An Efficient Local Search Heuristic with Row Weighting for the Unicost Set Covering Problem

Chao Gao[a], Xin Yao[a,b], Thomas Weise[a], Jinlong Li[a,*]

[a] *USTC-Birmingham Joint Research Institute in Intelligent Computation and Its Applications (UBRI), School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China.*
[b] *The Centre of Excellence for Research in Computational Intelligence and Applications (CERCIA), School of Computer Science, University of Birmingham, Edgbaston, Birmingham B15 2TT, U.K.*

Abstract

The Set Covering Problem (SCP) is $\mathcal{NP}$-hard. We propose a new Row Weighting Local Search (RWLS) algorithm for solving the unicost variant of the SCP, i.e., USCPs where the costs of all sets are identical. RWLS is a heuristic algorithm that has three major components united in its local search framework: **(1)** a weighting scheme, which updates the weights of uncovered elements to prevent convergence to local optima, **(2)** tabu strategies to avoid possible cycles during the search, and **(3)** a timestamp method to break ties when prioritizing sets. RWLS has been evaluated on a large number of problem instances from the OR-Library and compared with other approaches. It is able to find all the best known solutions (BKS) and improve 14 of them, although requiring a higher computational effort on several instances. RWLS is especially effective on the combinatorial OR-Library instances and can improve the best known solution to the hardest instance CYC11 considerably. RWLS is conceptually simple and has no instance-dependent parameters, which makes it a practical and easy-to-use USCP solver.

Keywords: Combinatorial Optimization; Unicost Set Covering Problem; Row Weighting Local Search

*Corresponding author. Tel: +8618019557504
Email addresses: chao.gao.ustc@gmail.com (Chao Gao), x.yao@cs.bham.ac.uk (Xin Yao), tweise@ustc.edu.cn (Thomas Weise), jlli@ustc.edu.cn (Jinlong Li)

1. Introduction

The Set Covering Problem (SCP) is a combinatorial optimization problem with many applications, ranging from crew scheduling in railways to job assignment in manufacturing and service location [2, 3]. It can be described as follows: Given a set of elements X, a set $S = \{s | s \subseteq X\}$ and $\bigcup_{s \in S} s = X$, where each subset in S is associated with a cost, and the goal is to find a set $F \subseteq S$ whose union is X (which contains all elements from X) at the minimal total cost. If all subsets in S have identical cost, the problem is referred to as the unicost set covering problem (USCP). Although being a special case of SCP, the unicost version is generally considered to be even harder to solve [4] and is the subject of this paper.

Formally, an SCP instance is usually defined as an $m \times n$ zero-one matrix $A = \{a_{ij}\}_{m \times n}$ where $a_{ij} = 1$ means column (set) j can cover row (element) i. The objective is to find a set of columns at the minimal cost to cover all rows. If the problem is a USCP, the objective is equivalent to finding the smallest set of columns that cover all rows. A candidate solution C can then be represented as a subset of $N = \{1, \ldots, n\}$. Such a solution is feasible if and only if $\sum_{j \in C} a_{ij} \geq 1, \ \forall i \in M = \{1, \ldots, m\}$.

In this paper, we propose a stochastic Row Weighting Local Search (RWLS) algorithm for solving USCPs. RWLS uses two search operators to perturb the candidate solution and combines three major existing strategies into its local search procedure:

(1) a weighting scheme, which updates the weights of the uncovered rows in order to escape from local optima,
(2) different tabu strategies, which prevent possible cycles during the search, and
(3) a timestamp method to break ties, which makes the sets that are not moved into or out of the candidate solution for a long time are more likely to be selected.

In our experimental studies, RWLS has improved 14 best known solutions in the literature for 87 USCP benchmark instances from the OR-Library [5] and shown excellent performance. It is especially effective on the problems where the number of rows (elements) is much larger than the number of columns (sets). However, RWLS is also effective in other cases, for example, for the seven railway crew scheduling instances, with up to millions of

columns and thousands of rows. Combining a problem size reduction technique from solving Lagrangian Relaxation, although RWLS obtains inferior results to a sophisticated algorithm [6] from the literature for this set of railway crew scheduling instances, it still outperforms CPLEX12.5[1] consistently and succeeds in finding good solutions to all seven instances while CPLEX12.5 failed on four larger instances. Overall, RWLS is simple, efficient, and only needs a single parameter to indicate the stopping criterion.

The rest of this paper is organized as follows. We first discuss related work in Section 2 and then give a detailed description of RWLS in Section 3. The experimental studies are presented in Section 4 and compared with several approaches from the related work. Conclusions and future work are finally given in Section 5.

2. Related Work

The SCP is NP-hard in the strong sense [7]. Many algorithms have been developed for solving the SCP. Exact approaches [8, 9, 10, 11] are mostly based on branch-and-bound or branch-and-cut. Caprara et al. [12] compared different exact algorithms and found that the best exact approach is CPLEX. However, although exact algorithms can guarantee the optimality of the found solutions, they usually require substantial computational efforts when facing large scale problems.

Therefore, large instances of SCP are typically tackled by heuristic algorithms. The simplest one for SCPs is the greedy algorithm [13]. Later, several randomized greedy algorithms [14, 15] are proposed. They usually produce better results than the deterministic greedy one. A variety of other heuristic algorithms have also been proposed, including some general meta-heuristics, such as Genetic Algorithms [16], Simulated Annealing [17] and Lagrangian Relaxation-based heuristics [18, 19, 20, 6], among which the heuristic methods by Ceria et al [19], Caprara et al. [20] and Yagiura et al. [6] are able to achieve excellent results on the very large-scale instances by exploiting their specific features. In particular, the 3-flip neighborhood local search (3FNLS) method by Yagirua et al. [6], which combines 3-flip local search, adaptive penalty weights and Lagrangian Relaxation, has the best performance on the very large-scale railway crew scheduling problems. For a good survey of relaxation-based heuristics for the SCP, see [21].

Lan et al. [22] noticed that the cost information plays an important role in determining the peformance of Genetic algorithm [16], Simulated Anneal-

[1]CPLEX is an optimization software package from IBM: `http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.5.1/maps/ic-homepage.html`

ing [17] and the Lagrangian-based heuristic [18], and did not recommend these algorithms for USCPs. They proposed the Meta-RaPS approach that works effectively for both unicost and non-unicost SCPs. Yelbay et al. [4] gave a detailed explanation of the usefulness and limitations of dual information from Lagrangian Relaxation or Linear Programming (LP) Relaxation, and pointed out that the unicost problems may be more challenging than the non-unicost problems.

There are heuristics dedicated to specifically solving USCPs. Grossman and Wool [23] compared nine heuristics, including several greedy variants and a neural network algorithm. In their report [23], the randomized greedy variant R-Gr has the best performance on a large set of instances from the OR-Library [5]. A newer GRASP algorithm incorporating a local improvement procedure from solving Satisfiability (SAT) has shown to be able to obtain better results than R-Gr [24].

The Electromagnetism Meta-heuristic (EM) proposed by Naji-Azimi et al. [25] creates an initial population by generating a pool of solutions, and then a fixed number of local search and movement iterations are applied based on the "electromagnetism" theory. In order to escape from local optima, mutation is also adopted. The computational results in [25] show that EM performs much better than GRASP, but in comparison with Meta-RaPS on the combinatorial problem set, for 3 instances the solution qualities obtained by EM are inferior.

Stochastic local search is a popular approach for solving hard combinatorial problems [26]. Musliu [27] proposed a local search algorithm for the USCP using a simple fitness function, which is the number of uncovered elements plus the cardinality of the candidate solution. New candidate solutions are created by adding or removing sets from the current one. To avoid cycles during the local improvement phase, a tabu mechanism is used. According to our investigation, Musliu's algorithm is able to find most of the best known solutions on 80 instances from the OR-Library [5] as unicost problems and 5 small instances from the Steiner triple systems [28].

Since in USCP instances all sets have the same cost, the task can be seen as minimizing the number of sets in a solution. RWLS utilizes the special property of USCP, and uses a general search framework to iteratively reduce the size of the candidate solution.

3. Row Weighting Local Search (RWLS)

3.1. Notations and Definitions

As mentioned above, the USCP can be presented as an $m \times n$ zero-one matrix. M and N indicates the set of rows (elements in X) and columns

(subset $s \in S$) , respectively. We define J_i as the set of columns that are able to cover row i, and I_j as the set of rows covered by column j:

$$J_i = \{j \in N | a_{ij} = 1\}, \quad i = 1, \ldots, m, \tag{1}$$
$$I_j = \{i \in M | a_{ij} = 1\}, \quad j = 1, \ldots, n. \tag{2}$$

A candidate solution C is a subset of columns: $C \subseteq N$. For all i in M, we say that row i is covered if and only if there exists a j in C that satisfies $i \in I_j$.

