Designing an Application Program Interface to Efficiently Handle Optimisation Problem Data

Rodrigo Lankaites Pinheiro, Dario Landa-Silva, Rong Qu, and Edson Yanaga

***Abstract*—Literature presents many APIs and frameworks focusing on providing state of the art algorithms and solving techniques for optimisation problems. The same can not be said about APIs and frameworks focused on problem data itself and the reason is simple: due to the peculiarities and details of each variant of a problem, it is virtually impossible to provide general tools that is broad enough to be useful for many people. However there are benefits of employing such APIs, specially in a R&D environment where we have heterogeneous teams of researchers and developers. Therefore in this work we propose a design methodology for tailored optimisation problems based on a data-centric development framework. Our methodology relies on a data parser to handle the problem specification files and on a set of efficient data structures to handle the information on memory in a way that it is intuitive for researchers and efficient for the solving algorithms. Additionally, we bring three design patterns aimed to improve the performance of the API and techniques to improve the memory access by the user application. Also, we present the concepts of a Solution Builder that can manage solutions objects in memory better than built-in garbage collectors. Finally, we describe the positive results of employing a tailored API to a project involving the development of optimisation solutions for workforce scheduling and routing problems.**

Keywords*—*optimisation problems, data API, efficient data structures, research and development projects

I. Introduction

The research on decision support systems (*DSS*) is a multidisciplinary field, widely disseminated on the literature, that saw great improvements in the past few decades (1). One specific type of DSS include systems focused on solving optimisation problems, such as workforce scheduling (2), vehicle routing (3) and many other industry applications. In this context, literature presents many application program interfaces (API) and frameworks to help researchers and practitioners to apply state of the art solving techniques to optimisation problems, such as ParadisEO (4), jMetal (5) and Opt4J (6).

These tools and APIs have in common the fact that they provide flexible implementations of state of the art algorithms that can be adapted to most optimisation problems, given that the objective-function is known. However, on an applied research and development (R&D) environment, understanding the problem can be an important asset to solve it, because with a comprehensive understanding of the problem one can achieve improved tailored solutions. Thus, in this situation, having an API to handle the problem itself, including data, features, constraints and objective function can be beneficial to the project.

Pinheiro and Landa-Silva (7) raised the benefits of having a R&D methodology centred on the problem data itself. The benefits include the obtention of a greater understanding of the problem, a higher integration between researchers and practitioners and an improved environment for the development of the solving techniques, where multiple researchers from different backgrounds can efficiently work as a team. In that context, including a common API to handle the problem data can further extend the usefulness of their proposed framework as it can increase the productivity of researchers and developers by simplifying the data access, avoiding rework, improving computational performance and promoting standard solution comparison measures.

Nonetheless, there is a reason on why APIs and frameworks focused on problems are not common: they highly depend on the problem being tackled and a single optimisation problem possess many variants, making it unfeasible to define a unified model that covers all possible versions. Therefore in this work we present a set of guidelines and instructions to design a tailored API that can be adapted to any optimisation problem emerging from a R&D project. Our design follows the framework proposed by Pinheiro and Landa-Silva (7), hence in the core of the API is the data model represented by a set of XML files. Our proposed design is composed by three components. The first is a parser for the files that is able to read from and write to the modelled format. The second are the data structures containing the relevant optimisation data kept on memory. These data structures are designed in a way to maximise performance to access the data during the optimisation process. Also, we bring the Object Pool, Dirty Flag and the Data Locality design patterns and techniques to reduce CPU cache misses in order to improve the API computational efficiency. Lastly we propose a feature called Solution Builder which centralises the objective-function and provides a repository for solution objects that recycles solution objects and aims to minimise the interference of the built-in garbage collector and memory fragmentation, hence increasing computational performance.

Finally we present the results of the application of this API to an ongoing R&D project. We also present an empirical study about the efficiency of applying the Solution Builder in detriment of relying on the garbage collector of modern languages.

The remaining of this paper is structured as follows. Section II outlines the Workforce Scheduling and Routing Problems Project which is used to illustrate the application of the pro-

R. L. Pinheiro, D. Landa-Silva and Rong Qu are with the ASAP Group, School of Computer Science, University of Nottingham, UK e-mail: {rodrigo.lankaitespinheiro,dario.landasilva,rong.qu}@nottingham.ac.uk.

E. Yanaga is with Unicesumar.

posed API. Section III presents the guidelines and instructions on designing the API. Section IV presents the results obtained and section V concludes this work.

II. The WSRP Project and Related Work

In this work we illustrate the design of the proposed API using a Workforce Scheduling and Routing Problem (*WSRP*). In general terms the WSRP is a class of problems where a set of workers (nurses, doctors, technicians, security personnel, etc.), each one possessing a set of skills, must perform a set of visits. Each visit may be located in different geographical locations, requires a set of skills and must be attended at a specified time frame. Working regulations such as maximum working hours and contractual limitations must be attended. This definition is quite general and many problems can be considered WSRPs.

This work considers a variant of this problem, the home healthcare scheduling and routing problem. Workers in this scenario are nurses, doctors and care workers, while the visits represent performing activities on patients who are in their houses. In this problem, the main objective of the optimisation is to minimise distances and costs while maximising worker and client preferences satisfaction and avoiding (if possible) the violation of area and time availabilities. For more information regarding the WSRP we recommend the works of Castillo-Salazar et al. (8, 9, 10) and Laesanklang et al. (11).

We are engaged in a R&D project in collaboration with an industrial partner in order to develop the optimisation engine for tackling large WSRP scenarios. The existing information system collects all the problem-related data and provides an interface to assist human decision makers in the process of assigning workers to visits. We are in charge of developing the decision support module that couples well with the information management system being developed and maintained by the industrial partner. Hence the proposed API is being used by the research team and later it integrates to the current system.

Many APIs and implementations available in the literature focus on the solving techniques. We can highlight the ParadisEO (4), the jMetal (5) and the Opt4J (6). They are all frameworks that provide several solving algorithms for both single and multiobjective problems. They all have in common the fact that they are built around the solving methods and they are flexible enough to be applied for many optimisation problems.

In the literature we can also find frameworks and APIs with a stronger focus on the problems being solved rather than on the solving techniques.

- Matias et al. (12) and Mestre et al. (13) propose a web-based Java API to solve nonlinear optimisation problems. The API incorporates a set of constrained and unconstrained problems and gives the user the possibility to define his own problems with custom-made objective functions. However, as with the many works that focus on the solving techniques, defining exclusively the objective-function may be too restricting to the research of the solver. Hence, our API could be integrated with this or any framework focused on the solving algorithm as we focus on how to efficiently access the data and build solution objects.
- In his work, Huang (14) proposes a new API for evaluating functions and specifying optimisation problems at runtime. They propose a Fortran interface FEFAR for the evaluation of objective functions and a new definition language LEFAR for the specification of objective problems at runtime. Conceptually we differ from them as we are not proposing a general API, but instead conceptualising the design of a tailored API that can help on the research and development of optimisation solutions.
- Pinheiro and Landa-Silva (7) propose a framework to aid on the development and integration of optimisation-based enterprise solutions in a collaborative R&D environment. The framework is divided into three components, namely a data model that serves as a layer between practitioners and researchers, a data extractor that can filter and format the data contained in the information management system to the modelled format and a visualisation platform to help researchers to fairly compare and visualise solutions coming from different solving techniques. In their work they mention the importance of an API that extends the usefulness of the data model, in this work we expand that concept and describe how to design and implement key components of such API.

