Predicting algorithmic complexity through structure analysis and compression

Zoltán Ádám Mann and Pál András Papp

This paper was published in *Applied Soft Computing* 13:(8) pp. 3582-3596, 2013.

Predicting algorithmic complexity through structure analysis and compression

Zoltán Ádám Mann[a,*], Pál András Papp[a]

[a] *Department of Computer Science and Information Theory*
Budapest University of Technology and Economics
Magyar tudósok körútja 2., 1117 Budapest, Hungary

Abstract

The complexity of an algorithm is usually specified by the maximum number of steps made by the algorithm, as a function of the size of the input. However, as different inputs of equal size can yield dramatically different algorithm runtime, the size of the input is not always an appropriate basis for predicting algorithm runtime. In this paper, we argue that the compressed size of the input is more appropriate for this purpose. In particular, we devise a genetic algorithm for compressing a graph by finding the most compact description of its structure, and we demonstrate how the compressed size of the problem instance correlates with the runtime of an exact algorithm for two hard combinatorial problems (graph coloring and Boolean satisfiability).

Keywords: algorithm complexity, compression, genetic algorithm, graph coloring, #SAT

1. Introduction

1.1. Motivation

When analyzing and comparing the complexity of combinatorial algorithms, the most widely-used approach in computer science is worst-case analysis [1]. That is, the algorithm's complexity is characterized by the maximum number of steps of the algorithm, as a function of the size of the input. For example, the Floyd-Warshall algorithm for computing shortest paths in a graph with n vertices runs in $O(n^3)$ time, meaning that the algorithm makes *at most* cn^3 steps on any graph with n vertices, where c is a constant [2]. In fact, this algorithm makes approximately cn^3 steps on *all* graphs with n vertices. That is, the algorithm's runtime does not de-

*Corresponding author
Email addresses: zoltan.mann@gmail.com (Zoltán Ádám Mann), papp.pal.andras@gmail.com (Pál András Papp)

pend on the specific input graph, only on its size. This is quite typical for polynomially solvable problems.

Exact algorithms for NP-hard problems are very different. Their worst-case complexity is usually exponential (e.g. $O(2^n)$). However, smart algorithms – such as branch-and-bound and similar backtrack search methods – can achieve much better performance for a significant portion of inputs. As a result, the *typical-case complexity* of such algorithms can be significantly lower than their worst-case complexity [1]. Some specific examples of this phenomenon:

- For the graph coloring problem, backtrack search has an exponential worst-case complexity, but constant average-case complexity [3, 4].
- For the satisfiability problem, state-of-the-art SAT solvers can often solve problem instances with thousands of variables and tens to hundreds of thousands of clauses in reasonable time, while their worst-case runtime is still $O(2^n)$ [5].
- For linear programming, the simplex method is known to have worst-case complexity of $O(2^n)$, but it works much faster on most practical inputs [6].

One possible explanation of this phenomenon is that problem instances arising in practical applications tend to be *structured*, and smart algorithms can exploit this structure to yield significantly better performance than in the worst case. Indeed, it has been reported several times in the literature that structured problem instances arising in practical applications tend to be much easier than random problem instances of equal size [7, 8, 9]. For example, in the SAT Competitions[1], random SAT instances usually have $10^2 \ldots 10^4$ variables, whereas industrial benchmarks usually have $10^3 \ldots 10^6$ variables, and these two sets of problem instances pose comparable challenge to the participating solvers. Again, this can be attributed to our algorithms' ability to exploit the structure found in real-world problem instances, while in random problem instances, no such structure can be used.

1.2. Suggested approach

As can be seen from the above, the size of the problem instance is, in general, a poor indicator of its complexity. Of course, there is some correlation between size and complexity, but this correlation is quite weak, mostly because the size itself does not accomodate any structural information.

Hence the aim of this research is to propose a new metric that can be used instead of the size of the problem instance to predict algoritm runtime. This new metric should fulfil the following requirements:

[1] `http://www.satcompetition.org/`

- It should permit better prediction about the complexity of the given problem instance than size alone, by also incorporating structural information.
- Computation of the metric should be quick.

Therefore, we propose to carry out an analysis of the structure of the problem instance[2] with the aim of compressing it. Then, the *compressed size* of the problem instance can be used to predict its complexity. This metric has several advantages:

- It is a simple and intuitive metric, which can be computed with relative ease.
- It combines in a natural way the size of the problem instance and information about its structure. In particular, we expect structured problem instances to allow better compression than random problem instances.

We started this research with the hypothesis that compressed size is indeed a better basis for predicting algorithm runtime on the given problem instance than the sheer size of the problem instance. The primary objective of the paper is to investigate whether this hypothesis is true.

The secondary objective of the paper is to compare different compression methods from this point of view. That is, our aim is to develop a compression method with the property that the compressed size of a problem instance permits as good a prediction as possible about algorithmic complexity of the given problem instance.

In particular, we devised a genetic algorithm called GSA (Graph Structure Analyzer) to compress problem instances given in the form of a graph. We demonstrate with extensive empirical results that the compressed size returned by GSA is a better indication of algorithm runtime than the size of the problem instance.

1.3. Paper organization

In the following, we start with a review of related work in Section 2.

We use two particular application domains for presenting and evaluating our approach: graph coloring and Boolean satisfiability. Both are fundamental and widely-studied NP-hard combinatorial problems with many applications [1]. We use state-of-the-art exact algorithms for solving these problems, exhibiting exponential worst-case complexity, but far better typical-case complexity. These prelimiaries are described in the necessary detail in Section 3.

[2] It is important to note that we address here the *input structure*, as opposed to *problem structure*, as these two do not need to correlate in any way.

The GSA algorithm itself is presented in Section 4. In particular, it is described how a genetic algorithm can be used to evolve a compact description of the structure of a graph.

In Section 5, we compare our approach empirically with the traditional approach of predicting algorithm runtime based on the size of the problem instance. From the empirical results it is clear that the compressed size is indeed a better indication of algorithm runtime than size itself. Moreover, we compare GSA with other possible approaches for compressing graphs.

Section 6 discusses the results and puts them in the context of previous work. Finally, Section 7 concludes the paper.

2. Related work

The fact that an algorithm's runtime can vary drastically between problem instances of the same size has been known for decades. Systematic studies of this phenomenen and its possible reasons were first published in the beginning of the 1990s. In particular, Cheeseman *et al.* and Mitchell *et al.* investigated the complexity of the satisfiability problem and several graph theoretic problems (e.g., graph coloring) on random problem instances [10, 11]. They found that solver runtime depends largely on the ratio of the number of constraints to the number of variables. If the number of constraints is low (under-constrained case) or high (over-constrained case), then the investigated algorithms find the given problem instance easy. There is a relatively narrow range between the above two regions (critically constrained case) where complexity is much higher. This model was refined and extended later on, see e.g. [12, 13] and references therein.

More recently, researchers have begun to use such insight for predicting the complexity associated with a given problem instance. Leyton-Brown and co-workers showed that simple and easily computable statistical characteristics of the problem instance can be combined to predict algorithm runtime; moreover, the dependence of algorithm runtime on these parameters can be found automatically using machine learning techniques. These so-called empirical hardness models can be used for a variety of purposes, including the selection of the most appropriate algorithm from an algorithm portfolio, or the tuning of algorithm parameters [14, 15, 16].

In principle, these works have a goal similar to ours. However, concerning the details, the two approaches are quite different. The above works aim to obtain a prediction as accurate as possible, at the cost of substantial problem- and solver-specific tuning effort and using a large number of simple, potentially useful statistical metrics. In contrast, our aim is to investigate whether a single metric – the compressed size of the problem instance – provides a robust way to enhance prediction quality without any additional tuning.

In this respect, our research is more related to previous work on the minimum description length (MDL) principle. MDL is a fundamental principle, stating that the best explanation of a phenomenon or set of data is given by its simplest model, i.e. the model that explains the phenomenon and has the shortest description [17]. On the one hand, MDL is strongly related to the popular "Occam's razor" principle for judging scientific theories, stating that simpler theories should be preferred over unnecessarily complex theories. On the other hand, and more importantly for our research, MDL is also strongly related to Kolmogorov complexity. Kolmogorov complexity is a kind of compression method, using the shortest description of a Turing machine capable of generating a given string as the string's compressed size [18]. Unfortunately, computing the Kolmogorov complexity of a string is intractable in general, but approximations are possible.

Some researchers used MDL explicitly for structure identification. The Subdue system, developed by Cook and Holder, uses MDL to select the shortest description of a given graph, where new candidate descriptions are generated by identifying frequent patterns in the graph through a beam search [19, 20]. Subdue has been successfully applied to a number of real-world pattern mining tasks in various domains, such as chemical compound analysis, image analysis, and the analysis of electronic circuits.

In another application of MDL, Lutz used it to find the most compact hierarchical decomposition (i.e., nested module structure) of a software system [21]. Lutz uses a genetic algorithm to evolve candidate hierarchical decompositions; description length is used as the cost function.

The methods used in this paper are somewhat similar to those used in Subdue and in the algorithm of Lutz. However, there are also significant differences. In particular, while Subdue is searching for frequent patterns and the algorithm of Lutz is searching for hierarchical decompositions, our work focuses on general constructs in a graph that are typical of structured, real-world graphs (e.g., paths and cliques).

Finally, we mention the work of Hogg on Approximate Entropy [22]. His intent was to apply the knowledge about the complexity profile of "problem ensembles" (e.g., random versus structured) to specific problem instances, by identifying automatically to which ensemble the problem instance belongs. For this purpose, Hogg proposed an information-theoretic approach, resulting in the metric of Approximate Entropy to quantify to which extent the given problem instance is random. In terms of graph coloring, Approximate Entropy is based on the frequency of different subgraphs. In order to keep it tractable, Hogg proposed to only look at subgraphs with up to 3 vertices. In contrast, our algorithm investigates also much bigger subgraphs, but only those that are highly structured.

3. Preliminaries

Although our approach can be applied generally to combinatorial algorithms, we had to choose specific application domains for evaluating it. We chose graph coloring and Boolean satisfiability, both of them much-investigated combinatorial problems with many different applications.

3.1. Graph coloring

In the decision version of graph coloring, we are given a graph G and a positive integer k, and the task is to decide whether each vertex of the graph can be assigned one of the k colors such that the color of adjacent vertices are different. The minimum value of k, such that G admits a coloring with k colors is its *chromatic number*. In the optimization version of the graph coloring problem, we are given a graph G, and the task is to determine its chromatic number. In this paper, we focus on the optimization version of the problem.

For our investigations, we used an exact graph coloring algorithm that uses backtrack search to traverse the search space and several intelligent heuristics (for vertex selection, color selection, symmetry breaking, initial assignment etc.) to make it as efficient as possible [23]. As with other smart exact algorithms for NP-hard problems, this algorithm also exhibits large variances in its runtime.

We used two families of problem instances:

1. Random graphs. We use the $G_{n,p}$ model ($n \in \mathbb{Z}^+$ and $0 \leq p \leq 1$ are given parameters): the graph has n vertices, and each pair of vertices is connected with probability p, independently from the other pairs.
2. Real-world benchmarks, stemming from instances of the Register Allocation Problem (RAP). Register allocation is one of the most widespread applications of graph coloring [24]. In particular, we used the graphs made publicly available by Andrew W. Appel and Lal George at `http://www.cs.princeton.edu/~appel/graphdata/`. These are register interference graphs generated by Standard ML of New Jersey version 1.09, compiling itself.

