Discrete Mathematics and Theoretical Computer Science DMTCS vol. **17:1**, 2015, 201–218

Cost-effectiveness of algorithms

Graham Farr*

Faculty of Information Technology, Monash University, Clayton, Victoria 3800, Australia

received 18th Feb. 2013, revised 14th Feb. 2014, accepted 16th Mar. 2015.

In this paper we discuss how to assess the performance of algorithms for optimisation problems in a way that balances solution quality and time. We propose measures of cost-effectiveness for such algorithms. These measures give the gain in solution quality per time unit over a sequence of inputs, and give a basis for deciding which algorithm to use when aiming for best accumulated solution quality for a given time investment over such an input sequence. Cost-effectiveness measures can be defined for both average-case and worst-case performance. We apply these ideas to three problems: maximum matching, graph colouring and Kolmogorov complexity. For the latter, we propose a cost-effectiveness measure for the time-bounded complexity $K_\tau(x)$, and argue that it can be used to measure the cost-effectiveness both of finding a short program to output x and of generating x from such a program. Under mild assumptions, we show that (roughly speaking) if the time-bounded complexity $K_\tau(x)$ is to be a cost-effective approximation to $K(x)$ then $\tau(n) = O(n^2)$.

Keywords: optimisation, algorithms, cost-effectiveness, approximation algorithm, performance measure, graph colouring, matching, Kolmogorov complexity

1 Introduction

In solving an optimisation problem, we seek an algorithm that is, among other things, (a) fast, in that its worst-case or average-case complexity is reasonably low, and (b) accurate, in that the value of the solution it finds is not far from the value of an optimum solution. In this paper we seek to quantify the trade-off between these two aims, and investigate some situations in which this can be done.

Consider graph colouring, as a motivating example. If we were in the extreme position of having no time to even look at the input graph, and knew only that it had n vertices, and were required to supply a proper (vertex-)colouring, then we would have to give a different colour to each vertex, using n colours in all (the *universal colouring*). If we had a more sensible algorithm, then it would generally use fewer colours. If it uses λ colours, then its *colour gain* is $n - \lambda$, since this is the number of colours the algorithm saves, in comparison with the "knee-jerk response" of just using n colours without looking at the details of the input graph. This gain has come at the cost of taking more time. The quotient

$$\frac{\text{colour gain}}{\text{extra time taken}} \tag{1}$$

*Email: `Graham.Farr@monash.edu`. This work was supported in part by Australian Research Council Large Grant A49703170. Some of the work of this paper was done while the author visited the Department of Computer Science, Royal Holloway, University of London.

tells us how much is gained, on average, per extra time step (*i.e.*, for each time step beyond the time required to produce the universal colouring). For example, consider an algorithm A_1 that takes time $4n$ and uses $n/4$ colours, and an algorithm A_2 that takes time $5n$ and uses $n/5$ colours. Assume that producing the universal colouring takes time n. Then the colour gain per extra time step for A_1 is $(n - n/4)/(4n - n) = 1/4$, while for A_2 it is $(n - n/5)/(5n - n) = 1/5$. The quotient (1) can be thought of as measuring the *cost-effectiveness* of the algorithm for the input in question, so in our simple example A_1 is more cost-effective than A_2. We will see in the next section that these ideas are easily extended to other optimisation problems.

We focus on situations where a long sequence of separate inputs must be solved, so that, when one input is finished with, work starts immediately on the next. We also assume that the overall value of the sequence of solutions obtained is just the sum of the values of the individual solutions. In such cases, we argue that the cost-effectiveness measures we discuss may be useful in deciding whether to use (for example) a fast but not particularly accurate algorithm or a slower near-optimal one.

Our measures should not be used to claim that one such algorithm is *always* better than another. The choice of algorithm will depend on the economics of the situations in which the algorithm is to be used, which can be complex and varied. They do, however, give a basis for such choice in some simple and natural situations.

Our emphasis is on the analysis of algorithms, from the perspective of cost-effectiveness, rather than the design of new algorithms. We find that cost-effectiveness can bring a new perspective to the analysis of algorithms, and that simple, well-known algorithms are sometimes provably more cost-effective in our sense than more sophisticated ones. This happens at greatly differing levels of complexity. We illustrate this point with graph matching (solvable in polynomial time), graph colouring (NP-hard), and then study Kolmogorov complexity (which is uncomputable [7, 21]). For the latter, we consider the cost-effectiveness of time-bounded Kolmogorov complexity, as an approximation, and show that, under reasonable assumptions, it cannot be cost-effective unless the time bound is $O(n^2)$.

Historically, the most common ways of measuring the quality of solutions returned by an approximation algorithm A for an optimisation problem have been: the absolute error, *i.e.*, the difference $f_A - f^*$ (or $f^* - f_A$) between the value f^* of an optimal solution and the value f_A of one found by A; the ratio f^*/f_A (or f_A/f^*) of these values; or the relative error $(f_A - f^*)/f^*$ (or $(f^* - f_A)/f^*$). A number of authors have proposed dividing the error by the difference between values of worst (f_0) and best (f^*) solutions, using $(f_A - f^*)/(f_0 - f^*)$, or, essentially equivalently, replacing the numerator here by the gain given by A over the worst solution, using $(f_0 - f_A)/(f_0 - f^*)$: see, *e.g.*, [1, 2, 3, 5, 8, 10, 11, 14, 27]. The former ratio here is called the *proximity degree* in [5], and is part of the definition of a *z-approximation* in [14]; the latter is called a *differential approximation ratio* in [11, 10]. Hassin and Khuller [14, §1] give a good survey of work on these and related measures. As shown in [3, 27, 11], they are invariant under certain natural transformations of the problem (*affine transformations* in [11]). It is found, for example, that the quality of approximations for finding maximum independent sets and minimum vertex covers in graphs are the same under this measure, since these problems are equivalent under the transformations considered (see the discussion in [11, §1, §3.2.1]). Some of the papers prove results about the class of measures that possess such invariance properties [27, 11]. Some propose using some reference value appropriate to the problem at hand rather than necessarily using the worst value [3, 8, 27], and we find this convenient too: for some problems, it is difficult to find the very worst solution, whereas it may be easier to find some other feasible solution.

Returning to our motivating example, there has been a series of papers on algorithms for approximate graph colouring with respect to these measures [9, 12, 13, 15, 26].

While the papers cited in the previous two paragraphs do take account of the gain over the worst (or reference) solution (as we do), it is not their aim to look at the trade-off between solution quality and time.

A qualitative discussion of aspects of this trade-off can be found in [17, §7.2].

There is a loose analogy between the setting we consider and that of online algorithms (see, *e.g.*, [18]). There, elementary pieces of the input arrive in a sequence, and must be processed without knowing the whole input. The outcome is compared with that of a normal (offline) algorithm which can wait for the entire input and then process it, and so has perfect information about it. Our setting for the study of cost-effectiveness has coarser granularity, in that it considers sequences of whole inputs rather than pieces of inputs. We also use simple (often trivial) heuristics as benchmarks, against which performance is compared, rather than the optimal offline algorithms used for competitive analysis of online algorithms.

