Whitepaper

IPROFS

Aircraft Scheduling, DNA Sequencing & Sudoku

Solving Constraint Satisfaction Problems with Java

Overview

Topics

Constraint satisfaction, constraint based reasoning, artificial intelligence, expert systems, decision support systems, software engineering and architecture, algorithms, complexity, search spaces, combinatorial explosion, constraint propagation, NP-complete problems, constraint relaxation, CSP frameworks, heuristics, constraint optimisation, meta-constraints.

Solution Overview

Generate fast and high-quality results for complex search problems where traditional, brute-force algorithms fail.

Applicable Industries

- Logistics
- Telecommunications
- Bioinformatics
- Data mining
- Planning
- Scheduling
- Resource Allocation
- Gaming

Frameworks Discussed

- CSP
- ILOG
- Choco CSP Framework

„Constraint Satisfaction represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it."
E.C. Freuder

Have you ever tried to schedule a meeting for a group of busy persons with lots of agenda constraints, solve a crossword or Sudoku puzzle or optimally allocate scarce resources to activities? If so, you have consciously or unconsciously been dealing with a Constraint Satisfaction Problem (CSP). Typically, these problems require some form of human intelligence to be solved adequately, and cannot be solved automatically by straight-forward algorithms within an acceptable amount of time. This paper discusses an effective way to solve these problems and examples of how to implement it in Java.

"Constraint satisfaction problems or CSPs are mathematical problems where one must find states or objects that satisfy a number of constraints or criteria. CSPs are the subject of intense research in both artificial intelligence and operations research. Many CSPs require a combination of heuristics and combinatorial search methods to be solved in a reasonable time."

Wikipedia.

Constraint Satisfaction Problems – An Example

Commercial Harbor

At a commercial harbor, schedules need to be made for loading and unloading 10 ships, using merely 5 docks. Many criteria exist for selecting the right dock for a specific ship. For example, some docks are too small for some ships, some ships need to be loaded prior to other ships, some docks are more expensive to use than others, the load of some ships is located nearer to some docks than to others, et cetera.

One possibility for finding a good or optimal schedule is generate and test - generate all possible permutations of ships and docks, select the solutions that are suitable, determine the cost of each correct solution, and finally select the most cost-efficient solution. Within this example, this means that 5^10 (approximately 10.000.000) possible alternatives need to be evaluated when using this brute force approach.

Assuming that the computer used to solve this problem is able to evaluate one alternative per, for example, millisecond, this problem will be solved within approximately 3 hours.

Business Is Booming – And The Technology?

Assume now that, 10 years later, business is booming, and the situation has evolved into 20 ships and 10 docks. Finding an optimal schedule now involves the evaluation of approximately 10^20 alternatives, which results in a processing time of approximately 3.000.000.000 years.

Of course, a 10 times more powerful processor might be purchased, reducing the processing time to a mere 300.000.000 years ;-) The obvious conclusion is that such or similar approaches do not work for these classes of problems. Transforming this basic brute force algorithm into a, for example, more elaborate backtracking algorithm will improve the situation a little, but not significantly.

Another (perhaps attractive, though naïve) strategy might be to split the harbor into two segments, and schedule each segment separately. This will result in a processing time of approximately 6 hours - which shows the enormous impact the number of variables has on these classes of problems. However, the fact is that the solutions within this approach will be far from optimal and, even worse, one does not even know how far from optimal, and thus how much money is being wasted. One might just as well run the 3.000.000.000 years program for 6 hours, and select the cheapest solution within its generated solution space. The result will most probably be of the same (poor) quality.

Using Constraints

By efficiently incorporating the constraints of this problem within the search algorithms, the search space will be decreased enormously. Within the above-mentioned situation, the application and propagation of, for example, the constraint 'dock length > ship length' alone within the problem model and search algorithms will in itself rule out millions of 'possibilities' a priori. Perhaps ironically, the more constraints that are imposed onto a CSP, the more complex the problem is perceived by humans, but the more efficiently the problem is solved by a computer when using Constraint Satisfaction strategies.

IPROFS

"Problems that can be expressed as constraint satisfaction problems are for example the Eight Queens puzzle, the Sudoku solving problem, the Boolean satisfiability problem, scheduling problems and various problems on graphs such as the graph coloring problem."

Wikipedia.

Constraint Satisfaction Problems – Overview

General

Constraint Satisfaction Problems (CSPs) [2] are, at their core, complex search problems which are stated in terms of variables, their domains and constraints on those variables.

