Solving Set Partitioning Problems with Constraint Programming

Tobias M"uller
Programming Systems Lab
Universit"at des Saarlandes
Postfach 15 11 50
66041 Saarbr"ucken, Germany
tmueller@ps.uni-sb.de

Abstract

This paper investigates the potential of constraint programming for solving set partitioning problems occurring in crew scheduling, where constraint programming is restricted to not employ external solvers, as for instance integer linear programming solvers. We evaluate preprocessing steps known from the OR literature on moderately sized set partitioning problems. Further, we propose a new preprocessing technique which allows to reduce problem size more effectively than standard preprocessing techniques but with similar computational effort. Additionally, we propose a propagation algorithm for a global set partitioning constraint which, compared with other constraint programming approaches, finds and proves optimal solutions significantly faster resp. produces better solutions in a given time period.

1 Introduction

Set partitioning problems occur as subproblems in various combinatorial optimization problems, as for example in airline scheduling (see [10] for details). A subtask of airline scheduling, called crew scheduling, takes as input data a set of crew pairings, where a crew pairing is a sequence of nonstop flights of a single crew and a pairing starts and ends at the same base station. Such a set of pairings is generated according to the flights an airline offers to its costumers where a large number of constraints have to be taken into account, as for instance labour contracts and union schedules. The selection of crew pairings which cause minimal costs and ensure that each flight is covered exactly once, can be
modelled as a set partitioning problem (SPP). Typically, SPPs are solved by integer linear programming (ILP) solvers.

This paper explores the potential of constraint programming (CP) [19, 18] for solving SPPs. Throughout this paper, we assume CP not to employ an ILP solver. Constraint programming offers the programmer a natural and compact way to model problems and adequate control and flexibility to solve problems quickly and efficiently even if additional side constraints have to be met or additional subproblems have to be solved. In our experiments we use a subset of SPPs taken from Hoffman & Padberg’s problem suite of air crew scheduling problems [9].

To our knowledge, the only work on solving SPPs with constraint programming without using an ILP solver has been done by Carmen Gervet. She proposed an SPP solver using set constraints and employing a demanding formal apparatus. Her solver operates on sets of sets which complicates the implementation [3, 5]. This work was the inspiration to develop a propagation algorithm based on index sets for a global set partitioning constraint (see Section 4).

An SPP can be stated as follows: for a given finite ground set $G$ (with cardinality $m$) and a set $P$ of $n$ subsets $X_j$ associated with costs $C_j$, find a minimum cost partition of $G$, i.e. a subset of $P$ where all elements are disjoint of each other and the union of the elements is $G$.

The common 0-1 linear integer programming model [13] is as follows:

$$\begin{align*}
\min & \quad cx \\
\text{s.t.} & \quad Ax = 1 \\
& \quad x_j \in \{0, 1\} \quad j = 1, \ldots, n.
\end{align*}$$

The 0-1 matrix $A$ has $m$ rows and $n$ columns. A row of $A$ corresponds to an element in the ground set $G$ and a column is the characteristic vector of a subset $X_j$. The vector $c$ holds the cost $C_j$ associated with a subset $X_j$ and the solution vector $x$ indicates whether a subset $X_j$ belongs to a partition or not. Such a system can be efficiently solved by an ILP solver.

The corresponding set-based model, which is somewhat closer to the actual problem formulation, can be phrased as:

$$\begin{align*}
\text{find a set } \text{Sol} \subseteq P \text{ s.t. } \min & \quad \Sigma_{j \in I} C_j \\
& \quad \bigcup_{j \in I} X_j = G \\
& \quad \forall i, j \in I, i \neq j : X_i \cap X_j = \emptyset \quad \text{where } I = \{i \mid X_i \in \text{Sol}\}.
\end{align*}$$

We have to find a set of subsets which produces minimal cost and is a partition of $G$.

The set-based model can be directly implemented using set constraints [4, 11] in conjunction with finite domain constraints [1]. But this turns out to be not powerful enough to solve larger instances of SPPs. To improve the situation, we explored two directions:

**Reducing the problem size of an SPP by performing preprocessing before solving it.**

We evaluated in detail different preprocessing steps from the OR literature [9] and propose a (to our knowledge) new preprocessing technique. This new technique allows for the considered problems a significantly improved problem size reduction...
with comparable computational effort with respect to standard preprocessing techniques (see Section 2).

**Improvement of the performance of the constraint solver.** We propose an algorithm for a global set partitioning constraint which allows to solve moderately sized SPPs within the constraint programming framework (see Section 4).

To do an experimental evaluation, the proposed algorithms and techniques have been implemented in Oz 3 [14, 17], a concurrent constraint language. For the experiments, we selected a subset of moderately sized problems of Hoffman&Padberg's problem suite (i.e. problems with up to 6774 pairings) since these problems could be practically handled.

The results obtained show that CP cannot compete with ILP solvers which are able to solve problems up to 1.000.000 pairings. But for CP, i.e. for Carmen Gervet’s SPP solver, we were able to reduce the time taken to solve problems (i.e., finding and proving an optimal solution) on average by a factor of 7 resp. to produce better solutions within a given time period if the optimal solution could not be found resp. proved. Therefore, we think to provide an alternative approach to solve SPPs, in particular if an SPP occurs as a part of a larger combinatorial problem preferably tackled with CP.

**Plan of the Paper.** The following section proceeds by giving an overview of preprocessing approaches, introducing a new preprocessing step and evaluating the effectiveness of preprocessing. Then, it defines a constraint model for solving SPPs and develops a propagation algorithm for the central constraint of the constraint model. In Section 5, the obtained solutions are discussed. Detailed experimental result are given in the Appendices A and B.

### 2 Preprocessing

Preprocessing aims at reducing the size of a SPP, i.e., at reducing the number of subsets and the cardinality of the ground set.

This section introduces a new preprocessing step to discard subsumed subsets in Section 2.2 and compares its effectiveness with standard preprocessing techniques in the OR literature [9]. Further, we provide for all considered preprocessing techniques intuitive algorithms in terms of sets. To obtain a maximal number of discard subsets resp. removed elements, the algorithms have to be applied iteratively until no further reduction of the problem size is possible, i.e., until a fixed point is reached.

**Prerequisites and Notation.** In the following, index sets will be used to model preprocessing approaches. An index set \( I_e \) contains all indices \( j \) of the subsets \( X_j \) where \( e \in X_j \). This corresponds to the column indices \( j \) of \( A \) where \( a_{ej} = 1 \). We mean by \( \#S \) the cardinality of \( S \).

