available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cosrev

Survey

# Certifying algorithms

R.M. McConnell[a], K. Mehlhorn[b,*], S. Näher[c], P. Schweitzer[d]

[a] Computer Science Department, Colorado State University Fort Collins, USA
[b] Max Planck Institute for Informatics and Saarland University, Saarbrücken, Germany
[c] Fachbereich Informatik, Universität Trier, Trier, Germany
[d] College of Engineering and Computer Science, Australian National University, Canberra, Australia

ARTICLE INFO

*Article history:*
Received 2 June 2010
Received in revised form
29 September 2010
Accepted 29 September 2010

*Keywords:*
Algorithms
Software reliability
Certification

ABSTRACT

A certifying algorithm is an algorithm that produces, with each output, a certificate or witness (easy-to-verify proof) that the particular output has not been compromised by a bug. A user of a certifying algorithm inputs $x$, receives the output $y$ and the certificate $w$, and then checks, either manually or by use of a program, that $w$ proves that $y$ is a correct output for input $x$. In this way, he/she can be sure of the correctness of the output without having to trust the algorithm.

We put forward the thesis that certifying algorithms are much superior to non-certifying algorithms, and that for complex algorithmic tasks, only certifying algorithms are satisfactory. Acceptance of this thesis would lead to a change of how algorithms are taught and how algorithms are researched. The widespread use of certifying algorithms would greatly enhance the reliability of algorithmic software.

We survey the state of the art in certifying algorithms and add to it. In particular, we start a theory of certifying algorithms and prove that the concept is universal.



## 1. Introduction

One of the most prominent and costly problems in software engineering is correctness of software. When the user gives $x$ as an input and the program outputs $y$, the user usually has no way of knowing whether $y$ is a correct output on input $x$ or whether it has been compromised by a bug. The user is at the mercy of the program. A *certifying algorithm* is an algorithm that produces with each output a *certificate* or *witness* that the *particular output* has not been compromised by a bug. By inspecting the witness, the user can convince himself that the output is correct, or reject the output as buggy. He is no longer on the mercy of the program. Fig. 1 contrasts a standard algorithm with a certifying algorithm for computing a function $f$.

A user of a certifying algorithm inputs $x$ and receives the output $y$ and the witness $w$. He then checks that $w$ proves that $y$ is a correct output for input $x$. The process of checking $w$ can be automated with a *checker*, which is an algorithm for verifying that $w$ proves that $y$ is a correct output for $x$. In many cases, the checker is so simple that a trusted implementation of it can be produced, perhaps even in a different language where the semantics are fully specified. A formal proof of correctness of the implementation of the certifying algorithm may be out of reach; however, a formal proof of the correctness of the checker may be feasible. Having checked

* *Corresponding author.*
E-mail address: mehlhorn@mpi-inf.mpg.de (K. Mehlhorn).

1574-0137/$ - see front matter 
doi:10.1016/j.cosrev.2010.09.009

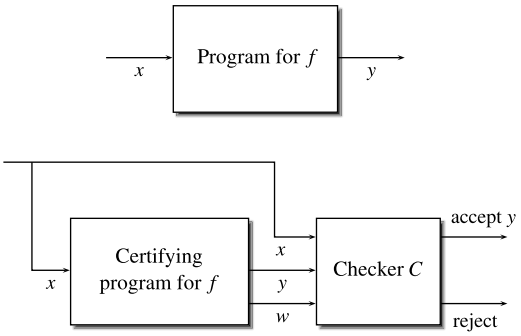



**Fig. 1 – The top figure shows the I/O behavior of a conventional program for computing a function $f$. The user feeds an input $x$ to the program and the program returns an output $y$. The user of the program has no way of knowing whether $y$ is equal to $f(x)$. The bottom figure shows the I/O behavior of a certifying algorithm, which computes $y$ and a witness $w$. The checker $C$ accepts the triple $(x, y, w)$ if and only if $w$ is a valid witness for the equality $y = f(x)$.**

the witness, the user may proceed with complete confidence that output $y$ has not been compromised.

We want to stress that it does not suffice for the checker to be a simple algorithm. It is equally important that it also easy for a user to understand *why* $w$ proves that $y$ is a correct output for input $x$.

A tutorial example that we describe in more detail below is the problem of recognizing whether a graph is bipartite, that is, whether the vertices can be partitioned into two sets such that all edges have an endpoint in each set. A non-certifying algorithm for testing bipartiteness outputs a single bit $y$; the bit tells whether the graph is bipartite or not. A certifying algorithm does more; it proves its answer correct by providing an appropriate witness $w$, see Fig. 2. If the graph is bipartite, then $w$ is a bipartition. If the graph is not bipartite, $w$ is an odd-length cycle in $G$. Clearly odd-length cycles are not bipartite. Thus an odd-length cycle proves that a graph is nonbipartite. For the user of the program there is no need to know that and why a nonbipartite graph always contains an odd-length cycle. The checker verifies either that all edges have endpoints in both bipartition classes, or verifies that the cycle is, in fact, a cycle, that it has odd length, and that all the edges of the cycle exist in the graph. We come back to this example in Section 2.1 where we will show that a simple modification of the standard algorithm for deciding bipartiteness makes the algorithm certifying.

*We put forward the thesis that certifying algorithms are much superior to non-certifying algorithms and that for complex algorithmic tasks only certifying algorithms are satisfactory. Acceptance of this thesis would lead to a change of how algorithms are taught and how algorithms are researched. The widespread use of certifying algorithms would greatly enhance the reliability of algorithmic software. In this paper, we survey the state of the art in certifying algorithms, give several new certifying algorithms, and start a theory of certifying algorithms. In particular, we show that every program can be made weakly certifying without asymptotic loss of efficiency. This assumes that a correctness proof in some formal system is available.*

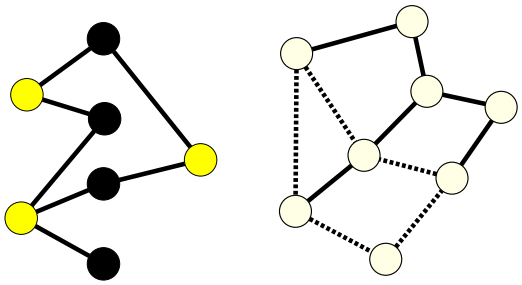


**Fig. 2 – The graph on the left is bipartite. The two sides of the bipartition are indicated by node colors. The graph on the right is nonbipartite. The edges of an odd-length cycle are highlighted. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)**

The usage of certifying algorithms protects not only against incorrect programs, but even against incorrect algorithms. It allows the use of arbitrarily complicated algorithms and programs to produce $y$ from $x$, which may be beyond the competence of the user to understand. All that is required for the user, in order to accept an output $y$ with the certainty that it has not been corrupted, is an easy proof of why $w$ proves that $y$ is a correct output for $x$. In particular, there is no need for the user to know that or understand why a certificate exists for all valid input–output pairs and how this certificate is computed.

The occurrence of an error is recognized immediately when the checker fails to authenticate the validity of the certificate. This means either that $y$ was compromised, or that only $w$ was compromised. In either case, the user rejects $y$ as untrusted. If, in fact, the program is correct, *an occasion never arises when the user has reason to question the program's output.* This is despite the fact that the correctness of the implementation may not be known with certainty to anybody, even the programmer.

The reason why the approach is more practical to implement is that it sidesteps completely the issue of whether the certifying algorithm is correctly implemented, which is difficult, by showing only that a particular output is correct, which is often easy. There are nevertheless many problems for which certifying algorithms are not yet known, and thinking of an appropriate certificate is an art form that is still developing.

In the discussion above, we used vague terms such as "simple" or "easy to understand". We will make these notions more precise in later sections, however, we will not give a formal definition of certifying algorithm. We hope that the reader will develop an intuitive understanding of what constitutes a certifying algorithm and what does not, which will allow him to recognize a certifying algorithm when he sees one.

The designers of algorithms have traditionally *proved that their algorithm is correct*. However, even a sound proof for the correctness of an algorithm is far from a guarantee that an *implementation* of it is correct. There are numerous examples of incorrect implementations of correct algorithms (see Section 3).

*History:* The notion of a certifying algorithm is ancient. Already al-Kharizmi in his book on algebra described how to (partially) check the correctness of a multiplication; see Section 10.2. The extended Euclidean algorithm for greatest common divisors is also certifying; see Section 2.3. All primal–dual algorithms in combinatorial optimization are certifying. Sullivan and Masson [1,2] advocated certification trails as a means for checking the instance correctness of a program. The seminal work by Blum and Kannan [3] on programs that check their work put result checking in the limelight. There is however an important difference between certifying algorithms and their work. Certifying programs produce with each output an easy-to-check certificate of correctness; Blum and Kannan are mainly concerned with checking the work of programs in their standard form. Mehlhorn and Näher were the first to recognize the potential of certifying algorithms for software development. In the context of the LEDA project [4,5], they used certifying algorithms as a technique for increasing the reliability of the LEDA implementations. The term "certifying algorithm" was first used in [6]. Before that Mehlhorn and Näher used the term *Program Checking* or *Result Checking*, see [7–9]. We give a detailed account of extant work in Section 4.

