Java Constraint Library: bringing constraints technology on the Internet using the Java language

Marc Torrens, Rainer Weigel and Boi Faltings

Laboratoire d'Intelligence Artificielle
Ecole Polytechnique Fédérale de Lausanne (EPFL)
IN-Ecublens, CH-1015 Lausanne
Switzerland
{torrens,weigel,faltings}@lia.di.epfl.ch

Abstract

Distributed problem solving on the web is becoming more and more important. Client server architectures are often confronted with server overload. The process of browsing a large number of alternative solutions is particularly tedious. In this paper, we present a methodology for distributing the computation between server and client. The idea is to formalise the problem as a constraint satisfaction problem (CSP). This formalisation supports a natural decomposition of the task into two subtasks: generation of the CSP by the server from its database, and generating and browsing the solutions on the client. In this way, the browsing process runs locally and can be very fast, while the server is only accessed once during the process. We provide the Java Constraint library (JCL) for implementing the agent that solves the CSP on the client. We illustrate the concept on the example of planning air travel.

Introduction

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: I can combine a meeting in Geneva in the morning with another in Basel at lunch, but if the lunch meeting is in Lucerne this is not possible as the flight/train connection takes too long time. In the current state of affairs, reliable schedule information can only be obtained by queries to travel agents or WWW servers for particular routes, dates and times. Thus, really finding the optimal plan would require database queries for every of every alternative itinerary. Since each query implies response times on the order of 1 minute, this makes travel planning very tedious. Recently, one researcher in our laboratory spent an entire afternoon collecting flight schedules on the WWW just for one rather simple trip! A better solution consists of performing just one single database access where all relevant information is collected, and then searching for a particular best solution locally. Thus, we decompose the process into two parts:

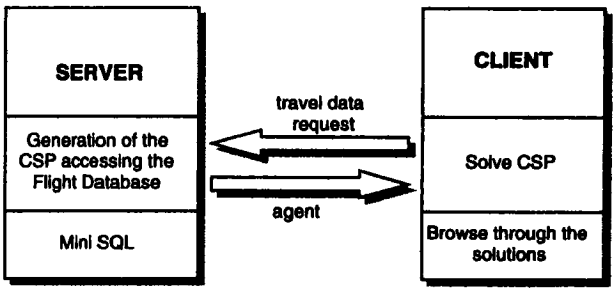

Figure 1: *A client server architecture of ATP.*

- the information server compiles all relevant information from the database in order that the client build the constraint satisfaction problem. The CSP is a compact representation of all solutions that the problem can have given the initial restrictions of places and dates.
- the server sends the user an agent consisting of the CSP and solution algorithms. It allows the user to browse through the different solution possibilities. 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.

We have implemented a Java constraint library which allows us to package constraint satisfaction problems and their solvers in agents. We will first give a brief introduction to constraint satisfaction techniques and the way in which they can represent solution spaces. We then describe the Java library and its application on the problem of air travel planning. Figure 1 describes the basic architecture of Air Travel Planning System (ATP). Using ATP a user can generate flight plans according to his wishes. In order to generate the CSP that represents the flight plans, we have to access our world wide through flights database that contains all scheduled airline passenger services November 1995. After having generated the CSP there is no longer a need of accessing the server and thus the user can interact locally on the Java client.

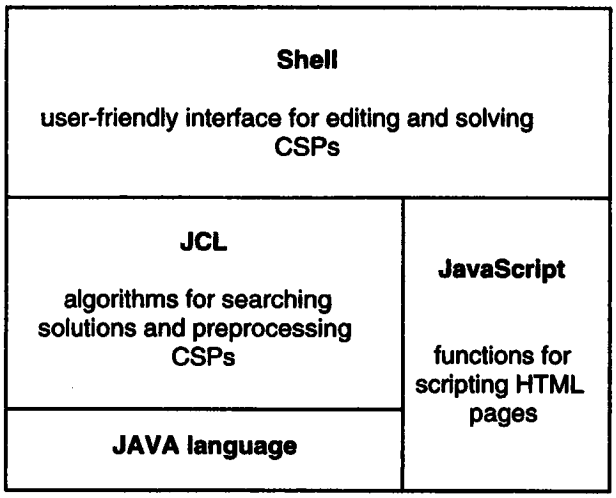

Figure 2: *The components of the JCL environment.*

Constraint Satisfaction Problems and JCL

Many important applications, such as configuration, resource allocation and diagnosis can be modeled as discrete Constraint Satisfaction Problems (CSPs). A CSP is defined by $P = (X, D, C, R)$, where $X = \{X_1, \ldots, X_n\}$ is a set of variables, $D = \{D_1, \ldots, D_n\}$ a set of finite domains associated with the variables, $C = \{C_1, \ldots, C_m\}$ a set of constraints, and $R = \{R_{ij} \subset D_i \times D_j$ for a constraint applicable to X_i and $X_j\}$ a set of relations that define the constraints. Solving a discrete CSP amounts to finding value assignments to variables subject to constraints. The theoretical complexity for solving CSPs was shown to be exponential, however for many real world applications the corresponding CSP can be transformed in a reasonable amount of time into a CSP that can be solved in linear time. A large body of techniques exists for efficiently solving CSPs.

