Evolving Staff Training Schedules using an Extensible Fitness Function and a Domain Specific Language

Neil Urquhart,Kelly Hunter

School of Computing, Edinburgh Napier University, UK
{n.urquhart,k.hunter}@napier.ac.uk

Abstract. When using a meta-heuristic based optimiser in some industrial scenarios, there may be a need to amend the objective function as time progresses to encompass constraints that did not exist during the development phase of the software. We propose a means by which a Domain Specific Language (DSL) can be used to allow constraints to be expressed in language familiar to a domain expert, allowing additional constraints to be added to the objective function without the need to recompile the solver. To illustrate the approach, we consider the construction of staff training schedules within an organisation where staff are already managed within highly constrained schedules. A set of constraints are hard-coded into the objective function in a conventional manner as part of a Java application. A custom built domain specific language (named Basil) was developed by the authors which is used to specify additional constraints affecting individual members of staff or groups. We demonstrate the use of Basil and show how it allows the specification of additional constraints that enable the software to meet the requirements of the user without any technical knowledge.

1 Introduction and Motivation

Evolutionary Algorithms and related meta-heuristics have been developed to solve a range of real-world problems. However, it is inevitable that an organisation's needs change over time: as a result, the constraints within the original system may no longer meet their needs. While the users of a system may have problem domain knowledge, they may not have any software engineering experience. Updating constraints within the system may pose difficulties, requiring a software-engineering specialist to alter the system. This paper proposes a mechanism by which a user with knowledge of the problem domain can add additional constraints to the objective function, in a manner that does not require specialist software engineering skills, using a custom built domain-specific language (DSL) designed and implemented by the authors.

We consider the specific case of an industrial partner within the public transportation sector, who has limited software development expertise. The partner has a requirement to provide staff training schedules within a heavily constrained environment. Staff must be to be allocated to training slots in a manner that causes least disruption to the existing schedules. As with many problems that involve the scheduling of people, many constraints exist based around the specific requirements of individuals that are not necessarily known in the original design phase.

The contribution of this paper is to address the following research questions:

1. To what extent can constraints be expressed in a custom built DSL a manner that is achievable by a domain expert?
2. What mechanisms could be used to evaluate constraints specified using a DSL against candidate solutions at run time?
3. How does the evaluation time scale as more constraints are added at runtime?

The principle contribution of this paper is the development of the DSL and the use of pattern matching to evaluate constraints.

This paper is organised as follows, Section 2 describes related work in the field of optimisation. The problem domain is described in Section 3 and the evolutionary algorithm used to produce solutions is described in Section 4. The development of a domain specific language (DSL) specifically for this application is described in Section 5 along with the mechanism by which Basil statements are compiled into regular expressions which are then matched against the solution being evaluated. Finally, conclusions and future work are described in Section 6.1.

2 Related Work

Evolutionary Algorithms and other meta-heuristics have been applied to problems related to a number of industrial sectors including timetabling [11, 1], staff scheduling, [3, 2, 10], vehicle routing and logistics [4, 6] and job shop/factory scheduling [5]. The domain of staff scheduling, and in particular nurse scheduling has received a great deal of attention, for a recent survey of this domain the reader is directed to [7]. Whilst some industrial scheduling problems map closely to traditional benchmark problem types (e.g., the Travelling Salesman Problem, Vehicle Routing Problem or Flow Shop Problems) many incorporate constraints that are specific to the organisation who own the problem.

There exists the issue of how to specify these organisation specific constraints in a manner that is suitable for organisations who do not have the capability to modify the underlying software. One option is the use of a domain specific language (DSL) to specify constraints. Regenell and Kuchcinski [8] describe the use of an embedded DSL for combinatorial optimisation. The approach taken is based on the Scala platform, the resultant DSL making use of the Scala syntax. Whilst this approach has much to recommend it, not least the ability to integrate the DSL compilation with that of the main Scala application. Constraints can also be specified in a constraint modelling language such as MiniZinc [12].

