Discrete Real-world Problems in a Black-box Optimization Benchmark

Sebastian Raggl
Research Group HEAL
FH Upper Austria
Hagenberg, Austria
sebastian.raggl@fh-ooe.at

Andreas Beham
[1] Research Group HEAL
FH Upper Austria
Hagenberg, Austria
[2] Johannes Kepler University
Linz, Austria
andreas.beham@fh-ooe.at

Viktoria Hauder
[1] Research Group HEAL
FH Upper Austria
Hagenberg, Austria
[2] Johannes Kepler University
Linz, Austria
viktoria.hauder@fh-ooe.at

Stefan Wagner
Research Group HEAL
FH Upper Austria
Hagenberg, Austria
stefan.wagner@fh-ooe.at

Michael Affenzeller
[1] Research Group HEAL
FH Upper Austria
Hagenberg, Austria
[2] Johannes Kepler University
Linz, Austria
michael.affenzeller@fh-ooe.at

ABSTRACT

Combinatorial optimization problems come in a wide variety of types but five common problem components can be identified. This categorization can aid the selection of interesting and diverse set of problems for inclusion in the combinatorial black-box problem benchmark. We suggest two real-world problems for inclusion into the benchmark. One is a transport-lot building problem and the other one is the clustered generalized quadratic assignment problem. We look into designing an interface for discrete black-box problems that can accommodate problems belonging to all of the described categories as well real-world problems that often feature multiple problem components. We describe three different interfaces for black-box problems, the first using a general encoding for all types of problems the second one using specialized encodings per problem type and the last one describes problems in terms of the available operators. We compare the strengths and weaknesses of the three designs.

CCS CONCEPTS

• **Theory of computation → Discrete optimization**; *Design and analysis of algorithms*;

KEYWORDS

black-box optimization, benchmark design, real-world problems

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208280

ACM Reference Format:
Sebastian Raggl, Andreas Beham, Viktoria Hauder, Stefan Wagner and Michael Affenzeller. 2018. Discrete Real-world Problems in a Black-box Optimization Benchmark. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205651.3208280

1 INTRODUCTION

Combinatorial optimization problems arise in a wide range of industries and there is often a strong financial incentive to solve them well. As a result a staggering number of different problems and problem variants have been described over the last decades. Identifying problems and problem instances that should be part of a black-box optimization benchmark set is therefore no easy task.

To aid the selection of suitable problems we suggest five problem types or components, along with common variations, that are part of many typical discrete optimization problems. We argue that in order for the set of benchmark functions to be representative we should include problems featuring all those types because they have very different characteristics. Real-world problems often contain multiple of these components that interact in some way. How algorithms can deal with these interactions is especially interesting. Because of this we describe two hard real-world problems that feature an unusual combination of problem components. For each of the problems we provide real-world instances.

When solving discrete optimization problems we want the solution encoding to capture the structure of the problem as well as possible, because invariants in the encoding can drastically reduce the search space. The operators used should aim to preserve these invariants in the encoding. Specialized encodings and operators on these encodings have been created for problems of all the types described below. The usage of specialized encodings and operators is fundamentally at odds with the notion of a black-box problem. We propose three different approaches to dealing with this apparent incompatibility.

The rest of this article is organized as follows. In the next section we describe the problem components including typical problems as well as specialized encodings and operators. After that we present two real-world problems featuring multiple problem components. Finally we describe three possible interfaces to black-box problems that enable representing discrete and combinatorial problems of all the described problem components.

2 PROBLEM COMPONENTS

The problem components presented in this section are chosen to capture meaningful differences in problem characteristics over a wide variety of problems. The problem components are partially overlapping, because depending on how a problem is framed it might appear to fall into different categories. The resulting choice of encodings and operators is by no means irrelevant as is shown by the large number of publications about specialized algorithms for different problem types.

2.1 Assignment

In this type of problem the task is to find the best assignment of two or more sets (in a mathematical sense) of items. The solution space is mainly described by the cardinalities that are possible in the assignment. The cardinalities on both sides can indicate optionality, identity, and multiplicity resulting in the following choices:

- Multiple items of set 1 may be assigned to an item of set 2
- An item of set 1 may be assigned to multiple items of set 2
- Not all items of set 1 need to be assigned
- Not all items of set 2 need to be assigned

Since the first simple one to one assignment problem (AP) has been formulated a large number of problem variants including the bottlenecked AP, the generalized AP and the quadratic AP have been created [4, 10].