Real-world examples of CSPs are:

- Bioinformatics: DNA sequence analysis
- Advanced planning and scheduling problems
- Resource allocation problems
- Logistics
- Natural language processing
- Circuit design
- Puzzles and games like Sudoku, Go, crossword puzzles, Chess, Checkers, Mastermind, Scrabble

From a complexity point-of-view, these problems are largely the same. What they have in common is the fact that they have extremely large search spaces, which tend to grow combinatorially when their number of variables grow – this characteristic attribute of CSPs is called combinatorial explosion. These problems are classified as so-called NP-complete problems, and are in general impossible to solve with traditional (e.g., brute force) algorithms within a reasonable amount of time (even with hardcore processors).

To this end, Constraint Satisfaction algorithms and CSP-specific heuristics have been developed which make intelligent use of the fundamental properties of CSPs. These algorithms decrease search spaces significantly by using elaborate constraint propagation strategies, prune search trees by cutting their dead branches and efficiently crawl through the resulting trees by using systematic search algorithms, thus enabling the efficient detection of solutions and avoiding sub-optimal solutions (local minima) within an acceptable amount of time.

Several CSP suites have been developed which assist software engineers in stating problems as the ones mentioned above as CSPs, by providing an abstraction and modeling layer, and which assist in solving them, by providing off-the-shelf CSP algorithms. The de-facto standard in commercial CSP solvers is the ILOG suite, which uses Java (amongst others) as implementation platform. There are also many open-source Java CSP solvers available, one of which will be discussed in further detail – the Choco framework.

Formal Definition

A Constraint Satisfaction Problem consists of:

- a set of variables $X=\{x_1,...,x_n\}$,
- for each variable x_i, a finite set D_i of possible values (its domain),
- and a set of constraints restricting the values that the variables can simultaneously take.

For example, the following simplistic CSP may be defined:

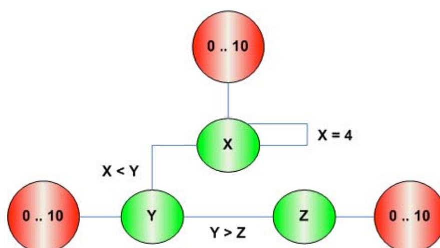
- variables: $V = \{X, Y, Z\}$,
- domains: $D_X = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, $D_Y = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, $D_Z = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$,
- constraints: $C_1 = [X = 4]$, $C_2 = [X < Y]$, $C_3 = [Y > Z]$.

Although this CSP is very basic, on a fundamental level it does not differ from the aforementioned CSP with regards to scheduling ships at a dock. The latter problem merely contains more variables, domains and constraints, which makes it more complex from a computational point of view.

> "Constraint satisfaction originated in the field of artificial intelligence in the 1970s. During the 1980s and 1990s, the use of constraints was embedded within programming languages."
>
> Wikipedia.

CSP Graph

The aforementioned CSP may also be visualised using a CSP graph:



This graph contains the three CSP variables, their respective domains, and the three mentioned constraints – one unary constraint on variable X, and two binary constraints between variables X and Y and between Y and Z. CSPs may also contain constraints of a higher order (e.g., tertiary constraints), but these can always be reduced to several binary constraints ('constraint binarisation'). Furthermore, please note that unary constraints may always be removed from a CSP by merely reducing the domains of the variables to which these unary constraints apply. Therefore, binary CSPs are all we need to focus on within the realm of CSPs.

Solutions

A solution to a CSP adheres to the following [2]:

- Each variable is initialised with a value from its domain,
- Each constraint is satisfied (i.e., no constraint is violated).

For CSPs, we may wish to find just one (random) solution, all solutions, or an optimal solution (with regards to a cost function defined in terms of the CSP variables).

The following is an example of a solution to the aforementioned CSP, which can easily be deducted:

- X = 4,
- Y = 6,
- Z = 5.

No computer is needed to solve this problem, and when a computer *is* used, a simple brute force, generate-and-test algorithm will suffice to find solutions quickly. The same goes for finding *all* solutions to this problem, and for finding optimal solutions when certain weights are assigned to certain solutions. One merely has to generate all possible permutations of X, Y and Z (called the search space, solution space or feasible region), and then select the correct (or optimal) solutions.

This is however not the case when the CSP contains many more variables and constraints. The problems with CSPs always lie in the numbers. Real-world CSPs may contain hundreds of thousands variables and constraints, and due to combinatorial explosion and the fact that these problems are NP-complete, applying brute force algorithms will not suffice.

