To appear in *Management Analytics*
Vol. 00, No. 00, Month 20XX, 1–24

An Application Programming Interface with Increased Performance for Optimisation Problems Data

Rodrigo Lankaites Pinheiro[a*], Dario Landa-Silva[a], Rong Qu[a], Ademir Aparecido Constantino[b] and Edson Yanaga[c]

[a]*ASAP Research Group, School of Computer Science, University of Nottingham, UK*
[b]*Departamento de Informática, Universidade Estadual de Maringá, Maringá, Brazil*
[c]*Unicesumar - Centro Universitário Cesumar, Maringá, Brazil*

(*Received 00 Month 20XX; accepted 00 Month 20XX*)

An optimisation problem can have many forms and variants. It may consider different objectives, constraints, and variables. For that reason, providing a general application programming interface (API) to handle the problem data efficiently in all scenarios is impracticable. Nonetheless, on a R&D environment involving personnel from distinct backgrounds, having such an API can help with the development process because the team can focus on the research instead of implementations of data parsing, objective function calculation, and data structures. Also, some researchers might have a stronger background in programming than others, hence having a standard efficient API to handle the problem data improves reliability and productivity. This paper presents a design methodology to enable the development of efficient APIs to handle optimisation problems data based on a data-centric development framework. The proposed methodology involves the design of a data parser to handle the problem definition and data files and on a set of efficient data structures to hold the data in memory. 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 and provide an integrated objective function so that researchers can easily compare solutions from different solving techniques. 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

1. Introduction

Research on decision support systems (*DSS*) is multidisciplinary and has undergone numerous improvements in the past few decades [1]. One particular type of DSS includes systems focused on solving optimisation problems such as workforce scheduling [2], vehicle routing [3] and many other kinds of problems with industry applications. In this context, 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, such as ParadisEO [4], jMetal [5] and Opt4J [6].

The aforementioned APIs provide flexible implementations of state-of-the-art solving algorithms that can be applied to many optimisation problems. The user can code their representation

*Corresponding author. Email: rodrigo.pinheiro@nottingham.ac.uk

of the problem and objective function and plug-in the algorithms to solve instances of the problem. Analogously, having an API to handle the problem data can also be very useful: users could save time by not having to code data parsers and data structures to hold the problem-related data in memory. Less experienced programmers could have access to efficient implementations of the internal structures to handle the problem data, etc.

Pinheiro and Landa-Silva [7] discussed the benefits of having a R&D methodology centred on the problem data. Such benefits include obtaining a deeper 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 an 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 re-work, improving computational performance and promoting standard solution comparison measures.

However, it is impractical to provide a general API for a given optimisation problem because each optimisation problem has many variants. Thus, even if it were possible to include all variants in a single API, that would have a performance impact as mechanisms for verification and validation would have to be integrated, as well as extra data fields and structures to accommodate different characteristics. Also, having such a flexible API would increase the complexity of using it. Yet, the recent literature presents few related contributions, such as an API to solve nonlinear optimisation problems [8, 9] and a new API for evaluating functions and specifying optimisation problems at runtime [10].

More recently, Pinheiro et al. [11] recognised the importance of an API for optimisation problems data and proposed a methodology to allow researchers and practitioners to design and build their own data-centric API. The methodology is composed of three components. The first is a parser for the data files that can read from and write to the modelled format. The second are the data structures containing the relevant optimisation data kept on memory. The third is a mechanism to store and manage solution objects in memory. In this work, we extend that concept by providing further insights and information on how to apply the methodology to build the API. Additionally, we bring the Object Pool, Data Locality, Matrix Ordering, Memory Alignment and the Dirty Flag programming design patterns to reduce CPU cache misses and improve the API computational efficiency. Moreover, we propose a feature called Solution Builder which centralises the objective function and provides a repository that recycles solution objects to minimise the interference of the built-in garbage collector, therefore improving memory fragmentation and further increasing computational performance.

Lastly, we use an R&D project undertaken in a partnership with a software development company to validate the application of the proposed methodology by building an API for a workforce scheduling and routing optimisation problem. We present an empirical study about the efficiency of applying the Solution Builder in detriment of relying on the garbage collector of modern languages. We also present experimental results for the design patterns presented in this work to show that substantial computational performance can be gained by employing the proposed methods.

The main contributions of this work are:

(1) A methodology that allows researchers and practitioners to build efficient data-driven APIs for optimisation problem data. This methodology promotes reusability of code, better use of research staff skills, improved integration between researchers and practitioners and facilitates the implementation of efficient solutions.
(2) The proposal and assessment of design patterns and techniques gathered from the literature and applied to optimisation problems that provide substantial performance gains to optimisation applications.

(3) A centralised objective function evaluation mechanism that promotes fair comparison of solutions obtained from different methods and techniques. Also, the mechanism recycles solution objects held in memory to avoid the garbage collector of modern programming languages and increase the overall processing performance of the developed solvers.

The remaining of this paper is structured as follows. Section 2 outlines the Workforce Scheduling and Routing Problems (WSRP) 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 WSRP. In general terms, WSRP refers to 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 for patients at their homes. In this problem, the primary 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. [12, 13, 14] and Laesanklang et al. [15].

We are engaged in an R&D project in collaboration with an industrial partner to develop an 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 designed 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 ParadisEO [4], jMetal [5] and Opt4J [6] as examples. They are all frameworks that provide several solving algorithms for both single- and multi-objective 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 rather than on the solving techniques.

- Matias et al. [8] and Mestre et al. [9] 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 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 for researching the solver. Hence, our API could be integrated with this or any framework focused on the solving algorithm as we focus on how to access the data efficiently and build solution objects.
- Huang [10] proposes a new API for evaluating functions and specifying optimisation problems at runtime. Huang proposes 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, our proposal differs from Huang's because we are not proposing a general API, but instead conceptualising the design of a tailored API that can help in the research and development of optimisation solutions.

- Groth et al. [16] propose an API that extends the Linked Data architecture for pharmacology data, to enable researchers to read, access and integrate multiple datasets. Their API highlights the importance of a mechanism to handle data access. However, the approach of providing an API to the community cannot be readily applied to optimisation problems because of the high computational performance required by the algorithms.
- Pinheiro and Landa-Silva [7] 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.
- Swaminathan et al. [17] conduct a survey of APIs for genomic-related applications to improve healthcare practice. The studied APIs (Google Genomics, SMART Genomics and 23andMe) provide mechanisms to parse genomic data and sets of operations to aid researchers and developers to access and operate on the complex genomic data. Although not related to optimisation, the authors show the importance of such tool on R&D projects.
- Recently, Pinheiro et al. [11] promote the use of data-centric APIs for optimisation problems data as a support tool for R&D projects. They propose a methodology for the development of tailored APIs to handle the problem's data efficiently. The proposed method has three components: a data parser to load/save the problem's data from/to files; the efficient and intuitive internal data structures; and a solution dispenser that keeps a pool of solution objects in memory and recycles them as required, hence reducing garbage collector calls. This work further extends the proposed methodology. We present new details and insights on the proposed components. Additionally, we recommend the use of programming design patterns to improve computational performance. Lastly, we extend the solution dispenser to create the Solution Builder, which not only keeps a pool of solution objects but also presents a centralised objective function.

