Towards an Efficient API for Optimisation Problems Data

Rodrigo Lankaites Pinheiro[1], Dario Landa-Silva[1], Rong Qu[1], Edson Yanaga[2] and Ademir Aparecido Constantino[3]

[1]*ASAP Research Group, School of Computer Science, University of Nottingham, United Kingdom*
[2] *Unicesumar - Centro Universitário Cesumar, Maringá, Brazil*
[3]*Departamento de Informática, Universidade Estadual de Maringá, Maringá, Brazil*
rodrigo.lankaitespinheiro@nottingham.ac.uk, dario.landasilva@nottingham.ac.uk, rong.qu@nottingham.ac.uk, edson@yanaga.com, aaconstantino@uem.br

Keywords: application programming interface, workforce scheduling and routing problems, decision support systems, research and development.

Abstract: The literature presents many application programming interfaces (APIs) and frameworks that provide state of the art algorithms and techniques for solving optimisation problems. The same cannot be said about APIs and frameworks focused on the problem data itself because with the peculiarities and details of each variant of a problem, it is virtually impossible to provide general tools that are broad enough to be useful on a large scale. However, there are benefits of employing problem-centred APIs in a R&D environment: improving the understanding of the problem, providing fairness on the results comparison, providing efficient data structures for different solving techniques, etc. Therefore, in this work we propose a novel design methodology for an API focused on an optimisation problem. 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 an intuitive fashion for researchers and efficient for the solving algorithms. Also, we present the concepts of a solution dispenser 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.

1 Introduction

The 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. Examples of such APIs and frameworks are ParadisEO (Cahon et al., 2004), jMetal (Durillo and Nebro, 2011) and Opt4J (Lukasiewycz et al., 2011). 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 a research and development (R&D) environment, understanding the problem can be an important asset to solve it. With a comprehensive understanding of the problem, one can achieve improved tailored solutions. Thus, having an API to handle the problem itself, including data, features, constraints and objective-function can be beneficial to the project.

Few APIs with a stronger focus on the problem were recently proposed, including an API to solve nonlinear optimisation problems (Matias et al., 2010; Mestre et al., 2010) and a new API for evaluating functions and specifying optimisation problems at runtime (Huang, 2012). Nonetheless, there is a reason why APIs and frameworks focused on problems are not common: they highly depend on the problem being tackled and a single optimisation problem possesses many variants, making it impracticable to define a unified model that covers all possible versions. Therefore, in this work we present a set of guidelines and recommendations 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 (2014), hence in the core of the API is the data model represented by a set of XML files. Our proposed design is composed of three novel 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 in memory. These data structures are designed to maximise

performance to access the data during the optimisation process. Lastly, we propose a feature called Solution Dispenser 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 further increasing computational performance. Although we use a workforce scheduling and routing problem to illustrate the components, the concepts of the API can be adapted to any optimisation problem because of the data-driven approach employed.

Finally, we present the results of employing the proposed API to an ongoing R&D project. By designing a problem-centric API, the team can avoid rework, improve the quality of the software and promote fairness to compare different developed solving techniques. We also present an empirical study of the efficiency of applying the solution dispenser instead of relying on the garbage collector of modern languages and we show that substantial computational performance can be gained by employing them.

The main contributions of this work are twofold:

1. A development methodology for problem-driven optimisation APIs that can be applied by both researchers and practitioners to increase productivity, reliability and code efficiency.
2. An object-pool technique to store solution objects to increase the performance of heuristic optimisation algorithms.

The remaining of this paper is structured as follows. Section 2 outlines the Workforce Scheduling and Routing Problems Project, which is used to illustrate the application of the proposed API. Section 3 presents the guidelines and instructions on designing the API. Section 4 presents the results obtained and section 5 concludes this work.

2 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's preference 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. (2012, 2014), Laesanklang et al. (2015a,b) and Pinheiro et al. (2016).

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 responsible for 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 into the current system.