For all j in N, an attribute denoted $j.score$, which is later used to prioritize the columns for covering, is defined and calculated according to Equation (3).

$$j.score = \begin{cases} \sum\limits_{\substack{i \in I_j \\ \sigma(C,i)=0}} i.weight & \text{if } j \notin C, \\ -\sum\limits_{\substack{i \in I_j \\ \sigma(C,i)=1}} i.weight & \text{if } j \in C. \end{cases} \tag{3}$$

In Equation (3), $i.weight$ is the weight of row i and $\sigma(C, i) = |C \cap J_i|$ represents the number of columns in C covering row i. When a column $j \notin C$, $j.score$ is the sum of all the weights of uncovered rows that j is able to cover. If a column $j \in C$, $j.score$ is the negation of the sum of the weights of rows which are only covered by j in C. It can be seen from the definition of $j.score$ that if we move j into or out of C, the *score* of j should be negated. A row i with $\sigma(C, i) > 1$ has no contribution to the score value of the corresponding columns in J_i, since it has been covered more than once by the candidate solution C.

Each column has a timestamp associated with it, which gets updated whenever it is moved into or out of the candidate solution. It is used to break ties when two or more columns have the same *score*.

We also define a neighborhood relationship for columns as follows: For all j_1, j_2 in N, and $j_1 \neq j_2$, if $\exists i \in M, i \in I_{j_1} \cap I_{j_2}$, we call j_1 and j_2 *neighbors*. The notation $neighbor(j)$ contains all the neighbors of j, defined as

$$neighbor(j) = \{d \in N | d \neq j \wedge I_d \cap I_j \neq \emptyset\}, \quad j = 1, \ldots, n \tag{4}$$

Each column has a Boolean attribute, named $j.canAddToSolution$, which is used to implement one of the two tabu strategies in RWLS. A column j can only be added to C if $j.canAddToSolutin$ is true. The uncovered rows are maintained in a set named L in RWLS.

3.2. The RWLS Algorithm

RWLS is a USCP solver. It tries to find the smallest set of columns that covers all the rows in M. For this purpose, we adopt a two phase search procedure. In the first phase, an initial solution C is constructed in a greedy manner. Then, a local search improvement is conducted with the adaptive weighting scheme. The overall procedure of RWLS is described as Algorithm 1.

A preprocessing step is necessary when there are rows that are only covered by one column. Such columns must be selected into the candidate solution, and the rows they cover can be removed from the problem. We examine the number of columns covering each row, i.e., $|J_i|$ for each i, and for rows that are only covered by one single column, the corresponding column is selected into the solution and marked permanently not to be removed. This preprocessing step has a time complexity of at most $O(n')$, where n' is the number of ones in the matrix A.

Algorithm 1 The RWLS algorithm

```
1: function RWLS( )
2:     read problem instance
3:     set stopping criteria
4:     preprocessing if necessary
5:     INIT( )
6:     LOCALSEARCH( )
7: end function
```

3.2.1. Initialization

Algorithm 2 describes the initialization phase, which builds a set C, representing the initial solution. The set L is initialized as M, representing the set of uncovered rows. Every row is initialized to have a weight of 1 and each column j has a $j.canAddToSolution$ of *true*, a *timestamp* of 1 and a *score* computed according to Equation (3), i.e., the number $|I_j|$ of rows it can cover. C is then constructed greedily in a loop until L becomes empty. The obtained initial solution C is then used as the candidate solution in the subsequent local improvement phase.

$ADD(j)$ is a simple operator, in which the score of j is negated, and then the scores of $neighbor(j)$ are updated according to Equation (3). The $canAddSolution$s of $neighbor(j)$ are set to *true*. Note that when $ADD(j)$ is called, the removal of newly covered rows from L is conducted inside the operator.

Algorithm 2 Initialization

```
1: function INIT(C candidate solution)
2:     for j ∈ N do
3:         j.score ← 0
4:         j.timestamp ← 1
5:         j.canAddToSolution ← true
6:     end for
7:     L ← ∅
8:     for i ∈ M do
9:         i.weight ← 1
10:        add i into L
11:        for d ∈ J_i do
12:            d.score ← d.score + i.weight
13:        end for
14:    end for
15:    C ← ∅
16:    while L ≠ ∅ do
17:        j ← rand({d ∈ N\C ∧ d.score = max{d1.score|d1 ∈ N\C}})
18:        ADD(j)
19:    end while
20: end function
```

3.2.2. Local Search

Let the size of the initial solution C be $k = |C|$. If there is any better solution, it must have a size less than k. If we always maintain k as the size of the best solution we have encountered so far, then the local search improvement can also be regarded as to solve a series of new problems: given the original problem and an integer number k, find a $k-1$ size solution which is able to cover all the rows in M.

Therefore, we take the initial solution C as the candidate solution into the local search improvement phase defined as Algorithm 3. Here, the *REMOVE* function is first called, which removes a column from C. C then becomes a partial solution with $k-1$ columns. However C may have redundant columns, and if one of such columns is removed, we get an even better solution, then the stored best solution and the variable k will be updated. We continue to remove columns until C becomes a partial solution which cannot cover all rows in M. As a partial solution of size $k-1$ has been obtained, a pair of operations (*ADD* and *REMOVE*) are used to perturb C. The weighting scheme is also applied, which means that the weights of uncovered rows are increased. The weighting scheme improves the chance of uncovered rows of being covered in the following iterations.

As described in Algorithm 3, in each iteration, C becomes a partial solution of size $k-1$. The *REMOVE* operator deletes the columns with the highest negative *score* (the one closest to zero) in C. RWLS keeps track of two previously added columns in the tabu list, i.e., a FIFO queue of size two, to prevent them from being removed again immediately. We found that even with a tabu list length of one, good results can be achieved, but

Algorithm 3 Local search improvement

```
1: function LOCALSEARCH( )
2:     step ← 1
3:     while stop criteria not satisfied do
4:         while L = ∅ do
5:             update the best solution
6:             select j ∈ C with the greatest score
7:             REMOVE(j)
8:         end while
9:         select j ∈ C ∧ j ∉ tabu_list with greatest score and the oldest if there is a tie
10:        REMOVE(j)
11:        r ← rand(L)
12:        select d ∈ {d1 ∈ J_r|d1.canAddToSolution = true} with the greatest score and the oldest if
   there is a tie
13:        ADD(d)
14:        for i ∈ L do
15:            i.weight ← i.weight + 1
16:        end for
17:        put d to tabu_list
18:        j.timestamp ← step
19:        d.timestamp ← step
20:        step ← step + 1
21:    end while
22: end function
```

with length two the algorithm tends to proceed faster. After a column removal, a row is randomly selected from the uncovered row set L and the column with the highest *score* and $canAddToSolution = true$ is chosen to be added to C. The $REMOVE(j)$ operator is symmetric of $ADD(j)$, in which the scores associated with j and $neighbor(j)$ are updated according to Equation (3), and $j.canAddToSolution$ is set to *false*, whereas its neighbors' $canAddToSolution$ are updated to *true*.

The $canAddToSolution = true$ restriction in Line 11 is the second tabu strategy applied in RWLS. Generally, we do not want the column which has been removed from C to be added back again if none of its neighbors' states have changed since its removal. We set $j.canAddToSolution = false$ if j leaves C, which means j is not eligible to be added to C. If one of the states of j's neighbors changes (due to their removal or addition), $j.canAddToSolution$ is updated to *true*. To save computing time, we implement this strategy along with the operators ADD and $REMOVE$.

Finally, the timestamp used in Algorithm 3 makes sure that columns that have not been selected for a longer time are preferred; i.e., when two or more columns have the same score, we break ties by preferring the oldest one with the smallest timestamp.

The viability of Line 11 in Algorithm 3 is guaranteed by an observation, as below:

Lemma 3.1. $\forall i \in L$, $|\{j \in J_i | j.canAddToSolution = true\}| \geq 1$.

Proof: Before the proof, we reassert that after necessary preprocessing, the remaining rows are covered by two or more columns. Then we consider the following two circumstances.

(a) Initially, all the columns have $canAddToSolution = true$, and then an initial solution is constructed. At this time, no columns have left C and no column has a *false* value for *canAddToSolution*. Thus, the claim holds.

(b) During local search, when a column j leaves C, we set $j.canAddToSolution = false$, and $\forall j' \in neighbor(j)$, $j'.canAddToSolution = true$. Let's assume that the removal of j causes row $r \in I_j$ to be uncovered, because $J_r \cap neighbor(j) \neq \emptyset$. Then, there is at least one column in J_r whose $canAddToSolution = true$ and, thus, the claim holds.

3.3. The Row Weighting Scheme in RWLS

The row weighting scheme plays an important role in our algorithm. In RWLS, each row is associated with a weight, which is represented by a positive integer number. Initially, all the rows are given a weight of 1. During the local search improvement phase, whenever the candidate solution C becomes a partial solution in an iteration, the weight scheme is applied, which means the weights of uncovered rows are increased. In RWLS, the simplest additive increasing method is adopted, which means the weights are simply increased by 1.