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 effective way.

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 Builder* to instantiate and dispose of solution objects of the optimisation problem. The *Solution Builder* 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 optimisation algorithms to query the data and to compare solutions from different approaches. We describe next how an

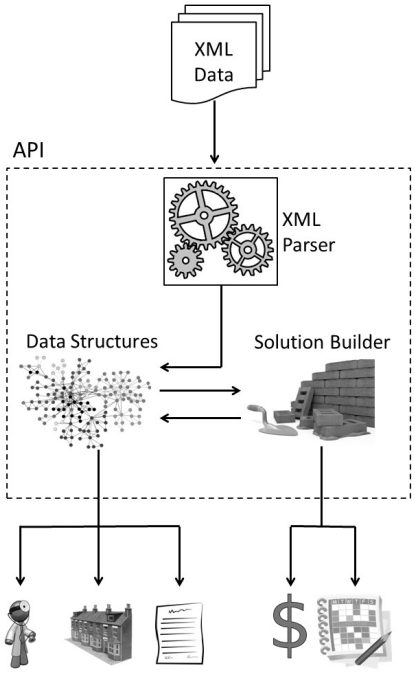


Figure 1.: Overview of the API.

algorithmic solver interacts with the API.

3.1. *API Concept*

Figure 2 presents 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 quickly access various features of the problem, such as values, constraints and basic operations on the data. Therefore, the researchers can plug in existing solving APIs and frameworks or new algorithms.
2. **Problem Data:** we consider that each problem instance can be contained in a set of data files (we use XML files) with 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 problem data in 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.

The API should provide secure and efficient access to the internal data structures and methods. We now present what we consider essential to be provided by the WSRP API:

4. **Constraints:** the API provides several methods to assess and evaluate the underlying 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.

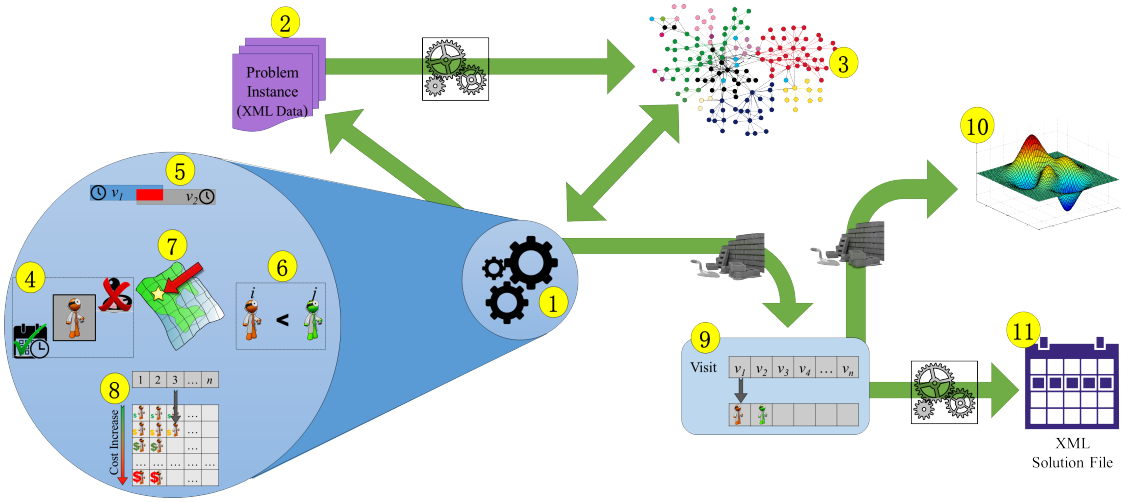


Figure 2.: Main features of the WSRP API, where (1) refers to the optimisation solver (mathematical solver, heuristic algorithms, etc.), (2) and (10) to the data parser, (3) to the internal memory structures, (4) to (8) to the operations and methods supported by the API and (9) to the Solution Dispenser.

- 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 the following ones:
   - Check if two given visits are time-conflicting.
   - Check if a worker is qualified to perform two given visits.
   - Identify 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 preferred worker to perform a given visit based on historical data.
   - Identify the visits that a given worker prefers to perform.
   - Select 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 specific 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 4).

Finally, the API also provides functionalities for building and maintaining solution objects for the optimisation problem:

9. **Solution Handling:** the solver may deal with a single or with multiple solutions simultaneously. The Solution Builder provides an interface for the solver to create and store solutions in 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 given solution.
   - Dispose a solution.
10. **Evaluation Mechanism:** the builder also provides an interface so that a solution can be eval-

uated. The evaluation mechanism comprises of:
- A built-in objective function that calculates the values of objectives and constraint violations.
- A scalarisation function that takes the objective values and weights and generates a solution fitness value. The objective weights can be customised.
- Dispose a solution.

11. **Save Solution:** additionally, the data parser can 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.

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.

