Using Neighbourhood Search to Solve Generalised Staff Rostering Problems

I.D. Cleland, A.J. Mason, M.J. O'Sullivan
Department of Engineering Science
University of Auckland
New Zealand
icle701@aucklanduni.ac.nz

Abstract

The Staff Rostering Problem involves optimising the assignment of staff to shifts, whilst fulfilling rules associated with these assignments. There are significant time-saving benefits to solving Staff Rostering Problems automatically and so there have been many attempts over the years to solve these problems with optimisation and heuristics. However, these problems are very difficult to solve automatically; they are NP-Hard, regularly have a very large number of variables, have a non-linear or multi-objective cost function, and are tightly constrained.

In this paper, we will be demonstrating our preliminary investigations into several novel algorithms which involve a hybrid of column generation and local search.

Key words: Column Generation, Dantzig Wolfe, Matheuristics, Neighbourhood Search, Staff Rostering, Scheduling, Nurse Rostering, VNS, Local Branching

1 Background

1.1 Staff Rostering Problems

The Staff Rostering Problem is a problem within the field of Operations Research (OR) involving optimising the assignment of staff to shifts, whilst fulfilling rules (e.g. limited numbers of weekends worked, limited night shifts in a row, minimum hours worked) associated with each staff's roster-line (the set of shifts to which a staff member is assigned) and satisfying the demand for each shift. There have been extensive reviews around this subject by Ernst et al. on generic staff rostering (Ernst et al. 2004) and Burke et al. and Cheang et al. focused on general nurse rostering (Burke et al. 2004) (Cheang et al. 2003). It is usually solved for a set period of time, with several different types of shifts differentiated by their start time, finish time, demand and skill requirement. Some common applications of staff rostering include crew scheduling (the process of assigning crews to operate transportation systems), nurse scheduling, and call center scheduling. The rules for making rosters for each of these applications differ significantly so most papers focus on just one application.

Figure 1 shows a typical roster solution to a staff rostering problem. In this staff rostering problem example, we assume that each staff member can only have one

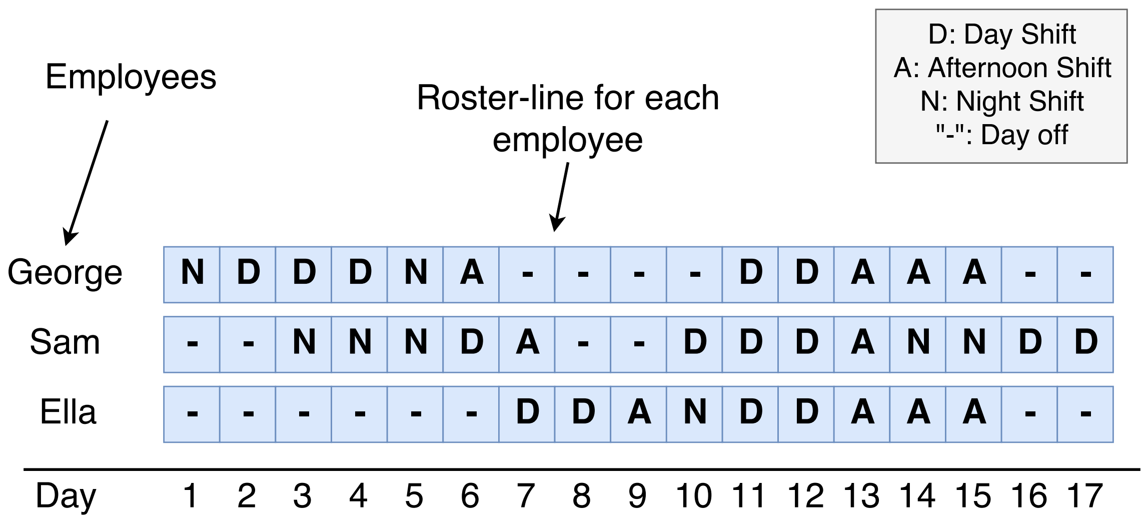

D: Day Shift
A: Afternoon Shift
N: Night Shift
"-": Day off
Employees
Roster-line for each employee
George N D D D N A - - - - D D A A A - -
Sam - - N N N D A - - D D D A N N D D
Ella - - - - - - D D A N D D A A A - -
Day 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Figure 1: Staff Rostering Problem

shift per day, which is the type of problem we are attempting to solve during this paper.