The DSL-based approaches outlined above have the disadvantages that the DSLs are difficult to use by domain experts who are not software engineers. In this work, the DSL presented (Basil) is designed specifically around entities within the problem domain in order to make it usable by problem domain experts within an organisation that does not have software development expertise.

3 Problem Domain

3.1 Problem Definition

A major Scottish public transport provider employs over 2,000 drivers. It is a requirement under current UK/EU legislation that professional drivers must undertake mandatory Certificate of Professional Competence (CPC) training [9]. Within a five-year period each driver must undertake 35 hours of training: failure to complete the required amount of training results in the drivers' license expiring, losing their right to drive on a commercial basis.

An existing proprietary software package is used to schedule drivers to their routine duties, but this system does not schedule the time required for CPC training. The policy of the organisation is that each driver is allocated one CPC training day per year - this ensures that they will have undertaken the required 35 hours within the five-year period. Each driver has a specific license expiry date based on when they completed their initial training which specifies the deadline by which their CPC training must have been completed within the fifth year.

A total of 2014 drivers employed are split into groups, representing the area of the organisation that they work for (see Table 1), each driver must have one CPC training day per year. Each training day can accommodate 12 trainees, training takes place for 40 weeks per year for 5 days per week, creating 2400 training places. Assuming that each member of staff attends on the day that they are scheduled then there is a 16 % spare capacity. In practice this capacity is required to cover situations such as non-attendance due to illness or where the staff member cannot be released for training due operational requirements.

Group Size	50	24	750	400	450	140	140	60
Max. Trainees per Day	1	1	5	3	3	1	1	1

Table 1. The employee group sizes with the problem being considered. The maximum number of employees which may be allocated to training from each group on the same day is shown.

There are a number of constraints that govern the CPC training schedule:

1. Any driver whose license expires in the current year must have their training day, for that year, prior to their license expiring.
2. Each driver may only attend one CPC training day each year.

3. Each training day can only accommodate 12 trainees.
4. The number of trainees on each day from a specific group must not exceed the limit set for that group (see Table 1).
5. In their normal duties, each week drivers are allocated to duties that are classified as either 'early' or 'late', if possible, CPC training days should be scheduled for drivers when they are already allocated to early duties, this makes it easier to release them for the training.

3.2 Problem Instances

The problem instances used in this paper are generated randomly, based on statistics supplied by the partner. This avoids having to share commercially sensitive data during the development stage.

Parameter	Value
Class_Size	12
Early_Shift_Probability	0.5
Probability_license_expires	0.2
Training_Weeks	40
Training_Days_Week	5

Table 2. The parameters used when generating the test instances.

4 Solving Using an Evolutionary Algorithm

4.1 Algorithm Description

The algorithm used within this paper is named *CPC-EA* and is described in Algorithm 1, the parameters used are given in Table 3. A steady state population is employed: within the generational loop (Lines 4-23) one new *child* solution is created by either recombination of two parents (Lines 6-8) or by cloning a single parent (Line 10). The child then replaces the loser of a tournament (Line 15) providing the child fitness is an improvement on the loser (Lines 16-18).

Most academic use of EAs described in studies execute the EA for a fixed number of evaluations (referred to as an *evaluation budget*). In this application our aim is to produce a usable schedule for the business, so there is no requirement to limit the evaluations to a specific time frame, but instead the algorithm can execute until it cannot find any more improvements to the solution. A parameter *MAX_EVALS* is used to specify the maximum number of evaluations that will be carried out in order to prevent excessively long execution times.