Throughout the paper we will use the following terms synonymously: ‘number of subsets’ is synonymous to ‘number of columns’ and ‘cardinality of the ground set’ is synonymous to ‘number of rows’.
2.1 Detecting Multiple Sets (MS)

Find equal sets and keep the set with minimum cost. A naive algorithm comparing pairwise all subsets has complexity of $O(n^2)$. The use of appropriately sorted subsets can reduce the complexity to $O(n \log n)$. Despite the simplicity of this preprocessing approach, the number of subsets of a SPP can be already reduced quite significantly (see Section 2.5).

2.2 Detecting Subsumed Subsets (SS)

**Idea.** A subset $X_j$ can be discarded from an SPP if it can be partitioned by other subsets with less cost than $C_j$. Although the idea is very straightforward, to the knowledge of the authors it has not been published resp. used by now.

This preprocessing approach turns out to reduce the number of subsets better than any standard approach with similar computational effort.

**Algorithm.** An exhaustive search for subsumed subsets would incur an exponential complexity and is hence not tractable. Therefore, we propose a greedy heuristics to keep the computational effort low. Further, the proposed technique discards significantly more subsets than standard preprocessing approaches (see Section 2.5).

**Auxiliary functions:**

- `vect_head((x_1, \ldots, x_n)) \rightarrow x_1`
- `vect_tail((x_1, x_2, \ldots, x_n)) \rightarrow (x_2, \ldots, x_n)`

**Require:**

- $SCV = (SC_1 = (S_1, C_1), \ldots, SC_n = (S_n, C_n))$: vector of subsets and associated costs sorted such that costs are in descending order

```
1: function subsumedSubsets(in SCV : set of indices)
2: begin
3:   R \leftarrow \emptyset
4:   i \leftarrow 1
5:   while SCV \neq \emptyset do
6:     if checkForSubsumption(vect_head(SCV), vect_tail(SCV)) = true then
7:       R \leftarrow R \cup \{i\}
8:     endif
9:     i \leftarrow i + 1
10:    SCV \leftarrow vect_tail(SCV)
11:   endwhile
12:   return R
13: end.
```

Figure 1: Detecting subsumed subsets (driver loop).

The algorithm consists of two parts. The driver loop is shown in Figure 1. It is essential that $SCV$ is appropriately sorted, i.e. with descending costs. Otherwise, the probability to find partitions with less costs would decrease and the effectiveness of the algorithm would suffer. The algorithm starts with initializing the index counter $i$ and the accumulator for the return value (line 3–4). The while-loop invokes the function `checkForSubsumption()`, which checks if there exists for the head of $SCV$ a cheaper partition in the tail of $SCV$, for
all fields of SCV (line 6). In case a less costly partition can be found for a subset, the index of this subset is added to the accumulator R. Next, the index counter is incremented and the first field is removed from SCV. This algorithm has to be applied until the returned index set is empty to obtain the maximal effect.

The boolean function checkForSubsumption() is depicted in Figure 2. It tries recursively to prove that there is a partition for S in the remaining vector SCV. If it finds an empty vector it returns false (line 4–5). Otherwise it checks two cases: First, it checks if the head field of SCV, i.e. S_{head}, is subsumed by S and has less cost than the head element of SCV. If so, it enters recursion to prove the existence of a partition for reduced cost-annotated subset (S \ S_{head}, C - C_{head}) in the tail of SCV (line 8–10). Second, in case the head field of SCV is equal to S and less costly than C, checkForSubsumption() returns true to indicate that it found a valid partition. If both cases do not apply, it enters recursion to perhaps find a valid partition for (S, C) in the tail of SCV.

Require: SCV = (S_1, C_1),..., S_n = (S_n, C_n) vector of subsets and associated costs

1: function checkForSubsumption(in (S, C), in SCV) : {true, false}
2: begin
3: (S_{head}, C_{head}) ← vect_{head}(SCV)
4: if SCV = () then
5: return false
6: else
7: if S_{head} ⊆ S then
8: if C > C_{head} then
9: return checkForSubsumption((S \ S_{head}, C - C_{head}), vect_{tail}(SCV))
10: endif
11: elseif S_{head} = S then
12: if C ≥ C_{head} then
13: return true
14: endif
15: endif
16: return checkForSubsumption((S, C), vect_{tail}(SCV))
17: endif
18: end.

Figure 2: Detecting subsumed subsets (check for subsumption).

The complexity of the algorithm shown in Figure 2 is O(n), since with each iteration the vector of subsets (SCV) is shortened by one subset. The loop in the algorithm in Figure 1 invokes checkForSubsumption() for all possible vector tails, i.e. for n of them, that the complexity of subsumedSubsets() results in O(n^2). We observe that the typical number of iterations is compared to n very small, so that it can be neglected.

2.3 Clique Analysis (CA)

Idea. Suppose, we derive a graph from an SPP, such that the nodes of the graph correspond to subsets and if two nodes share at least a single element then there is an edge
between the nodes. A trivial clique $C_e$ in such a graph is the set of all nodes containing a certain element $e$ of the ground set. An important property of a clique is that only one member of it can be part of the solution and all others have to be discarded. If one could find a clique $C$ that properly subsumes $C_e$ then all nodes contained only in $C$ can be discarded, because they are ruled out by the node to be selected from $C_e$. Now the problem can be formulated as follows: find a set of subsets $E$ where all members $X_j$ exclude each other so that exactly one $X_j$ can be part of the partition sought. Further, let $K$ be the index set of $E$, i.e., $K = \{ J : X_j \in E \}$. In case there is an index set $M_e$ which is properly subsumed by $K$, i.e., $M_e \subset K$, then all sets indexed by $K \setminus M_e$ can be removed, since the presence of $X_j : j \in M_e$ excludes each of $X_i : i \in K \setminus M_e$ from the partition.

**Example.** Suppose, the ground set $\{1,2,3,4\}$ has to be partitioned by the sets $X_1 = \{1,2\}$, $X_2 = \{1,3\}$, $X_3 = \{2,3\}$, and $X_4 = \{2,4\}$. Since $X_1$ and $X_2$ contribute the element 1, one of the has to be part of the partition. But there is a clique $\{X_1,X_2,X_3\}$, because all of these subsets share pairwise an element. Subset $X_3$ can be discarded from the problem since either $X_1$ or $X_2$ will be part of a solution and both rule out $X_3$.

<table>
<thead>
<tr>
<th>Require:</th>
<th>$SV = \langle S_1, \ldots, S_n \rangle$ vector of subsets $G$ ground set</th>
</tr>
</thead>
</table>

1: function cliqueAnalysisStep(in G, in SV) : set of indices
2: begin
3: \( R \leftarrow \emptyset \)
4: \( \forall e \in G : I_e \leftarrow \{ i \mid e \in S_i \} \)
5: \( \text{forall } e \in G \text{ do} \)
6: \( \text{forall } i \in \{1, \ldots, n\} \setminus I_e \text{ do} \)
7: \( \text{if } \forall j \in I_e : S_i \cap S_j \neq \emptyset \text{ then} \)
8: \( R \leftarrow R \cup \{ i \} \)
9: \( \text{endif} \)
10: \( \text{endfor} \)
11: \( \text{endfor} \)
12: return $R$
13: end.