*Generality:* How general is the approach? The pragmatic answer is that we know of 100+ certifying algorithms; in particular, there are certifying algorithms for the problems that are usually treated in an introductory algorithms course, such as connected and strong connectedness, minimum spanning trees, shortest paths, maximum flows, and maximum matchings. Also, Mehlhorn and Näher succeeded in making many of the programs in LEDA certifying. We give more than a dozen examples of certifying algorithms in this paper.

The theoretical answer is that every algorithm can be made weakly certifying without asymptotic loss of efficiency; however, there are problems that do not have a strongly certifying algorithm; see Section 5. A strongly certifying algorithms halts for all inputs. The witness proves that either the input did not satisfy the precondition or the output satisfies the postcondition. It also tells which of the two alternative holds. A weakly certifying algorithm is only required to halt for inputs satisfying the precondition. The witness proves that either the input did not satisfy the precondition or the output satisfies the postcondition, but it does not tell which alternative holds. The construction underlying the positive results is artificial and requires a correctness proof in some formal system. However, the result is also assuring: certifying algorithms are not elusive. The challenge is to find natural ones.

*Relation to testing and verification:* The two main approaches to program correctness are program testing and program verification. *Program testing* [10] executes a given program on inputs for which the correct output is already known by some other means, e.g., by human effort or by execution of another program that is believed to be correct. However, in most cases, it is infeasible to determine the correct output for all possible inputs by other means or to test the software on all possible inputs. Thus testing is usually incomplete and therefore bugs may evade detection; testing does not show the absence of bugs. The Pentium bug is an example [11]. Certifying programs greatly enhance the power of testing. A certifying program can be tested on every input. The test is whether the checker accepts the triple $(x, y, w)$. If it does not, either the output or the witness is incorrect.

*Program verification* refers to (formal) proofs of correctness of programs. The principles are well established [12,13]. However, handwritten proofs are only possible for small programs owing to the complexity and tediousness of the proof process. Using computer-assisted proof systems, formal proofs for interesting theorems were recently given, e.g., for the four-color theorem [14] and the correctness of an implementation of Dijkstra's shortest path algorithm [15]. A difficulty of the verification approach is that, strictly speaking, it requires the verification of the entire hardware and software stack (processor, operating system, compiler) and that it can only be applied to programming languages for which a formal semantics is available. For many popular programming languages, e.g., C, C++, and Java, this is not the case. The project Verisoft [16] undertakes the verification of a complete hardware and software stack. Checkers are usually much simpler than the algorithms they check. Therefore formal verification of the checker will be easier than formal verification of the program itself; see [17] for an example. Moreover, the checker has usually lower asymptotic complexity than the program, and so the checker could be written in a possibly less efficient language with a formal semantics or without the use of complex language features.

*Organization of paper:* In the upcoming Section 2, we first illustrate the concept on four tutorial examples suitable for the undergraduate computer-science curriculum (testing whether a graph is bipartite, determining the number of connected components of a graph, verifying shortest path distances, and computing greatest common divisors). We then give an example that illustrates a simple certificate for an optimization problem that is complicated to solve (finding a maximum matching in a general graph). We conclude the section with an account of how a bug in the LEDA module for planarity testing led Mehlhorn and Näher to the conclusion that certifying algorithms should be a design principle for algorithmic libraries. In Section 3 we give some examples of program failures in widely distributed software and Section 4 discusses extant work. In Section 5 we start a theory of certifying algorithms and formally introduce three kinds of certifying algorithms. We show that every deterministic program can be made weakly certifying without asymptotic loss of efficiency. This assumes that a formal proof of correctness is available. In Section 6 we discuss checkers and in Section 7 we highlight the advantages of certifying algorithms. General techniques for the development of certifying algorithms are the topic of Section 8. In Section 9 we give further examples of certifying algorithms from a variety of subfields of algorithmics and also survey the literature on certifying algorithms. Randomized algorithms and checkers are the topic of Section 10. In Section 11, we discuss the relation between certification and verification. Section 12 discusses certification in the context of data structures. Finally, Section 13 discusses the implications for teaching algorithms, Section 14 lists some open problems, and Section 15 offers some conclusions.

## 2. First examples

### 2.1. Tutorial example 1: testing whether a graph is bipartite

A graph $G = (V, E)$ is called *bipartite* if its vertices can be colored with two colors, say red and green, such that every edge of the graph connects a red and a green vertex. Consider the function

*is_bipartite* : set of all finite graphs $\to \{0, 1\}$,

which for a graph $G$ has value 1 if the graph is bipartite and has value 0 if the graph is nonbipartite. A conventional algorithm for this function returns a boolean value and so does a conventional program. Does the bit returned by the conventional program really give additional information about the graph? We believe that it does not.

What are witnesses for being bipartite and for being nonbipartite? What could a certifying algorithm for testing bipartiteness return? A witness for being bipartite is a two-coloring of the vertices of the graph, indeed this is the definition of being bipartite. A witness for being nonbipartite is an odd cycle, i.e., an odd-length sequence of edges $(v_0, v_1)$, $(v_1, v_2), \ldots, (v_{2k}, v_{2k+1})$ forming a closed cycle, i.e., $v_0 = v_{2k+1}$. Indeed, for all $i$, the vertices $v_i$ and $v_{i+1}$ must have distinct colors, and hence all even numbered nodes have one color and all odd numbered nodes have the other color. But $v_0 = v_{2k+1}$ and hence $v_0$ must have both colors, a contradiction.

Three simple observations can be made at this point:

First, a two-coloring certifies bipartiteness (in fact, it is the very definition of bipartiteness) and it is very easy to check whether an assignment of colors red and blue to the vertices of a graph is a two-coloring.

Second, an odd cycle certifies nonbipartiteness, as we argued above. Also, it is very easy to check that a given set of edges is indeed an odd cycle in a graph. Observe that, in order to check the certificate for a given input, there is no need to know that a nonbipartite graph always contains an odd cycle. One only needs to understand, that an odd cycle proves nonbipartiteness.

It is quite reasonable to assume that a program C for testing the validity of a two-coloring and for verifying that a given set of edges is an odd cycle can be implemented correctly. It is even within the reach of formal program verification to prove the correctness of such a program.

Let us finally argue that there is certifying algorithm for bipartiteness, i.e., an algorithm that computes said witnesses. In fact, a small variation of the standard greedy algorithms will do:

1. Choose an arbitrary node $r$ and color it red; also declare $r$ unfinished.
2. As long as there are unfinished nodes choose one of them, say $v$, and declare it finished. Go through all uncolored neighbors, color them with the other color (i.e., the color different from $v$'s color), and add them to the set of unfinished nodes. Also, record that they got their color from $v$. When step 2 terminates, all nodes are colored.

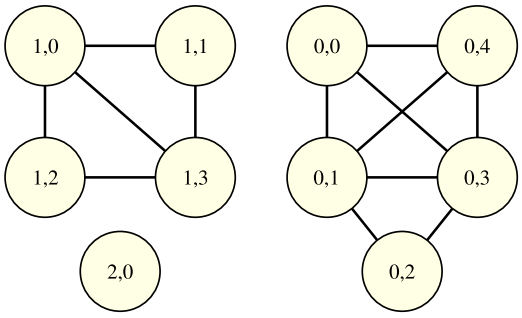



**Fig. 3 – A graph with three connected components. The vertices are labeled with pairs $(i, j)$. The first label is the number of the component to which the vertex belongs and the second label is the number within the component. Within a component, every vertex has a smaller numbered neighbor except for the vertex numbered zero.**

3. Iterate over all edges and check that their endpoints have distinct colors. If so, output the two-coloring. If not, let $e = (u, v)$ be an edge whose endpoints have the same color. Follow the paths $p_u$ and $p_v$ of color-giving edges from $u$ to $r$ and from $v$ to $r$; the paths together with $e$ form an odd cycle. Why? Since $u$ and $v$ have the same color, the paths $p_u$ and $p_v$ either have both even length (if the color of $u$ and $v$ is the same as the color of $r$) or have both odd length (if the color of $u$ and $v$ differs from $r$'s color). Thus the combined length of $p_u$ and $p_v$ is even and hence the edge $(u, v)$ closes an odd cycle.

We summarize: There is a certifying algorithm for testing bipartiteness of graphs and it is reasonable to assume the availability of a correct checker for the witnesses produced by this algorithm.

### 2.2. Tutorial example 2: the connected components of an undirected graph

It is not hard to find the connected components of a graph. However, we show that once they are found, a witness can be produced that makes it possible to check the result even more simply.

We number the components. The witness assigns a pair of nonnegative numbers $(i, j)$ to each vertex $v$. The first number $i$ of the pair is the number of the component to which $v$ belongs. The second number $j$ is the number of the vertex within the component. We number the vertices within a component such that every vertex except for one has a lower numbered neighbor. Observe that such a numbering proves that every vertex within a component is connected to the vertex with the smallest vertex number. Fig. 3 shows an example.

