An Improved Column Generation Algorithm for Crew Scheduling Problems

Shijun Chen, Yindong Shen

Image Processing and Intelligent Control Key Laboratory of Education Ministry
Department of Control Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China

Abstract

Column Generation (CG) technique is popularly applied in solving the crew scheduling problem of large size, which is generally modeled as an Integer Linear Programming (ILP) problem. The traditional CG algorithms for bus and rail crew scheduling encounter difficulties on either the generation of all potential shifts or solving subproblems. A subproblem embedded in the CG is generally represented as a Resource-constrained Shortest Path Problem (RCSPP), which is NP-hard and constitutes a major reason for the CG’s slow convergence. This paper presents a novel CG strategy to improve the efficiency of the proposed crew scheduling approach. The main idea is that a reasonably large set of “good” potential shifts (called a shift-pool) is pre-compiled using problem-specific knowledge. During the CG process, the RCSPP is not constructed to generate potential shifts as new columns until no columns with Negative Reduced Cost (NRC) exist in the shift-pool. Experiments are carried out on a series of Chinese real-world problem instances, and the computational results show that the solving process is significantly accelerated.

Keywords: Public Transport; Column Generation; Crew Scheduling

1 Introduction

Crew scheduling is one of the main problems arising in public transport operations and an efficient crew schedule can save an enormous amount of money for the enterprises. As a well-known NP-hard problem, crew scheduling has attracted much research interest since 1960’s. Up to now, lots of models and sophisticated solving methods have been proposed [1]. The Integer Linear Programming (ILP) based approach is one of the most popular approaches, in which the crew scheduling problem is modeled as a set covering or partitioning problem, and each variable corresponds to a precompiled potential shift. For large problems, the number of variables could be astronomical, which have to be limited by some heuristic methods, e.g. using soft parameters, in order to fit for the capacity of integer solution techniques such as Branch-and-Bound or Cutting...
Plane [2]. Optimality of the solution is therefore compromised. Since 1980s, some meta-heuristic approaches such as Genetic Algorithm [3], Tabu Search [4] etc. have been developed to solve large crew scheduling problems. They can often get a “good” or acceptable solution within short computational time, but hard to tell how far the solution is from the optimality.

In order to enlarge the problem size that the ILP solver can deal with, column generation technique is introduced [5, 6]. The basic idea is to decompose the original problem into a master problem and one or several subproblems, and then solve them in turn iteratively. At each iteration, dual multipliers obtained by solving the master problem are provided to the subproblems to generate new columns with Negative Reduced Cost (NRC). Although column generation technique has successful application to crew scheduling problems, it has the drawback of slow convergence, especially for problems of large size [7]. One main reason is that it is very time-consuming to solve iteratively the subproblem represented as a Resource-constrained Shortest Path Problem (RCSPP), which is NP-hard. Therefore, many research works have been focused on accelerating the solution process of RCSPP [8, 9, 10]. However, some methods proposed can be only used under some specific conditions, e.g. tight resource constraints required [9], so might not be suitable for a general case. Instead of using the RCSPP to solve subproblems, Fores et al. [11] proposed an alternative approach. The approach first creates a large set of valid shifts (called shift superset) according to labor agreement rules and some arbitrary soft rules. Then, the column generation technique is applied within a Branch-and-Bound framework to select a best set of shifts (i.e. the best solution) from the shift superset. At each iteration, new columns are generated by simply selecting the shifts with NRC, without the need of calling the RCSPP to create new shifts. Computational tests on large problem instances show that Fores’ methods outperform standard mathematical programming solver. However, the solution obtained cannot be guaranteed to be optimal for the original problem since a set of arbitrary soft rules have restricted some valid shifts to be created.

To overcome the drawbacks of the above-mentioned approaches, a novel column generation strategy is proposed. The main idea is to reduce the times of calling RCSPP to generate new shifts with NRC. A reasonably large set of “good” potential shifts (called a shift-pool) is pre-compiled using problem-specific knowledge. During the column generation process, the RCSPP is not constructed to generate potential shifts as new columns until no columns with Negative Reduced Cost (NRC) exist in the shift-pool. The resulting crew scheduling approach is expected to not only accelerate the solving process, but also guarantee the quality of solution. Experiments are to be carried out on real-world crew scheduling problems from Chinese public transport companies.

The structure of remaining sections in this paper is as follows. The next section describes the diver scheduling problem and gives the ILP formulation. Section 3 describes the improved column generation algorithm framework and the method to generate the shift-pool. In Section 4, the computational results are reported. The conclusion remarks are given at the end of the paper.

