Many problems in operations research and digital circuit design are combinatorial problems. No general and efficient algorithms exist to solve these difficult (NP-complete [11]) problems, which can be viewed as search problems with constraints. Two main approaches can be distinguished at the moment: general tools and specialized programs. General tools (like theorem provers [28]) support the declarative statement of problems but are too inefficient. Specialized programs require much programming effort and are hard to maintain.

Logic programming, as exemplified by PROLOG, provides a powerful language for a logical (declarative) formulation of combinatorial problems. Its relational form and the logical variables are entirely adequate to stating problems in a declarative way, and its nondeterministic computation liberates the user from tree-search programming. However, until now, logic-programming languages have been used to solve only “small” combinatorial problems. The reason is the inefficiency of these languages due to their search procedure based on the generate-and-test paradigm.

On the other hand, constraint programming and consistency-checking techniques provide better problem-solving paradigms. Constraint programming emphasizes...
local value-propagation techniques combined with a demon-driven computation [23, 22]. Consistency-checking techniques include search algorithms like forward checking and lookahead [13] as well as general algorithms like arc and path consistency [19, 8]. The general idea behind these techniques is to use constraints to prune the search space in an \textit{a priori} way, i.e., before the generation of values, instead of using them as tests as in the case of generate-and-test method. This introduces a new problem-solving paradigm: reasoning by \textit{constraint propagation}. These techniques have been used in some problem solvers such as \texttt{REF-ARF} [7] and \texttt{ALICE} [17]. It is of course possible to write logic programs using such techniques. But this requires more programming effort and is rather inefficient, since we have to write a kind of metainterpreter.

For this reason, we have studied the introduction of constraint solving and consistency-checking techniques inside logic programming [24, 4, 5, 25–27, 21, 2]. These ideas have been implemented in a system called \texttt{CHIP} (Constraint Handling in Prolog). \texttt{CHIP} is a logic-programming language based on the concept of “active” constraints [9]. Its reasoning mechanism on constraints consists of a sophisticated constraint solving and consistency-checking (forward and lookahead) techniques combined with a demon-driven computation. This mechanism has been further specialized to three important computation domains: \textit{boolean expressions}, \textit{linear arithmetic terms on rational numbers}, and \textit{finite domains}. In addition to the primitive constraints (equality, disequality, inequalities, and more complex ones) available in \texttt{CHIP}, the user can define his/her own constraints (which can be any logic program) and the strategy to use them. Finally, some higher-order predicates for optimization purposes (providing logic programming with a kind of depth-first branch-and-bound technique) enable combinatorial optimization and integer linear-programming problems to be solved.

With these extensions, we have solved in \texttt{CHIP} several problems which were infeasible within standard PROLOG, with an efficiency comparable to that of specific programs written in procedural languages. Some of them are “real-life” problems in the areas of operations research (e.g., graph coloring, project management, job-shop scheduling, warehouse location, integer linear programming) and digital circuit design (e.g., simulation, symbolic verification, fault diagnosis, test generation, channel routing). \texttt{CHIP} is also very powerful for solving logical-arithmetic puzzles (e.g., cryptarithmetic problems, \textit{n} queens, crosswords, mastermind).

The approach taken in \texttt{CHIP} is related to recent work in the PROLOG-III and CLP projects, which are also aiming at the introduction of constraint-solving techniques inside logic programming. PROLOG-III [3] uses a simplex-like algorithm to solve linear equations and inequations on rational numbers. It has also a saturation method to deal with boolean algebra. CLP [14] defines a formal framework which provides a theoretical basis for building a logic-programming language based on constraint solving (which generalizes the notion of unification). An instance of this scheme for handling linear equations and inequations on real numbers has been implemented in CLP(R) [15]. Besides some technical differences, \texttt{CHIP} differs from these two implementations mainly in putting the emphasis on (discrete) combinatorial problems and providing more general control mechanisms (demon-driven computation, forward and lookahead checking).

In this paper, we show how we solve in \texttt{CHIP} three real-life problems: a scheduling problem with disjunctive constraints, a graph-coloring problem, and a
microcode-generation problem. These very difficult combinatorial problems have not yet been attacked seriously in logic programming because of their great complexity (NP-completeness). We show that on these problems logic programs can be as efficient as special-purpose programs written in imperative languages while being quite easier to write and maintain. For instance, for the last problem (microcode label assignment) the resulting CHIP program not only is much more concise and easier to modify than a large specific FORTRAN program, but also has roughly the same efficiency.

In the following we will not give an overview of the CHIP system in a separate section. The techniques necessary for the present examples will be explained briefly inside the text. For more details about CHIP (including features like boolean unification, delay mechanisms, and demons), the reader should refer to the above mentioned publications.

2. EXAMPLE 1: SCHEDULING WITH DISJUNCTIVE CONSTRAINTS

The purpose of this example is to show the way CHIP can be used to solve a real-life scheduling problem. It contains a discussion about how precedence and distance constraints are introduced and how disjunctive constraints are best handled in this context.

2.1. Problem Statement

The problem is to find a schedule that minimizes the time to build a five-segment bridge (see Figure 1). It is taken from Bartusch's Ph.D. thesis on scheduling problems [1]. The project contains a set of 46 tasks (see Figure 2) and a set of constraints between these tasks. Beside the usual precedence constraints, there are disjunctive constraints due to the restricted resources which must be shared by several tasks. For example, the caterpillar is used by tasks V1 and V2, which therefore cannot be performed at the same time.