To check this certificate, it suffices to check, for each node, that its labels are nonnegative, for each edge, that the endpoints have the same component number and distinct vertex numbers within the component, and to mark the endpoint with the larger number if it is not already marked, and for each component, that exactly one of its vertices is unmarked.

### 2.3. Tutorial example 3: greatest common divisor

The greatest common divisor of two nonnegative integers $a$ and $b$, not both zero, is the largest integer $g$ that divides $a$ and $b$. We write $g = \gcd(a, b)$. The Euclidean algorithm for computing the greatest common divisor is one of the first algorithms invented. In its recursive form it is as follows.

Procedure $GCD(a, b)$: $a$ and $b$ are integers with $a \geq b \geq 0$ and $a > 0$
If $b = 0$, return $a$;
return $GCD(b, a \bmod b)$;

The Euclidean algorithm is non-certifying. A simple modification, known as the extended Euclidean algorithm, makes it certifying. In addition, to computing $g = \gcd(a, b)$, it also computes integers $x$ and $y$ such that $g = xa + yb$.[1]

**Lemma 1.** *Let $a$, $b$ and $g$ be nonnegative integers, $a$ and $b$ not both zero, and let $g = xa + yb$ for integers $x$ and $y$. If $g$ divides $a$ and $b$ then $g = \gcd(a, b)$.*

**Proof.** Let $d$ be any divisor of $a$ and $b$. Then

$$g = xd\frac{a}{d} + yd\frac{b}{d} = d\left(x\frac{a}{d} + y\frac{b}{d}\right)$$

and hence $d$ divides $g$. In particular, $\gcd(a, b)$ divides $g$. Since $g$ divides $a$ and $b$, $g$ divides $\gcd(a, b)$. Thus $g = \gcd(a, b)$. □

It is easy to extend the recursive procedure above such that it also computes appropriate $x$ and $y$: If $a > b = 0$ then $\gcd(a, b) = a = 1 \cdot a + 0 \cdot b$, and if $a > b > 0$ and $\gcd(a \bmod b, b) = x(a \bmod b) + yb$ then $\gcd(a, b) = \gcd(a \bmod b, b) = x(a \bmod b) + yb = x(a - \lfloor a/b \rfloor b) + yb = xa + (y - \lfloor a/b \rfloor)b$. This leads to the following recursive program.

Procedure $EGCD(a, b)$; assumes $a \geq b \geq 0$ and $a > 0$;
returns $(g, x, y)$ such that $g = \gcd(a, b)$
and $g = xa + yb$.
If $b = 0$, return $(a, 1, 0)$;
let $(g, y, x) = EGCD(b, a \bmod b)$; return $(g, x, y - \lfloor a/b \rfloor)$.

Thus, to check the correctness of an output of the extended Euclidean algorithm, it suffices to verify for the provided integers $x$ and $y$ (they constitute the witness) that $g = xa + yb$ and that $g$ divides $a$ and $b$.

### 2.4. Tutorial example 4: shortest path trees

Our next example, the computation of shortest paths, is a basic subroutine in numerous algorithms (see e.g. Section 10.4). Let $G = (V, E)$ be a directed graph, let $s$ be a special vertex, the source, and let $c : E \to \mathbb{R}_{>0}$ be a positive cost function on the edges. The cost $c(p)$ of a path $p$ is the sum of the costs of its edges and the cost of a shortest (cheapest) path from $s$ to $v$ is

$$d(v) = \min\{\, c(p) \mid p \text{ is a path from } s \text{ to } v \,\}.$$

[1] The existence of these integers $x$ and $y$ is referred to as the Lemma of Bézout, as he proved the general statement for polynomials. In fact, the extended Euclidean algorithm is a certifying algorithm that computes greatest common divisors in any Euclidean domain.

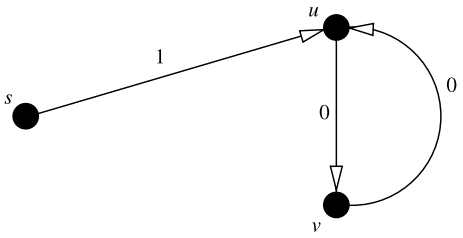



**Fig. 4 – The presence of edges of cost 0 makes stronger justification necessary: the edge $(s, u)$ has cost 1 and the edges $(u, v)$ and $(v, u)$ have cost 0. Then $d(s) = 0$ and $d(u) = d(v) = 1$. However, $D(s) = D(u) = D(v) = 0$ satisfies all three conditions sufficient for certification in the case of a positive cost function.**

How can we verify that an output function $D : V \to \mathbb{R}_{\geq 0}$ is equal to the actual distance function $d$? This is easy: if

$D(s) = 0$ (start value)
for every edge $(u, v)$ $D(v) \leq D(u) + c(u, v)$ (triangle inequality)
for $v \neq s$ there is an edge $(u, v)$ with
$D(v) = D(u) + c(u, v)$ (justification)

then $d(v) = D(v)$ for all $v$.

Indeed, $d(s) = 0$ since the empty path has cost zero and any path has nonnegative cost. Thus $d(s) = D(s)$. We next show $D(v) \leq d(v)$ for all $v$. Consider any $v \neq s$ and let $v_0 = s, v_1, \ldots, v_k = v$ be the path from $s$ to $v$ that defines $d(v)$. Then $d(v_{i+1}) = d(v_i) + c(v_i, v_{i+1})$ for all $i \geq 0$ and $D(v_0) = 0 \leq 0 = d(v_0)$. Assume now that we have shown $D(v_i) \leq d(v_i)$ for some $i \geq 0$. Then

$$\begin{aligned} D(v_{i+1}) &\leq D(v_i) + c(v_i, v_{i+1}) && \text{triangle inequality} \\ &\leq d(v_i) + c(v_i, v_{i+1}) && \text{since } D(v_i) \leq d(v_i) \\ &= d(v_{i+1}) \end{aligned}$$

and so the claim follows by induction. Assume finally, for the sake of a contradiction, that $D(v) < d(v)$ for some $v$. Among the $v$ with $D(v) < d(v)$ choose the one with smallest $D(v)$. Since $D(s) = d(s) = 0$, we know that $v \neq s$. Then there must be a vertex $u$ such that $D(v) = D(u) + c(u, v)$. Also $D(u) < D(v)$ since edge costs are assumed to be positive. By our choice of $v$, $D(u) = d(u)$. Thus there is a path of cost $D(u)$ from $s$ to $u$ and hence a path of cost $D(u) + c(u, v) = D(v)$ from $s$ to $v$. Therefore, $d(v) \leq D(v)$, a contradiction to our assumption that $D(v) < d(v)$ for some $v$.

In order to ease the verification that an output function $D$ is the distance function $d$ a certifying algorithm may for every vertex $v$ indicate the justifying edge $(u, v)$.

Note that the argument above crucially uses the fact that edge weights are positive, see Fig. 4 for an counterexample when edges of weight zero are allowed. In the presence of edges of weight zero, a stronger justification required. In this case we require that every vertex $v$ is assigned an integer $k(v)$, and for every justification edge $(u, v)$ of weight zero $k(u) < k(v)$ holds, i.e. additionally to $D(v) = D(u) + 0$ we require $k(u) < k(v)$.

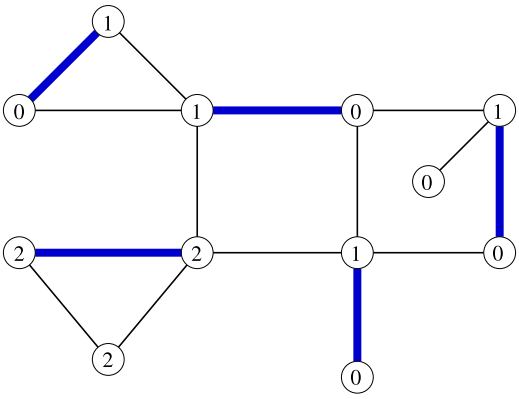



**Fig. 5 – The node labels certify that the indicated matching is of maximum cardinality: All edges of the graph have either both endpoints labeled as two or at least one endpoint labeled as one. Therefore, any matching can use at most one edge with two endpoints labeled two and at most four edges that have an endpoint labeled one. Therefore, no matching has more than five edges. The matching shown consists of five edges.**

### 2.5. *Example: maximum cardinality matchings in graphs*

The previous examples illustrate what we mean by a certifying algorithm on four simple examples, but do not illustrate the full potential of the approach, since, e.g., determining whether a graph is bipartite or determining the connected components of a graph are not difficult problems to begin with.

We now give a more typical example, which shows how the approach can render trivial the checking of correctness of an output produced by an implementation of a complex algorithm. A *matching* in a graph $G$ is a subset $M$ of the edges of $G$ such that no two share an endpoint. A matching has maximum cardinality if its cardinality is at least as large as that of any other matching. Fig. 5 shows a graph and a maximum cardinality matching. Observe that the matching leaves two nodes unmatched, which gives rise to the question whether there exists a matching of larger cardinality. What is a witness for a matching being maximum cardinality? Edmonds [18,19] gave the first efficient algorithm for maximum cardinality matchings. The algorithm is certifying.