Many APIs and implementations available in the literature focus on the solving techniques. We can highlight the ParadisEO (Cahon et al., 2004), the jMetal (Durillo and Nebro, 2011) and the Opt4J (Lukasiewycz et al., 2011). 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 to 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. (2010) and Mestre et al. (2010) 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, defining exclusively the objective-function may be too restricting to the research of the solver because solvers may require access to individual features of the problem (constraints, objectives, etc.) or to partial evaluations. 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 (2012) 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 optimisation problems at runtime. Conceptually, we differ from them as we are not proposing an implemented tool applicable to specific scenarios, but instead, we provide the concepts of a tailored API that can help on the research and development of optimisation solutions.
- Pinheiro and Landa-Silva (2014) propose a framework to aid in 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 extend that concept and describe how to design and implement the key components of a problem-oriented API. While that work focuses on the research and development methodology, here we focus on the design of an API that can further facilitate the development of optimisation solutions.

Although the aforementioned related works attempt to provide problem-oriented tools, to the best of our knowledge they are restrictive and possibly not widely applicable due to their limitations. Therefore, the design of a tailored API for an optimisation problem can be beneficial to an R&D project as long as the time spent on its development is justifiable. Hence, we now present a set of guidelines to facilitate the development of such tools.

3 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.

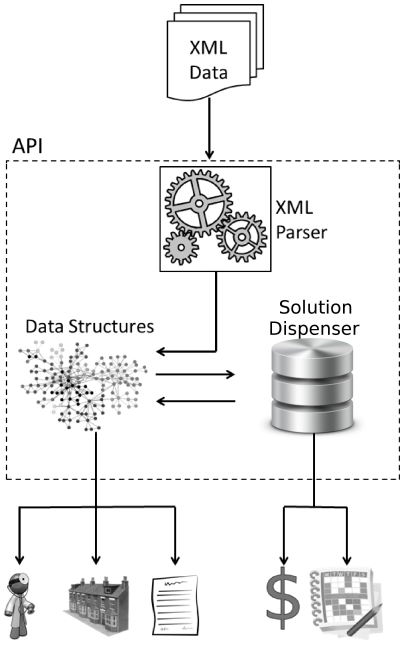

Figure 1: Overview of the API.

Figure 1 presents an overview of the API components. On top, we have the XML Data (the files containing a problem instance definition) as input for the API. The XML Parser decodes the files and builds the Data Structures that can be accessed by the user of the API. The user can also access the Solution Dispenser to instantiate and dispose of solution objects of the optimisation problem. The Solution Dispenser communicates with the Data Structures to evaluate and update current solutions.

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 how an algorithmic solver interacts with the API.

3.1 API Concept

Figure 2 present an overview of the API employed in the WSRP project and how the developed solvers interact with the API. Following, we describe each feature numbered in the figure. Some features extend to all problems and can potentially be employed by an API tackling any problem.

1. **Solver:** the API facilitates the use of different solving techniques (exact solvers, heuristic algorithms, etc.) on the problem because it provides a set of methods to easily access different features of the problem, such as values, constraints and basic operations on the data. Therefore, the