III. API for optimisation Problems Data

The proposed API is composed of three main components that allow the user to decode the data files of a problem scenario, to load the data into efficient and easy-to-access data structures and to build and evaluate solutions in a straightforward and efficient way. Figure 1 presents an overview of the API components.

These features facilitate the development of both experimental solving techniques and final release versions. Additionally they provide a reliable way for the algorithms to query the data and to compare solutions from different approaches. We describe next the first feature of the proposed API, the data parser.

A. XML Data Parser

Pinheiro and Landa-Silva (7) in their work proposed the use of a data model to represent the optimisation problem features and data. In a collaborative environment, where practitioners work with the academia to develop a decision support system, it is common that an information system already exists and that the decision support system is a feature of the main software. In that context, a data model to represent the problem was proposed to improve several aspects of the project:

- Improved definition of the problem: the process of defining the data model can promote discussions and deeper understanding of the problem by both academics and practitioners. While practitioners have a business vision of the problem, academics are often biased towards technical content found in the literature, hence

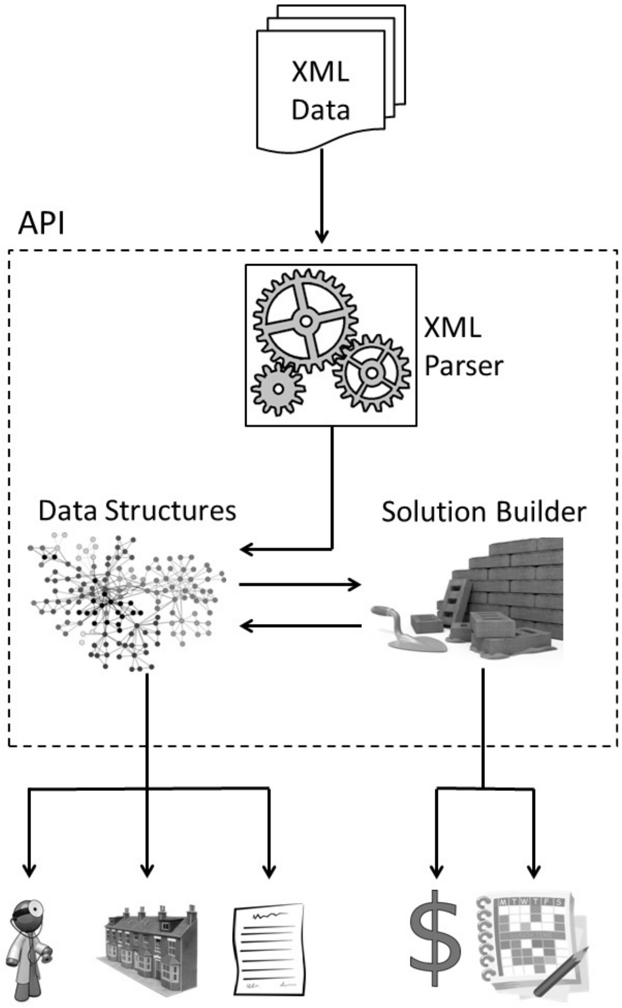


Fig. 1: Overview of the API.

divergences may arise. A clear definition of the model can help spot such differences and the establishment of a common ground.

- Development independence: in the aforementioned scenario, we assume that the R&D team, mostly composed by academics, is not the same team that develop the main information system. Hence, after setting the data model, a team will require less from the other as they will both be dealing with the data model, hence it virtually represents a layer in which both teams can rely.
- Easy integration: having a data model early in the project helps to integrate the final solution to the current information system. This happens because all data will be translated to and from the data model, hence both R&D team and main development team will be making use of the model when adding the optimisation module to the main system. Thus, the development is modularised and the main system and the decision support module only communicate through the data model, which is well known by both teams.

Figure 2 present the overall idea of the data model. We see that the problem features, represented by the mathematical formulation of the optimisation problem, relates to the XML files in a manner that it is intuitive for human beings to understand (grouping related features in separated files).

Following this data modelling concept, the first component of the API is a parser to read the files from the data model and build the data structures. The parser is also responsible for converting a solution of the optimisation problem into the XML data file. Additionally, the parser must be implemented in such a way to easily accommodate extensions or updates in the data model.

For that purpose we employed a serialisation library. Since we are using Java, we employed the XStream Java library (15), however most high level languages have XML serialisation mechanisms available. The advantage of such approach is that we can create a set of classes that corresponds immediately to the modelled data, hence the serialisation library can handle all the file parsing. This is easy to develop and do not require much programming time, however it is likely that the objects in memory are not best suited for performance or for intuitive access because the serialisation mechanism often required intermediate classes and public access to attributes. Hence the parser is used exclusively to translate the information from and to the files. For efficient and easy access to the data we need another set of data structures.

B. Internal Data Structures

The second component of the API is composed by the data structures to hold the problem-related data. It is important to emphasise that while the aforementioned data model is intended to be a clear representation of the optimisation problem, the internal data structures must be efficient for access during the optimisation process.

Therefore we must ensure that the operations invoked during the optimisation are performed on constant ($O(1)$) time when possible. Additionally, the API should be flexible enough to easily accommodate different solving techniques, as we are not only concerned about the final product, but with the entire process of developing the R&D project. Hence, in our work we divided the API in arrays, following the data model orientation.

We defined arrays of workers, tasks, areas and contracts. The advantage of using arrays is that the the random access using indexes are very efficient ($O(1)$). The disadvantage is that to load the data we must first assess its size in order to allocate the right length for the arrays (or use dynamic arrays structures which could also hinder the performance if the pre-allocated size is not large enough). To increase the loading performance we added a new XML file to the model, called 'metadata.xml' that accommodates several information and statistics of the problem instance, such as number of workers, tasks, the date format used in the files etc.