An *odd-set cover* $OSC$ of $G$ is a labeling of the nodes of $G$ with nonnegative integers such that every edge of $G$ is either incident to a node labeled 1 or connects two nodes labeled with the same number $i \geq 2$.

**Theorem 1** ([18]). *Let $N$ be any matching in $G$ and let $OSC$ be an odd-set cover of $G$. For any $i \geq 0$, let $n_i$ be the number of nodes labeled $i$. Then*

$$|N| \leq n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor.$$

**Proof.** For $i$, $i \geq 2$, let $N_i$ be the edges in $N$ that connect two nodes labeled $i$ and let $N_1$ be the remaining edges in $N$. Then

$$|N_i| \leq \lfloor n_i/2 \rfloor \quad \text{and} \quad |N_1| \leq n_1$$

and the bound follows. □

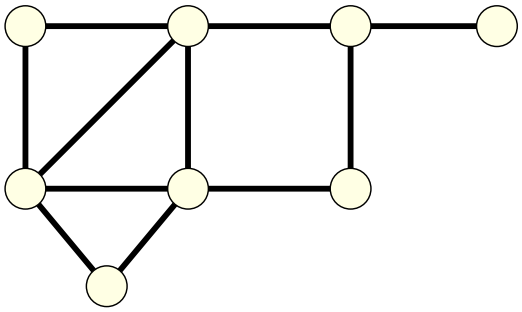


**Fig. 6 – The depicted connected planar graph has 8 vertices, 5 faces (including the outer face), and thus $8 + 5 - 2 = 11$ edges.**

It can be shown (but this is non-trivial) that for any maximum cardinality matching $M$ there is an odd-set cover $OSC$ with

$$|M| = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor, \tag{1}$$

thus proving the optimality of $M$. In such a cover all $n_i$ with $i \geq 2$ are odd, hence the name.

The *certifying algorithm for maximum cardinality matching* returns a matching $M$ and an odd-set cover $OSC$ such that (1) holds. By the argument above, the odd-set cover proves the optimality of the matching. Observe, that is it *not* necessary to understand why odd-set covers proving optimality exist. It is only required to understand the simple proof of Theorem 1, showing that Eq. (1) proves optimality. Also, a correct program which checks whether a set of edges is a matching and a node labeling is an odd-set cover which together satisfy (1) is easy to write.

### 2.6. *Case study: the LEDA planar embedding package*

A *planar embedding* of an undirected graph $G$ is a drawing of the graph in the plane such that no two edges of $G$ cross and no edge crosses over a vertex. See Fig. 6 for an example. A graph is *planar* if it is possible to embed it in the plane in this way. Planar graphs were among the first classes of graphs that were studied extensively.

A *face* of a planar embedding is a connected region of the plane that remains when points on the embedding are removed. Let $n$ be the number of vertices, $m$ the number of edges, and let $f$ be the number of faces of a planar embedding. Euler gave what is known as *Euler's formula* for a connected planar embedded graph: $n + f = m + 2$ (see Fig. 6).

The proof of this is frequently used as undergraduate exercise on induction on the number of edges. As a base case, $m = n - 1$, the graph is a tree, and an embedding of a tree always has exactly one face. The formula holds. For the induction step, suppose $G$ is a planar graph with $m > n - 1$ and the formula holds for all planar embeddings of connected graphs with fewer than $m$ edges. Since $m > n-1$, $G$ has a cycle. Removal of an edge of the cycle in any planar embedding of $G$ causes two faces to merge, leaving $n' = n$ vertices, $f' = f - 1$ faces, and $e' = e - 1$ edges. By the induction hypothesis, it holds that $n' + f' = e' + 2$, which implies $n + f = e + 2$, and the formula holds for the original planar embedding of $G$.

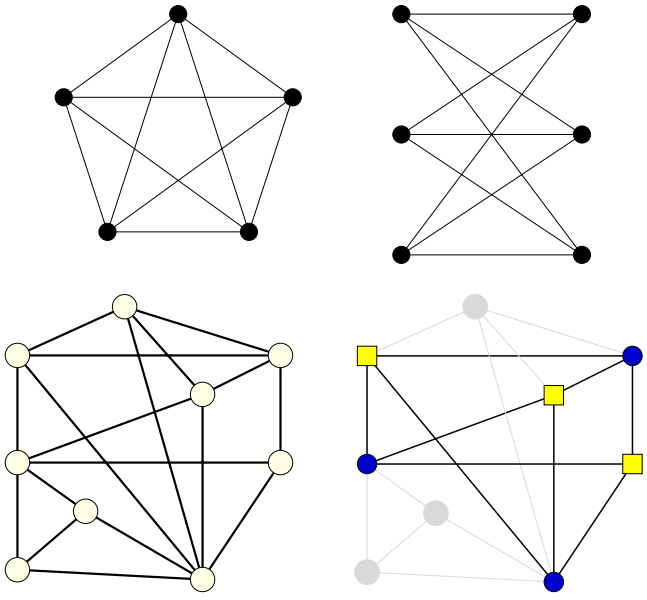


**Fig. 7 – Every non-planar graph contains a subdivision of one of the depicted Kuratowski graphs $K_5$ and $K_{3,3}$ as a subgraph. The lower part of the figure shows a non-planar graph. Non-planarity is witnessed by a $K_{3,3}$.**

In the early 1900's, there was an extensive effort to give a *characterization* of those graphs that are planar. In 1920, Kuratowski gave what remains one of the most famous theorems in graph theory: a graph is planar if and only if it has no *subdivision* of a $K_5$ or a $K_{3,3}$ as a subgraph (see Fig. 7). The $K_5$ is the complete graph on five vertices, the $K_{3,3}$ is the complete bipartite graph with three vertices in each bipartition class, and a subdivision of a graph is what is obtained by repeatedly subdividing edges by inserting vertices of degree two on them.

The planarity test in the Library of Efficient Data Structures and Algorithms (LEDA), a popular package of implementations of many combinatorial and geometric algorithms [20,5], played a crucial role in the development of certifying algorithms and the development of LEDA.

There are several linear time algorithms for planarity testing [21–23]. An implementation of the Hopcroft and Tarjan algorithm was added to LEDA in 1991. The implementation had been tested on a small number of graphs. In 1993, a researcher sent Mehlhorn and Näher a graph together with a planar drawing of it, and pointed out that the program declared the graph non-planar. It took Mehlhorn and Näher some days to discover the bug.

More importantly, they realized that a complex question of the form "is this graph planar" deserves more than a yes–no answer. They adopted the thesis that

*a program should justify (prove) its answers in a way that is easily checked by the user of the program.*

What does this mean for the planarity test?

If a graph is declared planar, a proof should be given in the form of a planar drawing or an embedding, which the program already did. If a graph is non-planar, the program should not just declare this; it should supply a proof of this. The existence of an obvious proof is known by Kuratowski's theorem: it suffices to point out a Kuratowski subgraph.

Linear time algorithms [24,25] for finding Kuratowski subgraphs were known in 1993; unfortunately, Mehlhorn and Näher found the algorithms too complicated to implement. There is however a trivial quadratic time algorithm which is very simple to implement.

```
for all edges e of G do
  if G \ e is non-planar then
    remove e from G;
  end if
end for
```

The output of this algorithm is a subgraph of the original graph. The subgraph is non-planar, since an edge is only removed if non-planarity is preserved, and removal of any edge of it, makes it planar. Thus the subgraph is a minimal (with respect to edge inclusion) non-planar subgraph and hence a subdivision of a Kuratowski graph.

Therefore, in 1994, they had a quadratic time certifying algorithm for planarity and a linear time non-certifying one. They later developed their own certifying linear time algorithm for finding Kuratowski subgraphs [20, Section 8.7].

Note that the proof of Kuratowski's theorem, which is that a subdivision of a $K_5$ or a $K_{3,3}$ always exists in a non-planar graph, is irrelevant to the protocol for convincing a user that a graph is non-planar. All that is needed is for the user to understand the following proof:

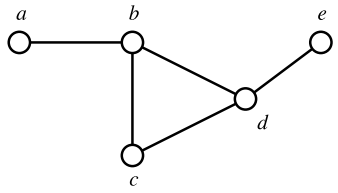
**Lemma 2.** *A graph that contains a subdivision of $K_5$ or $K_{3,3}$ is non-planar.*

**Proof.** Subdividing an edge cannot make a non-planar graph planar, so it suffices to show that $K_5$ and $K_{3,3}$ are non-planar. Suppose there is a planar embedding of $K_5$. Each face touches at least three edges and each edge touches at most two faces. The $K_5$ has ten edges, so the number of faces is at most $\lfloor(2/3)\cdot 10\rfloor = 6$. By Euler's formula, $5 + 6 \geq 10 + 2$, a contradiction. Similarly, suppose $K_{3,3}$ has a planar embedding. Since the graph is bipartite, the cycle around each face must be even, so each face touches at least four edges and each edge once again touches at most two faces. The number of faces is at most $\lfloor(2/4)9\rfloor = 4$. By Euler's formula we obtain $6 + 4 \geq 9 + 2$, a contradiction. □

Let us now examine the checker. In the case of non-planarity, no matter how the Kuratowski subgraph is found, a convenient certificate is obtained by returning it as sequences of edges. A $K_5$ has ten edges, so if the Kuratowski subgraph is a subdivision of a $K_5$, it consists of ten disjoint paths sharing five ending vertices. The ending vertices are listed, and the ten paths are each given as a sequence of edges, in order. The checker verifies that for each pair of ending vertices there is a path with these vertices as beginning and end, cycling through the paths, checking and marking vertices along the way, to make sure that the paths are disjoint except at their endpoints. The check of a $K_{3,3}$ is similar.