2 Problem Description

2.1 Crew Scheduling

Generally, vehicle scheduling is the precedent phase of crew scheduling in the operations planning
for public transport agency [12]. Therefore we assume that a sequence of vehicle blocks is determined a priori by solving the vehicle scheduling problem. In order to describe the crew scheduling problem, some definitions are given as follows.

A block represents a set of journeys to be operated consecutively by one vehicle during a day, beginning with a pull-out from, and ending with a pull-in, to a depot. Usually it contains a sequence of locations called relief points for crew’s relieving. The time when the relieving takes place is called relief time. The relief time and the relief point form a time/location pair, which is called relief opportunity (RO). The vehicle task between any two consecutive ROs is defined as a piece of work (piece for short). One or more consecutive pieces on a vehicle form a spell of work. A shift may contain several spells separated by one or more breaks. In addition, a shift generally starts working with a sign-on activity and finishes working with a sign-off activity. Fig. 1 displays an example of some blocks in a vehicle schedule, where each dot denotes a relief point coded by a single alphabet. The means of piece, spells and shifts are also illustrated in Fig. 1.

![Fig. 1: Vehicle work and crew shift](image)

Notice that a feasible shift must respect to a series of labor rules or regulations according to union agreement. The content of these regulations may vary in different operations enterprises and may vary according to the shift type. In our context, according to the practical situations from Chinese public transport operations, we consider three types of shift, which are straight shift, split shift and tripper shift. A straight shift consists of two or three spells, and there must be has at least one meal break between them. For split shift, there must be has a long rest break between two consecutive spells. Tripper shift only contains one spell. Eight labor rules are considered: (1) Minimum length of a spell; (2) Maximum length of a spell; (3) Maximum number of spells in a shift; (4) Maximum working time in a shift; (5) Maximum driving time in a shift; (6) Maximum spread-over in a shift; (7) Minimum efficient break time (not include walking time or mealtime) in a straight shift; (8) Minimum time difference between two spells in a split shift.

Crew scheduling consists of generating a set of shifts (i.e. crew schedule), such that:

- Each piece of work is assigned to a shift;
- All shifts must be subject to all labor rules and operational constraints;
- The total number of shifts or the total cost (payable work in minutes) is minimized.

### 2.2 Set Covering Formulation

In general, the crew scheduling problem can be formulated as set covering problem that relies on following notation. Let \( M = \{1, 2, \cdots, m\} \) be the set of pieces to cover and \( N = \{1, 2, \cdots, n\} \) be the set of all valid shifts. The cost of shift \( j \in N \) is denoted by \( d_j \) which can be calculated by some predefined computational methods. With each piece \( i \in M \) and each shift \( j \), we associate
a binary parameter $a_{ij}$ that take value 1 if shift $j$ covers piece $i$, and 0 otherwise. Finally, define a binary variable $x_j$ for each shift $j$ that indicates if shift $j$ is chose in the solution. Now, crew scheduling problem can be formulated as a set covering problem:

Minimize $Z = \sum_{j \in N} d_j x_j$ \hspace{1cm} (1)

subject to: $\sum_{j \in N} a_{ij} x_j \geq 1, \forall i \in M,$ \hspace{1cm} (2)\n
$x_j \in \{0, 1\}, \forall j \in N.$ \hspace{1cm} (3)

The objective function (1) minimizes the total cost of shift $j$. In our problem, minimizing the number of crews is more important than the total operational time. Hence, we calculate the $d_j$ as follows:

$$d_j = W + c_j$$ \hspace{1cm} (4)

where $c_j$ is the payable working time, $W$ is a constant number (larger than $c_j$) to ensure that minimizing the number of crews is the first objective. Constraints (2) ensure that each piece is covered at least once. Constraints (3) define the decision variables as binary.

3 Solution Approaches

3.1 Improved Column Generation Algorithm

When the constrain (3) in the set covering problem in Section 2.2 is relaxed, the formulation become a Linear Programming (LP) relaxation corresponding to the Master Problem (MP) in column generation context. The approach of column generation is to solve a Restricted Master Problem (RMP) and a Subproblem (SP) iteratively. The RMP contains only a subset of columns in MP and SP is an optimization problem to generate NRC columns. At each iterate, dual variables are firstly obtained by solving a RMP. Then, dual variables are delivered to the SP for generating NRC columns. If no NRC column exits, the solution of MP reaches to the optimality. Otherwise, columns with NRC are added to RMP and the above process is repeated.