Whenever a partial solution with $k-1$ columns has been obtained, RWLS repeatedly perturbs the candidate solution. Since the columns in C are changing dynamically, the uncovered rows in L also change accordingly. Because of the weight increasing scheme, the "hard to cover" rows, which have larger weights, may have a good chance to be covered in the following iterations. Both the perturbation and increasing weights help RWLS to escape from potential local optima.

3.4. Analysis of the ADD and REMOVE Operators

As shown in Algorithm 3, the operators *ADD* and *REMOVE* are crucial to RWLS. Therefore, it is necessary to precisely specify them, as shown in Algorithms 4 and 5.

When column j is added or removed, the scores of j and its neighbors are calculated according to Equation (3) and Equation (4), respectively. The *canAddToSolution* of $neighbor(j)$ are updated to *true*. Only when j is removed, $j.canAddToSolution$ is set to *false*. The time complexity of these two operators depends on the size of $neighbor(j)$. To further analyze it, we define variables p, q and t. For all d in $neighbor(j)$, let $\gamma(j, d)$ be the set of rows

Algorithm 4 Add a column into C

```
1: function ADD(j)
2:     add j to C
3:     j.score ← −j.score
4:     for d ∈ neighbor(j) do
5:         d.canAddToSolution ← true
6:         for r ∈ I_d ∩ I_j do
7:             if |J_r ∩ C| = 1 then
8:                 L ← L\{r}
9:                 d.score ← d.score − r.weight
10:            else if |J_r ∩ C| = 2 then
11:                if d ∈ C then
12:                    d.score ← d.score + r.weight
13:                end if
14:            end if
15:        end for
16:    end for
17: end function
```

Algorithm 5 Remove a column from C

```
1: function REMOVE(j)
2:     remove j from C
3:     j.score ← −j.score
4:     j.canAddToSolution ← false
5:     for d ∈ neighbor(j) do
6:         d.canAddToSolution ← true
7:         for r ∈ I_d ∩ I_j do
8:             if |J_r ∩ C| = 1 then
9:                 d.score ← d.score − r.weight
10:            else if |J_r ∩ C| = 0 then
11:                L ← L ∪ {r}
12:                d.score ← d.score + r.weight
13:            end if
14:        end for
15:    end for
16: end function
```

that they both can cover.

$$\gamma(j,d) = I_d \cap I_j \quad (5)$$
$$q = \max\{|\gamma(j,d)||d \in neighbor(j)\} \quad (6)$$

Therefore, if column j is added or removed, the time complexity of the two operators is $\mathcal{O}(|neighbor(j)| \times q)$. More generally, if we define

$$t = \max\{|J_r||r \in M\} \quad (7)$$
$$p = \max\{|I_j||j \in N\} \quad (8)$$

where $\forall j \in N, |neighbor(j)| \leq t$, and $\forall d, j \in N, |\gamma(j,d)| \leq p$, we can conclude that the time complexity of these two operators will not exceed $\mathcal{O}(tp)$. Thus, the two operators used to perturb C are efficient when the product of t and p is relatively small, which is often the case.

4. Computational Results

In order to demonstrate the effectiveness of RWLS, we evaluate it on a large number of instances from the OR-Library [5] as well as instances from the Steiner Triple Systems (STS) [28]. There are 87 SCP instances from the OR-Library, of which 70 are randomly generated, 7 are very large-scale instances arising from crew-scheduling at Italian railways. The remaining 10 are unicost instances from two combinatorial mathematical models. Similar to previous work on the USCP [23, 24, 25, 27], we convert the non-unicost instances into USCPs by ignoring the cost information.

4.1. Problem Instances

Table 1 contains the details of the 70 random instances, divided into 12 problem sets (from 4 to NRH) with the number of rows ranging from 50 to 1000 and the number of columns spanning from 500 to 10000. Each set of instances is generated according to a specific density, i.e., a percentage of non-zero entries in the (sparse) matrix A. Sets 4 to 6 are from [29], A to E are from [8], and NRE to NRH are from [18].

Table 1: Details of the random problem sets [8, 18, 29], the combinatorial problems [23] and the STS instances [28].

Set	m	n	Density(%)	Max number of 1s per row	Num of Instances
4	200	1000	2	36	10
5	200	2000	2	60	10
6	200	1000	5	71	5
A	300	3000	2	81	5
B	300	3000	5	192	5
C	400	4000	2	105	5
D	400	4000	5	244	5
E	50	500	20	124	5
NRE	500	5000	10	561	5
NRF	500	5000	20	1086	5
NRG	1000	10000	2	258	5
NRH	1000	10000	5	580	5
CYC06	240	192	2.1	4	1
CYC07	672	448	0.9	4	1
CYC08	1792	1024	0.4	4	1
CYC09	4608	2304	0.2	4	1
CYC10	11520	5120	0.08	4	1
CYC11	28160	11264	0.02	4	1
CLR10	511	210	12.3	126	1
CLR11	1023	330	12.4	210	1
CLR12	2047	495	12.5	330	1
CLR13	4095	715	12.5	495	1
STS243	9801	243	1.2	3	1
STS405	27270	405	0.7	3	1
STS729	88452	729	0.4	3	1
STS1215	245835	1215	0.2	3	1

+ We ignore the first four STS instances (STS27 to STS135) due to their very small problem scale.

As non-unicost SCPs, the random instances from 4 to 6, A to D, NRE and NRF, as well as instances NRG1 to NRG4, are relatively easy to solve and their optima are known. The mixed integer linear programming tool CPLEX can solve them in reasonable time [12]. However, no optima are known for the instances that are converted to USCPs. The instances from set E are randomly generated USCPs and their optima can be easily obtained by a greedy procedure [23].

Table 1 also contains the ten combinatorial problem instances (CYC and CLR). The only instance whose optimum is known is CYC06. One obvious feature of the CYC instances is that each row is exactly covered by 4 columns. Different from the random instances, m is always larger than n. For a detailed explanation of the CLR and CYC problems, see [23].

The STS instances are unicost problems with regular structures, such as $|J_i| = 3, \forall i \in M$ and $|I_{j_1} \cap I_{j_2}| = 1, \forall j_1 \neq j_2 \in N$. They are generally regarded as very difficult for the existing algorithms [6]. The perl script used to generate the STS instances can be found from the website [2]. In STS problems, the number m is also much larger than n.

For the 7 railway crew scheduling instances, because of their very large sizes, we will consider them separately in Section 4.4.

[2] `http://www.co.cm.is.nagoya-u.ac.jp/~yagiura/scp/stcp/`

4.2. Experimental Results

Our algorithm is programmed in `C`, compiled with `gcc` with `-O2` optimization, running on a machine with Intel(R) Core(TM) i5 650 3.20GHz CPU and 4 GB RAM under a 64-bit Linux system. The maximum number of search steps for the random instances is set to 3×10^7, and to 1×10^8 for the combinatorial and STS problems at first, to show the best solutions that RWLS is able to achieve, and then we give direct comparisons to the most effective heuristics found in the literature.

4.2.1. Comparison of Best Solutions Found by Different Algorithms

Tables 2 and 3 contain the best results found by GRASP [24], EM [25], the local search algorithm by Musliu [27] and RWLS on the random instances. For convenience, in the rest of this paper, we will refer to the local search algorithm proposed by Musliu [27] as Musliu. The best known solution (BKS) value for each instance is also included in the table and we highlight those improved by RWLS in starred boldface. For each instance, 10 runs are executed by RWLS with different random seeds, in addition to the *best* solution value, we also report the number of runs detecting the *best*, and the average *time* over these "successful" runs.

It can be observed from Table 2 that the best solution values of EM are generally better than those of GRASP. EM found the BKS of NRE1, which is 16, whereas Musliu's algorithm has achieved the remaining BKSs on these instances. However, RWLS is able to surpass Musliu, since it has discovered all the BKSs and even improved 12 of them.

Table 3 contains the best solution values found by GRASP [24], EM [25], Meta-RaPS [22], Musliu [27], 3FNLS and RWLS on combinatorial and STS instances. Since 3FNLS has not been tested on the CLR and CYC problems by its authors, the best results obtained by 3FNLS on these instances are unknown in the literature. The same is true for GRASP, EM and Meta-RaPS on the STS problems.

From Table 3, we find that the best solutions obtained by RWLS are better than those of the other four approaches. RWLS has improved 2 BKSs on these 10 combinatorial problems. In particular, the best solution value of CYC11 is improved from 4088 to 3968. The only instance on which RWLS did not achieve the BKS is CYC10, whose BKS is 1792 [30]. However, the *best* solution from RWLS on CYC10 is still better than those from GRASP, EM, Meta-RaPS and Musliu.

Combining the results in Tables 2 and 3, we can see that RWLS has improved 14 BKSs in total, at the expense of more computation time on some instances. Since Musliu's algorithm has the best overall performance among all other algorithms, we choose Musliu's algorithm for further comparison.