The same goes for backtracking algorithms and other systematic search algorithms. Backtracking [2] is a refinement of brute force search, and is in fact nothing more than a depth-first search, implemented by using recursion or preferably iteration. During the search, if an alternative doesn't work, the search backtracks to the choice point, the place which presented different alternatives, and tries the next alternative. When the alternatives are exhausted, the search returns to the previous choice point and tries the next alternative there. If there are no more choice points, the search fails. Although this approach is more efficient than mere generate-and-test, it is still far from adequate for solving hardcore, real-world CSPs within an acceptable amount of time.

IPROFS

"Constraint satisfaction is a decision problem that involves finite choices. It is ubiquitous. A wide range of techniques have been applied to constraint satisfaction. This includes constraint propagation, optimisation, heuristics, complete search and local search methods such as neural network and evolutionary computation."

Constraint Programming and Optimisation Laboratory, University of Essex.

Tackling Constraint Satisfaction Problems

Overview

As a result of the aforementioned problems with regards to solving CSPs, elaborate algorithms, heuristics and strategies have been developed which are able to solve CSPs in an acceptable amount of time. The following will be discussed shortly [3]:

- Systematic search algorithms,
- Consistency techniques,
- Constraint propagation,
- Variable and value ordering.

Systematic search

Systematic search algorithms include generate-and-test and backtracking, as well as variations hereof. As mentioned, these algorithms are far from adequate for real-world CSPs. However, they are used as the basis for more elaborate algorithms, as discussed later.

Consistency techniques

Consistency techniques form effective strategies within the context of solving hard search problems and are used to prune the search space and cut off its dead branches prior to the actual performance of search within that space. In other words, assignments of values from domains to variables which *before-hand* are recognised as leading towards invalid 'solutions' (i.e., assignments which lead to constraint violations) are removed from the search space, thus limiting search to a smaller search space. Consistency techniques are deterministic and are performed prior to (non-deterministic) search algorithms. Two consistency techniques will be discussed – node consistency and arc consistency.

A variable (node) V in a constraint graph G is node consistent if every value X in the domain D of V satisfies all unary constraints C on V. In other words, an arc consistency algorithm crawls to all variables which are unary constrained, removes all values within the domains of those variables which do not comply with those unary constraints and subsequently removes those unary constraints (which have become redundant at this point). Applying a node consistency algorithm on a CSP results in a CSP with no unary constraints and smaller variable domains, thus easing the work for search algorithms. The CSP is said to be node consistent, as no possible inconsistent assignments of values to variables within the context of unary constraints exist.

At this point, no unary constraints exist anymore within the mentioned CSP, as a result of the fact that the CSP is node consistent. The next step consists of looking at the binary constraints within the CSP.

As mentioned, every n-ary constraint (of a higher order than two) can be transformed into multiple binary constraints, and as a result, we limit ourselves to binary constraints. If a CSP does however contain tertiary (or higher) constraints (e.g., X+Y=Z), these can automatically be transformed to binary constraints using algorithms (which will not be discussed within this paper).

As we are now merely working with CSPs containing binary constraints, we need to look at the consistency of these binary constraints. Within a CSP graph, a binary constraint is visualised as an arc between two nodes (variables), and therefore the next consistency technique is called arc consistency.

An arc (X, Y) between two nodes X and Y is arc consistent if for every value x in the domain of X there is some value y in the domain of Y such that X=x and Y=y is permitted by the binary constrain between X and Y. An arc consistency algorithm removes all values within the domains of variables that are binary constrained which do not adhere to this principle, thus easing the work for search algorithms.

IPROFS

"Constraint propagation techniques are methods used to modify a constraint satisfaction problem. More precisely, they are methods that enforce a form of local consistency, which are conditions related to the consistency of a group of variables and/or constraints. Constraint propagation has various uses. First, it turns a problem into one that is equivalent but is usually simpler to solve. Second, it may prove the (un)satisfiability of problems."

Wikipedia.

Constraint propagation

An extension to the aforementioned backtracking (depth first search) and consistency techniques consists of embedding a consistency algorithm within a backtracking algorithm, thus enabling a more intelligent way of search. Such search algorithms propagate the characteristics of constraints within the search space while they search, and are therefore called constraint propagation algorithms.

A backtracking algorithm instantiates the variables of a CSP one at a time, using the values within the variables' domains, and incrementally checks whether all constraints thus far are satisfied. In a sense, the backtracking algorithm tests arc consistency among already instantiated variables.