**Figure 3:** Clique analysis.

**Algorithm.** The algorithm is shown in Figure 3. It returns a set of indices for the subsets to be discarded. The algorithm starts with initializing the accumulator $R$ to be an empty set (line 3) and by generating the index sets for each element of the ground set $G$ (line 4). It then tries to find subsets which have a nonempty intersection with all subsets referred by a certain index set (line 5–7). If such a subset is found, its index is added to $R$ (line 8). The algorithm is applied on reduced problems until $R$ is found to be empty.

The complexity of the algorithm in Figure 3 is $O(\#G \times n^2)$, due to the nested loops (lines 5–6) which contribute $O(\#G \times n)$ and the test inside the loops (line 7) which contributes $O(n)$. 

6
2.4 Dominance Analysis (DA)

This preprocessing step aims mainly at reducing the cardinality of the ground set which leads in the ILP model (see Equation 1) to a reduction of the number of equations, i.e., a decrease of \( m \). There are cases that reduce also the number of subsets which is desirable for the constraint model.

**Idea.** The intention is to find an element \( k \) of the ground set which only occurs in those subsets in which also element \( l \) is contained. We say, \( k \) is dominated by \( l \) and can remove \( k \) from the ground set and all subsets, since all subsets that might be discarded due to the presence of \( k \) will be already be discarded due to \( l \). In terms of index sets, the idea can be rephrased as follows: an element \( k \) which occurs only in subsets where the element \( l \) is also contained, i.e. \( I_k \subseteq I_l \), allows to remove all subsets \( X_j \) where \( j \in I_l \setminus I_k \) and to remove the element \( k \) from all \( X_j \) and \( G \).

\[
I_k \subseteq I_l \rightarrow X_{j \in I_l \setminus I_k} \leftarrow \emptyset
\]

\[
\forall j \in \{1, \ldots, n\} : X_j \leftarrow X_j \setminus \{k\} \quad \bigwedge \quad G \leftarrow G \setminus \{k\}
\]

In case \( I_k = I_l \) either element \( k \) or \( l \) can be removed from the SPP, i.e. only reduction 4 has an effect.

Situations where \( I_k \) and \( I_l \) differ in exactly two elements \( i \) and \( j \), i.e. \( I_k = I \cup \{i\} \land I_l = I \cup \{j\} \) with \( I = (I_k \cap I_l) \neq \emptyset \), are exploited too. The subsets \( X_i \) and \( X_j \) either belong both to the partition or both do not. This can be easily seen, since subtracting row \( l \) from \( m \) in the ILP model results in \( x_l - x_m = 0 \rightarrow x_l = x_m \). There are two cases to consider:

1. In case \( X_l \cap X_m = \emptyset \), both subsets can be merged to a single subset \( X'_l = X_l \cup X_m \) and \( C'_l = C_l + C_m \).

2. In case \( X_l \cap X_m \neq \emptyset \), \( X_l \) and \( X_m \) can be removed from the SPP, since both exclude each other and therefore cannot be part of the partition at the same time.

**Algorithm.** The algorithm is depicted in Figure 4. It returns a 3-tuple consisting of a index set designating subsets to be removed, a set of elements to be removed (according to reduction 4), and a set of 2-tuples designating pairs of subsets to be merged. The algorithm starts with initializing the accumulators for the return value and the index sets (line 3–4). It proceeds by picking subsequently all elements \( i \) of the ground set and checking if there are index sets other than \( I_i \) which subsume \( I_i \) properly (line 7–9) resp. are properly subsumed by \( I_i \) (line 10–12). If so, the appropriate elements (according to reductions 3 and 4) are added to accumulators of the return value. Next, it is tested if \( I_i \) is equal to some other index set which would lead to mark element \( i \) to be removed (line 15–16). In case index sets are detected which differ in exactly two elements but have a nonempty intersection then they are either marked to be merged or to be discarded, depending on if the intersection of the subsets indexed by the differing elements is empty or nonempty, respectively (line 17-22). The algorithm is applied until all components of the return value are empty.

7
Require $SV = \{S_1, \ldots, S_n\}$ vector of subsets
$G$ ground set

1: function dominanceAnalysisStep(in $G$, in $SV$) : (set of indices, set of elements, merge set)
2: begin
3: $RI \leftarrow \emptyset$, $RE \leftarrow \emptyset$, $M \leftarrow \emptyset$
4: $\forall e \in G : I_e \leftarrow \{i \mid e \in S_i\}$
5: for all $i \in G$ do
6:     for all $j \in G \setminus \{i\} \land i < j$ do
7:         if $I_i \subseteq I_j$ then
8:             $RI \leftarrow RI \cup I_j \setminus I_i$, $RE \leftarrow RE \cup \{i\}$
9:         elseif $I_j \subseteq I_i$ then
10:            $RI \leftarrow RI \cup I_i \setminus I_j$, $RE \leftarrow RE \cup \{j\}$
11:         else
12:             if $I_i = I_j$ then
13:                 $RE \leftarrow RE \cup \{i\}$
14:             elseif $\exists e_1, e_2 \in G, e_1 \neq e_2 : I_i = I \cup \{e_1\} \land I_j = I \cup \{e_2\}$ where $I = I_i \cap I_j \land I \neq \emptyset$ then
15:                 if $I_i \cap I_j = \emptyset$ then
16:                     $M \leftarrow M \cup \{\{e_1, e_2\}\}$
17:                 else
18:                     $RI \leftarrow RI \cup \{e_1, e_2\}$
19:             end if
20:         end if
21:     end for
22: end for
23: return $(RI, RE, M)$
24: end

Figure 4: Dominance analysis.
The complexity of the algorithm in Figure 4 is $O(G^2 \times n)$, due to the nested loops (lines 5–6) which contribute $O(G^2)$ and the set operation inside the loops (lines 7 and 10) which contribute $O(n)$.

### 2.5 Evaluation

To run experiments, we implemented the presented preprocessing steps in Oz 3.0 providing sets of integers as built-in data types [11] which simplified the implementation significantly.

The experiments have been done on problems of Hoffman&Padberg’s problem suite [9]. The names of problems follow the naming convention taken over from Hoffman&Padberg’s benchmark suite ‘nwnb.rows.cols’, where nb is the problem number, rows is the number of rows, and cols is the number of columns of the problem.