Since the results of the other approaches are from the literature, we list the computer configurations used by each algorithm as below:

- GRASP [24]: programmed in C, running on a Pentium 4 1800 MHz CPU machine with 512 MB RAM under Linux System. The stopping criterion is a predefined maximum number of iterations. The computation times in finding the best reported solutions were not given.
- EM [25]: programmed in C, running on an Intel Core Duo 1.7GHz CPU with 1 GB RAM. It stops when an indicated global time is reached.
- Meta-RaPS [22]: running on an Intel Pentium IV 1.7 GHz PC.
- Musliu [27]: written in C++, running on an Intel Pentium 4, 2.4GHz CPU with 512MB RAM machine. For each instance, 10 runs are performed, and the average *time* over the runs detecting the *best* is reported.
- 3FNLS [6]: implemented in C, running on a Sun Ultra 2 Model 2300 (two Ultra SPARC II 300 MHz processors with 1 GB RAM) workstation, 10 runs are performed for each instance using various time limits.

Table 2: The best solutions found by GRASP [24], EM [25], Musliu [27] and RWLS on the random USCP instances. The starred boldface indicates the improvement over the best known solution in the literature.

Inst.	BKS	GRASP	EM		Musliu		RWLS		
			best	time	best	time	best	#best	time
4.1	38	38	38	22.70	38	0.5	38	10	0.02
4.2	37	37	37	1.57	37	0.0	37	10	0.01
4.3	38	38	38	3.00	38	0.0	38	10	0.01
4.4	38	39	38	74.38	38	0.7	38	10	0.17
4.5	38	38	38	2.40	38	0.4	38	10	0.02
4.6	37	38	38	3.17	37	0.8	37	10	0.13
4.7	38	38	38	17.00	38	1.1	38	10	0.07
4.8	37	38	38	3.07	37	1.0	37	10	0.08
4.9	38	38	38	0.57	38	1.0	38	10	0.02
4.10	38	38	38	6.72	38	1.2	38	10	0.15
5.1	34	35	34	6.84	34	1.0	34	10	0.40
5.2	34	34	34	220.05	34	3.2	34	10	0.10
5.3	34	35	34	25.91	34	0.8	34	10	0.04
5.4	34	34	34	27.16	34	1.6	34	10	0.07
5.5	34	34	34	5.84	34	2.2	34	10	0.06
5.6	34	34	34	297.62	34	3.1	34	10	0.09
5.7	34	34	34	6.34	34	0.6	34	10	0.04
5.8	34	35	34	61.41	34	2.2	34	10	0.17
5.9	35	36	35	33.30	35	0.6	35	10	0.03
5.10	34	35	34	7.87	34	3.4	34	10	0.16
6.1	21	21	21	1.87	21	0.0	21	10	0.02
6.2	20	20	20	79.71	20	0.7	20	10	0.17
6.3	21	21	21	0.08	21	0.0	21	10	0.02
6.4	20	21	21	10.89	20	0.6	20	10	0.48
6.5	21	21	21	2.09	21	0.0	21	10	0.03
A.1	39	39	39	28.85	39	3.2	**38***	10	320.27
A.2	38	39	39	182.56	38	-	38	10	3.46
A.3	39	39	39	140.21	39	1.8	**38***	10	181.40
A.4	37	38	38	18.74	37	5.7	37	10	6.04
A.5	38	39	38	7.81	38	6.1	38	10	0.42
B.1	22	22	22	44.32	22	8.3	22	10	0.35
B.2	22	22	22	7.39	22	2.0	22	10	0.31
B.3	22	22	22	7.04	22	1.1	22	10	0.68
B.4	22	22	22	29.04	22	11.6	22	10	1.07
B.5	22	22	22	42.86	22	12.1	22	10	0.68
C.1	43	43	43	921.96	43	5.9	43	10	0.81
C.2	43	44	43	1023.79	43	9.5	43	10	1.14
C.3	43	44	43	918.46	43	10.2	43	10	0.73
C.4	43	44	43	28.49	43	11.6	43	10	0.82
C.5	43	44	43	1007.09	43	2.1	43	10	4.25
D.1	24	25	25	49.06	24	-	24	10	9.73
D.2	25	25	25	18.76	25	2.2	**24***	10	285.28
D.3	24	25	25	73.53	24	21.6	24	10	364.87
D.4	25	25	25	50.56	25	17.7	**24***	10	270.87
D.5	25	25	25	212.62	25	24.1	**24***	10	346.74
E.1	5	5	5	0.01	5	0.0	5	10	0.0
E.2	5	5	5	0.01	5	0.0	5	10	0.0
E.3	5	5	5	0.01	5	0.0	5	10	0.0
E.4	5	5	5	0.05	5	0.0	5	10	0.0
E.5	5	5	5	0.02	5	0.0	5	10	0.0
NRE1	16	17	16	1305.72	17	2.2	16	2	3441.29
NRE2	17	17	17	22.61	17	1.5	**16***	2	6437.21
NRE3	17	17	17	50.67	17	16.5	**16***	1	2935.55
NRE4	16	17	17	43.31	16	-	16	5	2925.08
NRE5	17	17	17	10.78	17	4.5	**16***	2	3576.99
NRF1	10	10	10	145.71	10	17.3	10	10	35.09
NRF2	10	10	10	1325.44	10	43.9	10	10	84.45
NRF3	10	10	10	1399.77	10	48.7	10	10	71.61
NRF4	10	10	10	120.20	10	17.9	10	10	36.48
NRF5	10	10	10	1651.14	10	29.4	10	10	47.25
NRG1	61	-	63	101.75	61	27.3	61	10	74.27
NRG2	62	-	63	127.87	62	29.8	**61***	10	124.62
NRG3	62	-	63	124.47	62	20.8	**61***	10	104.36
NRG4	62	-	63	117.15	62	41.8	**61***	10	139.69
NRG5	62	-	63	32.38	62	40.2	**61***	10	176.31
NRH1	34	-	34	755.72	34	8.7	34	10	280.76
NRH2	34	-	34	464.40	34	7.8	34	10	407.19
NRH3	34	-	34	1760.62	34	19.1	34	10	409.27
NRH4	34	-	34	227.73	34	26.1	34	10	533.78
NRH5	34	-	34	1912.47	34	50.3	34	10	495.10

+ RWLS uses a maximum search iteration of 3×10^7 as its stopping criterion.

+ The best solution values of A.2, D.1 and NRE4 of Musliu were found after parameter tuning, and the computation times were not reported.

+ The *time* of EM was the computation time of finding the *best*. The *time* of Musliu and RWLS was the average *time* over the successful runs that found the corresponding *best*.

Table 3: The best solutions found by GRASP [24], EM [25], Meta-RaPS [22], Musliu [27], 3FNLS [6] and RWLS on the combinatorial and STS instances [28]. The starred boldface indicates the newly discovered BKSs. "-" indicates that the algorithm has not been evaluated on this problem instance.

Inst.	BKS	GRASP	EM		Meta-RaPS		Musliu		3FNLS	RWLS		
			best	time	best	time	best	time		best	#best	time
CLR10	25	25	25	0.57	25	0.05	25	0.0	-	25	10	0.01
CLR11	23	23	23	15.53	23	3.03	23	0.0	-	23	10	0.08
CLR12	23	23	23	109.69	23	4.13	23	3.7	-	23	10	0.38
CLR13	23	23	23	3539.45	23	48.74	23	79	-	23	10	3.89
CYC06	60	60	60	0.08	60	0.0	60	0.0	-	60	10	0.00
CYC07	144	144	144	1.97	144	0.0	144	0.0	-	144	10	0.02
CYC08	342	348	344	303.40	344	38.91	342	11.1	-	342	10	0.30
CYC09	774	813	812	407.63	793	88.36	774	110.4	-	**772***	2	266.70
CYC10	1792	1916	1915	1892.06	1826	80.56	1820	488.9	-	1798	7	663.73
CYC11	4088	4268	4272	12922.03	4140	12656.75	4088	1497.8	-	**3968***	1	520.69
STS243	198	-	-	-	-	-	198	29.6	198	198	10	0.09
STS405	335	-	-	-	-	-	-	-	337	335	5	321.26
STS729	617	-	-	-	-	-	-	-	617	617	10	23.36
STS1215	-	-	-	-	-	-	-	-	-	1063	1	886.25

+ For 3FNLS, the *best* is obtained by setting the time limit of 1800 seconds for STS243, 3600 seconds for STS405 and STS729.
+ The *time* of Meta-RaPS and EM is the computation time of finding the corresponding *best*.
+ The *time* of Musliu and RWLS is the average time over the runs finding the corresponding *best*.
+ For RWLS, the stopping criterion is set as a maximum iteration number of 1×10^8.
+ The BKS 198 of STS243 has been proven to optimality in [31].
+ The BKS 335 of STS405 is recently found by Resende et al. [32] using a biased random-key genetic algorithm dedicated to the STS problems.
+The best known solution to STS1215 is unknown in the literature.