The number of vertices in the used RAP benchmarks were between 34 and 337. When generating the random graphs, we wanted to make them comparable to the RAP graphs. Therefore, we created for each RAP graph one random graph with the same number of vertices and approximately the same number of edges. For this purpose, we chose the value of p such that the expected number of edges will be the same as the number of edges of the RAP graph.

We will often refer to the adjacency matrix of a graph. For a graph with vertices $v_1, v_2, \ldots, v_n$, this is an $n \times n$-matrix, in which the jth element of the ith row is 1 if there is an edge between v_i and v_j, otherwise it is 0.

3.2. Boolean satisfiability

We are given a logical formula of Boolean variables in conjunctive normal form. That is, the formula is the conjunction of clauses, each clause is a disjunction of literals, and each literal is either a variable or the negation of a variable. A *satisfying assignment* is an assignment of truth values to the variables such that the formula evaluates to TRUE. Here, we deal with the counting version of the problem, denoted as #SAT, in which the goal is to compute the number of satisfying assignments.

For solving #SAT problem instances, we used the *sharpSAT* solver, a highly efficient algorithm implementation, available from `http://marcthurley.heliohost.org/sharpSAT/index.html` and described in [25].

The structured benchmark problems are taken from `http://web.cecs.pdx.edu/~whung/BLRP/`. These are *board-level multi-terminal net routing* problems, arising in FPGA-based hardware design [26]. Actually, these problem instances were too hard for the solver to finish in acceptable time. Therefore, we cut the instances in two or three parts. The resulting smaller problem instances had 75-285 variables. As we experienced, the half of a structured benchmark is still quite structured (although a bit less), but cutting into many small pieces would let the structure diminish.

For each structured formula, we created a random formula with the same number of variables and clauses. Since the structured formulae were all satisfiable, we made sure that the random formulae are also satisfiable. For this purpose, we generated a random assignment and only allowed clauses in the random formula that are satisfied by this assignment.

In order to apply the GSA algorithm, we had to transform the Boolean formulae to graphs. We used the following construction: each variable is represented by a vertex, and two vertices are connected by an edge if and only if there is a clause containing both variables.

3.3. Evaluation framework

As programming and testing environment, we used BCAT, the Budapest Complexity Analysis Toolkit [27]. BCAT was designed specifically to facilitate the development and testing of algorithms. It offers the possibility to set up different problem classes, analyzers of these problems, converters that transform the instances of a certain problem to the instances of another problem, and algorithms that solve the problems. Afterwards, one can give the settings of the program by means of a configuration file:

- what problem instances should be loaded or generated,
- into which problems should they be converted with which converters,
- with what analyzers should the problems be analyzed,
- what algorithms should be applied to solve the problem instances.

BCAT delivers the output data in structured files that can be easily processed e.g. in a spreadsheet application.

In the terminology of BCAT, the backtrack search for coloring graphs and the sharpSAT solver are *algorithms*, whereas the algorithm described in this paper for structure analysis and compression of a graph is an *analyzer*.

4. Graph Structure Analyzer

4.1. Decomposition of a graph

We now present our GSA (Graph Structure Analyzer) algorithm that aims at finding a compact description of a graph by identifying structured parts of the graph. In the following, these structured parts will be called *object*s. We tried to define object types that (i) are unlikely to be created randomly, but are typical of structured graphs and (ii) may have significant impact on the runtime of combinatorial algorithms. Therefore, we defined the following object types ($G = (V, E)$ is the input graph):

- *Pseudo-complete graph*: a subgraph that is almost a complete graph, with only a small number of edges missing. That is, a pseudo-complete graph object is given by the pair (G', X), where $G' = (V', E')$ is an induced subgraph of G and $(V', E' \cup X)$ is a complete graph. Here, X is the set of missing edges and thus, $|X|$ is assumed to be small.

- *Pseudo-empty graph*: a sub-graph that has only a small number of edges. That is, a pseudo-empty graph object is an induced subgraph $G' = (V', E')$, where $|E'|$ is assumed to be small.

- *Path*: a – not necessarily induced – subgraph of G that is a path.

- *Chaos*: an induced subgraph that is too dense to be a pseudo-empty graph but too sparse for a pseudo-complete graph. It is defined by its adjacency matrix.

- *Further vertex*: can be any vertex of G.

It should be noted that the first three object types can leverage the structure of the graph; the last two object types only serve the purpose of completeness, so that the whole graph can be decomposed into the set of available object types. Moreover, this generic list of object types can be extended with further, possibly problem-specific object types if needed.

Beside the objects, we also defined two types of *connection*s:

- *Further edge*: an edge that is not contained in an object, but connects two objects.

- *Overlap*: we allow the objects to overlap, i.e. a vertex can be contained in more than one object. For this purpose, it must be specified that vertex v_1 in object o_1 is the same as vertex v_2 in object o_2.

Using the above 5 object types and 2 connection types, a structural description of a graph can be constructed. From this structural description, the graph can be reconstructed unambiguously and without loss of information. On the other hand, there are several possibilities for decomposing a graph; as we will see later, it is the main task of GSA to find the best such decomposition.

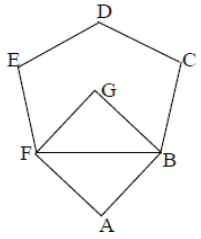


Figure 1: Example graph

In the following, we illustrate the object and connection types, and how they can be used to create a description of a graph, on an example. We will use the graph in Figure 1.

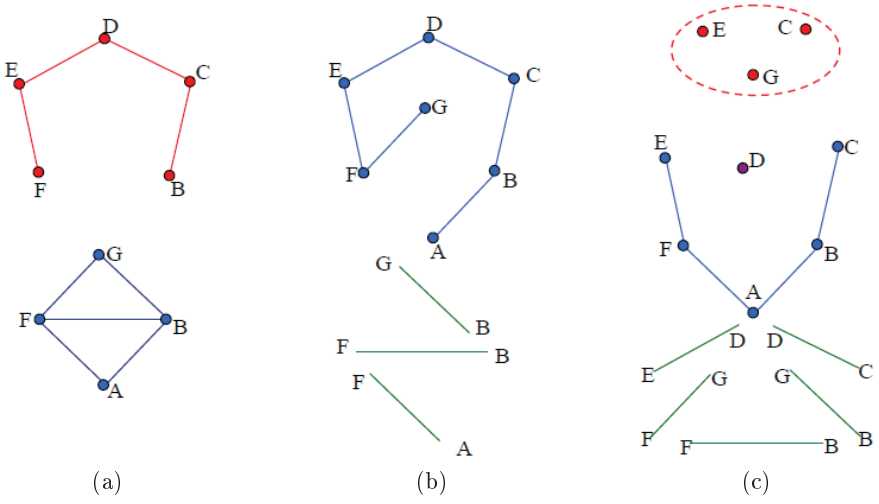


Figure 2: Possibilities to decompose the graph of Figure 1

There are many possibilities for decomposing this graph using the available object and connection types, e.g.:

- A path of length 4, made up of vertices B, C, D, E, F; a pseudo-complete graph with 4 vertices (A, B, F, G) and 1 missing edge (AG); and 2 overlaps (B and F). See Figure 2a.
- A path of length 6, made up of vertices A, B, C, D, E, F, G; and 3 further edges (BG, BF, AF). See Figure 2b.
- A path of length 4, made up of vertices E, F, A, B, C; a pseudo-empty graph with 3 vertices (G, C, E) and 0 edge; a further vertex (D); 5 further edges (FB, FG, GB, ED, EC); and 2 overlaps (E and C). See Figure 2c.

It is apparent from this example that, while all three presented decompositions are valid, they are not equally descriptive of the structure that is intuitively visible to humans from Figure 1. In particular, the third decomposition is clearly more verbose than the first two and also less useful in terms of structural description. This is in line with the MDL principle: the shortest description yields the most insight into the structure.

4.2. The case for a genetic algorithm

In accordance with the MDL principle, the aim of GSA is to find a decomposition of the graph with minimum description length. Such decompositions are hard to find optimally, as shown by the NP-hardness results on some related problems [28, 29, 30]. Since we need a quick algorithm that delivers a good, but not necessarily optimal result, we settled for using a heuristic. The use of a genetic algorithm was motivated by the genetic algorithm of Lutz to attack a related problem [21]. Genetic algorithms have been used extensively in the last decades to solve a number of hard optimization problems; they have proved to offer good characteristics in terms of robustness and noise toleration, and are quite efficient due to their implicit parallelism [31], making them a lucrative solution for our purposes.

Hence, GSA is a genetic algorithm, in which the individuals are decompositions of the input graph. The cost of an individual is its description length, and the aim is to minimize cost. Through evolution, GSA is capable of finding an individual of a relatively short description.

On the other hand, the adaptation of the concept of genetic algorithms to our particular task was quite challenging. The application of a genetic algorithm is straight-forward if individuals can be represented in a natural way as a linear string; genetic operators like mutation and recombination can then be applied in the standard way. In our case, individuals represent decompositions of graphs into different objects using different connections – a highly non-linear problem, posing several challenges in the application of the genetic algorithm paradigm.

In the following, we describe how we tackled this challenge. We describe in detail the cost function, the creation of the initial population, and the genetic operators.

4.3. The cost function

In order to define the cost function, a convention is needed to describe a decomposition of the input graph G by means of an unambiguously decodable bit string. Our coding scheme is based on the idea that, for each object or connection type, first the number of occurences is given, followed by the sequence of the descriptions of those objects or connections. This guarantees that the whole description is indeed decodable.

Table 1: Coding scheme

	Data	Nr. of bits
1	Nr. of vertices of the input graph (n)	constant
2	Nr. of pseudo-complete subgraphs	$\lceil \log_2 n \rceil$
3	*For each pseudo-complete subgraph:*	
4	Nr. of vertices in the pseudo-complete subgraph (v)	$\lceil \log_2 n \rceil$
5	Nr. of missing edges	$\lceil \log_2 \binom{v}{2} \rceil$
6	*For each missing edge:*	
7	End vertices of the missing edge	$2 \lceil \log_2 v \rceil$
8	Nr. of pseudo-empty subgraphs	$\lceil \log_2 n \rceil$
9	*For each pseudo-empty subgraph:*	
10	Nr. of vertices in the pseudo-empty subgraph (v)	$\lceil \log_2 n \rceil$
11	Nr. of edges	$\lceil \log_2 \binom{v}{2} \rceil$
12	*For each edge:*	
13	End vertices of the edge	$2 \lceil \log_2 v \rceil$
14	Nr. of paths	$\lceil \log_2 n \rceil$
15	*For each path:*	
16	Nr. of vertices in the path	$\lceil \log_2 n \rceil$
17	Nr. of chaoses	$\lceil \log_2 n \rceil$
18	*For each chaos:*	
19	Nr. of vertices in the chaos (v)	$\lceil \log_2 n \rceil$
20	Adjacency matrix of the chaos	$\frac{v(v-1)}{2}$
21	Nr. of further vertices	$\lceil \log_2 n \rceil$
22	Nr. of further edges	$\lceil \log_2 \binom{n}{2} \rceil$
23	*For each further edge:*	
24	*For each end vertex of the edge:*	
25	Identifier of the object	$\lceil \log_2 o \rceil$
26	Identifier of the vertex within the object	$\lceil \log_2 v \rceil$
27	Nr. of vertices present in more than one object (overlaps)	$\lceil \log_2 n \rceil$
28	*For each overlap:*	
29	Nr. of objects that the given vertex belongs to	$\lceil \log_2 o \rceil$
30	*For each such object:*	
31	Identifier of the object	$\lceil \log_2 o \rceil$
32	Identifier of the vertex within the object	$\lceil \log_2 v \rceil$

The used coding scheme is shown in Table 1. Some remarks to the table:

- Row 1: we assume that the number of vertices is not more than a predefined maximum, say, 2^{16}.
- Rows 2, 8, 14, 17: we assume that each object contains at least one vertex not contained in the others, hence the number of objects of each type is at most n.
- Row 20: As the graph is undirected, only one half of its adjacency matrix needs to be stored.
- Rows 25, 29, 31: o denotes the total number of objects.
- Rows 25-26 and 31-32: vertices are identified relative to the object that they reside in, i.e. using the identifier of the object plus the identifier of the vertex within the object.
- Rows 26, 32: v denotes the number of vertices in the object that the given vertex resides in.