Assignment problems can generally be encoded directly using a list of sets where every set contains all the items assigned to one target. If at most one set of items can be assigned to multiple items of the other set, the encoding can simply be a vector where every position represents one item and the value at this position is the target this item is assigned to. For assignments where items can be assigned to multiple target and targets can have multiple items assigned to them, a fixed size encoding can still be used, if we view the problem as selecting from the set of all possible combinations of items and targets, which is described in Section 2.2.

2.2 Selection

In selection problems, the task is to find a subset of a set of items. Selection is a special case of assignment where we can assign each item either to the set of selected items or to the set of unselected items. Here we consider:

- Some items must never or always be selected together.
- A certain minimum and maximum number of items need to be selected.

The classical problem of this type is the knapsack problem [9]. The encoding is typically a binary vector encoding.

2.3 Grouping

In this type of problem, the task is to partition a set of items into groups. The main difference to the assignment problems, described in Section 2.1, is that groups are anonymous. While we can assign all items to target '2', but we cannot group all items into group '2', instead we group all items into a single anonymous group. Here we consider:

- There must be a certain minimal and maximal number of groups
- The groups may have a minimum and maximal size
- Some items may not be placed in the same group with others

Grouping problems are often related to graph theory with the classical example being the graph colouring problem. There are several algorithms that use encodings and operators specially designed for grouping problems. The most well known one is certainly the grouping genetic algorithm [5]. There are also grouping variants of particle swarm optimization [8] and evolution strategies [7]. The linear linkage encoding along with several new operators has been defined in [14].

2.4 Sequencing

A sequencing problem asks us to find the best order for a number of given items. In general, a permutation of items can be used to describe the solution space, however some properties, especially regarding the identity of two sequences can be defined a priori if a choice in each of the following statements is made:

- All items are unique or some items are identical to each other.
- The sequence either starts at the beginning or it has no start.
- The sequence is interpreted from left to right or the direction does not matter.

The classical problems here are the travelling salesman problem [13] and the job shop scheduling problem [2].

Specialized encodings for these problems include the permutation encoding as well as the random key encoding. While the random key encoding normally uses real-valued vectors there is no reason the same idea cannot also be applied to integer vectors.

2.5 Value Picking

This is the most general problem category, where we have a fixed number variables, that each need to be assigned a value out of a given domain. A typical constraint would be that a linear combination of values must be smaller than some limit. Every other type of problem described above can be viewed as a value picking problem.

3 REAL-WORLD PROBLEMS

While classical problems typically fall into a single category real-world problems often belong to multiple of the categories described in Section 2.

Many real world scheduling problems, consist of an assignment and a sequencing part and so does the vehicle routing problem [15].

Apart from real-world problems that naturally feature multiple problem components there are artificial problems designed to study the interactions that occur when multiple types of problems are combined. The best known problem of this type is probably the

travelling thief problem[3]. It is a combination of a selection and sequencing problem, more specifically of the travelling salesman and the knapsack problem. There exist comprehensive benchmark sets [11].

We would also like to suggest two real-world problems we have encountered that have interesting properties. In the Appendix we include a collection of problem instances for both problems.

3.1 Transport-lot Building Problem

This problem was first published together with a model based evolutionary algorithm for solving it [12]. There are items stored in stacks and only the items on top of a stack are accessible. All the items must be moved by a transporter with limited capacity. While moving the items to the transporter we should never need to move another item out of the way. Some items must not be in the same transport-lot at all while for others just incur some penalty. The goal is to find the smallest number of easily retrievable transport lots with minimal penalties.

There are n items in the set $\mathcal{I} = \{i_1, ..., i_n\}$ that need to be grouped into an ordered set of groups $\mathcal{S}$. An undirected weighted graph $G_p = (\mathcal{I}, \mathcal{R})$ describes the relationships between the items. The vertices of this graph are items and the weights of the edges represent the costs of putting two items into the same group. The function $C(\mathcal{S}) \rightarrow \mathbb{R}$ calculates the costs of a group using the weights on G_p. The goal is to find the minimal number of transport lots with the lowest total cost.