If a constraint is violated due to an incorrect instantiation of a variable, the backtracking algorithms detects this as this happens (real-time), and the algorithm tracks back to the previous choice-point within the search three where things were still ok. From that choice-point on, the algorithm now travels through another branch within the search tree, until all variables are instantiated and no constraints are violated.

Although this basic strategy is far more efficient than a basic generate-and-test approach, the problem is that constraint violations are merely detected when they happen, after which the algorithm needs to recover to a previous state of the search space, and then try alternative routes. As a result, a lot of (expensive) search still needs to be performed.

This problem can be tackled by preventing possible future conflicts (constraint violations) within a search space, which is more efficient than constantly having to recover from them. One constraint propagation strategy that does exactly this is called forwarding checking.

As mentioned, basic backtracking performs arc consistency to already instantiated variables in order to check whether a (partial) solution is a valid one. On the other hand, a forward checking algorithm performs (as its names suggests) arc consistency between the already instantiated variables and variables which are not yet instantiated. It does this one variable at a time, always from the perspective of the variable that is currently being instantiated. As soon as a variable is instantiated, any value within the domain of a 'future' variable which conflicts with this instantiation (value) is (temporarily) removed from the domain of that future variable. If the domain of a future variable becomes empty, the algorithms knows that the current partial solution is inconsistent. As a result, forward checking prunes invalid branches within the search tree earlier than basic backtracking, which results in more efficient search.

Variable and value ordering

Within CSP search algorithms, the order in which variables are visited and the order in which values are assigned to variables, needs to be specified. Furthermore, these orders have an impact on the efficiency of the CSP search algorithm.

With regards to the order in which variables are visited (variable ordering), two types of ordering exist: static ordering (specified before-hand, fixed) and dynamic ordering (choosing the next variable to visit is not defined before-hand, and is subjected to the current state of the search space).

Variable ordering is, for example, applicable within the context of the aforementioned forward checking algorithm (which uses arc consistency).

"Constraint programming is the study of computational systems based on constraints. The idea of constraint programming is to solve problems by stating constraints (conditions, properties) which must be satisfied by the solution."

Roman Bartak, On-line Guide to Constraint Programming.

One heuristic (rule of thumb) that is used within variable ordering is the first-fail principle which states "*to succeed, try first where you are most likely to fail*". Using this variable ordering heuristic, the variable with the fewest possible remaining alternatives is selected as the next candidate for instantiation. The assumption is that any value is equally likely to participate in a solution, so the more values there are, the more likely it is that one of them will be a successful one. The reason however for choosing the variable with the fewest remaining alternatives (values) is that (within the context of forward checking) if the current (partial) solution does not lead to a complete solution, then the sooner we discover this the better. The first-fail principle tries to reduce the average depth of branches in the search tree by triggering early failure.

Similar strategies exist for value ordering, where the order in which values need to be assigned to the next variable in line is determined based on heuristics.

CSP Frameworks

Overview

Within the previous section, the basic strategies and algorithms for efficiently solving CSPs in an acceptable amount of time have been discussed. These algorithms, and many more, can be implemented based on the information presented within this paper and information publicly available on the Internet.

On the other hand, several CSP frameworks have been developed which provide its users with off-the-shelf facilities for efficiently modeling CSPs and solving them with state-of-the-art algorithms.

ILOG

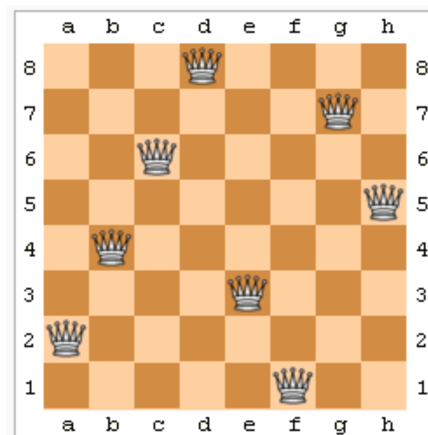
The most widely used commercial CSP framework is ILOG [4], which, for example, also contains specialised modules for scheduling problems. ILOG is available for both the Java and C++ programming language.

Choco CSP Framework

There are also several open-source Java CSP frameworks available, one of which will be discussed here – the Choco CSP framework [5].

The Choco CSP framework (publicly available at Sourceforge) is a framework developed in Java, which facilitates the creation of constraint satisfaction problems by providing an abstraction layer, and which enables their solution by providing off-the-shelf CSP algorithms. One can easily define variables, their domains and a large number of default constraints on those variables, thus modeling a CSP, and subsequently solve the created CSP. One can also define new classes of constraints. Choco is able to find one solution to a problem, all solutions and an optimized solution.