There are also the following additional constraints.

(1) The time between the completion of a particular formwork and the completion of its corresponding concrete foundation is at most 4 days.

(2) There are at most 3 days between the end of a particular excavation (or foundation piles) and the beginning of the corresponding formwork.

(3) The formworks must start at least 6 days after the beginning of the erection of the temporary housing.

(4) The removal of the temporary housing can start two days before the end of the last masonry work.

(5) The delivery of the performed bearers occurs exactly 30 days after the beginning of the project.

2.2. Problem Solution

The solution of the problem is presented by successive refinements. We first show how a scheduling problem with precedence constraints only can be solved by the
<table>
<thead>
<tr>
<th>N</th>
<th>Name</th>
<th>description</th>
<th>duration</th>
<th>resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PA</td>
<td>beginning of project</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>A1</td>
<td>excavation (abutment 1)</td>
<td>4</td>
<td>excavator</td>
</tr>
<tr>
<td>3</td>
<td>A2</td>
<td>excavation (pillar 1)</td>
<td>2</td>
<td>excavator</td>
</tr>
<tr>
<td>4</td>
<td>A3</td>
<td>excavation (pillar 2)</td>
<td>2</td>
<td>excavator</td>
</tr>
<tr>
<td>5</td>
<td>A4</td>
<td>excavation (pillar 3)</td>
<td>2</td>
<td>excavator</td>
</tr>
<tr>
<td>6</td>
<td>A5</td>
<td>excavation (pillar 4)</td>
<td>2</td>
<td>excavator</td>
</tr>
<tr>
<td>7</td>
<td>A6</td>
<td>excavation (abutment 2)</td>
<td>5</td>
<td>excavator</td>
</tr>
<tr>
<td>8</td>
<td>P1</td>
<td>foundation piles 2</td>
<td>20</td>
<td>pile-driver</td>
</tr>
<tr>
<td>9</td>
<td>P2</td>
<td>foundation piles 3</td>
<td>13</td>
<td>pile-driver</td>
</tr>
<tr>
<td>10</td>
<td>UE</td>
<td>erection of temporary housing</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>S1</td>
<td>formwork (abutment 1)</td>
<td>8</td>
<td>carpentry</td>
</tr>
<tr>
<td>12</td>
<td>S2</td>
<td>formwork (pillar 1)</td>
<td>4</td>
<td>carpentry</td>
</tr>
<tr>
<td>13</td>
<td>S3</td>
<td>formwork (pillar 2)</td>
<td>4</td>
<td>carpentry</td>
</tr>
<tr>
<td>14</td>
<td>S4</td>
<td>formwork (pillar 3)</td>
<td>4</td>
<td>carpentry</td>
</tr>
<tr>
<td>15</td>
<td>S5</td>
<td>formwork (pillar 4)</td>
<td>4</td>
<td>carpentry</td>
</tr>
<tr>
<td>16</td>
<td>S6</td>
<td>formwork (abutment 2)</td>
<td>10</td>
<td>carpentry</td>
</tr>
<tr>
<td>17</td>
<td>B1</td>
<td>concrete foundation (abutment 1)</td>
<td>1</td>
<td>concrete-mixer</td>
</tr>
<tr>
<td>18</td>
<td>B2</td>
<td>concrete foundation (pillar 1)</td>
<td>1</td>
<td>concrete-mixer</td>
</tr>
<tr>
<td>19</td>
<td>B3</td>
<td>concrete foundation (pillar 2)</td>
<td>1</td>
<td>concrete-mixer</td>
</tr>
<tr>
<td>20</td>
<td>B4</td>
<td>concrete foundation (pillar 3)</td>
<td>1</td>
<td>concrete-mixer</td>
</tr>
<tr>
<td>21</td>
<td>B5</td>
<td>concrete foundation (pillar 4)</td>
<td>1</td>
<td>concrete-mixer</td>
</tr>
<tr>
<td>22</td>
<td>B6</td>
<td>concrete foundation (abutment 2)</td>
<td>1</td>
<td>concrete-mixer</td>
</tr>
<tr>
<td>23</td>
<td>AB1</td>
<td>concrete setting time (abutment 1)</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>AB2</td>
<td>concrete setting time (pillar 1)</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>AB3</td>
<td>concrete setting time (pillar 2)</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>26</td>
<td>AB4</td>
<td>concrete setting time (pillar 3)</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>27</td>
<td>AB5</td>
<td>concrete setting time (pillar 4)</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>28</td>
<td>AB6</td>
<td>concrete setting time (abutment 2)</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>29</td>
<td>M1</td>
<td>masonry work (abutment 1)</td>
<td>16</td>
<td>bricklaying</td>
</tr>
<tr>
<td>30</td>
<td>M2</td>
<td>masonry work (pillar 1)</td>
<td>8</td>
<td>bricklaying</td>
</tr>
<tr>
<td>31</td>
<td>M3</td>
<td>masonry work (pillar 2)</td>
<td>8</td>
<td>bricklaying</td>
</tr>
<tr>
<td>32</td>
<td>M4</td>
<td>masonry work (pillar 3)</td>
<td>8</td>
<td>bricklaying</td>
</tr>
<tr>
<td>33</td>
<td>M5</td>
<td>masonry work (pillar 4)</td>
<td>8</td>
<td>bricklaying</td>
</tr>
<tr>
<td>34</td>
<td>M6</td>
<td>masonry work (abutment 2)</td>
<td>20</td>
<td>bricklaying</td>
</tr>
<tr>
<td>35</td>
<td>L</td>
<td>delivery of the preformed bearers</td>
<td>2</td>
<td>crane</td>
</tr>
<tr>
<td>36</td>
<td>T1</td>
<td>positioning (preformed bearer 1)</td>
<td>12</td>
<td>crane</td>
</tr>
<tr>
<td>37</td>
<td>T2</td>
<td>positioning (preformed bearer 2)</td>
<td>12</td>
<td>crane</td>
</tr>
<tr>
<td>38</td>
<td>T3</td>
<td>positioning (preformed bearer 3)</td>
<td>12</td>
<td>crane</td>
</tr>
<tr>
<td>39</td>
<td>T4</td>
<td>positioning (preformed bearer 4)</td>
<td>12</td>
<td>crane</td>
</tr>
<tr>
<td>40</td>
<td>T5</td>
<td>positioning (preformed bearer 5)</td>
<td>12</td>
<td>crane</td>
</tr>
<tr>
<td>41</td>
<td>UA</td>
<td>removal of the temporary housing</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>42</td>
<td>V1</td>
<td>filling 1</td>
<td>15</td>
<td>caterpillar</td>
</tr>
<tr>
<td>43</td>
<td>V2</td>
<td>filling 2</td>
<td>10</td>
<td>caterpillar</td>
</tr>
<tr>
<td>44</td>
<td>K1</td>
<td>point 1 of cost function</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>45</td>
<td>K2</td>
<td>point 2 of cost function</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>PE</td>
<td>end of project</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