The problem originated in the operation of a continuous caster at a steel plant. The items are steel slabs that are produced need to be processed further or stored at different locations depending on a number of physical properties. In this setting it is more important to maximise transporter utilisation, but it can be preferable to group some items into a separate transport lot instead of distributing them. We therefore choose to add the two objectives together because this means we can tune this by adjusting the weights on G_p.

$$\min \ |S| + \sum_{s \in \mathcal{S}} C(s) \tag{1}$$

$$s.t. \ (a, b) \in R \qquad \forall_{s \in \mathcal{S}} \, \forall_{a \in s, \ b \in s} \tag{2}$$

$$S(a) \leq S(b) \qquad \forall_{(a,b) \in \mathcal{D}} \tag{3}$$

$$|s| \leq N \qquad \forall_{s \in \mathcal{S}} \tag{4}$$

Two items that are connected by an edge in G_p with a large weight should not be part of the same lot, but if they are not connected by an edge in G_p at all, constraint 2 ensures that they are not in the same group. The order of the groups is determined by dependencies between items, which are described by a directed graph $G_d = (\mathcal{I}, \mathcal{D})$. An item depends on another when it lies below the other item in the same stack. If an item a depends on another item b, item a must be either in the same group as b, or in a group with a higher index in $\mathcal{S}$. Such a dependency is modelled as an edge in the dependency graph $(a, b) \in D$. This is enforced by constraint 3 with the help of the function $S(\mathcal{I}) \rightarrow \mathbb{Z}$, which maps an item to the index of the group it belongs to. The order of groups that do not depend on each other is irrelevant. Constraint 4 limits the maximum size of transport lots to the capacity of the transporter.

In order to generate random but realistic problem instances of any size we suggest the following procedure. Choose the number of items and the maximal transport lot size. Based on that choose the number of item types and number of stacks such that the number of items. Then create the desired number of items and randomly assign it a number between zero and the number of item types. The cost of putting two items in the same lot is determined by the difference between their type numbers plus some small random value. All items are then randomly distributed to different stacks and the positions in the stacks determine the dependencies. The number of items on every stack and of every type should, on average, be a few times larger than the maximal number of items in a transport lot. This ensures that both the grouping costs as well as the dependency constraints are relevant to the solver.

This problem can either be seen as an assignment problem or as a combination of a grouping and an sequencing problem. Viewing it as a combined problem can enable more efficient solvers because the sequencing part can be solved in linear time. Given a grouping of the items, a group dependency graph can be derived from the item dependencies and any topological sorting of this group dependency graph is a valid solution. This is why the solver used in the orignal paper uses a linear linkage encoding and group based crossover and mutation operators.

3.2 Clustered Generalized Quadratic Assignment Problem

This extension of the generalized quadratic assignment problem was introduced together with a mixed integer linear program and two different linearisations for solving it [6]. The problem was first encountered at a steel manufacturer where steel slabs are stored on stacks in multiple yards. It is relevant to warehousing in general where products can be stored at locations in different facilities. It is a 1:n assignment problem where the goal is to assign items to one or multiple locations taking capacity constraints into account. Locations are clustered into storage areas and there are distances between the locations. The same item must not be stored in locations belonging to different storage areas. Given the probability that two pieces of items are used together, the goal is to minimize the expected travel distance during processing as well as the number of storage areas used.

$$\min \ \sum_{i,k,j,h} d_{ij} w_{kh} x_{ik} x_{jh} + \delta \sum_{a \in A} u_a \tag{1}$$

$$s.t. \ \sum_{k \in N} x_{ik} c_k \geq r_i \qquad \forall i \in M \tag{2}$$

$$\sum_{i \in M} x_{ik} \leq 1 \qquad \forall k \in N \tag{3}$$

$$x_{ik} x_{ih} \leq a(k, h) \qquad \forall k, h \in N, i \in M \tag{4}$$

There is a set of locations N, a set of items M and a set of areas A. The binary decision variable x_{ik} is one when item i is assigned to location k and zero otherwise. The costs of assigning item i to location k while also assigning item j to location h is determined by the distance between the locations d_{kh} and the probability that they are processed together w_{ij}. The value of u_a is one if any item

is assigned to a location belonging to the area a and zero otherwise. More formally the problem can be defined as:

The goal is to minimize the total costs as well as the number of used areas. Using an area has costs of δ. Every location has a capacity c and every item has requirements r and constraint 2 ensures that all requirements are fulfilled. Every location can only contain a single item as specified in constraint 3. Finally to ensure that no item is assigned to locations in different areas, constraint 4 uses a function a which takes two locations and returns one if they are in the same area and zero otherwise.