Using arrays may be sufficient to the decision support system, as it allows fast random access and quick interaction through the elements. However it may be a problem to the

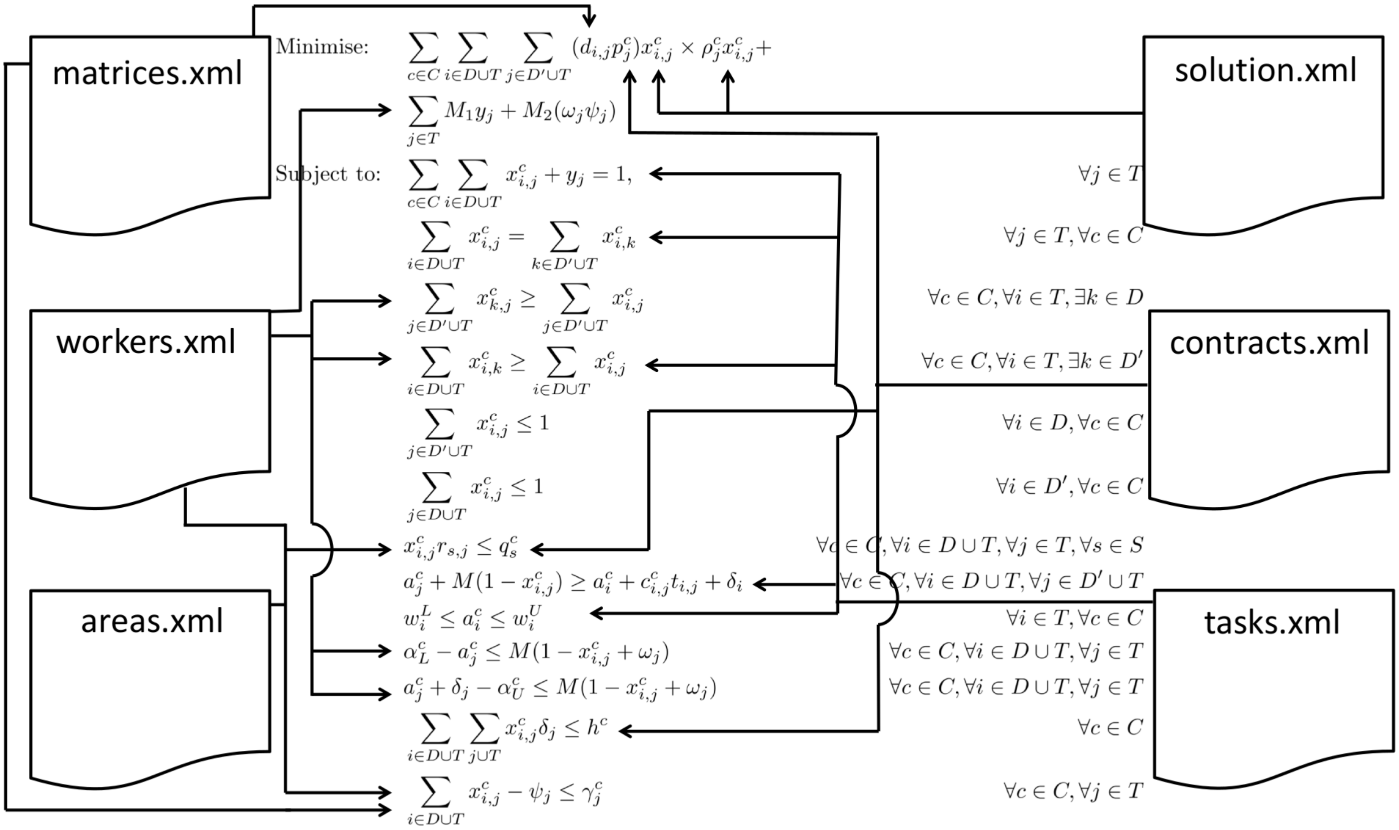

matrices.xml
workers.xml
areas.xml
solution.xml
contracts.xml
tasks.xml
Minimise: $\sum_{c\in C}\sum_{i\in D\cup T}\sum_{j\in D'\cup T}(d_{i,j}p_j^c)x_{i,j}^c \times \rho_j^c x_{i,j}^c +$
$\sum_{j\in T} M_1 y_j + M_2(\omega_j \psi_j)$
Subject to: $\sum_{c\in C}\sum_{i\in D\cup T} x_{i,j}^c + y_j = 1,$ $\forall j \in T$
$\sum_{i\in D\cup T} x_{i,j}^c = \sum_{k\in D'\cup T} x_{i,k}^c$ $\forall j \in T, \forall c \in C$
$\sum_{j\in D'\cup T} x_{k,j}^c \geq \sum_{j\in D'\cup T} x_{i,j}^c$ $\forall c \in C, \forall i \in T, \exists k \in D$
$\sum_{i\in D\cup T} x_{i,k}^c \geq \sum_{i\in D\cup T} x_{i,j}^c$ $\forall c \in C, \forall i \in T, \exists k \in D'$
$\sum_{j\in D'\cup T} x_{i,j}^c \leq 1$ $\forall i \in D, \forall c \in C$
$\sum_{j\in D\cup T} x_{i,j}^c \leq 1$ $\forall i \in D', \forall c \in C$
$x_{i,j}^c r_{s,j} \leq q_s^c$ $\forall c \in C, \forall i \in D \cup T, \forall j \in T, \forall s \in S$
$a_j^c + M(1 - x_{i,j}^c) \geq a_i^c + c_{i,j}^c t_{i,j} + \delta_i$ $\forall c \in C, \forall i \in D \cup T, \forall j \in D' \cup T$
$w_i^L \leq a_i^c \leq w_i^U$ $\forall i \in T, \forall c \in C$
$\alpha_L^c - a_j^c \leq M(1 - x_{i,j}^c + \omega_j)$ $\forall c \in C, \forall i \in D \cup T, \forall j \in T$
$a_j^c + \delta_j - \alpha_U^c \leq M(1 - x_{i,j}^c + \omega_j)$ $\forall c \in C, \forall i \in D \cup T, \forall j \in T$
$\sum_{i\in D\cup T}\sum_{j\cup T} x_{i,j}^c \delta_j \leq h^c$ $\forall c \in C$
$\sum_{i\in D\cup T} x_{i,j}^c - \psi_j \leq \gamma_j^c$ $\forall c \in C, \forall j \in T$

Fig. 2: Representation of the XML Model.

main system, as it often handles elements using its database identification number, which are stored in the XML, but are not consistent with the index of the arrays. To solve this matter we employ a second data structure, a hash table, linking the identification numbers of the database entries to its respective objects. In order to provide the best usability with both data structures, we encapsulated both the hash table and the arrays, for each type of objects, into a single class representing the set of elements.

Finally, to improve the usability of the API, we define a naming convention to make it clear regarding the performance of the operations. All methods starting with the words ’get’, ’is’ and ’has’ are guaranteed to perform in $O(1)$ time. All methods starting with the word ’calculate’ are guaranteed to perform in $O(n^k)$ time.

Figure 3 presents a class diagram of the data structures. For simplicity we included only the main class that defines the optimisation problem and the classes that defines the tasks and the set of tasks. The main class, *WorkforceSchedulingAndRoutingProblem*, is composed by the sets of elements included in a problem instance, namely areas, tasks, human resources and contracts. This class provides an interface such that the user can retrieve each set and its elements. Also, this class allows the user to obtain the Solution Builder and the objective function, explained in the subsequent section. Note that the *calculateMinimumNumberOfAssignments* method, as aforementioned starts with the ’calculate’ word hence it performs because it performs in $O(n)$ time while all ’get’ methods performs in $O(1)$.