Algorithm 1 CPC-EA

```
1: $pop = initialise(POP\_SIZE)$
2: $bestSol = findBest(pop)$
3: $evalLeft = TIMEOUT$
4: **while** $evalsLeft > 0$ **do**
5:     **if** $random() < XOVERRATE$ **then**
6:         $p1 = tournament(pop, TOURSIZE)$
7:         $p2 = tournament(pop, TOURSIZE)$
8:         $child = recombine(p1, p2)$
9:     **else if** $random() >= XOVERRATE$ **then**
10:        $child = tournament(pop, TOURSIZE)$
11:    $mutate(child)$
12:    $evalute(child)$
13:    $evals++$
14:    $evalsLeft--$
15:    $rip = tournamentLoose(pop, TOURSIZE)$
16:    **if** $child.fitness < rip.fitness$ **then**
17:        $pop.remove(rip)$
18:        $pop.add(child)$
19:        **if** $child.fitness < bestSol.fitness$ **then**
20:            $bestSol = child$
21:            $evalLefts = TIMEOUT$
22:    **if** $evals > MAX\_EVALS$ **then**
23:        $return$
```

Parameter	**Value**
POP_SIZE	100
TIME_OUT	25,000
XOVER_RATE	0.5
TOUR_SIZE	2
MAX_EVALS	250,000

Table 3. Parameters used within CPC-EA in this paper.

Representation Each solution comprises a list of training slots, the total number of slots being calculated as $Class\_Size \times Training\_Days\_Week \times Training\_Weeks$ (see Table 2), for the instances under consideration this equates to 2400 training slots. Each slot may be empty or have a driver allocated to it. All the drivers in the problem must be allocated to a slot for a valid solution to exist (in our definition a valid solution is one in which all drivers are allocated a training day regardless of any other constraints). Table 4 gives an example of the representation used.

Driver	**Training Slot**
00001	2.1.1
00002	40.4.4
00003	34.3.5
00004	21.2.1
00005	16.1.12
00006	14.5.5
00007	21.2.3
00008	22.3.4
00009	35.2.2
00010	35.4.7
00011	2.1.2
00012	9.3.12
...	...

Unused Slots
17.1.1
40.4.12
8.5.11
16.2.8
21.5.4
24.5.12
33.1.11
35.3.1
26.2.12

Table 4. A truncated example of a solution, one entry exists for each driver which has a training slot associated with it (labelled as <week>.<day>.<no>). As there are more slots than drivers, a list of unused slots is also maintained.

Initialisation When initialising the population, each solution has a random unused training slot allocated to each driver. Algorithm 2 illustrates the means by which each individual is initialised. The MAX_TRIES variable was set to 150 in order to ensure that a reasonable proportion of training slots were potentially tried for each driver. For drivers whose license is due to expire (Line 6), the algorithm is biased towards finding a slot that allows training before their expiry date (Line 7). For the remaining drivers (Line 11) the selection is biased towards finding a training slot that coincides with an early shift pattern.

Operators When a new *child* solution is created, initially no training slots are allocated. Each driver d is then considered in turn, a parent is selected at random, and an attempt is made to allocate the slot associated with d in that parent. If the slot cannot be used (as it has already been allocated within the child) then the slot associated with d on the other parent is tried. If a slot from neither parent can be used, the d is allocated a slot chosen at random from the list of unused slots.

Two mutation operators are used:

- Select two drivers d_1 and d_2 at random from within the chromosome, swap the training slots allocated to d_1 and d_2.
- Select a driver d at random, move the training slot allocated to d to unused list. Select a slot from the unused list at random and allocate that slot to d.

Algorithm 2 Initialise Individual

```
1: $d = 0$
2: while d < drivers.length do
3:     $tries = 0$
4:     while $tries < MAX\_TRIES$ do
5:         $slot = getRandFreeSlot()$
6:         if $driver.expiresCurrentYear$ then
7:             if $slot.week < driver.expiry()$ then
8:                 $driver.trainingSlot = s$
9:                 $freeSlots.remove(s)$
10:                $tries = MAX\_TRIES$
11:            else
12:                if $driver.getShift(slot.week) == early)$ then
13:                    $driver.trainingSlot = s$
14:                    $freeSlots.remove(s)$
15:                    $tries = MAX\_TRIES$
```

Objective Function A penalty-based fitness function is used. The penalty weights used can be found in Table 5 (these values were determined by empirical investigation). The fitness function may be divided into two sub functions: *fitness-base* and *fitness-dsl* (see Section 5).