4.2.2. Further Comparison with Musliu's Algorithm and 3FNLS

The experimental results in the previous section have demonstrated the ability of RWLS in finding high quality solutions. It has found 14 new best known solutions among 80 benchmark problem instances in the OR-library. This section will examine the reason of RWLS's superior performance in finding high quality solutions. Is it because it used much longer computation time than other algorithms or is it because of the novel combination of different search operators and the adaptive weighting scheme? To answer such questions in detail, we will compare RWLS against Musliu's algorithm as well as 3FNLS [6]. We have seen from Tables 2 and 3 that Musliu's algorithm has the best solution values among other existing algorithms on the USCP instances from the OR-Library. The 3FNLS algorithm, on the other hand, represents one of the most effective algorithms that combine Lagrangian Relaxation and local search. It has been known to be effective on a variety of instances, achieved state-of-the-art results on the non-unicost, very large-scale railway crew scheduling instances. However, the effectiveness of 3FNLS has not been tested on the OR-Library instances as unicost problems.

In order to make direct comparison between these three algorithms, we asked Musliu for the executable code of his solver on Linux. For 3FNLS, we asked Yagiura to share with us their source code of 3FNLS, which is written in `C`. Similar to RWLS, we compile 3FNLS on our machine using `gcc`, with O2 option. Both Musliu, 3FNLS and RWLS are running on the same Intel Core i5 3.2 GHz CPU, 4GB RAM machine under 64bit Linux system. Due

to the randomness of the three algorithms, we follow the experiment setup as Musliu, perform 3FNLS and RWLS 10 independent trials on each instance with 10 consecutive integer numbers ranging from 11 to 20 as random seeds. The computational results are presented as the best solution value (*best*), average solution value (*avg*) among the 10 trials, the number of trials finding the *best*, as well as the average time (*time*) over those "successful" runs that delivered the *best*. We follow the suggestion of Musliu, set the tabu factor for the random problems (4 to NRH) to 0.05, and 0.15 for the combinatorial and STS problems, respectively. The stopping criteria of the three algorithms are set to the same time limits. Note that because of the different features each algorithm possesses, it is difficult to set time limits fair to each algorithm. However, the *times* reported in [27] provide a reference of the hardness of these USCP instances from OR-Library. Hence we set the time limits for each set of instances roughly according to [27].

Tables 4 and 5 contain the computational results of Musliu, 3FNLS and RWLS. As shown in Table 4, for random instances from sets 4 to 6 and sets A to E, when RWLS and 3FNLS both achieve the same best solutions, RWLS is more efficient computationally, because it consumes less average times to achieve the same *best* and smaller *averages*. Moreover, RWLS has obtained 9 better solutions than Musliu and 3FNLS among 50 instances. As for the larger instances from sets NRE to NRH, the overall solution values of RWLS are still better than those of Musliu and 3FNLS, both of the *best* and the *average* solution values, while the *time* reported by RWLS are generally much larger than those of Musliu's algorithm. Observed that the computational times of Musliu are always very small even given larger time limits, we suspect that Musliu's algorithm might be quickly stuck in some local optima after certain iterations.

Table 5 also contains the comparison between Musliu's algorithm, 3FNLS and RWLS on the ten combinatorial and 4 STS instances. We can see that RWLS outperforms Musliu's algorithm in terms of solution quality by finding the same or better solutions in all cases, although it tends to consume a little more computation time on some small size problems. RWLS also outperforms 3FNLS on all the combinatorial and STS problems both in terms of best solution values and computational times. It is interesting that 3FNLS's solution values are generally better than those of Musliu although 3FNLS's performance on USCPs are not previously investigated by its authors. However, when compared to RWLS, it is easy to see that our algorithm outperforms 3FNLS, for it can always obtain better solution values within shorter runtimes, especially on the larger instances with many more rows than columns, such as CYC11 and STS1215.

Combining the results of Tables 4 and 5, we can see that RWLS has

obtained 19 better solutions than Musliu and 3FNLS given the same amount of running time on the same machine.

Table 4: Computational results from Musliu, 3FNLS and RWLS on the random USCP instances, obtained by running on the same Intel Core 3.25 GHz CPU with 4GB RAM under Linux system.

Inst	Musliu				3FNLS				RWLS			
	best	avg	#best	time	best	avg	#best	time	best	avg	#best	time
4.1	38	38.1	9	0.0	38	38.2	8	2.79	38	38.0	10	0.02
4.2	37	37.0	10	0.0	37	37.0	10	0.29	37	37.0	10	0.01
4.3	38	38.0	10	0.0	38	38.0	10	0.22	38	38.0	10	0.01
4.4	38	38.9	3	0.0	38	38.9	1	5.19	38	38.0	10	0.17
4.5	38	38.1	9	0.0	38	38.0	10	1.96	38	38.0	10	0.02
4.6	37	37.4	6	0.0	37	37.6	4	5.58	37	37.0	10	0.13
4.7	38	38.5	5	0.1	38	38.2	8	3.88	38	38.0	10	0.07
4.8	37	37.9	1	0.0	37	37.7	3	5.68	37	37.0	10	0.08
4.9	38	38.2	8	0.0	38	38.0	10	2.91	38	38.0	10	0.02
4.10	38	38.4	7	0.0	38	38.9	1	1.14	38	38.0	10	0.15
5.1	35	35.1	9	0.1	35	35.0	10	0.25	**34**	34.0	10	0.40
5.2	35	35.1	9	0.0	34	34.5	5	4.52	34	34.0	10	0.10
5.3	35	35.3	7	0.0	34	34.0	10	1.20	34	34.0	10	0.04
5.4	35	35.1	9	0.0	34	34.4	6	3.86	34	34.0	10	0.07
5.5	35	35.1	9	0.0	34	34.0	10	2.57	34	34.0	10	0.06
5.6	34	34.3	6	0.5	34	34.3	7	5.62	34	34.0	10	0.09
5.7	34	34.9	3	0.0	34	34.0	10	1.64	34	34.0	10	0.04
5.8	35	35.4	7	0.2	34	34.7	3	1.77	34	34.0	10	0.17
5.9	35	36.7	1	0.0	35	35.0	10	1.18	35	35.0	10	0.03
5.10	35	35.6	5	0.0	34	34.9	1	6.24	34	34.0	10	0.16
6.1	21	21.2	8	0.0	21	21.0	10	0.32	21	21.0	10	0.02
6.2	20	20.7	3	0.1	20	20.7	3	7.35	20	20.0	10	0.17
6.3	21	21.1	9	0.1	21	21.0	10	0.59	21	21.0	10	0.02
6.4	21	21.1	9	0.0	21	21.0	10	1.34	**20**	20.0	10	0.48
6.5	21	21.0	10	0.0	21	21.0	10	1.36	21	21.0	10	0.03
A.1	39	39.0	10	0.0	39	39.0	10	5.56	**38**	38.9	1	10.03
A.2	39	39.1	9	0.4	39	39.0	10	3.40	**38**	38.0	10	3.46
A.3	39	39.0	10	0.0	39	39.0	10	3.01	**38**	38.7	3	14.10
A.4	37	37.9	1	1.0	38	38.0	10	4.42	37	37.1	9	4.39
A.5	38	38.9	1	0.4	38	38.9	1	17.84	38	38.0	10	0.42
B.1	22	22.9	2	0.2	22	22.4	6	10.49	22	22.0	10	0.35
B.2	22	22.0	10	0.0	22	22.0	10	6.09	22	22.0	10	0.31
B.3	22	22.0	10	0.0	22	22.3	7	12.33	22	22.0	10	0.68
B.4	23	23.0	10	0.0	23	23.0	10	1.73	22	22.0	10	1.07
B.5	22	22.5	5	0.6	22	22.1	9	6.37	22	22.0	10	0.68
C.1	43	43.8	2	1.1	43	43.4	6	8.66	43	43.0	10	0.81
C.2	43	43.9	2	0.5	43	43.9	1	17.63	43	43.0	10	1.14
C.3	43	43.5	5	1.0	43	43.8	2	13.39	43	43.0	10	0.73
C.4	44	43.9	1	0.7	43	43.8	2	9.65	43	43.0	10	0.82
C.5	43	43.7	3	0.6	43	43.9	1	5.75	43	43.0	10	4.25
D.1	25	25.8	2	0.0	25	25.1	9	12.37	**24**	24.1	9	7.27
D.2	25	25.3	7	0.9	25	25.0	10	4.86	**24**	24.9	1	8.54
D.3	25	25.3	7	0.3	25	25.4	6	14.32	**24**	24.9	1	5.60
D.4	25	25.6	4	0.5	26	26.0	10	0.00	25	25.0	10	1.69
D.5	25	25.4	6	1.0	26	26.0	4	14.42	**24**	24.9	1	18.27
E.1	5	5.0	10	0.0	5	5.0	10	0.0	5	5.0	10	0.0
E.2	5	5.0	10	0.0	5	5.0	10	0.0	5	5.0	10	0.0
E.3	5	5.0	10	0.0	5	5.0	10	0.0	5	5.0	10	0.0
E.4	5	5.0	10	0.0	5	5.0	10	0.0	5	5.0	10	0.0
E.5	5	5.0	10	0.0	5	5.0	10	0.0	5	5.0	10	0.0

+ Time limits for instance set 4 – 6 and A – E are set to 10 seconds and 20 seconds, respectively.
+ For each instance, the results are reported as the best solution (*best*), the average solution (*avg*) from the 10 runs, the number of runs (#*best*) that the *best* is found as well as the average time (*time*) over those runs detecting the *best*.
+ We emphasize our better solutions than Musliu and 3FNLS with boldface.