The *Tasks*, *Areas*, *HumanResources* and *Contracts* classes contains both the arrays of elements and the hash table linked by each element’s identification number. Hence, when using these classes it is possible to interact through all elements or retrieve a specific one given its identification number or index, as we can find in the *Tasks* class. We see that from this class it is possible to identify an ordered list of tasks *ls* representing the array and the hash table *hashTable* containing the mapping of identification numbers. Finally, the class *Task* contains all the methods to access the data from a single task plus some useful operations, such as *isTimeConflictingWith* which checks if a second given task conflicts in time with the current task (hence they can not be performed by the same worker).

1) Code Optimisation: McShaffry (16) evaluates that the performance of an application can be influenced by the data structures employed and by how the data itself is organised and accessed by the code. Additionally, a good use of the processor’s internal cache can potentially increase performance by up to 50 times (17). Hence we now present some techniques found in the literature that can improve the performance of the data access and subsequently of the algorithms that make use of the API.

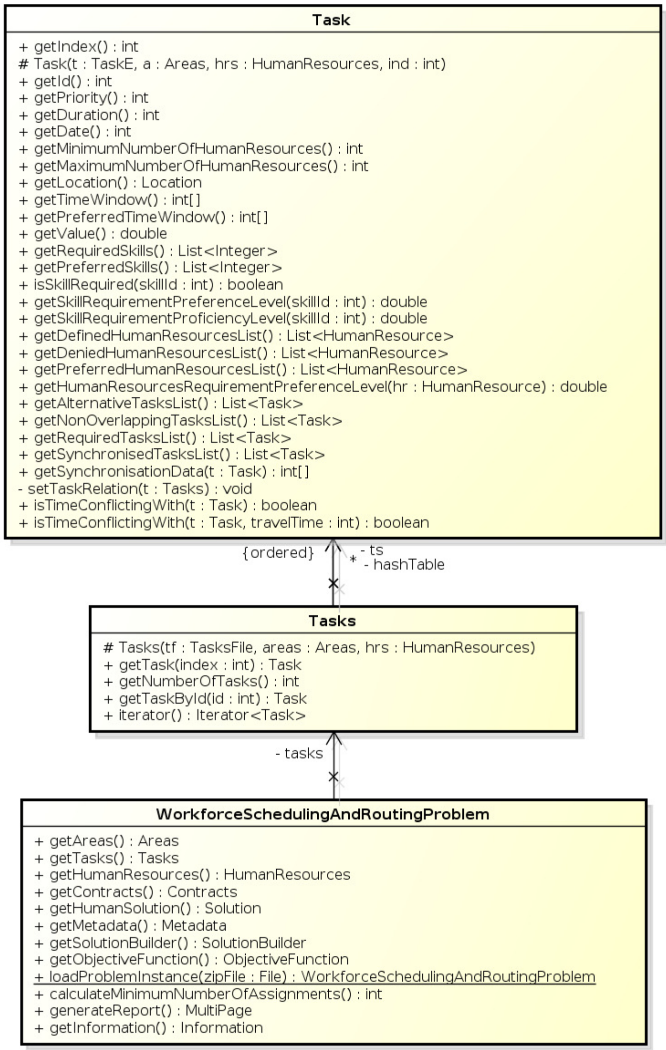

Task
+ getIndex() : int
# Task(t : TaskE, a : Areas, hrs : HumanResources, ind : int)
+ getId() : int
+ getPriority() : int
+ getDuration() : int
+ getDate() : int
+ getMinimumNumberOfHumanResources() : int
+ getMaximumNumberOfHumanResources() : int
+ getLocation() : Location
+ getTimeWindow() : int[]
+ getPreferredTimeWindow() : int[]
+ getValue() : double
+ getRequiredSkills() : List<Integer>
+ getPreferredSkills() : List<Integer>
+ isSkillRequired(skillId : int) : boolean
+ getSkillRequirementPreferenceLevel(skillId : int) : double
+ getSkillRequirementProficiencyLevel(skillId : int) : double
+ getDefinedHumanResourcesList() : List<HumanResource>
+ getDeniedHumanResourcesList() : List<HumanResource>
+ getPreferredHumanResourcesList() : List<HumanResource>
+ getHumanResourcesRequirementPreferenceLevel(hr : HumanResource) : double
+ getAlternativeTasksList() : List<Task>
+ getNonOverlappingTasksList() : List<Task>
+ getRequiredTasksList() : List<Task>
+ getSynchronisedTasksList() : List<Task>
+ getSynchronisationData(t : Task) : int[]
- setTaskRelation(t : Tasks) : void
+ isTimeConflictingWith(t : Task) : boolean
+ isTimeConflictingWith(t : Task, travelTime : int) : boolean
{ordered}
- ts
*
- hashTable
Tasks
# Tasks(tf : TasksFile, areas : Areas, hrs : HumanResources)
+ getTask(index : int) : Task
+ getNumberOfTasks() : int
+ getTaskById(id : int) : Task
+ iterator() : Iterator<Task>
- tasks
WorkforceSchedulingAndRoutingProblem
+ getAreas() : Areas
+ getTasks() : Tasks
+ getHumanResources() : HumanResources
+ getContracts() : Contracts
+ getHumanSolution() : Solution
+ getMetadata() : Metadata
+ getSolutionBuilder() : SolutionBuilder
+ getObjectiveFunction() : ObjectiveFunction
+ loadProblemInstance(zipFile : File) : WorkforceSchedulingAndRoutingProblem
+ calculateMinimumNumberOfAssignments() : int
+ generateReport() : MultiPage
+ getInformation() : Information

Fig. 3: Class diagram for the main problem class and the tasks-related classes.

a) Data Locality.: One of the most overlooked way to gain (or lose) performance in an application is due to the internal processor's cache memory. Modern computers possess an internal processor memory (cache memory) that bridges the access to the main RAM memory in order to increase the system performance. In summary, when the application requires some data in the memory, the CPU loads an entire section of the main memory into the faster internal cache. When requiring the next data, it first checks if it is already in the cache. In case it is (*cache hit*), the access is very fast as the data is promptly available. In case it is not in the cache, we have a *cache miss* and a section of the memory containing the required data is loaded into the cache (18).

(17) proposes a design pattern denominated *data locality* that attempts to reduce the number of cache misses in the application. The design pattern consists of sacrificing some abstraction and object oriented concepts in order to better arrange the data inside of an object such that when accessing the object attributes the number of cache misses is minimised.

Take for instance the *Task* class in the sample API. If we sequentially define the variables for id, priority, duration and date in the class and we always read them in that order (let us say in the objective-function), we are promoting cache hits because they are likely to be loaded altogether into the cache.