Until recently, most staff rostering problems were done manually, which is very time consuming and means the quality of the roster is difficult to assess. There are huge time-saving benefits to solving these problems automatically and so there have been many attempts over the years to solve these problems using both optimization and heuristics. However, these problems are very difficult to solve automatically; they are NP-Hard (Richard and Karp 1971), regularly have a very large number of variables, a non-linear or multi-objective cost function, and are tightly constrained.

1.2 Original Software

This paper will focus on an improvement to the original software called Genie++ (Dohn and Mason 2013) (Mason and Smith 1998) which was developed for the purpose of generalized staff rostering. Genie++ uses nested column generation and generic programming to be able to solve a broad range of staff rostering problems quickly.

Column generation is an algorithm used to solve large linear programs (LPs) efficiently where there are a very large number of variables. Instead of solving the entire LP, only a subset of the variables (columns) are included at once as a restricted LP. Then, negative reduced cost variables are generated in a separate subproblem (multiple generation methods are used including dynamic programs (DPs), mixed integer programs (MIPs), constraint programs (CPs) etc.) and added to the restricted LP which is re-solved until no more negative reduced cost columns can be generated and the solution is optimal to the original LP. For more information, the book Column Generation (Irnich, Desaulniers, and Others 2005) is a comprehensive review of most modern techniques.

The following model shows the set partitioning formulation we use to generate rosters from a limited pool of roster-lines (ie columns specifying potential sets of shifts and days off for a given employee).

Indices

$i =$ shift index,

j = employee roster-line (J is the set of all roster-lines),
k = employee index.

Parameters

c_j = cost of employee roster-line j;
$A_{i,j}$ = 1 if employee roster-line j fulfills demand for shift i, 0 otherwise;
$E_{k,j}$ = 1 if roster-line j is associated with employee k, 0 otherwise;
b_i = demand for each shift i;
s_i = maximum number of additional employees that can do shift i after the demand has been reached.

Decision variables

λ_j = 1 if an employee does roster-line j, 0 otherwise;
y_i = surplus variable.

Staff Rostering Mixed Integer Program (SRMIP)

Minimize $\sum_{j \in J} c_j \lambda_j$

$$s.t. \sum_{j \in J} A_{i,j} \lambda_j - y_i = b_i \quad \forall i \in \text{Shifts} \tag{1}$$

$$\sum_{j \in J} E_{k,j} \lambda_j = \mathbf{1} \quad \forall k \in \text{Employees} \tag{2}$$