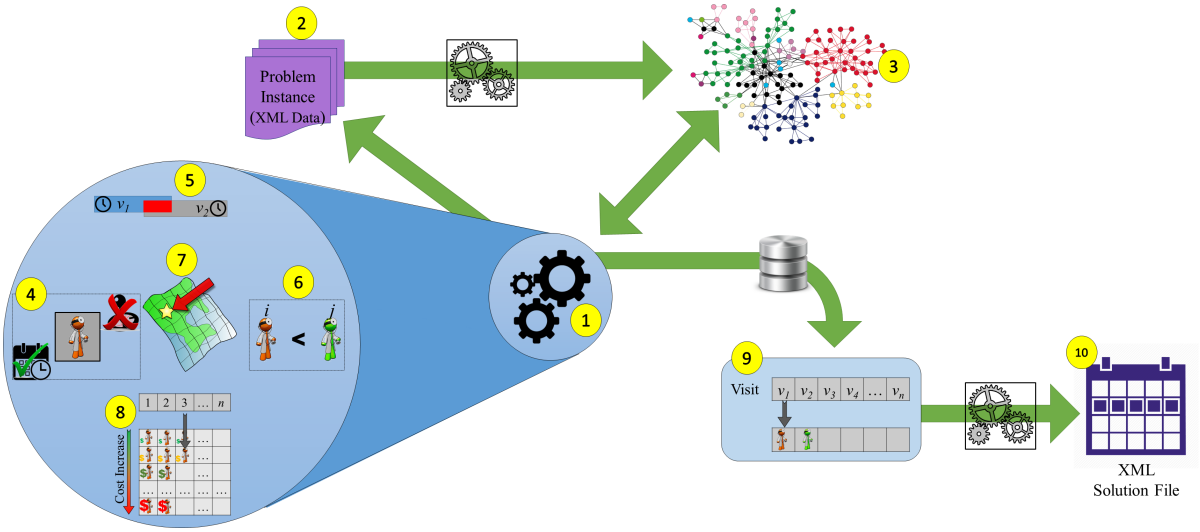

Figure 2: Main features of the WSRP API, where (1) refers to the optimisation solver (mathematical solver, heuristic algorithms, etc.), (2) and (10) to the data parser, (3) to the internal memory structures, (4) to (8) to the operations and methods supported by the API and (9) to the Solution Dispenser.

researchers can plug in existing solving APIs and frameworks or new algorithms.

2. **Problem Data:** we consider that each problem instance can be represented in a set of datafiles (in this case we use XML files). These files must contain all information related to a single instance of the optimisation problem.
3. **Internal Data Structures:** one of the main functionalities of the API is to provide a set of efficient data structures to hold the data in the memory. After the solver selects a problem instance to load, the data parser reads the files and allocate the problem data into memory. These structures are designed to provide maximum access performance to the solver.

Problem-specific features must be adapted to the problem at hand. The API should provide easy access to the internal data structures and methods for easy and efficient access. Following, we present what we considered to be essential to be provided by the WSRP API:

4. **Constraints:** the API provides several methods to assess and evaluate the basic problem constraints related to assignments. For example, given an assignment (a visit and a worker), the API provides methods to:
   - Check if the worker is skilled for the visit.
   - Check if the worker possesses a valid contract to perform that visit.
   - Check if the worker is available at the time of the visit.
   - Check if the worker can commute to the place of the visit.
5. **Operations:** the API also provides several operations to be used by the solver. These operations can be simple checks and validations or complex functionalities. Among the operations in the WSRP we highlight:
   - Check if two given visits are time conflicting.
   - Check if a worker is qualified to perform two given visits.
   - Calculate the best contract for an assignment.
6. **Preferences:** in addition to constraint methods, the WSRP API also provides methods for evaluating preferences (worker, staff and patient's preferences):
   - Retrieve the worker preferred to perform a given visit based on historical data.
   - Calculate the visits that a given worker prefers to perform.
   - Locate the worker that best matches a given patient preferences.
7. **Geographical Methods:** because the WSRP combines scheduling and routing, we designed a set of methods related to the geographical data, including:
   - Check if a given visit is within a given area.
   - Given a transportation mode, calculate the time to commute from a visit to another.
   - Check which transportation modes can be used within an area.

- Retrieve all visits within an area.

8. **Overall Data Access:** it is important to notice that although the provided methods and operations are extensive, specific solvers or techniques may require a different way to access the data, hence, the WSRP API also provides basic 'get' methods for all information (Figure 3).

Finally, the API also provides functionalities related to building and maintaining solutions for the optimisation problem:

9. **Solution Handling:** the solver may deal with a single or with multiple solutions simultaneously. The Solution Dispenser provides an interface for the solver to create and store solutions in the memory. This component provides the following methods:
   - Create a new solution.
   - Add assignment – given a visit, a worker, a contract, start time and transportation mode, the method uses this information and creates an assignment in a currently specified solution.
   - Dispose of the solution.
10. **Save Solution:** finally, the data parser is able to save a solution object to an XML file that can later be retrieved and loaded back into memory.