**FIGURE 2.** Data for the bridge problem.
system. Next, we describe how distance constraints and disjunctive constraints can be added to it. Finally, we show how the minimization constraint can be introduced in order to find the optimal solution.

2.2.1. Scheduling with Precedence Constraints

2.2.1.1. Problem Statement. We first consider a scheduling problem where only the precedence constraints are present. More precisely, the following definition can be stated. We are given a set of \( n \) tasks where each task \( i \) is characterized by its duration \( d_i \) and a set of precedence constraints with some other tasks. A precedence constraint between tasks \( i \) and \( j \) implies that task \( j \) must start after the completion of task \( i \). The project starts at time 0, and the problem is to find a schedule that minimizes the total duration of the project, i.e., the time of its termination. This problem is known as the critical-path problem. For convenience, we add a fictitious task of duration 0, called the end task, which is preceded by all other tasks.

2.2.1.2. Problem Notation. Since no task can start before all the tasks that precede it have been completed, the earliest date \( t_i \) to start task \( i \) is

\[
t_i = \max(t_j + d_j)
\]

for all tasks \( j \) which precede task \( i \). It follows that the minimal duration of the project is \( t_{\text{end}} \), i.e., the earliest date to start the end task.

If the duration of the project is \( t_{\text{end}} \), the latest date \( T_i \) to start task \( i \) is given by

\[
T_i = \min(T_j - d_j)
\]

for all tasks \( j \) such that task \( i \) precedes task \( j \), assuming that \( t_{\text{end}} \) is equal to \( t_{\text{end}} \). The float or slack \( m_i \) of task \( i \) is defined to be the difference between the earliest and the latest date \( (m_i = T_i - t_i) \). The tasks whose floats are zero are called critical tasks. If one of these is delayed, to whatever extent, the minimal duration of the project will be increased to the same extent.

2.2.1.3. Problem Representation. One of the basic extensions of CHIP is domain variables, that is, variables that range over a finite set of values [24]. Domains are specified through domain declarations and form the basis for consistency-checking techniques. Domain variables will be used for all examples in this paper.

The scheduling problem can now be represented in the following way. To each task we associate a domain variable \( S_i \) representing the starting date of task \( i \). Its domain can be defined, for instance, as the interval \([0, l_g]\) where \( l_g \) is the sum of all task durations. Now a precedence constraint between task \( i \) and task \( j \) can be expressed as

\[
S_j \geq S_i + d_j.
\]

2.2.1.4. Example. To explain the technique used in CHIP, we first switch to a simpler example, shown in Figure 3 taken from [12].

Given these data, a simple logic program can be written following the above scheme. The program is shown in Figure 4. The domain variables \( S_4, \ldots, S_K \) represent the starting dates of the tasks of the project, and \( S_{\text{end}} \) is the starting date
<table>
<thead>
<tr>
<th>code of the task</th>
<th>name of the task</th>
<th>duration (in weeks)</th>
<th>Previous tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Masonry</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Carpentry for roof</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>Roof</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>Sanitary and electrical installations</td>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Front</td>
<td>2</td>
<td>D,C</td>
</tr>
<tr>
<td>F</td>
<td>Windows</td>
<td>1</td>
<td>D,C</td>
</tr>
<tr>
<td>G</td>
<td>Garden</td>
<td>1</td>
<td>D,C</td>
</tr>
<tr>
<td>H</td>
<td>Ceiling</td>
<td>3</td>
<td>F</td>
</tr>
<tr>
<td>J</td>
<td>Painting</td>
<td>2</td>
<td>H</td>
</tr>
<tr>
<td>K</td>
<td>Moving in</td>
<td>1</td>
<td>E,G,J</td>
</tr>
</tbody>
</table>

**FIGURE 3.** An example of scheduling with precedence constraints.

of the end task, i.e., the finishing date. The precedence constraints are then stated as above described. The *minval* procedure assigns $S_{end}$ to the smallest possible value.