JCL is a Java library that can be used in a Java enabled browser (applet) and in stand-alone Java applications. Figure 2 shows the components of the JCL environment. Its purpose is to provide the building blocks for agents that solve binary Constraint Satisfaction Problems (CSPs). JCL is divided into two parts: A basic constraint library available on the network and a constraint shell build on top of this library, allowing CSPs to be opened, saved, edited and solved. JCL allows the development of portable applications and applets using the constraint mechanisms.

Algorithms in JCL

The library contains search and preprocessing algorithms.

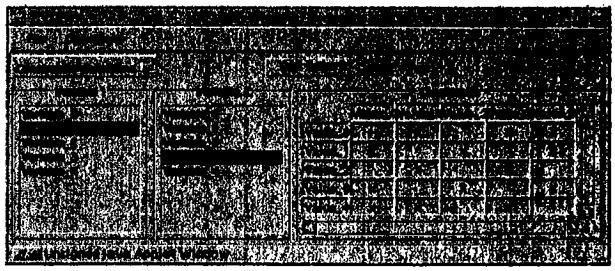

Figure 3: *The constraints editor in the Constraint Shell.*

Search Algorithms

In JCL we have implemented 13 algorithms adapted from (van Beek). There are three main algorithms derived from Chronological Backtracking (BT) that are: Backmarking (BM), Backjumping (BJ) and Forward Checking (FC) (Tsang 1993). Some combinations of them are implemented in (van Beek) and adapted in JCL. FC, for example, performs the consistency checks forward. At each level, the domains of the future variables are filtered in such a way that all values inconsistent with the current instantiation are removed. FC is very efficient because of its ability to discover inconsistencies early. The size of the backtrack tree is reduced. However, FC sometimes performs more consistency checks than backward algorithms. In (Kondrak 1994) there are the hierarchies of some algorithms with respect to the number of visited nodes in the search tree and with respect to the consistency checks.

Preprocessing Algorithms

The objective of the preprocessing is to reduce the size of the CSP by removing redundant values from the domains of the variables, or by removing redundant compound labels from the constraints. A value of a variable is redundant when its removal does not affect the solutions of the CSP. If all the values from the domain of a variable are removed, then the CSP is insoluble. In JCL we have implemented two preprocessing algorithms adapted from (van Beek) : Arc-consistency and Path-consistency.

The Constraint Shell

The purpose of the shell is to provide a user-friendly interface to the library. The following aspects are taken into consideration:

- CSP problems definition and generation,
- algorithms application,
- intermediate and final solution management.

Figure 3 shows how constraints in between variables can be edited using mouse and menus.

Another important window is the "solving control" window shown in figure 4. It lets the user choose the algorithm, solution options, displaying options, and

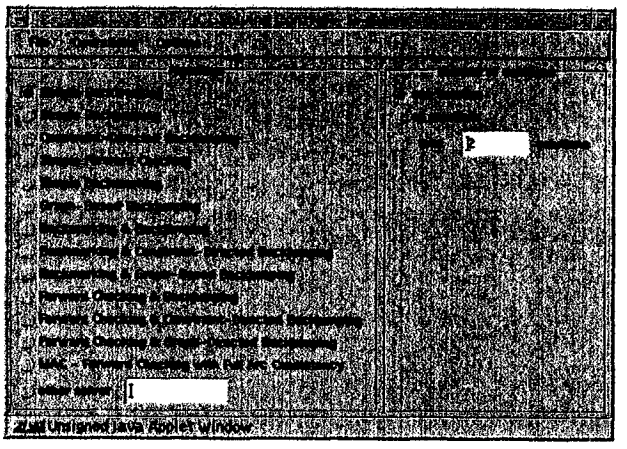

Figure 4: *The solving control window.*

start the algorithm. The solutions and the displaying options panel selects the different options for solution output. The HTML output produces output in a browser window. The "Algorithm" panel permits algorithm selection between the JCL algorithms or other algorithms implemented by the user. While the algorithm is working, a "Solving in progress" window is displayed by the default solution manager, indicating among other things how many solutions have been found until now, and allowing to suspend, resume or stop the resolution. JCL is available from the Internet at: http://liawww.epfl.ch/torrens

Prototypical business Air Travel Planning system

The prototypical business Air Travel Planning (ATP) system is designed to illustrate the use of JCL for planning. A air travel plan is a sequence of flights connecting different cities a user wants to visit. Given such a set of cities together with possible time slots to visit each city, the system generates a representation of plans from which the user can easily select the most preferred one. Before describing the details and an example we will first present the architecture of ATP.