In the next section, we develop cost-effectiveness measures in detail. We show how to measure the cost-effectiveness of dealing with single inputs, and introduce average-case and worst-case cost-effectiveness measures. In subsequent sections, we show how these ideas may be applied to discrete optimisation problems at different levels of complexity: maximum matching (§3), graph colouring (§3), and Kolmogorov complexity (§4).

2 Measuring cost-effectiveness

Let $\Pi = (I, sol, m, goal)$ be an optimisation problem, where we use the notation of [19, 4]: I is the set of allowed inputs (*i.e.*, instances), *sol* is a function that assigns a set of feasible solutions $sol(x)$ to any $x \in I$, m is the objective function (assigning a value $m(x, y)$ to any $x \in I$ and $y \in sol(x)$), and $goal \in \{\min, \max\}$ indicates whether the aim is to minimise or maximise $m(x, y)$ over $y \in sol(x)$. We focus on minimisation problems; it is trivial to modify our treatment to deal with maximisation problems, and we frequently do so parenthetically. Note that we do not require that Π belongs to the class NPO (for which see, *e.g.*, [4]), although most cases of practical interest are in that class. For convenience, put $I_n = \{x \in I \mid |x| = n\}$.

A *search algorithm* for Π takes $x \in I$ as input and outputs some $y \in sol(x)$, with $m(x, y)$ hopefully not too far from optimum. An *evaluation algorithm* takes $x \in I$ and outputs some bound (upper for min, lower for max) for the optimal $m(x, y)$, without necessarily exhibiting any y. We overload terminology slightly by speaking of the *value returned* by a search or evaluation algorithm, even though the search algorithm outputs a solution rather than its value. For a search algorithm, the value returned is $m(x, y)$; for an evaluation algorithm, the value returned is just the bound that it outputs.

We assume that the objective function is affinely related to the real value (or cost) of solutions, and that this value/cost is additive over sequences of inputs.

Suppose we have two approximation algorithms A_1 and A_2 for Π. Our discussion will mostly allow them to either both be search algorithms, or both be evaluation algorithms.

We wish to compare these algorithms with respect to their cost-effectiveness, using some measure that depends only on the time taken and the value returned. To facilitate such comparison, let $x_1, \ldots, x_N \in I_n$ be a sequence of n-bit inputs with the property that, for each of them, A_1 takes time t_1 and returns value f_1, and A_2 takes time t_2 and returns value f_2. Suppose without loss of generality that $t_1 < t_2$. We restrict our attention to $t_1 = \Omega(n)$, $t_2 = \Omega(n)$.

We want to compare the quality of solutions obtained when the algorithms use the same total running time. Let T be the total time taken by A_1 on all these N inputs. In this time, it is not possible for the slower A_2 to process as many inputs as A_1. So there comes a point when A_2 must be abandoned, and the *leftover inputs* (*i.e.*, those which A_1 had the time to process, but A_2 did not) must be accounted for somehow, with respect to both solution value and time. We suppose that each leftover input is deemed to cost t_0 time steps and is assigned a value of f_0.

There are several different approaches to dealing with leftover inputs, including:

(a) Leftover inputs are processed by some algorithm A_0 that is faster than either A_1 or A_2. This seems especially appropriate when some trivial solution can be found in a short time.

(b) Leftover inputs are assigned the worst possible solution value for inputs of size n: for minimisation problems, $f_0 = \max\{m(x, y) \mid x \in I_n, y \in \mathit{sol}(x)\}$. (For maximisation problems, replace $\max$ here by $\min$.) The time taken, t_0, is just the time taken to find this worst possible value. The rationale here is that, if A_2 runs out of time to process the leftover inputs itself, then it cannot be credited with finding anything better than worst possible solutions for the remaining inputs. For many problems, a worst solution can be found easily. For others, finding a worst solution may be difficult — perhaps as hard, or harder, than finding a best solution (*e.g.*, the Travelling Salesman Problem). In such cases, this method of dealing with leftover inputs may not be appropriate.

(c) Suppose there is a set U_n of "universal" solutions that belong to the solution set of every possible input of size n: $U_n = \bigcap_{x \in I_n} \mathit{sol}(x)$. Leftover inputs are assigned the objective function value of some universal solution of size n: $f_0 = m(x, y)$ where $y \in U_n$. (In graph colouring, the colouring that assigns a different colour to each vertex is such a universal solution.) We are, in effect, saying that a universal solution can be proposed at negligible cost, without knowing any details of the input except its size, so it is reasonable to use it if there is no time to do any actual computation on the input. In many cases, approaches (b) and (c) amount to the same thing, as in the graph colouring example discussed earlier. The time t_0 is just the time taken to find such a universal solution. (Typically $t_0 = O(n)$.)

(d) In (b) and (c), we used the same solution value (worst-case or universal, respectively) for all leftover inputs. In accounting for time, we assumed in effect that this solution value was found from scratch for each leftover input. It might be better to assume that such solution values cost us only constant time. The rationale here is that, once such a solution is found, we could just refer to it briefly for each remaining leftover input, rather than keep constructing it from scratch each time. So leftover inputs can be processed in the time it takes to issue a directive along the lines of "use universal solution value". (We need here to assume a setting that allows such a directive to be constructed and understood.)

(e) Sometimes it might be appropriate to assume that leftover inputs are not entitled to proper processing, and so must be assigned a value f_0 that is strictly *worse* than any solution for any $x \in I_n$. This value will be greater than (or less than, for a max problem) the value of any worst-case value such as that used in (b) above. The chosen f_0 should be easy to compute, even if the value of the worst solution is not. In this approach, it is reasonable to make t_0 constant. It might help to envisage augmenting the set of all possible solutions by a special symbol that indicates that there was not enough time to find a valid solution, and giving this special 'solution' the value f_0.

(f) In (e), it might be appropriate to give leftover inputs time $t_0 = 0$. This can lead to a simpler treatment, but may run into problems. In principle, it allows arbitrarily many inputs to be given some kind of solution value (even if a bad one, as a penalty) in zero total time, which does not seem computationally realistic.

For the reasons given in (f), we assume $t_0 \geq 1$ unless stated otherwise. The exact forms of the penalties envisaged in (e)–(f) depend on the particular problem, and on the use to which the solutions are to be put. For the time being, we suppose that each $x_i \in I_n$ in our sequence of inputs gives the same leftover solution value $f_0 = f_0(x_i)$.