Next, we detail each component of the proposed methodology to build an API for optimisation problems data.

3.2 XML Data Parser

Pinheiro and Landa-Silva (2014) 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 to be incorporated. In this context, a data model to represent the problem was proposed to improve the development of the definition of the optimisation problem being tackled, the independence between the development team and the research team and to promote an easy integration of the solver to the actual information system.

Extending the data modelling concept proposed by Pinheiro and Landa-Silva (2014), the first component of the proposed API is a parser to read the input files from the data model and build the data structures. Additionally, the parser is also responsible for converting a solution of the optimisation problem into the data file. Also, 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 to serialise the data to and from an XML data format.

Since we are using Java, we employed the XStream Java library (Walnes, 2016), 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 requires 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.

3.3 Internal Data Structures

The second component of the API is composed of 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 into groups of objects, following the data model orientation. Therefore, we have in the WSRP API the following groups:

- Visits – containing information about the requirements of each visit, e.g. number of workers required, start time, duration, location, skills required, preferences, etc.
- Workers – containing information related to the workers themselves, e.g. skills, availability, home location, preferences.
- Areas and locations.
- Transportation modes.
- Contracts – containing information regarding maximum and minimum working hours and costs.

To hold the objects, we first use arrays. The advantage of using arrays is that the random access using indexes is 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 array 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 the number of workers, tasks, the date format used in the files, etc.

Using arrays may be sufficient for most solving algorithms as it allows fast random access and quick interaction through the elements. However, it may be a problem with specialised heuristics or external software that might have access to the API. Such systems are often linked with a database, hence, they handle elements using their own identification number, which is stored in the XML, but is 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 better 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 in the worst case.

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 define the tasks and the set of tasks. The main class, *WorkforceSchedulingAndRoutingProblem*, is composed of 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 Dispenser, explained in the subsequent section. Note that the *calculateMinimumNumberOfAssignments* method, as aforementioned, starts with the 'calculate' word, hence in the worst case it performs in $O(n)$, while all 'get' methods performs with $O(1)$ time complexity.

The *Tasks*, *Areas*, *HumanResources* and *Contracts* classes contain 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

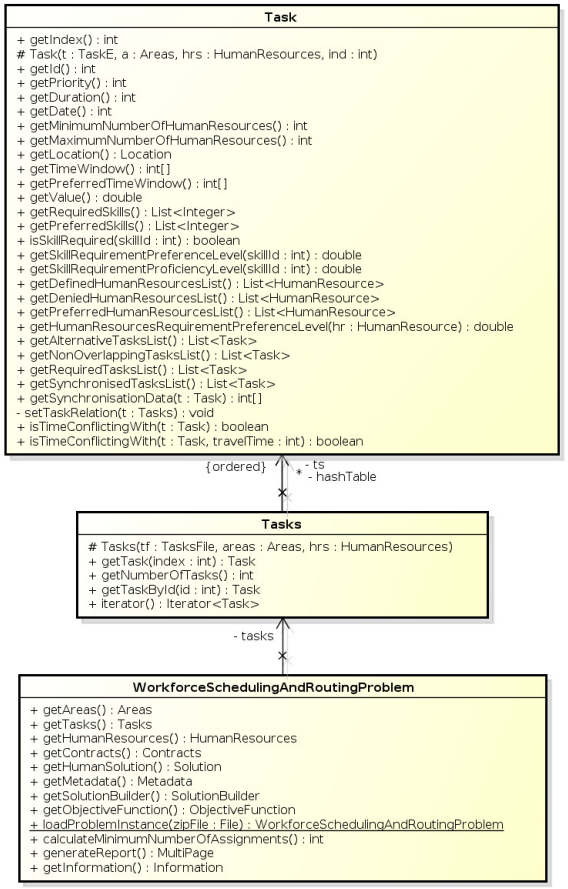

Figure 3: Class diagram for the main problem class and the tasks-related classes.

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 conflict in time with the current task (hence they cannot be performed by the same worker).