A client server architecture for ATP

The basic idea is that a client sends a request containing the users raw travel data to a server. The server will have access the flight database in order to generate a Constraint Problem whose solutions are the possible travel plans that will satisfy to the users requests. This CSP together is packaged with search algorithms from the JCL to form an agent which can interact with the user. Building the CSP requires only a small fraction of time compared to solving the CSP, so having the agent execute on the client significantly reduces server load.

Code	Flight	Dep.	Time	Ar.	Time	Days
IB	4248	BCN	1015	AMS	1225	1234567
KL	354	BCN	1125	AMS	1345	1234567
KL	356	BCN	1610	AMS	1830	1234567
IB	4262	BCN	1640	AMS	1855	1234567

Table 1: *Description of some rows of the flight database.*

Accessing to the server database

In order to access to the flight data needed to build the CSP, we have created a database using MiniSQL[1]. We use a Java class library called MsqlJava[2] which allows applications or applets to access and manipulate MiniSQL databases. In the server the MiniSQL server is running in background, and thus is possible that the applet client can access to the flight database in order to build the CSP. Table 1 describes some rows of the created database.

Problem Formulation

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 can take place. A solution of 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. We have formulated the problem of finding a travel plan as a binary constraint satisfaction problem (CSP). The variables of the travel plan CSP is the union of meeting variables MV and flight variables FV. The meeting variables together with the constraints between them induce a constraint problem that we call meeting CSP (MCSP). Similarly the CSP induced by the flight variables FV is called flight CSP (FCSP). Next we describe the MCSP, the FCSP and the constraints between them in more detail.

Meeting CSP (MCSP) For every meeting M_i a variable $MV_i (i = 1, \ldots, n)$ is created and the domain of the variable is the set of possible days where the meeting M_i can take place. There is an inequality constraint between two variables if their domains intersect in at least one value. A solution of the MCSP is a assignment of day to each meeting such that no two meetings can be held on the same day. Having a solution does of course not guarantee that the solution can be refined down to the actual flights. Consider as example the travel data from table 2. Solving MCSP is computational equivalent with solving the list coloring problem which is known to be NP-complete.

[1]Reference at http://Huges.com.au

[2]Reference at http://mama.minet.uq.os.au/msqljava

M	City	Time-Slots for November		
M1	AMS	1^{st} 12h-16h	3^{th} 13h-15h	
M2	BCN	1^{st} 12h-15h	2^{nd} 13h-17h	
M3	LON	2^{nd} 12h-15h	8^{th} 11h-14h	
M4	GVA	2^{nd} 9h-12h	4^{th} 9h-12h	5^{th} 10h-15h
M5	PAR	5^{th} 8h-12h	8^{th} 8h-12h	
M6	BER	6^{th} 15h-18h	8^{th} 10h-16h	
M7	FRA	4^{th} 8h-12h	7^{th} 8h-12h	

Table 2: *Input Data to be send to the server.*

Meeting	First Solution	Second Solution
M1	1^{st} 12h-16h	3^{th} 13h-15h
M2	1^{st} 12h-15h	2^{nd} 13h-17h
M3	2^{nd} 12h-15h	8^{th} 11h-14h
M4	2^{nd} 9h-12h	4^{th} 9h-12h
M5	5^{th} 8h-12h	8^{th} 8h-12h
M6	6^{th} 15h-18h	8^{th} 10h-16h
M7	4^{th} 8h-12h	7^{th} 8h-12h

Table 3: *Two of the possible solutions to the MCSP.*

Flight CSP (FCSP) The FCSP can be considered as the planning or sequencing part of the travel planning CSP. For each planning step exists a variable and the values of these planning steps are the flights in between cities where the meetings could take place. In order to apply a flight action from meeting j to meeting i is the traveler must simply have finished meeting j and he should arrive before the meeting i starts. Whenever a meeting is scheduled for day k we create a "Before Meeting" variable BM_k and a "After Meeting" variable AM_k. Using such a model lets us easily express the constraint that a traveler would like to stay in town X after the meeting and takes the flight on the next day. The values for a BM_i variable are either the flight from a meeting he had before day i to the possible meeting places at day k ($k \geq i$) arriving before the meeting starts or a "no-action" value if he is already in a city where the meeting takes place.