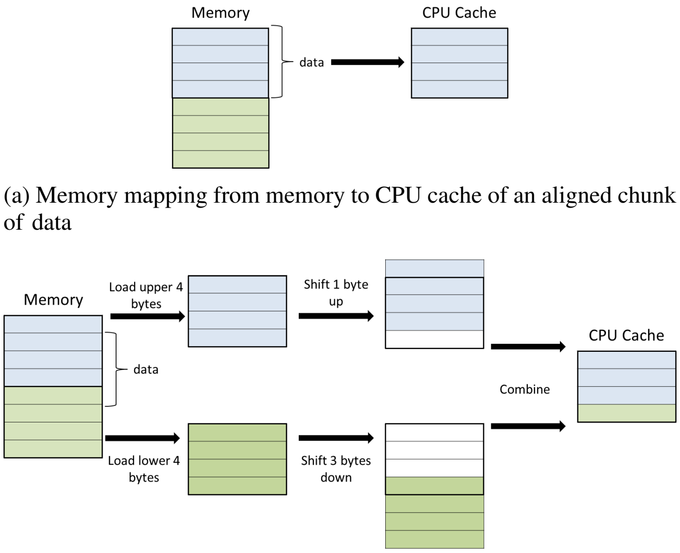
Another way to promote cache hits is to employ the use of arrays for homogeneous data. Suppose some heavy calculations requires the costs of all tasks to be processed. Instead of having an attribute ’cost’ inside the task, it is more efficient to have an array containing all the costs for all tasks such that when a calculation is performed, the sections loaded into memory will contain the costs for multiple consecutive tasks, hence effectively promoting multiple cache hits.

Another concern of this patter is with the use of getters and setters. Employing such methods is considered to be a good object oriented practice, as it promotes encapsulation, error control and readability. However it can cause cache misses because of the indirect referencing. Languages such as Java have in-line optimisations which can convert a setter or getter method into direct access to the attribute during the compilation of the code (19), hence setters and getters can be used without fear of hindering the performance. It is important to know beforehand the characteristics of the programming language used and the compiler employed before deciding whether make use of setters and getters or not.

b) Matrices Ordering.: Still focusing on optimising the cache memory access, another issue that may be overlooked is the ordering of matrices in the memory (16). Optimisation problems data often present data matrices. Accessing this data in the correct order to avoid cache misses can outcome a huge performance gain. Suppose the data from a matrix is stored in row order (i.e. [0,0],[0,1],[0,2] are respectively adjacent). If we follow the row ordering to access the data, entire sequential sections of the memory will be loaded into the cache, hence promoting cache hits. Now consider the opposite scenario, if we access the matrix in column order and given that a row is larger than the section that will be loaded into memory, we have the worst case scenario where every access will result in a cache miss.

Some languages such as C++ define its array in row major order while others such as some versions of PASCAL defines them in column order, hence it is important to be aware of how the chosen language allocates the arrays and matrices into the memory.

c) Memory Alignment.: (16) notes that the CPU reads and write memory-aligned data noticeably faster than misaligned data. A given data type is memory aligned if its starting address is evenly divisible by its size (in bytes). Al aligned chunk of data is promptly loaded into the cache while an unaligned chunk of data must be read in parts, then shifted to the target frame and then loaded. Figure 4a presents a diagram of a memory mapping of aligned data. As we can see, the mapping is direct meaning that the data is directly transferred from the RAM memory to the internal cache memory. Figure 4b presents the copy of misaligned data. We can observe that



(a) Memory mapping from memory to CPU cache of an aligned chunk of data

(b) Performance loss of a memory mapping of misaligned data.

Fig. 4: Comparison of cacheing aligned and misaligned data.

the CPU reads the two chunks containing the parts of the required data. It then shifts each chunk to select the wanted information and merge into a single chunk which is then copied to the cache. Clearly the second represent a much slower mechanism.

The best way to take advantage of this is to make sure that the internal data types, structures and classes of the API possess a number of bytes that is a power of 2. If a data type has less bytes, we could add a dummy variable with the required number of bytes in order to force it to have a desired value. Also, it is imperative to be aware of the overhead of space required by the programming language for classes and data structures before computing the total size.

C. The Solution Builder

The last component of the API is the Solution Builder (SB). The SB have two main roles:

1) Provide a standard mechanism to calculate the fitness (objective-value) of a given solution for the optimisation problem.
2) Provide an efficient way to handle solution objects in the memory.

The SB provides an interface for the user to build and assess a solution for a given problem instance. Once the problem is loaded into a *WorkforceSchedulingAndRoutingProblem* object, the user can invoke the SB to create a new solution object. A new empty solution is created and an identification number is returned to the user. He/she then can use this number to access the solution and add new assignments and evaluate the solution according to multiple criteria (preferences, objectives or constraints). The user can also invoke the objective function to evaluate the solution. Figure 5 presents a schematic for the SB component.

1) Centralised Evaluation Mechanism: Attached to the SB is the objective function. Pinheiro and Landa-Silva (7) mentioned in their work the importance of a mechanism to ensure the fairness in the comparison of results between multiple techniques implemented by different researchers. Following that concept, having a centralised objective-function in which everyone can rely is beneficial to the team and since the API will be used by everyone, having the evaluation mechanism integrated helps to avoid re-work and to keep consistency.

In that context the weights of the objective function are initialised with standard values, however the SB allows the user to set them according to his needs. The user then can evaluate specific aspects of the problem (total distance, total costs, constraints violations, preferences, etc) or obtain the overall fitness of a given solution.

We argue that it is important to have a centralised evaluation mechanism because often in real world problems the calculations of the objective function are complex and involve several computational procedures. Therefore, to compare solutions using multiple implementations is tricky as there may be inconsistencies between the algorithms. Also, because the results are likely to be different between distinct solving techniques, to spot the inconsistencies may be hard, hence having everyone making use of the same mechanism helps avoiding this problem. Additionally, the fact that multiple people are using the code helps on the identification of flaws and inaccuracies in the code.

2) Solution Dispenser: Modern programming languages, such as Java and C#, provides a convenient way to handle objects: a garbage collector. When the user doesn't need an object anymore, all he have to do is to get rid of the links and pointers to that object. At every given period of time, the garbage collector starts processing, seeking objects that are not linked by the user's program and freeing the memory. That can lead to two problems: the first is the fragmentation of the memory, which is a problem because the defragmentation process can be slow; the second is the extra processing time to seek, to free the memory and later to allocate new objects (20, 21).

Leaving the disposal of objects to the garbage collector can lead to a decrease in performance that, aside from being marginal on most applications, can have an unacceptable impact on optimisation algorithms. Hence the SB not only provide an user interface to create and handle solutions, but also internally implements an Object Pool design pattern (17) to recycle the objects. To do that we employ a factory object, implemented using the Factory design pattern (22) for easy creation of the objects. This factory is responsible for creating new solution objects.

When a new solution request is invoked, the factory seeks its internal solution repository (a list of disposed solutions). If there is a solution available in the repository, it retrieves it, clears the solution and returns it to the solution builder who will add the solution to the active solutions list. The user now has an empty solution that he can use. When the user does not need the solution anymore he can dispose the solution by invoking the specific method in the SB. The factory then receives the disposed solution and stores it in the list.