We choose as a measure for the effectiveness of a preprocessing approach the percentage of discarded subsets. That means to take the number of columns discarded and to divide it by the number of columns before preprocessing. The tables in this section are condensed to give the reader a good overview. They show the arithmetic means of all individual results in the columns denoted with avr (for average). Further, we give also the minimal and maximal effectiveness in the columns min resp. max. The column of every table indicates the preprocessing steps performed. The detailed data can be found in Appendix A.

Table 1 shows the reductions obtained by applying a single preprocessing step to initial problems. The straightforward approach (MS) yields very good reduction due to the redundancies in the problems. The newly presented approach (SS) does even better by achieving an average effectiveness of 35%. Clique and dominance analysis result in very limited reductions.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Benefit</th>
<th>avr</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MS)</td>
<td>0%</td>
<td>21%</td>
<td>33%</td>
</tr>
<tr>
<td>(SS)</td>
<td>1%</td>
<td>35%</td>
<td>76%</td>
</tr>
<tr>
<td>(CA)</td>
<td>0%</td>
<td>0%</td>
<td>3%</td>
</tr>
<tr>
<td>(DA)</td>
<td>0%</td>
<td>0%</td>
<td>2%</td>
</tr>
</tbody>
</table>

Table 1: Effectiveness of single preprocessing approaches on initial problems.

Since (MS) is very straightforward but very effective, we were interested to find out how the approaches (DA), (CA), and (SS) behave on problems previously processed by (MS). Dominance analysis turned out to be very effective on problem ‘nw08.24.434’ by discarding 284 columns. But in fact, it was not as effective as (SS) which dropped 295 columns (see line 4 in Table 5). Clique analysis does not contribute significantly to problem reduction.

Table 3 shows the results obtained by applying dominance and clique analysis on problems preprocessed by (MS) and (SS). Again, only (DA) is able to contribute significant reduction for certain problems.
Approach Benefit relative to (MS) Benefit
min avr max min avr max
(MS DA) 0% 2% 58% 0% 23% 65%
(MS CA) 0% 0% 1% 1% 21% 33%
(MS SS) 1% 24% 70% 1% 40% 77%

Table 2: Effectiveness of preprocessing approaches on problems preprocessed by (MS).

Approach Benefit relative to (MS SS) Benefit
min avr max min avr max
(MS SS DA) 0% 2% 55% 1% 41% 86%
(MS SS DA CA) 0% 2% 55% 2% 41% 86%
(MS SS CA) 0% 0% 2% 2% 40% 77%
(MS SS CA DA) 0% 1% 19% 2% 41% 77%

Table 3: Comparing the effectiveness of different preprocessing approaches if multiple subsets are already discarded.

The results suggest as most effective combination of preprocessing steps (MS SS DA) which we used to eventually finding solutions (see Section 5). Clique analysis was not further regarded due to the insignificant problem reduction obtained.

In Section 2.4 it was explained that dominance analysis may lead to a reduction of rows (resp. a reduction of the cardinality of the ground set). In our experiments that happened only once for the data set ‘nw08.24.434’, where three rows could be discarded (see line 4 in Table 5). Due to it minor importance, we did not include figures for that kind of problem reduction in the tables above.

3 Solving SPPs with CP

This section introduces briefly concepts and notions of CP used in Oz [16] which are relevant to implement the set partitioning constraint in Section 4. Further, we propose a constraint model for SPPs.

Constraint Solving in Oz. Constraint solving consists basically of two components: constraint propagation and distribution. Constraint propagation is done over a computation space which consists of the constraint store and the associated propagators. The constraint store holds a conjunction of constraints like \( x \neq n \), \( x = y \), and \( x \in D \), where \( x \) and \( y \) are variables, \( n \) is a positive integer, and \( D \) is a finite subset of positive integers. For these constraints satisfiability and entailment can be efficiently decided. A variable is determined if it entails the constraint \( x = n \). More expressive constraints, as e.g. \( x < y \), are imposed by propagators. A propagator is a concurrent computational agent implementing
a constraint \( P \). It performs constraint propagation by removing elements from the domains of its parameters (e.g. \( x \) and \( y \) in \( x < y \)), i.e. it tells a constraint \( B \) to the constraint store \( C \) if \( C \land P \) entails \( B \), resulting in a store \( C \land B \). If a propagator becomes inconsistent with the constraint store is becomes failed which causes the whole computation space to be failed. A propagator that is entailed by the constraints in the store, i.e., cannot further amplify the constraint store, becomes entailed and is without any effect to further constraint propagation. If none of the previous cases applies a propagator is suspended until its parameters get involved in further constraint propagation. Usually, constraint propagation is not sufficient to find a solution. Therefore, a computation space is distributed by cloning it and adding a constraint \( D \) and \( \neg D \) to the original space and its clone, respectively. Finding an appropriate constraint \( D \) is subject to clever search heuristics.

### A Constraint Model for SPP

The basic idea is to introduce annotated subsets \( S_i \) which are a 3-tupels \( (X_i, C_i, R_i) \). The component \( X_i \) is the actual set, \( C_i \) the cost of the set, and \( R_i \in \{0, 1\} \) determines whether \( X_i \) is part of the partition or not (\( R_i = 1 \) resp. \( R_i = 0 \)). An \( X_i \) designates a set value, \( C_i \) is an integer, and \( R_i \) is a boolean finite domain variable.

\[
\min \sum_{i \in I} R_i \times C_i 
\]

\[
\text{s.t. } G = \bigcup_{i \in I} X_i \land \forall i, j \in I, i \neq j : X_i \cap X_j = \emptyset 
\]

where \( I = \{i \mid i \in \{1, \ldots, n\} \land R_i = 1\} \).

The constraint model consists mainly of two parts: first, the partitioning constraint (Equation 6) which will be discussed in Section 4 and second, the objective function (Equation 5) which uses sum constraints present in almost any available finite domain constraint solver. Search for an optimal solution can be done using branch&bound search, also provided by nearly all solvers (see for example [15]).

### 4 A Global Constraint for Solving SPPs

This section presents an efficient algorithm to implement a set partitioning constraint, as used in Equation 6 of the constraint model to solve SPPs in Section 3.

A first version of the set partitioning constraint was based on reasoning over lower and upper bounds of set interval constraints (see [4]) which turned out to be too inefficient. Inspired by tackling SPPs with an approach using so-called successor sets (see [4] for details), we developed a propagation algorithm based on index sets that requires only sets of integers rather than sets of sets.