Using this coding scheme, the description length of a decomposition can be calculated easily. Thus, the cost of an individual is readily available.

As an example, we revisit the decompositions in Figure 2. Their encoding is as follows (for the sake of readability, we present the encoding in decimal form; when calculating the fitness, the length of the binary representation is considered):

- Figure 2a: 7 1 4 1 0 3 0 1 4 0 0 0 2 2 0 1 1 0 2 0 2 1 4
- Figure 2b: 7 0 0 1 7 0 0 3 0 1 0 6 0 1 0 5 0 0 0 5 0
- Figure 2c: 7 0 1 3 0 1 5 0 1 5 0 1 2 0 0 0 2 0 0 2 1 1 0 2 1 3 1 1 1 3 2 2 0 0 1 4 2 0 1 1 0

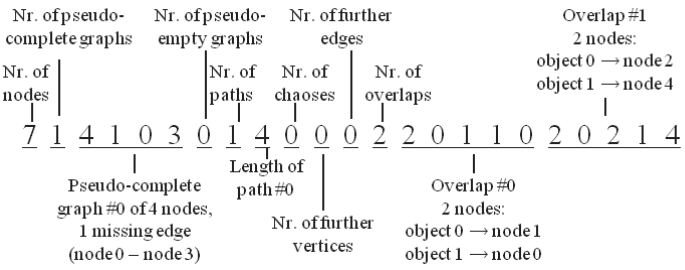


Figure 3: Example for the encoding of a graph decomposition

Figure 3 explains the encoding of the decomposition of Figure 2a, thus exemplifying the logic of the encoding. It should be noted that – in line with intuition – the third encoding is significantly longer than the first two.

4.4. Initial population

The initial population should contain some valid decompositions that are easy to create. There are two trivial decompositions: each vertex is an object (a "further vertex") on its own, or the whole graph is a single object (a chaos). However, it is beneficial if the initial population already contains quite good individuals, because thus we can reduce the number of generations necessary before reaching the final result.

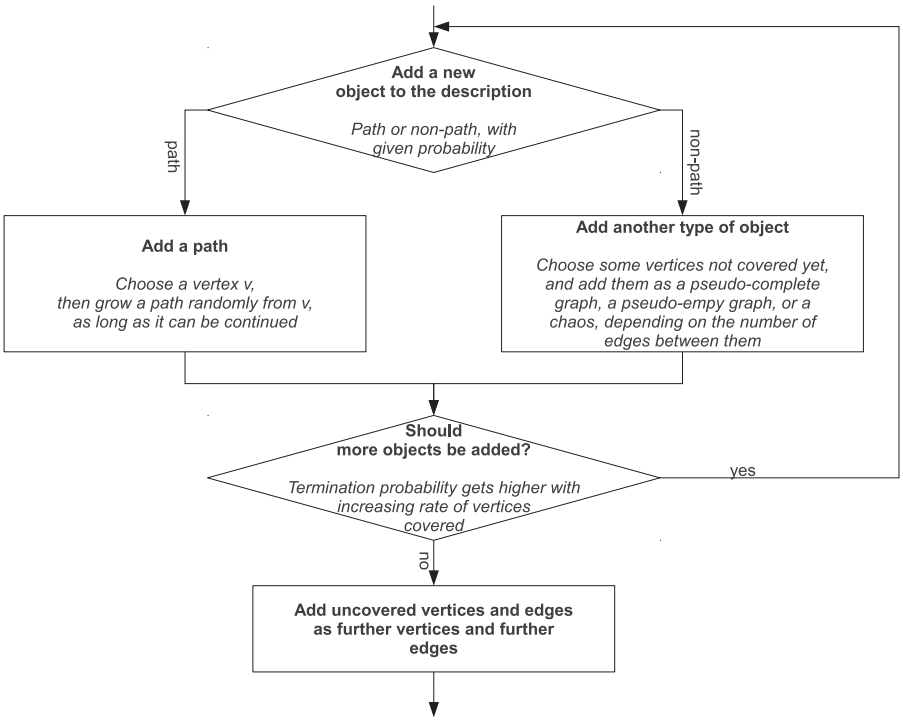


Figure 4: Heuristic for generating an individual in the initial population

Therefore, we devised a heuristric that greedily grows other objects as well. As a simple rule for creating pseudo-complete and pseudo-empty graphs, it considers some vertices not yet covered by the already created objects, and turns them – depending on the number of edges between them – into a pseudo-complete subgraph, a pseudo-empty subgraph, or a chaos (see Figure 4).

4.5. Mutation

In GSA, we tried to implement mutation in an intelligent way, so that the new individual resulting from mutation will probably have a low cost value.

GSA employs seven types of mutation; in each type, there are multiple specific mutation operations. Each mutation operation has a given probability; these probabilities are important parameters that may be tuned to

improve the algorithm's performance. The seven mutation types are as follows:

1. Creation of a new object.
2. Deletion of an object.
3. Increasing the size of an object.
4. Decreasing the size of an object.
5. Converting an object to another object type.
6. Merging two objects into a single object.
7. Splitting an object into two objects.

The specific mutation operations of each type are described in the Appendix.

4.6. Recombination

In GSA, recombination creates one offspring from two parents. It works by taking a subset of the union of objects present in the parents, and extending this with further vertices and edges as necessary to obtain a decomposition covering all vertices of the input graph.

The key question is, which objects from the union of the objects of the parents to select. Even if the parents themselves are quite fit, i.e. they represent compact descriptions of the input graph, taking random objects from their union may lead to a highly redundant and thus not very fit offspring. That is, we should take objects that cover as much as possible from the input graph, while minimizing the redundancy between them.

Therefore, we use the following heuristic. We take objects one by one randomly from the parents, but the selection probabilities are adjusted in each step based on the above idea. The selection probability of an object is monotonously increasing in the number of its vertices that are not covered by the objects already selected, and monotonously decreasing in the number of its vertices that are already covered by previously selected objects.

4.7. Creating the next generation

In each iteration, the next generation is created using mutation, recombination, and copying of some individuals from the old population. For all these operations, individuals are selected randomly, but the selection probability of an individual is monotonously decreasing in its cost.

5. Empirical results

5.1. Parameter tuning

Our algorithm exhibits a large number of parameters. This is mainly due to the high number of available mutation operations: there are almost 30 mutation operations (in 7 groups), and the probability of each mutation

operation is a parameter. Also, the construction of the new generation from the old one is governed by several parameters which specify how many individuals are created by mutation, recombination, or simple copying from the top x percent of the old population. Therefore, finding the best parameter configuration was a real challenge. An exhaustive search of all possible parameter configurations was obviously not possible, not even if only a handful of possible values had been tried for each parameter.

We created a small training set, consisting of 8 structured graphs (with the number of vertices between 60 and 105) and 2 similar random graphs for each structured graph, i.e. 24 graphs in total. We had to keep the size of the training set small in order to keep the total time needed for parameter tuning in a reasonable range. (Of course, the graphs in the training set were only used for parameter tuning, and were discarded afterwards.)

In the first round of experiments, we dealt with the relative probabilities of the mutation operation within one mutation group, independently from all other parameters. After trying 10-15 different values for all of these parameters, they were fixed one by one to the value that led to the best compression ratio on the training set. These parameter values were kept in all later experiments as well.

In the next stage, we optimized the relative probabilities of the 7 mutation groups. In addition, we also considered as an 8^{th} parameter the number of elementary mutation steps carried out on an individual that was selected for mutation. In order to find a good combination for these 8 parameters, we employed a greedy hill-climbing heuristic in the parameter space. That is, we started with a random parameter configuration, and in each step, we considered the slight increase or decrease of a parameter; such a change was only carried out if it improved the compression ratio achieved on the training set. The heuristic stopped when a local optimum was reached, i.e. if no parameter could be changed without worthening the compression ratio. This procedure was carried out multiple times, restarting from a new random parameter configuration.

The parameters that govern how the new population is created from the old one were optimized by means of a second hill-climbing. Interestingly, it turned out that mutation is more useful than recombination and that copying of individuals without alteration is not needed at all. In the final configuration, 60% of the new population is filled by mutation, 40% is filled by recombination.

Finally, we experimented with different population sizes and different stopping criteria. We found that for bigger graphs we need either more individuals or more generations to reach a reasonable solution than for smaller graphs. However, this has obviously an impact on the running time of GSA. In the end, we settled for a configuration in which the size of the population is constant, but the number of generations is a linear function of the size of the input graph.

5.2. Experience with the GSA algorithm

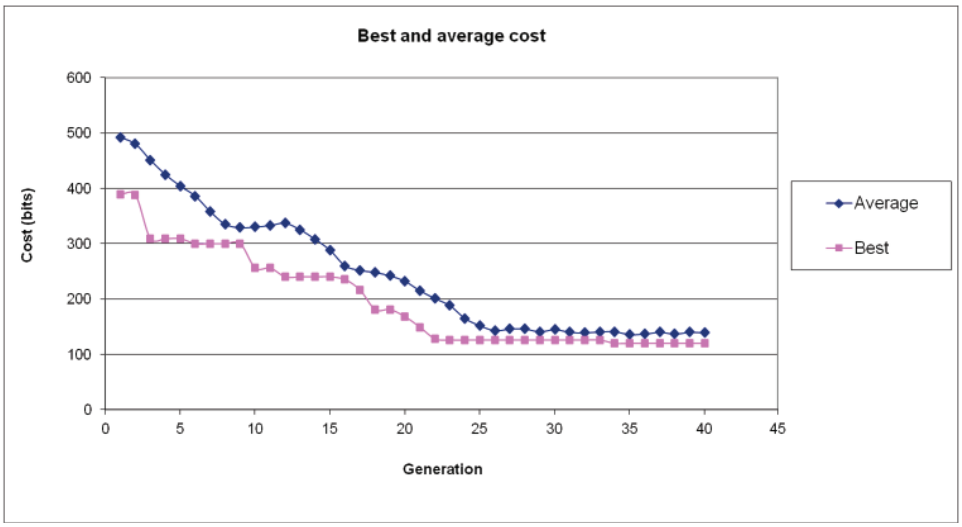


Figure 5: A typical run of GSA

Figure 5 shows the evolution of the cost of the best individual found so far as well as the average cost in the population, in a typical run of the algorithm. As expected, the best cost is monotonously decreasing; small decreases correspond to small mutations, bigger decreases correspond to either exceptionally successful mutations or recombinations. The average cost follows the best cost and also improves significantly over time. At the end, the average cost oscillates slightly above the best cost; it is not monotonously decreasing, as in this stage mutations will usually produce worse individuals than the already quite good individuals of the population, and this can worsen the average cost from time to time.

