Published in the Proceedings of PA Java99
The Practical Application of Java, London, April 1999

Waltz Filtering in Java with JSolver

Andy Hon Wai Chun[1]

City University of Hong Kong
Department of Electronic Engineering
Tat Chee Avenue
Kowloon, Hong Kong
Tel: (852)-2788-7194 Fax: (852)-2784-4242
eehwchun@cityu.edu.hk

Abstract

This paper uses the Waltz Filtering Algorithm and the Line-Labelling Problem to illustrate how AI algorithms can easily be implemented using Java and JSolver[2] – a constraint-programming class library. The Waltz Filtering Algorithm popularised the technique of constraint propagation – the cornerstone for algorithms to solve constraint-satisfaction problems (CSP). It was initially developed as a computer-vision algorithm to interpret line drawings of three-dimensional scenes. This algorithm was originally implemented in Micro-Planner and Lisp. Although most AI algorithms were invented using either Lisp or Prolog, many commercial applications require the use of modern computer languages such as Java. In this paper, we will illustrate how a line-labelling program can developed using constraint-programming techniques provided by JSolver. JSolver implements advanced AI techniques such as constraint propagation, declarative programming, and non-deterministic search in Java. These techniques have been successfully used for scheduling and resource allocation systems.

1. INTRODUCTION

Waltz [17, 18] initially proposed the filtering algorithm as a way to reduce combinatorics associated with line labelling of three-dimensional scenes. This algorithm popularised the technique of constraint propagation and is known as the Waltz Filtering Algorithm. It substantially reduces search by pruning the search space early on using domain-specific constraints. For the problem of line labelling, the search space contains all the possible combinations of potential line labels for a scene. For complicated scenes, this search space can be quite enormous. Waltz discovered that filtering can prune a search space and very often eliminate the need for search altogether. Since then, numerous researchers have investigated and formalised the

[1] Dr. Andy Chun is also founder and Managing Director of Advanced Object Technologies Ltd., a high-tech company specialising in resource optimisation systems.
[2] JSolver is available for download from http://www.aotl.com

technique of constraint propagation [4, 10].

Constraint propagation combined with non-deterministic search became a success paradigm to solve constraint-satisfaction problems (CSP) [2, 8, 14, 15, 16]. In recent years, CSP algorithms have be used successfully to solve many different types of real life problems [9, 12], such as resource allocation, job-shop scheduling, timetabling, and duty rostering.

The Waltz Filtering Algorithm was originally implemented in Micro-Planner and Lisp. However, software technology has advanced quite a lot since then. This paper shows how the line-labelling problem can be solved using JSolver that combines object-orientation provided by Java with advanced AI techniques such as constraint propagation, declarative programming, and non-deterministic search. Tools like JSolver permit advanced AI algorithms to be easily integrated within Java applications.

2. THE WALTZ FILTERING ALGORITHM

The Waltz Filtering Algorithm analyses line drawings by assigning a label to each of the edges. The line label is selected from a finite set of possible labels. Each label provides specific semantic information on the nature of the edge and the regions on each of its sides. This approach to analysing line drawings [3, 1, 5] was extended by Waltz to include shadows and cracks. Waltz also enhanced the line-labelling search with his filtering algorithm.

For illustration purposes, this paper will only use a small subset of the original labels used by Waltz. Using our label subset, the following shows a cube with its four possible sets of line labels:

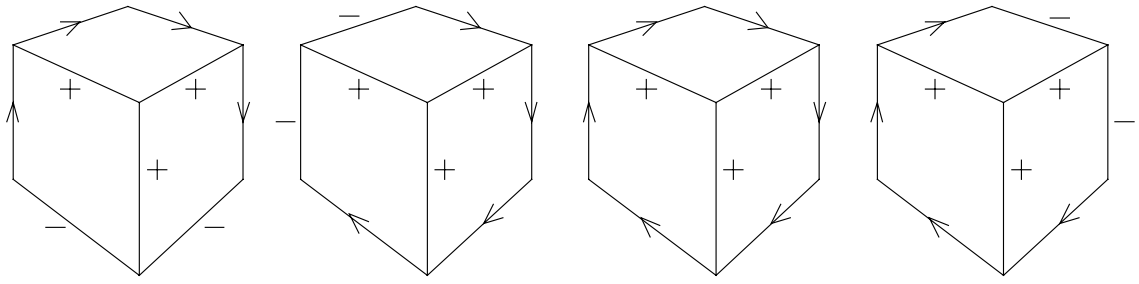


Fig. 1 A cube with its four possible sets of line labels.