The *combined solution value* found by an algorithm A for the sequence of inputs $x_1, \ldots, x_N \in I$ is defined to be just the sum of the solution values found by A for the individual inputs in the sequence: $f(x_1, \ldots, x_N) := \sum_{i=1}^{N} f(x_i)$, where the latter $f(x)$ is unary and denotes the solution value found by A for input x, while the former f is N-ary. (The inputs in the sequence may be given, or random, or chosen by an adversary, according to context.) In many cases, the inputs $x_1, \ldots, x_N$ can be combined to form a single larger input $\sum_{i=1}^{N} x_i$, to the same problem Π, whose value under f is the sum of the values under f of the individual inputs. Typically, $sol(\sum_{i=1}^{N} x_i) = \sum_{i=1}^{N} sol(x_i)$ and $m(\sum_{i=1}^{N} x_i, (y_1, \ldots, y_N)) = \sum_{i=1}^{N} m(x_i, y_i)$, with the optimum solution f^* obeying $f^*(\sum_{i=1}^{N} x_i) = \sum_{i=1}^{N} f^*(x_i)$. For example, if $G_1, \ldots, G_N$ are graphs (regarded as inputs to the graph colouring problem), then the sum $G_1 + \cdots + G_N$ is obtained by taking disjoint copies of the G_i and adding all possible edges between G_i and G_j for each i, j $(1 \leq i < j \leq N)$. Here, $G_1 + \cdots + G_N$ is itself a graph (and so an input to the original problem in its own right), and not just an N-tuple or formal sum. Observe that $\chi(G_1 + \cdots + G_N) = \sum_{i=1}^{N} \chi(G_i)$, where χ denotes chromatic number.) In such cases, our use of addition to combine solution values is especially well motivated.

While this will not cover all combination methods that are useful in practice, it is simple and natural. Also, some combination methods can be put in this form by an appropriate transformation of values. For example, if the solution values for some particular problem are regarded as probabilities that should really be multiplied rather than added (with inputs assumed to be independent in an appropriate sense), then we can work with their logarithms and so make them fit into our framework. We do not propose to cover all possible combination methods in full generality, but focus on addition.

Suppose A_2 deals with r inputs, leaving s leftover inputs, and A_1 deals with $r + s$ inputs. The time taken by A_2 is $rt_2 + st_0$, and the time taken by A_1 is $(r + s)t_1$. These must both equal T, which implies

$$\begin{aligned} r &= \frac{T}{t_1} \cdot \frac{t_1 - t_0}{t_2 - t_0}\,, \\ s &= \frac{T}{t_1} \cdot \frac{t_2 - t_1}{t_2 - t_0}\,. \end{aligned}$$

Under our measures, the combined solution values F_1 and F_2 obtained by A_1 and A_2 respectively are

$$\begin{aligned} F_1 &= (r + s)f_1, \\ F_2 &= rf_2 + sf_0. \end{aligned}$$

We can use F_1 and F_2 to compare the performance of the two algorithms, given the same sequence of inputs and the same total computation time. Algorithm A_1 is to be preferred if and only if $F_1 < F_2$. Making the above substitutions for r and s, we see that this occurs if and only if

$$\frac{t_1 - t_0}{t_2 - t_0} \cdot f_2 \; + \; \frac{t_2 - t_1}{t_2 - t_0} \cdot f_0 \; > \; f_1\,, \tag{2}$$

i.e., in a plot of solution value versus time, (t_1, f_1) lies below the line joining (t_0, f_0) and (t_2, f_2). This condition (2) holds if and only if

$$\frac{f_0 - f_1}{t_1 - t_0} > \frac{f_0 - f_2}{t_2 - t_0}\,, \tag{3}$$

which gives each algorithm its own side of the inequality. (The inequalities for this paragraph are reversed for a max problem.)

We therefore propose the following simple measure of cost-effectiveness of an algorithm A when running on input x for which it takes time $t = t(x)$ and produces solutions of value $f = f(x)$:

$$\mathrm{ce}(A; x) = \frac{f_0 - f}{t - t_0}\,, \tag{4}$$

where (as above) t_0 and f_0 are the time taken and value assigned to leftover inputs. (For a max problem, we negate the expressions for the cost-effectiveness functions ce(_) and ce_1(_). We meet the latter below.) Note that, if for some reason the differences $f_0 - f^*$ and $f - f^*$ from the optimum value f^* are more readily available than the values f_0 and f themselves, then it may be convenient to use

$$\mathrm{ce}(A; x) = \frac{(f_0 - f^*) - (f - f^*)}{t - t_0}\,.$$

We emphasise that our cost-effectiveness measure must always be based on some assumed way of accounting for leftover inputs, with associated t_0 and f_0, such as outlined in (a)–(f) above, and that t_0 must be less than the time t taken by A. (If $t_0 \ll t$ then the approximation $\mathrm{ce}(A; x) \simeq (f_0 - f)/t$ may be convenient.) If leftover inputs are dealt with by approach (a), then we are essentially measuring the cost-effectiveness of algorithm A relative to a faster algorithm A_0. This raises the possibility of then measuring the cost-effectiveness of A_0 relative to some even faster algorithm, and so on. We eventually face the question of whether it is reasonable to nominate some ultimate basis against which cost-effectiveness can be measured, or equivalently some way of accounting for leftover inputs that uses the least possible time. Sometimes (e) above might provide this. However, it may not always be suitable. In practice, if A is to be applied to a sequence of inputs but there is not enough time to apply it to them all, then it may well be advantageous to use some faster but nontrivial algorithm for some of the later inputs, rather than just push A to the bitter end and leave the leftover inputs to be dealt with in some trivial fashion. The choice of how to deal with leftovers, and of t_0 and f_0, depends on the situation in which the algorithm is being used.

Now consider the cost-effectiveness of dealing with the whole sequence of inputs $\mathbf{x} = (x_1, \ldots, x_N)$. Treating the whole sequence like a single large input, it is natural to extend (4) as follows:

$$\mathrm{ce}(A; \mathbf{x}) = \frac{f_0(\mathbf{x}) - f(\mathbf{x})}{t(\mathbf{x}) - t_0(\mathbf{x})}\,. \tag{5}$$

Assuming additive combination of solution values as discussed above, we have

$$\mathrm{ce}(A; \mathbf{x}) = \frac{\sum_i f_0(x_i) - \sum_i f(x_i)}{\sum_i t(x_i) - \sum_i t_0(x_i)}. \tag{6}$$

Suppose the successive x_i are drawn independently from the same probability distribution P_n on I_n. For any $g \in \{f_0, f, t_0, t\}$, write $\overline{g} = \overline{g}(n)$ for the expectation of $g(x)$ over $x \in I_n$, according to P_n. Taking the limit of (6) as $N \to \infty$ yields the following measure of *average-case cost-effectiveness* for an algorithm A and input size n:

$$\text{ce}_1(A; P_n; n) = \frac{\overline{f_0}(n) - \overline{f}(n)}{\overline{t}(n) - \overline{t_0}(n)}. \tag{7}$$

It is worth noting that, if $f^{(\max)}(n) = \max_{x \in I_n} f(x)$ and $t^{(\max)}(n) = \max_{x \in I_n} t(x)$ are the worst-case solution value and time, respectively, for A on inputs of size n, then

$$\text{ce}_1(A; P; n) \geq \frac{\overline{f_0}(n) - f^{(\max)}(n)}{t^{(\max)}(n) - \overline{t_0}(n)}.$$

(For a max problem, this lower bound must be modified: replace $f^{(\max)}$ by $f^{(\min)}$ and negate the resulting quotient.)