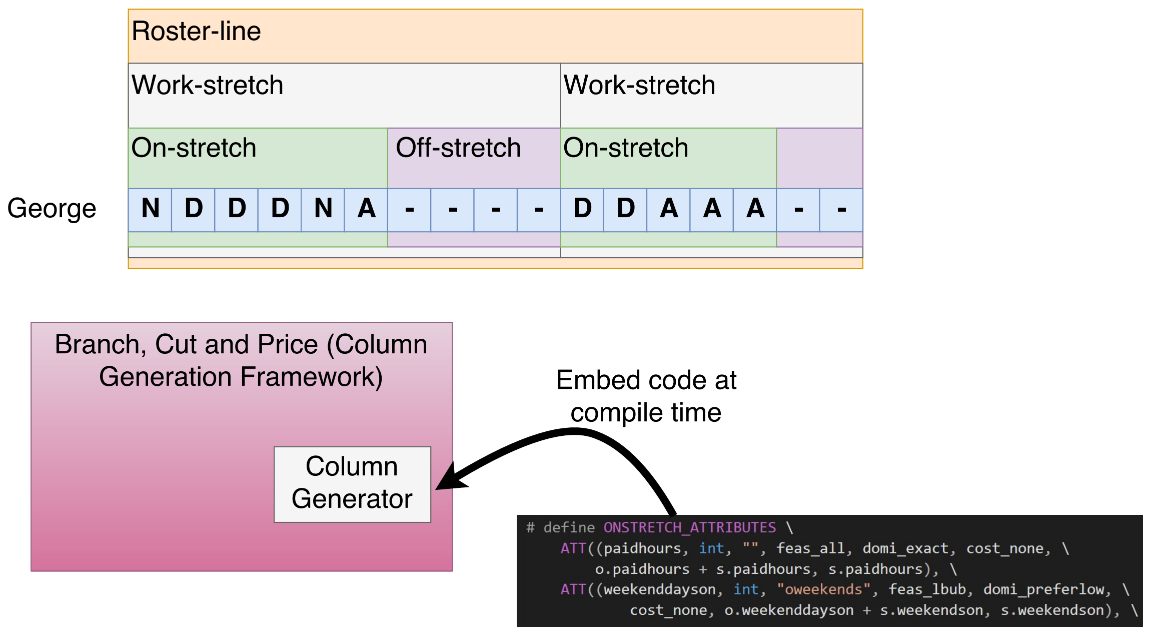
$$\lambda_j \geq 0, Integer$$

$$0 \leq y_i \leq s_i.$$

Explanation The objective is to minimize the total cost of the roster which is the sum the costs of each employee's roster-line. The first set of constraints (1) ensure that the demands for each shift are met. The generalised upper bound (GUB) constraints (2) ensure each employee has exactly one roster-line.

The Genie++ column generation subproblem consists of a nested dynamic program. Genie++ models the roster-lines of each employee using nested building blocks called entities. These entities are demonstrated on Figure 2. There are five main entities: a shift, an on-stretch, an off-stretch, a work-stretch and the roster-line itself. A shift is the basic building block. An on-stretch is a sequence of shifts on consecutive days (Genie++ currently makes the assumption that only one shift is done by one employee per day). An off-stretch is a set of consecutive days off. A work-stretch is made with one on-stretch and one off-stretch with no days in between the end of the on-stretch and the start of the off-stretch. A roster-line is made up of a series of consecutive work-stretches.

Each entity has certain attributes, such as the number of weekends off in an off-stretch or the number of hours worked in an on-stretch. Each attribute has rules for how the value of the attribute is initialized from the value of the attributes of the first sub-entity added (e.g. a new on-stretch is initialised by one shift; the start time attribute value for the new on-stretch is equal to the start time attribute value for the first shift) and how the value accumulates as more sub-entities are added (e.g. the total number of weekend days worked in the onstretch is incremented by 1 if the added shift is on a weekend day). There are also rules defined based on these attribute values which are either strictly enforced (e.g. no working during any


Roster-line
Work-stretch
Work-stretch
On-stretch
Off-stretch
On-stretch
George
N D D D N A - - - - D D A A A - -
Branch, Cut and Price (Column Generation Framework)
Column Generator
Embed code at compile time

```
# define ONSTRETCH_ATTRIBUTES \
    ATT((paidhours, int, "", feas_all, domi_exact, cost_none, \
        o.paidhours + s.paidhours, s.paidhours), \
    ATT((weekenddayson, int, "oweekends", feas_lbub, domi_preferlow, \
            cost_none, o.weekenddayson + s.weekendson, s.weekendson), \
```

Figure 2: Genie++ System

weekend) or just add a cost to the roster-line objective function if broken (e.g. add 1 to the cost of the roster-line for each weekend worked). The attributes and rules can be modeled in many different ways which makes Genie++ very flexible in modeling problems.

