Distributing Problem Solving on the Web using Constraint Technology.

Marc Torrens, Rainer Weigel and Boi Faltings

Artificial Intelligence Laboratory
Swiss Federal Institute of Technology in Lausanne (EPFL)
IN-Ecublens, CH-1015 Lausanne, Switzerland

Abstract
WWW servers have to serve many clients simultaneously and thus cannot provide intelligent services. We present an approach where intelligent problem solving is distributed so that compute-expensive tasks are carried out on the client side. To this end, we have implemented a library of constraint satisfaction techniques, called the JAVA constraint library, which allows composing applets that solve CSPs. We present the library and show several examples of applications.

Keywords: Intelligent Internet agents, Problem Solving on the Web, Product Configuration, Constraint-based Reasoning.

1 Distributing Problem Solving

As a result of the spread of the world-wide web, interactive information servers are becoming more and more commonplace. Browsing through databases requires quick response times which are difficult to achieve when users interact directly with a server. Bandwidth restrictions and server overload are the two major problems when users are browsing information directly on a
Web server. In addition, very often information on the web servers are not well-structured to fit certain complex queries.

Consider for example web pages of an airline company. They often provide access to their flight databases by requiring information about departure and destination airports. The result is a list of possible flight connections. A more difficult query where one plans a business roundtrip visiting several cities in an arbitrary order would be much more complicated and cumbersome using the standard interface. This is true in particular if one would like to minimize the hours of flight, costs, etc.

In order to answer such queries a system must provide problem solving capabilities. Our general approach is the following: The answer of the query is considered to be a solution to a constraint satisfaction problem (CSP). The formalisation as a CSP supports a natural decomposition of the task into two subtasks: generation of the CSP on the server, and solving the CSP on the client. The advantages of the approach are that

1. if there are many solutions to the problem, then the CSP can be regarded as a compact representation of the solution space. It represents it using a minimum amount of information that needs to be send to the client, and

2. finding the solutions requires only computing power of the client and thus avoids server overload. Furthermore, browsing through a large number of possible answers to a query is independent of the server and can be done locally.

The architecture supporting this methodology is shown in Fig. 1. The client sends a request containing the user constraints to the server. The server will access databases in order to generate the corresponding CSP taking into consideration the constraints of the user. The CSP is packaged with search algorithms for finding the solutions to form an agent which is returned to the user. In this way the user can browse through the different solutions by interacting with the agent locally.

We decompose the process into two parts:
the information server compiles all relevant information from the database and the user constraints (query) into the corresponding CSP. The CSP is a compact representation of all solutions that the problem can have given the initial restrictions of the user.

- the server sends an agent consisting of the CSP and search algorithms to the client. This allows the user to browse through all the possible solutions. Since the agent executes on the client, response time can be very fast and the user can compare different alternatives without placing unnecessary load on the server.

Building the CSP requires only a small fraction of time compared to solving the CSP, so having the agent executed on the client significantly reduces server overload. After having generated the CSP there is no longer need of accessing the server, so the agent sent by the server is completely autonomous. This architecture is shown in Fig. 1.

![Figure 1: A client-server architecture for distributing problem solving on the Web.](image)

In the next Section we introduce the Java Constraint Library (JCL) that supports our architecture described above.

## 2 Java Constraint Library

We implemented the Java Constraint Library (JCL) which allows us to package constraint satisfaction problems and their solvers in agents on the Internet. We will first give a brief introduction to constraint satisfaction techniques and then describe JCL. Finally we describe the JCL shell, a tool to solve CSPs on the Internet by means of a Java applet, and present a resource allocation problem to illustrate the usage of JCL.
2.1 Constraint Satisfaction Problems (CSPs)

CSPs are ubiquitous in applications like configuration [10, 9], planning [12, 7, 3], resource allocation [1, 11], scheduling [8, 2], timetabling [6] and many others. A CSP is specified by a set of variables and constraints among them. A solution to a CSP is a set of value assignments to all variables such that all constraints are satisfied. There can be either many, 1 or no solutions. The main advantages of constraint based programming are as following:

- It offers a general framework within which many real world problems can be stated in a succinct and elegant way.
- A constraint based representation can be used to synthesize solutions of the problem as well as for verification purposes (i.e. showing that a solution satisfies all constraints).
- The nature of the representation allows a formal description of the problems as well as a declarative description of search heuristics.