It is also possible to define an appropriate measure of worst-case cost-effectiveness. The starting point is again (4), but now the input $x \in I_n$ is supposed to be chosen by an adversary. This may suggest the measure

$$\min_{x \in I_n} \frac{f_0(x) - f(x)}{t(x) - t_0(x)}.$$

However, this is just 0 if there exists $x \in I_n$ such that $f_0(x) = f(x)$, as might often occur (for example, in graph colouring, when x is a complete graph). When it does happen, it may be because algorithm A performs badly on that x, in which case it may be reasonable enough for cost-effectiveness to be zero. However, in some cases (such as our complete graph example) the input itself is so unfavourable that there simply does not exist a solution value better than $f_0(x)$. In such cases it seems unfair to set the cost-effectiveness to be zero. One way out is to scale the cost-effectiveness by the difference between $f_0(x)$ and the optimal solution value $f^*(x)$. Instead of measuring gain in solution quality per unit of time spent (above leftover time), we measure the *proportionate* gain, compared with the gain for an optimal solution, per unit time. The case $f_0(x) = f^*(x)$ still requires special treatment, but continuity now serves as a guide. We obtain

$$\begin{aligned} \text{ce}_2(A; n) \quad &= \quad \min \ \left\{ \frac{f_0(x) - f(x)}{f_0(x) - f^*(x)} \cdot \frac{1}{t(x) - t_0(x)} \mid x \in I_n \,, f_0(x) \neq f^*(x) \right\} \\ &\qquad \cup \left\{ \frac{1}{t(x) - t_0(x)} \mid x \in I_n \,, f_0(x) = f^*(x) \right\} \end{aligned} \tag{8}$$

Note that (8) is written so as to highlight the presence here (as the first of the two quotients on the right hand side) of a close relative of the z-approximation measure of Zemel [14, 27]. Indeed, if t and t_0 depend only on $|x|$, then (for a given A) we just want to minimise that measure.

It is tempting to introduce the measure

$$\begin{aligned} \text{ce}_3(A; n) \quad &= \quad \min_{x \in I_n} \frac{f_0(x) - f(x)}{f_0(x) - f^*(x)} \cdot \frac{t^*(x) - t_0(x)}{t(x) - t_0(x)} \\ &= \quad \min_{x \in I_n} \frac{\text{ce}(A; x)}{\text{ce}(A^*; x)} \end{aligned} \tag{9}$$

where $t^*(x)$ is the time taken to find the optimal solution using some optimal algorithm A^*. This has the advantage that it is unchanged by affine transformations on our measure of time. It has the disadvantage of dependence on choice of A^*.

Alternatives to ce_2, which avoid all mention of finding optimal solutions, may be to replace f^* by either (i) some more efficiently computable lower bound g for f^* (so we would need to consider the choice of g, and how to account for the time taken to compute it), or (ii) some approximation $f^{(t)} \geq f^*$ computable in some specified time bound $t(n)$ (which may give a sort of time-bounded differential approximation ratio). We do not pursue any of these alternatives here, but rather focus on $\mathrm{ce}_2(\_)$ as our worst-case measure.

In situations where the leftover quantities f_0 and t_0 depend only on n, not on x itself, it can be useful to observe that

$$\mathrm{ce}_2(A;n) \geq \frac{f_0(n) - f^{(\max)}(n)}{f_0(n) - f^*(n)} \cdot \frac{1}{t^{(\max)}(n) - t_0(n)}, \tag{10}$$

where $f^*(n) = \min_{x \in I_n} f^*(x)$. This lower bound on worst-case cost-effectiveness might be a good approximation to it, if inputs for which poor solution values are found tend to be those that take a long time to deal with, but this will certainly not always be the case. The quantities $f^{(\max)}(n)$ and $t^{(\max)}(n)$ are often readily found, perhaps from a published analysis of the algorithm in question, so (10) may give a convenient back-of-envelope bound for worst-case cost-effectiveness.

For max problems, the definitions of the worst-case cost-effectiveness measures $\mathrm{ce}_2(\_)$ and $\mathrm{ce}_3(\_)$ are unchanged (although, for the lower bound (10), $f^{(\max)}$ must be replaced by $f^{(\min)}$).

Not surprisingly, worst-case cost-effectiveness appears to be easier to work with than the average-case measure. This is analogous to the situation for ordinary time complexity: worst-case complexity is easier to work with than average-case complexity, and is more widely studied, partly for this reason, even though average-case complexity is often a better measure of the real cost in time. The next two sections use worst-case cost-effectiveness. The final section, §4, uses the single-input measure $\mathrm{ce}(A;x)$.

3 Graph problems

The numbers of vertices and edges in a graph G are denoted by n and m respectively, and $\overline{m}$ is the number of edges in the complement $\overline{G}$ of G.

The problem of finding a maximum matching in a graph G is well known to be solvable in polynomial time (see [22]). The best time complexity bound known is $O(m\sqrt{n})$ for the algorithms of Hopcroft and Karp [16], for bipartite graphs, and Micali and Vazirani [24], for general graphs. To assess cost-effectiveness of matching algorithms, we put $f_0(n) = 0$ (assuming, in effect, that leftover inputs are given the null matching $\emptyset$, as a universal solution) and assume $t/t_0 = \Omega(t)$ for any running time $t = t(n)$ of any of the algorithms we shall consider. The worst-case cost-effectiveness of the Hopcroft-Karp and Micali-Vazirani algorithms is then seen to satisfy

$$\mathrm{ce}_2(A^*;n) = O((m\sqrt{n})^{-1}),$$

where A^* is either of these algorithms.

Consider now any simple linear-time algorithm A for finding a maxim*al* matching in a graph. It is easy to show that a maximal matching must be at least half the size of a maximum one. From this we have the following.

Theorem 1 *The worst-case cost-effectiveness of A is $\Omega(m^{-1})$.* □

This gives an example of a problem for which an optimal polynomial time algorithm exists but for which a faster suboptimal algorithm is provably more cost-effective.

The chromatic number $\chi(G)$ of a graph G is known to be NP-hard to determine, and difficult even to approximate (see [4, pp. 371–372]). A number of authors have considered the complementary problem of maximising $n - \chi(G)$, which gives the number of colours that are *unused* when compared with the universal colouring [9, 12, 13, 15, 26]. This turns out to be easier to approximate. The cited authors give algorithms for which they establish lower bounds on the ratio $(n - f(G))/(n - \chi(G))$ of: 1/2 [9, 15], 2/3 [15], 5/7 [12], 3/4 [13, 26]. Here, $f(G)$ is the number of colours actually used by the algorithm in question.