Fitness-base is hard-coded at design time and incorporates those constraints identified during the analysis and development stage undertaken with the partner. The base fitness function examines the solution for violations of constraints 1-3 (see Table 5). *Fitness-dsl* is used to allow the end-user to specify additional constraints using a custom DSL (see Section 5).

	Constraint	Penalty
1	Final training day after license has expired	15
2	Unbalanced training group (see Table 1)	5
3	Training scheduled during late shift.	5
4	Custom Constraint (low priority)	1
5	Custom Constraint (medium priority)	5
6	Custom Constraint (high priority)	10

Table 5. The penalties used within the fitness function. Note that constraints 1-3 are evaluated by the base fitness function and 4-6 are evaluated by the extended fitness function using. Each Basil statement is compiled into a custom constraint.

4.2 Initial Results

Table 6 shows the results obtained with *CPC-EA* on the five test instances. In each case the *CPC-EA* was run 10 times and the best result shown (best being defined as lowest fitness). Note that in these instances no extended fitness function was specified.

A small number of training slots violated the late shift constraint (Table 5 Item 3). Examination of the solutions suggested that in most cases this occurred as the driver affected had a license that expired early in the year and so the training had to take place within the first few weeks, even if that meant violating the late shift constraint. Figure 2 shows the total number of late shift violations by week. Note that the violations all occur before week 12 and that 60% of the violations occur in the first two weeks. It should also be noted that every one of the drivers whose training week clashed with a late shift had a license due to expire in the current year.

As we are carrying out EA runs that do not have a fixed number of evaluations, we should examine the relationship between performance (fitness) and the number of evaluations used. Figure 1 plots the fitness and total evaluations for all 40 initial runs of *CPC-EA*. The results of the Pearson coefficient suggest that there is a significant small relationship, examination of the plot shows that there are a number of runs where a smaller number of evaluations has been accompanied by a low fitness, thus justifying the practice of executing the algorithm 10 times and selecting the best result achieved.

Data Set	**Fitness**	**Constraint Violations**		
		Expired License	**Imbalanced Groups**	**Late Shifts**
801	60 (83.5)	0	0	6 (8.1)
480	50(71.5)	0	0	10(14.3)
665	40 (58.5)	0	0	8 (11.7)
135	30 (40.5)	0	0	6 (8.1)

Table 6. The initial results obtained when using *fitness-base* only on the test instances. Results shown are based on the best of 10 runs - the average being shown in parenthesis.

5 Extending the Fitness Function

5.1 Introduction

When using an Evolutionary Algorithm within an industrial environment, a hard-coded fitness function can present a major disadvantage. As business and operational needs change, the constraints on the problem under consideration may change, requiring the fitness function to be modified. Modifying the fitness function is difficult and potentially expensive, to address this, as discussed in Section 4.1 we divide the fitness function into two sub functions:

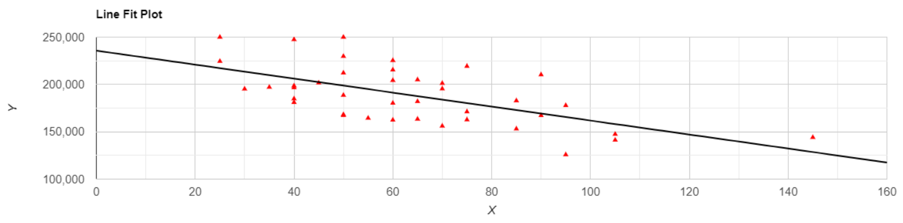

Fig. 1. A line fit plot for the fitness (x axis) versus evaluations y axis. The Pearson correlation coefficient returns a result of $r = -0.6058, p = 0.00003447$, which suggests a significant very small relationship between x and y.