A finite, discrete Constraint Satisfaction Problem (CSP) is defined by a tuple \( P = (X, D, C, R) \) where \( X = \{X_1, \ldots, X_n\} \) is a finite set of variables, each associated with a domain of discrete values \( D = \{D_1, \ldots, D_n\} \), and a set of constraints \( C = \{C_1, \ldots, C_l\} \). Each constraint \( C_i \) is expressed by a relation \( R_i \) on some subset of variables. This subset of variables is called the connection of the constraint and denoted by \( \text{con}(C_i) \). The relation \( R_i \) over the connection of a constraint \( C_i \) is defined by \( R_i \subseteq D_{i1} \times \ldots \times D_{ik} \) and denotes the tuples that satisfy \( C_i \). The arity of a constraint \( C \) is the size of its connection. In particular, a constraint is called binary if it is between 2 variables (the size of its connection is equal to 2). If all constraints are binary, then the CSP is called binary. Otherwise the CSP is called a non-binary CSP.

A solution of a CSP \( P \) is a compound label of size \( n \) such that all constraints are satisfied simultaneously. A CSP is said to be satisfiable if a solution exists. Depending on the goals one wants to satisfy, one can either try to show that a CSP is satisfiable, i.e., show that a solution exists, find all solutions of a CSP, or find some optimal solutions, where optimality can be defined...
according to specified domain criteria. A large body of techniques exists for efficiently solving CSPs. For more details see [14].

2.2 The Constraint Library

The library provides services for:

- creating, managing and representing binary discrete CSPs
- applying search and preprocessing algorithms to CSPs

JCL can be used either in a Java enabled browser (applet) or in a stand-alone Java application. In Fig. 2 we present the main components of the JCL environment. The purpose of JCL is to provide a framework for easily building agents that solve CSPs on the Web. JCL is divided into two parts: A basic constraint library available on the network and a constraint shell build on the top of this library, allowing CSPs to be edited and solved. JCL allows the development of portable applications and applets using the constraint mechanisms. JCL can be downloaded from http://liawww.epfl.ch/~torrens. For more details see [13].

![Figure 2: The components of the JCL environment.](image)

The algorithms

The library contains search and preprocessing algorithms for CSPs. The search algorithms allow us to find the solutions of a CSP, while the preprocessing algorithms are used to simplify a
CSP by eliminating values and compound labels that do not affect its solutions. Several search algorithms are implemented in JCL. There are three main algorithms derived from *Chronological Backtracking* (BT) that are: *Backmarking* (BM), *Backjumping* (BJ) and *Forward Checking* (FC) [4]. Some combinations of them are implemented in [15] and adapted in JCL. Fig. 3 shows a hierarchy of the algorithms in JCL.

![Diagram](image)

**Figure 3:** *A hierarchy of the search algorithms implemented in JCL.*

The following two preprocessing algorithms are implemented in JCL: *Arc-consistency* (AC) and *Path-consistency* (PC) [5].

### 2.3 the JCL Shell

The purpose of the shell is to provide a user-friendly interface to the library. The JCL shell can be executed as a stand-alone application or as an applet. The following aspects are taken into consideration:

- CSP definition and generation from scratch,
- algorithm application and
- intermediate and final solution management.
Fig. 4 illustrates the relations between the library, the shell and the external world.