Let us turn to witnesses of planarity. The obvious witness is a planar drawing. There is a drawback of using a planar drawing as the witness: It is non-trivial to check in linear time whether a drawing is actually planar.

A less obvious form of witness is a *combinatorial planar embedding*, see Fig. 8. We first explain, what a *combinatorial embedding* is. A combinatorial embedding uses two *twin*




| vertex | counterclockwise cyclic adjacency list |
|---|---|
| $a$ | $\langle b \rangle$ |
| $b$ | $\langle a, c, d \rangle$ |
| $c$ | $\langle d, b \rangle$ |
| $d$ | $\langle b, c, e \rangle$ |
| $e$ | $\langle d \rangle$ |

**Fig. 8 – A planar graph and the corresponding planar combinatorial embedding: for each vertex, the incident edges are listed in counterclockwise order. The boundary cycles are $(a, b)$, $(b, d)$, $(d, e)$, $(e, d)$, $(d, c)$, $(c, b)$, $(b, a)$ and $(c, d)$, $(d, b)$, $(b, c)$. Observe that the successor of $(d, e)$ is determined as follows: go to the twin $(e, d)$ and then to the next edge in cyclic order, i.e., $(e, d)$.**

directed edges $(u, v)$ and $(v, u)$ for each undirected edge of the graph. These twins have pointers to each other. For each vertex $u$ of $G$, the edges incident to it are arranged in a circular linked list $L(u)$. A combinatorial embedding is *planar* if there is a planar embedding of $G$ in which for each vertex $u$ the counterclockwise order of the edges incident to $u$ agrees with the cyclic order in $L(u)$.

How does one verify whether a combinatorial embedding is planar? One simply checks Euler's relation. A combinatorial embedding gives rise to a decomposition of the directed edges into a collection of cycles, called *boundary cycles*. The boundary cycle containing a particular directed edge $(u, v)$ is defined as follows. Let $(u, v)$ be a directed edge with twin $(v, u)$ and let $(v, u')$ be the directed edge after $(v, u)$ in the circular list $L(v)$. Then $(v, u')$ is the next edge in the boundary cycle containing $(u, v)$, cf. Fig. 8. We continue in this way until we are back at $(u, v)$. Why are we guaranteed to come back to where we started? To establish this, it suffices to note that the edge preceding $(u, v)$ in the boundary cycle is also uniquely defined. Let $(u, v')$ be the edge before $(u, v')$ in $L(u)$ and let $(v', u)$ be the twin of $(u, v')$. Then $(v', u)$ precedes $(u, v)$ in the boundary cycle containing it.

**Lemma 3.** *Let $G$ be a connected graph with $n > 1$ vertices, and $m$ edges. Then a combinatorial embedding of $G$ with $f$ boundary cycles is planar if and only if $n + f = m + 2$.*

**Proof.** As before, we argue by induction on the number of edges. As a base case, $m = n - 1$ and the graph is a tree. Any combinatorial embedding of a tree is planar and gives rise to a single boundary cycle; hence $f = 1$ and $n + f = m + 2$.

For the induction step, let $I$ be a combinatorial embedding of a connected graph $G$ with $m > n - 1$ edges, and we assume by induction that the lemma applies to all combinatorial embeddings of graphs with $m - 1$ edges. If $f = 1$, then the embedding cannot be planar, since $G$ has a single boundary cycle and any planar drawing has more than one face in accordance with the fact that $n + f = m + 2$ does not hold.

Suppose $f \geq 2$. We claim that there must be an edge $e = \{(u, v), (v, u)\}$ such that $(u, v)$ and $(v, u)$ lie on different boundary cycles; otherwise, the edges in cyclic order around each vertex are forced to be in the same boundary cycle, and since $G$ is connected, there is only one boundary cycle, a contradiction. Let $G' = G - e$ and let $I'$ be the combinatorial embedding of $G'$ obtained by removing $e$ from $I$. This merges two boundary cycles into one, call it $C$, so $I'$ has $f' = f - 1$ boundary cycles, $m' = m - 1$ edges, and $n' = n$ vertices. By basic algebra, $I'$ satisfies the formula if and only if $I$ does. By the induction hypothesis $I'$, hence $I$, satisfies the formula if and only if $I'$ is planar. If $I'$ is not planar, then neither is $I$, since $I'$ is a subembedding. If $I'$ is planar, then $C$ is the boundary of a face, say $F$, in some planar drawing $D'$ of $G'$ corresponding to $I'$. Since $u$ and $v$ are on $C$, $F$ can be subdivided in $D'$ by drawing the edge $\{u, v\}$ so that it is internal to $F$ except at its endpoints. This yields a planar drawing $D$ corresponding to $I$, so $I$ is planar. □

So in the case of a planarity a convenient witness is a combinatorial embedding of $G$. The checker determines the number $f$ of boundary cycles of the combinatorial embedding and accepts the combinatorial embedding if $n + f = m + 2$.

The example illustrates that the reason why a certificate always exists (the proof of Kuratowski's theorem in this case) is irrelevant to the protocol for convincing a user that a particular output is correct. All that is needed is for the user to understand the proofs of Lemmas 2 and 3.

What does the case study performed in this section illustrate about the current state of algorithm design and software engineering? The algorithm on which the program was based is well-known to be correct. Obviously, the programmer made a mistake in implementing it. However, another problem was that users were willing to accept a declaration that a graph was non-planar, based partly on the knowledge that it was based on an algorithm that is well-known to be correct.

Additionally, another problem was that the designers of algorithms traditionally consider their work done when they have produced a proof that their algorithm never produces an incorrect output. It has gone unrecognized in the algorithm-design community that, in view of the implementation obstacles, algorithm design should also assist the process of obtaining a correct implementation.

## 3. Examples of program failures

To emphasize the importance of methods that facilitate reliability of computation, we give some examples of failures in algorithmic software. The failures are either bugs or due to the use of approximate arithmetic. The first kind of failure shows that even in prevalent, widely distributed programs software bugs have historically appeared. Of course, once bugs become known, they are repaired and hence the bugs reported here do no longer exist in the current version of the programs.

*Planarity test in LEDA 2.0:* As extensively illustrated in Section 2.6, the planarity test in LEDA 2.0 was incorrect. It declared some planar graphs non-planar. Mehlhorn and Näher corrected the error in LEDA 2.1 and made the planarity test certifying. This was their first use of a certifying algorithm. However, the first solution was far from satisfying, as the running time of the certifying algorithm was quadratic compared to linear time for the non-certifying algorithm. It took some time to develop an equally efficient certifying algorithm.

*Triconnectivity of graphs:* A connected graph is triconnected if the removal of any pair of vertices does not disconnect it. There are linear time algorithms for deciding triconnectivity of graphs [26,27]. Gutwenger and Mutzel [28] implemented the former algorithm and reported that some non-triconnected graphs are declared triconnected by it. They provided a correction. We come back to this problem in Section 9.8.

*Constrained optimization in mathematica 4.2:* Version 4.2 of Mathematica (a software environment for mathematical computation) fails to solve a small integer linear program; the example is so small that we can give the full input and output. The first problem asks to compute the minimum of the function $x$ subject to the constraints $x = 1$ and $x = 2$. The system returns the optimal value is 2 and that the substitution $x \to 2$ attains it. The second problems asks to maximize the function under the same constraints. The system answers that the optimal value is infinite and that $x$ has an indeterminate value in this solution.

```
In[1] := ConstrainedMin[ x , {x==1,x==2} , {x} ]
Out[1] = {2, {x->2}}

In[1] := ConstrainedMax[ x , {x==1,x==2} , {x} ]
ConstrainedMax::"lpsub": "The problem is unbounded."
Out[2] = {Infinity, {x -> Indeterminate}}
```

*Pentium bug:* A version of the Pentium chip contained an error in the hardware for division [11]. This hardware error was caused by an incomplete lookup table.

*Linear programming:* A linear optimization problem is the task to maximize or minimize a linear objective function subject to a set of linear constraints. In mathematical language,

$$\max c^T x \quad \text{subject to } Ax \leq b \text{ and } x \geq 0$$

where $x$ is an $n$-vector of real variables, $c$ is a real $n$-vector, $A$ is an $m \times n$ matrix of reals and $b$ is a real $m$-vector. A large number of important problems can be formulated as linear programs, see Section 8.2 for some examples. In applications of linear programming, the entries of $c$, $A$ and $b$ are rational. Linear programming solvers such as CPLEX or SOPLEX use floating point arithmetic for their internal computations and hence are susceptible to round-off errors. The solvers do not claim to find the optimal solution for all problem instances. Of course, a user would like to know whether the solution for his/her particular instance is correct. The papers [29,30], see also Section 8.2, discuss how solutions to linear programs can be verified. They also give examples of instances for which popular solvers such as CPLEX and SOPLEX fail.