In this label subset, the "+" label defines a convex edge, a "-" defines a concave edge, and the arrow defines the outer edge of an object with the object on the right-hand-side along the direction of the arrow. For a simple polyhedron, we can classify all junctions into three basic categories – an L-shaped junction, an arrow-shaped junction, and a fork junction. For each type of junction, we can enumerate all the physically realisable interpretations of that junction. Figure 2 shows all the possible labels for each of the three junction types.

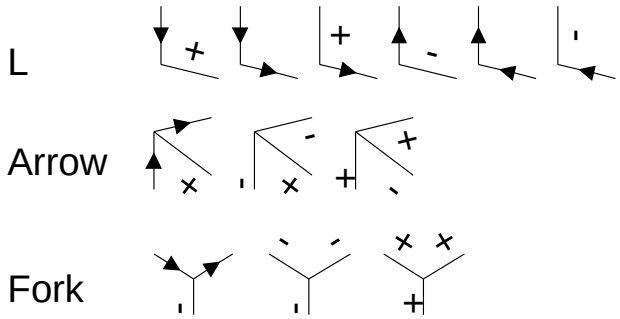


Fig. 2 Finite set of potential labels for each type of junction used in our examples.

3. CONSTRAINT-SATISFACTION PROBLEM

Many optimisation problems can be formulated as a constraint-satisfaction problem (CSP) [8] that involves the assignment of values to variables subjected to a set of constraints. CSP can be defined as consisting of a finite set of n variables $v_1, v_2, ..., v_n$, a set of domains $d_1, d_2, ..., d_n$, and a set of constraint relations $c_1, c_2, ..., c_m$. Each d_i defines a finite set of values (or solutions) that variable v_i may be assigned. A constraint c_j specifies the consistent or inconsistent choices among variables and is defined as a subset of the Cartesian product: $c_j \subseteq d_1 \times d_2 \times ... \times d_n$. The goal of a CSP algorithm is to find one tuple from $d_1 \times d_2 \times ... \times d_n$ such that n assignments of values to variables satisfy all constraints simultaneously.

When the line-labelling problem is formulated as a CSP, one approach is to represent each unknown edge label as a constrained variable. The domain of this *edge variable* is the set of all possible labels. In our example, this set shall consist of the labels "+," "–," "→," and "←." The constraints of this problem are that the edges must form junctions that are physically realisable, and that each edge can have only one single label, i.e., the line labels at both ends of an edge must agree.

Although constraint programming has a relatively long history [15], with constraint language extensions found in Prolog [2, 16], Lisp [14], and C++ [6, 9, 11], it is only recently that constraint-programming techniques can be found in Java [7]. The example line-labelling problem in this paper is implemented using the JSolver constraint-programming class library.

4. THE JSOLVER LINE LABELLING PROGRAM

This Section describes the design and implementation of a Java line-labelling program written using the JSolver class library. The line-labelling problem will be represented as a CSP. The example Java program will produce all the possible sets of labelling for the polyhedron object shown in Figure 3. This object consists of 11 junctions (5 'Arrow' junctions, 3 'L' junctions, and 3 'Fork' junctions) and 15 edges that form these junctions.

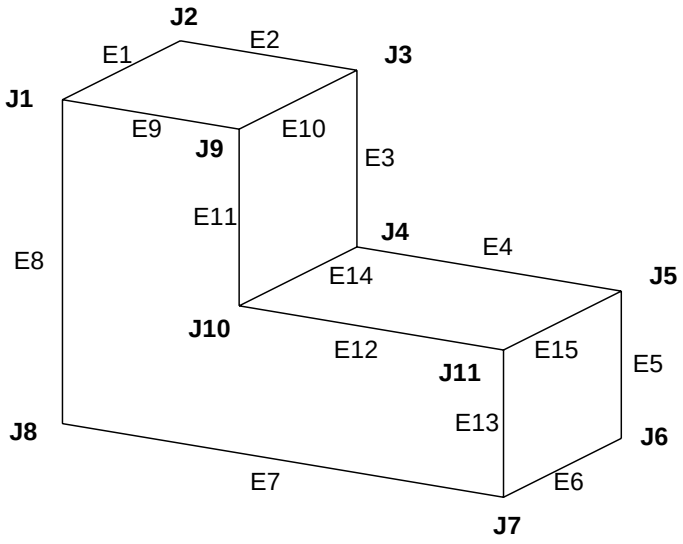


Fig. 3 The polyhedron used in our Java line labelling program.