This is a NP-hard problem just like the QAP and the GQAP. What makes it interesting is that while it is clearly an assignment problem it also has a grouping aspect. Like in many grouping problems, the goal is to find a solution using as few groups, in this case called areas, as possible. But instead of grouping the items directly, they are grouped via the locations they are assigned to. This means that two components of the problem cannot be solved independently but instead must be considered simultaneously.

4 INTERFACES TO BLACK-BOX PROBLEMS

Defining the interface of a discrete black-box problem involves making a lot of decisions. We describe three areas that must be considered:

- Constraint violations
- Problem dimensions
- Solution encodings

How to deal with constraint violations. Combinatorial problems often feature hard constraints and solutions violating such a constraint are considered infeasible. There can be a large difference in the number of feasible solutions between instances of the same problem. There is a choice of how to handle solutions that violate the constraints. Either a hard distinction between feasible and infeasible solutions is drawn or constraint violations are factored into the solution quality. If there is the concept of an infeasible solution, either it is only reported that a constraint was violated or a count of violations is provided. The shape of the fitness landscape has strong influence on algorithm performance and simply declaring a solution infeasible without any indication how far away it is from a feasible solution leaves algorithms without a search direction. We therefore suggest to at least provide information about the number of violated constraints. It can also help to report the severity of a violation but this is very much problem dependent. Whether the violations are reported separately or factored into the quality is much less important. However we suggest reporting them separately, because it is easy go from a quality value and a count of violations to a combined quality, but not the other way around.

Problem dimensions. For real-valued black-box problems, the search space can be described as a hypercube with a fixed number of dimensions. Every vector inside of this hypercube describes a valid solution to the problem with some associated quality and every vector outside does not represent a solution. Therefore, the search space always has the same size and this size only depends on the number of dimensions.

For combinatorial problems the size of the search space depends heavily on the type of problem. Figure 1 shows the number of unique solutions for different types of problems, depending on the number of items. While the assignment of n items to 10 targets has the most unique solutions for problem sizes up to ten, it is quickly overtaken by an assignment of n items to n targets. The number of possible sequences and possible groupings eventually outgrows the number of assignments to any fixed number of targets but always grows slower than the number of assignments to n targets. This is because both sequences and groupings can be viewed as assignments to n targets where many solutions are either invalid or equivalent.

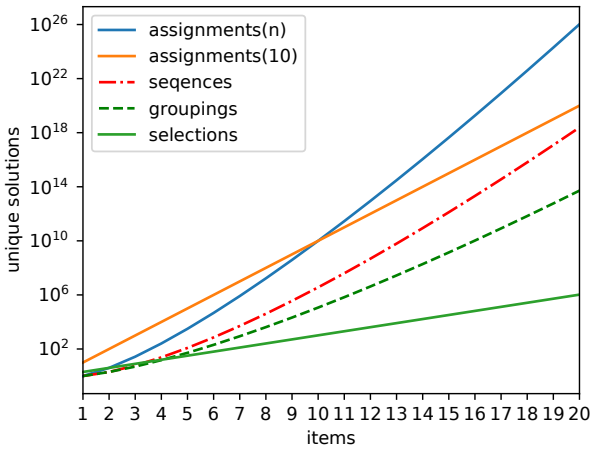

Figure 1: The number of unique solutions by problem type.

How to encode solutions. The number of possible values, given some encoding, is not necessarily equal to the number of unique solutions. For example using a integer vector encoding, in order to encode a valid solution to a sequencing problem, the vector must not contain duplicate values, otherwise it does not uniquely identify a sequence. A one to one assignment problem also requires the vector to not contain duplicated values. There are a number of possible strategies for dealing with values that do not encode a solution:

- Report the vector as invalid. This has the same problems as described before with constraint violations.
- Interpret the reasons that it is not a solution as constraint violation and report them accordingly. So in the case of a permutation, every repeated value would count as one constraint violation. This only works for simple cases.
- Use encodings that do not contain non-coding values. An example would be the random key encoding for sequencing problems where every possible value encodes a solution. The drawback is that the search space grows dramatically.
- Describe the encoding sufficiently. Non-coding vectors can be avoided by describing the conditions for a vector to be interpretable as a solution, so an optimizer can choose operators that avoid producing invalid vectors.