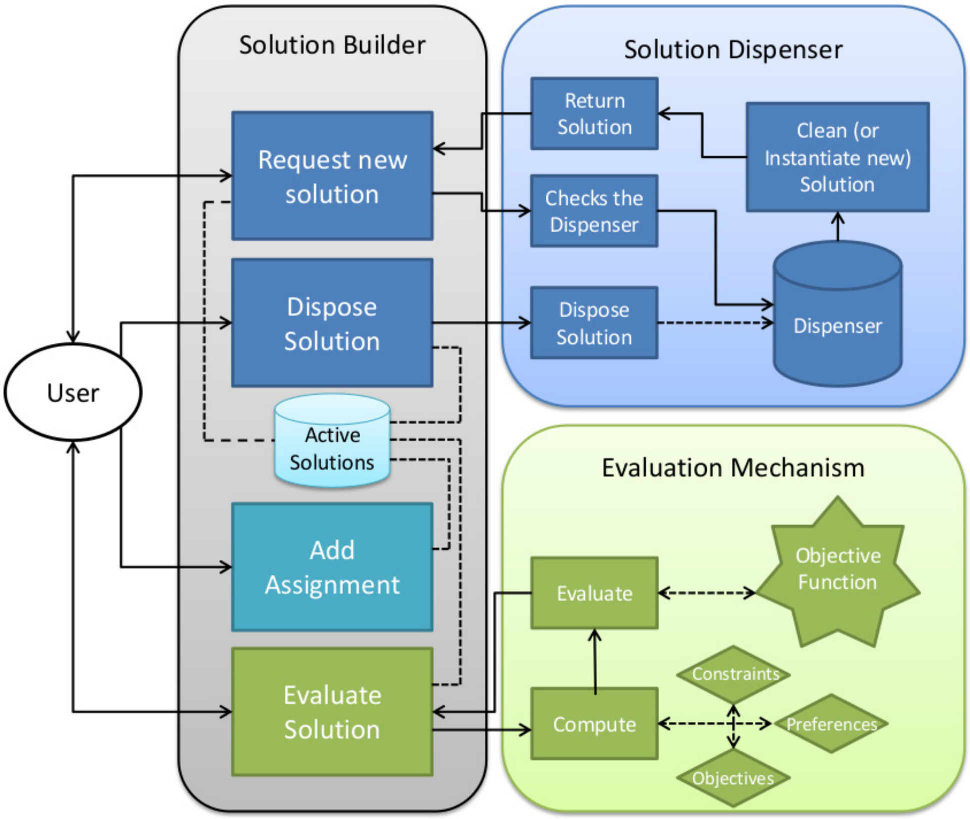


Fig. 5: Solution Builder schematics.

Potentially, the use of the the solution dispenser can provide significant performance gains. Take for instance a population-based algorithm that process one generation per second with a population of 100 individuals. That means 100 solutions being disposed per second. After ten minutes running, the algorithm will have disposed 60000 solutions, which potentially could fragment the memory and cause several garbage collection calls. Now, when using the dispenser, considering the worst case in that a new population is created before disposing the old one, we need 100 active solutions per population, totalling 200 total solutions active that will be recycled during the execution. Thus, in this hypothetical scenario we could have a decrease of 99.6% on the number of objects used.

3) Code Optimisation: The SB itself is a component to improve the efficiency of the API with the dispenser being a specialist implementation of an Object Pool design pattern. On the evaluation mechanism, however, programming techniques can be applied to improve the overall evaluation performance.

a) Dirty Flag.: (17) proposes a design pattern called *dirty flag*. Basically, it consists of a mechanism to avoid unnecessary recalculations when you have nested operations, usually on recursive calls. In the optimisation problems context, the objective function can be very costly and recursive calculations may appear. Additionally, algorithms may require specific parts of the objective function to be calculated in different points and in multiple times. This pattern consists of having a flag (boolean variable) to define if the state of an object has changed. If so, the values that relies on that object must be re-calculated. Thus, in a recursive operation where a value would be always calculated, now it will be calculated only if the flag is up. Essentially, after a solution is built, only one evaluation of its values are made. If an algorithm calls the evaluation (or partial evaluation) again on a same unchanged solution, baceuse the flag is down (no changes) the last calculated value is returned.

IV. Experiments and Results

We now present the results of the use of the API in the WSRP project. First we show the gains of employing the API in the project development then we perform a technical analysis on the Solution Builder component and show how much of a benefit it can represent.

A. Improvements on the R&D

b) Rework.: In our project we had multiple researchers with different background investigating the WSRP. The problem is complex and the data model, although easy to understand, is not so easy to decode and load into appropriate memory objects, due to its inherent complexity. Therefore, having a centralised API containing a parser and efficient internal data structures helped both teams to avoid performing rework.

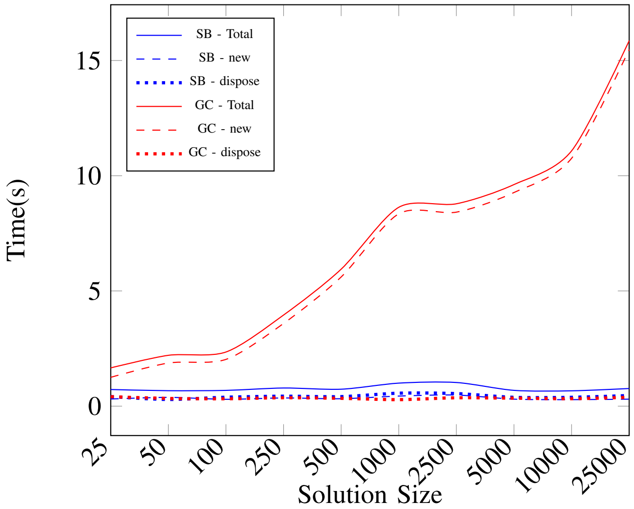


Fig. 6: Time comparison between the Solution Builder (SB) and the Java Garbage Collector (GC) to instantiate and dispose new solutions.

c) Time saving.: Additionally, using the API meant considerable time saving for us. New members joining the research team were able to quickly learn how to use the API and start implementing their methods right away, without hassle. Also, the company's development team were able to easily handle the data files by using the integrated parser.

d) Solver Efficiency.: When designing the API, specially the internal data structures, the entire team could assess its efficiency. Thus later, when using the API, everyone was confident that they were using the best option. This is crucial in a scenario where researchers have multiple backgrounds, as it was in our project, because the performance of some technique may be hindered by an inefficient implementation of some data structure, hence, having the best known data structures available for everyone helps to achieve improved efficiency in all solvers.

e) Consistency.: Pinheiro and Landa-Silva (7) in their work stated that one of the main concerns on such development environment is the consistency in the comparison of multiple techniques. We managed to spot problems on early stages of the project, where each researcher had its own fitness calculation mechanism. After the release of the API, the use of the integrated Solution Builder helped our team to assess and compare the developed solvers because it guaranteed that the fitness calculations were consistent between methods.

f) Error identification.: Having multiple people working using a single API helps to spot code errors and bugs faster than having each researcher to find errors alone on his own code. During the first months of our project we had the research team release several versions of the API until we obtained a stable version. This increased the confidence of the team on the tools at hand and helped us ensure that we had a reliable component to base our research.

B. *Solution Builder*

When presenting the Solution Builder component we argued that theoretically it can provide performance gains to the final algorithms, as it helps avoiding the disadvantages of the garbage collector. We now present an empirical analysis on this matter.