3.4 The Solution Dispenser

The last component of the API is the Solution Dispenser (SD). The SD provides an interface for the user to build and assess a solution for a given problem instance. Once the problem is loaded into an object, the user can invoke the SD 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).

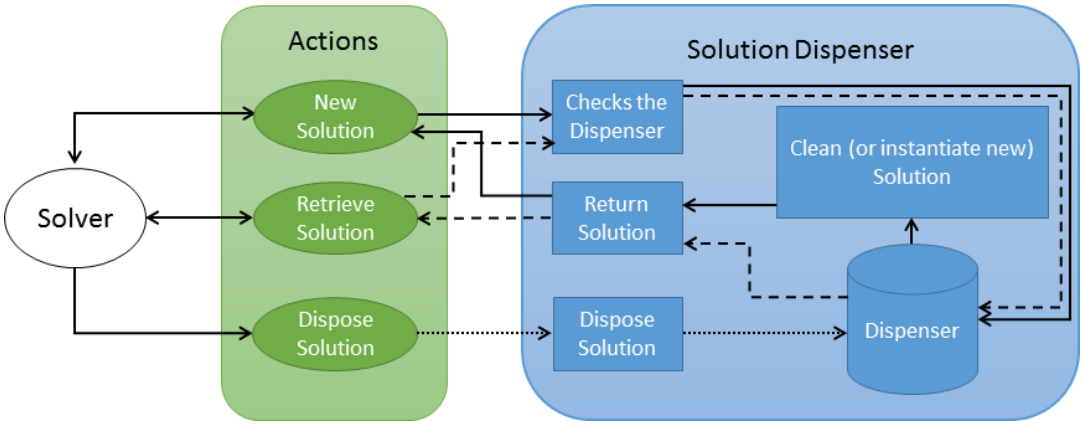

Figure 4: Flow diagram for the Solution Dispenser. Solid lines represent the flow for the 'New Solution' action, dashed lines for the 'Retrieve Solution' action and dotted lines for the 'Dispose Solution' action.

The user can also invoke the objective function to evaluate the solution. Figure 4 presents a schematic for the SD component.

Modern programming languages, such as Java and C#, provide a convenient way to handle objects: a garbage collector. When the user doesn't need an object anymore, all he has to do is to get rid of the links and pointers to that object. At a given 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 (Siebert, 2000; Bacon et al., 2003).

Leaving the disposal of objects to the garbage collector can lead to a decrease in performance that, aside from being marginal for most applications, can have an unacceptable impact on optimisation algorithms. Hence, the SD internally implements an Object Pool design pattern (Nystrom, 2014) to recycle the objects. To do that we employ a factory object implemented using the *factory* design pattern (Yener et al., 2014) for easy creation of the objects. This factory is responsible for creating new solution objects.

When a new solution request is invoked (Figure 4), 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 solver. The solver now has an empty solution that it can use. When the solver does not need the solution anymore, it can dispose of the solution by invoking the specific method in the SD. The factory then receives the disposed solution and stores it in the list.

Potentially, the use of the solution dispenser can provide significant performance gains. Take for instance a population-based algorithm that processes one generation per second with a population of 100 individuals. That means 100 solutions being disposed of per second. After ten minutes running, the algorithm will have disposed of 60000 solutions, which potentially could fragment the memory and cause several garbage collection calls. Now, when using the dispenser, considering the worst case, when a new population is created before disposing of 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, which could represent a reduction of 97.5% on the processing time and memory consumed by the garbage collector (see section 4.1).

4 Experiments and Results

We now present the results of the use of the API in the WSRP project. Having a centralised API containing a parser and efficient internal data structures helped both teams to avoid performing rework. Also, the company's development team were able to easily handle the data files by using the integrated parser, saving time. The internal data structures allowed everyone to feel confident that they were using an efficient implementation. Having the best-known data structures available for everyone, helps to achieve improved efficiency in all solvers developed. Additionally, after the release of the API, the use of the integrated Solution Dispenser helped our team to assess and compare the developed solvers because it guaranteed that both single point algorithms and population-based were handled efficiently. Finally, having mul-