3.2. *XML Data Parser*

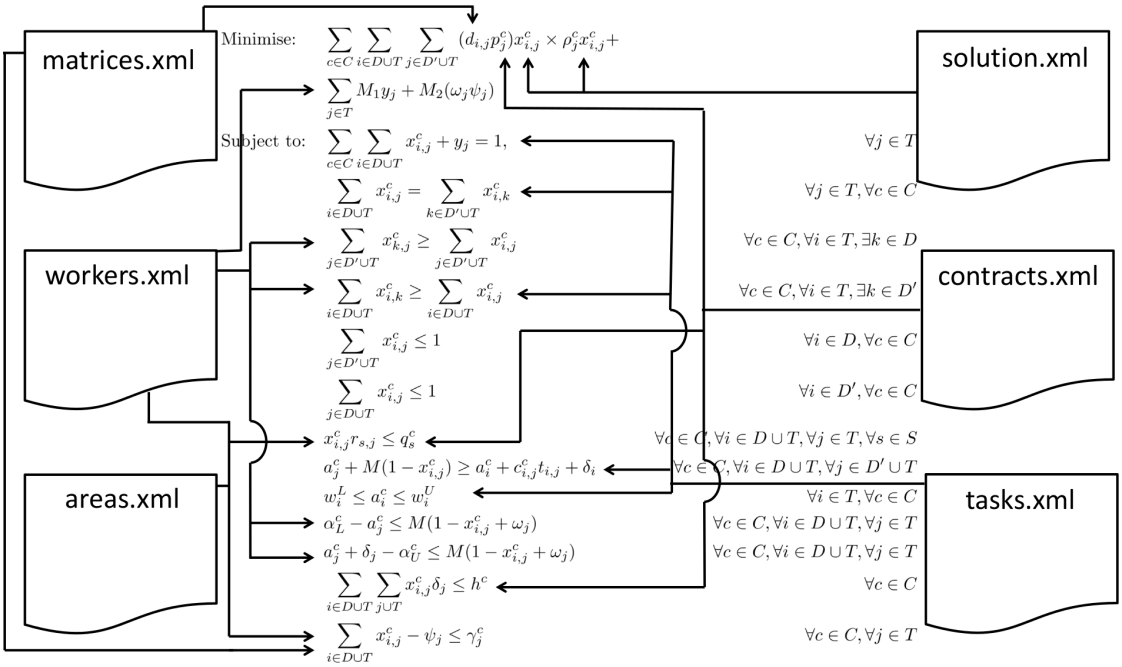


Figure 3.: Representation of the XML Model.

Pinheiro and Landa-Silva [7] 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 DSS, it is common that an information system already exists and that the DSS 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, scholars often focus on the technical content. Hence, divergences may arise and a precise definition of the model can help to establish

a common ground.
- Development independence: in the scenario mentioned above, we assume that the R&D team, mostly composed of academics, is not the same team that develops the central information system. Hence, the data model virtually represents a layer in which both teams can rely upon.
- Easy integration: having a data model early in the project helps to integrate the final solution into the current information system. That happens because all data will be translated to and from the data model. Hence both R&D team and leading development team use the model to integrate the optimisation module to the main system. Thus, the development is modularised, and the primary system and the decision support module only communicate through the data model, which is well known for both teams.

Figure 3 presents the overall idea of the data model. We see that the problem features, represented by the mathematical formulation of the optimisation problem, relate to the XML files in a manner that is intuitive for human beings to understand (grouping-related features in separate files).

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 implementation should accommodate extensions and updates in the data model.

For that purpose, we employed a serialisation library. Since we used Java, we employed the XStream Java library [18], however, most high-level languages have XML serialisation mechanisms available. The advantage of such approach is that a set of classes that corresponds to the modelled data can be used, and the serialisation library handles all the file parsing. This method is easy to develop and does not require extensive programming effort. However, it is likely that the objects in memory are not best suited for performance or 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 that hold the problem-related data in memory. It is important to emphasise that while the aforementioned data model is intended to be an explicit representation of the optimisation problem, the internal data structures must be efficient for access by solving algorithms.

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 accommodate different solving techniques easily, as we are not only concerned about the final product but with the entire process of executing the R&D project. Hence, in our work, we divided the API into groups of objects, following the data model orientation:

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

In order to hold the objects in memory, 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 memory allocation requires to estimate the size of the data or use dynamic memory allocation which could also hinder performance. 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 and the date format used in the files.

Using arrays may be sufficient for most solving algorithms as it allows fast random access. However, it may be a problem with specialised heuristics or external software accessing the API. Such systems are often linked to a database. Hence, they handle elements using an 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. 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 according to the operation performance. 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 4.: Class Diagram for the Main Problem Class and the Tasks-related Classes.

Figure 4 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 in this class that 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).

3.3.1. Code Optimisation

McShaffry [19] states 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 [20]. 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.

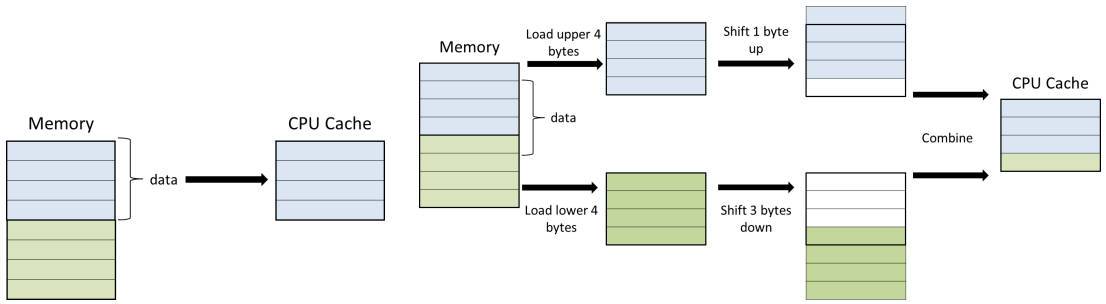
1. Data Locality. One of the most overlooked ways to increase (or decrease) performance in an application is due to the internal processor's cache memory. Modern computers possess an internal processor memory (cache memory) that bridges the access to the main RAM memory to increase the system performance. 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 requesting 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 are 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 are loaded into the cache [21].

[20] proposes the *data locality* design pattern that attempts to reduce the number of cache misses in an application. The design pattern consists of sacrificing some abstraction and object-oriented concepts 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 together into the cache.

Another way to improve cache hits is to use arrays of homogeneous data. Suppose some massive calculations require the costs of all tasks to be processed. Instead of having an attribute 'cost' inside task objects, 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 include the costs for multiple consecutive tasks, hence actually promoting multiple cache hits.

This pattern affects the way getter and setter methods are implemented. Employing such meth-



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

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

Figure 5.: Comparison of Cacheing Aligned and Misaligned Data.

ods 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 [22]. 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 to establish whether making use of setters and getters impacts performance or not.

2. Matrices Ordering. Another pattern aimed to improve cache memory access focuses on an issue that may be overlooked: the ordering of matrices in memory [19]. Optimisation problems data are often organised in matrices. Accessing these data in the correct order to avoid cache misses can potentially improve performance. Suppose the data from a matrix are stored in memory ordered by row (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 accessed 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 results in a cache miss.

3. Memory Alignment. McShaffry [19] notes that the CPU reads and writes 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). An aligned chunk of data is promptly loaded into the cache while an unaligned chunk of data must be read in parts, shifted to the target frame and then loaded. Figure 5(a) presents a diagram of a memory mapping of aligned data. The mapping is direct, meaning that the data are directly transferred from the RAM memory to the internal cache memory. Figure 5(b) presents the copy of misaligned data. The CPU reads the two chunks containing the parts of the required data, shifts each piece to select the requested information and merge into a single chunk which is then copied into the cache. Clearly, the second case represents a much slower mechanism.

The best way to take advantage of memory alignment is to make sure that the internal data types, structures and classes of the API have a number of bytes that is a power of 2. In the case a data type has fewer bytes, dummy variables can be added to force the structure (or object) to have the 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.