To solve the nested dynamic program and generate roster-lines, two nested dynamic programs (DPs) are solved. The first DP is used to generate and price (i.e. compute reduced costs for) all feasible, dominant (has the lowest reduced cost out of all entities with the same attribute values) on-stretches. Then an enumeration process generates and prices all feasible, dominant work-stretches by combining every on-stretch that we have generated with any feasible off-stretch for that on-stretch. The second DP is used to combine work-stretches to generate and price all feasible, dominant roster-lines. By solving the DP, in a nested way, many roster-lines that won't be optimal or feasible are removed by the dominance or feasibility checks performed on sub-entities before the roster-line is priced, which makes the subproblem much more efficient in finding the optimal roster-line.

To speed up the column generation subproblem, Genie++ uses generic programming which involves using the C++ Boost Preprocessor library functionality to write individualized code for the model's subproblem. This individualized code dictates how feasibility and dominance checks are performed for each entity based on its attributes and how attributes are initialized and accumulate. The code is generated directly from the model description which is currently in the form of a C++ header file. An example of some of the model description code can be seen in Figure 2. This functionality improves the speed of the dynamic program significantly as only the necessary calculations for the specific subproblem are made. This functionality also means the code is compile-time optimised by the C++ compiler which makes it significantly faster.

Genie++ was implemented using COINOR's Branch, Cut and Price (BCP) library which is a parallel framework for handling column generations problems. However, for our tests, we will be using a faster, custom column generation library which we have developed for efficient testing of column generation based algorithms.

One of the core limitations of Genie++ in its original implementation was that it was still slow for some types Staff Rostering Problems. As we want it to be used for the largest variety of Staff Rostering Problems possible, we wish to develop multiple solve methods which can be used on different classifications of problems so that Genie++ can consistently solve Staff Rostering Problems quickly.

1.3 Neighbourhood Search in Column Generation

One method of improving the solve times for some classifications of Staff Rostering Problems is by using Variable neighbourhood Search within the column generation framework.

Variable Neighbourhood Search (VNS) is an extension of standard local search. Local Search involves searching a subset of the solution space in the neighbourhood of a feasible, integer solution. A neighbourhood is defined as all the solutions which only differ by a defined small aspect (e.g. any one shift swapped in any one of the roster-lines in a roster solution) from the original solution.

In VNS, the local optimum is found via a local search, then the neighbourhood is changed and the search continues until another local optimum is found until no better solution can be found from changing the neighbourhood. Jumps are then performed by randomly changing one part of the best solution found so far and exploring the multiple neighbourhoods again. A recent, detailed exploration of applications and methods of VNS was performed by Hansen (Hansen, Mladenovi, and Moreno Perez 2010). VNS can not only be used to find feasible upper bounds to the branch and bound tree but can also be used to improve the speed of solving the relaxed problem in a MIP (Hansen et al. 2007).

VNS is referenced several times in the Staff Rostering Problem literature. Burke et al. establish a neighbourhood of various swaps including 1-swaps and 2-swaps within a roster-line as well as swapping whole roster-lines. This is used to improve solutions from a MIP model which isn't initially solved to optimality (Burke, Li, and Qu 2010). Rahimian et al. use a similar but larger neighbourhood than Burke et al. and also performs jumps in their VNS by using a MIP to optimise a small number of employees' roster-lines in the current best feasible solution to hopefully produce a new solution with a better local optimum. Rahimian et al.'s VNS only uses simple greedy heuristics to attain starting solutions and is comparable to Burke et al. column generation framework on many benchmark instances (Rahimian, Akartunali, and Levine 2017). Several mathematical models have also been used to explore neighbourhoods. Santos et al. use a variation on VNS called Variable Neighbourhood Descent (VND). VND involves fixing a certain number of variables in the formulation to be equal to corresponding variables in an incumbent solution and solving to integer optimality, then changing the neighbourhood by increasing or decreasing the number of variables being fixed until no further improvements can be made or a time limit is reached. Santos et al. perform VND on staff rostering problems by fixing both shifts and days to the same as those in an incumbent. Santos et al. initially fixed everything except one day or shift and unfixed more variables every iteration of their VND (Santos et al. 2016). Smet et al. use three neighbourhoods in their search: VND on each day as was used by Santos, VND on each individual employee, and VNS using local branching which involves restricting the Hamming distance (sum of differences in variable values between LP solution and an incumbent solution) with a constraint, solving the MIP to optimality, then changing the maximum Hamming distance and repeating the process (Smet, Ernst, and Vanden Berghe 2016). It should be noted that neither Santos et al. nor Smet et al. use a column generation decomposition with their VNS methods and are both using 0-1 MIP formulations with these neighbourhood searches.