The example Java implementation follows the following four simple steps:

- Define classes to represent domain objects
- Define the line-labelling constraints
- Create the polyhedron
- Search for solution labelling

4.1 Defining the Domain Classes

Figure 4 is the UML [13] class diagram of all the Java classes in our line-labelling program.

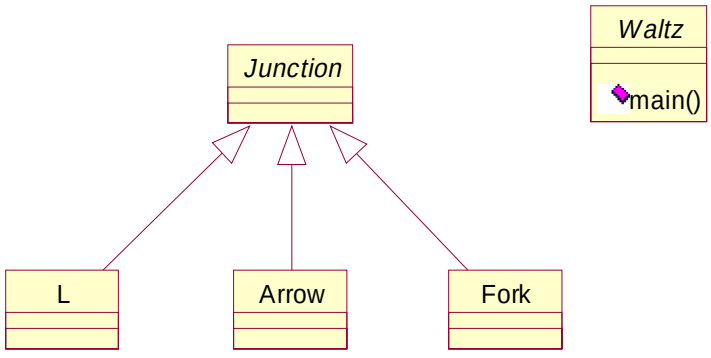
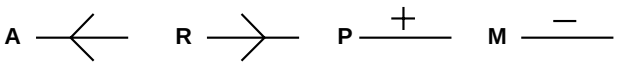


Fig. 4 The UML class diagram of the JSolver line-labelling program.

The junctions used in our example are represented by the classes *L*, *Arrow*, and *Fork*, which are subclasses of the *Junction* abstract class. The *Waltz* utility class contains our main program. The following is the Java implementation of the *Junction* class:

```
public abstract class Junction {
  protected final static int A = 0, R = 1, M = 2, P = 3;
}
```

The code for the *Junction* parent class is surprisingly simple. We have defined the *Junction* class to be an abstract class since it will not have any direct instances. The four class variables – A, R, M, and P – represent the four possible labels and provide the convenience of Lisp-like symbol processing.




```
public final class L extends Junction {
  public L(Var x, Var y) throws FailException {
    JSolver.post((x.eq(R).and(y.eq(P))).or
                 (x.eq(R).and(y.eq(R))).or
                 (x.eq(P).and(y.eq(R))).or
                 (x.eq(A).and(y.eq(M))).or
                 (x.eq(A).and(y.eq(A))).or
                 (x.eq(M).and(y.eq(A))));
  }
}
```

The *L* class represents an L-shaped junction and is implemented as a subclass of *Junction*, as shown above. The arguments to the constructor are the two edges that form the L-shaped junction. Notice that the constructor not only creates an instance of an *L* junction but also defines line-labelling constraints related to the L-shaped junction.

4.2 Defining the Constraints

Constraints are posted using the *JSolver.post()* function, which provides *declarative-style programming* where one simply state the constraints of the problem and let the built-in JSolver non-deterministic algorithm search for solutions. For our example, the constraint declares that an L-shaped junction can at most have 6 physically realisable edge combinations. Posting a constraint may potentially throw a *FailException* if the domain of any variable reduces to null.

Posted constraints will trigger Waltz-like constraint propagation and automatic domain reduction during the non-deterministic search. The *eq()*, *and()*, and *or()* functions represent JSolver built-in equality and logical constraints respectively.

Since the labels of the edges are unknown, we represent them as CSP variables. These constrained integer variables are implemented using the *Var* class provided by JSolver. Each *Var* instance represents an edge in the line drawing.