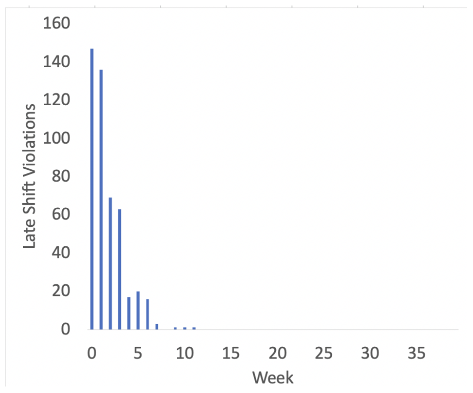

Fig. 2. The total number of late shift constraint violations found within the 400 solutions summarised in Table 6. In every case the driver with the constraint violation also had a license due to expire in the current year.

– *fitness-base* : This function evaluates a set of constraints that are hard-coded in Java, it is not intended to be modified by the end-user.
– *fitness-dsl* : This function evaluates a set of constraints specified using the DSL by the end-user. The constraints are compiled and evaluated at run-time.

The fitness value assigned to a candidate solution is the sum of the penalty values assigned by *fitness-base* and *fitness-dsl*. As the end users do not have software engineering experience it is not desirable to use an existing scripting language, we investigate the development of a domain specific language (DSL) named Basil that allows constraints to be specified. The DSL is based around entities within the problem domain that will be familiar to the end user, making it easier for them to use, it is only intended for the specification of constraints for this problem domain. Each constraint within Basil specifies whether a particular characteristic should not appear in the solution, in this manner the constraints specified using Basil are binary.

5.2 The Basil Language Syntax, Compilation and Evaluation

The Basil language is a DSL used to specify constraints which the user wishes to apply to the solution, as it is only used to specify constraints based around entities in the problem domain, it is not Turing complete (it has no branch or jump constructs).

A Basil script comprises a list of constraints and the priorities associated with these constraints. A constraint specified in Basil takes the basic form:

```
<entity> <condition> <time> <priority>
```

Each constraint specifies that a particular entity (driver or group of drivers) should be placed (or not placed) before, after or in a specific time (training week). Optionally, a priority may be assigned to the constraint, each priority level has a penalty value assigned to it (see Table 5 Items 4-6).

– **entity** The entities upon which constraints may be imposed are Drivers or Groups. These are denoted by use of the keywords *driver* or *group*, followed by the appropriate driver or group identifier.
– **condition** Each condition begins with the phrase *must be* (which can be negated with the phrase **must not be** followed by one of the keywords *before*, *after* or *in*
– **time** Times are specified using the keyword *week* followed by the week number.
– **priority** The optional priority may be set using the *priority* keyword followed by *high*, *medium* or *low*. Where a priority is not specified, the constraint is allocated a medium priority.

An example Basil script may be seen Algorithm 3.

Algorithm 3 An example of a Basil script. Line 1 is a comment, lines 2-5 describe constraints to be applied to the problem being solved.

```
1: #A set of test constraints
2: driver 123 must not be before week 23 with high priority
3: driver 226 must not be after week 23 with low priority
4: driver 1500 must not be in week 12 with medium priority
5: group depot1 must not be in week 35
```

5.3 Basil Execution

Basil is based around the concept of regular expression-based pattern matching, each Basil constraint being compiled into a regular expression. In order to evaluate a solution against a regular expression each solution is converted into what is termed an *intermediate* format (Figure 3) describing the allocation of training slots to drivers. Table 7 shows examples of Basil statements (constraints) and their resulting Regex expressions.

```
...
:ID: 1987:GR:DP1:WK:31:DY:2:DT:2:XW:33:FY:0
:ID: 1988:GR:DP1:WK:28:DY:2:DT:2:XW:06:FY:0
:ID: 1989:GR:DP1:WK:21:DY:1:DT:2:XW:37:FY:0
...
```

Fig. 3. An extract from the intermediate format. This format presents the solution in a manner that supports pattern matching via regular expressions. Each line describes the assignment of one driver to a training slot.