#### 4.1 The Constraint in Terms of Index Sets

According to the constraint model, the set partitioning constraint reads as follows:

\[
\text{partition}(\bar{x}, g, \bar{r}) : \quad g = \bigcup_{i \in I} x_i \land \forall i, j \in I, i \neq j : x_i \cap x_j = \emptyset \\
\text{where } I = \{i \mid r_i = 1\} \text{ and } \\
\bar{x} = \langle x_1, \ldots, x_n \rangle \text{ and } \bar{r} = \langle r_1, \ldots, r_n \rangle.
\]
In terms of index sets \( \text{partition}(\mathcal{X}, g, \mathcal{T}) \) can be stated as follows:

\[
\text{partition}(\mathcal{X}, g, \mathcal{T}) : \quad \forall e \in g : I_e = \{i\} \land r_i = 1
\]

\[
\land \forall i \in \bigcup_{e \in g} I_e : x_i = \{j \mid i \in I_j\}.
\]

The equation 7 expresses the disjointness of all subsets being part of the partition (each element of \( g \) is contributed only by one subset) and the union of them yields the ground set (there must be for all elements at least one supporting subset). The connection between the index formulation and the “standard” formulation is established by Equation 7. It says, that if some subset \( x_i \) is part of a partition for all its elements the corresponding index sets must refer to \( x_i \), i.e. \( \forall e \in x_i : I_e = \{i\} \).

The following reduction rules describe operationally the propagation of the partitioning constraint in terms of index sets:

\[
\forall e \in g : I_e = \{i \mid e \in x_i\}
\]

\[
r_i = 0 \rightarrow \forall j \in x_i : I_j \leftarrow I_j \setminus i
\]

\[
r_i = 1 \rightarrow \forall j \in x_i : I_j \leftarrow \{i\}
\]

\[
I_j = \{i\} \rightarrow r_i \leftarrow 1
\]

\[
\forall i \in \{a \mid a \in \{1, \ldots, n\} \land \neg \exists e \in g : a \in I_e\} \rightarrow r_i \leftarrow 0
\]

\[
\exists e \in g : \#I_e = 0 \rightarrow \text{failed}
\]

\[
\forall e \in g : \#I_e = 1 \rightarrow \text{entailed}
\]

These rules are to be applied until a fixed point is reached. A certain instance of a rule can only be applied once. Note that a rule \( r_i = n \) has to read as “\( r_i \) has been determined to \( n \)”.

The first two rules project constraints on the boolean variable to the index sets. Rule 8 removes the index of \( b_i \) from all index sets where \( i \) occurred in. For the opposite case, Rule 9 sets all index sets containing \( i \) to contain only \( i \). The next two rules constrain boolean variables based on the current values of the index sets. Rule 10 constrains \( r_i \) to 1 if there exists a index sets containing only \( i \). Indices absent in all index sets cause the corresponding boolean variables to be set to 0, as done in Rule 11. The last two rules detect if the constraint is entailed or failed. That corresponds to the notions introduced in Section 3. As soon as either an empty index set (inconsistent with constraint store; Rule 12) or all index sets contain only one element (entailed by the store; Rule 13) propagation can be aborted. If none of the Rules 12 and 13 are applicable and a fixed point is reached, the constraint is suspended (which has to be signalled to the solver).

### 4.2 The Algorithm

The algorithm implements the collection of reduction rules presented in the previous section and is depicted in Figure 6. It uses the function \( \text{resetAllBut}(I, SI, k, G) \) which sets all
index sets containing \( k \) to contain only \( k \). It records in \( SI \) all index sets which have recently become a singleton set. The algorithm is given by Figure 5.

The local variable \( R \) is initialized to the union of all index sets in which \( k \) occurs, but \( k \) is not contained. Next, all index sets are processed; either an index set is set to contain only \( k \) (5–6) or by employing \( R \), all subsets to be discarded are removed from the index sets not containing \( k \) (line 8). In case this leads to an inconsistency, the value \( false \) is returned. If not, it is checked whether an index set has become a singleton which is recorded in \( SI \).

The algorithm for the set partitioning constraint (Figure 6) starts with initializing the index sets. The local variable \( SI \) holds index sets that have become singletons. Next, the index set are updated according to the current state of the constraint store (represented by the constraint on \( R_i \), lines 5–24). In lines 7–10 the case \( R_i = 1 \) is handled employing the function \( \text{resetAllBut}() \) (that corresponds to Rule 9). The opposite case, i.e. \( R_i = 0 \), is handled by removing \( i \) from all index sets (corresponding to Rule 8). It is further checked if an inconsistency occurred (lines 13–15) or singleton index sets were created (lines 16–21). Since function \( \text{resetAllBut}() \) records in its second argument newly created singleton index sets, it is looped until all singleton index sets are taken into account (lines 25–31). According to Rule 10, for all singleton index sets the corresponding boolean variables are determined to 1 (lines 32–36). Further, Rule 10 is implemented in lines 37–39. Last, it is check if the constraint is entailed by testing all index sets to be singletons (Rule 13; lines 41–42). Since inconsistencies are detected as they arise in the course of the algorithm, being not entailed requires to suspend propagation (line 43). The solver is informed about the outcome of propagation by the returned value which is either \( failed \), \( suspended \), or
entailed.

The complexity of the algorithm shown in Figure 5 is $O(\#G \times n)$, since it loops for all elements of the ground set $G$ and inside the loop is a set operation contributing $O(n)$ (line 8). The algorithm in Figure 6 invokes `resetAllBut()` for all subsets, i.e. $n$ times, and additionally maximal another $\#G$ times to take newly generated singletons into account (lines 25–31). Because, $\#G$ is typically small against $n$, the complexity of the whole propagation algorithm is $O(\#G \times n^2)$.

4.3 Implementation

The presented algorithm is implemented as an Oz propagator via the constraint propagator interface (CPI) of Oz [12] taking 590 lines of C++ code. The implementation takes advantage of a feature of the CPI that propagators maintain an internal state. That is used to keep track of intermediate results of propagation to avoid frequent rebuilding the index sets from scratch from the current constraint store. Instead, the index sets are incrementally updated on each invocation by only taking modified variables $r_i$ into account.

5 Experimental Results of Solving SPPs

We implemented the constraint model presented in Section 3 using Oz 3.0 featuring finite set and finite domain constraints [11, 7]. The set partitioning constraint (Equation 6 of the constraint model) was implemented by a propagator using the algorithm presented in Section 4. The objective function of the constraint model (Equation 5) was implemented using the generic sum constraint of the finite domain library of Oz 3.0. The search engine of Oz provides for `branch&bound` search which we used to find optimal solutions.