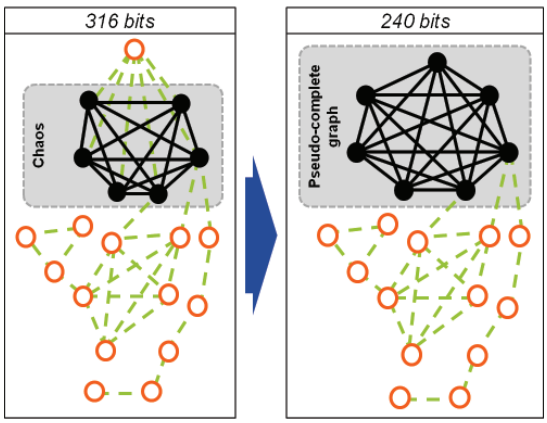


Figure 6: A mutation, taking place in generation 10 of the run shown in Figure 5

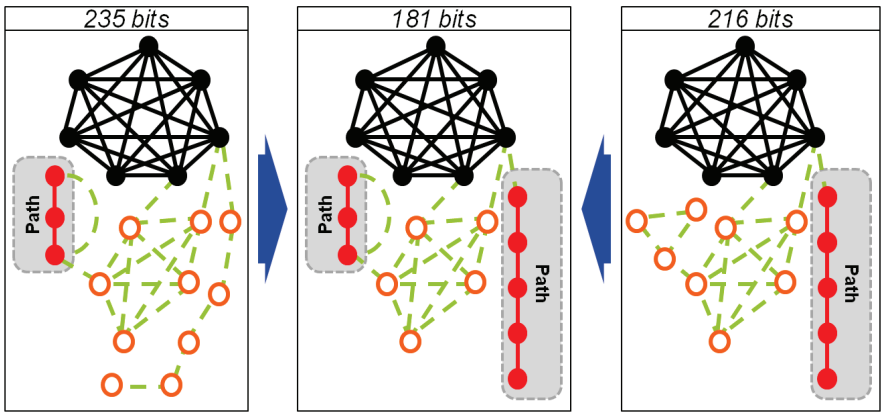


Figure 7: A recombination, taking place in generation 18

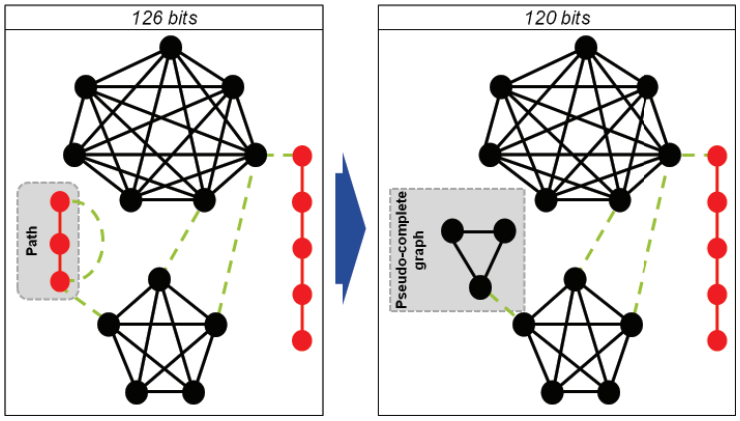


Figure 8: A mutation, taking place in generation 34

To demonstrate the operation of the algorithm, we show some examples in Figures 6-8, corresponding to some steps of the algorithm's run depicted in Figure 5. In these figures, filled circles and solid edges correspond to the vertices and edges, respectively, of the found objects; empty circles and dashed lines correspond to further vertices and further edges, respectively.

As can be seen in Figure 6, the big decrease in cost in generation 10 is due to a mutation, in which a chaos and a further vertex are turned into a pseudo-complete graph. Figure 7 shows a lucky recombination of two individuals. In both parents, a part of the graph is recognized as a path, resulting in two paths in the offspring. Figure 8 shows a small mutation, in which a path and a further edge are turned into a pseudo-complete graph. As these examples demonstrate, the implemented operations and the fitness function are indeed capable of evolving better and better decompositions of

a graph.

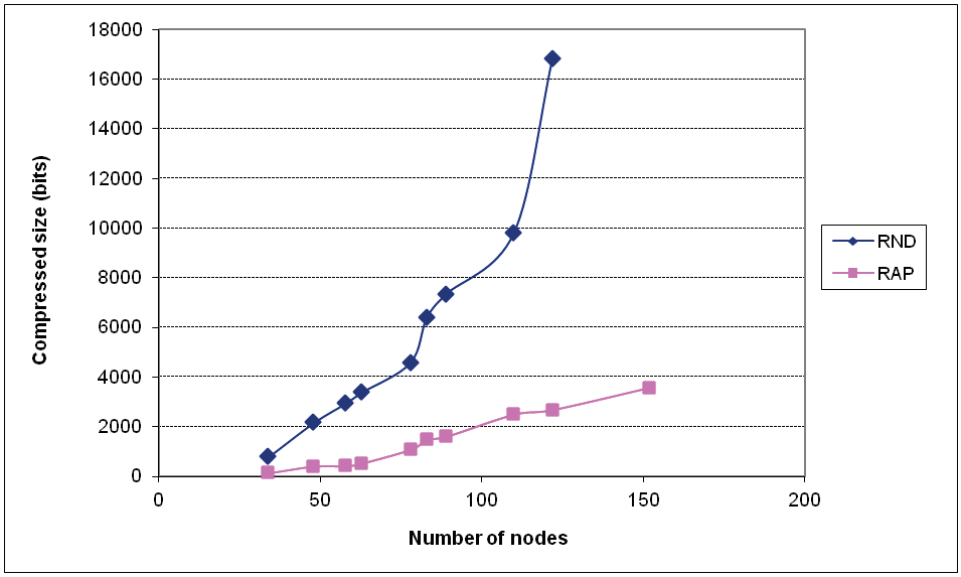


Figure 9: Compression by GSA, as a function of the graph's size

Next, we verified that GSA can indeed compress structured graphs more than random graphs. As shown in Figure 9, this is indeed the case: the compressed size of random graphs (the RND series in the Figure) is clearly higher than the compressed size of the real-world graphs corresponding to instances of the Register Allocation Problem (the RAP series). Moreover, the difference between the two series is clearly increasing with the graph's size.

5.3. Running time

Concerning the running time of the GSA algorithm, our only expectation was that it should be significantly faster than typical exact solvers, e.g. the exact graph coloring algorithm mentioned in Section 3, at least for big inputs where this really matters. This is fulfilled, as the running time of GSA grows only polynomially with the size of the problem instance, whereas the exact solvers' runtime typically grows exponentially.

Specifically for the graph coloring algorithm, we observed the following. For small inputs, GSA is not faster than the graph coloring algorithm itself. This is not a problem, as predicting the runtime of the solver becomes important only for bigger inputs where the solver is slow. Indeed, for big problem instances, the graph algorithm's runtime is multiple orders of magnitude higher than that of GSA. For example, for random graphs with 122 vertices, the runtime of the graph coloring algorithm is approximately 40 times higher than that of GSA.

5.4. Comparing high-level hardness prediction

In the next round of experiments, our aim was to determine to what extent the GSA algorithm fulfils our original goals, and how it compares to other methods for graph compression. Specifically, we compared it to the following methods:

- *ZIP*: the adjacency matrix of the graph is written to a file, which is then compressed using a standard zip archiver.
- *PAQ*: similar to ZIP, but using the more advanced PAQ9 compression package.
- *Subdue*: the graph is analyzed using Subdue [19, 20]; the found frequent patterns of the graph are replaced by a single vertex with a reference to the extracted substructure, thereby compressing the graph's description.
- *Subdue-recursive*: Subdue also supports a recursive mode, in which the same procedure is called for the compressed graph etc., leading to multiple iterations of the algorithm.

It is important to note that the objective of this research was to be able to predict solver performance based on compressed problem size; hence, the aim of the comparison is to determine which algorithm is best suited for this purpose.

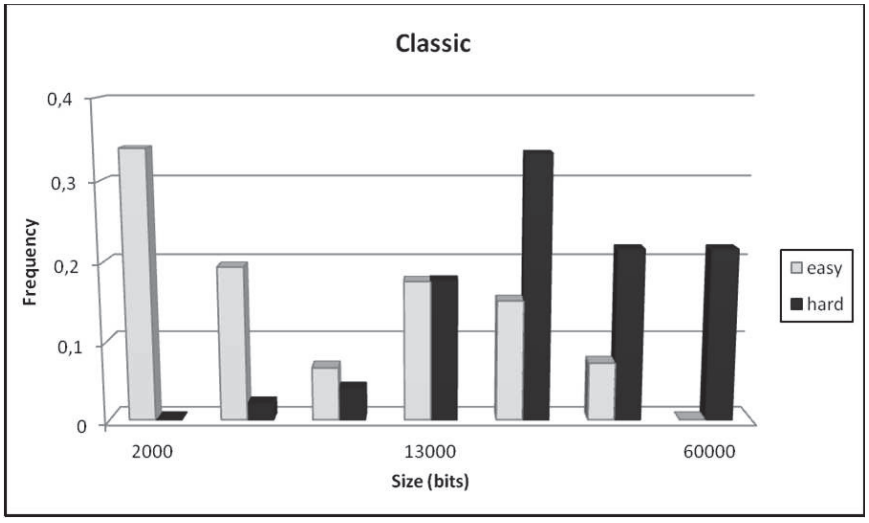


Figure 10: Histogram of easy and hard problem instances, based on their uncompressed size

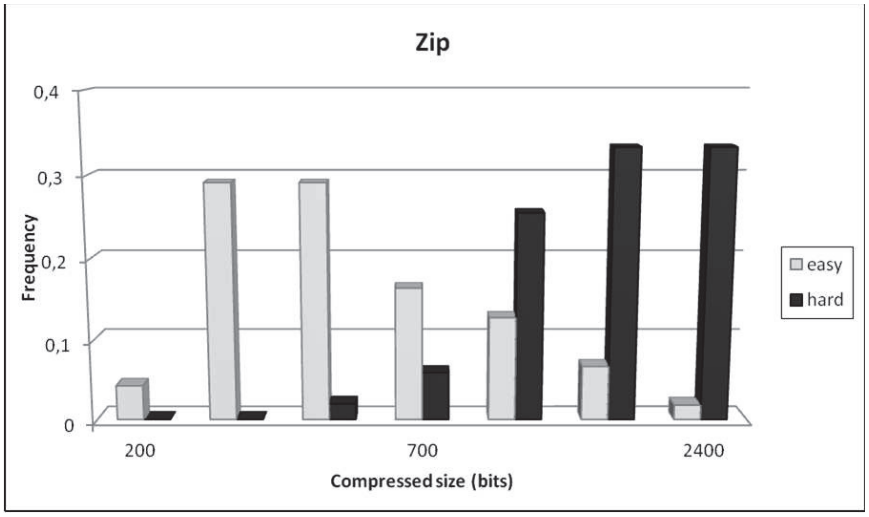


Figure 11: Histogram of easy and hard problem instances, based on their compressed size after compression using ZIP