Basil Statement	Regex	Appears Flag	Priority Weight
driver 123 must not be before week 23 with low priority	:ID:123:GR:...:WK:23:DY:.:DT:.:XW:..:FY:.	false	low
group DP1 must not be in week 35	:ID:...:GR:DP1:WK:35:DY:..:DT:.:XW:..:FY:.	false	medium (default)
driver 456 must be before week 23 with high priority	:ID:456:GR:...:WK:23:DY:..:DT:.:XW:..:FY:.	true	high
driver 567 must be after week 33 with high priority	:ID:567:GR:...:WK:33:DY:..:DT:.:XW:..:FY:.	true	high

Table 7. Statements written in Basil are parsed and compiled into regular expressions which are then evaluated against a solution presented in the intermediate format (Figure 3. The appears flag specifies if the regex expression must appear in the solution or not. The priority weight field specifies the penalty weight to be associated with a violation.

5.4 Results with Basil

Basil scripts containing 10, 25, 50 and 100 constraints were generated at random. The generation of constraints at random simulates the arbitrary constraints which might stem from individual staff requests and organisational constraints. Some of these random constraints will conflict with each other or with base constraints (Table 5 Items 1-3). Our interest is not in avoiding this conflict but in managing it.

The results obtained may be seen in Table 8. The reader should note that the best solutions never break the license expiry constraint: this is very desirable given the importance to the business of ensuring that drivers' licenses are not allowed to expire. If we explore the relationship between the fitness of the best solution found (over 10 runs) and the number of custom constraints, we find that there exists a significant large positive relationship (calculated using a Pearson Correlation Coefficient where r=0.5651 and p = 0009). This is as we might expect, adding more constraints results in a reduction in solution quality.

As we are examining an industrial application, we should examine the effects of adding the additional constraints and the overhead of evaluating them. We are not concerned with the overall run-time required, the user can adjust the $TIME\_OUT$ property to find an appropriate balance between the time they are willing to wait and the quality of the resulting solution. In this section we are concerned with the general effect of adding numbers of additional constraints into the fitness function via Basil and the increased time taken to evaluate these constraints. Figure 4 shows the average time ev (milliseconds) to evaluate 2,000 individuals we are concerned with the trend in ev, as the number constraints is increased. Figure 4 suggests that the increase in evaluation time is super linear. The initial system was implemented in Java and executed on a MacBook based round the Apple M1 CPU.

6 Discussion and Future Work

6.1 Conclusions

The principle contribution presented in this paper is the development of the Basil DSL and the mechanism by which constraints are evaluated at run time. The basic problem is not novel, nor is the algorithm used to solve it. The contribution of this paper lies the development and the use of the Basil DSL and the use of the intermediate representation and pattern matching (see section 5) to allow evaluation of constraints at run-time.

In addressing the first research question stated in the introduction, the results presented in Table 8 suggest that can be implemented within a DSL and evaluated at run time. Assessing whether the DSL is usable by a domain expert is more difficult proposition, informal discussions suggest that domain experts can utilise the BASIL language, future work will include a more formal evaluation of BASIL with regards to usability through a user study.

The mechanism described in Section 5 highlights the use of the intermediate representation and pattern matching as the means of addressing the second research question. The use of regular expressions allows and existing well-proven mechanism to be used, one which is implemented in most commonly used programming environments. This avoids the use of more complex solutions such as

