig, 1979) is an algorithm similar to BT except that it behaves in a more intelligent manner when a dead-end (Xi) is found. Instead of backtracking to the previous variable (Xi−1), BJ backjumps to the deepest past variable Xj , with $j < i$, that is in conflict with the current variable Xi . It is said that variable Xj is in conflict with the current variable Xi if the instantiation of Xj precludes one of the values in Xi . Changing the instantiation of Xj may make it possible to find a consistent instantiation of the current variable. Thus, BJ avoids the redundant work that BT does by trying to reassign variables between Xj and the current variable Xi .

Conflict-directed backjumping (Prosser, 1993), backmarking (Gaschnig, 1977), and learning (Frost & Dechter, 1994) are other examples of look-back algorithms. Look-Forward Algorithms As we have explained, look-back algorithms try to enhance the performance of BT by a more intelligent behavior when a dead-end is found. Nevertheless, they still perform only backward consistency checks and they ignore the future variables.

2). Look-forward algorithms

Make forward checks at each step of the search. Let us assume that, when searching for a solution, variable Xi is given a value which excludes all the possible values for a later variable Xj . When using uninformed search, this will only turn out when Xj will be considered for instantiation. Moreover, in case of BT, thrashing will occur: the search tree will be expanded again and again till Xj , as long as the level of backtracking does not reach Xi . Both anomalies could be avoided by recognizing that the chosen value for Xi cannot be part of a solution, as there is no value for Xj which is compatible with it. Lookahead algorithms do this, by accepting a value for the current variable only if after having looked ahead, it could not be seen that the instantiation would lead to a dead-end. When checking this, problem reduction can also take place, by removing the values from the domain of the future variables which are not compatible with the current instantiation. The algorithms differ in how far and thorough they look ahead and how much reduction they perform.
Forward-checking (FC) is one of the most common look-forward algorithms. It checks the satisfiability of the constraints, and removes the values of the future variables which are not compatible with the current variable's instantiation. At each step, FC checks the current assignment against all the values of future variables that are constrained with the current variable. All values of future variables that are not consistent with the current assignment are removed from their domains. If a domain of a future variable becomes empty, the assignment of the current variable is undone and a new value is assigned. If no value is consistent then backtracking is carried out. Thus, FC guarantees that at each step the current partial solution is consistent with each value in each future variable. Thus, FC can identify dead-ends and prune the search space sooner.

B. Incomplete Search Algorithms

Although complete search techniques, such as those described in the previous sections, always return a solution if there is one, or prove that there is no solution, we may sometimes want to use other techniques that don't possess this desirable property if they are more convenient from other points of view. Incomplete search methods (Michalewicz & Fogel, 2000) do not explore the whole search space. They search the space either non-systematically or in a systematic manner, but with a limit on some resource. These approaches do not ensure to collect all the solutions, nor to find a solution if there is at least one, nor to detect inconsistency, but their computational time can be much shorter compared to systematic search techniques.
Moreover, they may be sufficient when just some solution, or a good enough solution, is needed. These methods, known as metaheuristics, covers a very large class of resolution paradigms, from evolutionary algorithms to local search techniques. The main approaches for incomplete search are 4 based on constructive methods or on iterative repair methods. The first ones gradually extend a partial solution to a complete one, while the second ones start with an initial solution and incrementally modify the values to get a better one.
For instance, local search does not instantiate one variable at a time, but (in its simpler version) start with a complete assignment to all the variables, and then modifies it slightly to pass to a new complete assignment which is closer to be a solution, or closer to optimality.

C. Consistency Techniques

Consistency techniques were introduced to simplify CSPs and to improve the efficiency of systematic search techniques. The number of possible solutions can be huge, while only very few may be consistent. By eliminating redundant values from the problem definition, the size of the solution space decreases. Reduction of the problem can be done once, as a pre-processing step, or it can be interleaved with the exploration of the solution space by a search algorithm. Local inconsistencies are single values or combination of values for variables that cannot participate in any solution because they do not satisfy some local consistency property.
For instance, if a value a of variable x is not compatible with all the values in a variable y that is constrained with x, then a is inconsistent and this value can be removed from the domain of the variable x. In the following paragraphs we introduce the most well-known and widely used algorithms for binary CSPs. – A CSP is node-consistent if all the unary constraints are satisfied by all the elements of the variable domains. The straightforward node-consistency algorithm (NC), which removes the redundant elements by checking the domains one after the other, has O(dn) time complexity, where d is the maximum size of the domains and n is the number of variables. Thus, enforcing this consistency property ensures that all values of a variable satisfy all the unary constraints on that variable. – A CSP is arc-consistent if for any pair of constrained variables xi , xj , for every value a in Di there is at least one value b in Dj such that the assignment (xi , a) and (xj , b) satisfies the constraint between xi and xj , and viceversa. Any value in the domain Di of variable xi that is not arc-consistent can be removed from Di since it cannot be part of any solution. Arc-consistency has become very important in CSP solving and it is in the heart of many constraint programming languages.