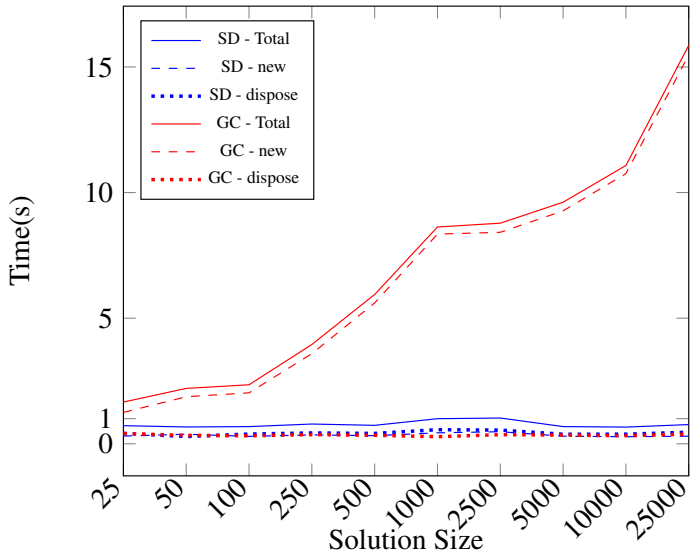

Figure 5: Time comparison between the Solution Dispenser (SD) and the Java Garbage Collector (GC) to instantiate and dispose new solutions.

tiple 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, ensuring higher quality on the software developed.

4.1 Solution Dispenser

We now present an empirical analysis of the efficiency of the Solution Dispenser. The SD is responsible for holding solution objects of given problem, hence small problems using larger encoding consume more memory than larger problems using smaller encoding. 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 which is of size *number of workers* $\times$ *number of tasks*). Therefore, to test the solution dispenser 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 (Durillo and Nebro, 2011), 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 of one million solution objects. For the experiments using the garbage collector, the disposing process merely unlinked the objects to free them to the Garbage Collector (GC) while for the SD it called the internal dispose process and cleared 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 32GB 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 for public use (Cahon et al., 2004; Durillo and Nebro, 2011; Lukasiewycz et al., 2011).

Figure 5 presents the results of the time computation. The red lines represent 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 dispenser. The solid lines represent the total time, the dashed lines represent the time used by the *'new'* method, which allocates new solution objects and the dotted lines represent the *'dispose'* method. We can see that the time spent by the SD follows a constant trend throughout all problem sizes. This happens because both the *'new'* and *'dispose'* operations of the solution dispenser perform in constant time and since there are no objects freed in the memory (they are being kept alive by the SD), the garbage collector (automatically activated by the Java virtual machine) just quickly checks for dead objects, finds nothing, and is deactivated without any extra processing.

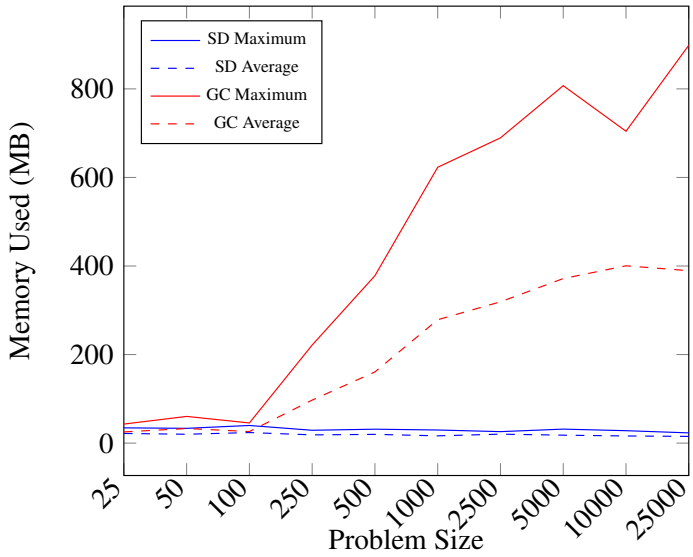

Figure 6: Memory comparison between the Solution Dispenser (SD) and the Java Garbage Collector (GC).