DataSet	Custom Constraints	Fitness		Constraints			Custom Constraints		
				Shift Violation (late shift)	Expired License	Unbalanced Groups	Low	Medium	High
135	0	Best	55	8	0	0			
		Avg	88.5	14.1	0	0			
	5	Best	2934	13	0	109	4	0	234
		Avg	2663.4	15.9	0.1	109.4	4.3	0	230.7
	10	Best	943	11	0	0	878	0	0
		Avg	962.3	10.9	0	0	880.8	0	0
	25	Best	449	9	0	0	3	70	0
		Avg	485.2	14.5	0	1	3.4	77.7	0
	50	Best	3815	6	0	113	566	60	218
		Avg	3844.7	10.7	0	126.2	575.7	61.9	225.9
	100	Best	7311	19	0	145	445	775	204
		Avg	7353.4	23.1	0.2	138.3	448.4	778.2	210.9
801	0	Best	105	17	0	0			
		Avg	127.5	20.7	0	0			
	5	Best	125	16	0	0	35	0	0
		Avg	145.5	16.4	0	0	36.5	0	0
	10	Best	976	16	0	0	610	50	1
		Avg	973.5	18.6	0	0	611.2	53.2	1
	25	Best	880	17	0	17	25	16	50
		Avg	913.7	21.9	1.3	21.1	26.2	17.8	53.8
	50	Best	5896	20	0	177	5	144	412
		Avg	5914.7	22.8	0.4	179.4	6.2	145	414
	100	Best	1918	19	0	22	448	28	105
		Avg	1956.9	21.4	0.1	27.7	451.4	29.5	108.7
665	0	Best	65	10	0	0			
		Avg	84.5	13.6	0	0			
	5	Best	80	13	0	0	0	0	3
		Avg	100.6	14.3	0	0	0.1	3.1	0
	10	Best	131	12	0	0	1	12	0
		Avg	156.3	15.7	0	0	1.8	12.1	0
	25	Best	954	11	0	61	22	2	54
		Avg	977	16	0	61.4	23.5	2	54.1
	50	Best	239	11	0	0	39	19	2
		Avg	261.6	13.4	0.1	0	44.6	20.1	2.8
	100	Best	8234	13	0	0	506	23	752
		Avg	8250.9	15.8	0	0	507.4	23.9	753.2
480	0	Best	75	11	0	0			
		Avg	98.5	15.3	0	0			
	5	Best	644	10	0	3	19	107	1
		Avg	670.6	10	0	1.7	20.1	108.3	1
	10	Best	535	11	0	0	455	1	1
		Avg	564.8	14.7	0	0	0	1.1	1.2
	25	Best	565	15	0	8	35	1	33
		Avg	581.1	18.9	0	10.5	38.1	1	37.1
	50	Best	1332	15	0	0	966	7	21
		Avg	1355.3	19.8	0.5	0	966.8	7.6	21.5
	100	Best	3807	15	0	38	10	392	147
		Avg	3838.2	18.9	0	44.7	10.7	392.4	152.7

Table 8. Results obtained using up to 100 randomly specified Basil constraints.

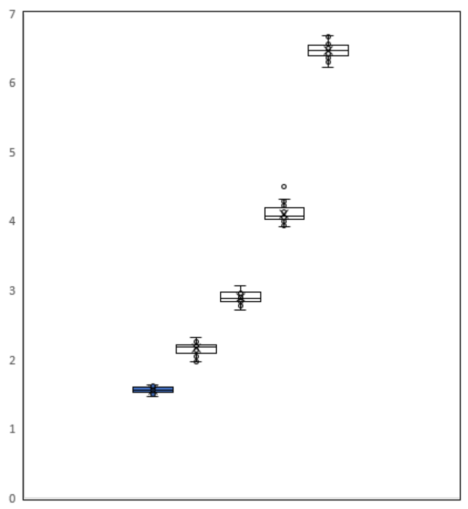

Fig. 4. Milliseconds per evaluation on the results obtained in Table 8 the box plots show evaluation times for 0, 10, 25, 50 and 100 custom constraints respectively. Note the lack of overlap between the box plots, also note that the trend in the evaluation time versus quantities of custom constraint is super-linear.

full compilation of the custom constraints into the main code base, a procedure which also has potential security and integrity issues.

If we consider the last question, figure 4 shows the increase in evaluation time as the number of custom constraints increases. We note that further work may be required on the execution mechanism for Basil as scripts of more than 100 constraints are not viable in terms of execution. This could be partially negated by executing the software on more powerful hardware and examining the implementation of the regex pattern matcher.