Although VNS and VND are referenced frequently with both swapping and moving heuristics, and in solving 0-1 MIP formulations (Hanafi and Todosijević 2017), (Hanafi et al. 2015), there has not been much discussion with regard to using these methods within a column generation solver and consequently also within a column generation based staff rostering solver. Hansen et al. propose using local branching as a variable neighbourhood with VNS for an effective way to explore large neighbourhoods around incumbent solutions quickly (Hansen, Mladenović, and Urošević 2006). However, to our knowledge, this has not been applied to general column generation. In this paper, we experiment with this novel column generation technique by both using Local Branching in the LP and by restricting the column generator itself.

2 Implementation

2.1 Variable neighbourhood Search in Genie++

We implemented a simple variable neighbourhood search algorithm shown below (Algorithm 1). Each local search neighbourhood is defined as all rosters which are k distance (where distance is the maximum amount of changes allowed from the original roster) from the incumbent roster. Local Search is performed on a given incumbent roster by solving the Staff Rostering Problem using Genie++ with constraints in the Linear Program or in the column generation subproblem to restrict the solutions to be k distance from the incumbent roster.

Local search is performed with a given neighbourhood until no improvements to the incumbent roster can be made. The neighbourhood is then increased by k_{step} until the size of the neighbourhood, k, reaches the given maximum neighbourhood size, k_{max}.

Algorithm 1 Basic Variable neighbourhood Search

```
 1: procedure VNS($x, k_{min}, k_{max}, k_{step}$)
 2:     $k \leftarrow k_{min}$
 3:     $x' \leftarrow x$
 4:     while $k \leq k_{max}$ do
 5:         while $f(x'') < f(x')$ do
 6:             $x' \leftarrow x''$
 7:             $x'' \leftarrow LocalSearch(x', k)$
 8:         if $f(x') < f(x)$ then
 9:             $x \leftarrow x'$
10:             $k \leftarrow k_{min}$
11:         else
12:             $k \leftarrow k + k_{step}$
13:     return $x$
```

We experimented with three separate neighbourhoods within Genie++ all of which take one parameter, k. These are: 1) Local branching with employee roster-lines, 2) Local branching with employee-shift variables, 3) Column-wise Restricted neighbourhood Search. We next will discuss these in detail.

2.2 Local Branching - Employees

Our first type of Local Search uses Local Branching, which involves adding the constraint shown below (3) to our SRMIP formulation. This constraint forces a minimum number of roster-lines in the original incumbent roster to be in any new rosters produced by this Local Search method. In this case, the neighbourhood distance parameter, k, describes the number of roster-lines which can be different from those in the original incumbent roster.

Parameters

$\alpha_j = 1$ if roster-line j is in the incumbent solution, 0 otherwise,
$n =$ total number of employees,
$k =$ neighbourhood distance parameter.

Additional Constraints

$$\sum_{j \in J} \alpha_j \lambda_j \geq n - k \tag{3}$$

This method of local branching is the easiest to implement as no changes to our column generator, nor any dual manipulation, are required for this formulation.