To study the cost-effectiveness of graph colouring algorithms, we assume that the leftover strategy is to use universal colourings, so that $f_0(n) = n$. Assume again that t_0 is small enough that $t/t_0 = \Omega(t)$, where t is as usual the running time of the algorithm under consideration. For a given colouring algorithm A, we wish to determine or estimate $\mathrm{ce}_2(A; n)$, where now I_n is the set of graphs on n vertices. To do this, we must minimise (over $G \in I_n$) the product of the above ratio $(n - f(G))/(n - \chi(G))$ and the reciprocal of $t - t_0$: see (8).

Hassin and Lahav [15, Algorithm 1] begin by proposing the following simple algorithm, which we call HL1: given G, find a maximal matching in $\overline{G}$, and each pair of vertices so matched in $\overline{G}$ becomes a colour class (of size 2) in G, and all the unmatched vertices become singleton colour classes. (We assume that the maximal matching is found in the obvious greedy sequential manner.) HL1 is shown to achieve performance ratio $1/2$, the same as had been proved for a more complex relative in [9, Theorem 1], and to have time complexity linear in $\overline{m}$. We now consider its cost-effectiveness.

Theorem 2 *For suitable representations of the input graph,* $\mathrm{ce}_2(\mathit{HL1}; n) = \Theta(\overline{m}^{-1})$.

Proof: Suppose the input graph G is represented in such a way that the edges in $\overline{G}$ are easily extracted. A data structure that contains an adjacency list representation of $\overline{G}$ would suffice. For such representations, we have $t - t_0 \le c\overline{m}$ (for an appropriate constant c). Also, $(n - f(G))/(n - \chi(G)) \ge 1/2$, as shown in [15]. Hence

$$\begin{aligned}
\mathrm{ce}_2(\mathrm{HL1}; n) &= \min_{G \in I_n} \frac{n - f(G)}{n - \chi(G)} \cdot \frac{1}{t(G) - t_0} \\
&\ge \min_{G \in I_n} \frac{n - f(G)}{n - \chi(G)} \cdot \min_{G \in I_n} \frac{1}{t(G) - t_0} \\
&\ge \frac{1}{2} \cdot \frac{1}{c\overline{m}}.
\end{aligned}$$

We now turn to the upper bound. Consider the null graph $\overline{K}_n$. If HL1 is run on $\overline{K}_n$, the maximum matching found in its complement K_n has $\lfloor n/2 \rfloor$ edges, and the time taken to find it is $c'\overline{m}$, where c' is a constant independent of n and $\overline{m} = |E(\overline{K}_n)|$. The colouring thus found by HL1 has $\lceil n/2 \rceil$ colours, and the ratio $(n - f(G))/(n - \chi(G)) = 1/2 + o(n^{-1})$. Hence

$$\begin{aligned}
\mathrm{ce}_2(\mathrm{HL1}; n) &\le \frac{n - f(G_k)}{n - \chi(G_k)} \cdot \frac{1}{t(G_k) - t_0} \\
&\le \frac{1}{2c''\overline{m}}
\end{aligned}$$

for an appropriate constant c''. The result follows. □

For some other kinds of representation of G (*e.g.*, an adjacency list representation of G only), we would have $\mathrm{ce}_2(\mathrm{HL1}; n) = \Theta(m^{-1})$.

The running time of Hassin and Lahav's main algorithm [15, Algorithm 3] is given as $O(n\overline{m})$, and inputs where this bound is attained can be used to establish an upper bound on cost-effectiveness of $O((n\overline{m})^{-1})$. Those authors offer an improvement to speed up their algorithm. While that improvement does not lower their worst-case complexity bound, it is possible that further improvements may do so, and so lead to improved worst-case cost-effectiveness.

Halldórsson's algorithm [13] runs in time $O(n^3)$. Input graphs that take this long for the algorithm to colour can be used to give an upper bound $O(n^{-3})$ on the algorithm's cost-efffectiveness.

Tzeng and King's algorithm [26] includes a step that finds a maximum matching in $\overline{G}$. It is necessary that the matching found be maxim*um* rather than merely maxim*al*, so with a best known matching algorithm, and sufficiently "slow" inputs, the cost-effectiveness is $O((\overline{m}\sqrt{n})^{-1})$.

All the algorithms discussed in this section were designed specifically to make the ratio $(n-f(G))/(n-\chi(G))$ as large as possible. It is interesting to consider the cost-effectiveness of colouring algorithms developed without this specific objective in mind. Consider the classical sequential colouring heuristic [23]: look at the vertices in order, and give each vertex the smallest colour not used by any of its already-coloured neighbours. This takes time $O(m)$.

Theorem 3 *The sequential colouring heuristic satisfies* $(n - f(G))/(n - \chi(G)) \geq 1/2$ *and has worst-case cost-effectiveness* $\Omega(m^{-1})$.

Proof: Suppose the heuristic is applied to a graph G, and that it uses $f(G) = k$ colours on G. If $k \leq n/2$, then $(n - f(G))/(n - \chi(G)) \geq (n - n/2)/(n - 0) = 1/2$. If $k \geq n/2$, then there are at least $k - (n - k) = 2k - n$ colours that are each only assigned to a single vertex. Let the set of these singleton-colour-class vertices be U. These vertices must form a clique in G, since otherwise the algorithm would not have given them all different colours. (In particular, if $u, v \in U$ with $uv \notin E(G)$ and (without loss of generality) $u < v$ in the vertex ordering, then v could have been given the same colour as u.) Hence G has a $(2k-n)$-clique, and $\chi(G) \geq 2k-n$. Hence $(n-f(G))/(n-\chi(G)) \geq (n-k)/(n-(2k-n)) = 1/2$. This proves the first claim of the theorem.

It is straightforward to find graphs for which the algorithm takes time $\Theta(m)$. The bound on cost-effectiveness follows. □

For cost-effective graph colouring, it looks like a good approach might be to use sequential colouring for graphs with fewer edges than their complement and HL1 for (appropriately represented) graphs with more edges than their complement.

It will be seen from this and the previous section that, for cost-effectiveness, it is generally better to have a fast algorithm with *some* constant differential approximation ratio than a slower one with a *better* (or even optimum) ratio. Indeed, in graph colouring we have seen that simple sequential algorithms can be the most cost-effective, even though their performance guarantees are among the weakest. It would be interesting to investigate how widespread this phenomenon is.