Table 5: Computational results from Musliu, 3FNLS and RWLS on the random NRE to NRH, combinatorial and STS problems, obtained by running on the same Intel Core 3.25 GHz CPU with 4GB RAM under Linux system.

Inst	Musliu				3FNLS				RWLS			
	best	avg	#best	time	best	avg	#best	time	best	avg	#best	time
NRE1	17	17.2	8	2.4	17	17.3	7	62.16	17	17.0	10	8.82
NRE2	17	17.2	8	0.8	17	17.2	8	38.80	17	17.0	10	3.64
NRE3	17	17.2	8	0.0	17	17.0	10	51.22	17	17.0	10	3.05
NRE4	17	17.1	9	0.1	17	17.4	6	54.98	17	17.0	10	3.64
NRE5	17	17.4	6	0.7	17	17.2	8	67.59	17	17.0	10	10.76
NRF1	10	10.8	2	1.2	11	11.0	10	0.00	10	10.1	9	25.31
NRF2	10	10.8	2	3.2	10	10.9	1	89.45	10	10.2	8	55.24
NRF3	10	10.6	4	3.2	10	10.9	1	24.44	10	10.1	9	28.34
NRF4	10	10.9	1	3.8	11	11.0	10	0.00	10	10.0	10	36.29
NRF5	10	10.8	2	2.9	11	11.0	10	0.00	10	10.1	9	34.91
NRG1	62	62.6	4	3.2	63	63.3	7	63.81	**61**	61.3	7	54.05
NRG2	62	62.2	8	3.4	62	63.0	1	88.96	**61**	61.5	5	69.76
NRG3	63	63.0	10	2.7	63	63.4	6	55.69	**61**	61.7	3	82.09
NRG4	63	63.2	8	2.6	63	63.3	7	50.41	**61**	61.9	1	86.72
NRG5	62	63.1	1	3.0	63	63.4	6	30.09	**61**	61.9	1	88.44
NRH1	34	34.8	2	7.4	35	35.3	7	53.36	34	34.9	1	53.22
NRH2	35	35.0	10	1.5	35	35.2	8	62.59	35	35.0	10	15.39
NRH3	35	35.0	10	0.5	34	35.2	1	59.35	34	34.9	1	97.02
NRH4	34	34.9	1	6.9	35	35.3	7	75.01	34	34.8	2	97.46
NRH5	34	34.9	1	4.3	35	35.1	9	50.73	34	34.9	1	25.56
CLR10	25	25.1	9	0.0	25	25.0	10	2.75	25	25.0	10	0.01
CLR11	23	23.0	10	0.0	23	23.1	9	11.69	23	23.0	10	0.08
CLR12	23	23.0	10	0.5	23	25.1	1	13.87	23	23.0	10	0.38
CLR13	23	24.1	5	5.6	29	29.7	5	16.71	23	23.0	10	3.89
CYC06	60	60.0	10	0.0	60	60.0	10	0.00	60	60.0	10	0.00
CYC07	144	144	10	0.0	144	144.0	10	0.00	144	144.0	10	0.02
CYC08	349	349.9	6	1.8	342	343.8	1	46.60	342	342.0	10	0.30
CYC09	809	813.0	3	15.6	780	780.1	9	104.39	**772**	773.6	2	266.70
CYC10	1894	1909.9	1	42.7	1801	1807.2	1	819.80	**1798**	1798.6	7	663.73
CYC11	4270	4271.1	1	0.9	4103	4144.9	3	392.36	**3968**	4021.1	1	520.69
STS243	198	201.7	2	1.6	198	198.0	10	170.50	198	198.0	10	0.09
STS405	343	345.0	4	7.6	336	336.0	10	151.07	**335**	335.7	3	117.81
STS729	649	649.8	4	87.9	617	630.3	3	831.78	617	617.0	10	23.36
STS1215	1119	1119.0	10	0.1	1071	1076.3	1	1659.63	**1063**	1065.9	1	886.25

+ Time limit for instance set NRE – NRH is set to 100 seconds.
+ Time limit for CLR10 – CLR13, CYC06 – CYC08 and STS243 is set to 20 seconds.
+ Time limit for CYC09 – CYC11 and STS243 – STS729 instances is set to 1000 seconds.
+ Time limit for STS1215 is set to 2000 seconds due to its larger size.
+ For each instance, the results are reported as the best solution (*best*), the average solution (*avg*) from the 10 runs, the number of runs (*#best*) that the *best* is found as well as the average time (*time*) over these runs detecting the *best*.
+ We emphasize our better solutions than Musliu and 3FNLS with boldface.

In summary, RWLS can achieve better or the same solution quality within the same time limits as Musliu and 3FNLS on the 70 random USCP instances, 10 combinatorial and 4 STS instances, which demonstrates the advantages of RWLS in solving USCP. In order to gain a deeper understanding of what caused the good performance of RWLS on USCP instances, several main differences and similarities between Musliu, 3FNLS and RWLS are worth noting.

First, we would like to discuss the fitness functions of the three algorithms. For Musliu and 3FNLS, they both define a penalty function as their fitness functions. Specifically, Musliu's fitness function is defined as the number of uncovered rows plus the cardinality of the candidate solution. When solving USCPs, the penalty function of 3FNLS can be seen as the sum of the penalty weights of rows plus the cardinality of the current candidate solution. The

main difference turns out to be that 3FNLS assigns penalty weights to rows, and during its local search, it adaptively adjusts the weights whenever the search gets stuck, while the fitness function of Musliu can be seen as each row is assigned to a constant weight of 1, and never changes during its local search. According to our experimental results, although Musliu can almost always find a good solution quickly, 3FNLS usually obtains the same or better solutions than Musliu as runtime increases. Whereas for RWLS, we do not define an explicit fitness function, instead, we always prefer a candidate solution with a larger total score value of the columns, even though the score values of columns are changing each iteration with the adjusting of the weights of rows.

Second, the tabu mechanisms are different. Musliu defines a tabu list whose size is the product of a tabu factor and the cardinality of the initial solution. It stores the information of the columns which are removed or added in the previous iterations, and such columns are not permitted to be selected in the following iterations. For RWLS, a variety of tabu strategies are adopted, including a timestamp method, the $canAddToSolution$ restriction, as well as not allowing the last two removed or added columns to be selected immediately in the next iteration. The main advantage of our tabu mechanism is that it is general for different instances, whereas the performance of Musliu heavily depends on fine tuning of the tabu factor for different kinds of instances. In fact, the best solutions previously reported by Musliu are obtained by exploiting different tabu factors for each individual instance, which is beyond our work, thus we only use the suggested parameters in our experiments. 3FNLS does not use any tabu mechanisms.

Third, the strategies used to escape from local optima are different, which may be the most significant difference between Musliu, 3FNLS and RWLS. RWLS uses a weighting scheme to update the weights of uncovered rows. The adaptive adjustment of weights in each iteration leads to enhanced opportunities of escaping from a local optimum. As mentioned above, Musliu can be seen as assigning all rows a constant weight of 1, and thus may get stuck in a local optimum after certain iterations. Different from Musliu, 3FNLS adaptively adjusts the weights of rows during its local search, but unlike our weighting scheme, 3FNLS uses complex weighting adjust techniques that tend to consume more computation time.

Fourth, although 3FNLS has a sophisticated local search procedure, it adopts the information from solving Lagrangian Relaxation for prioritizing columns during the search. However, in some situations, information from Lagrangian Relaxation would become useless, such as for the STS problems [6].

4.3. The Effectiveness of the Row Weighting Scheme

In RWLS, the weighting scheme is used to help the search from escaping from local optima. To investigate the effectiveness of this method, we execute our algorithm without it, which means that the weights of the rows remain 1 all the time. We then compare the results with the original RWLS on the hardest combinatorial instance CYC11. To distinguish between these two algorithms, we name the one without the row weighting scheme as RWLS-1.

Table 6: The effectiveness of the row weighting scheme on CYC11. RWLS is the original algorithm, while RWLS-1 is the algorithm without the adaptive weighting scheme. Results are obtained with random seed 11.