*Geometric software:* Geometric software is a particular challenge for correctness. Algorithms are formulated for a Real RAM, i.e., a machine that can compute with real numbers, but programs run on machines that offer only floating point and integer arithmetic on bounded length numbers. The gap is hard to bridge; see [31] for some illustrative examples of how geometric programs can fail. Reliable geometric computing is now an active area of research [32–34]. Section 9.1 discusses certification of geometric software.

Needless to say, most bugs in current software are unknown. In fact it is quite natural to assume that no large software library is flawless. Certifying algorithms may detect these flaws, but more importantly they assure us that the currently given answer is correct. Before we discuss further advantages of certifying algorithms in Section 7, we discuss the relation to extant work and concretize our definition of certifying algorithms.

## 4. Relation to extant work

Program correctness is one of the major problems in software engineering. The theoretically most satisfying approach to program correctness is formal verification. The program is formulated in a language with well-defined semantics, pre- and postcondition are formulated in some formal system, and the proof of correctness is carried out in this formal system (and with the help of a theorem prover). Work on program verification started in the late '60s [12,13, 35] and tremendous progress was made since then [16]. Unfortunately, the verification of complex algorithms written in popular programming languages such as C, C++ or Java is still beyond the state of the art.

Program testing is the most widely used approach to program correctness, see [10] for a recent account. A list of correct input/output pairs $(x_i, y_i)$ is maintained. A program is accepted if, for each $x_i$ on the list, it produces the corresponding $y_i$. The common objections against testing are that it can prove the presence of errors, but never their absence, and that a program can only be tested on inputs for which the correct output is already known by other means.

It has been known since the 1950's that linear programming duality [36,37] provides a method for checking the result of linear programs. In a pair of solutions $(x, \bar{x})$, one for a linear program and one for its dual, both solutions are optimal if and only if they have the same objective value, see Section 8.2. A special case of linear programming duality is the max-flow-min-cut theorem for network flows.

Sullivan and Masson [1,2,38–41] introduced the concept of a certification trail for checking. The idea is that a program leaves a trail of information that can be used to check whether it worked correctly. They applied the idea mainly to data structures. In later work [17], they combined certification trails with formal verification. We discuss checking of data structures in Section 12. Certification trails catch errors ultimately, but not immediately.

Blum and Kannan [42,3] started a theory of program checking; see [43–46,11] for follow-up work. Given a program $P$ allegedly computing a function[2] $f$, how can one check whether $P(x_0) = f(x_0)$ for a particular input $x_0$. A checker $C$ is a probabilistic expected-polynomial-time oracle machine (it uses $P$ as a subroutine) with the property: if $P(x) = f(x)$ for all $x$, then $C$ on input $x_0$ accepts with high probability. If $P(x_0) \neq f(x_0)$, $C$ on input $x_0$ rejects with high probability. If $P(x_0) = f(x_0)$, but $P \neq f$, the checker may accept or reject. They describe, among others, checkers for graph isomorphism, the extended gcd, and sorting reals. In [45] the concept is generalized to self-correcting algorithms. For example, assume *add* is a function that correctly adds for most pairs

[2] They consider only programs with trivial preconditions.

of inputs. Then $add(x, y) = add(add(x, r), add(y, -r))$, where $r$ is a random number, correctly adds all pairs with nonzero probability, since the concrete addition $x + y$ is replaced by three random additions.

There are essential differences between certifying algorithms and the work by Blum et al. First, they mention but do not explore that adding additional output (= our witness) may ease the life of the checker; they give the extended gcd and maximum flow as examples. In contrast, we insist that certifying programs return witnesses that prove the correctness of their output. In exceptional cases, the witness may be trivial. The second essential difference is that they allow checkers to be arbitrarily complex programs (as long as they are polynomial-time) and nevertheless assume checkers to be correct. In contrast, we insist that checkers are simple program and assume that only the simplest programming tasks can be done without error; see Section 6.1.

Our approach has already shown its usefulness in the LEDA project [47]. At the time of this writing, LEDA contains checkers for all network and matching algorithms (mostly based on linear programming duality), for various connectivity problems on graphs, for planarity testing, for priority queues, and for the basic geometric algorithms (convex hulls, Delaunay diagrams, and Voronoi diagrams). Program checking has greatly increased the reliability of the implementations in LEDA. There are many other problems for which certifying algorithms are known. We review some of them in the sections to come and give a guide to the literature in Section 9.8.

Proof-carrying code [48,49] is a "mechanism by which a host system can determine with certainty that it is safe to execute a program supplied by an untrusted source. For this to be possible, the untrusted code producer must supply with the code a safety proof that attests to the code's adherence to a previously defined safety policy. The host can then easily and quickly validate the proof (quote from [49])". We will use methods akin to proof-carrying code in Section 5.

An interactive proof system [50] is a protocol that enables a verifier to be convinced by a prover of some output via a series of message exchanges. In the language of certifying algorithms, their execution needs a bidirectional communication between the checker and the certifying algorithm. However, to be in accordance with the requirements we pose onto certifying algorithms, further simplicity constraints have to be introduced. If done so, they constitute an extension of certifying algorithms that carries several, but not all of the advantages of certifying algorithms mentioned in Section 7.

## 5. Definitions and formal framework

We consider algorithms taking an input from a set $X$ and producing an output in a set $Y$. The input $x \in X$ is supposed to satisfy a precondition[3] $\varphi(x)$ and the input together with the output $y \in Y$ is supposed to satisfy a postcondition $\psi(x, y)$. Here $\varphi : X \to \{T, F\}$ and $\psi : X \times Y \to \{T, F\}$. We call the pair $(\varphi, \psi)$ an I/O-*specification* or an I/O-*behavior*.

In the case of graph bipartition for example, we have $Y = \{bipartite, nonbipartite\}$. With respect to $X$, we can take different standpoints. As an algorithm designer or a programmer using a strongly typed programming language, we could take $X$ as the set of all finite undirected graphs. Then $\varphi(x) = T$ for all $x \in X$ and $\psi(x, y) = T$ iff

$x$ is a bipartite graph and $y = bipartite$ or

$x$ is a nonbipartite graph and $y = nonbipartite$.

As a programmer using Turing machines or an untyped programming language, we would take $X$ as the set of all conceivable inputs, say $\Sigma^*$ in the case of Turing machines or all memory states in the case of the untyped programming language. Then $\varphi(x) = T$ iff $x$ is the well-formed representation of an undirected graph and $\psi(x, y) = T$ iff the precondition is true and

$x$ represents a bipartite graph and $y = bipartite$ or

$x$ represents a nonbipartite graph and $y = nonbipartite$.

In all examples considered in this paper, it is easy to check whether representations are well-formed. We can therefore safely ignore the issue of input representation in most of our discussions.

We specifically allow the possibility that a pair $(x, y)$ of input and output satisfies the postcondition, even though $x$ does not satisfy the precondition. We will next define three kinds of certifying algorithms: strongly certifying, certifying, and weakly certifying algorithms. In the case of a trivial precondition, i.e., $\varphi(x) = T$ for all $x$, the three notions coincide.

### 5.1. *Strongly certifying algorithms*

Strongly certifying algorithms are the most desirable kind of algorithm. On every input, a strongly certifying algorithm proves to its users that it worked correctly or that the user provided it with an illegal input; it also says which of the two alternative holds. More precisely, for any input $x$, it either produces a witness showing that $x$ does not satisfy the precondition or it produces an output $y$ and a witness showing that the pair $(x, y)$ satisfies the postcondition. For technical reasons, in the first case, we want the algorithm to also produce an answering output, in addition to the witness. We could have it return an arbitrary output but find it more natural to extend the output set $Y$ by a special symbol $\perp$ and use $\perp$ as an indicator for a violated precondition. We call $Y^\perp := Y \cup \{\perp\}$ the *extended output set*. We use $W$ to denote the set of witnesses.

A *strong witness predicate* for an I/O-specification $(\varphi, \psi)$ is a predicate $\mathcal{W} : X \times Y^\perp \times W \to \{T, F\}$ with the following properties:

Strong witness property: Let $(x, y, w) \in X \times Y^\perp \times W$ satisfy the witness predicate. If $y = \perp$, $w$ proves that $x$ does not satisfy the precondition and if $y \in Y$, $w$ proves that $(x, y)$ satisfies the postcondition, i.e.,

$$\forall x, y, w \quad \begin{array}{l} (y = \perp \wedge \mathcal{W}(x, y, w)) \Longrightarrow \neg\varphi(x) \quad \text{and} \\ (y \in Y \wedge \mathcal{W}(x, y, w)) \Longrightarrow \psi(x, y). \end{array} \tag{2}$$

[3] For a predicate $P : X \to \{T, F\}$, we use "$x$ satisfies $P$" or $P(x) = T$ interchangeably. Similarly, we use $P(x) = F$ and $x$ does not satisfy $P$ interchangeably. In formulae we write $P(x)$ for $P(x) = T$ and $\neg P(x)$ for $P(x) = F$.