4 Kolmogorov complexity

Suppose we have a string x and we wish to find a short description of x, in the form of a program p that, when given to some fixed universal Turing machine U, causes U to output $x = U(p)$ and stop. (We assume that the set of legal programs forms a prefix code.) The length of a shortest such p is the ***Kolmogorov complexity*** $K(x)$ of x. For further information, see [21]. $K(x)$ is well known to be uncomputable: see [7] or [21, Theorem 2.6]. Some of the interest in Kolmogorov complexity comes from inductive inference, where x is regarded as data of some kind and p is regarded as a model, or explanation, for x. In such cases it can be of considerable practical importance to find short strings p that generate x in the above manner. Since finding a shortest such p is uncomputable, we must settle for approximations.

In this section we discuss cost-effective approximation of $K(x)$ (the evaluation problem) and of the associated search problem of finding p. We also consider cost-effectiveness in relation to the generation of x from p.

Since we cannot compute $K(x)$ in general — when there is no limit on the time we must allow a program p to take to generate a string x — it is natural to put some limit $\tau(n)$ on the time taken by p when $|x| = n$. The length of the shortest p that generates x under this time restriction is the ***time-bounded Kolmogorov complexity*** $K_{\tau(n)}(x)$ [21, §7.1]. We restrict our attention to computable τ, so that $K_{\tau(n)}(x)$ is computable, and provides a computable upper bound on $K(x)$. The higher we allow $\tau(n)$ to be, the better will be our approximation to $K(x)$. We ask: what time limit $\tau(n)$ is most cost-effective? We consider this for the search, evaluation, and generation problems in turn.

To answer this question, we will apply our cost-effectiveness measure $\mathrm{ce}(A; x)$ to time-bounded Kolmogorov complexity. To do so, we need as always to settle on t_0 and f_0. In this section, we suppose that the values returned are at least as large as the maximum possible (see options (d), (e)). We also suppose that our $f_0(x)$ depends only on n, so we may put $f_0(n) = f_0(x)$.

Recall that the maximum possible value of $K(x)$, over strings x of length n, is $n + K(n) + O(1)$, and that no computable upper bound to this value is always within an additive constant of it. We therefore suppose that $f_0(n)$ is some easily computable strict upper bound: $f_0(n) > \max\{K(x) \mid |x| = n\}$. (For example, we could take $f_0(n) = n + 2\log n + c$, for some suitable constant c.)

We will argue below that the following measure of cost-effectiveness is appropriate:

$$\kappa(n, x, \tau) = \frac{f_0(n) - K_{\tau(n)}(x)}{\tau(n) - \tau_0(n)},$$

where we will usually want $\tau_0(n) = O(f_0(n))$ but postpone further consideration of τ_0 for the moment. The similarity of the measure $\kappa(n, x, \tau)$ to those introduced in §2 will be evident. The main difference is that the denominator here uses the resource-bounded Kolmogorov complexity's time bound $\tau(n)$, rather than the actual time taken to compute $K_{\tau(n)}$.

We will need to use a standard program for generating a string x, along the lines of "print x, then stop" (appropriately formulated in the prefix code recognised by the universal Turing machine in question): call this program $p_0(x)$. We can assume that the map $x \mapsto p_0(x)$ is computable. This implies $|p_0(x)| > n + K(n) + c$, for some constant c.

4.1 The search problem

For the search problem, each leftover string x will be processed by a simple algorithm that takes x as input and outputs the program $p_0(x)$. Let $t_0(n)$ be the time complexity of this algorithm. It is reasonable to insist that $t_0(n) \leq af_0(n)$ for some a.

Let $A^{(\tau)}$ be the following basic search algorithm for computing $K_{\tau(n)}(x)$. A string x is given as input, with $n = |x|$. Take all programs p of length $< f_0(n)$ and run them in parallel (or, at least, simulate their parallel execution). Each p is stopped after $\tau(n)$ steps, if it has not already done so. Of those programs that have halted naturally (*i.e.*, without being forced to do so), we find a shortest one that produced output x. Call it p_τ. The length of this program is $K_{\tau(n)}(x)$.

We assume that the total time taken by $A^{(\tau)}$ can be measured (to sufficient accuracy for this exercise) by $t(n) = \alpha 2^{f_0(n)}\tau(n)$, for some α independent of n. Certainly $\alpha = 1$ would give an easy upper bound, and it is routine to show that $\alpha = \Omega$ gives a lower bound, where Ω is the probability that U eventually halts when given a random program p [6].

Following §2, it is natural to measure the cost-effectiveness of this algorithm by

$$\kappa'(n, x, \tau) = \frac{f_0(n) - K_{\tau(n)}(x)}{t(n) - t_0(n)} \cdot$$

For leftover inputs, it is natural to use the trivial search algorithm that just outputs $p_0(x)$, described at the start of this section. Hence we have

$$\kappa'(n, x, \tau) = \frac{1}{\alpha 2^{f_0(n)}} \cdot \frac{f_0(n) - K_{\tau(n)}(x)}{\tau(n) - t_0(n)(\alpha 2^{f_0(n)})^{-1}},$$

The above expression differs from the measure $\kappa(n, x, \tau)$ only by (i) a factor of $\alpha 2^{f_0(n)}$, which is independent of x, $\tau(n)$ and $K_{\tau(n)}(x)$, and (ii) our explicit choice of $\tau_0(n) = t_0(n)(\alpha 2^{f_0(n)})^{-1}$. So the most cost-effective approximation to $K(x)$, among those obtained by computing $K_{\tau(n)}(x)$ using $A^{(\tau)}$, is the one obtained by choosing τ to maximise $\kappa'(n, x, \tau)$; it will also maximise $\kappa(n, x, \tau)$ with appropriate τ_0 (and in any case the exact choice of τ_0 is unimportant if it is small). This justifies using $\kappa(n, x, \tau)$ as a cost-effectiveness measure, provided we recognise that it is not measured in quite the same units as the measures proposed in §2.

Other search algorithms are possible, of course, and may suggest different cost-effectiveness measures. Consider, for example, the universal search algorithm of Levin [20] (see also [21, §7.4.1]). If a program p outputs x in τ steps, then the Levin search algorithm finds p in $2^{|p| + \log \tau + 1}$ steps, with cost-effectiveness

$$\frac{f_0(n) - |p|}{2^{|p| + \log \tau + 1}} .$$

This is maximised by the p that minimises

$$|p| + \log \tau - \log(f_0(n) - |p|). \tag{11}$$

This expression (at its minimum) is similar to Levin's

$$Kt(x) = \min_p(|p| + \log \tau).$$

The final term in (11) may be viewed as modifying Levin's Kt to take more careful account of cost-effectiveness, but will have little effect unless $|p| = O(\log n)$.