As an example, the n-queens problem is discussed here.



"Constraints arise naturally in most areas of human endeavor. The three angles of a triangle sum to 180 degrees, the sum of the currents floating into a node must equal zero, the position of the scroller in the window scrollbar must reflect the visible part of the underlying document. These are some examples of constraints which appear in the real world. Thus, constraints are a natural medium for people to express problems in many fields."

Roman Bartak, On-line Guide to Constraint Programming.

The n-queens puzzle [2] is a classic CSP. Its goal is to put n chess queens on an n×n chessboard such that none of them is able to capture any other using the standard chess queen's moves. Thus, a solution requires that no two queens share the same row, column, or diagonal.

The previous picture shows a visualisation of a (solved) n-queens problem where n=8 (the 8-queens problem). The 8-queens problem is computationally expensive, as there are 283.274.583.552 (64x63x..x58x57/8!) possible arrangements.

Within the Choco framework, this CSP is modeled and solved as described below [5].

First, an empty CSP is created as follows:

```
AbstractProblem CSP = new
Problem();
```

Then, the variables are created and initialised:

```
int n = 8;
IntVar[] queens = new IntVar[n];

for (int i = 0; i < n; i++)
{
   queens[i] =
   CSP.makeEnumIntVar("Q" + i,
   1, n);
}
```

Then, the constraints are added. The main type of constraint used here is the 'not equals' (neq) constraint:

```
for (int i = 0; i < n; i++)
{
   for (int j = i + 1; j < n;
   j++) {
      int k = j - i;

      CSP.post(CSP.neq(queens[i],
      queens[j]));

      CSP.post(CSP.neq(queens[i],
      CSP.plus(queens[j], k)));

      // diagonal constraints
      CSP.post(CSP.neq(queens[i],
      CSP.minus(queens[j], k)));
   }
}
```

Then, find all the solutions to the problem:

```
CSP.solveAll();
```

And finally, tell us (for example) how many solutions have been found:

```
System.out.println("NbSol: " +
CSP.getSolver().getNbSolutions());
```

The solver will tell us it has found 92 solutions.

This is a simple example of how a CSP can be efficiently modeled and solved in Java using the Choco CSP framework. With this framework, a software engineer does not have to be bothered with creating variables, domains and constraints, nor with creating the necessary algorithms (although all of these can be created and expanded within the framework). One can focus on the problem that needs to be solved using an elaborate abstraction layer and off-the-shelf CSP algorithms.

For More Information

For more information about **IPROFS**, call us on +31 – (0)23 – 5476369. To access information via the World Wide Web, go to:
www.iprofs.nl

For referenced material and more information about Constraint Satisfaction Problems and Constraint Based Reasoning, go to:

[1] www.aaai.org
[2] www.wikipedia.org
[3] kti.mff.cuni.cz/~bartak/constraints/
[4] www.ilog.com
[5] choco.sourceforge.net

© 2008 **IPROFS**. All rights reserved. This whitepaper is for informational purposes only. IPROFS makes no warranties, express or implied, in this paper. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Document published June 2008.

Constraint Relaxation

A more advanced topic within the CSP realm is Constraint Relaxation, which aims at solving CSPs which are over-constrained. This is an important theme, as most of the real-world CSPs are inherently over-constrained.

Constraint Relaxation approaches these initially impossible CSPs by making a differentiation between hard and soft constraints and by relaxing those soft constraints (within acceptable, to be defined boundaries). In many cases, these soft constraints are then transformed into cost functions, and the goal is to minimise these cost functions, thus minimising the amount of (generally expensive) relaxation that needs to take place, and thus transforming the original CSP into a Constraint Optimisation Problem.

An important, practical aspect within this context is to provide the end-user with feedback regarding which constraints have been relaxed, and to which extent.

Author

Nicolas van Hanxleden Houwert

is a Solutions Architect at IPROFS with an interest in expert and decision support systems, complex search problems and their solution using specialised models and algorithmic strategies. His experience gained within this field covers several domains, including those of automated aircraft scheduling and decision support (Dutch National Aerospace Laboratory, Schiphol Airport, Eurocontrol), optimised routing of telephone calls for the largest call-center in the Netherlands (Siemens Business Services, Belastingdienst), bioinformatics and DNA sequencing (University College Dublin, Ireland). In the past, he has evangelised these topics within his capacity as a Software Engineering lecturer.

Within the next whitepaper, we will elaborate on how to solve other classes of search problems using genetic algorithms.

For more information about IPROFS and what we can do for you, go to: www.iprofs.nl