2.3 Local Branching - Employee-Shift

The second type of Local Search also uses Local Branching but instead involves adding the constraint shown below (4) to our SRMIP formulation. This constraint imposes a limit on the number of employee-shift variables that can be different from the original incumbent roster in any new rosters produced by this Local Search method. An employee-shift variable represents whether an employee does a particular shift in the roster. As such, when generating new columns within the local search, the roster-line needs to be compared with the same employee's roster-line in the original incumbent roster and the number of shift differences is calculated and added to the model as the parameter β_j for roster-line j. k is the maximum permitted for the total sum of these differences over the whole roster.

Parameters

$\beta_j =$ total number of shift differences compared to incumbent roster-line,
$k =$ neighbourhood distance parameter.

Additional Constraints

$$\sum_{j \in J} \beta_j \lambda_j \leq k \tag{4}$$

No changes are required in our column generator for this formulation. However, there are complexities with regards to the dual for the additional constraint. Since each column is built one shift at a time, in order to find the column with the optimal dual cost, the dual for the additional constraint must be added to the column each time there is either a shift being added on some d that is different from the incumbent for the employee or if the column generator isn't adding a shift that is done by the employee in the incumbent.

We have not found any publications that have used this same technique for rostering or even column generation. However, it poses a significant risk of increased fractionality and hence longer solve times due to the additional constraint not having binary coefficients.

2.4 Column-wise Restricted neighbourhood Search

The third type of Local Search involves putting restrictions within the column generator itself. A new attribute is tracked in the dynamic program called Incumbent Resource which tracks the total shift differences in each entity (shifts, on-stretches, off-stretches, work-stretches, roster-lines) from entities in the incumbent roster. Any entity generated with more than k changes from the incumbent roster-line of the same employee is discarded. So in this case, k represents the maximum number of employee-shift variable differences from the incumbent for each employee's roster-line.

Figure 3 shows the accumulation of the Incumbent Resource in the Column Generator. If a shift is added to the on-stretch which isn't part of the incumbent roster-line, such as the morning shift in the example, the Incumbent Resource attribute is incremented by 1. This attribute is limited by k when generating roster-lines.

2.5 Experiments

In order to test our various Local Search strategies for effectiveness, we have initially used the International Nurse Rostering Competition (INRC) test problems.

The INRC problems are of 3 different lengths: 'sprints' which required rostering 10 nurses on for 4 weeks with 4 shift types (11 seconds were allowed in the competition to solve these), 'mediums' which required rostering 30 nurses on for 4 weeks with 4 different shift types (11 minutes were allowed in the competition to solve these) and 'longs' which required rostering 49 nurses on for 4 weeks with 5 different shift types (11 hours were allowed in the competition to solve these).

We performed 4 tests on each of the INRC problems. The first was a basic solve with standard Branch, Cut, and Price. This test is intended to be a benchmark with which to compare the other methods.

The other 3 tests followed the same protocol. An initial feasible integer solution was generated using a simple construction heuristic. Then each Local Search algorithm is called with a different parameters (k_{min}, k_{step}, and k_{max}) for each algorithm. Different parameters were used for each experiment because they have a very different effect on each algorithm. For example $k = 1$ for employee Local Branching has a much larger effect on the solution than $k = 1$ for employee-shift Local Branching since changing an employee changes many employee-shift variables.