6.2 Future Work

This paper has laid a solid foundation for future work on the problem of incorporating, custom constraints into a solver in a manner that is appropriate for non-technical users. As well as more technical work in he implementation, future work will also include the integration of Natural Language Processing into Basil to allow constraints to be expressed in natural language. In a problem such as this where there are many stakeholders (e.g., over 2,000 drivers) the ability for them to articulate their constraints directly to the system would be a very powerful feature.

Acknowledgements The authors are indebted to management of the industrial partner for their time in explaining the problem and the feedback given on the work undertaken.

References

1. A tabu search algorithm with controlled randomization for constructing feasible university course timetables. Computers and Operations Research **123**, 105007 (2020). https://doi.org/https://doi.org/10.1016/j.cor.2020.105007
2. Abdelghany, M., Yahia, Z., Eltawil, A.B.: A new two-stage variable neighborhood search algorithm for the nurse rostering problem. RAIRO - Operations Research **55**(2), 673–687 (2021). https://doi.org/10.1051/ro/2021027
3. Burke, E.K., Curtois, T., Qu, R., Vanden Berghe, G.: A time predefined variable depth search for nurse rostering. INFORMS J. on Computing **25**(3), 411–419 (jul 2013). https://doi.org/10.1287/ijoc.1120.0510, https://doi.org/10.1287/ijoc.1120.0510
4. Kent, E., Atkin, J.A.D., Qu, R.: Vehicle routing in a forestry commissioning operation using ant colony optimisation. In: Dediu, A.H., Lozano, M., Martín-Vide, C. (eds.) Theory and Practice of Natural Computing. pp. 95–106. Springer International Publishing, Cham (2014)
5. Kittel, F., Enenkel, J., Guckert, M., Holznigenkemper, J., Urquhart, N.: Optimisation algorithms for parallel machine scheduling problems with setup times. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. p. 131–132. GECCO '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3449726.3459487, https://doi.org/10.1145/3449726.3459487
6. Kondratenko, Y., Kondratenko, G., Sidenko, I., Taranov, M.: Fuzzy and evolutionary algorithms for transport logistics under uncertainty. In: Kahraman, C., Cevik Onar, S., Oztaysi, B., Sari, I.U., Cebi, S., Tolga, A.C. (eds.) Intelligent and Fuzzy Techniques: Smart and Innovative Solutions. pp. 1456–1463. Springer International Publishing, Cham (2021)
7. Ngoo, C.M., Goh, S.L., Sze, S.N., Sabar, N.R., Abdullah, S., Kendall, G.: A survey of the nurse rostering solution methodologies: The state-of-the-art and emerging trends. IEEE Access **10**, 56504–56524 (2022). https://doi.org/10.1109/access.2022.3177280
8. Regnell, B., Kuchcinski, K.: A scala embedded dsl for combinatorial optimization in software requirements engineering. First Workshop on Domain Specific Languages in Combinatorial Optimization p. 19–34 (Sep 2013)
9. Service, G.D.: Driver cpc training for qualified drivers (Mar 2021), https://www.gov.uk/driver-cpc-training
10. Si Ying, P., Mohd Yusoh, Z.I.: Staff scheduling for a courier distribution centre using evolutionary algorithm. Indonesian Journal of Electrical Engineering and Computer Science **27**(2), 1043 (2022). https://doi.org/10.11591/ijeecs.v27.i2.pp1043-1050
11. Siddiqui, A.W., Arshad Raza, S.: A general ontological timetabling-model driven metaheuristics approach based on elite solutions. Expert Systems with Applications **170**, 114268 (2021). https://doi.org/https://doi.org/10.1016/j.eswa.2020.114268, https://www.sciencedirect.com/science/article/pii/S0957417420309799
12. University, M.: Minizinc constraint modelling language (2020), https://www.minizinc.org/