Correctly identifying the type of problem can make the difference between an effective search and an unsuccessful one. For small

problems, it may even be the difference between being able to enumerate every solution and only trying a tiny fraction of the possible solutions. This is the reason why so many papers have been written about specialized encodings and operators for different problem types [2, 5, 14].

The problem sizes must be chosen so that the solvers have a chance to find interesting solutions within the allowed evaluation budget. But the relationship between problem size and search space is very different for different problem types and also depends on the encoding. Therefore the question is, how can we design a interface to a black-box problem so that we can compare optimizers on all types of problems?

How to design an interface to a black-box problem. The simplest possible interface is a single evaluation function that takes a list of integers as input and outputs a quality value or a signal that the solution is infeasible or not a solution at all. This reveals very little information about the problem upfront and likely results in many algorithms either being inapplicable because they require fixed sized encodings or simply failing to find any solution at all within the allowed evaluation budget. Algorithms that can solve problems given such a limited interface are likely to contain some parts specifically tailored to gain information about the problem. Like systematically trying lists of different sizes to determine the allowed length and value ranges. Algorithms are also encouraged to determine through trial and error in which range the values can be and whether or not duplicated values are allowed or if the order is significant. Such elements are not present in algorithms usually used to solve combinatorial problems and are therefore of little practical use.

4.1 Interfaces Using a General Encoding

Solutions to set-oriented problems such as grouping and assignment problems are most naturally modelled using variable sized sets or lists of integers. However all the types described above can be represented as a fixed sized integer vector. Every index of the vector stands for one item while the value at that index is the value associated with that item. How to interpret these values depends on the type of problem. The drawback is that not all possible vectors correspond to a solution to a given problem.

Let us now examine what kinds of constraints we need in order to be able to avoid non-coding vectors. Every vector that is not the right size cannot encode a valid solution, so the first restriction is the size of the vector. For every position in the vector, there is a set of values that can be used at that position. Since we can map every set to a range between zero and the number of elements in the set, we simply need to specify the maximum value. For value picking problems we can have a different range for every item, so we need to specify a maximum for every position. All the other problems generally use the same maximum across the entire vector.

As described earlier, some problems allow the same value to appear multiple times in the vector and some problems do not, so this is also a constraint. These three constraints, the length of the vector, the ranges that constrain the allowed values and whether or not repetition of values is allowed are actually enough to avoid non-coding vectors for any problem that belongs to one of our problem categories. A vector that meets these constraints can still encode a solution that is infeasible for a given problem instance, but it will always encode a solution.

Problem type	Values	Range	Repetition	Solutions
Value picking	Value	$0..x_i$	yes	$\prod x_i$
Assignment	Target	$0..m$	yes | no	$m^n \mid \frac{m!}{(m-n)!}$
Sequencing	Position	$0..n$	no	$n!$
Grouping	GroupId	$0..n$	yes	B_n
Selection	Selected	$0, 1$	yes	2^n

Table 1: Integer vector encoding for different problem types

Table 1 describes for all the problem types what the values in the vector represent, in which range the values have to be, whether or not the vector can contain the same value twice and what that means for the maximum number of unique solutions. The variable n generally stands for the number of items while m stands for the number of assignment targets. B_n is the n^{th} bell number, which is the number of possible partitions of a set of size n.

While assignment and grouping can have the same range and repetition the number of distinct solutions is different. The difference comes from the fact that in grouping problems many vectors can describe the same solution because the group numbers can be chosen arbitrarily. This is primarily a concern with grouping and sequencing problems. For example a TSP solution can be described in n different ways where n is the number of cities just by altering the starting point. Describing vector encodings that uniquely identify every solution in a generic fashion is a lot harder to do than describing encodings that avoid non-coding vectors. It is also a lot less useful, because a encoding does not help much if there are no operators for it.

Problems that contain multiple problem components like the ones described in Section 3 can also be handled with the approach presented here by simply concatenating the encodings of the different parts. The length of the new encoding is simply the sum of the lengths of the parts. The valid ranges and whether or not repetition is allowed must be described separately for every part of the combined encoding. While encodings of this type have the advantage of being very simple to describe, they are often not a very natural way to describe a solution.