We experimented with various search heuristics and used problems preprocessed by (MS SS DA). It turned out that sorting the subsets of preprocessed problems by the minimal element of a subset as first criterion and by the cost per element as second criterion (in ascending order) yielded the best results. For all except three problems (‘nw19.40.2879’, ‘nw09.40.3103’, and ‘nw06.50.6774’), we are able to find the optimal solution in maximal about 10 minutes and for the majority of problems, even within a fraction of a minute. Proving optimality was possible in less than 10 minutes, except for problems ‘nw29.18.2540’ and ‘nw33.23.3068’ which took 43 minutes and 16 minutes, respectively. These results can be found in Table 6 in Appendix B.

Table 4 shows the best solutions founds within an one-hour time-limit. Table entries of the form `mins:secs.msecs` denote execution times where `mins` denotes minutes, `secs` denotes seconds, and `msec` denotes milliseconds. In table entries of the kind `c/r`, `c` denotes the number of columns and `r` denotes the number of rows. The last column of the table shows the distance to the optimal solution, i.e., the difference of the found solution to the optimal solution divided by the optimal solution. For problem ‘nw19.40.2879’ a solution 1% from the optimum was found after 16:29 minutes and for problem ‘nw09.40.3103’ a solution 3% from the optimum was found after 18:15 minutes (which could be slightly improved to 2% after 37:21 minutes). The result for problem ‘nw06.50.6774’ with a solution 19% from the optimal solution was unsatisfactory, so that we tried different heuristics. This
Figure 6: Propagation algorithm for the set partitioning constraint.
heuristics was already proposed in [6] and improved the best solution found to be 15% from the optimal solution. Interestingly, for the other problems it produced much worse results than the heuristics mentioned above.

We have solved the preprocessed problems in Table 6 (see Appendix B) using Carmen Gervet’s SPP solver on the same platform that we used for the benchmarks with our solver. Her solver is implemented in ECLP$^e$ [2] using the finite set constraint library Conjunto [3]. On these problems we obtained an average speed-up of factor 7 with our solver. Further, we run the three hardest problems with an one-hour time-limit. For problem ‘nw.40.2879’, Carmen Gervet’s solver found a solution with the same cost as we did. For the other two problems her solver did worse: after an hour the solution for the ‘nw06.50.6774’ problem has a cost of 27172 which is 248% of the optimal cost and for the ‘nw09.40.3103’ problem, the solution has a cost of 78644, i.e. 16% distance form the optimum. The results show that due to the improved performance of our solver, better solutions can be found in a given time period.

6 Related Work

In Section 1 we already mentioned the work of Carmen Gervet, who uses CP to tackle SPPs. There are other works on solving SPPs which take different approaches. We will shortly explain them and characterize their features.

Peter Szeredi implemented a solver for SPPs in Prolog which implements a model similar to the set-based model (Equation 2) in Section 1. Sets are represented as integers and search is done by backtracking.

Guerinik and Van Caneghem implemented a SPP solver in CHIP [1] which additionally employs a simplex-based solver. Bringing a simplex-based solver into play allows one to solve significantly larger instances of SPPs (up to 100.000 pairings). Further, byproducts of the simplex are be used to fix variables by employing reduced cost analysis. Additionally, variable selection can be based on reasoning over the optimal solution of the linear relaxation of the SPP.

Branch-and-Cut solver known from OR literature (e.g., see [9]) are able to solve problem instances up to 1.000.000 pairings. Such solvers also employ preprocessing to reduce the problem size.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Problem Size</th>
<th>Solutions found</th>
<th>Best found</th>
<th>Time to find Solution</th>
<th>Optimal Solution</th>
<th>Distance to Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>nw19.40.2879</td>
<td>1661/40</td>
<td>12</td>
<td>11060</td>
<td>16:29.750</td>
<td>10898</td>
<td>1%</td>
</tr>
<tr>
<td>nw09.40.3103</td>
<td>1427/40</td>
<td>11</td>
<td>69262</td>
<td>37:21.960</td>
<td>67760</td>
<td>2%</td>
</tr>
<tr>
<td>nw06.50.6774</td>
<td>5510/50</td>
<td>20</td>
<td>9322</td>
<td>52:11.830</td>
<td>7810</td>
<td>19%</td>
</tr>
</tbody>
</table>

Table 4: Best solutions found for the three hardest problems considered.
7 Conclusion and Future Work

We presented a new preprocessing approach that achieves a significantly improved problem size reduction on SPPs occurring in air crew scheduling compared to standard approaches with similar computational effort. Further, we present a propagation algorithm for a set partitioning constraint which allows to solve problems of moderate size without using an ILP solver. The average speed-up against previous CP solvers is 7.

Although, the performance of ILP solvers could not be reached, we are confident to offer an alternative approach to solve moderately sized SPPs, especially if SPPs occur as components of other problems preferably solved by CP.

Future work will concentrate on the development of more sophisticated search heuristics. Another promising direction is to use techniques that use intermediate solutions to reduce the problem size further.

Acknowledgements. I would like to thank Peter Szederi for providing his solver. Further, I am grateful to Carmen Gervet and Martin Müller for fruitful discussions. Martin Müller, Joachim P. Walser, and Jörg Würz also helped me by commenting on draft versions of the paper. Finally, I would like to thank the anonymous referees for their comments.

The research reported in this paper has been supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FKZ ITW 9601), the Esprit Working Group CCL-II (EP 22457), and the Deutsche Forschungsgemeinschaft through SFB378.