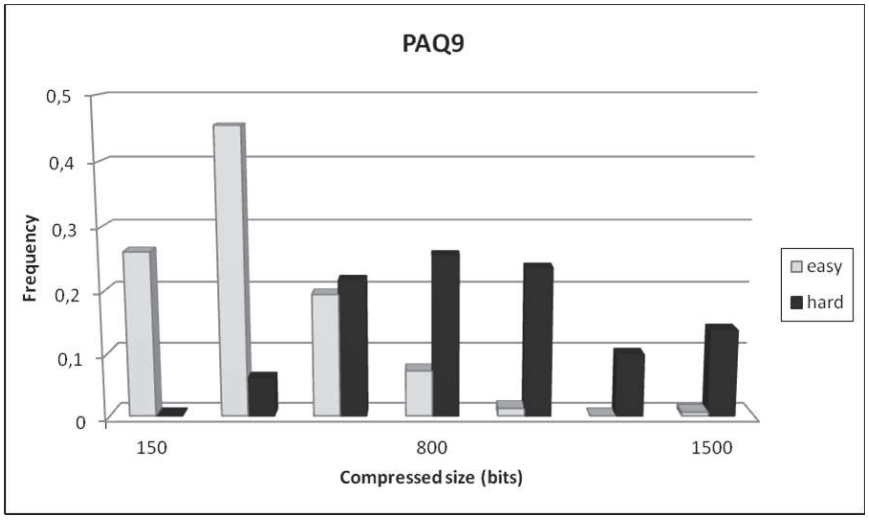


Figure 12: Histogram of easy and hard problem instances, based on their compressed size after compression using PAQ9

Our experiments revealed an interesting dichotomy: there were problem instances that the solver could solve within 1-2 minutes, and there were others that the solver did not solve within 24 hours. So, on a high level, we can differentiate between easy and hard problem instances. Hence, we tested to what extent the different compression methods can be used as basis for classifying problem instances as easy or hard. The results are shown in Figures

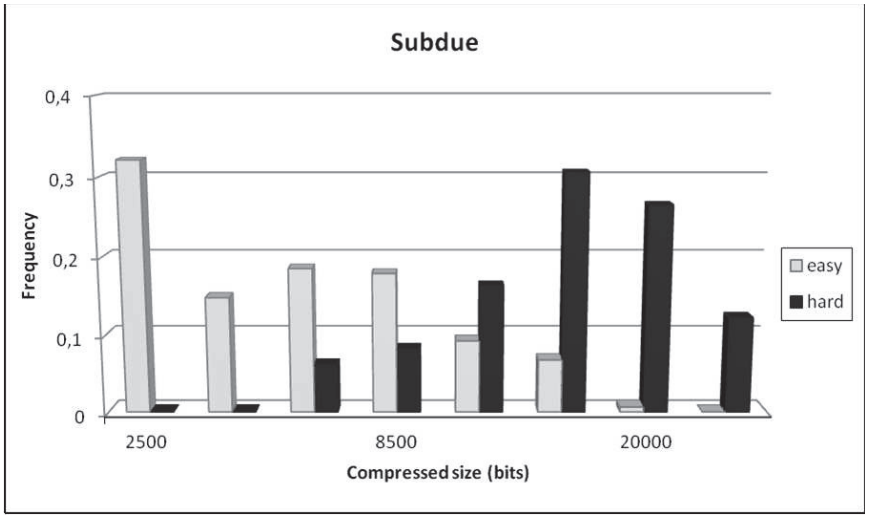


Figure 13: Histogram of easy and hard problem instances, based on their compressed size after compression using Subdue

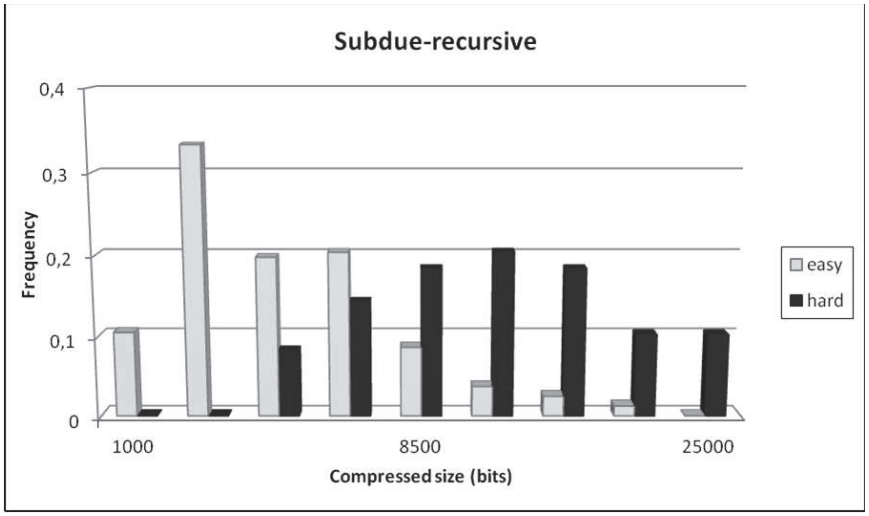


Figure 14: Histogram of easy and hard problem instances, based on their compressed size after compression using Subdue-recursive

10-15. In all of these figures, hard instances tend to have higher (compressed or, in the case of Figure 10, uncompressed) size than easy instances. That is, instances can be categorized based on size. However, there is an overlap between the two curves, leading to an error in the categorization; since the overlap is different in each figure, so is the error. Based on the figures, it can be seen that the overlap – and hence the predicion error – is largest with the

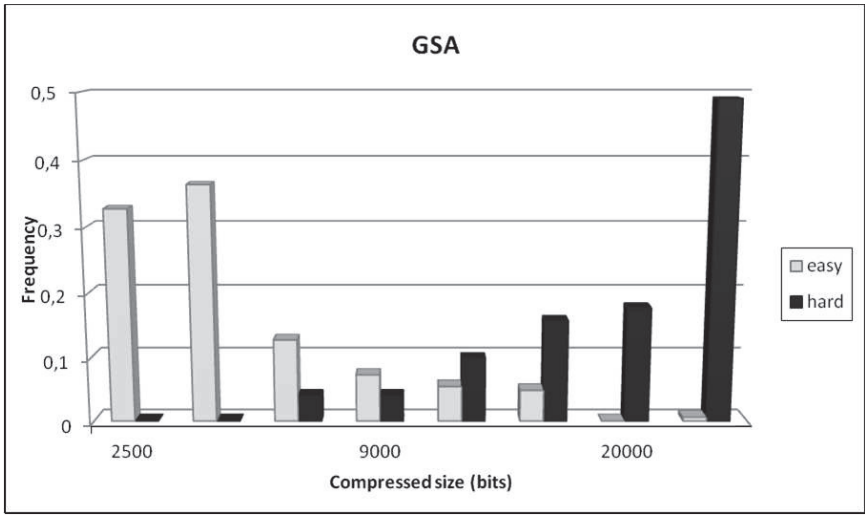


Figure 15: Histogram of easy and hard problem instances, based on their compressed size after compression using GSA

"classic" method, when using the uncompressed size of the problem instance. All investigated compression methods yield smaller overlap. The overlap is smallest in the case of GSA.

Table 2: Error of optimal categorization based on (uncompressed or compressed) size. Smaller numbers are better.

Compression method	Categorization error
No compression	20.7%
ZIP	12.2%
PAQ	12.6%
Subdue	13.1%
Subdue-recursive	18.0%
GSA	9.7%

In order to quantify these findings, we assume the best possible categorization that predicts an instance to be easy if its size is under a threshold x_0 and hard if its size is above x_0. The optimal threshold can be found by minimizing the error $|\{\text{number of easy instances above } x_0\}| + |\{\text{number of hard instances below } x_0\}|$. The resulting error ratios are shown in Table 2. These numbers show that GSA can achieve a categorization that is a factor 2 better than the classic method of relying on the uncompressed size of the problem instance.

5.5. Comparing numerical runtime prediction

Since the problem instances labelled "hard" in the above categorization cannot be solved within reasonable time, we use the "easy" instances to numerically assess the quality of solver runtime prediction. It should be noted that the "easy" instances are by no means trivial[3]; the word "easy" just means that they can be solved within reasonable time. Some of these instances can be solved within milliseconds, others take seconds or even some minutes to solve, so there are still multiple orders of magnitude between them. Our aim is to investigate to what extent uncompressed or compressed size can be used as the basis for predicting solver runtime.

As mentioned earlier, the weakness of the conventional approach of using the input size as the basis for predicting solver runtime is that inputs of the same size can lead to solver runtimes of great variance. Specifically, there tends to be a gap between the complexity on random and structured inputs. This motivates the basic idea of our comparison: a compression method is useful for our purposes if the solver's runtime on random versus structured inputs of the same *compressed* size is closer to each other than the solver's runtime on random versus structured inputs of the same *uncompressed* size. The smaller the gap between the solver's runtime on random versus structured inputs of the same compressed size, the better the compression method.

Based on this insight, we can visually demonstrate the usefulness of a compression method by depicting two pairs of functions:

- The solver's runtime on random respectively structured inputs as a function of the input's *uncompressed* size. For the graph coloring case study, these functions will be called "RND" and "RAP", respectively.
- The solver's runtime on random respectively structured inputs as a function of the input's *compressed* size. These functions will be called "RND cmpr" and "RAP cmpr", respectively.

If the two latter curves are closer to each other than the two former curves, then the compression method can be considered useful.

Beside comparing the pairs of functions visually on the charts, it is also useful to quantify the differences. For this purpose, we used two metrics. One is the *correlation* between the solver's runtime and the (compressed or uncompressed) size of the problem instance. For this, we combine all of the investigated problem instances – i.e., both random and structured – in a single dataset. By measuring the solver's runtime and the compressed size of each problem instance, we obtain two data series, and we can compute their

[3]To illustrate this: the used graph coloring algorithm investigates up to $4 \cdot 10^6$ states of the search space when solving these problem instances.

correlation coefficient. Obviously, the higher the correlation coefficient, the better the used compression method. We also compute the corresponding p-value in order to assess the statistical significance of the correlation coefficient (i.e., to quantify how likely/unlikely it is to obtain the given correlation coefficient purely by chance).

The other metric that we used is related to what the diagrams show: the distances between pairs of functions. First, we define the *average distance* between two discrete functions f and g as follows:

$$d(f,g) := \frac{1}{N}\sum_{i=1}^{N}\left(f(x_i)-g(x_i)\right)^2,$$

where $x_1,\ldots,x_N$ are the points in which both f and g are defined. In our case, this is a bit tricky: although the domain of the two functions that we want to compare contain a common interval, they are defined in different discrete points only. In order to overcome this difficulty, we interpolate f linearly between its actual measurement points, so that its value is defined in all measurement points corresponding to g. This way, we can use the measurement points of g that lie within the domain of f for calculating $d(f,g)$ using the above formula.