Similar to the L junction, the *Arrow* and *Fork* classes can be defined as follow:

```
public final class Arrow extends Junction {
  public Arrow(Var x, Var y, Var z) throws FailException {
    JSolver.post((x.eq(A).and(y.eq(P).and(z.eq(A)))).or
                 (x.eq(M).and(y.eq(P).and(z.eq(M)))).or
                 (x.eq(P).and(y.eq(M).and(z.eq(P)))));
  }
}

public final class Fork extends Junction {
  public Fork(Var x, Var y, Var z) throws FailException {
    JSolver.post((x.eq(A).and(y.eq(A).and(z.eq(M)))).or
                 (x.eq(M).and(y.eq(A).and(z.eq(A)))).or
                 (x.eq(A).and(y.eq(M).and(z.eq(A)))).or
                 (x.eq(P).and(y.eq(P).and(z.eq(P)))).or
                 (x.eq(M).and(y.eq(M).and(z.eq(M)))));
  }
}
```

So far we have defined four classes to represent the domain objects plus the junction constraints of the line-labelling problem. The next task is to create the instances that represent the actual edges and junctions of the polyhedron to be labelled (shown in Figure 3). This will be performed within the *main()* function of the *Waltz* utility class.

4.3 Creating the Polyhedron

We will first create a vector to store the 15 edges of our example polyhedron using the JSolver *VarVector* class that represents a vector of constrained integer variables. Each variable within this vector will have an initial domain between [0..3], representing the four possible labels of our example – 0 represents the label 'A' and 3 represents the label 'P' as defined previously in the *Junction* abstract class. The first argument "15" to *JSolver.varVector()* indicates the number of constrained variables to be created and stored in the vector.

```
VarVector edges = JSolver.varVector(15, 0, 3);
Var E1 = edges.elementAt(0);
Var E2 = edges.elementAt(1);
Var E3 = edges.elementAt(2);
Var E4 = edges.elementAt(3);
Var E5 = edges.elementAt(4);
Var E6 = edges.elementAt(5);
Var E7 = edges.elementAt(6);
Var E8 = edges.elementAt(7);
Var E9 = edges.elementAt(8);
Var E10 = edges.elementAt(9);
Var E11 = edges.elementAt(10);
Var E12 = edges.elementAt(11);
Var E13 = edges.elementAt(12);
Var E14 = edges.elementAt(13);
Var E15 = edges.elementAt(14);
```

For convenience, the program above also assigns a Java variable to each element of the *VarVector* to represent each of the 15 edges. After creating the edges, we next connect them together to form the junctions of our polyhedron. We do this by creating instances of the *L*, *Fork* and *Arrow* classes:

```
L J2 = new L(E1, E2);
L J6 = new L(E5, E6);
L J8 = new L(E7, E8);
Fork J4 = new Fork(E4, E3, E14);
Fork J9 = new Fork(E9, E11, E10);
Fork J11 = new Fork(E12, E13, E15);
Arrow J1 = new Arrow(E8, E9, E1);
Arrow J3 = new Arrow(E2, E10, E3);
Arrow J5 = new Arrow(E4,  E15, E5);
Arrow J7 = new Arrow(E6, E13, E7);
Arrow J10 = new Arrow(E12, E14, E11);
```

The constraint that the junction at each end of an edge must label the shared edge consistently is encoded in our program by having the two junction objects "share" the same edge variable object and the fact that a constrained variable can only take on one value at a time.

4.4 Searching for Solutions

We have now created all the CSP variables, defined their domains, and posted all the necessary constraints relevant to our line-labelling problem. The remaining step is to perform non-deterministic search to generate sets of labels. This can be done using two simple lines of code:

```
JSolver.activate(JSolver.generate(edges));
while (JSolver.nextSolution()) System.out.println(edges);
```

JSolver.activate() is a JSolver function to place a goal on the goal stack. *JSolver.generate()* is a predefined goal that instantiates each constrained variable within a vector of variables. Basically, *activate()* defines a goal to solve all the variables stored in *edges*. The *while* loop generates all the solutions in Prolog-like fashion and prints them out to the screen. The *toString*() method for *VarVector* is predefined to display all the variables stored in the vector. The program prints out the following four possible solutions or labelling for this polyhedron:

```
[[0] [0] [0] [0] [0] [0] [0] [0] [3] [3] [3] [3] [3] [2] [3]]
[[0] [0] [0] [0] [0] [2] [2] [0] [3] [3] [3] [3] [3] [2] [3]]
[[0] [2] [2] [2] [2] [0] [0] [0] [3] [3] [3] [3] [3] [2] [3]]
[[2] [0] [0] [0] [0] [0] [0] [2] [3] [3] [3] [3] [3] [2] [3]]
```