Checkability: For a triple $(x, y, w)$ it is trivial to determine the value $\mathcal{W}(x, y, w)$.
Simplicity: The implications (2) have a simple proof.

For the moment, we want to leave the checkability and the simplicity property informal notions. We invite the reader to check our examples against his/her notion of simplicity and checkability. We discuss this further in Section 5.5.

A *strongly certifying algorithm* for I/O-specification $(\varphi, \psi)$ and strong witness predicate $\mathcal{W}$ is an algorithm with the following properties:

- It halts for all inputs $x \in X$.
- On input $x \in X$ it outputs a $y \in Y^{\perp}$ and a $w \in W$ such that $\mathcal{W}(x, y, w)$ holds.

We illustrate the definition with two examples. The first example is the test for bipartiteness already used in the introduction. Here, $X$ is the set of all undirected graphs and the precondition is trivial. Any graph is a good input. The output set is $Y = \{\text{YES}, \text{NO}\}$. If the output is YES, the witness is a two-coloring, if the output is NO, the witness is an odd cycle.

Next, we give an example, where the precondition is non-trivial. We describe a strongly certifying algorithm that five-colors any planar graph. The algorithm does not decide planarity. So $X$ is the set of undirected graphs. We use $G$ to denote an input graph. The precondition is that $G$ is planar. For a planar graph $G = (V, E)$, the algorithm is supposed to color the vertices of $G$ with five colors such that any two adjacent vertices have distinct colors, i.e., the algorithm constructs a mapping $c : V \to \{1, 2, 3, 4, 5\}$ such that for any edge $e = (u, v) \in E$, $c(u) \neq c(v)$. For a non-planar graph, the algorithm will either prove non-planarity or provide a five-coloring. Non-planarity is proven by exhibiting a sequence of planarity preserving transformations that transform the input graph to a graph that is clearly non-planar. Consider the following recursive algorithm. If $G$ has at most five vertices, the algorithm returns a coloring. So assume that $G$ has more than five vertices. If $G$ has more than $3n - 6$ edges, the algorithm declares the graph non-planar. The witness is the number of edges. If $G$ has at most $3n - 6$ edges, $G$ must have a vertex of degree five or less. Let $v$ be such a vertex. If $v$ has degree four or less, the algorithm removes $v$ from the graph and calls itself on $G - v$. Clearly, removal of a vertex preserves planarity. If the recursive call returns a five-coloring, it is easily extended to $G$. If $v$ has degree five, consider the neighbors of $v$. If the neighbors of $v$ form a $K_5$, return it as a witness for non-planarity. Otherwise, there must be two neighbors, say $x$ and $y$, that are non-adjacent. Remove $v$ from the graph, identify $x$ and $y$, and remove any parallel edges this may create. It is crucial to observe here that removal of $v$ and identification of $x$ and $y$, does not destroy planarity. This is easy to see by conceptualizing a planar drawing and performing the operation there. If the recursive call returns a five-coloring, it is easy to extend it to $G$. We use for $x$ and $y$ the color that was given to their contraction in the smaller graph and we give a color that was not used on the neighbors of $v$ for $v$. In this way, we either find a coloring of the input graph $G$ or a sequence of planarity preserving reductions to a graph $G'$ that is clearly non-planar; either because it has too many edges or because it contains a $K_5$. For any planar graph, the algorithm will produce a five-coloring. It also will produce five-colorings of some non-planar graphs. For example, it will color a $K_5$. If the algorithm fails to find a five-coloring, it produces a witness that the input graph is non-planar. Note that the algorithm does not decide whether the precondition (that the graph is planar) was met.

### 5.2. *Certifying algorithms*

In some situations, we have to settle for less. The algorithm will only prove that either the precondition is violated or the postcondition is satisfied. It will, in general, not be able to also indicate which of the alternatives holds.

As an example, consider binary search. Its input is a number $z$ and an array $A[1..n]$ of numbers. The precondition states that $A$ is sorted in increasing order. The search outputs YES, if $z$ is stored in the array, and it outputs NO, if $z$ is not stored in the array. In the former case, a witness is an index $i$ such that $A[i] = z$, in the latter case, a witness is an index $i$ such that $A[i-1] < z < A[i]$; here $A[0] = -\infty$ and $A[n+1] = +\infty$ for convenience. Binary search maintains two indices $\ell$ and $r$ with $A[\ell] < z < A[r]$ and $\ell < r$. Initially, $\ell = 0$ and $r = n + 1$. As long as $r > \ell + 1$, it compares $z$ with $A[m]$, where $m = \lfloor (r + \ell)/2 \rfloor$. If $z = A[m]$, the algorithms stops. If $z < A[m]$, $r$ is set to $m$ and the algorithms continues. If $z > A[m]$, $\ell$ is set to $m$ and the algorithm continues. Observe that binary search does not discover any violation of its precondition. In fact discovering all violations would require linear time in general. This example leads us to the definition of an (ordinary) certifying algorithm:

A *witness predicate* for an I/O-specification $(\varphi, \psi)$ is a predicate $\mathcal{W} : X \times Y^{\perp} \times W \to \{T, F\}$ with the following properties:

Witness property: Let $(x, y, w)$ satisfy the witness predicate. If $y = \perp$, $w$ proves that $x$ does not satisfy the precondition. If $y \in Y$, $w$ proves that $x$ does not satisfy the precondition or $(x, y)$ satisfies the postcondition, i.e.,

$$\forall x, y, w \quad \begin{array}{l} (y = \perp \wedge \mathcal{W}(x, y, w)) \Longrightarrow \neg\varphi(x) \quad \text{and} \\ (y \in Y \wedge \mathcal{W}(x, y, w)) \Longrightarrow \neg\varphi(x) \vee \psi(x, y). \end{array} \tag{3}$$

The second implication may also be written as $y \in Y \wedge \varphi(x) \wedge \mathcal{W}(x, y, w) \Longrightarrow \psi(x, y)$.

Checkability: For a triple $(x, y, w)$ it is trivial to determine the value $\mathcal{W}(x, y, w)$.
Simplicity: The implications (3) have a simple proof.

In the case of binary search, $X$ is the set of pairs $(A, z)$ where $A$ is an array of numbers and $z$ is a number, $Y$ is $\{\text{YES}, \text{NO}\}$ and $W$ is $\{0..n\}$. For $(A, z) \in X$, $y \in Y^{\perp}$ and $w \in W$, we have

$$\mathcal{W}(x, y, w) = T \text{ iff } \begin{cases} y = \text{YES} \wedge w = i \in \{1..n\} \wedge z = A[i] \text{ or} \\ y = \text{NO} \wedge w = i \in \{0..n\} \wedge A[i] < z < A[i+1]. \end{cases}$$

A *certifying algorithm* for I/O-specification $(\varphi, \psi)$ and witness predicate $\mathcal{W}$ is an algorithm $P$ with the following properties:

- It halts for all inputs $x \in X$.
- On input $x \in X$ it outputs a $y \in Y^{\perp}$ and a $w \in W$ such that $\mathcal{W}(x, y, w)$ holds.

### 5.3. *Weakly certifying algorithms*

Sometimes, we have to settle for even less. A *weakly certifying algorithm* for I/O-specification $(\varphi, \psi)$ and witness predicate $\mathcal{W}$ is an algorithm with the following properties:

- It halts for all inputs $x \in X$ satisfying the precondition. For inputs not satisfying the precondition, it may or may not halt.
- If it halts on input $x \in X$, it outputs a $y \in Y^{\perp}$ and a $w \in W$ such that $\mathcal{W}(x, y, w)$ holds.

As an example, consider a naive randomized SAT-solver: It is given a formula x, of which it is supposed to prove satisfiability. In other words our precondition is

$\varphi(x) =$ (x is a satisfiable boolean formula).

It tries random assignments until is has found a satisfying assignment $w$. It then outputs $y =$ YES, together with $w$ as a witness. Checking the satisfiability of $x$ is trivial; $w$ proves it. This algorithm is certifying, however since it does not halt on unsatisfiable input clauses it is only weakly certifying. A more sophisticated example is the random SAT-solver analyzed by Moser [51], which finds satisfying assignments of certain sparse SAT-clauses in polynomial time.