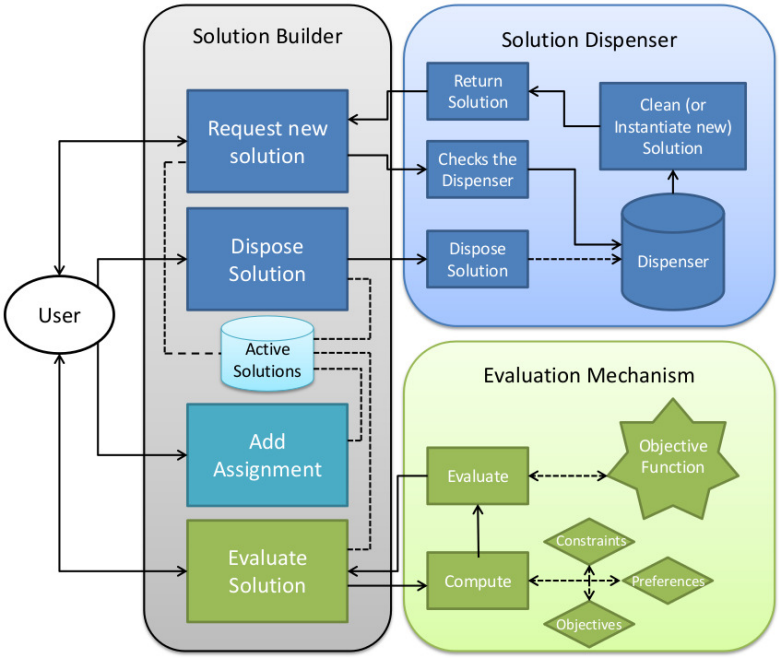


Figure 6.: Flow Diagram for the Solution Builder. The Rectangles Inside the Solution Builder Box Represent the Functionalities Given to the User. Arrows Represent Data Flow.

3.4. *The Solution Builder*

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

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

The SB provides an interface for the user to build and assess a solution to a given problem instance. Once the problem is loaded into an 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. This number can be used to access the solution, add new assignments and evaluate the solution fitness according to multiple criteria (preferences, objectives or constraints). The user can also invoke the objective function to evaluate the solution. Figure 6 presents a schematic of the SB component.

3.4.1. Centralised Evaluation Mechanism

Integrated into the SB is the solution evaluation mechanism. Pinheiro and Landa-Silva [7] pointed at the importance of a mechanism to ensure fairness in the comparison of results from different techniques implemented. Having a centralised objective function is beneficial and helps to avoid re-work and maintain consistency.

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

3.4.2. Solution Dispenser

Modern programming languages, such as Java and C#, provide a convenient way to handle objects: a garbage collector. The user only needs to dispose links and pointers to objects, then the garbage collector looks for objects not linked by the user's program and frees the memory. That can lead to two problems: memory fragmentation and extra processing time to seek, to free memory and later to allocate new objects [23, 24].

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 SB internally implements an Object Pool design pattern [20] to recycle objects. We employ a factory object implemented using the *factory* design pattern [25] for easy creation of the objects. This factory is responsible for creating new solution objects.

When a new solution request is invoked (Figure 6), 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 data 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 a specific method in the SB. 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 60,000 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 active solutions 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).

3.4.3. 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. *Dirty Flag* Nystrom [20] proposes a design pattern called *dirty flag*. This pattern consists of a mechanism to avoid unnecessary recalculations when you have nested operations, usually on recursive calls. In the solution evaluation context, the objective function can be very costly, and recursive calculations may be needed. Additionally, algorithms may require specific parts of the objective function to be calculated at 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 rely on that object must be re-calculated. Thus, in a recursive operation where a value would always be calculated, it will now be calculated if the flag indicated that. Essentially, after a solution is built, only one evaluation of its values are made. If an algorithm calls the evaluation (or partial evaluation) on an unchanged solution, the last calculated value is returned.

4. Experiments and Results

We now present the results of employing the API in the WSRP project. We initially discuss the advantages for the R&D project. Later, we present the results of the technical analysis on the Solution Builder component and the benefits of using it. Additionally, we provide evidence

of computational performance gains when using the design patterns suggested in the previous section.

4.1. *Improvements on the R&D Project*

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 to speed the research conducted by the academic researchers.

Solver Efficiency During the design of the API, specifically the internal data structures, team members could use it and assess it gaining confidence that the implementation is effective, efficient and has the best-known data structures available, hence helping to achieve improved efficiency in all solvers.

Consistency As discussed in Pinheiro and Landa-Silva [7], one of the main concerns on a R&D optimisation project is consistency in the comparison of multiple solving techniques. Using the integrated Solution Builder in the API helped our team to assess and compare the developed solvers because it guaranteed that the fitness calculations were consistent between methods.

Error Identification Having multiple people working on a single API helped to spot code errors and bugs faster than having researchers relying solely on their code. During the first months of our project, we had the research team releasing several versions of the API until we obtained a stable version. This process increased the confidence of the team in the tool and helped us ensure that we had a reliable component to support the research.

4.2. *Solution Builder*

We now present an empirical analysis of the efficiency of the Solution Builder. The SB is responsible for holding solution objects for the given problem. Small problems using larger encoding might consume more memory than larger problems using smaller encoding. For example, an integer array representation (an array the size equal to the number of tasks) is an example of smaller encoding and a binary array representation (a matrix which is of size *number of workers* $\times$ *number of tasks*) is an example of a larger encoding. Then, to test the solution dispenser we compared different encoding schemes, integer arrays varying from small (25 elements) to large (25,000 elements), and binary representation. Note that even though integer variables are employed to represent binary encoding (thus, a larger data type than necessary), the literature commonly presents the encoding of binary variables 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 encoding 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). For the SB, the internal dispose process is called and also the object data are cleared. We ran each set-up for five times and computed the average results. Additionally, to measure time and memory we used the integrated profiler available in 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 in Java on a quad-core Intel i7 machine with 32GB memory running Linux. The main reason for choosing Java is that it is a mature language, multi-platform and widely employed for optimisation problems with a large number of optimisation algorithms and frameworks implemented and available for public use [4–6].