Consider now the program behavior. It can be seen as a three-step process:

1. Forward propagation of constraints.
2. Assignment of the end-task variable.

The forward propagation ensures that for each task $i$

$$t_i \geq \max(t_j + d_j)$$

for all tasks $j$ that precede task $i$. In other words, it computes for each task the

$$\text{domain } \text{pert}(0..30).$$

$$\text{pert}([\text{SA, SB, SD, SC, SE, SF, SG, SH, SJ, SK, Send}) \leftarrow \text{precedence}([\text{SA, SB, SD, SC, SE, SF, SG, SH, SJ, SK, Send}), \text{minval}(\text{Send}).$$

$$\text{precedence}([\text{SA, SB, SD, SC, SE, SF, SG, SH, SJ, SK, Send}) \leftarrow$$

$$\begin{align*}
\text{SB} & \geq \text{SA} + 7, \quad (1) \\
\text{SD} & \geq \text{SA} + 7, \quad (2) \\
\text{SC} & \geq \text{SB} + 3, \quad (3) \\
\text{SE} & \geq \text{SC} + 1, \quad (4) \\
\text{SE} & \geq \text{SD} + 8, \quad (5) \\
\text{SG} & \geq \text{SC} + 1, \quad (6) \\
\text{SG} & \geq \text{SD} + 8, \quad (7) \\
\text{SF} & \geq \text{SD} + 8, \quad (8) \\
\text{SF} & \geq \text{SC} + 1, \quad (9) \\
\text{SH} & \geq \text{SF} + 1, \quad (10) \\
\text{SJ} & \geq \text{SH} + 3, \quad (11) \\
\text{SK} & \geq \text{SG} + 1, \quad (12) \\
\text{SK} & \geq \text{SE} + 2, \quad (13) \\
\text{SK} & \geq \text{SJ} + 2, \quad (14) \\
\text{Send} & \geq \text{SK} + 1. \quad (15)
\end{align*}$$

**FIGURE 4.** Scheduling with precedence constraints.
earliest date to start it. This is achieved by the reasoning about variation intervals.
Given a precedence constraint \( S_j \geq S_i + D_i \), we immediately deduce that:

1. \( S_j \geq \min(S_j) + D_i \), where \( \min(S_j) \) is the minimum value in the domain of \( S_j \).
2. \( S_j \leq \max(S_j) - D_i \), where \( \max(S_j) \) is the maximal value in the domain of \( S_j \).

For instance, for our example we deduce that
\[
S_B \geq 7 \text{ and } S_A \leq 23 \text{ (constraint 1)},
\]
\[
S_D \geq 7 \text{ (constraint 2)},
\]
\[
S_C \geq 10 \text{ and } S_B \leq 27 \text{ (constraint 3), and}
\]
\[
S_A \leq 20 \text{ (by reconsidering constraint 1)}.
\]

Pursuing the same reasoning for the other constraints, we find that
\[
S_B \in [0, 8], \quad S_A \in [7, 19], \quad S_C \in [10, 22], \quad S_D \in [7, 15],
\]
\[
S_k \in [15, 27], \quad S_F \in [15, 23], \quad S_E \in [15, 28], \quad S_H \in [16, 24],
\]
\[
S_J \in [19, 27], \quad S_K \in [21, 29], \quad S_{\text{end}} \in [22, 30].
\]

Since the constraints are not yet solved, they remain in the resolvent.

The second step consists simply in assigning the minimum possible value to \( S_{\text{end}} \).
This is achieved by the \texttt{minval} procedure, which assigns 22 to \( S_{\text{end}} \), the smallest
value in its domain.

The third step is the backward propagation. Due to the instantiation of \( S_{\text{end}} \), the
inequality constraints are reconsidered. The constraint (15),
\[
S_{\text{end}} \geq S_K + 1
\]

now becomes \( S_K \leq 21 \). This implies that all the constraints involving \( S_K \), i.e. (11),
(12), and (13), are reconsidered, possibly reducing the domains of \( S_E, S_G \) and \( S_J \).

The reasoning continues for all the other constraints. At the end we have
\[
S_A = 0, \quad S_B = 7, \quad S_F = 15, \quad S_H = 16,
\]
\[
S_J = 19, \quad S_K = 21, \quad S_{\text{end}} = 22,
\]
\[
S_B \in [7, 11],
\]
\[
S_C \in [10, 14],
\]
\[
S_E \in [15, 19],
\]
\[
S_G \in [15, 20].
\]

with one remaining inequality
\[
S_C \geq S_B + 3.
\]

After this three-step process, the variables corresponding to critical tasks have
been instantiated to their unique possible value. In terms of the critical-path problem, the problem is solved. But in terms of logic programming there is still one floundering (remaining) constraint, which is \( S_C \geq S_B + 3 \). This can be solved by assigning values for \( S_C \) and \( S_B \), chosen from their domains satisfying the inequation (for instance by assigning the smallest possible values).

As we can see from the presentation, the interpretation of the program by CHIP follows the usual approach used in operations research. It is important to point out
that no explicit programming is necessary: the user simply states the constraints. This should be compared with the usual logic-programming approach to this problem [16].