In traditional column generation algorithms, the columns except those in the initial RMP, are normally generated during the solution process to the RCSPP by dynamic programming [13]. The RCSPP is usually solved iteratively with many (e.g. several hundreds of) times to reach the optimality, therefore, the whole algorithm is generally very time-consuming. To accelerate the algorithm, an improved column generation algorithm is developed. The basic idea is that a reasonably large set of “good” potential shifts (called a shift-pool) is pre-compiled using problem-specific knowledge; during the column generation process, the RCSPP is not constructed to generate potential shifts as new columns until no columns with Negative Reduced Cost (NRC) exist in the shift-pool. The detail of the improved algorithm is as follows.

In the $k$-th iteration of column generation, we denote by $\text{RMP}_k$ and $\text{SP}_k$ the $k$-th restricted master problem and sub-problem respectively. Correspondingly, the shift set, the coefficient matrix and objective coefficient vector of $\text{RMP}_k$ are denoted by $\bar{N}_k$, $\bar{A}_k$, $\bar{D}_k$ respectively. The
dual variable are denoted by $u^k_i (i = 1, 2, \cdots, m)$. For each shift $j$, its reduced cost is calculated as:

$$rc_j = d_j - \sum_{i \in M} a_{ij} u^k_i$$  \(5\)

Let $A$ be coefficient matrix of the set covering problem and $a_j$ be the $j$-th column in $A$, i.e. $A = (a_1, a_2, \cdots, a_n)$. The aim of subproblem (SP) is to identify the minimized NRC column:

$$a_p = \arg\min_{a_j \in A} \{rc_j | rc_j < 0, j \in N\}$$  \(6\)

Denotes by $PS$ the shift-pool and $P_k$ the set of NRC shifts selected from $PS$ or generated by RCSPP in the $k$-th iteration. The Improved Column Generation Algorithm (ICGA) is given in Algorithm 1.

**Algorithm 1** ICGA - Improved column generation algorithm

1: Create a shift-pool $PS$, $k \leftarrow 0$
2: Choose an initial set of shifts from $PS$ to form initial column set $\bar{A}_0$
3: Form an initial restricted master problem RMP$_0$ based on $\bar{A}_0$
4: repeat
5: Solve the RMP$_k$ to obtain dual variables $u^k_i (i = 1, 2, \cdots, m)$
6: $P_k \leftarrow \emptyset$
7: Calculate the reduced cost $rc_p$ for each shift $p \in PS$ and denotes by $PS_{NRC}$ the set of NRC shifts in $PS$
8: Sort all shift in $PS_{NRC}$ by non-decreasing order of their reduced cost
9: Select the first $r = \min\{|PS_{NRC}|, n_a\}$ shifts to add to $P_k$
10: if $P_k = \emptyset$ then
11: Formulate the SP$_k$ as a RCSPP, solve the RCSPP
12: Add the NRC columns (if any) obtained to $P_k$
13: end if
14: for all $a_p \in P_k$ do
15: $\bar{N}_k \leftarrow \bar{N}_k \cup \{p\}$, $\bar{A}_k \leftarrow [\bar{A}_k, a_p]$, $\bar{D}_k \leftarrow [\bar{D}_k, d_p]$
16: end for
17: $k \leftarrow k + 1$
18: until $P_k = \emptyset$

In Algorithm 1, the parameter $n_a$ is used to control the number of shifts selected from shift-pool; the RCSPP was solved by a multilabel based dynamic programming method proposed by Feillet et al. [8]; the RMP$_0$ was constructed by greedily selecting some shifts from the PS according to their efficiency ratio. For the shift $j$, its efficiency ratio $r_j$ is defined as:

$$r_j = w_j/d_j$$  \(7\)

where $w_j$ is the working time of shift $j$. For each piece $i$, from the shifts set covering the piece $i$, the two largest efficiency ratio shifts were selected to add to the RMP$_0$. In addition, how to generate efficient PS directly influences the performance of ICGA and it will be introduced in the next section.
3.2 Generating Shift-pool

The aim of forming shift-pool is such that the most of efficient shifts (e.g. in optimal solution) should be included in $P_S$. At the same time, we should not take too much time to form the $P_S$. Otherwise, it may be an extra load for overall algorithm and decrease the efficiency of ICGA. Based on problem-specific knowledge, the shifts with following characteristics are more efficient:

- contain longer spells to increase the efficient working time;
- with shorter break or join-up time to reduce the inefficient time;
- two-spell shifts, since two-spell shifts are most preferable;
- straight shifts, since straight shifts are most preferable;
- shifts with its working time bigger than a number determined by the transit operators.