Figure 7 presents the results of the computation time. The red lines represent the time spent in

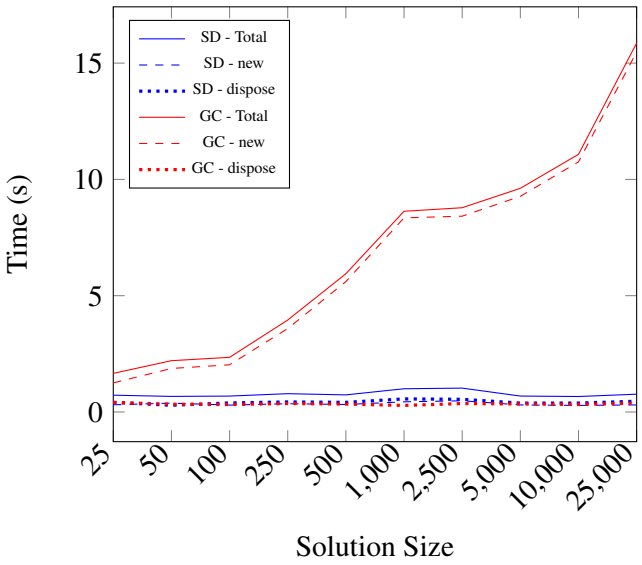


Figure 7.: Time comparison between the Solution Dispenser (SD) and the Java Garbage Collector (GC) to instantiate and dispose new solutions.

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. The time spent by the SB follows a constant trend throughout all encoding or solution 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 SB), 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.

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 that 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 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 8 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, it is clear 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 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.

Finally, we conducted experiments running a Genetic Algorithm (GA) Goldberg [26] through the WSRP API to solve real-world scenarios [27]. To isolate the impact of employing the Solution Builder, we ran the experiments on a single test instance varying only the size of the data

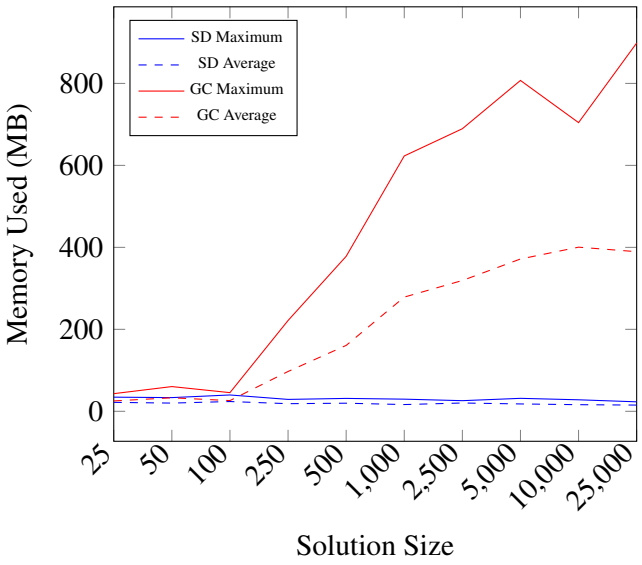


Figure 8.: Memory comparison between the Solution Dispenser (SD) and the Java Garbage Collector (GC).

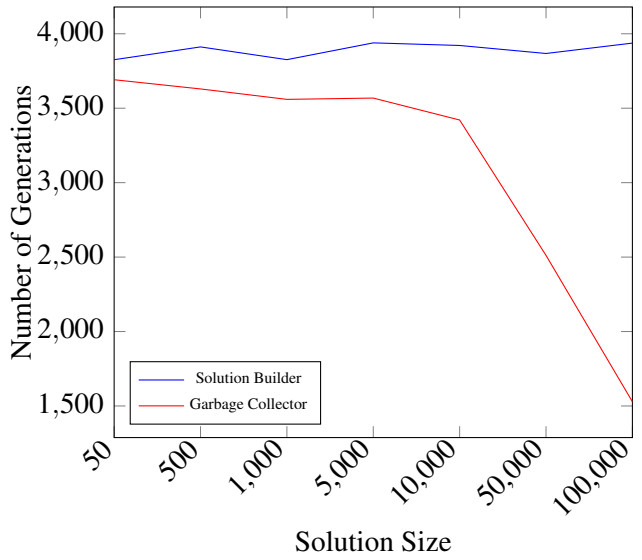


Figure 9.: Number of generations of a genetic algorithm with different solution sizes.

structure used to represent a solution. Therefore, all the experiments required the same computational effort regarding the genetic operators and solution evaluation. Also, because of the stochastic nature of GAs, we used fixed seeds for the random number generators to increase the fairness of the comparison. We performed eight runs of the GA, both for the solution builder and the garbage collector, and computed the average number of generations after one minute. Figures 9 and 10 present the results. In Figure 9, we present the total number of generations for the GA as the size of the solution representation increases. It is evident that when employing the solution builder the average number of generation is roughly constant regarding the increase in the size of the representation. It is also noticeable that if the solution size is not large enough (in this case the equivalent of 10,000 integer values), the performance gain of using the SB is below 10% (Figure 10), but still substantial. Nonetheless, when the size of the representation increases, the benefit of employing the SB represented up to 150% rise in the number of generations.

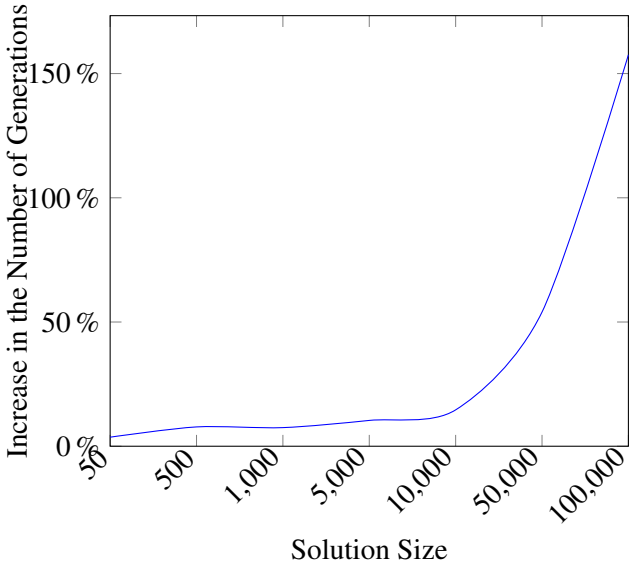


Figure 10.: Increase in the number of generation when employing the Solution Builder in detriment of the built-in Garbage Collector.

Thus, it is apparent that by using the solution builder we can achieve substantial improvements in both time and memory consumption. This is particularly true for problems where the solution encoding is large. Also, the idea of recycling objects could be implemented in the solver algorithms themselves, especially in population-based algorithms (because of the high number of created and disposed solutions), to maintain their individual object pools.

4.3. *Design Patterns*

Finally, we provide evidence that the proposed programming design patterns can further increase the computational performance of the API. This set of experiments requires extra control of the hardware to ensure consistency of results. Hence, we avoided languages with a built-in garbage collector such as Java and C#. We opted to employ the C language (C11 standard) and the gcc compiler (version 5.2.1) under Linux OS (kernel version 4.2.0-19-generic) due to explicit memory and processor control, the non-existence of a garbage collector and because we could easily extend our experiment to consider GPU processing.