Using this formalism for the average distance between two functions, we can define the *prediction ratio* of a compression method as follows:

$$R := \frac{d(RAP,RND)}{d(RAP\,cmpr,RND\,cmpr)}.$$

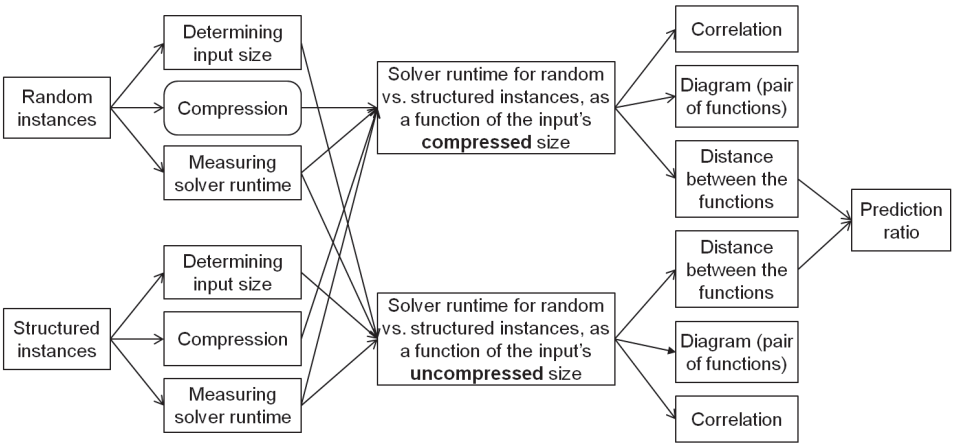


Figure 16: Comparison methodology

If the prediction ratio is higher than 1, then the compressed size has more predictive power than the uncompressed size. More generally, the higher the

prediction ratio, the more appropriate the compression method for predicting solver runtime.

Figure 16 summarizes our methodology for evaluating the appropriateness of a compression method, using the visualization of the two pairs of functions, the correlation coefficients, and the prediction ratio.

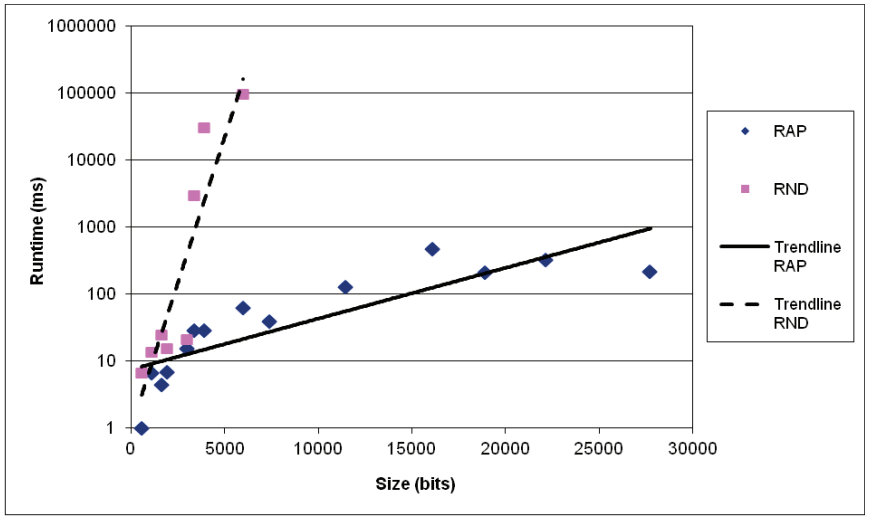


Figure 17: Results achieved on graph coloring using the classic method, i.e. no compression

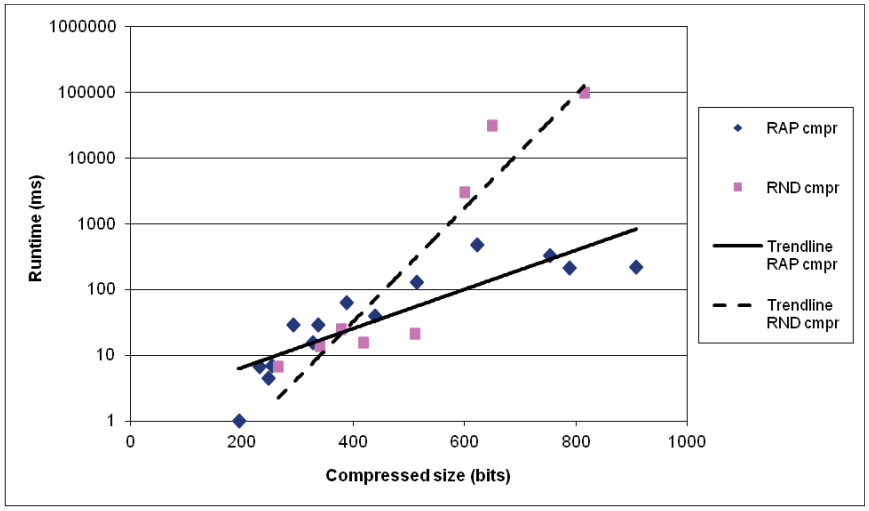


Figure 18: Results achieved on graph coloring using the ZIP compression method

We compared GSA with ZIP, PAQ, Subdue and Subdue-recursive, in connection with the exact graph coloring algorithm implemented in BCAT,

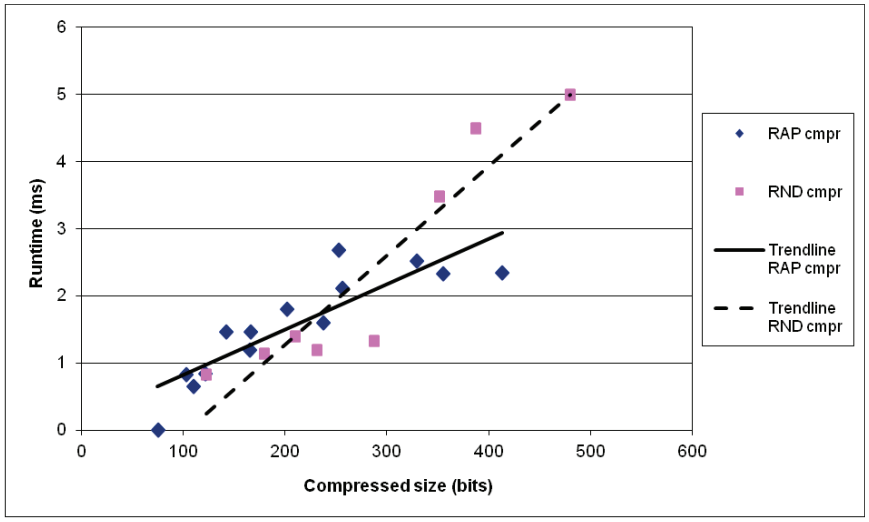


Figure 19: Results achieved on graph coloring using the PAQ9 compression method

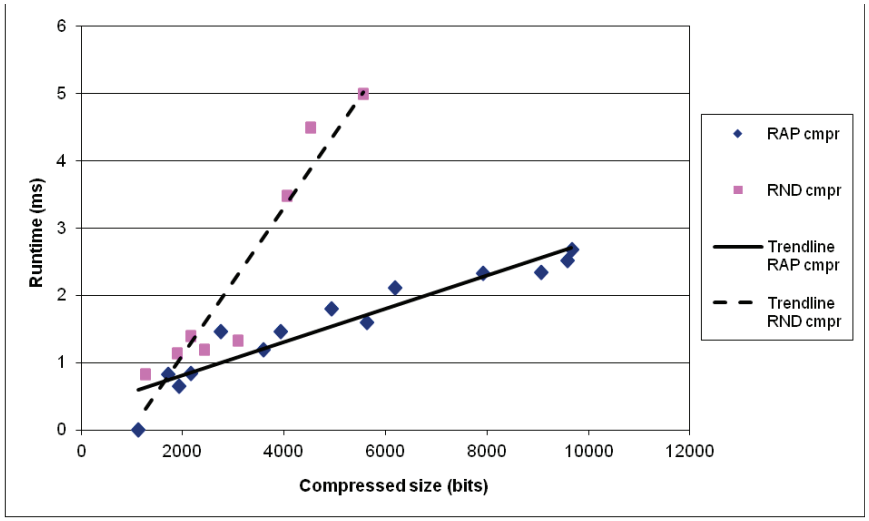


Figure 20: Results achieved on graph coloring using Subdue

on the random and real-world (RAP) graphs mentioned in Section 3. The results are depicted in Figures 17-22. In these plots, each data point is the average of up to 7 measurement results. It should be noted that the vertical axis has logarithmic scale in all cases.

For small sizes, the RAP and RND (or RAP cmpr and RND cmpr) functions are close to each other in each case. As the size increases, the difference between the two curves increases in all cases. However, there are significant differences in the rate by which the two functions diverge from each other,

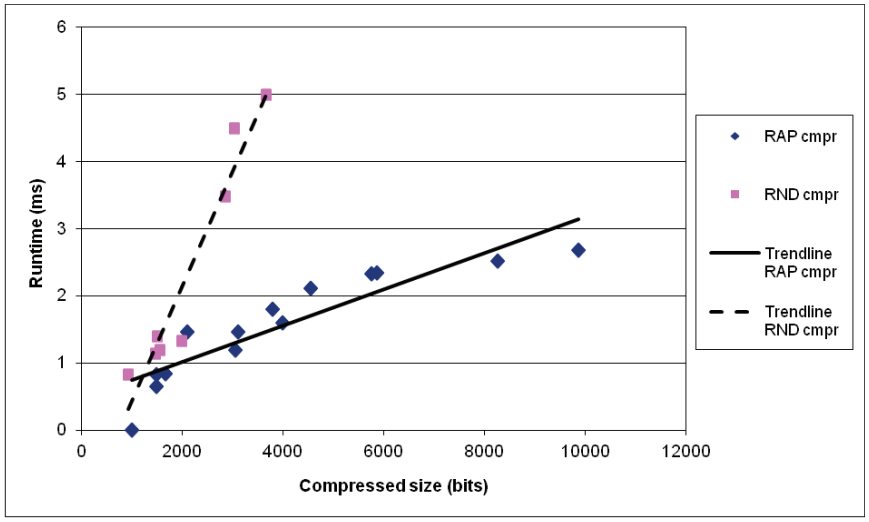


Figure 21: Results achieved on graph coloring using Subdue-recursive

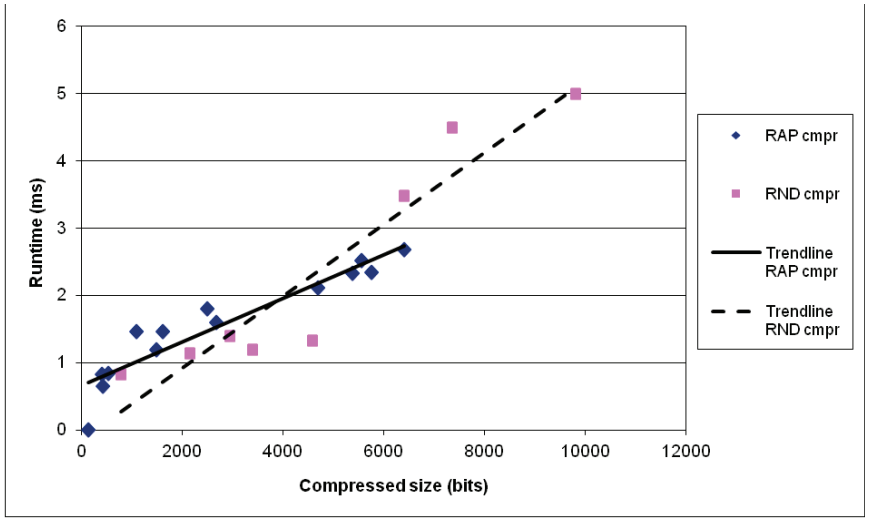


Figure 22: Results achieved on graph coloring using GSA

as demonstrated by the angle between the corresponding trendlines, which are shown in Table 3. In particular, they diverge very rapidly in the cases where the horizontal axis shows uncompressed size (Figure 17), the size after compression with Subdue (Figure 20) or Subdue-recursive (Figure 21). The difference grows less rapidly in the case of ZIP (Figure 18), and the situation is even better with PAQ9 (Figure 19) and GSA (Figure 22).