The solution builder is responsible for holding solution object for a given problem, hence a small problem using a large representation consumes more memory than a larger problem using a small representation. This is particularly true when comparing an integer array representation (an array the size of the number of tasks) and a binary array representation (a matrix workers × tasks). Therefore, to test the solution builder we used integer arrays varying from very small (25 elements) to very large (25000 elements) representing respectively a small problem using integer representation and a large problem using binary representation. Note that although the decision variables may be binary, for several reasons it not uncommon to find these arrays implemented using complex objects (5), hence it is reasonable to use integer variables instead of binary ones in our experiments.

We defined our experiments as follows: for each problem size we sequentially created and disposed 1,000,000 solution objects. For the experiments using the garbage collector, the dispose process merely unlink the objects to free them, while for the SB it calls the internal dispose process and clears the object data. We run each set up for five times and computed the average results. Additionally, to measure time and memory we used the integrated profiler available on Netbeans which can accurately measure the time spent on each method and the memory allocated during the execution of the application. The experiments were performed on a quad-core Intel i7 machine with 12GB memory on the Java platform. The main reason for choosing Java is that is a mature language, multi-platform and widely employed for optimisation problems with a large number of optimisation algorithms implemented and available

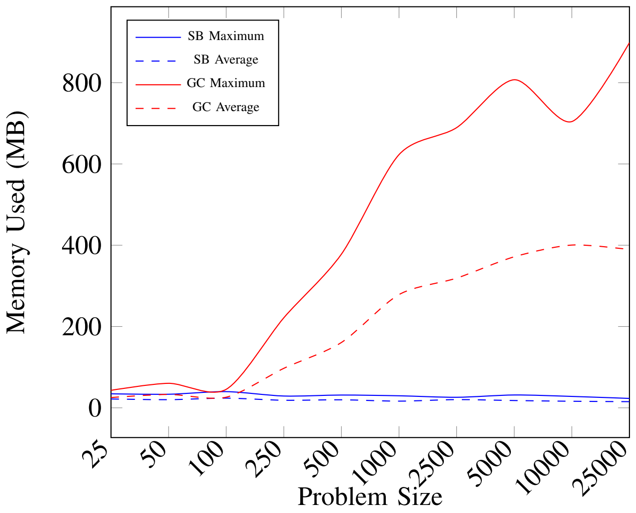
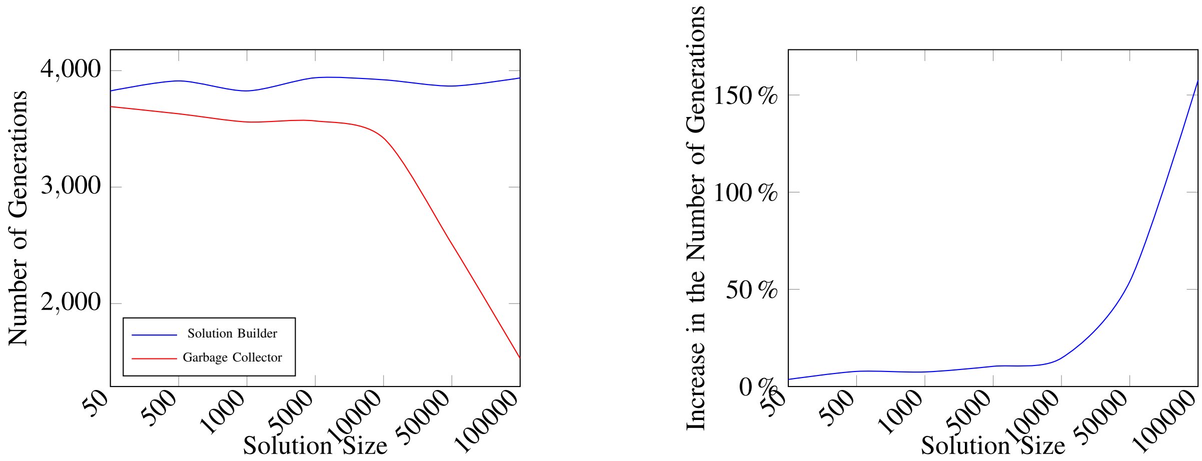


Fig. 7: Memory comparison between the Solution Builder (SB) and the Java Garbage Collector (GC).



(a) Number of generations of a genetic algorithm varying only the size of the solution representation.

(b) Increase in the number of generation when employing the Solution Builder in detriment of the built-in Garbage Collector.

Fig. 8: Comparison of the Solution Builder against the Garbage Collector on a Genetic Algorithm.

for public use (4, 5, 6).

Figure 6 presents the results for the time computations. The red lines represents the time spent in seconds on experiments using the Java garbage collector and the blue lines represent the time spent on experiments employing the solution builder. The solid line is the total time, the dashed line is the time used by the *new* method, which allocates a new solution object and the dotted line represents the *dispose* method. We can see that the time spent by the SB follows a constant trend throughout all problem sizes. This happens because both the *new* and *dispose* operations of the solution builder performs in constant time and since there are no objects free in the memory, the garbage collector does not need to activate. Regarding the tests using the garbage collector, we can see that the *dispose* operation performs in constant time, but the *new* method requires higher time proportional to the size of the solutions. We can clearly see how relying on the garbage collector to dispose and allocate new objects can hinder the performance of the application. Also, it is important to to notice that on our experiments we did not specify any parameters for the Java virtual machine, hence the experiments has as much memory as it was required. In a real-world environment that might not be the case. Many processes may be active in the machine and the memory might be limited, which would make the garbage collector to be active more often than it was on the tests, hence decreasing the performance even more.

In Figure 7 we have the results for the memory allocation measurement. The red lines represents the experiments using the garbage collector and the blue lines the solution builder. Also, the solid lines represents the maximum memory allocated in MB and the dashed lines the average memory allocated. Analogously to the previous chart, we can clearly see that the memory required by the SB, not surprisingly, is constant throughout all experiments. Although the size of the array changes on each experiment, only one object is allocated in memory during the runtime. However, when relying on the garbage collector we can see that it makes use of much more memory, which reinforce our previous statement that in a scenario where the memory is limited the garbage collector will have to be activated more often.

Finally, we employed the API designed for the WSRP Project to perform experiments using a Genetic Algorithm (GA) Goldberg (23) to solve real-world scenarios. In order to isolate the impact of employing the Solution Builder, we run the experiments on a single test instance varying only the size of the data structure used to represent a solution. Therefore all the experiments required the same computational effort regarding the genetic operators and solution evaluation. Also, we used fixed seeds for the random number generators in order to increase the fairness of the comparison. We performed eight runs of the GA, both for the SB and the GC, and computed the average number of generations after one minute of processing time. Figure 8 presents the results. In (a) we see the total number of generations for the GA as the size of the solution representation increases. It is evident that when employing the SB the average number of generation is roughly constant regarding the increase on the size of the representation. It is also noticeable that if the solution size is not large enough (in this case the equivalent of 10000 integer values), the advantage of using the SB is below 10% (b), but still remarkable. However, when the size of the representation increases, the benefit of employing the SB represented up to 150% raise in the number of generations.