4.2 Interfaces Using Specialized Encodings

Many combinatorial optimization problems are fundamentally about sets of items, especially the ones falling into the categories grouping and assignment. In this section we explore how to describe encodings to a very wide range of problems using only a few concepts. The approach laid out here is similar to the modelling language of a commercial mathematical optimization solver called LocalSolver [1].

There are integer variables and there are ranges. Every variable has a range of values that can be assigned to it. There are collections which contain multiple variables or collections of the same type. The number of elements in a collection is again constraint by a range, if that range contains only one possible value the size of the collection is fixed.

There are several different kinds of collections defined by a few properties. A collection can require all contained variables to have unique values. The position of a value within a collection can either be significant or it can be insignificant. So based on these two choices there are four different types of collections. Collections without a restriction on the uniqueness of values are called lists if the position of a value is significant and multiset if it is not. A collection of unique values where order does not matter is a set, while if the order also matters it is called a permutation.

Using only those four collections we can describe many problems, but we cannot model value picking problems, where we need a fixed number of variables with distinct value ranges. We also cannot model combined problems that require multiple different collections to describe the parts of the solution. This can be fixed by adding a tuple type that can contain an arbitrary but fixed number of different variable and collection types.

In order to describe grouping problems and assignment problems that do not ask for one to one assignments, we need to be able to express two additional closely related concepts. The first one is disjointness and the second is partitioning. A collection of collections is disjoint if there is no value that appears in more than one of the inner collections. A collection of collections forms a partition if it is disjoint and every value in the range is contained.

This are all the concepts needed to describe the solution spaces of a wide range of combinatorial and discrete problems. In order to illustrate the usage we will describe the solutions to some famous problems, as well as the problems described in Section 3, using the concepts introduced above. The notation for the examples is as follows first is the name of the collection and then in parenthesis the parameters separated by a comma. The first parameter is the type of elements in the collection and it can either be a collection or a variable. All variables have ranges from zero to a certain maximum value and they are referred to by their maximum value. The second parameter to every collection is the range of allowed sizes. If the size is fixed we could write the range as $n..n$ but instead we can abbreviate this to just n.

Listing 1: Well-known problems described using the introduced concepts.

```
// knapsack
list(1, n)

// job shop scheduling
permutation(n, n)

// graph colouring problem
set(set(n, 1..n), 1..n, partition)

// vehicle routing problem
list(permutation(n, 0..n), n, partition)

// value picking
tuple(a, b, ...)

// transport-lot building (3.1)
permutation(set(n, 1..n), 1..n, partition)

// CGQAP (3.2)
list(set(n, 1..n), m, disjoint)
```

4.3 Interfaces Based on Operators

In all of the previously described interfaces, the problems are entirely defined by the evaluation function that takes a solution encoded in some way and returns the quality and/or the number of constraint violations. The main question is how much information about the problem is revealed through the solution encoding. There is an alternative approach that completely sidesteps this issue by not providing any externally visible encoding. In this model instead of defining a problem by the evaluation function, it is defined by the operators it offers. An operator is simply a function that takes a certain number of solutions and returns a new solution. There are different kinds of operators:

Creation operators. They take no arguments and when called, they create a new random solution, call the evaluation function on it and return the result. Every problem must have at least a single creation operator, otherwise no solutions can ever be created. Using only creation operators, different random sampling algorithms can be implemented.

Mutation operators. They take a solution and return a slightly modified version. They can be used to implement various evolution strategies.

Crossover operators. They take two solutions and combine them into a third. They are used in genetic algorithms.

Neighbourhood iterators. They take a solution and return an iterator over all the neighbouring solutions as defined by some neighbourhood. So it is similar to the mutation operator, with the key difference that mutation returns any neighbouring solution while the neighbourhood iterator eventually returns all neighbouring solutions.

Listing 2: Possible API

```
class Solution {
  SolutionId id;
  double GetQuality();
  int GetViolations();
}

interface Neighbours{
  Solution Next();
}

interface Problem{
  Set<CreatorId> GetCreators();
  Set<CrossoverId> GetCrossovers();
  Set<MutatorId> GetMutators();
  Set<NeighboursId> GetNeighbourhoods();

  Solution Create(CreatorId);
```

```
    Solution Mutate(Solution, MutatorId);
    Solution Crossover(Solution, Solution,
                       CrossoverId);
    Neighbours Neighbours(Solution,
                          NeighboursId);
}
```