2.2.2. Adding Distance Constraints. Some additional distance constraints can be easily included in the previous formulation without changing the deterministic nature of the problem. For instance, a constraint like "task $i$ must start at least 10 days after the completion of task $j$" can be expressed as

$$S_i \geq S_j + d_j + 10.$$ 

where $S_i$ and $S_j$ are the variables associated to tasks $i$ and $j$. This constraint can be simply added to the program to take such constraints into account.

2.2.3. Adding Disjunctive Constraints. In CHIP, precedence and distance constraints can be solved in a deterministic way, without making choices, just by constraint propagation. This is not the case if we add disjunctive constraints. With disjunctive constraints the scheduling problem becomes NP-complete [11]. A disjunctive constraint states that two tasks $i$ and $j$ cannot be performed simultaneously, i.e., either task $i$ must precede task $j$ or task $j$ must precede task $i$. These constraints are due to the fact that tasks $i$ and $j$ use the same resource.

For that purpose we define a predicate

$$\text{disjunctive}(S_i, D_i, S_j, D_j),$$

where $S_i$ and $S_j$ represent the variables associated to the tasks, and $D_i$ and $D_j$ are their respective durations. This predicate can be defined by the following clauses:

\[
\text{disjunctive}(S_i, D_i, S_j, D_j) \leftarrow \\
S_j \geq S_i + D_i.
\]

\[
\text{disjunctive}(S_i, D_i, S_j, D_j) \leftarrow \\
S_i \geq S_j + D_j.
\]

We now have to decide how to use this predicate in a procedural way. In CHIP, at least three different strategies can be used:

1. Applying forward checking.
2. Applying lookahead.
3. Using the clauses as a choice point.

The first approach consists in stating a forward declaration

**forward** disjunctive($d, g, d, g$).

A forward declaration [25] is a control mechanism that selects a constraint as soon as at most one domain variable remains uninstantiated. The domain of this variable is then reduced in such a way that all the remaining values satisfy the constraint. This handling solves the constraint once for all.

The above declaration will thus specify that the predicate **disjunctive** will be used as soon as one variable ($S_i$ or $S_j$) has received a value to reduce the domain of the
other. More precisely, each value that does not satisfy the constraint is removed from the domain of that variable.

The second approach consists in stating a lookahead declaration

```
lookahead disjunctive(d,g,d,g).
```

A lookahead declaration [26] is a control mechanism that selects a constraint when several domain variables remain uninstantiated. The domain of each variable is then reduced in such a way that values that cannot possibly satisfy the constraint are removed.

For the present constraint, all combinations of possible values for the variables \( S_i \) and \( S_j \) will be checked immediately. Each time the domain of one variable is modified, the constraint will be reconsidered. This way, the domains are reduced early, but the computation cost is very high.

The third possibility consists in using the clauses as choice points. In this case one clause is chosen which introduces an inequality constraint of the form

\[
S_j \geq S_i + d_i,
\]

which is added to the current resolvent. This means that we assume that task \( j \) has to be scheduled after task \( i \). The system now tries to find a schedule solving the problem with this additional constraint. The immediate effect of the inequation is to reduce the domain of \( S_j \) and \( S_i \). This starts a constraint propagation as all the constraints involving \( S_i \) or \( S_j \) are reconsidered. This possibly reduces the domain of other variables awakening other constraints, and so on. If later we backtrack to this point, the alternative clause will be chosen, i.e., task \( i \) will be scheduled after task \( j \).

For scheduling problems, the last approach seems to be much more appropriate than the first two. Forward checking and lookahead use a local saturation method on values, which is very expensive in the case of the large domains. The main problem in disjunctive scheduling is to find an ordering of the tasks. Once this is done, an assignment of starting dates can be given easily. For that reason, the last approach which defines an ordering of tasks should be preferred. This defines a least-commitment strategy.

Note that the user can choose the strategy which is most appropriate for his/her problem (this is not the case in problem solvers like ALICE [17]). At the same time, he/she does not need to program the constraint-propagation and consistency-checking mechanisms, which would be necessary in other languages.

### 2.2.4. Adding the Minimization Constraint

To come back to our example of the bridge-building problem, the basic program will look like

```
bridge(L,End) <--
define_domain(L,End),
stating_precedence_constraints(L),
stating_distance_constraints(L),
choosing(L,End).
```

\( L \) is the list of all starting dates of the tasks. The first predicate is used to define their domain. \( End \) is the domain variable representing the end date of the project. The next two predicates will set up the precedence and distance constraints as
described in the previous section. The predicate choosing\((L, \text{End})\) can be defined as

\[
\text{choosing}(L, \text{End}) \leftarrow \\
\text{stating_disjunctive_constraints}(L), \\
\text{minval}(\text{End}).
\]

The first predicate will use the disjunctive constraints to make choices, while the \text{minval} procedure will assign a value to the end date.

It now remains to express the minimality constraint. Operations-research problems often require to find a solution that optimizes (i.e., maximizes or minimizes) an evaluation function. For this purpose, several higher-order predicates are available in \text{CHIP}. They endow logic programming with a kind of branch-and-bound technique. For the present example, we may use the higher-order predicate

\[
\text{minimize_maximmum}(G, \text{List})
\]

where \(G\) is a goal and \(\text{List}\) is a list of domain variables or integers. This predicate will find the solution of \(G\) that minimizes the maximum value of the elements of \(\text{List}\). At the implementation level, this predicate will first search for a solution. It will then search for another solution with the additional constraint that its cost must be better than the already found solutions, and repeat this process until no better solution can be found. The optimal solution is then reached.