A similar conclusion can be drawn from the quantitative results as well; these are shown in Table 3.

Table 3: Quantitative comparison of the investigated compression methods on graph coloring (for the prediction ratio and the correlation coefficient, higher numbers are better; for the angle between trendlines and the p-value, smaller numbers are better)

Compression method	Angle between trendlines	Prediction ratio	Correlation coefficient	p-value
No compression	55°	1	0.39	0.042
ZIP	25°	4.07	0.85	$<$0.001
PAQ	17°	6.83	0.92	$<$0.001
Subdue	40°	1.03	0.57	0.004
Subdue-recursive	49°	0.9	0.46	0.018
GSA	14°	11.30	0.93	$<$0.001

As can be seen from the table, GSA provides the best performance both in terms of prediction ratio and correlation coefficient, closely followed by PAQ and ZIP. This is due to the fact that GSA makes the most use of the structure present in the RAP graphs, and hence, GSA compresses RAP graphs to one third of their uncompressed size on average, whereas it can hardly compress RND graphs. Incidentally, ZIP and PAQ achieve an even better compression ratio for RAP graphs, but they can also compress RND graphs, hence their sensitivity to structure is less than that of GSA, and in our application, this sensitivity is more important than the pure compression ratio. Altogether, GSA has 11 times more predictive power concerning solver runtime than the uncompressed input size. Moreover, since p-values less than 0.05 are usually considered statistically significant, we can conclude that all rows of the table contain statistically significant values for the correlation coefficient.

Finally, we also compared the performance of GSA, ZIP, and PAQ on #SAT benchmarks, as described in Section 3. We did not include Subdue and Subdue-recursive in this comparison because we concluded from the experiments with graph coloring that they are less appropriate for our purposes.

The charts resulting from these experiments were similar to the ones already presented for graph coloring, hence we do not present them here. Instead, we show the quantitative results in Table 4.

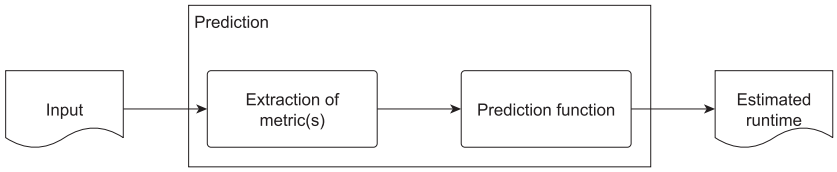
As can be seen, PAQ and ZIP are again much better than the classic estimation based on uncompressed size, but GSA clearly outperforms them. The fact that GSA achieves such impressive results on this second problem as well (which is not related to graph coloring) is a strong sign of the usefulness of the suggested approach.

Table 4: Quantitative comparison of the investigated compression methods on #SAT

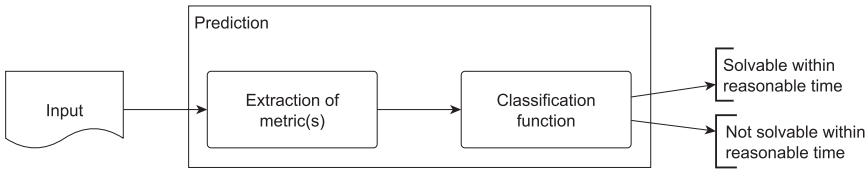
Compression method	Prediction ratio	Correlation coefficient	p-value
No compression	1	0.62	<0.001
ZIP	9.38	0.86	<0.001
PAQ	4.03	0.84	<0.001
GSA	10.49	0.92	<0.001

6. Discussion

After having presented the GSA algorithm and empirical results of its practical application, it is worth revisiting the goals of this research and discussing to what extent they have been reached.



(a) Numerical runtime prediction



(b) Runtime categorization

Figure 23: Theoretical model of runtime prediction

As shown in Figure 23, there are two main scenarios for predicting the runtime of an algorithm on a given input. Both scenarios consist of two steps: first some metrics are extracted from the input, which are then fed into a prediction or classification function that computes the estimated runtime or predicts whether or not the solver will finish within reasonable time, respectively. The second step is necessarily algorithm-specific, but the first step can be algorithm-independent, or even problem-independent.

The most widely used metric, the problem's size, is totally algorithm- and even problem-independent. Its usage is ubiquitous: whenever we talk about polynomial, exponential, linear, quadratic etc. algorithms, we always mean it with respect to this metric. Moreover, claims like "this algorithm can cope with graphs of up to several hundred nodes and thousands of edges" are

quite common. In such cases, the input's size is used for classifying inputs as easy vs. hard.

However, as we pointed out in the Introduction, using the input's size as a metric for runtime prediction (either numerically or with classification) suffers from the problem that inputs of the same size can lead to significantly different runtimes, particularly when comparing structured and random instances of the same size. Whatever the prediction or classification function is, this difference will cause a large error in the estimation.

Hence, our main objective was to propose a metric that has the same merits – i.e. can be used for any problem and any algorithm, is easily computed and intuitive – but mitigates the above problem. We proposed the input's compressed size as new metric. Using a generic compression program like ZIP or PAQ9, this metric can be easily computed for *any* problem instance, and is completely algorithm-agnostic. In our empirical experiments, we demonstrated that the compressed size allows clearly better hardness classification than the uncompressed size. We also showed on two case studies that instances of the same compressed size yield runtimes that are much closer to each other than runtimes arising from instances of the same uncompressed size. That is, the compressed size is indeed as general as the uncompressed size, but will yield more precise estimations – with *any* reasonable[4] (explicit or implicit) prediction function.

Our second objective was to show that a genetic algorithm that searches for the most compact structural description of the input yields a compression that is even more suitable for hardness classification and runtime prediction than generic compression utilities, such as ZIP and PAQ9. As shown in the empirical results, GSA yields consistently better results than ZIP and PAQ9 both for hardness classification and runtime prediction, although the difference is not so significant. It should also be noted that – in contrast to ZIP and PAQ9 – GSA is a research prototype, that would benefit from further engineering effort. In this respect, our results with this first prototype are already very promising.

In order to be fair, it should also be stated that in terms of compression ratio, ZIP and PAQ9 are clearly superior to GSA. In particular, ZIP and PAQ9 can also compress random inputs by using non-structural inhomogeneities (e.g., in a graph with vertices numbered from 1 to 200, the digit 1 appears more ofther than the others), whereas GSA cannot. However, GSA's sensitivity to structure is higher, and this is what counts in our case.

The compressed size delivered by GSA is a quite general metric: it can be used for any algorithms and any problems, where (a substantial part of) the

[4]By reasonable, we mean that the prediction function maps bigger differences in its input to bigger differences in its output. This is for example the case (asymptotically) for all polynomial and exponential functions.

input is a graph, or can be represented by a graph. (The #SAT case study was an example for the latter case.) That is, GSA's application domain is not as wide as that of ZIP/PAQ9, but still quite wide, including a significant portion of the practically relevant combinatorial algorithmic problems.

Another issue is simplicity. Obviously, compressing the input using ZIP/PAQ9 and then determining the compressed size is simple. The reader might have the feeling that using GSA is more tedious, because of the tuning and post-processing steps that we described in this paper. However, the tuning was necessary only once, and the post-processing was only needed to make a meaningful comparison between GSA and the other methods. Applying GSA to a new problem does not require any tuning, nor post-processing; it is just as easy as applying ZIP or PAQ9.

Finally, we would like to compare our method with some previous work that we already mentioned in Section 2. In particular, several authors pointed out that the ratio of the number of constraints and number of variables often has a dramatic impact on solver runtime. Hence, this ratio could also be used as a metric for runtime prediction, and it is quite simple. However, this approach would have several drawbacks. Most importantly, it does not solve the issue that random instances are usually harder than structured instances. This is obvious from our experiments: we deliberately chose structured and random instances where the ratio of constraints to variables was roughly equal, but they lead to significantly different solver runtimes. Second, this ratio only makes sense for problems that can be formulated in terms of variables and constraints (constraint satisfaction problems). Third, solver runtime is not monotonous in this ratio; its maximum is highly problem-dependent, which makes reasoning about solver runtime based on the value of the ratio more difficult.

Another approach is the one taken by Leyton-Brown and co-workers [14, 15, 16]. They build a so-called empirical hardness model of an algorithm by extracting dozens of – potentially useful – simple statistical features of the problem instance and using machine learning to find how the algorithm's runtime depends on those features. In the terms of Figure 23, this approach uses a high number of metrics instead of just one; in fact, any metrics that can be extracted easily and might help in improving the prediction. Then, the prediction function is learned using thousands of runs of the algorithm on some training data. During this phase, the weights of the features are determined, and those features that are not important get weights near 0 automatically. In terms of precision of prediction, this method will surely yield better results than our method of using only one generic metric. On the other hand, our method works out of the box on any problem and any algorithm, whereas the method of Leyton-Brown et al requires substantial work before it can be applied to a new problem, a new algorithm or even after a modification of the algorithm. Actually, the two methods can be combined: the compressed size of the input can be used as one of the features used by

the method of Leyton-Brown et al. Since our focus is on the metric, whereas their focus is on the prediction function, the two approaches are largely complimentary.

7. Conclusion and future work

In this paper, we argued that the compressed size of the input to an exact algorithm is a much better metric for hardness classification or to predict the algorithm's runtime than the – commonly used – metric of its uncompressed size. We provided substantial empirical evidence to support this idea. In particular, even using a general-purpose compression method like ZIP or PAQ9, the compressed size has significantly more predictive power than the uncompressed size.

Moreover, we presented GSA, a genetic algorithm for analyzing the structure of a graph, and compressing it using the identified highly structured parts. According to the MDL principle, the shortest description of the graph will best explain its structure, and our algorithm explicitly searches for this shortest description, using an evolution of graph decompositions. According to our empirical results, the compression with GSA provides even higher predictive power than using ZIP or PAQ9, both for hardness classification and numerical runtime prediction.

It should be noted that, in principle, our methods can be applied to a wide range of problems and algorithms, but our experimental evaluation was limited to two problems with one solver for each problem. Although the results are promising, we cannot conclude that these methods will be equally successful for each problem and each solver. In the future, we hope to obtain a better understanding as to when the compressed input size (and GSA in particular) yields such good results.

Our experiments show that using a genetic algorithm for graph structure analysis is a viable option. In particular, as shown in Section 5.2, both the mutation and recombination operations help in improving the decomposition of the graph. Nevertheless, there might be even more effective algorithms than GA for this purpose. A possible future research direction is to experiment with the application of other heuristics.

Another goal for our future work is to further improve the scalability of GSA. Moreover, we plan to investigate to what extent the *content* of the output of GSA can be used to enrich the prediction concerning solver runtime (at the moment, we only use the *length* of the output of GSA).