Algorithm \ Step	1	10^2	10^3	10^4	10^5	10^6	10^7	10^8
RWLS	4799	4742	4628	4598	4516	4289	4199	3968
RWLS-1	4799	4742	4626	4421	4371	4317	4317	4317

The first row of Table 6 contains the results obtained by RWLS, and the second row contains the results of RWLS-1, at different search steps, respectively. We can see that, initially at Step 1, RWLS and RWLS-1 have the same solution because they share the same initial solution construction method. During the first 10^5 search steps, RWLS-1 is able to obtain better solutions than RWLS, but after that, the solutions found by RWLS continue to improve, whereas those of RWLS-1 remain the same. It appears that the row weighting scheme helped RWLS to avoid being trapped in a local optimum and to continue exploring the solution space. Such phenomenon can be observed on other instances as well.

4.4. Evaluation RWLS on the Railway Crew Scheduling Instances

Table 7 gives the details of the railway crew scheduling instances from the OR-Library [5]. These instances are very large, rising up to thousands of rows and millions of columns. Hence, directly tackling them can seldom produce high quality solutions. Noting that the railway instances have many more columns than rows, one approach to deal with such instances is to use Lagrangian relaxation and its dual information to reduce the number of columns. This has been shown to be very effective for the non-unicost instances [19, 20, 6]. We will incorporate such a technique into RWLS.

Table 7: Details of the railway instances

Instance	m	n	Density	Instance	m	n	Density
RAIL507	507	63009	1.2	RAIL2586	2586	920683	0.4
RAIL516	516	47311	1.3	RAIL4284	4284	1092610	0.2
RAIL582	582	55515	1.2	RAIL4872	4872	968672	0.2
RAIL2536	2536	1081841	0.4				

[+] We turn this set of instances into unicost problems by ignoring the cost information.

More precisely, we use the problem size reduction technique from 3FNLS [6], which is based on Lagrangian relaxation and uses the subgradient method to solve the *core problem* defined by [20]. By incorporating this technique, we adapt our RWLS to Algorithm 6, which is noted as RWLS-R. The problem size reduction in 3FNLS, which is also named variable fixing, has two phases, i.e., the initial fixing stage and the modification stage. Initially, the subgradient method is called to solve the Lagrangian dual relaxation to obtain the Lagrangian cost for columns (variables) and only columns good enough are selected into the local search (the other variables are set to 0). Then, whenever the local search stops, the fixed variables are heuristically adjusted by freeing some variables whose value are zero previously. In RWLS-R, the initial column selection in Line 6 and column addition in Line 9 are based on the first-fixing phase and the modify-fixing phase in 3FNLS, respectively. The interested reader is referred to [6] for more details.

From Algorithm 6, we can see that the local search is conducted many times, i.e., each time on different sets of selected columns. In Line 5, the local search is on the original problem, and in Line 8, the local search is only on the small set of columns which have been selected. In our experiments, we set the maximum number of search steps for the local search in Line 5 to 1000, and that in Line 8 to $10 * selected\_n$, where $selected\_n$ is the number of selected columns.

In order to compare the results with yet another solver CPLEX, we run our algorithm and 3FNLS on an Intel Duo Core 2.4GHz CPU with 2 GB RAM machine, which has CPLEX12.5 [3] installed. We set the maximum runtime for RWLS-R and 3FNLS to 100 seconds for instances RAIL507, RAIL516 and RAIL586, and 1000 seconds for RAIL2536, RAIL2586, RAIL4284 and RAIL4872 because of their larger sizes. For each instance, the results of ten independent runs of RWLS-R and 3FNLS are shown in Table 8. To our best knowledge, no results have ever been reported by other methods that

[3] `http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.5.1/maps/ic-homepage.html`

Algorithm 6 RWLS-R: Incorporating RWLS with problem size reduction

```
1: function RWLS-R( )
2:     read problem instance
3:     preprocessing to add columns that able to cover some rows alone to the candidate solution permanently
4:     initialize a solution
5:     local search
6:     column selection
7:     while stopping time not reached do
8:         local search on the selected columns
9:         add some new columns to the selected columns
10:        restart the local search by initializing the candidate solution as the best found solution
11:    end while
12: end function
```

treat these railway instances as USCPs.

From Table 8, we can see that for the first three instances (RAIL507, RAIL516, RAIL582), CPLEX is able to solve them to optimality in 1375.57 seconds, 6.49 seconds and 158.67 seconds, respectively. However, because the last four instances (RAIL2536, RAIL2586, RAIL4284, RAIL4872) are very large, CPLEX failed to produce solutions on these instances, even when more than ten thousands of seconds are given.

According to Table 8, RWLS-R can find good solutions to all railway instances, including the last four large instances where CPLEX fails to produce a solution. On the first three instances, both RWLS-R and CPLEX find good solutions, whereas the best solution found by 3FNLS on RAIL582 is inferior to those found by CPLEX and RWLS-R. However, on the four larger ones (RAIL2536, RAIL2586, RAIL4284, RAIL4872), 3FNLS is able to obtain better solutions than RWLS-R. We note that the lower bounds found by 3FNLS on instances RAIL2536, RAIL2586, RAIL4284 and RAIL4872 are 363, 505, 579 and 857, respectively.

Table 8: Test railway instances as unicost problems. Italic indicates the solution values that are worse than those produced by 3FNLS.

Instance	RWLS-R				3FNLS				CPLEX12.5	
	best	avg	#best	time	best	avg	#best	time	sol	time
RAIL507	96	96.9	1	35.24	96	96.7	3	67.88	96	1375.57
RAIL516	134	134.1	9	65.45	135	135.6	4	40.93	134	6.49
RAIL582	126	126.3	7	27.60	126	126.0	10	31.37	126	158.67
RAIL2536	*381*	381.6	4	373.12	378	379.2	2	733.34	-	-
RAIL2586	*520*	521.6	2	300.62	518	518.9	2	391.10	-	-
RAIL4284	*597*	599.4	3	550.40	594	595.0	3	834.31	-	-
RAIL4872	*882*	884.6	2	778.28	879	880.5	1	817.12	-	-

[+] For 3FNLS, we convert these problems to unicost by replacing the cost information to 1 for all columns.

4.5. How Good Is CPLEX in Solving USCP?

For non-unicost SCPs, it has been shown that the random instances from sets 4 to 6, A to D, NRE, NRF and NRG1 to NRG4 can be solved to optimality by CPLEX in reasonable time [12]. However, the USCP is generally considered to be harder to solve than non-unicost SCPs [4]. In order to find out the difficulties of the random USCP instances from 4 to 6 and A to D, we apply CPLEX to these 45 instances.

In Table 9, we report the best solutions found by CPLEX within 100 seconds for groups 4 to 6, since they are quite small and generally regarded as easy, and 1000 seconds for groups A to D because of their larger sizes. The table includes the BKS (not those updated by RWLS) for comparison. It can be seen that for these 45 instances, CPLEX can only achieve seven BKSs. In fact, according to our experience, as time increases, solutions found by CPLEX improve very slowly. For instance 4.1, CPLEX can find a solution of 39 in 100 seconds, but it takes about 1000 seconds to achieve the BKS of 38. Similarly on the NRE1 instance, CPLEX needs about 15000 seconds to achieve a solution of 17. It keeps running for about 50000 seconds before terminating due to an out of memory error and the solution still remains 17.

Table 9: Results of CPLEX12.5 on instances set 4 – 6 and A – D as USCPs.

Instance	BKS	CPLEX12.5	Instance	BKS	CPLEX12.5
4.1	38	39	A.1	39	40
4.2	37	37	A.2	38	40
4.3	38	38	A.3	39	40
4.4	38	40	A.4	37	38
4.5	38	38	A.5	38	39
4.6	37	38	B.1	22	22
4.7	38	39	B.2	22	23
4.8	37	38	B.3	22	23
4.9	38	39	B.4	22	23
4.10	38	39	B.5	22	23
5.1	34	35	C.1	43	44
5.2	34	35	C.2	43	44
5.3	34	34	C.3	43	45
5.4	34	35	C.4	43	44
5.5	34	36	C.5	43	45
5.6	34	35	D.1	24	25
5.7	34	35	D.2	25	26
5.8	34	34	D.3	24	26
5.9	35	36	D.4	25	26
5.10	34	36	D.5	25	26
6.1	21	22			
6.2	20	21			
6.3	21	22			
6.4	20	21			
6.5	21	21			

+ Time limits are set to 100 seconds for instances from 4 to 6.
+ Time limits are set to 1000 seconds for instances from A to D.
+ We place the best know solutions along side to show the solution qualities of CPLEX.
+ The results are obtained by an Intel Core Due 2.4GHz CPU with 2 GB RAM machine.

The results in Table 9 indicate that the 45 USCP instances are not easy to solve, although their non-unicost versions are. Combining the results of RWLS from Table 4, we can conclude that RWLS is much better than CPLEX on USCP instances, because it almost always achieves or even improves the BKSs.