The program now becomes

\[
\text{bridge}(L, \text{End}) \leftarrow \\
\text{define_domain}(L, \text{End}), \\
\text{stating_precedence_constraints}(L), \\
\text{stating_distance_constraints}(L), \\
\text{minimize_maximmum}(\text{choosing}(L, \text{End}), [\text{End}]).
\]

2.3. Computation Results

The above program finds a first solution for the bridge problem with a cost of 110 after 4.5 seconds. The optimal solution with a cost of 104 is found after 7.5 seconds. The total time including proof of optimality is 42.5 seconds. All our execution times are given for a Sun 3/160.

In his thesis [1], Bartusch takes a different approach using a relaxation technique. He removes some of the disjunctive constraints to solve a simplified problem. The solution to the simplified problem gives a lower bound to the initial problem. If we relax all disjunctive constraints, we obtain a critical-path problem. Deciding which disjunctive constraints to keep is a difficult problem. Bartusch uses an iterative approach in his \text{FORTRAN} program. Details about the total execution time are not explicitly given in his thesis. But within this approach, he has to solve a relaxed problem that requires 20 minutes of computation time on a CYBER 175. The techniques used in [1] have been recently refined in [20]. Its implementation on a Sun 3 solves the problem in 43 seconds.
3. EXAMPLE 2: GRAPH COLORING

In the following example we show how CHIP can be used efficiently to solve graph-coloring problems. We apply here an intelligent choice of variables and a procedure to avoid redundant assignments. The program is faster than ALICE and special-purpose programs [6].

3.1. Problem Statement

The problem is to find the minimum number of colors to label the vertices of a graph so that no two adjacent vertices are assigned to the same color. This minimum number of colors is known as the chromatic number of the graph. Graph coloring, which is a generalization of map coloring, is also an NP-complete problem [11] and has many useful applications. For instance, problems of production scheduling, timetable planning, and clustering can be stated as graph-coloring problems [6, 12, 18].

3.2. Problem Solution

The basic idea of the solution presented here is to associate with each vertex a domain variable ranging from 1 to the number of vertices, each number representing a possible color. There is a disequality constraint (i.e. \(\neq\)) between every pair of vertices which are adjacent. A simple program consists in stating the disequality constraints and using a generator of values for the variables. The higher-order predicate minimize_maximum can be used for the minimization, since we want to minimize the maximum value given to the variables:

\[
\text{color_graph(Vertices)} \leftarrow \\
\text{state_constraints(Vertices),} \\
\text{minimize_maximum(labeling(Vertices),Vertices)}.
\]

where \(\text{state_constraints(Vertices)}\) is used to create a list of domain variables and to set up the disequality constraints, while \(\text{labeling(Vertices)}\) is a generator of values for the variables.

This generator should be designed carefully in order to

1. choose a good generation ordering for the variables,
2. avoid exploring redundant assignment of values.

We now consider both issues in some detail.

3.2.1. Choosing a Good Generation Ordering. The generation ordering plays a crucial role in the behavior of the program. The problem which appears at this level is to select the next variable to instantiate. The basic idea for this issue is to use the first-fail principle, which recommends to choose the most constrained variable. CHIP supports this principle by providing the programmer with various built-in predicates. In the present case, we may use a

\[
\text{deleteffc(Var,L,Rest)}
\]
predicate, where \( L \) is a list of domain variables and ground terms, and \( \text{Var} \) and \( \text{Rest} \) are variables. It assigns to \( \text{Var} \) the most constrained element of \( L \), while \( \text{Rest} \) is \( L \) without this element. Within this predicate, the most constrained element is the variable with the smallest domain or, in case of equality, the one that appears in the greatest number of constraints. In the terminology of graph theory, this means that we choose the variable that has the maximum degree. This selection has the advantage of reducing immediately the domains of many other domain variables, thus pruning the search very early.

3.2.2. Avoiding Redundant Assignments. Suppose that we have \( n \) colors at our disposal and that the vertices from 1 to \( n \) are to be labeled. Given a partial assignment of the vertices using \( i \) colors, we should consider only \( i + 1 \) colors for the vertex chosen to extend the current partial solution. The reason is that only two possibilities have to be considered:

1. assigning an already used color, i.e., a color whose number ranges between 1 and \( i \);
2. assigning a new color, i.e., a color that has not been used in the partial labeling and whose number ranges over \([i + 1, n]\).

While in the first case all \( i \) colors must be considered, in the second case only one needs to be chosen. Since this color does not appear in the partial solution, it does not matter which one we take from the remaining colors. They are simply different conventions for the same choice, and backtracking on this choice leads to redundancy because of the symmetry.