Thus we can clearly see that by employing the solution builder we can achieve substantial improvements both in time and memory consumption, specially on larger problems or problems where the solution representation is larger. Also, the idea of the solution builder of recycling objects could be implemented in the solver algorithms themselves, specially on population-based algorithms (because of the high number of created and disposed solutions), to maintain their individuals pool.

V. Conclusion

Several APIs and frameworks providing optimisation algorithms are available in the literature. Their importance is unquestioned as both academia and industry can benefit of efficient implementation of state of the art optimisation methods. Having tailored APIs to handle the problem data can also be very beneficial in a R&D environment because it can save time by avoiding doing rework, it can decrease the complexity of accessing the data, increase computational performance and raise the reliability of the solutions obtained by different solving methods.

Because it is not possible to provide a general API that suits all problems, we instead provided in this work a set of guidelines and instructions to aid on the tailoring of an API to efficiently handle the optimisation problem data and to help increasing the performance of solving techniques. We first stipulated a file parser that can read the modelled format and load all pertinent information into memory. Then we defined an intuitive and efficient approach to store this information using efficient data structures that are clear and computationally efficient, hence it can improve on the research process and be applied to a final solver algorithm. We applied efficiency design patterns to our model in order to improve its effectiveness, such as the Data Locality, Dirty Flag and Object Pool design patterns. Also, we proposed efficient techniques to improve matrices efficiency and to avoid cache misses during the execution of the code. Finally we proposed a novel component called Solution Builder which centralises the objective-function, hence promoting fairness on the comparison of solutions arising from different solving techniques, and provides a solution repository that handles the memory allocation of solution objects in an improved way.

We discussed that having the API available for academics and practitioners greatly helped us on our project. We were able to minimise the rework done by multiple researchers from different backgrounds, we were able to reduce the time spent on implementations and the assessment of solving techniques started earlier. We were also able to increase the solvers efficiencies and to promote a consistent mean to compare solutions and we managed to improve the identification of glitches and bugs in the code, raising the reliability of the software being developed from early stages of the project. Also, we analysed the advantages of using the Solution Builder and presented the computational gains that can be obtained by employing such technique.

References

[1] D. Power, F. Burstein, and R. Sharda, "Reflections on the past and future of decision support systems: Perspective of eleven pioneers," in *Decision Support*, ser. Annals of Information Systems. Springer New York, 2011, vol. 14, pp. 25–48.

[2] M. Pinedo, *Scheduling: theory, algorithms, and systems*, ser. Prentice Hall international series in industrial and systems engineering. Prentice Hall, 2002. [Online]. Available: http://books.google.co.uk/books?id=FvVTAAAAMAAJ

[3] B. Golden, S. Raghavan, and E. Wasil, *The Vehicle Routing Problem: Latest Advances and New Challenges: latest advances and new challenges*, ser. Operations research/computer science interfaces series. Springer, 2008. [Online]. Available: http://books.google.co.uk/books?id=-3ta5ne3-owC

[4] S. Cahon, N. Melab, and E. Talbi, "Paradiseo: a framework for the reusable design of parallel and distributed metaheuristics," *Journal of heuristics*, vol. 10, pp. 357–380, 2004.

[5] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, pp. 760–771, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0965997811001219

[6] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich, "Opt4J - A Modular Framework for Meta-heuristic Optimization," in *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)*, Dublin, Ireland, 2011, pp. 1723–1730.

[7] R. L. Pinheiro and D. Landa-Silva, "A development and integration framework for optimisation-based enterprise solutions," in

ICORES 2014 - Proceedings of the 3rd International Conference on Operations Research and Enterprise Systems, Angers, Loire Valley, France, March 6-8, 2014., 2014, pp. 233–240.

[8] J. A. Castillo-Salazar, D. Landa-Silva, and R. Qu, "A survey on workforce scheduling and routing problems," in *Proceedings of the 9th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2012)*, Son, Norway, August 2012, pp. 283–302.

[9] ——, "Workforce scheduling and routing problems: literature survey and computational study," *Annals of Operations Research*, 2014.

[10] ——, "Computational study for workforce scheduling and routing problems," in *ICORES 2014 - Proceedings of the 3rd International Conference on Operations Research and Enterprise Systems*, 2014, pp. 434–444.

[11] W. Laesanklang, D. Landa-Silva, and J. A. Castillo-Salazar, "Mixed integer programming with decomposition to solve a workforce scheduling and routing problem," in *ICORES 2015 - Proceedings of the 4rd International Conference on Operations Research and Enterprise Systems*, 2015, pp. 283–293.

[12] J. Matias, A. Correia, C. Mestre, P. Graga, and C. Serodio, "Web-based application programming interface to solve nonlinear optimization problems," in *Proceedings of the World Congress on Engineering 2010, Vol III*, 2010.

[13] P. Mestre, J. Matias, A. Correia, and C. S., "Direct search optimization application programming interface with remote access," *IAENG International Journal of Applied Mathematics*, pp. 251–261, 2010.

[14] F. Huang, "A New Application Programming Interface and a Fortran-like Modeling Language for Evaluating Functions and Specifying Optimization Problems at Runtime," *International Journal of Advanced Computer Science and Applications(IJACSA)*, vol. 3, no. 4, 2012. [Online]. Available: http://ijacsa.thesai.org/

[15] J. Walnes, "Xstream," April 2015, http://xstream.codehaus.org/.

[16] M. McShaffry, *Game Coding Complete, Fourth Edition*, ser. IT-Pro collection. Course Technology PTR, 2012. [Online]. Available: https://books.google.co.uk/books?id=HWYKAAAAQBAJ

[17] R. Nystrom, *Game Programming Patterns*. Genever — Benning, 2014. [Online]. Available: https://books.google.co.uk/books?id=9fIwBQAAQBAJ

[18] J. Hennessy, D. Patterson, and K. Asanović, *Computer Architecture: A Quantitative Approach*, ser. Computer Architecture: A Quantitative Approach. Morgan Kaufmann/Elsevier, 2012. [Online]. Available: https://books.google.co.uk/books?id=v3-1hVwHnHwC

[19] S. Oaks, *Java Performance: The Definitive Guide*. O'Reilly Media, 2014. [Online]. Available: https://books.google.co.uk/books?id=aIhUAwAAQBAJ

[20] F. Siebert, "Eliminating external fragmentation in a non-moving garbage collector for java," in *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '00. New York, NY, USA: ACM, 2000, pp. 9–17. [Online]. Available: http://doi.acm.org/10.1145/354880.354883

[21] D. F. Bacon, P. Cheng, and V. T. Rajan, "Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java," *SIGPLAN Not.*, vol. 38, no. 7, pp. 81–92, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/780731.780744

[22] M. Yener, A. Theedom, and R. Rahman, *Professional Java EE Design Patterns*. Wiley, 2014. [Online]. Available: https://books.google.co.uk/books?id=W7_lBQAAQBAJ

[23] D. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, ser. Artificial Intelligence. Addison-Wesley Publishing Company, 1989. [Online]. Available: https://books.google.co.uk/books?id=3_RQAAAAMAAJ