The code in Listing 2 shows a possible API. The interface does not contain an explicit evaluation method. Instead every operator returns evaluated solutions. Every solution consists of some id as well as the associated quality and number of constraint violations. The SolutionId either somehow encodes a solution or it is simply a index into the list of previously created solutions. Either way, an algorithm trying to solve the problem can simply use the available operators and does not need to know anything about encodings, problem types and which operators can be used for which combination. The problem has a method for each type of operator that returns the set of valid operators of this type. The operator ids are always the last parameter to operator methods and define which of the available operators should be called. The reason to use operator ids is that choosing which operators to use at which time in the search is one of the most important things algorithms have to decide in this setting. It is therefore convenient to have a unique identifier to refer to each operator, which can be used to collect statistics that guide the operator selection.

The set of available operator types can always be extended, for example an operator that combines more than two solutions might be useful. It could also be interesting to include operators that perform more than a single step, such as a hill climbing or a path relinking.

5 DISCUSSION

Now that we have presented three different choices of how to design the interface to a combinatorial black-box problem. lets carefully consider advantages and disadvantages of the different approaches. The central questions are which types of problems should be considered and how much information about the black-box problems should be revealed through its interface.

We described how providing no information about the solution space of a problem encourages the usage of very specialized algorithms that try to infer this information. Since these specialized algorithm are only useful in the context of benchmark instances we should provide this information as part of the interface and we have looked at two possible choices of what exactly could be exposed in Section 4.1 and 4.2. The former provides the same interface for all problems, namely an evaluation method taking a fixed sized integer vector and a method for providing some additional information about the allowed values. This has the advantage that a wide range of algorithms can be applied, because while there is additional information about the problem available, every algorithm can choose to ignore it. However, all algorithms have the opportunity to internally choose specialized encodings and operators according to the available information. A drawback is that the list of problems whose solution space can be adequately described is limited. In particular many real-world problems and grouping problems are not included.

The second alternative is to have a specialised interface for every type of problem where the solution space is defined by the type of the single parameter to the evaluation method. Given only a few concepts, the solution spaces of a wide variety of problems can be accurately modelled this way. The problems are still black-boxes because neither the evaluation function nor the constraints are known but they are less of a black-box than with the general encoding. It becomes possible for algorithms to choose operators based on the properties of the solution space of the problem that is solved. A drawback is that it is no longer possible to apply every algorithm to every problem. Measurements of algorithm performance in this setting are more comparable because we are not comparing algorithms that use knowledge about the search space versus ones that do not. Algorithms that perform well in this setting are also more interesting for real-world applications, because in the real world we always have at least that much information about a problem.

It is also possible to provide both interfaces since it is relatively easy to translate from a more specialised encoding into a general vector encoding. It could also be interesting to provide both interfaces to all problems and compare how the solver performance changes with the additional information.

In Section 4.3 we looked at describing the problems in terms of available operators. This is very different from the first two interfaces and brings a change of perspective. The focus changes from low level considerations about encodings and operators, to a high level view centred around search strategies. All problem types can be included in the benchmark set without major differences in how well the encoding captures the solutions space. This is especially significant for real-world problems where the invariants are difficult to describe generically. Algorithms are easy to implement because it is not necessary to write and choose specialized encodings or operators. A drawback is that algorithms are limited to the operators defined for a problem. This can be mitigated by carefully choosing the set of operators to enable a broad range of algorithms, but it can never be avoided completely. However, accepting this limitation means that every algorithm be applied to every problem without any changes. By using this interface we can collect a lot of information about each algorithm. We know exactly which operators an algorithm has used at which point in the search and what that means for the solution quality. This information can be the basis for some very interesting analysis that is much harder to do using one of the previously described interfaces.

6 CONCLUSION

We identified five important types of combinatorial optimization problems, namely assignment, sequencing, selection, grouping and value picking. Every type of problem has different solution space characteristics and capturing those is a big part of what makes an algorithm successful. Classical well studied optimization problems typically fall into a single type while many real-world problems feature elements of multiple types. This makes them harder to solve in a black-box setting because of the high number of constraints this imposes on solutions. We therefore suggested two real-world problems for inclusion in the benchmark. The problems are a grouping problem that features a sequencing element as well as assignment

problem that has features of classical grouping problems. The appendix contains problem instances for both problems.