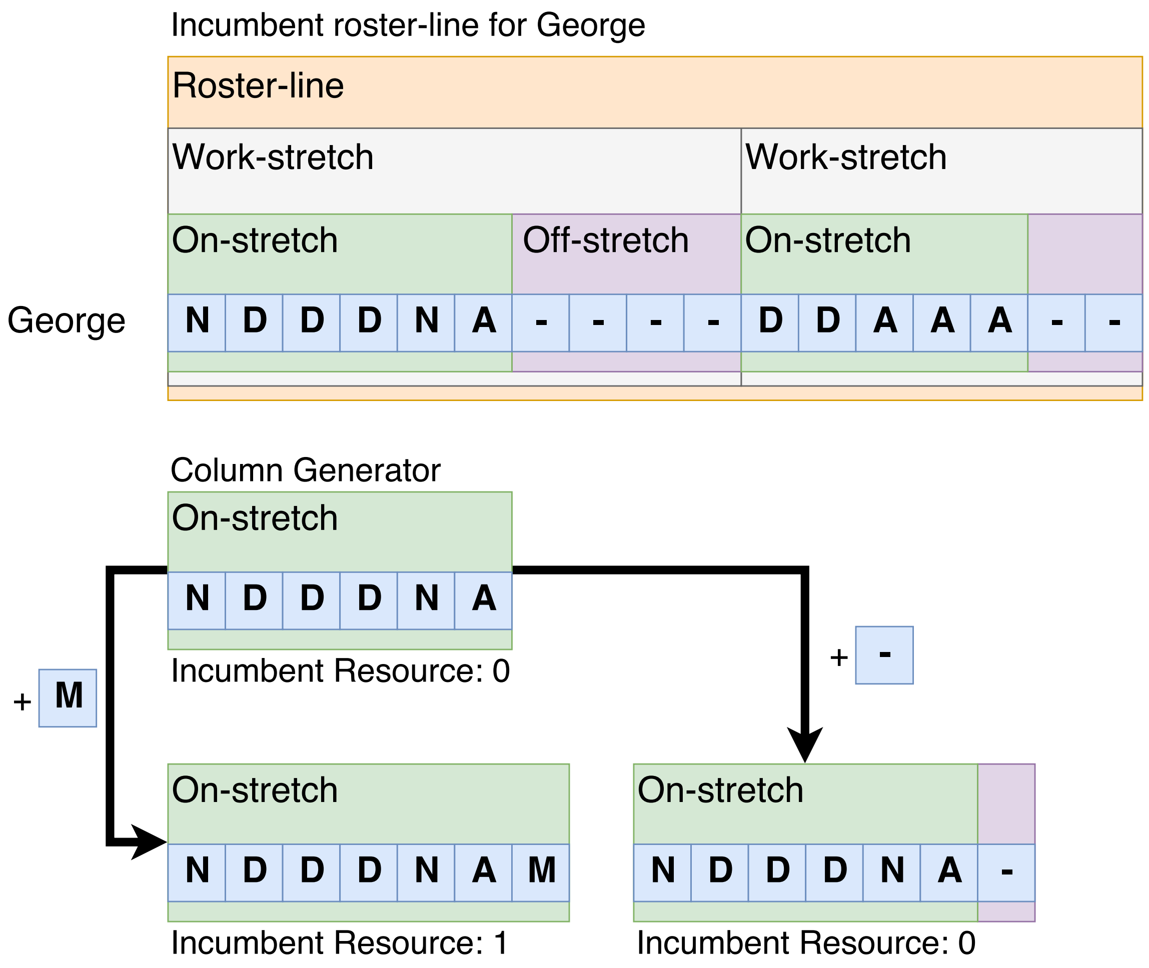


Figure 3: Accumulation of Incumbent Resource Attribute

3 Results

Benchmark	Standard		Emp LB		Emp-Shift LB		Col Restricted	
	% Error	Time(s)	% Error	Time(s)	% Error	Time(s)	% Error	Time(s)
Medium01	0	391	12	4423	N/A	N/A	0	586
Long01	0	311	9	7636	N/A	N/A	0	996.604

Table 1: Accumulation of Incumbent Resource Attribute

The results of our tests can be seen in the table above. The results are merely preliminary and limited at this stage as we are currently in the process of doing experiments and refining our software for this research.

The preliminary results seem to show that Column-wise Restricted Local Search is significantly better than Employee Local Branching. However, none of the Local Search algorithms have yet been refined enough to beat the times of our original BCP system.

The test data we have been using has also been shown to only be one type of Staff Rostering Problem as all the INRC problems are modeled very similarly and behave in similar ways when being solved.