Fig. 5 shows how constraints in between variables can be edited using mouse and menus. We can select a pair of variables and then mark the allowed combination of values defining the constraint. Another important window is the solving control window shown in Fig. 6. It allows the user choose the algorithm, solution options, displaying options and start the algorithm. The HTML output produces the output in a browser window. The algorithm panel permits algorithm selection among the JCL algorithms or other algorithms that could be implemented by the user. While the algorithms are running, a “solving in progress” window is displayed indicating how many solutions have been found so far. In this window one can also suspend, resume or stop the resolution process.
We describe an applet for solving Resource Allocation problems. It is a simple application to demonstrate the flexibility and adaptability of the JCL for programming applications using CSP mechanisms on the Web. Resource Allocation (RA) problems can be defined as follows (see [1]): “Given a set of tasks with fixed endpoints, and given, for each task, a set of resources that can carry out the task, assign one resource to each of the tasks such that no resource is assigned to two different tasks at the same time”. The RA problem can be modeled as a discrete binary CSP where the variables are the tasks, the values are the resources and there are constraints of mutual exclusion between two variables if the corresponding tasks intersect in time. Consider the interval representation of the small example in Fig. 7, where we have 7 tasks $T_1, \ldots, T_7$ and resources $R_1, \ldots, R_5$. Two tasks intersecting in time are not allowed to use the same resource.

The applet of solving RA problems can be executed from the Web at: http://liawww.epfl.ch/~torrens/exercise/exercise.html. The tasks to be executed are represented as rectangles, and rectangles intersecting horizontally do intersect in time. Each
resource corresponds to a different color. When a resource is assigned to a task then the task is colored with the color of the resource. The applet allow the user to generate random Resource Allocation problems, select a search algorithm, and solve the CSP. A step-by-step mode allows a user to trace the execution of the algorithms. In Fig. 8 we show the appearance of this applet.

Figure 7: Interval orders: a schedule of seven tasks whose start time and duration are fixed. For each task, a set of possible resources is shown. A correct assignment of resources to the tasks is illustrated in red.

Figure 8: Resource Allocation applet searching a solution.
3 Applications

3.1 Air Travel Planning (ATP) system

From time to time, we are all faced with the problem of arranging business trips. Typically, we have to meet with a set of people in different cities, each of which has certain days where they are available for a meeting. Transportation schedules impose additional constraints. In the current state of affairs, schedule information can only be obtained by queries to travel agents or WWW servers for particular routes, dates and times. Thus, finding the optimal plan would require separate queries for every part of every alternative itinerary. Since each query implies response times on the order of 1 minute, this makes travel planning very tedious. A better solution is to use a client server architecture distributing problem solving on the Web.

The prototypical business Air Travel Planning (ATP) system is designed to facilitate arranging these kinds of business trips using JCL. An air travel plan is a sequence of flights connecting different cities a user plans to visit. Given such a set of cities together with possible time slots to visit each city, the system generates a set of plans. This plan consists of the meeting together with all possible flight connections from which the user can easily select the most preferred one.

The input data for ATP system is a set of meetings, where every meeting is described by the place and the possible time-slots for the different days the meeting could take place (see Table 1 and Fig. 9). We formulate the problem of finding a travel plan as a binary CSP. For each meeting $M_i$ there exists a constraint variable and the domain are the flights in between cities where the meetings could take place. The constraints require that flights are available such that the person can attend all the meetings. They can roughly be formulated as follows: a flight action from meeting $j$ to meeting $i$ can be accepted if meeting $j$ finishes before the departure time and the plane arrives before meeting $i$ starts. A solution of the CSP corresponding to the business travel problem can be seen as a sequence of flights in between the cities of the meetings. For each meeting, one of the possible days must be assigned and it must be guaranteed that then exists at least one flight connection between consecutive meetings. Consider as example the travel data
<table>
<thead>
<tr>
<th>M</th>
<th>City</th>
<th>Time-Slots for November</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>AMS</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; 12h-16h</td>
</tr>
<tr>
<td>M2</td>
<td>BCN</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; 12h-15h</td>
</tr>
<tr>
<td>M3</td>
<td>LON</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; 12h-15h</td>
</tr>
<tr>
<td>M4</td>
<td>GVA</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; 9h-12h</td>
</tr>
<tr>
<td>M5</td>
<td>PAR</td>
<td>5&lt;sup&gt;th&lt;/sup&gt; 8h-12h</td>
</tr>
<tr>
<td>M6</td>
<td>BER</td>
<td>6&lt;sup&gt;th&lt;/sup&gt; 15h-18h</td>
</tr>
<tr>
<td>M7</td>
<td>FRA</td>
<td>4&lt;sup&gt;th&lt;/sup&gt; 8h-12h</td>
</tr>
</tbody>
</table>

Table 1: *Input Data to be send to the server.*

from Table 1.