We therefore define a predicate

\[
\text{labelvar}(X,N,\text{New})
\]

which assigns a value between 1 and \( N + 1 \) to \( X \) and the maximum of \( X \) and \( N \) to \( \text{New} \). If \( N \) represents the number of colors used in the partial solution, this predicate will instantiate \( X \) either to an already used color or to the first not yet used color. \( \text{New} \) will be the number of colors used in the extended partial solution. We can implement \( \text{labelvar}(X,N,\text{New}) \) as

\[
\text{labelvar}(X,N,N) \leftarrow \\
\text{between}(X,1,N). \\
\text{labelvar}(X,N,X) \leftarrow \\
X = N + 1.
\]

where \( \text{between}(X,\text{Min},\text{Max}) \) generates values for \( X \) in the interval \([\text{Min},\text{Max}]\). The labeling procedure is thus

\[
\text{labeling}(L) \leftarrow \\
\text{labelingffc}(L,0). \\
\text{labellingffc}([],\text{Newup}). \\
\text{labellingffc}([X|Y],\text{Up}) \leftarrow \\
\text{deletelfc}(\text{Var},[X|Y],\text{Rest}), \\
\text{labelvar}(\text{Var},\text{Up},\text{Newup}), \\
\text{labellingffc}(\text{Rest},\text{Newup}).
\]
TABLE 1. Results for the graph-coloring problem.

<table>
<thead>
<tr>
<th>Graph</th>
<th>$N_v$</th>
<th>$N_c$</th>
<th>$T_c$</th>
<th>$T_o$</th>
<th>$T_p$</th>
<th>$T_T$</th>
<th>$N_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>110</td>
<td>324</td>
<td>0.13</td>
<td>1.33</td>
<td>1.25</td>
<td>2.7</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>125</td>
<td>0.1</td>
<td>0.15</td>
<td>0.45</td>
<td>0.7</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>67</td>
<td>73</td>
<td>0.06</td>
<td>0.6</td>
<td>0.46</td>
<td>1.1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>99</td>
<td>2112</td>
<td>1.95</td>
<td>2.8</td>
<td>4.9</td>
<td>9.6</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>99</td>
<td>578</td>
<td>0.93</td>
<td>1.43</td>
<td>2.17</td>
<td>4.5</td>
<td>1</td>
</tr>
</tbody>
</table>

$Up$ represents the number of colors used in the current partial solution. Since, at the beginning, the partial solution is empty, this predicate must be called by `labelingffc(L,0)`, where $L$ is the list of variables to be assigned.

The ability to take into account the symmetry of the problem is an advantage of our approach compared with a problem solver like ALICE. Since ALICE does not recognize symmetries, it spends much time in exploring redundant choices.

3.3. Computation Results

The above program has been used to color the planar graph (i.e. map) of [10] and the graphs given in [6]. The results are reported in Table 1.

We have given the number of vertices ($N_v$), the number of constraints ($N_c$), the time for generating the constraints ($T_c$), the time to find the optimal solution ($T_o$), the time to prove optimality ($T_p$), the total time ($T_T$), and the number of backtracks ($N_b$). Remember that all the times are given for a Sun 3/160. We can compare our CHIP program, for example on the last problem, with a special purpose program of Dodu et al. [6], which spends 822 seconds on a CDC 6600, and with ALICE, which spends 71 seconds on an IBM 370/168 and makes 124 backtrackings, while we spend 4.5 seconds and make only 1 backtrack.

4. EXAMPLE 3: MICROCODE LABEL ASSIGNMENT

In this section, we show that CHIP can easily be applied to other application domains besides operations research. The following example comes from the area of computer firmware development.

4.1. Problem Statement

The problem is to assign labels of symbolic microcode to binary addresses in a 256-address page of microcode memory. To improve efficiency and decrease memory usage, multiway branch targets share certain bits and contain the branch condition as part of the address. Therefore the labels cannot be assigned to consecutive addresses as in assembly language, but must be distributed over the address space to satisfy all the constraints. Other constraints used are increment and bit-mask operations. In addition, all labels must be assigned to different values. The example uses 178 out of 256 possible addresses.
4.2. Problem Solution

4.2.1. Domains. We solve the problem for one page of microcode memory at a time. This page contains 256 addresses. Each label is represented by a domain variable with a domain between 0 to 255.

4.2.2. Multiway Branches. There are 2-, 4-, 8-, and 16-way branch instructions. We explain the resulting constraints on a 4-way branch instruction

L0: BRANCH4 L1,L2,L3,L4

If we look at the binary representation (e.g. \( L1 = X7 \times 128 + X6 \times 64 + X5 \times 32 + 0 \times 16 + 0 \times 8 + X2 \times 4 + X1 \times 2 + X0 \)) of the addresses, we see the following constraints (Note that equal variables imply that the corresponding bits must be equal):

- \( L0: X7X6Y5Y4Y3Y2Y1Y0 \)
- \( L1: X7X6X5 00X2X1X0 \)
- \( L2: X7X6X5 01X2X1X0 \)
- \( L3: X7X6X5 10X2X1X0 \)
- \( L4: X7X6X5 11X2X1X0 \)

We can see three types of constraints.

4.2.2.1. EQUAL BITS. Some bits (\( X7, X6 \)) are shared between the origin (LO) and the targets (L1, L2, L3, L4) of the branch instruction. By using a mask (binary: 1100000, hexadecimal: \$C0\) and the bitwise "and" operation (denoted \( \land \)), we can formulate the constraint as

\[
\text{equal_bits}(L0,\$C0,L1),
\text{equal_bits}(L0,\$C0,L2),
\text{equal_bits}(L0,\$C0,L3),
\text{equal_bits}(L0,\$C0,L4).
\]

\( \text{equal_bits} \left( \text{Label1}, \text{Mask}, \text{Label2} \right) \leftarrow \text{Label1} \land \text{Mask} = \text{Label2} \land \text{Mask}. \)

We use a forward declaration

\( \text{forward equal_bits(d,g,d)}. \)

to solve this constraint as soon as one of the labels receives a value reducing the domain of the other variable.