5. Conclusion and Future Work

In this paper, we have introduced a new local search heuristic, named RWLS, for USCPs. We proposed a local improvement framework to iteratively reduce the size of the currently best solution, which is realized by using two efficient operators to perturb the currently best solution when it becomes infeasible. In addition, several widely used strategies are integrated with RWLS, including an adaptive weighting scheme that adaptively updates the weights of rows (elements) to help RWLS to escape from local optima,

two tabu strategies to avoid cycles, and a timestamp method to break ties when adding or removing a column (set). RWLS successfully hybridized these general strategies into its local search framework.

The effectiveness and efficiency of RWLS have been evaluated on a large number of instances from the OR-Library [5] and Steiner triple systems [28], which vary from hundreds of rows (elements) and thousands of columns (sets) to tens of thousands of rows and columns. The experimental results show that RWLS has an excellent performance and outperforms existing state-of-the-art algorithms in terms of the best solutions found. It has improved 14 best known solutions in the literature. For the combinatorial instance CYC11, the best known solution value is improved from 4088 to 3968.

RWLS is especially effective on the ten combinatorial instances from the OR-Library as well as instances from the Steiner triple systems, which contain many more rows than columns. However, for instances containing a significantly larger number of columns but a few rows, the problem size reduction techniques should be adopted. In Section 4.4, we showed the effectiveness of RWLS in dealing with such large USCP instances by incorporating the problem size reduction technique from [6]. This is the first time that the seven railway crew scheduling instances from the OR-Library were solved as USCPs, outperforming CPLEX 12.5.

In spite of excellent performance of RWLS on 91 USCP benchmark instances, more work is needed in the future. First, the extended algorithm RLWS-R was outperformed by 3FNLS on the four larger railway crew scheduling instances. The reason for this needs to be studied. Second, the study in this paper is experimental in nature. It will be necessary to analyze the algorithm as well as the characteristics of the benchmark instances (especially the hardest ones) theoretically, so that we can understand more which algorithmic features are most important in solving what kinds of USCP instances. Third, we have not analyzed in depth the impact of different tabu strategies on RWLS's performance, which should be done in the future. Fourth, it would be useful to investigate automatic stopping techniques for RWLS, instead of setting a time limit in advance. Fifth, it would be interesting to adapt some of RWLS's ideas to solve other hard combinatorial optimization problems.

Acknowledgments

We are grateful to N. Musliu and M. Yagiura for providing their executable and source code of their algorithms, respectively. We appreciate the valuable comments of the anonymous reviewers, which have been very helpful in improving the quality and presentation of this paper.

This research work was supported by an EPSRC grant (No. EP/I010297/1), an EU FP7 IRSES grant (No. 247619) and the Fundamental Research Funds for the Central Universities (WK0110000023). Xin Yao was supported by a Royal Society Wolfson Research Merit Award. Thomas Weise was supported by the National Natural Science Foundation of China under Grants 61150110488 and 61329302, the Special Financial Grant 201104329 from the China Postdoctoral Science Foundation, and the Chinese Academy of Sciences (CAS) Fellowship for Young International Scientists 2011Y1GB01.

References

[1] C. Gao, X. Yao, T. Weise, J. Li, An efficient local search heuristic with row weighting for the unicost set covering problem, European Journal of Operational Research (EJOR) 3 (246) (2015) 750–761. doi:10.1016/j.ejor.2015.05.038.

[2] A. Caprara, M. Fischetti, P. Toth, D. Vigo, P. L. Guida, Algorithms for railway crew management, Mathematical Programming 79 (1-3) (1997) 125–141.

[3] J. Bautista, J. Pereira, Modeling the problem of locating collection areas for urban waste management. an application to the metropolitan area of barcelona, Omega 34 (6) (2006) 617–629.

[4] B. Yelbay, Ş.İ. Birbil, K. Bülbül, Set covering problem revisited: An empirical study of the value of dual information, Journal of Industrial Management and Optimization 11 (2) (2015) 575–594.

[5] J. E. Beasley, OR-Library: distributing test problems by electronic mail, Journal of the Operational Research Society (1990) 1069–1072.

[6] M. Yagiura, M. Kishida, T. Ibaraki, A 3-flip neighborhood local search for the set covering problem, European Journal of Operational Research 172 (2) (2006) 472–499.

[7] M. R. Garey, D. S. Johnson, Computers and Intractability; A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., New York, NY, USA, 1990.

[8] J. E. Beasley, An algorithm for set covering problem, European Journal of Operational Research 31 (1) (1987) 85–93.

[9] M. L. Fisher, P. Kedia, Optimal solution of set covering/partitioning problems using dual heuristics, Management Science 36 (6) (1990) 674–688.

[10] E. Balas, M. C. Carrera, A dynamic subgradient-based branch-and-bound procedure for set covering, Operations Research 44 (6) (1996) 875–890.

[11] J. E. Beasley, K. Jörnsten, Enhancing an algorithm for set covering problems, European Journal of Operational Research 58 (2) (1992) 293–300.

[12] A. Caprara, M. Fischetti, P. Toth, Algorithms for the set covering problem, Annals of Operations Research 98 (1-4) (2000) 353–371.

[13] V. Chvatal, A greedy heuristic for the set-covering problem, Mathematics of Operations Research 4 (3) (1979) 233–235.

[14] F. J. Vasko, An efficient heuristic for large set covering problems, Naval Research Logistics Quarterly 31 (1) (1984) 163–171.

[15] T. A. Feo, M. G. Resende, A probabilistic heuristic for a computationally difficult set covering problem, Operations Research Letters 8 (2) (1989) 67–71.

[16] J. E. Beasley, P. C. Chu, A genetic algorithm for the set covering problem, European Journal of Operational Research 94 (2) (1996) 392–404.

[17] L. W. Jacobs, M. J. Brusco, Note: A local-search heuristic for large set-covering problems, Naval Research Logistics 42 (7) (1995) 1129–1140.

[18] J. Beasley, A lagrangian heuristic for set-covering problems, Naval Research Logistics 37 (1) (1990) 151–164.

[19] S. Ceria, P. Nobili, A. Sassano, A lagrangian-based heuristic for large-scale set covering problems, Mathematical Programming 81 (2) (1998) 215–228.

[20] A. Caprara, M. Fischetti, P. Toth, A heuristic method for the set covering problem, Operations Research 47 (5) (1999) 730–743.

[21] S. Umetani, M. Yagiura, Relaxation heuristics for the set covering problem, Journal of the Operations Research Society of Japan 50 (4) (2007) 350–375.

[22] G. Lan, G. W. DePuy, G. E. Whitehouse, An effective and simple heuristic for the set covering problem, European Journal of Operational Research 176 (3) (2007) 1387–1403.

[23] T. Grossman, A. Wool, Computational experience with approximation algorithms for the set covering problem, European Journal of Operational Research 101 (1) (1997) 81–92.

[24] J. Bautista, J. Pereira, A GRASP algorithm to solve the unicost set covering problem, Computers & Operations Research 34 (10) (2007) 3162–3173.

[25] Z. Naji-Azimi, P. Toth, L. Galli, An electromagnetism metaheuristic for the unicost set covering problem, European Journal of Operational Research 205 (2) (2010) 290–300.

[26] H. H. Hoos, T. Stützle, Stochastic Local Search: Foundations and Applications, San Francisco, USA: Morgan Kaufmann, 2005.

[27] N. Musliu, Local search algorithm for unicost set covering problem, in: Advances in Applied Artificial Intelligence, Springer, 2006, pp. 302–311.

[28] D. Fulkerson, G. Nemhauser, L. Trotter, Two computationally difficult set covering problems that arise in computing the 1-width of incidence matrices of steiner triple systems, in: M. Balinski (Ed.), Approaches to Integer Programming, Vol. 2 of Mathematical Programming Studies, Springer Berlin Heidelberg, 1974, pp. 72–81.

[29] E. Balas, A. Ho, Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study, in: M. Padberg (Ed.), Combinatorial Optimization, Vol. 12 of Mathematical Programming Studies, Springer Berlin Heidelberg, 1980, pp. 37–60.

[30] H. Harborth, H. Nienborg, Maximum Number of Edges in a Six-cube Without Four-cycles, Bericht: Technische Universität Braunschweig, Institute für Mathematik, Techn. Univ., 1994.

[31] J. Ostrowski, J. Linderoth, F. Rossi, S. Smriglio, Solving large steiner triple covering problems, Operations Research Letters 39 (2) (2011) 127–131.

[32] M. G. Resende, R. F. Toso, J. F. Gonçalves, R. M. Silva, A biased random-key genetic algorithm for the steiner triple covering problem, Optimization Letters 6 (4) (2012) 605–619.

```
@article{GYWL2015AELSHWRWFTUSCP,
  author    = {Chao Gao and Xin Yao and Thomas Weise and Jinlong Li},
  title     = {{An Efficient Local Search Heuristic with Row Weighting for the Unicost
               Set Covering Problem}},
  journal   = {European Journal of Operational Research (EJOR)},
  number    = {246},
  volume    = {3},
  pages     = {750--761},
  year      = {2015},
  month     = nov,
  doi       = {10.1016/j.ejor.2015.05.038},
},
```