In the following we present a new methodology for solving the business travel problem on the Web using JCL. The main idea is to generate the CSP corresponding the business travel problem on the server and solve the problem locally on the client (see Fig. 1). The input required from the traveler includes all possible meeting slots. The input is sent to the server in order to build the corresponding CSP, taking flight databases and the user input into account. Then the CSP is packaged together with search algorithms from JCL into an autonomous agent. The CSP can then be solved on the client without having to access the server. This allows the user to browse autonomously through the different solutions. When the user selects a travel plan (a solution) then the actual flights can be presented in the form of a list such that the user can easily select the most preferred one. The example below describes some possible partial solution of the problem shown in Table 1:

- meeting *M1* (Amsterdam) is scheduled on the 3<sup>rd</sup> (from 13h to 15h)
- meeting *M2* (Barcelona) is scheduled on the 2<sup>nd</sup> (from 13h to 17h)
- meeting *M4* (Geneva) is scheduled on the 4<sup>th</sup> (from 9h to 12h)
- meeting *M5* (Paris) is scheduled on the 5<sup>th</sup> (from 8h to 12h)

The corresponding flights to this partial solution are shown in Table 2. In order to access flight data for building the CSP, we created a MiniSQL<sup>1</sup> database. We use a Java class library called

---

<sup>1</sup>Reference at http://Hugos.com.au
MsqlJava\(^2\) which allows applications or applets to access and manipulate MiniSQL databases. On the server the MiniSQL is running in background. This makes it possible for the applet on the client to access the flight database to build the corresponding CSP. Table 3 describes some rows of the created database.

\(^2\)Reference at http://mama.minet.uq.oz.au/msqljava
<table>
<thead>
<tr>
<th>Comp</th>
<th>Fly</th>
<th>From</th>
<th>To</th>
<th>Dep</th>
<th>Arr</th>
<th>Dur</th>
</tr>
</thead>
<tbody>
<tr>
<td>IB</td>
<td>4248</td>
<td>BCN</td>
<td>AMS</td>
<td>10:15</td>
<td>12:25</td>
<td>2:10</td>
</tr>
<tr>
<td>KL</td>
<td>352</td>
<td>BCN</td>
<td>AMS</td>
<td>7:05</td>
<td>9:25</td>
<td>2:20</td>
</tr>
<tr>
<td>SR</td>
<td>724</td>
<td>GVA</td>
<td>PAR</td>
<td>12:15</td>
<td>13:20</td>
<td>1:05</td>
</tr>
<tr>
<td>AF</td>
<td>2855</td>
<td>GVA</td>
<td>PAR</td>
<td>14:10</td>
<td>15:15</td>
<td>1:05</td>
</tr>
<tr>
<td>SR</td>
<td>726</td>
<td>GVA</td>
<td>PAR</td>
<td>16:15</td>
<td>17:20</td>
<td>1:05</td>
</tr>
<tr>
<td>AF</td>
<td>2835</td>
<td>GVA</td>
<td>PAR</td>
<td>17:15</td>
<td>18:20</td>
<td>1:05</td>
</tr>
<tr>
<td>AF</td>
<td>2893</td>
<td>GVA</td>
<td>PAR</td>
<td>18:05</td>
<td>19:10</td>
<td>1:05</td>
</tr>
<tr>
<td>SR</td>
<td>728</td>
<td>GVA</td>
<td>PAR</td>
<td>18:45</td>
<td>19:50</td>
<td>1:05</td>
</tr>
<tr>
<td>AF</td>
<td>2887</td>
<td>GVA</td>
<td>PAR</td>
<td>20:40</td>
<td>21:45</td>
<td>1:05</td>
</tr>
</tbody>
</table>

Table 2: Possible flights from Barcelona on 2nd (after 17h) to Amsterdam on 3rd (before 13h) and from Geneva on 4th (after 12h) to Paris on 5th (before 8h).