Data Locality. To evaluate the data locality pattern we created two types of structures: the *local structure* with all variables contained locally, hence appearing consecutively in the physical memory, and the *external structure* where the variables are pointers to the actual values, representing references (in a similar fashion to object-oriented languages) and thus appearing on different sections of the memory.

For the experiment, we created two arrays, one for each structure, each one containing 1,000,000 structure elements. For each array we sequentially iterated through all items, in sequential order, operating elementary operations on all variables within each element of the structure (thus accessing all values). We ran the iteration 1,000 times and computed the overall processing time.

Figure 11 presents the results. We run the experiment for different sizes of structures, varying from 104 to 4,824 bytes. The x-axis shows the size of the data structure in bytes. The y-axis shows the average time of 10 runs of the experiment, in seconds. It is clear that in both cases, as the size of the data increases, the access time increases in a linear trend. However, when we employ the use of local data, we have a reduction of the computational time averaging 77%.

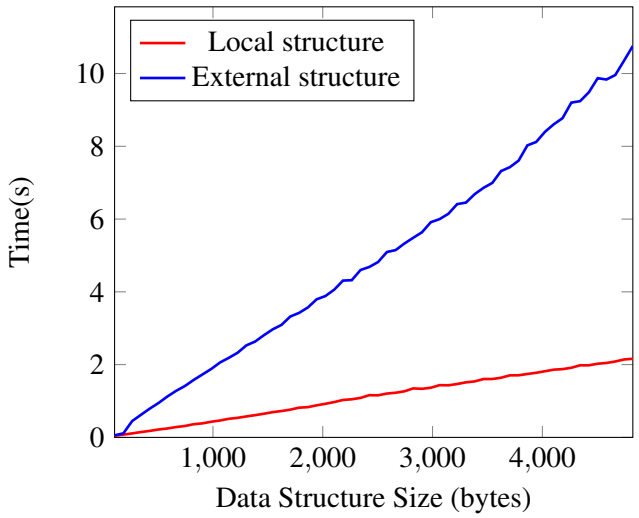


Figure 11.: Performance comparison of using values contained locally inside a structure (Local structure) against only the references to the values contained locally (External structure).

It is important to notice that this is the comparison between the best and worst cases, hence these results present the maximum potential gain. However, in languages such as Java, this represents the difference between employing primitive data types (which will be stored locally within an object) and employing object data types (which will be stored as references). These results suggest that languages that are purely objected-oriented are not the best choice regarding performance for optimisation systems.

Matrix Ordering. In order to evaluate the matrix ordering design pattern we created a matrix of elements of type double. We then sequentially iterated through all elements of the matrix 1,000 times, accessing its elements in row order and computed the total processing time. Then we repeated the process using column order.

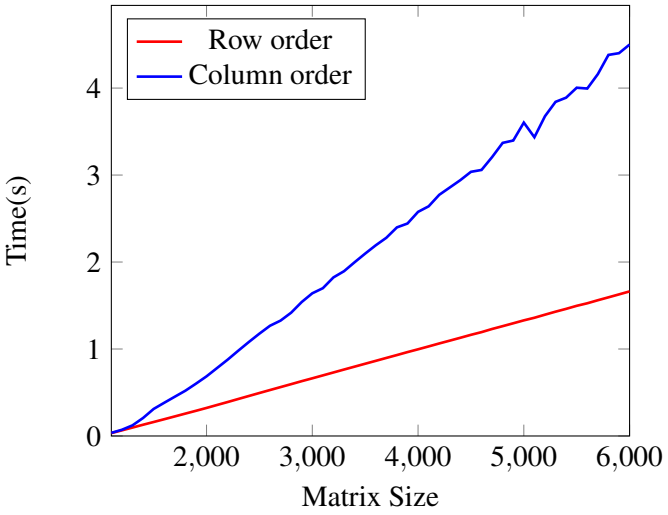


Figure 12.: Time comparison of performing sequential operations on a matrix using row order and using column order.

Figure 12 presents the results of the matrix ordering design pattern. The x-axis shows different matrix sizes in bytes, varying from 1,100 bytes to 6,000 bytes. The y-axis shows the average time of 10 runs of the experiments, in seconds. It is evident that performing the iteration in row

Table 1.: Results for the memory alignment pattern.

Manufacturer	Model	Architecture Codename	Year	Time (s)		Difference in Performance
				Aligned	Misaligned	
Intel	Core i7-4870HQ	Crystal Well	2014	4.4	5.1	17.4%
Intel	Core i5-4460	Haswell	2014	4.6	5.6	21.9%
Intel	Core i7-4770	Haswell	2013	4.0	4.9	21.8%
Intel	Core i5-4200U	Haswell	2013	6.5	8.0	22.4%
Intel	Xeon E5-2407 v2	Ivy Bridge EN	2014	8.6	10.4	21.3%
Intel	Core i5-3340	Ivy Bridge	2013	4.8	5.8	19.9%
AMD	A10-6800K	Richland	2013	6.2	8.1	30.0%
Intel	Core i7-3632QM	Ivy Bridge	2012	4.9	6.0	22.6%
Intel	Core i7-3820	Sandy Bridge E	2012	4.7	6.6	38.4%
AMD	FX-6100	Bulldozer	2012	7.8	12.0	53.5%
AMD	A6-3670	K10	2011	13.3	17.0	27.6%
Intel	Pentium Dual T2390	Merom	2007	17.9	25.9	44.8%
Nvidia	GTX 980M (CUDA 5.2)	Maxwell	2014	0.000314	0.000320	1.9%
Nvidia	GTX 750 (CUDA 5.0)	Maxwell	2014	0.000584	0.000586	0.3%

order achieved significant performance gain compared to employing column order. The time reduction averages 57%. This gain happens because the C compiler organises each line of the array sequentially in memory. Hence, when pulling data to the cache, several elements that will be read next are loaded into the cache as well. It is important to notice that these results depend on the programming language, and the results represent a potential gain. Because of that, the developer must ensure to use the right matrix ordering according to the specifications of the chosen programming language.

Memory Alignment. In order to assess the memory alignment, we created two types of structures, both consisting of a `char`, a `short`, and an `int` variable, totalling seven bytes. The *aligned* structure contained an extra byte, a padding byte, to round the structure size to eight. The *misaligned* structure did not contain the padding byte, and we ensured that by providing the `__attribute__((packed))` directive to the compiler because the C compiler automatically aligns misaligned structures.

The performance test used is similar to the one employed on the data locality pattern: we created two arrays, one for each type of structure, each one containing 1,000,000 elements. For each array we iterated through all the elements, in order, performing elementary operations on all variables of each structure element. We repeated the process 1,000 times and computed the total processing time.