According to the characteristics described above and relevant labor rules, we define some soft parameters to generate shift-pool with notations as follows:

1. $len_{Min}$: the minimal length of a spell, $len_{Min} \in [60, 240]$.
2. $break_{MStr}$: the maximum length of break time in a straight shift, $break_{MStr} \in [5, 150]$.
3. $work_{Min}$: the minimum length of working time, $work_{Min} \in [500, 800]$.

Notice that all the above intervals of the soft parameters are determined by the labor rules in advance. Before generating the shift-pool $P_S$, we generate all feasible spells by using an enumerative algorithm and sort all spells by its start time in non-decreasing order. Let $spell_S$ be the spell set containing all spells and $nS$ be the cardinality of $spell_S$. For the $i$-th spell $spell_i$, let $sT_i$, $eT_i$ be the start time and end time respectively. Denotes by $tw_{ij}$ the working time of the work joining $spell_i$ with $spell_j$. The pseudo-code for Generating the Shift-pool (GSP) is given in Algorithm 2.

**Algorithm 2** GSP - Generating the shift-pool

```
1: Input the spell set $spell_S$
2: for $i = 1$ to $nS - 1$ do
3:     if $eT_i - sT_i \geq len_{Min}$ then
4:         for $j = i + 1$ to $nS$ do
5:             if $eT_j - sT_j \geq len_{Min}$ then
6:                 $t_{ij} \leftarrow sT_j - eT_i$
7:                     if $0 < t_{ij} \leq break_{MStr}$ then
8:                         if $tw_{ij} \geq work_{Min}$ then
9:                             if $validStraight(i, j)$ then
10:                                Join $spell_i$ with $spell_j$ to form a straight shift and add it to $P_S$
11:                           end if
12:                        end if
13:                    end if
14:                end if
15:          end for
16:      end if
17:  end for
```

In Algorithm 2, the function $validStraight(i, j)$ checks the feasibility of combing $spell_i$ with
spell_j according to the labor rules associated to straight shift. All the parameters used will be determined by performing experimental tests. Finally, to guarantee all pieces are covered by some shifts, we generate all tripper shifts and add them to PS.

4 Computational Experiments

4.1 Test Instances

We use the real-world instances from the Chinese public transport companies to test the efficiency of the proposed algorithm. The instances contain the data set from the cities of Beijing, Huangshi and Shiyan etc. For each instance, Table 1 lists its number of trips, vehicle blocks, relief points and depots.

<table>
<thead>
<tr>
<th>Instance</th>
<th># trip</th>
<th># vehicle block</th>
<th># relief point</th>
<th># depot</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT08</td>
<td>107</td>
<td>9</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Hs9</td>
<td>298</td>
<td>44</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Bj41</td>
<td>410</td>
<td>54</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Syf2</td>
<td>701</td>
<td>55</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

In these instances, BT08 corresponds to the high-speed passenger rail-line between Beijing and Tianjin, the other instances correspond to urban public transit lines.

4.2 Parameters Setting

Before the implementation, it needs to set some parameters. First, W was set to 5000 to ensure that minimizing the number of crews is the first objective. For the other controlling parameters including n_a, len_Min and break_MStr, which were determined by experiments in the root node. By testing some preliminary experiments, we fixed the parameters as follows: n_a = 200, len_Min = 60, break_MStr = 150.

4.3 Results

The computational results of ICGA are to be reported in this section by comparison with those of the Traditional Column Generation Algorithm (TCGA). All procedures are implemented in C++ and the computation is carried out on a laptop with a Pentium Dual-Core T4300 2.1 GHz processor and 2 GB of RAM, using CPLEX 9.0 as underlying linear programming solver.

Notice that the main purpose of this paper is to improve the traditional column generation algorithm to solve the LP within the set covering problem, therefore, it would be adequate to compare only the computational results for the root node. In the TCGA, all trippers are included in the initial RMP. Table 2 gives the computational results of the ICGA in comparison with those of the TCGA, where iter denotes the number of times of calling RCSPP, nc denotes the number of columns generated by solving RCSPP, ts denotes the total computational time of solving RCSPP,
tsp is related only to the ICGA and denotes the sum of the time spending on the generation of the shift pool and the time spending on the generation of new columns from the shift pool, tm denotes the computational time of solving RMP, ttotal denotes the overall computational time. The times in Table 2 are all in seconds.