4.2 The evaluation problem

For the evaluation problem of just finding $K(x)$, leftover strings are handled by just outputting $f_0(n)$ (being a convenient upper bound for $|p_0(x)|$) in an appropriate prefix code. Let $\tau_0(n)$ be the time required to deal with each leftover string in this way. If $f_0(n) = O(n)$ then this time is $\leq a \log^* n$ steps for some constant a. (For the $\log^*$ function, see [25].) We certainly have $\tau_0(n) = O(\log n)$. Since each leftover input is given the same output in this case, we may find it appropriate, in the light of option (d) in §2, to suppose that this "leftover output" is only stated explicitly once, and just represented by a special symbol thereafter. We can then let τ_0 be a constant.

Although the choice of τ_0 is different, the argument of the previous subsection can be applied to the evaluation problem as well as the search problem. Again, we claim that $\kappa(n, x, \tau)$ is an appropriate cost-effectiveness measure.

4.3 Generating the target string

So far we have looked at the cost of finding short programs that output a string x (or their lengths). We now consider the time these programs take, when run by U, to compute x. If a program p was found by $A^{(\tau)}$, then it might take up to $\tau(n)$ steps to compute x. The reduction in program length, compared with the leftover length $f_0(n)$, is $f_0(n) - K_{\tau(n)}(x)$. Leftover inputs x are assigned the programs $p_0(x)$, and the time taken by such a program to output x is taken to be $\tau_0(n) \leq af_0(n)$. If we divide this length reduction by the time taken by p to compute x (with leftover time τ_0 deducted) then we obtain a measure of how cost-effective p is as a description of x. This measure is bounded below by $\kappa(n, x, \tau)$, and will equal it (with our choice of τ_0) if $\tau(n)$ is the actual time taken by p to generate x, rather than just an upper bound.

Note that this is a different kind of cost-effectiveness to that considered earlier. Originally we considered the cost-effectiveness of *finding* a short program for x (or the length of such a program) by a basic search algorithm. In the previous paragraph we considered the cost-effectiveness of *generating* x by a short program. We have seen that the same measure $\kappa(n, x, \tau)$ can be used for both kinds of cost-effectiveness.

4.4 Choice of time bound

We have argued that $\kappa(n, x, \tau)$ is an appropriate cost-effectiveness measure for approximations to Kolmogorov complexity. We now consider the issue of choice of $\tau(n)$.

We will show that, once $\tau(n)$ is large enough, cost-effectiveness cannot be optimum: specifically, it will be less cost-effective than certain fairly small time bounds.

A time bound $\tau_1(n)$ is *standard* if it satisfies each of the following:

(i) $\tau_0(n) < \tau_1(n)$ for all sufficiently large n;

(ii) $\tau_1(n)$ is a computable time bound at least large enough to allow the program $p_0(x)$ to run;

(iii) $\tau_1(n) = O(f_0(n))$;

(iv) $f_0(n) - K_{\tau_1}(x) \geq c$, for some positive constant c and all sufficiently large n.

The essence of these assumptions is that the time bound $\tau_1(n)$ should be reasonably small but not so small as to be useless, and that that the results it yields are better than the leftover values. We briefly discuss some of the assumptions in more detail. We need to ensure that the time bound τ_1 is large enough to allow x to be printed, hence (ii). We will only be interested in situations where $K_{\tau_1}(x)$ is (for large enough n) a strictly better approximation to $K(x)$ than is the leftover value $f_0(n)$, hence (iv). One situation in which (iv) is satisfied is when the leftover strategy is strictly worse than always using $p_0(x)$ for leftover inputs, in the spirit of option (e) in §2. In such a case, τ_1 could be just large enough to allow $p_0(x)$ to run. This would give $\tau_1(n) = O(|p_0(x)|) = O(f_0(n))$, satisfying (iii).

We also assume:

(v) $\tau_0(n) = O(f_0(n))$.

This should be satisfied by any reasonable leftover method.

It is routine to find a standard $\tau_1(n)$ with $\tau_0(n) = O(f_0(n))$. For example, suppose that $f_0(n) = n + 2\log n + c$, the program $p_0(x)$ runs in time $\le \alpha_1 n$ where $\alpha_1 \ge 1$, and $\tau_0(n) = \alpha_2 n$ for some $\alpha_2 > 0$. Then we could use $\tau_1(n) = \alpha' n$ for some $\alpha' > \max\{\alpha_1, \alpha_2\}$.

We are going to compare $K_{\tau_1}(x)$ and $K_{\tau_2}(x)$, as approximations to $K(x)$, in cases where $\tau_1(n)$ is standard and $\tau_2(n) = \Omega(f_0(n)^2)$. (When $f_0(n) = \Theta(n)$, this just means $\tau_2(n) = \Omega(n^2)$.)

Theorem 4 *Suppose $\tau_1(n)$ is standard and $\tau_0(n) = O(f_0(n))$. Then there exists a constant β such that, if $\tau_2(n) \ge \beta f_0(n)^2$ for all sufficiently large n, then (as approximations to $K(x)$) $K_{\tau_2}(x)$ is less cost-effective than $K_{\tau_1}(x)$.*

Proof: Let a, b be constants such that, for all sufficiently large n, $\tau_1(n) \le a f_0(n)$ and $\tau_0(n) \le b f_0(n)$.

Let $\beta > \max\{a/c, b\}$ be a constant, where c is from (iv). Suppose that $\tau_2(n)$ is a time bound such that $\tau_2(n) \ge \beta f_0(n)^2$ for all sufficiently large n.

We have, for sufficiently large n,

$$\begin{aligned}
\kappa(n, x, \tau_2) &= \frac{f_0(n) - K_{\tau_2}(x)}{\tau_2 - \tau_0} \\
&\le \frac{f_0(n)}{\beta f_0(n)^2 - b f_0(n)} \\
&= \frac{\beta^{-1}}{f_0(n) - \beta^{-1} b} \\
&< \frac{\beta^{-1}}{f_0(n) - 1} \qquad \text{(since } \beta > b\text{)} \\
&= \frac{\beta^{-1}}{f_0(n)} \frac{f_0(n)}{f_0(n) - 1} \\
&< \frac{c}{a f_0(n)} \qquad \text{(since } \beta > a/c\text{, and with } n \text{ large enough)} \\
&\le \frac{f_0(n) - K_{\tau_1}(x)}{\tau_1 - \tau_0} \qquad \text{(by (iv) and choice of } a\text{)} \\
&= \kappa(n, x, \tau_1).
\end{aligned}$$

□

We might paraphrase the situation by saying that if a description p of a string x takes $\omega(f_0(n)^2)$ steps to run, then it takes too long to be worth waiting for (or finding). Explanations that take a long time to process may not be cost-effective.

On the positive side, it follows that (under our mild assumptions) we can always find a program p that outputs x and maximises the cost-effectiveness: such p can be found by the following algorithm, B_β. This algorithm works exactly like $A^{(\tau)}$ with $\tau(n) = \beta n^2$ except that, from among all programs p that halted naturally and generated x, it chooses one that maximises the ratio

$$\frac{f_0(n) - |p|}{\tau_p - \tau_0(n)},$$

where τ_p is the time taken by p to generate x (with p being not necessarily the shortest program that does so). This p maximises the cost-effectiveness of generating x, *i.e.*, it is a most cost-effective explanation of x.