The optimal algorithms to make the CSP arc consistent require time O(ed2), where e is the number of constraints (arcs in the constraint network) and d is the size of domains. Arc-consistency can also be easily extended to non-binary constraints. – A CSP is path-consistent, if for every pair of values a and b for two variables xi and xj such that the assignments of a to xi and b to xj satisfies the constraint between xi and xj , there exist a value for each variable along any path between xi and xj such that all constraints along the path are satisfied. When a path-consistent problem is also node-consistent and arc-consistent, then the problem is said to be strongly path-consistent. Consistency techniques can be exploited during the forward checking stage of search algorithms. Each time some search decision is taken (for example, a value is assigned to the variable), the problem is made arc consistent. If failure is detected (that is, any domain becomes empty) then it is not necessary to instantiate other variables and backtracking occurs immediately

V. Theoretical Aspects Of CSPs

A. Decision Problems

CSPs are also studied in computational complexity theory and finite model theory. An important question is whether for each set of relations, the set of all CSPs that can be represented using only relations chosen from that set is either in P or NP- complete. If such a dichotomy theorem is true, then CSPs provide one of the largest known subsets of NP which avoids NP-intermediate problems, whose existence was demonstrated by Ladner's theorem under the assumption that P =! NP. Schaefer's dichotomy theorem handles the case when all the available relations are Boolean operators, that is, for domain size Schaefer's dichotomy theorem was recently generalized to a larger class of relations. Most classes of CSPs that are known to be tractable are those where the hypergraph of constraints has bounded treewidth (and there are no restrictions on the set of constraint relations), or where the constraints have arbitrary form but there exist essentially non-unary polymorphisms[clarification needed] of the set of constraint relations.
Every CSP can also be considered as a conjunctive query containment problem.

B. Function Problems

A similar situation exists between the functional classes FP and #P. By a generalization of Ladner's theorem, there are also problems in neither FP nor #p-complete as long as FP ≠ #P. As in the decision case, a problem in the #CSP is defined by a set of relations. Each problem takes as input a Boolean formula as input and the task is to compute the number of satisfying assignments. This can be further generalized by using larger domain sizes and attaching a weight to each satisfying assignment and computing the sum of these weights. It is known that any complex weighted #CSP problem is either in FP or #P- hard.

VI. Conclusion

This paper gave the brief explanation of all the aspects of constraint satisfaction problems. The CSPs yield a natural representation for a wide variety of problem .A classification of various algorithms to solve the combinatorial problem is proposed..

References

[1] Duffy, K.R.; Leith, D.J. (August 2013), "Decentralized Constraint Satisfaction", IEEE/ACM Transactions on Networking, 21(4), pp. 1298–1308
[2] Stuart Jonathan Russell, Peter Norvig (2010). Artificial Intelligence: A Modern Approach. Prentice Hall. p. Chapter 6. ISBN 9780136042594.
[3] Lecoutre, Christophe (2009). Constraint Networks: Techniques and Algorithms ISTE/Wiley.ISBN 978-1-84821-106-3
[4] Bartak, R. 2005. Constraint Satisfaction for Planning and Scheduling. In Vlahavas, Vrakas (eds.): Intelligent Techniques for Planning, 320–353.
[5] Apt, Krzysztof (2003). Principles of constraint programming. Cambridge University Press. ISBN 0-521-82583-0.
[6] Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. Artificial Intelligence, 136(2):147–188, 2002. 3.3.1, 8
[7] Javier Larossa, Pedro Meseguer, and Thomas Schiex. Maintaining reversible DAC for MAXCSP. Artificial Intelligence, 107(1):149–163, 1999. 4.2
[8] Sally C. Brailsford, Chris N. Potts, Barbara M. Smith," Constraint satisfaction problems: Algorithms and applications" European Journal of Operational Research 119 (1999) 557-581
[9] Daniel Frost , Rina Dechter, Look-ahead value ordering for constraint satisfaction problems (1995), In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence
[10]Steven Minton Sterling Software AAAI-93 Proceedings. Copyright © 1993, AAAI, Integrating heuristics fr constraint satisfaction problem a case study.
[11] Vipin Kumar: Algorithms For Constraint Satisfaction Problems : A Survey,AI Magazine,13(1):32.44,1992.

[12] Arc-Consistency in Dynamic Constraint Satisfaction Problems Christian Bessière CBessiere-AAAI,1991-LIRMM.Fr.
[13] M. Haralick and G.L. Elliot. Increasing tree-search efficiency for constraint satisfaction problems. In Artificial Intelligence 14, pages 263–313, 1980. 3.3.1, 3.3.2
[14] Gaschnig, 1979]Gaschnig, J. 1979. Performance measurement and analysis of certain search algorithms. Carnegie-Mellon University: Technical Report CMUCS-79-124.