Because the performance of memory alignment is dependent on the CPU architecture, we performed the experiments on multiple CPUs, on multiple architectures, and recorded the computation times. Also, we used the same OS and gcc versions in all experiments, and each executable was locally compiled to obtain architecture-specific optimisations. Also, because nowadays GPU computing is used for high-performance computing [28], we performed the experiments on two GPUs as well.

Table 1 presents the results. The first column lists the manufacturer of the chipset: for CPUs, we have Intel and AMD and for GPU we have Nvidia. Column *Model* gives the specific model of the processor and column *Architecture Codename* gives the version of the architecture used in the CPU (note that different processors may share the same architecture). Column *Year* shows the year in which that processor was released. The double column *Time(s)* shows the computational time (in seconds) for the aligned and misaligned structures. The last column gives the performance gain by applying aligned structures.

The results show that, for CPUs, there are benefits from using aligned structures. It is clear

that on older architectures that gain is more substantial, going up to 53.5% on the AMD Bulldozer (2012) and 44.8% on the Intel Merom (2007). However, it is also clear that on recent architectures, the manufacturers are minimising the impact of using misaligned structures. The two bottom rows of the table present the GPU results. It is important to notice that we made full use of the GPU multithread computing capability in these experiments. Hence, the processing times are just a fraction of the processing times for the CPUs. It is clear that GPUs do not suffer significant impairments from using misaligned structures. However, it might be the case that the CUDA compiler did not recognise the pack directive sent to the compiler and the data structures were padded regardless.

Even though modern processors handle misaligned structures more efficiently, there is substantial performance gain from using aligned data. Nonetheless, to assess memory misalignment, we had to force the compiler to accept misaligned structures. Most programming languages automatically add padding bytes to align such structures, hence, unless a developer is using legacy systems or programming on low-level languages such as assembly language, memory alignment issues can be ignored.

Overall. The previous experiments showed the potential gain from using the computational efficiency design patterns presented in this work. However, these experiments consider scenarios where the evaluated features are isolated, which is unlikely to happen in practice. Hence, the answer to questions such as *how important is the application of the proposed design patterns on optimisation algorithms?* is still not clear. Therefore, we implemented a straightforward optimisation algorithm for a simple, well-known problem in two versions: one employing data locality and matrix ordering, and another one not using these patterns. We opted to ignore the memory alignment for the reasons aforementioned.

The optimisation problem considered in this experiment is a Multidimensional Knapsack Problem (MKP) [29] with n items $(i = 1,\ldots,n)$, m weights w_j^i $(j = 1,\ldots,m)$ and p profits c_k^i $(k = 1,\ldots,p)$. A set of items must be selected for packing in the knapsack in order to maximise the p profits while not exceeding the capacities W_j of the knapsack. This problem can be formulated as follows:

$$\begin{aligned} \text{maximise} \quad & \sum_{i=1}^{n} c_k^i x_i && k = 1,\ldots,p \\ \text{subject to} \quad & \sum_{i=1}^{n} w_j^i x_i \leq W_j && j = 1,\ldots,m \\ & x_i \in 0,1 && i = 1,\ldots,n \end{aligned}$$

The solution representation for the above problem is a straightforward binary array representing the decision variables. The problem data were generated with random values of a uniform distribution [30]. The optimisation algorithm is a simple random search [31], where solutions are randomly generated, evaluated, and if the current solution is the best so far, then it is kept, otherwise it is discarded. The stop criterion is when the algorithm reaches a pre-defined number of solutions generated.

The experiment consists of performing the random search for 10,000 solution generations and recording the total computation time. We compared an implementation A, using data locality and matrix ordering against a second implementation B that ignored these patterns. The remaining of the application was kept identical. We performed the experiments for different number of items varying from 100 to 10,000. Figures 13 and 14 present the results.

Figure 13 shows that by employing the patterns, the computational time is consistently smaller. Figure 14 indicates that the performance loss of not using the patterns can be as high as 60% and it increases with the size of the problem instances. Therefore, even in an optimisation algorithm

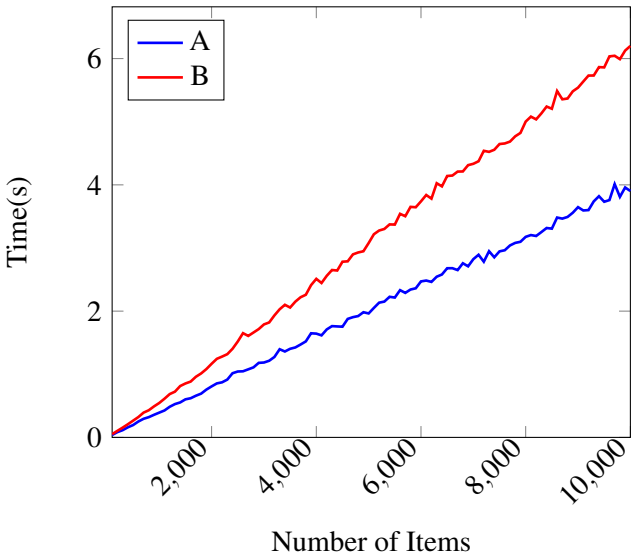


Figure 13.: Results for 10,000 generations of MKP solutions using a random search employing an implementation using the data locality and matrix ordering design patterns (A) and an implementation not using these patterns (B).

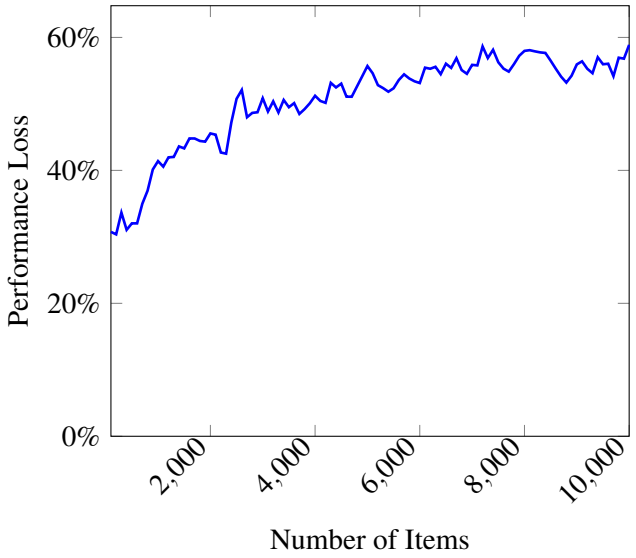


Figure 14.: Performance loss for not employing the data locality and matrix ordering design patterns.

environment, caring for matrix ordering and data locality can outcome substantial performance gains. Certainly, when combining these techniques with the SB, a higher performance increase can be achieved. Hence we recommend the use of the proposed techniques based on our experiments and the results obtained.