<table>
<thead>
<tr>
<th>Code</th>
<th>Flight</th>
<th>Dep.</th>
<th>Time</th>
<th>Ar.</th>
<th>Time</th>
<th>Days</th>
</tr>
</thead>
<tbody>
<tr>
<td>IB</td>
<td>4248</td>
<td>BCN</td>
<td>10:15</td>
<td>AMS</td>
<td>12:25</td>
<td>1234567</td>
</tr>
<tr>
<td>KL</td>
<td>354</td>
<td>BCN</td>
<td>11:25</td>
<td>AMS</td>
<td>13:45</td>
<td>1234567</td>
</tr>
<tr>
<td>KL</td>
<td>356</td>
<td>BCN</td>
<td>16:10</td>
<td>AMS</td>
<td>18:30</td>
<td>1234567</td>
</tr>
<tr>
<td>IB</td>
<td>4262</td>
<td>BCN</td>
<td>16:40</td>
<td>AMS</td>
<td>18:55</td>
<td>1234567</td>
</tr>
</tbody>
</table>

Table 3: Description of some rows of the flight database.

### 3.2 Product Configuration for Electronic Commerce

In recent years, manufacturing trends have changed from pure mass-production towards a more customer oriented one-of-a-kind production. The main reason for this change is that today’s customers have very specific and individual requirements which cannot be satisfied by mass-products. The one-of-a-kind production of many consumer and investment products imposes new challenges concerning the marketing of these products, in particular in Electronic Commerce.

With current electronic catalogs, customers are supposed to compose solutions themselves by selecting elements individually. In the future, product configuration utilities that synthesize products according to customer’s wishes will be indispensable for selling multi-variant products through Electronic Commerce, and will be an essential part of more intelligent electronic catalogs.

The general configuration task can be defined as follows: Given

- a set of predefined components,
- the knowledge of how components can be connected,

- the customer requirements for a specific configuration

find the sets of components fulfilling the user-requirements and respecting all the compatibility constraints. The configuration task can be formalized as a general constraint satisfaction problem (CSP). Two approaches for solving configuration problems represented as CSPs exist:

**Standard approach:** Using this technique one needs to solve the CSP corresponding to the configuration problem from scratch. That is the customer inputs the requirements and then the constraint solver tries to find the solutions. There are three major problems with this approach. The first is that user requirements may lead to an over-constrained CSP and thus no product can be configured. The second is that there are too many possible products satisfying the requirements and the customer is overloaded with information. The third and most serious problem is that often a user has only a vague idea about the product and cannot really express the input requirements.

**Case-based approach:** This approach allows the customer to select a configuration from a set of configurations sold to earlier (possibly fictitious) customers. Then the selected configuration can interactively be modified until a product satisfying all the requirements is found. The case-based method is especially well-suited when there are only a few “standard” products which could be represented in a catalog and when constraint based adaptation methods to modify these standard products are available. This approach avoids all the problems mentioned above.

In both cases, the JCL allows executing the computationally expensive parts of the problem on the customer side. Fig. 10 shows the resulting architecture for the case-based approach.
4 Conclusions

As a result of the spread of the world-wide web, interactive information servers are becoming more and more important. Browsing through databases requires quick response times which are difficult to achieve when users interact directly with a server. We have shown a methodology when agent techniques underlying Java can separate browsing from database access. The key element of this approach is to represent solution spaces of a problem as a CSP. This CSP will be bundled with search engines from JCL that allow the problem to be solved locally on the client without requiring computation resources from the server.

References