Note, though, that we do not know in advance what the time τ_p taken by a cost-effective p is going to be. We do know by Theorem 4 that $\tau_p \leq \beta f_0(n)^2$, which is why B_β works. But B_β itself is not in general cost-effective. (Note that B_β is just one way of *finding* such a p; it is not p itself, and the p so found may run in time less than βn^2.)

These observations contrast with the uncomputability of Kolmogorov complexity: although finding shortest explanations is uncomputable, finding most cost-effective explanations is not.

5 Further work

We have only opened up this topic and given some initial results. Here we mention a few of the many possible directions for future work.

Firstly, what are the most cost-effective algorithms for classical combinatorial optimisation problems? After considering our main running example of graph colouring, we suggested using sequential colouring or HL1 according to edge density. It would be worth looking at how to make this into a precise algorithm, as cost-effectively as possible. What about other specific problems?

Secondly, some experimental investigation of these ideas would complement the work of this paper and should help in setting further directions for theoretical enquiry.

Thirdly, one could try to prove more general results on cost-effectiveness that apply to whole classes of problems rather than just to specific problems. Are there classes of problems where the problems within a class exhibit similar behaviour, with respect to cost-effectiveness? Is there a useful notion of reducibility between problems? (It is not clear that the kind of classification used in complexity theory for decision problems, or optimisation problems, is necessarily going to be appropriate here.) What are the characteristics of problems whose algorithms have lowest cost-effectiveness?

Fourthly, what about enumeration problems? How would we approximate, in a cost-effective way, problems such as counting the colourings of a graph, or evaluating its Tutte polynomial at some fixed argument? How cost-effective are Markov Chain Monte Carlo methods?

Lastly, there is certainly room for debate on how to formulate cost-effectiveness measures. This paper presents some proposals and highlights some of the issues. Further discussion is welcome.

Acknowledgements

I am grateful for helpful comments by Cristian Calude, Margaret Mitchell and the referees.

References

[1] A. Aiello, E. Burattini, M. Furnari, A. Massarotti and F. Ventriglia. Computational complexity: the problem of approximation. In: *Algebra, Combinatorics and Logic in Computer Science* (Györ, Hungary, 1983), *Colloquia Mathematica Societatis János Bolyai*, 42:51–62, 1983.

[2] A. Aiello, E. Burattini, A. Massarotti and F. Ventriglia. A new evaluation function for approximation algorithms. In: *Proceedings of Informatica 77* (Bled, Yugoslavia, 1977), pages 1–4, 1977.

[3] A. Aiello, E. Burattini, A. Massarotti and F. Ventriglia. Towards a general principle of evaluation for approximate algorithms. *RAIRO Inform. Théor.*, 13:227–239, 1979.

[4] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, Berlin, 1999.

[5] G. Ausiello, A. D'Atri and M. Protasi. Structure preserving reductions among convex optimization problems. *J. Comput. System Sci.*, 21:136–153, 1980.

[6] G. J. Chaitin. A theory of program size formally identical to information theory. *J. Assoc. Comput. Mach.*, 22:329–340, 1975.

[7] G. J. Chaitin, A. Arslanov and C. S. Calude. Program-size complexity computes the halting problem. *EATCS Bulletin*, 57:198–200, 1995.

[8] G. Cornuejols, M. L. Fisher and G. L. Nemhauser. Location of bank accounts to optimize float: an analytic study of exact and approximate algorithms. *Management Sci.*, 23:789–810, 1977.

[9] M. Demange, P. Grisoni and V. Th. Paschos. Approximation results for the minimum graph coloring problem. *Inform. Process. Lett.*, 50:19–23, 1994.

[10] M. Demange, P. Grisoni and V. Th. Paschos. Differential approximation algorithms for some combinatorial optimization problems. *Theoret. Comput. Sci.*, 209:107–122, 1998.

[11] M. Demange and V. Th. Paschos. On an approximation measure founded on the links between optimization and polynomial approximation theory. *Theoret. Comput. Sci.*, 158:117–141, 1996.

[12] M. M. Halldórsson. Approximating discrete collections via local improvements. In: *Proc. 6th ACM-SIAM Symp. on Discrete Algorithms* (San Francisco, January 1995), pages 160–169. ACM, New York, 1995.

[13] M. M. Halldórsson. Approximating k-set cover and complementary graph coloring. In: W. H. Cunningham, S. T. McCormick and M. Queyranne, editors, *Integer Programming and Combinatorial Optimization: Proc. 5th Internat. IPCO Conf.* (Vancouver, BC, 1996), *Lecture Notes in Computer Science* 1084, pages 118–131. Springer, Berlin, 1996.

[14] R. Hassin and S. Khuller. z-Approximations. *J. Algorithms*, 41:429–442, 2001.

[15] R. Hassin and S. Lahav (Haddad). Maximizing the number of unused colors in the vertex coloring problem. *Inform. Process. Lett.*, 52:87–90, 1994.

[16] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2:225–231, 1973.

[17] J. Hromkovič, *Algorithms for Hard Problems*. Springer, Berlin, 2001.

[18] S. Irani and A. R. Karlin. Online computation. In: D. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*, Chapter 13, pages 521–564. PWS Publishing Company, Boston, Ma., USA, 1996.

[19] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 9:256–278, 1974.

[20] L. A. Levin. Universal sequential search problems. *Problems Inform. Transmission*, 9:265–266, 1973. Russian original: *Problemy Peredachi Informatsii*, 9:115–116, 1973.

[21] M. Li and P. M. B. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications* (2nd edition). Springer Verlag, New York, 1997.

[22] L. Lovász and M. D. Plummer. *Matching Theory*, North-Holland Mathematics Studies 121, *Ann. Discrete Math.* 29, North-Holland, Amsterdam, 1986.

[23] D. W. Matula, G. Marble and J. D. Isaacson. Graph coloring algorithms. In: R. C. Read, editor, *Graph Theory and Computing*, pages 109–122. Academic Press, New York, 1972.

[24] S. Micali and V. V. Vazirani. An $O(\sqrt{|V|}\,|E|)$ algorithm for finding maximum matching in general graphs. In: *Proc. 21st Ann. IEEE Symp. on Foundations of Comput. Sci.*, pages 17–27. IEEE, New York, 1980.

[25] J. Rissanen. A universal prior for integers and estimation by Minimum Description Length. *Ann. Statist.*, 11:416–431, 1983.

[26] W.-G. Tzeng and G.-H. King. Three-quarter approximation for the number of unused colors in graph coloring. *Inform. Sci.*, 114:105–126, 1999.

[27] E. Zemel. Measuring the quality of approximate solutions to zero-one programmming problems. *Math. Oper. Res.*, 6:319–332, 1981.