References


In table entries of the kind c/r, c denotes the number of columns and r denotes the number of rows. Further, MSS abbreviates MS SS.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Initial (MS)</th>
<th>(SS)</th>
<th>(DA)</th>
<th>(CA)</th>
<th>(MS DA)</th>
<th>(MS CA)</th>
<th>(MSS)</th>
<th>(MSS CA)</th>
<th>(MSS CA DA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nw41.17.197</td>
<td>197/17</td>
<td>177/17</td>
<td>105/17</td>
<td>197/17</td>
<td>197/17</td>
<td>177/17</td>
<td>103/17</td>
<td>103/17</td>
<td>103/17</td>
</tr>
<tr>
<td>nw32.19.294</td>
<td>294/19</td>
<td>252/19</td>
<td>191/19</td>
<td>294/19</td>
<td>294/19</td>
<td>252/19</td>
<td>185/19</td>
<td>185/19</td>
<td>185/19</td>
</tr>
<tr>
<td>nw40.19.404</td>
<td>404/19</td>
<td>336/19</td>
<td>303/19</td>
<td>404/19</td>
<td>404/19</td>
<td>336/19</td>
<td>284/19</td>
<td>284/19</td>
<td>284/19</td>
</tr>
<tr>
<td>nw08.24.434</td>
<td>434/24</td>
<td>356/24</td>
<td>148/24</td>
<td>432/21</td>
<td>434/24</td>
<td>150/21</td>
<td>139/24</td>
<td>62/21</td>
<td>139/24</td>
</tr>
<tr>
<td>nw19.31.467</td>
<td>467/31</td>
<td>465/31</td>
<td>460/31</td>
<td>467/31</td>
<td>464/31</td>
<td>465/31</td>
<td>460/31</td>
<td>458/31</td>
<td>458/31</td>
</tr>
<tr>
<td>nw20.22.685</td>
<td>685/22</td>
<td>566/22</td>
<td>500/22</td>
<td>685/22</td>
<td>685/22</td>
<td>566/22</td>
<td>468/22</td>
<td>468/22</td>
<td>468/22</td>
</tr>
<tr>
<td>nw23.19.711</td>
<td>711/19</td>
<td>474/19</td>
<td>484/19</td>
<td>711/19</td>
<td>711/19</td>
<td>474/19</td>
<td>392/19</td>
<td>392/19</td>
<td>392/19</td>
</tr>
<tr>
<td>nw37.19.770</td>
<td>770/19</td>
<td>639/19</td>
<td>503/19</td>
<td>770/19</td>
<td>770/19</td>
<td>639/19</td>
<td>469/19</td>
<td>469/19</td>
<td>469/19</td>
</tr>
<tr>
<td>nw34.20.899</td>
<td>899/20</td>
<td>750/20</td>
<td>672/20</td>
<td>899/20</td>
<td>899/20</td>
<td>750/20</td>
<td>749/20</td>
<td>749/20</td>
<td>749/20</td>
</tr>
<tr>
<td>nw43.18.1072</td>
<td>1072/18</td>
<td>983/18</td>
<td>828/18</td>
<td>1072/18</td>
<td>1072/18</td>
<td>983/18</td>
<td>793/18</td>
<td>793/18</td>
<td>793/18</td>
</tr>
<tr>
<td>nw42.23.1019</td>
<td>1019/23</td>
<td>895/23</td>
<td>789/23</td>
<td>1072/18</td>
<td>1072/18</td>
<td>895/23</td>
<td>749/23</td>
<td>749/23</td>
<td>749/23</td>
</tr>
<tr>
<td>nw28.18.1210</td>
<td>1210/18</td>
<td>825/18</td>
<td>927/18</td>
<td>1210/18</td>
<td>1210/18</td>
<td>825/18</td>
<td>787/18</td>
<td>787/18</td>
<td>787/18</td>
</tr>
<tr>
<td>nw25.20.1217</td>
<td>1217/20</td>
<td>844/20</td>
<td>564/20</td>
<td>1217/20</td>
<td>1217/20</td>
<td>844/20</td>
<td>508/20</td>
<td>508/20</td>
<td>508/20</td>
</tr>
<tr>
<td>nw38.23.1220</td>
<td>1220/23</td>
<td>911/23</td>
<td>1007/23</td>
<td>1217/20</td>
<td>1217/20</td>
<td>910/23</td>
<td>905/23</td>
<td>905/23</td>
<td>905/23</td>
</tr>
<tr>
<td>nw27.22.1355</td>
<td>1355/22</td>
<td>926/22</td>
<td>1027/22</td>
<td>1355/22</td>
<td>1353/22</td>
<td>926/22</td>
<td>829/22</td>
<td>829/22</td>
<td>829/22</td>
</tr>
<tr>
<td>nw24.19.1366</td>
<td>1366/19</td>
<td>926/19</td>
<td>559/19</td>
<td>1366/19</td>
<td>1366/19</td>
<td>926/19</td>
<td>490/19</td>
<td>490/19</td>
<td>490/19</td>
</tr>
<tr>
<td>nw35.23.1709</td>
<td>1709/23</td>
<td>1403/23</td>
<td>1279/23</td>
<td>1709/23</td>
<td>1709/23</td>
<td>1403/23</td>
<td>1204/23</td>
<td>1204/23</td>
<td>1204/23</td>
</tr>
<tr>
<td>nw36.20.1785</td>
<td>1785/20</td>
<td>1480/20</td>
<td>1422/20</td>
<td>1785/20</td>
<td>1785/20</td>
<td>1422/20</td>
<td>1350/20</td>
<td>1350/20</td>
<td>1350/20</td>
</tr>
<tr>
<td>nw29.18.2340</td>
<td>2340/18</td>
<td>2034/18</td>
<td>2004/18</td>
<td>2340/18</td>
<td>2340/18</td>
<td>2004/18</td>
<td>1836/18</td>
<td>1836/18</td>
<td>1836/18</td>
</tr>
<tr>
<td>nw30.26.2653</td>
<td>2653/26</td>
<td>1884/26</td>
<td>1427/26</td>
<td>2653/26</td>
<td>2653/26</td>
<td>1884/26</td>
<td>1270/26</td>
<td>1270/26</td>
<td>1270/26</td>
</tr>
<tr>
<td>nw31.26.2662</td>
<td>2662/26</td>
<td>183/26</td>
<td>1538/26</td>
<td>2662/26</td>
<td>2662/26</td>
<td>182/26</td>
<td>1398/26</td>
<td>1398/26</td>
<td>1398/26</td>
</tr>
<tr>
<td>nw19.40.2879</td>
<td>2879/40</td>
<td>2145/40</td>
<td>1833/40</td>
<td>2879/40</td>
<td>2879/40</td>
<td>2137/40</td>
<td>1669/40</td>
<td>1669/40</td>
<td>1669/40</td>
</tr>
<tr>
<td>nw33.23.3068</td>
<td>3068/23</td>
<td>2415/23</td>
<td>2389/23</td>
<td>3068/23</td>
<td>3068/23</td>
<td>2415/23</td>
<td>2112/23</td>
<td>2112/23</td>
<td>2112/23</td>
</tr>
<tr>
<td>nw09.40.3103</td>
<td>3103/40</td>
<td>2305/40</td>
<td>1569/40</td>
<td>3103/40</td>
<td>3103/40</td>
<td>2305/40</td>
<td>1429/40</td>
<td>1429/40</td>
<td>1429/40</td>
</tr>
<tr>
<td>nw06.50.6774</td>
<td>6774/50</td>
<td>597/50</td>
<td>5829/50</td>
<td>6774/50</td>
<td>6774/50</td>
<td>5964/50</td>
<td>5510/50</td>
<td>5510/50</td>
<td>5510/50</td>
</tr>
</tbody>
</table>

Table 5: Detailed results of preprocessing approaches.
B Benchmarks for Finding and Proving Optimal Solutions

In table entries of the kind $c/r$, $c$ denotes the number of columns and $r$ denotes the number of rows. Table entries of the form $\text{mins:secs.msecs}$ denote execution times where $\text{mins}$ denotes minutes, $\text{secs}$ denotes seconds, and $\text{msec}$ denotes milliseconds.