Constraints Having introduced the variables and the values for the MCSP and FCSP we will now describe the constraints that have to be satisfied. The following constraints need to be expressed:

- $[AM_k - BM_{k+1}]$: if we stay at the meeting place on day k, which is the "no-action" value AM_k, then we have to fly to the place of meeting $k+1$ arriving before the meeting starts. On the other hand, the flight to the meeting $k+1$ leaving the meeting city k after the meeting, is compatible with the "no-action" value for B_{k+1}.
- $[BM_k - AM_k]$: four combinations are possible : 1) The "flight-flight" tuple, if we arrive in the city directly before the meeting and leave the city on the same day. 2) The "no-action-flight" tuple if we are already in the city and leave directly after the meeting. 3) Similarly the "flight-no-action" and finally 4) the "no-action - no-action" tuple.

To describe how decisions made in the MCSP can be propagated to the FCSP, we have to describe the constraints in between MCSP and FCSP: If a variable M_i has the value k in its domain i.e. meeting M_i can be scheduled on the kth day, then there is a constraint in between the BM_k variable and the AM_k variable in the FCSP. The allowed value combinations are the following:

- $[M_i - BM_k]$: k is compatible with a "flight to city of meeting k" or with the "no-action" if we are already in the city of meeting k. All the other values of M_i are not related with the meetings on day k and are therefor valid.
- $[M_i - AM_k]$: similar to the other constraints above. k is compatible with a "flight from city of meeting k to another city" or with the no-action if we are planning to stay overnight in the city of meeting k. All the other values of M_i are not related with the meetings on day k and are therefor valid.

Example

We use the example already presented in table 2. The user will in a first step decide on the days the meeting can take place. That is he decides on the days the meetings will take place without having to consider more details. For doing so the system needs to solve the MCSP (see table 3). A solution to the MCSP is a partial solution of the overall CSP which can then be propagated to the FCSP. This allows the removal of values from the domains of the variables in FCSP that are not consistent with selected days by the users. One can show theoretically that making the CSP arc-consistent after having decided on the meeting days guarantees global consistency of the CSP. This implies that any partial solution of the FCSP can be extended to a global solution of the whole CSP without backtracking. The table 4 describes the variables and their domains of FCSP after having propagated the second partial solution from table 3. In table 4 one can observe for example that there are no flights leaving Barcelona with destination Amsterdam after 17h. This implies that the user has to fly to Amsterdam the next day. Furthermore no hotel is needed in Amsterdam since he is required to take the flight to Geneva directly after the meeting. The user can select a single flight from all flights leaving Barcelona and arriving in Amsterdam before 13h depicted in table 5. The actual flights can then be presented to the user in the form of a list such that the user can easily select the most preferred one.

Meeting	City	Day	TimeSlots	Variable BM	Variable AM
1. M2	BCN	2^{nd}	13h-17h	FL : M0 - M2	M2
2. M1	AMS	3^{th}	13h-15h	FL : M2 - M1	FL : M1 - M4
3. M4	GVA	4^{th}	9h-12h	M4	FL : M4 - M5
4. M5	PAR	5^{th}	8h-12h	M5	M5, FL : M5 - M6
5. M6	BER	6^{th}	15h-18h	M6, FL : M5 - M6	FL : M6 - M7
6. M7	FRA	7^{th}	8h-12h	M7	M7, FL : M7 - M3
7. M3	LON	8^{th}	11h-14h	M3, FL : M7 - M3	M3

Table 4: *The solution space after having propagated a partial solution.*

Comp	Fly	From	To	Dep	Arr	Dur
IB	4248	BCN	AMS	10:15	12:25	2:10
KL	352	BCN	AMS	7:05	9:25	2:20
SR	724	GVA	PAR	12:15	13:20	1:05
AF	2855	GVA	PAR	14:10	15:15	1:05
SR	726	GVA	PAR	16:15	17:20	1:05
AF	2835	GVA	PAR	17:15	18:20	1:05
AF	2893	GVA	PAR	18:05	19:10	1:05
SR	728	GVA	PAR	18:45	19:50	1:05
AF	2887	GVA	PAR	20:40	21:45	1:05

Table 5: *Possible flights arriving in Amsterdam from Barcelona before 13h. and the flights from Geneva to Paris on the 4^{th} day.*

Conclusion

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 an example of how the agent techniques underlying Java can separate browsing from database access. The key element of this approach is to represent solution spaces through constraint satisfaction problems. The same approach is applicable to other problems where a combination of elements needs to be configured into a coherent whole. Such configuration systems will be an important technology for many areas of electronic commerce.

References

Kondrak, G. 1994. A Theoretical Evaluation of Selected Backtracking Algorithms. Technical Report TR-94-10, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada.

Tsang, E. 1993. *Foundations of Constraint Satisfaction.* London, UK: Academic Press.

van Beek, P. *CSPLib : a CSP library written in C language.* vanbeek@cs.ualberta: University of Alberta.