All three types of interfaces that black-box can offer to solvers have their respective strengths and weaknesses. Which interface should be presented depends on the exact goals of the benchmark. We think that choosing an interface based on the operators is very interesting because it provides the opportunity to evaluate the performance of different types of algorithms on different types of problems. Which operator types can be used as well as which specific encodings and operators should be available per problem must definitely be subject to further discussion. If real-world problems like the ones we presented should be part of the benchmark set and the focus is more on algorithm implementation than on algorithm design we strongly suggest a interface using specialized encodings.

ACKNOWLEDGMENTS

The work described in this paper was done within the COMET Project Heuristic Optimization in Production and Logistics (HOPL), #843532 Funded by the Austrian Research Promotion Agency (FFG).

REFERENCES

[1] Thierry Benoist, Bertrand Estellon, Frédéric Gardi, Romain Megel, and Karim Nouioua. 2011. LocalSolver 1.x: a black-box local-search solver for 0-1 programming. *4OR* 9, 3 (25 Mar 2011), 299. https://doi.org/10.1007/s10288-011-0165-9

[2] Jacek Błażewicz, Wolfgang Domschke, and Erwin Pesch. 1996. The job shop scheduling problem: Conventional and new solution techniques. *European journal of operational research* 93, 1 (1996), 1–33.

[3] M. R. Bonyadi, Z. Michalewicz, and L. Barone. 2013. The travelling thief problem: The first step in the transition from theoretical problems to realistic problems. In *2013 IEEE Congress on Evolutionary Computation*. 1037–1044. https://doi.org/10.1109/CEC.2013.6557681

[4] Rainer E Burkard, Mauro Dell'Amico, and Silvano Martello. 2009. *Assignment problems, revised reprint*. Vol. 125. Siam.

[5] Emanuel Falkenauer. 1994. A new representation and operators for genetic algorithms applied to grouping problems. *Evolutionary computation* 2, 2 (1994), 123–144.

[6] Judith Fechter, Andreas Beham, Stefan Wagner, and Michael Affenzeller. 2015. Modelling a Clustered Generalized Quadratic Assignment Problem. In *Proceedings of the 27th European Modeling and Simulation Symposium (EMSS 2015)*.

[7] Ali Husseinzadeh Kashan, Ali Akbar Akbari, and Bakhtiar Ostadi. 2015. Grouping evolution strategies: An effective approach for grouping problems. *Applied Mathematical Modelling* 39, 9 (2015), 2703–2720.

[8] Ali Husseinzadeh Kashan, Mina Husseinzadeh Kashan, and Somayyeh Karimiyan. 2013. A particle swarm optimizer for grouping problems. *Information Sciences* 252 (2013), 81 – 95. https://doi.org/10.1016/j.ins.2012.10.036

[9] H. Kellerer, U. Pferschy, and D. Pisinger. 2004. *Knapsack Problems*. Springer. http://www.springer.com/de/book/9783540402862

[10] David W Pentico. 2007. Assignment problems: A golden anniversary survey. *European Journal of Operational Research* 176, 2 (2007), 774–793.

[11] Sergey Polyakovskiy, Mohammad Reza Bonyadi, Markus Wagner, Zbigniew Michalewicz, and Frank Neumann. 2014. A Comprehensive Benchmark Set and Heuristics for the Traveling Thief Problem. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO '14)*. ACM, New York, NY, USA, 477–484. https://doi.org/10.1145/2576768.2598249

[12] Sebastian Raggl, Andreas Beham, Stefan Wagner, and Michael Affenzeller. 2018. Analysing a Hybrid Model-Based Evolutionary Algorithm for a Hard Grouping Problem. In *Computer Aided Systems Theory – EUROCAST 2017*, Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia (Eds.). Springer International Publishing, Cham, 347–354.

[13] Gerhard Reinelt. 1994. *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag, Berlin, Heidelberg.

[14] Özgür Ülker, Ender Özcan, and Emin Erkan Korkmaz. 2007. Linear Linkage Encoding in Grouping Problems: Applications on Graph Coloring and Timetabling. In *Practice and Theory of Automated Timetabling VI: 6th International Conference, PATAT 2006 Brno, Czech Republic*. Springer Berlin Heidelberg, 347–363. https://doi.org/10.1007/978-3-540-77345-0_22

[15] Daniele Vigo. 2015. *Vehicle routing: problems, methods and applications*. Society for Industrial and Applied Mathematics.