<table>
<thead>
<tr>
<th>Problems</th>
<th>Problem Size</th>
<th>Solutions found</th>
<th>Optimal Solution</th>
<th>Time to find Optimum</th>
<th>Time to find&amp;prove Optimal Solution</th>
<th>Time to prove Optimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>nw41.17.197</td>
<td>103/17</td>
<td>3</td>
<td>11307</td>
<td>00:00.450</td>
<td>00:00.780</td>
<td>00:00.330</td>
</tr>
<tr>
<td>nw32.19.294</td>
<td>185/19</td>
<td>8</td>
<td>14877</td>
<td>00:00.2110</td>
<td>00:00.2190</td>
<td>00:00.0108</td>
</tr>
<tr>
<td>nw40.19.404</td>
<td>284/19</td>
<td>9</td>
<td>10809</td>
<td>00:00.2820</td>
<td>00:21.620</td>
<td>00:18.800</td>
</tr>
<tr>
<td>nw48.24.434</td>
<td>62/21</td>
<td>3</td>
<td>35894</td>
<td>00:00.0050</td>
<td>00:00.140</td>
<td>00:00.090</td>
</tr>
<tr>
<td>nw15.31.467</td>
<td>460/31</td>
<td>3</td>
<td>67743</td>
<td>00:03.2500</td>
<td>01:03.520</td>
<td>00:28.470</td>
</tr>
<tr>
<td>nw21.25.577</td>
<td>273/25</td>
<td>3</td>
<td>7408</td>
<td>00:01.4700</td>
<td>00:03.990</td>
<td>00:02.520</td>
</tr>
<tr>
<td>nw22.23.619</td>
<td>389/23</td>
<td>8</td>
<td>6984</td>
<td>00:01.3800</td>
<td>00:08.050</td>
<td>00:06.670</td>
</tr>
<tr>
<td>nw12.27.626</td>
<td>255/27</td>
<td>35</td>
<td>14118</td>
<td>03:11.7900</td>
<td>04:40.640</td>
<td>01:28.850</td>
</tr>
<tr>
<td>nw39.25.677</td>
<td>372/25</td>
<td>12</td>
<td>10080</td>
<td>00:02.1100</td>
<td>00:19.380</td>
<td>00:17.250</td>
</tr>
<tr>
<td>nw20.22.685</td>
<td>468/22</td>
<td>13</td>
<td>16812</td>
<td>00:30.7900</td>
<td>01:52.870</td>
<td>01:21.880</td>
</tr>
<tr>
<td>nw23.19.711</td>
<td>392/19</td>
<td>4</td>
<td>12334</td>
<td>00:02.1500</td>
<td>00:20.100</td>
<td>00:17.950</td>
</tr>
<tr>
<td>nw37.19.770</td>
<td>469/19</td>
<td>10</td>
<td>10068</td>
<td>00:05.0500</td>
<td>00:19.030</td>
<td>00:13.980</td>
</tr>
<tr>
<td>nw26.23.771</td>
<td>427/23</td>
<td>7</td>
<td>6796</td>
<td>01:40.7200</td>
<td>01:44.480</td>
<td>00:03.760</td>
</tr>
<tr>
<td>nw10.24.853</td>
<td>195/24</td>
<td>10</td>
<td>68271</td>
<td>00:30.9700</td>
<td>00:35.870</td>
<td>00:34.900</td>
</tr>
<tr>
<td>nw34.20.899</td>
<td>624/20</td>
<td>4</td>
<td>10488</td>
<td>00:02.1100</td>
<td>00:43.190</td>
<td>00:41.080</td>
</tr>
<tr>
<td>nw43.18.1072</td>
<td>793/18</td>
<td>8</td>
<td>8904</td>
<td>05:26.9500</td>
<td>09:19.280</td>
<td>03:52.330</td>
</tr>
<tr>
<td>nw42.23.1079</td>
<td>749/23</td>
<td>1</td>
<td>7856</td>
<td>00:03.3500</td>
<td>01:40.890</td>
<td>01:40.540</td>
</tr>
<tr>
<td>nw28.18.1210</td>
<td>787/18</td>
<td>6</td>
<td>8298</td>
<td>00:03.7000</td>
<td>00:36.430</td>
<td>00:32.730</td>
</tr>
<tr>
<td>nw25.20.1217</td>
<td>508/20</td>
<td>10</td>
<td>5960</td>
<td>00:02.4300</td>
<td>00:15.870</td>
<td>00:13.440</td>
</tr>
<tr>
<td>nw35.23.1220</td>
<td>901/23</td>
<td>8</td>
<td>5558</td>
<td>01:05.7900</td>
<td>03:14.460</td>
<td>02:08.670</td>
</tr>
<tr>
<td>nw27.22.1355</td>
<td>829/22</td>
<td>20</td>
<td>9933</td>
<td>00:08.9000</td>
<td>00:50.450</td>
<td>00:21.350</td>
</tr>
<tr>
<td>nw24.19.1366</td>
<td>490/19</td>
<td>6</td>
<td>6314</td>
<td>00:01.2600</td>
<td>00:04.820</td>
<td>00:03.560</td>
</tr>
<tr>
<td>nw35.23.1709</td>
<td>1204/23</td>
<td>13</td>
<td>7216</td>
<td>01:14.5000</td>
<td>02:02.330</td>
<td>00:47.830</td>
</tr>
<tr>
<td>nw36.20.1783</td>
<td>1350/20</td>
<td>13</td>
<td>7314</td>
<td>01:34.9700</td>
<td>06:45.920</td>
<td>04:50.950</td>
</tr>
<tr>
<td>nw41.19.2340</td>
<td>1814/21</td>
<td>9</td>
<td>42750</td>
<td>07:10.3000</td>
<td>01:31.870</td>
<td>01:41.700</td>
</tr>
<tr>
<td>nw30.26.2655</td>
<td>1270/26</td>
<td>3</td>
<td>3842</td>
<td>00:00.3800</td>
<td>00:49.670</td>
<td>00:49.780</td>
</tr>
<tr>
<td>nw31.26.2662</td>
<td>1398/26</td>
<td>2</td>
<td>8038</td>
<td>00:08.1000</td>
<td>05:31.590</td>
<td>05:43.490</td>
</tr>
<tr>
<td>nw33.23.3068</td>
<td>2112/23</td>
<td>8</td>
<td>6678</td>
<td>10:12.4800</td>
<td>16:12.480</td>
<td>06:00.000</td>
</tr>
</tbody>
</table>

Table 6: Finding and proving optimal solutions.

The benchmarks were done on a 200MHz Dual-Pentium-Pro machine with 256KB 2nd-level cache and 256MB main memory running Linux 2.0.31.