Acknowledgment

The results discussed above are supported by the grant TÁMOP - 4.2.2.B-10/1–2010-0009. The work of Zoltán Ádám Mann was partially supported

by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

[1] Z. Mann, Optimization in computer engineering – Theory and applications, Scientific Research Publishing, 2011.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, MIT Press, 2009.

[3] H. S. Wilf, Backtrack: an O(1) expected time algorithm for the graph coloring problem, Information Processing Letters 18 (1984) 119–121.

[4] Z. Mann, A. Szajkó, Improved bounds on the complexity of graph coloring, in: Proceedings of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2010, pp. 347–354.

[5] A. Biere, M. J. H. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, Vol. 185 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2009.

[6] D. A. Spielman, S.-H. Teng, Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time, Journal of the ACM 51 (3) (2004) 385–463.

[7] M. Samer, S. Szeider, Fixed-parameter tractability, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, IOS Press, 2009, Ch. 13, pp. 425–454.

[8] L. Zhang, S. Malik, The quest for efficient boolean satisfiability solvers, in: E. Brinksma, K. Larsen (Eds.), Proceedings of the 14th International Conference on Computer Aided Verification, Vol. 2404 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2002, pp. 641–653.

[9] T. Walsh, Search on high degree graphs, in: Proceedings of the 17th International Joint Conference On Artificial Intelligence, 2001, pp. 266–274.

[10] P. Cheeseman, B. Kanefsky, W. M. Taylor, Where the really hard problems are, in: 12th International Joint Conference on Artificial Intelligence (IJCAI '91), 1991, pp. 331–337.

[11] D. Mitchell, B. Selman, H. Levesque, Hard and easy distributions of sat problems, in: Proceedings of the tenth national conference on Artificial intelligence, AAAI'92, AAAI Press, 1992, pp. 459–465.

[12] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, L. Troyansky, Determining computational complexity from characteristic 'phase transitions', Nature 400 (1999) 133–137.

[13] R. Williams, C. P. Gomes, B. Selman, Backdoors to typical case complexity, in: Proceedings of the 18th international joint conference on Artificial intelligence, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003, pp. 1173–1178.

[14] K. Leyton-Brown, E. Nudelman, Y. Shoham, Learning the empirical hardness of optimization problems: The case of combinatorial auctions, in: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming, CP '02, Springer-Verlag, London, UK, 2002, pp. 556–572.

[15] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, Y. Shoham, Understanding random SAT: Beyond the clauses-to-variables ratio, in: Principles and Practice of Constraint Programming (CP-04), 2004, pp. 438–452.

[16] F. Hutter, Y. Hamadi, H. H. Hoos, K. Leyton-Brown, Performance prediction and automated tuning of randomized and parametric algorithms, in: F. Benhamou (Ed.), Principles and Practice of Constraint Programming - CP 2006, Proceedings, 2006, pp. 213–228.

[17] P. D. Grünwald, I. J. Myung, M. A. Pitt (Eds.), Advances in Minimum Description Length, MIT Press, 2005.

[18] M. Li, P. Vitanyi, An Introduction to Kolmogorov Complexity and Its Applications, 3rd Edition, Springer Verlag, 2008.

[19] D. J. Cook, L. B. Holder, Substructure discovery using minimum description length and background knowledge, Journal of Artificial Intelligence Research 1 (1994) 231–255.

[20] D. J. Cook, L. B. Holder, Graph-based data mining, IEEE Intelligent Systems 15 (2000) 32–41.

[21] R. Lutz, Evolving good hierarchical decompositions of complex systems, Journal of Systems Architecture 47 (2001) 613–634. doi:10.1016/S1383-7621(01)00019-4.

[22] T. Hogg, Which search problems are random?, in: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence, AAAI '98/IAAI '98, American Association for Artificial Intelligence, Menlo Park, CA, USA, 1998, pp. 438–443.

[23] T. Szép, Z. Ádám Mann, Graph coloring: The more colors, the better?, in: Proceedings of the 11th International Symposium on Computational Intelligence and Informatics (CINTI 2010), 2010, pp. 119–124.

[24] P. Briggs, K. D. Cooper, L. Torczon, Improvements to graph coloring register allocation, ACM Transactions on Programming Languages and Systems 16 (3) (1994) 428–455.

[25] M. Thurley, sharpSAT – counting models with advanced component caching and implicit BCP, in: Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006), 2006, pp. 424–429.

[26] X. Song, W. N. N. Hung, A. Mishchenko, M. Chrzanowska-Jeske, A. Coppola, A. Kennings, Board-level multiterminal net assignment, in: 12th ACM/IEEE Great Lakes Symposium on VLSI (GLSVLSI), 2002, pp. 130–135.

[27] Z. Mann, T. Szép, BCAT: A framework for analyzing the complexity of algorithms, in: 8th IEEE International Symposium on Intelligent Systems and Informatics, 2010, pp. 297–302.

[28] M. Křivánek, J. Morávek, NP-hard problems in hierarchical-tree clustering, Acta Inf. 23 (1986) 311–323.

[29] T. Feder, P. Hell, S. Klein, R. Motwani, Complexity of graph partition problems, in: Proceedings of the thirty-first annual ACM symposium on Theory of computing, STOC '99, ACM, New York, NY, USA, 1999, pp. 464–472.

[30] S. Böcker, S. Briesemeister, G. W. Klau, Exact algorithms for cluster editing: Evaluation and experiments, Algorithmica 60 (2011) 316–334.

[31] E. B. Baum, D. B. Boneh, C. B. Garrett, Where genetic algorithms excel, Evol. Comput. 9 (1) (2001) 93–124.

Appendix A. Mutation operators

In the following, we present the full list of mutation operators employed in GSA:

1. Creation of a new object.
   1.1 An edge of the input graph is turned into a path of length 1. If one of the end vertices of the edge is not a complex object, but a "further vertex", then the edge is selected with higher probability. If both end vertices are such, then the edge is selected with even higher probability.

1.2 An edge of the input graph is turned into a pseudo-complete graph with 2 vertices and 0 missing edge. Again, "further vertices" are preferred over vertices already covered by complex objects.

1.3 Two edges with a common end vertex are turned into a path of length 2. Again, "further vertices" are preferred over vertices already covered by complex objects.

1.4 Two edges with a common end vertex are turned into a pseudo-complete graph with 3 vertices and 0 or 1 missing edge. Again, "further vertices" are preferred over vertices already covered by complex objects.

1.5 Four vertices are turned into a new object. Depending on the number of edges among the vertices, the new object will be either a pseudo-complete graph, a pseudo-empty graph, or a chaos.

2. Deletion of an object.

2.1 An object with at most 4 vertices is destroyed, turning the vertices and edges not covered by other objects into "further" vertices and edges. The selection probability of an object is determined based on the number of other objects that cover these vertices.

2.2 An object with at least 70% of its vertices covered by other objects is removed, turning the vertices and edges not covered by other objects into "further" vertices and edges.

3. Increasing the size of an object.

3.1 A pseudo-complete graph is extended with a new vertex. The selection probability depends on the number of edges between the pseudo-complete graph and the new vertex. (The more edges, the higher the selection probability.)

3.2 A pseudo-empty graph is extended with a new vertex. The selection probability depends on the number of edges between the pseudo-empty graph and the new vertex. (The less edges, the higher the selection probability.)

3.3 A chaos is extended with a new vertex. The selection probability depends on the number of edges between the chaos and the new vertex. (The more edges, the higher the selection probability.)

3.4 A path is extended with a new vertex that is adjacent to the end vertex of the path.

4. Decreasing the size of an object.

4.1 A vertex is removed from a pseudo-complete graph. The selection probability depends on the number of edges between the vertex and the other vertices in the pseudo-complete graph. (The less edges, the higher the selection probability.)

4.2 A vertex is removed from a pseudo-empty graph. The selection probability depends on the number of edges between the vertex and the other vertices in the pseudo-empty graph. (The more edges, the higher the selection probability.)

4.3 A vertex is removed from a chaos. The selection probability depends on the number of edges between the vertex and the other vertices in the chaos. (The less edges, the higher the selection probability.)

4.4 One end vertex of a path is removed from it.

5. Converting an object to another object type.

5.1 If the number of edges among the vertices of a pseudo-complete graph is below a given limit, then the pseudo-complete graph is converted to a chaos.

5.2 If the number of edges among the vertices of a pseudo-empty graph is above a given limit, then the pseudo-empty graph is converted to a chaos.

5.3 If the number of edges among the vertices of a chaos is above a given limit, then the chaos is converted to a pseudo-complete graph.

5.4 If the number of edges among the vertices of a chaos is below a given limit, then the chaos is converted to a pseudo-empty graph.

6. Merging two objects into a single object.

6.1 Two pseudo-complete graphs are merged into a single pseudo-complete graph. The selection probability monotonously increases in the number of common vertices and the number of edges between the two pseudo-complete graphs.

6.2 Two pseudo-empty graphs are merged into a single pseudo-empty graph. The selection probability monotonously increases in the number of common vertices and monotonously decreases in the number of edges between the two pseudo-empty graphs.

6.3 Two paths with a common end vertex are merged into a single path.

7. Splitting an object into two objects.

7.1 A pseudo-complete graph G_1 is split into two pseudo-complete graphs G_2 and G_3. The splitting is performed vertex by vertex. If vertex v is put into, say, G_2, then all other vertices not yet placed into G_2 or G_3 that are not adjacent to v are placed into G_3. We use this heuristic to keep the number of edges between G_2 and G_3 low.

7.2 A pseudo-empty graph G_1 is split into two pseudo-empty graphs G_2 and G_3. This time, if vertex v is put into, say, G_2, then all other vertices not yet placed into G_2 or G_3 that are adjacent to v are placed into G_3. We use this heuristic to keep the number of edges inside G_2 and G_3 low.

7.3 A path is split into two paths at one of its inner vertices.

7.4 A relatively sparse part of a pseudo-complete graph G_1 is chopped off to form a chaos. For this purpose, a vertex v of G_1 is selected (the lower the number of neighbours of v within G_1, the higher the selection probability). Then v, along with all vertices of G_1 not adjacent to v are chopped off from G_1 to form a new chaos object. We use this heuristic to change pseudo-complete graphs in such a way that they remain dense.

7.5 Analogously, a relatively dense part of a pseudo-empty graph can be chopped off to form a chaos. We use an analogous heuristic as before to change pseudo-empty graphs in such a way that they remain sparse.

7.6 A path is removed from a pseudo-empty graph G_1. For this purpose, a vertex v is chosen from G_1 (the higher the number of neighbours of v within G_1, the higher the selection probability), then a paths is grown randomly starting with v, as long as it can be continued.

Now we revisit the examples presented in Figures 6 and 8 in Section 5.2. In Figure 6, a chaos with 6 vertices plus a further vertex are converted into a pseudo-complete graph with 7 vertices. This is the result of two elementary mutation operations: either the chaos is first extended with the further vertex to form a chaos with 7 vertices (mutation operator 3.3), which is then converted to a pseudo-complete graph (operator 5.3); or the chaos is first converted to a pseudo-complete graph (operator 5.3), which is then extended with the further vertex (operator 3.1). In Figure 8, a path consisting of 3 vertices and a further edge are converted to a pseudo-complete graph with 3 vertices. There is no mutation operator that would convert a path into a pseudo-complete graph, hence this happened also in two steps: first the path is destroyed (operator 2.1), and the resulting 3 loose vertices together with the 3 edges among them are then compiled into a pseudo-complete graph (operator 1.4).