The sets of variable values shown above correspond to the following sets of labelling for the given polyhedron line drawing:

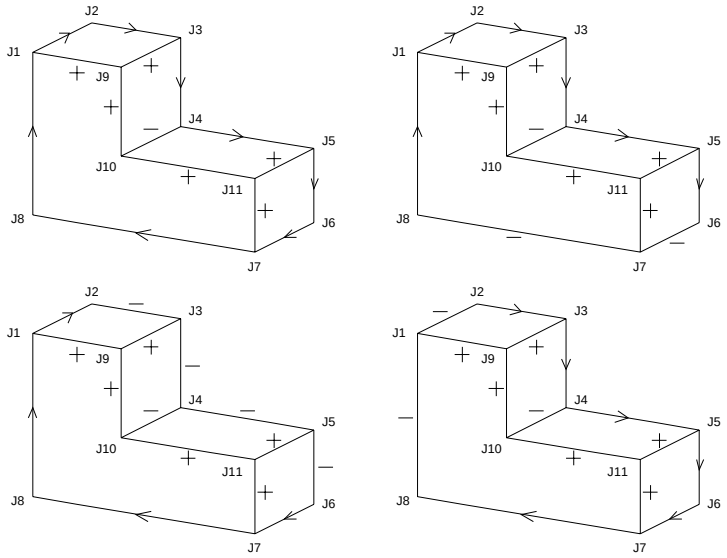


Fig. 6 Four sets of line labelling generated by the JSolver line labelling program.

The following is the complete listing of the Java line-labelling program:

```
public abstract class Junction {
  protected final static int A = 0, R = 1, M = 2, P = 3;
}
public final class L extends Junction {
  public L(Var x, Var y) throws FailException {
    JSolver.post((x.eq(A).and(y.eq(M))).or
                 (x.eq(A).and(y.eq(A))).or
                 (x.eq(M).and(y.eq(A))).or
                 (x.eq(R).and(y.eq(P))).or
                 (x.eq(R).and(y.eq(R))).or
                 (x.eq(P).and(y.eq(R))));
  }
}
public final class Arrow extends Junction {
  public Arrow(Var x, Var y, Var z) throws FailException {
    JSolver.post((x.eq(A).and(y.eq(P).and(z.eq(A)))).or
                 (x.eq(M).and(y.eq(P).and(z.eq(M)))).or
                 (x.eq(P).and(y.eq(M).and(z.eq(P)))));
  }
}
public final class Fork extends Junction {
  public Fork(Var x, Var y, Var z) throws FailException {
    JSolver.post((x.eq(A).and(y.eq(A).and(z.eq(M)))).or
                 (x.eq(M).and(y.eq(A).and(z.eq(A)))).or
                 (x.eq(A).and(y.eq(M).and(z.eq(A)))).or
                 (x.eq(P).and(y.eq(P).and(z.eq(P)))).or
                 (x.eq(M).and(y.eq(M).and(z.eq(M)))));
  }
}
```

```
public abstract class Waltz {
  public static void main(String argv[]) throws FailException {
    VarVector edges = JSolver.varVector(15, 0, 3);
    Var E1 = edges.elementAt(0);
    Var E2 = edges.elementAt(1);
    Var E3 = edges.elementAt(2);
    Var E4 = edges.elementAt(3);
    Var E5 = edges.elementAt(4);
    Var E6 = edges.elementAt(5);
    Var E7 = edges.elementAt(6);
    Var E8 = edges.elementAt(7);
    Var E9 = edges.elementAt(8);
    Var E10 = edges.elementAt(9);
    Var E11 = edges.elementAt(10);
    Var E12 = edges.elementAt(11);
    Var E13 = edges.elementAt(12);
    Var E14 = edges.elementAt(13);
    Var E15 = edges.elementAt(14);

    L J2 = new L(E1, E2);
    L J6 = new L(E5, E6);
    L J8 = new L(E7, E8);
    Fork J4 = new Fork(E4, E3, E14);
    Fork J9 = new Fork(E9, E11, E10);
    Fork J11 = new Fork(E12, E13, E15);
    Arrow J1 = new Arrow(E8, E9, E1);
    Arrow J3 = new Arrow(E2, E10, E3);
    Arrow J5 = new Arrow(E4,  E15, E5);
    Arrow J7 = new Arrow(E6, E13, E7);
    Arrow J10 = new Arrow(E12, E14, E11);

    JSolver.activate(JSolver.generate(edges));
    while (JSolver.nextSolution()) System.out.println(edges);
  }
}
```