Furthermore, the tests we have been performing have none of the integrated heuristics that we have built into Genie++ in use, so are purely for comparison purposes against the Local Search methods and not for benchmarking Genie++ against other INRC benchmarks.

4 Conclusions

We have developed a powerful system for using VNS with multiple column generation based local search methods to solve Staff Rostering Problems. These methods could potentially be more efficient than standard column generation for some problem types.

However, at this stage, no firm conclusions about the nature of column generation based local search can be made as the algorithm and parameters have not been refined as of yet and only one type of Staff Rostering Problem has been tested. Nevertheless, these approaches appear to be novel and could potentially be a good contribution to both the Staff Rostering literature and column generation literature if they are especially effective.

References

Burke, Edmund K., Patrick De Causmaecker, Greet Vanden Berghe, and Hendrik Van Landeghem. 2004. “The state of the art of nurse rostering.” *Journal of Scheduling* 7 (6): 441–449.

Burke, Edmund K., Jingpeng Li, and Rong Qu. 2010. “A hybrid model of integer programming and variable neighbourhood search for highly-constrained nurse rostering problems.” *European Journal of Operational Research* 203 (2): 484–493.

Cheang, B., H. Li, A. Lim, and B. Rodrigues. 2003. “Nurse rostering problems - A bibliographic survey.” *European Journal of Operational Research* 151 (3): 447–460.

Dohn, Anders, and Andrew Mason. 2013. “Branch-and-price for staff rostering: An efficient implementation using generic programming and nested column generation.” *European Journal of Operational Research* 230 (1): 157–169.

Ernst, A. T., H. Jiang, M. Krishnamoorthy, and D. Sier. 2004. “Staff scheduling and rostering: A review of applications, methods and models.” *European Journal of Operational Research* 153 (1): 3–27.

Hanafi, Saïd, Jasmina Lazíc, Nenad Mladenovíc, Christophe Wilbaut, and Igor Cevits. 2015. “New variable neighbourhood search based 0-1 MIP heuristics.” *Yugoslav Journal of Operations Research* 25 (3): 343–360.

Hanafi, Saïd, and Raca Todosijević. 2017. “Mathematical programming based heuristics for the 01 MIP: a survey.” *Journal of Heuristics*, pp. 165–206.

Hansen, Pierre, Jack Brimberg, Dragan Urošević, and Nenad Mladenović. 2007. “Primal-dual variable neighborhood search for the simple plant-location problem.” *INFORMS Journal on Computing* 19 (4): 552–564.

Hansen, Pierre, Nenad Mladenovi, and Jose A. Moreno Perez. 2010. “Variable neighbourhood search: Methods and applications.” *Annals of Operations Research* 175 (1): 367–407.

Hansen, Pierre, Nenad Mladenović, and Dragan Urošević. 2006. “Variable neighborhood search and local branching.” *Computers and Operations Research* 33 (10): 3034–3045.

Irnich, Stefan, Guy Desaulniers, and Others. 2005. *Shortest path problems with resource constraints.*

Mason, Andrew J., and Mark C Smith. 1998. “A Nested Column Generator for solving Rostering Problems with Integer Programming.” pp. 827–834.

Rahimian, Erfan, Kerem Akartunali, and John Levine. 2017. “A hybrid Integer Programming and Variable Neighbourhood Search algorithm to solve Nurse Rostering Problems.” *European Journal of Operational Research* 258 (2): 411–423.

Richard, T., and M. Karp. 1971. “Reducibility among combinatorial problems University of California at Berkeley.” *Science*, pp. 85–103.

Santos, Haroldo G., T??lio A M Toffolo, Rafael A M Gomes, and Sabir Ribas. 2016. “Integer programming techniques for the nurse rostering problem.” *Annals of Operations Research* 239 (1): 225–251.

Smet, Pieter, Andreas T. Ernst, and Greet Vanden Berghe. 2016. “Heuristic decomposition approaches for an integrated task scheduling and personnel rostering problem.” *Computers and Operations Research* 76:60–72.