Table 2: Comparison between TCGA and ICGA

<table>
<thead>
<tr>
<th>Instance</th>
<th>TCGA</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>ICGA</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>iter</td>
<td>nc</td>
<td>ts</td>
<td>tm</td>
<td>ttotal</td>
<td>iter</td>
<td>nc</td>
<td>tsp</td>
<td>ts</td>
</tr>
<tr>
<td>BT08</td>
<td>38</td>
<td>844</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>12</td>
<td>130</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Bj41</td>
<td>140</td>
<td>5219</td>
<td>169</td>
<td>37</td>
<td>208</td>
<td>5</td>
<td>21</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Hs9</td>
<td>131</td>
<td>5253</td>
<td>195</td>
<td>25</td>
<td>221</td>
<td>21</td>
<td>578</td>
<td>17</td>
<td>32</td>
</tr>
<tr>
<td>Syf2</td>
<td>429</td>
<td>16498</td>
<td>3080</td>
<td>919</td>
<td>4020</td>
<td>7</td>
<td>28</td>
<td>260</td>
<td>44</td>
</tr>
</tbody>
</table>

From the left part of Table 2, we can see that it takes much more time to solve RCSPP than to solve RMP in TCGA and the time of solving RCSPP is the main portion in total computational time. Comparing with TCGA, ICGA significantly reduces the number of iterations of calling RCSPP, since most of columns in the optimal solution have been selected from the shift-pool generated a priori. Therefore, we can see that the total computational time of ICGA is much shorter than TCGA for all of the instances.

Since ICGA is obviously faster than TCGA, we use it to search for the best integer solution in a branch and bound framework. We use the RO branching rule [11] and employ the best-bound strategy of node selecting. Although the RO branching rule usually is successful in practice, there exists the possibility of being failure to branch some nodes. Such nodes, if exist, are to be discarded. Since it is still time-consuming to solve some large-size problems, we limit the computational time to 4 hours. In Table 3, we report the relevant results including: the number of nodes explored (nN), the number of nodes explored before obtaining the best solution (ib), the number of shift in the solution (ns), the number of straight shifts (nStr) and the number of split shifts (nSpl) in the solution, the total cost of the solution (cost), the lower bound in the root node (lb), the best integer solution obtained (up), the total computational time (cpu, in seconds) and the gap between the best solution and the lower bound in the root node (gap). Note that the gap is calculated as: (up − lb) * 100/ib.

Table 3: Integer solution

<table>
<thead>
<tr>
<th>Instance</th>
<th>nN</th>
<th>ib</th>
<th>ns</th>
<th>nStr</th>
<th>nSpl</th>
<th>cost</th>
<th>lb</th>
<th>up</th>
<th>gap</th>
<th>cpu</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT08</td>
<td>3</td>
<td>1</td>
<td>19</td>
<td>15</td>
<td>4</td>
<td>9607</td>
<td>104581</td>
<td>104607</td>
<td>0.002</td>
<td>5</td>
</tr>
<tr>
<td>Bj41</td>
<td>72</td>
<td>16</td>
<td>100</td>
<td>99</td>
<td>1</td>
<td>34900</td>
<td>534893</td>
<td>534900</td>
<td>0.002</td>
<td>734</td>
</tr>
<tr>
<td>Hs9</td>
<td>7</td>
<td>5</td>
<td>62</td>
<td>52</td>
<td>10</td>
<td>29410</td>
<td>337947</td>
<td>339410</td>
<td>0.433</td>
<td>190</td>
</tr>
<tr>
<td>Syf2</td>
<td>55</td>
<td>54</td>
<td>80</td>
<td>80</td>
<td>0</td>
<td>32166</td>
<td>432142</td>
<td>432166</td>
<td>0.006</td>
<td>14400</td>
</tr>
</tbody>
</table>

From the Table 3, we can see all the gaps are less than 1% and the solution obtained is optimal or near-optimal. Notice that for all of the instances, no un-preferable tripper shifts exist in the best solution.
5 Conclusions

This paper presents an improved column generation algorithm. The main strategy is that a “good” shift-pool is generated a priori according to some problem-specific knowledge. By simply selecting the NRC columns from the shift-pool to add to the RMP, it can significantly reduce the times of calling the RCSPP, hence save much of time. Experimental results show that the improved column generation algorithm outperform the traditional column generation algorithms.

Although the approach proposed in this paper has been successfully accelerated the traditional column generation, it mainly focuses on solving the LP embedded in the set covering problem. Future research will be devoted to designing more sophisticated upper-bound strategies and more efficient branching rules to speed up overall solution process.

References