5. Conclusion

It is difficult to provide an API to handle the data of an optimisation problem that is generic enough to support many variants of the problem (hence be widely useful) and be computationally efficient. Therefore, in this work, we proposed a methodology to help researchers to design

and implement a tailored API for specific optimisation problems. Initially, we proposed the use of a data parser to read the files from disk and load them into memory. We defined a set of data structures to hold the data in memory and provided examples of methods and operations that further increase the usefulness of the API. We brought programming design patterns to our model to improve its effectiveness, such as the Data Locality, Dirty Flag and Object Pool. Also, we proposed efficient techniques to improve matrix access performance and to reduce cache misses during the execution of the code. Finally, we introduced a Solution Builder which centralises the objective function, hence promoting fairness on the comparison of solutions arising from different solving techniques. The Solution Builder also provides an object repository that handles the memory allocation and disposal of memory objects to avoid garbage collector calls.

In order to illustrate the application of the API, we used a real-world R&D project to develop DSS based on an optimisation solution for the WSRP, originating from a partnership between The University of Nottingham and a software development company. We discussed the benefits identified during the development of this project, such as

- **reduction of re-work**, because researchers and development team were using common implementations;
- **in the research team efficiency**, because they were not required to implement necessary (but time-consuming) programs to parse and process data;
- **in computational efficiency**, because all researchers used the best implementations known for the data structures and data access and
- **fair comparison** between the different techniques developed.

We also presented the empirical results from computational experiments with the solution builder, using optimisation algorithms to show that substantial performance can be gained by employing the technique instead of relying on garbage collection. We also demonstrated that the utilisation of the proposed design patterns further increases the performance of optimisation algorithms.

In summary, the methodology presented in this work is a tool for both researchers and practitioners to obtain improved results on R&D projects based on optimisation problems. Also, the API helps to enhance the performance of the software developed while researchers with weaker programming skills can still achieve improved computational performance in their algorithms. Future work includes applying the methodology on additional R&D projects and formally measure the impact of using the API in the software development life cycle.

References

[1] D. J. Power, F. Burstein, and R. Sharda. Reflections on the past and future of decision support systems: Perspective of eleven pioneers. In *Decision Support*, volume 14 of *Annals of Information Systems*, pages 25–48. Springer New York, 2011. ISBN 978-1-4419-6180-8.

[2] M. Pinedo. *Scheduling: theory, algorithms, and systems*. Prentice Hall international series in industrial and systems engineering. Prentice Hall, 2002. ISBN 9780130281388.

[3] B. L. Golden, S. Raghavan, and E. A. Wasil. *The Vehicle Routing Problem: Latest Advances and New Challenges: latest advances and new challenges*. Operations research/computer science interfaces series. Springer, 2008. ISBN 9780387777788.

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

[5] jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771, 2011. ISSN 0965-9978.

[6] Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. Opt4J - A Modular Framework for Meta-heuristic Optimization. In *Proceedings of the Genetic and Evolutionary Computing Conference (GECCO 2011)*, pages 1723–1730, Dublin, Ireland, 2011.

[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.*, pages 233–240, 2014.

[8] 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.

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

[10] 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)*, 3(4), 2012.

[11] R L. Pinheiro, D. Landa-Silva, R. Qu, E. Yanaga, and A. A. Constantino. Towards an efficient api for optimisation problems data. In *Proceedings of the 18th International Conference on Enterprise Information Systems*, pages 89–98, 2016. ISBN 978-989-758-187-8.

[12] 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)*, pages 283–302, Son, Norway, August 2012.

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

[14] J. A. Castillo-Salazar, D. Landa-Silva, and R. Qu. Computational study for workforce scheduling and routing problems. In *ICORES 2014 - Proceedings of the 3rd International Conference on Operations Research and Enterprise Systems*, pages 434–444, 2014.

[15] 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*, pages 283–293, 2015.

[16] P. Groth, A. Loizou, A. J. G. Gray, C. Goble, L. Harland, and S. Pettifer. Api-centric linked data integration: The open PHACTS discovery platform case study. *Web Semantics: Science, Services and Agents on the World Wide Web*, 29:12 – 18, 2014. ISSN 1570-8268. . Life Science and e-Science.

[17] R. Swaminathan, Y. Huang, S. Moosavinasab, R. Buckley, C. W. Bartlett, and S. M. Lin. A review on genomics APIs. *Computational and Structural Biotechnology Journal*, 14:8 – 15, 2016. ISSN 2001-0370. .

[18] J. Walnes. Xstream, October 2016. http://xstream.codehaus.org/.

[19] M. McShaffry. *Game Coding Complete, Fourth Edition*. ITPro collection. Course Technology PTR, 2012. ISBN 9781133776581.

[20] R. Nystrom. *Game Programming Patterns*. Genever — Benning, 2014. ISBN 9780990582915.

[21] J. L. Hennessy, D. A. Patterson, and K. Asanović. *Computer Architecture: A Quantitative Approach*. Computer Architecture: A Quantitative Approach. Morgan Kaufmann/Elsevier, 2012. ISBN 9780123838728.

[22] S. Oaks. *Java Performance: The Definitive Guide*. O'Reilly Media, 2014. ISBN 9781449363543.

[23] 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*, CASES '00, pages 9–17, New York, NY, USA, 2000. ACM. ISBN 1-58113-338-3.

[24] 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.*, 38(7):81–92, June 2003. ISSN 0362-1340.

[25] M. Yener, A. Theedom, and R. Rahman. *Professional Java EE Design Patterns*. Wiley, 2014. ISBN 9781118843451.

[26] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Artificial Intelligence. Addison-Wesley Publishing Company, 1989. ISBN 9780201157673.

[27] R. L. Pinheiro, D. Landa-Silva, and J. Atkin. *A Variable Neighbourhood Search for the Workforce Scheduling and Routing Problem*, pages 247–259. Springer International Publishing, Cham, 2016. ISBN 978-3-319-27400-3.

[28] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W.-M. Hwu. Gpu clusters for high-performance computing. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–8, Aug 2009.

[29] T. Lust and J. Teghem. The multiobjective multidimensional knapsack problem: a survey and a new approach. *CoRR*, abs/1007.4063, 2010.

[30] R. L. Pinheiro, D. Landa-Silva, and J. Atkin. Analysis of objectives relationships in multiobjective problems using trade-off region maps. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 735–742, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3472-3.

[31] F. J. Solis and R. J.-B. Wets. Minimization by random search techniques. *Mathematics of Operations Research*, 6(1):19–30, 1981.