Fig. 7 The Java line labelling program in its entirety.

5. CONCLUSION

Despite the complexity of the line-labelling problem, the program listing shown in Figure 7 is remarkably short and crisp. This illustrates the power of combining Java's object-orientation with constraint programming techniques. This paper showed how difficult AI problems, such as line labelling, can easily be implemented in Java using the JSolver class library that provides constraint propagation, declarative programming, and non-deterministic search. The simplicity with which constraint-programming techniques can be integrated with Java opens up a whole new spectrum of potential AI-enabled Java applications.

REFERENCES

[1] M. Clowes, "On Seeing Things," *Artificial Intelligence*, 2(1), 79-116, 1971.

[2] A. Colmerauer, *An Introduction to Prolog III,* Communications of the ACM, 33(7), pp.69-90, 1990.

[3] A. Guzman, "Computer Recognition of Three-Dimensional Objects in a Visual Scene," Technical Report AI-TR-228, MIT Artificial Intelligence Laboratory, Cambridge, MA, December 1968.

[4] R.M. Haralick and L. Shapiro, "The consistent labeling problem: Part 1" *IEEE Trans. Pattern Anal. Machine Intelligence,* PAMI-1, 173-184, 1979.

[5] D. Huffman, "Impossible Objects as Nonsense Sentences," in B. Meltzer and D. Michie (eds.), *Machine Intelligence,* Vol. 6, Edinburgh University Press, Edinburgh, UK, pp. 295-323, 1971.

[6] *ILOG Solver Reference Manual,* Version 4.3, ILOG, 1998.

[7] *JSolver Reference Manual,* Version 0.9, 1997.

[8] V. Kumar, "Algorithms for Constraint Satisfaction Problems: A Survey," In *AI Magazine,* 13(1), pp.32-44, 1992.

[9] C. Le Pape, "Using Object-Oriented Constraint Programming Tools to Implement Flexible "Easy-to-use" Scheduling Systems," In *Proceedings of the NSF Workshop on Intelligent, Dynamic Scheduling for Manufacturing,* Cocoa Beach, Florida, 1993.

[10] A.K. Mackworth, "Consistency in Networks of Relations," In *Artificial Intelligence,* 8, pp.99-118, 1977.

[11] J.-F. Puget, "A C++ Implementation of CLP," In *ILOG Solver Collected Papers,* ILOG SA, France, 1994.

[12] J.-F. Puget, "Object-Oriented Constraint Programming for Transportation Problems," In *ILOG Solver Collected Papers,* ILOG SA, France, 1994.

[13] *Rational Rose User's Guide,* Rational Software Corporation, Revision 3.0, 1995.

[14] J.M. Siskind and D.A. McAllester, "Nondeterministic Lisp as a Substrate for Constraint Logic Programming," In *Proceedings of the Eleventh National Conference on Artificial Intelligence,* Washington, DC, pp.133-138, July, 1993.

[15] G.L. Steele Jr., *The Definition and Implementation of a Computer Programming Language Based on Constraints,* Ph.D. Thesis, MIT, 1980.

[16] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming,* MIT Press, 1989.

[17] D.L. Waltz, "Generating Semantic Descriptions from Drawings of Scenes with Shadows," Technical Report, AI-TR-271, MIT Artificial Intelligence Laboratory, Cambridge, MA, November 1972.

[18] D.L. Waltz, "Understanding Line Drawings of Scenes with Shadows," In *The Psychology of Computer Vision,* McGraw-Hill, pp.19-91, 1975.