4.2.2.2. CONDITION BITS. Another constraint is that the condition bits of the targets are fixed. If the condition bits are 00 then we branch to \( L1 \), if they are 01 then we branch to \( L2 \), etc. This restricts the domain of the label to values which have the correct bit combination at the position of the condition bits. This can be
expressed as

\[
\text{fixed}(L1, \$18, \$00),\\
\text{fixed}(L2, \$18, \$08),\\
\text{fixed}(L3, \$18, \$10),\\
\text{fixed}(L4, \$18, \$18).
\]

\[
\text{fixed(Label, Mask, Value)} \leftarrow \\
\text{Label} \land \text{Mask} = \text{Value}.
\]

With the forward declaration

\[
\text{forward} \text{ fixed}(d, g, g).
\]

we solve these constraints immediately. All values which do not satisfy the constraint are directly removed. This constraint alone already restricts the domain of a variable from 256 values to 64 values.

4.2.2.3. \textbf{Equal Distances.} All remaining bits (X7, X6, X5, X2, X1, X0) of the targets are equal for all targets. There are two ways to state this constraint. First, we can directly use bit-mask operations as defined above using \textit{equal bits} with a mask of binary 11100111, hexadecimal \$E7. This leads to constraints like

\[
\text{equal_bits}(L1, \$E7, L2),
\]

Second, we can use the representation of labels as integers. L1 and L2 differ only in the condition bits. This means that their distance is 8, which can be expressed by the equation

\[
L2 = L1 + 8,
\]

The same holds between L2 and L3, and L3 and L4.

4.2.3. \textbf{Increments.} If a label (L1) directly follows a label (L0), we obtain constraints like

\[
L1 = L0 + 1,
\]

4.2.4. \textbf{Return Addresses.} If a label (L3) is the return address of a microprogram subroutine starting at a label (L2), we get similar constraints like

\[
L3 = L2 + 1,
\]

4.2.5. \textbf{Fixed Values.} Some labels should have fixed addresses given by the user. They are simply assigned to their value, for example

\[
L = \$34,
\]

4.2.6. \textbf{Fixed Bits.} Some labels have fixed bits; for example, a label must be on an even address. This leads to constraints of the form

\[
\text{fixed(Label, 1, 0)},
\]

which was defined in Section 4.2.2.2. This constraint is solved immediately, reducing the domain of \textit{Label}.

4.2.7. \textbf{Alldistinct}. All labels must be assigned to different addresses to avoid collision of several program parts. This can be expressed by the built-in predicate

\[
\text{alldistinct}([L0, L1, \ldots])
\]
4.2.8. Sketch of the Program. The CHIP program to solve the problem consists simply of a statement of all constraints and a labeling procedure to assign values. The global program looks like

\[
\text{microcode}(\text{Labels}) \leftarrow \\
\text{setup\_domain}(\text{Labels}), \\
\text{multiway\_branches}(\text{Labels}), \\
\text{increments}(\text{Labels}), \\
\text{return\_addresses}(\text{Labels}), \\
\text{fixed\_values}(\text{Labels}), \\
\text{fixed\_bits}(\text{Labels}), \\
\text{alldistinct}(\text{Labels}), \\
\text{labeling}(\text{Labels}).
\]

For the labeling we use the deleteffc predicate, which chooses the variable with the smallest domain first. More intelligent labeling procedures can be used to implement other heuristics if needed.

4.3. Computation Results

The problem and the data were given by BULL Systèmes, France. They use a special-purpose FORTRAN program to solve this problem which implements its own search and backtracking procedure. This program took several months of development time.

Our program takes two pages of CHIP code (without data). A first version was written in one day.

The example uses 178 labels in the 256-address space. A solution was found in CHIP after 40 seconds with 7 backtrack steps. The FORTRAN program in comparison takes about 20 seconds to find a solution on a similar machine.

5. CONCLUSION

Many real-life problems are combinatorial problems. They can be viewed as search problems with constraints. Logic programming, as exemplified by PROLOG, provides a powerful language for a logical (declarative) formulation of combinatorial problems. Its nondeterministic computation liberates the user from the tree-search programming. However, due to the inefficiency of their search procedure based on the generate-and-test paradigm, these languages have not yet been used to solve large problems.

In this paper we have shown how the introduction of constraint propagation and consistency checking techniques inside logic programming leads to a very powerful problem-solving paradigm. Since this paradigm is embedded in a programming language, heuristics specific to particular problems can be added when necessary. This approach has been used to solve three real-life problems: a scheduling problem with disjunctive constraints, a graph-coloring problem, and a microcode label-assignment problem. The programs are written in CHIP, a logic-programming language based on the "active"-constraint concept. It has been shown that the
programs not only are fully declarative, very concise, and easy to write, but also run on average as efficiently as special-purpose programs developed in imperative languages.

CHIP shows that logic-programming systems can be extended by constraint-solving techniques and sophisticated search procedures, providing a powerful problem-solving tool for a wide range of application domains.

An interpreter of CHIP written in C is running on Sun-3, SPS9, and SPS7.

We would like to thank H. Gallaire, A. Herold, and A. Aggoun for many fruitful discussions, and J.C. Madre and A. Daviaud from BULL systèmes for giving us the data concerning the microcode generation problem.

REFERENCES