Regarding the tests using the garbage collector, we see that the *'dispose'* operation is performed 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 notice that in our experiments we did not specify any parameters for the Java virtual machine, hence the experiments had 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 presented tests, hence further decreasing the performance.

In Figure 6 we have the results of the memory allocation measurement. The red lines represent the experiments using the garbage collector and the blue lines the solution dispenser. Also, the solid lines represent 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 SD, 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 see that it makes use of much more memory, which reinforces our previous statement that in a scenario where the memory is limited the garbage collector requires more frequent activation.

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

5 Conclusion

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 to increase 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 storing this information using efficient data structures that are clear and computationally efficient, hence it can improve the research process and be applied to a final solver algorithm. Finally, we proposed a component called Solution Dispenser which 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, 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. Also, 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. Moreover, we analysed the advantages of using the Solution Dispenser and presented the computational gains that can be obtained by employing such technique and the design patterns presented.

In conclusion, the guidelines of the API design proposed in this work can help academics and practitioners to achieve improved results, both for research or production purposes. While the API provides a fair mechanism for researchers to develop and assess their algorithms, it also provides a reliable framework for practitioners to ground their solutions. Additionally, the computational performance that can be gained from applying the aforementioned techniques can help researchers to develop their algorithms and the industry to improve their solutions or reduce costs. Finally, our reported experience, arising from a collaboration with an industrial partner, may be useful for other researchers and practitioners in a similar position.

Our future work will further investigate the data structures aiming to improve their efficiency. Also, we will test the proposed dispenser on a real-world environment, assessing it with optimisation algorithms solving real-world problems.

REFERENCES

Bacon, D. F., Cheng, P., and Rajan, V. T. (2003). Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. *SIGPLAN Not.*, 38(7):81–92.

Cahon, S., Melab, N., and Talbi, E. (2004). Paradiseo: a framework for the reusable design of parallel and distributed metaheuristics. *Journal of heuristics*, 10:357–380.

Castillo-Salazar, J. A., Landa-Silva, D., and Qu, R. (2012). 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)*, pages 283–302, Son, Norway.

Castillo-Salazar, J. A., Landa-Silva, D., and Qu, R. (2014). Workforce scheduling and routing problems: literature survey and computational study. *Annals of Operations Research*.

Durillo, J. J. and Nebro, A. J. (2011). jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771.

Huang, F. (2012). 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)*, 3(4).

Laesanklang, W., Landa-Silva, D., and Castillo-Salazar, J. A. (2015a). 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*, pages 283–293.

Laesanklang, W., Pinheiro, R., Algethami, H., and Landa-Silva, D. (2015b). Extended decomposition for mixed integer programming to solve a workforce scheduling and routing problem. In *Operations Research and Enterprise Systems*, volume 577 of *Communications in Computer and Information Science*, pages 191–211. Springer International Publishing.

Lukasiewycz, M., Glaß, M., Reimann, F., and Teich, J. (2011). Opt4J - A Modular Framework for Metaheuristic Optimization. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)*, pages 1723–1730, Dublin, Ireland.

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

Mestre, P., Matias, J., Correia, A., and S., C. (2010). Direct search optimization application programming interface with remote access. *IAENG International Journal of Applied Mathematics*, pages 251–261.

Nystrom, R. (2014). *Game Programming Patterns*. Genever — Benning.

Pinheiro, R., Landa-Silva, D., and Atkin, J. (2016). A variable neighbourhood search for the workforce scheduling and routing problem. In *Advances in Nature and Biologically Inspired Computing*, volume 419 of *Advances in Intelligent Systems and Computing*, pages 247–259. Springer International Publishing.

Pinheiro, R. L. and Landa-Silva, D. (2014). 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.*, pages 233–240.

Siebert, F. (2000). 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*, CASES '00, pages 9–17, New York, NY, USA. ACM.

Walnes, J. (2016). Xstream. http://x-stream.github.io/.

Yener, M., Theedom, A., and Rahman, R. (2014). *Professional Java EE Design Patterns*. Wiley.