**Theorem 2.** *Let $(\varphi, \psi)$ be an I/O-specification. A certifying decision algorithm for $\varphi$ plus a weakly certifying algorithm for behavior $(\varphi, \psi)$ can be combined to a strongly certifying algorithm for behavior $(\varphi, \psi)$.*

**Proof.** Let $x$ be any input. We first run the certifying decision algorithm for $\varphi$. It returns $y = \varphi(x) \in \{T, F\}$ and a witness $w$ certifying the correctness of the output. If y is $F$, we return $\perp$ and $w$ as a witness for $\neg\varphi(x)$. If y is $T$, we run the weakly certifying algorithm for I/O behavior $(\varphi, \psi)$ on x. Since $\varphi(x)$, the algorithm returns a $y'$ with $\psi(x, y')$ and a witness $w'$ certifying the correctness of the output. We return $y'$ and $w'$. □

### 5.4. *Efficiency*

We call a certifying algorithm $P$ *efficient* if there is an accompanying checker $C$, such that the asymptotic running of both $P$ and $C$ is at most the running time of the best known algorithm satisfying the I/O-specification. All examples we have treated so far are efficient. We now give an example, where no efficient certifying algorithm is known. The 3-connectedness problem asks whether a graph may be disconnected by removing two vertices. Linear time algorithms [26,27] for this problem are known, but none of them is certifying.

Certifying that a graph is not 3-connected is simple, it suffices to provide a set $S$ of vertices, $|S| \leq 2$, such that $G \setminus S$ is not connected. To certify that their removal disconnects the graph we can, for example, use the algorithm that certifies the connected components (see Section 2.2). Thus, we now focus on 3-connected graphs and describe an $O(n^2)$ algorithm that certifies 3-connectivity (a different certifying $O(n^2)$ algorithm can be found in [52]). We omit details on how to find a separating set during the execution of the algorithm, in case the input graph is not 3-connected. As certificate for 3-connectivity we will use a sequence of edge contractions resulting in the $K_4$, the complete graph on 4 vertices. The *contraction* of an edge $e = xy$ of a graph $G$ is the graph $G/e$ obtained by replacing $x$ and $y$ with a single vertex, whose neighbors are $N(x) \cup N(y) \setminus \{x, y\}$. We call an edge $e$ of a 3-connected graph $G$ *contractible* if the contracted graph $G/e$ is 3-connected. A separating pair is a pair of vertices whose removal disconnects the graph.

**Lemma 4.** *Let $e = (x, y)$ be an edge of a simple graph $G$ whose end-vertices have a degree of at least 3. If $G/e$ is 3-connected, then $G$ is 3-connected.*

**Proof.** Since contractions cannot connect a disconnected graph, the original graph $G$ is connected. There are no cut-vertices in $G$, as they would map to cut-vertices in $G/e$.

Any separating pair of $G$ must contain one of the end-vertices of edge $e$. Otherwise the pair is also separating in $G/e$. It cannot contain both $x$ and $y$, otherwise the contracted vertex $xy$ is a cut-vertex in $G/e$. It suffices now to show that $x, u$, with $u \in V(G) \setminus y$ is not a separating pair. Suppose otherwise, then the graph $G - \{x, u\}$ is disconnected, but the graph $G - \{x, y, u\}$ is not. Thus $\{y\}$ is a component of $G - \{x, u\}$. But this is a contradiction since $y$ has degree at least 3 in $G$. □

To certify the 3-connectivity of a graph $G$, it thus suffices to provide a sequence of edges which, when contracted in that order, have endpoints with a degree of a least 3 and whose contraction results in a $K_4$. We call such a sequence a *Tutte sequence*. We now focus on how to find the contraction sequence, given a 3-connected graph.

The $O(n^2)$ algorithm needs three ingredients: First we require the $O(n^2)$ algorithm by Nagamochi and Ibaraki [53] that finds a sparse spanning 3-connected subgraph of $G$ with at most $3n - 6$ edges. Second we require a linear time algorithm for 2-connectivity. Third we require a structure theorem, that shows how to determine a small candidate set of edges among which we find a contractible edge.

**Theorem 3** (*Krisell [54]*). *If no edge incident to a vertex $v$ of a 3-connected graph $G$ is contractible, then $v$ has a least four neighbors of degree 3, which each are incident with two contractible edges.*

Consider now a vertex $v$ of minimal degree in a 3-connected graph. If it has degree three, it cannot have four neighbors of degree three and hence must have an incident contractible edge. If it has degree four or more, it cannot have a neighbor of degree three (because otherwise, its degree would not be minimal) and hence must have an incident contractible edge. Also note that an edge $xy$ in a 3-connected graph is contractible, if $G - \{x, y\}$ is 2-connected.

We explain how to find the first $n/2$ contractions in time $O(n^2)$. By repeating the procedure we obtain an algorithm that has overall a running time of $O(n^2)$.

First use the algorithm by Nagamochi and Ibaraki [53]. The resulting graph has $3n - 6$ edges. Thus while performing the first $n/2$ contractions, there will always be a vertex with degree at most $2 \cdot 2 \cdot 3 = 12$. Choosing a vertex of minimal degree, we obtain a set of at most 12 candidate edges, one of which must be contractible. To test whether an edge $xy$ is contractible, we check whether $G - \{x, y\}$ is 2-connected with some linear time algorithm for 2-connectivity.

**Theorem 4** ([52]). *A Tutte sequence for a 3-connected graph can be found in time $O(n^2)$.*

It remains a challenge to find a linear time certifying algorithm for 3-connectivity of graphs. A linear time certifying algorithm for graphs 3-connectivity of Hamiltonian graphs was recently found [55]; it assumes that the Hamiltonian cycle is part of the input.

*Efficiency and usefulness:* For some problems, e.g., testing bipartiteness, maximum flow, matchings, and min-cost flows, the best known algorithms are certifying and the cost of checking the witness is negligible compared to the cost of computing it. For such programs, it is best to integrate the checker into the program. For other problems, e.g., planarity testing, certification increases running time by a non-negligible multiplicative factor (more than 2 and less than 10). Finally, there are problems, such as triconnectivity, where the best known certifying algorithm has worse asymptotic complexity than the best known non-certifying algorithm. Even for the latter kind of problem, certification is useful for two reasons. First, one can use the certifying version to generate test instances for the non-certifying version, and second, for small instances the slow certifying version may be fast enough.

### 5.5. *Simplicity and checkability*

The definition of a certifying algorithm and its variants involve two non-mathematical terms that we have not made precise: simplicity and checkability. They guarantee that it is "easy to check" whether a witness $w$ shows that an output is correct for a given input. We now elaborate on these terms.

*Checkability:* Given $x$, $y$, and $w$, we require that it is trivial to determine whether $\mathcal{W}(x, y, w)$ holds. We list a number of conceivable "formalizations" of "being trivial to determine".

- There is a decision algorithm for $\mathcal{W}$ that runs in linear time.
- $\mathcal{W}$ has a simple logical structure. For example, we might require that $x$, $y$, and $w$ are strings, that $\mathcal{W}$ is a conjunction of $O(|x| + |y| + |w|)$ elementary predicates and that each predicate tests a simple property of relatively short substrings.
- There is a simple logical system, in which we can decide whether $\mathcal{W}(x, y, w)$ holds.
- The correctness of a program deciding $\mathcal{W}(x, y, w)$ is obvious or can be established by an elementary proof.
- We can formally verify the correctness of the program deciding $\mathcal{W}(x, y, w)$.

Most witness predicates discussed in this article satisfy all definitions above. In Section 6 we further investigate checkers, i.e., programs that determine the value of a witness predicate $\mathcal{W}$.

*Simplicity:* The witness property is easily verified, i.e., the (equivalent) implications

$$\forall x, y, w \quad \mathcal{W}(x, y, w) \to \psi(x, y) \tag{4}$$

$$\forall x, y \quad \neg\psi(x, y) \to \nexists w\ \mathcal{W}(x, y, w) \tag{5}$$

have an elementary proof. Here, we assumed that the precondition is trivial. For the case of a non-trivial precondition, either statement (2) or (3) should have an elementary proof (the former in the case of a strongly certifying algorithm, the latter in the other cases). We find that all witness predicates discussed in this article fulfill the simplicity property.

Observe that we make no assumption about the difficulty of establishing the existence of a witness. In the case of a strongly certifying or certifying algorithm, this would be the statement

$$\forall x\ \exists y, w \quad \mathcal{W}(x, \bot, w) \vee \mathcal{W}(x, y, w).$$

In the case of a weakly certifying algorithm this would be the statement

$$\forall x \quad \varphi(x) \Longrightarrow (\exists y, w\ \mathcal{W}(x, \bot, w) \vee \mathcal{W}(x, y, w)).$$

Indeed, the existence of witnesses is usually a non-trivial mathematical statement, and its computation a non-trivial computational task. For example, it is non-trivial to prove that a non-planar graph necessarily contains a Kuratowski subgraph (Section 2.5) and is non-trivial to prove that a maximum matching can always be certified by an odd-set cover (Section 2.6). Fortunately, a user of a certifying algorithm does not need to understand why a witness exists in all cases. He only needs to convince himself that it is easy to recognize witnesses and that a witness, indeed, certifies the correctness of the algorithm on the given instance.

The "definition" above rules out some obviously undesirable situations:

1. Assume we have a complicated program $P$ for I/O behavior $(\varphi, \psi)$. We could take $w$ as the record of the computation of $P$ on $x$ and define $\mathcal{W}(x, y, w)$ as "$w$ is the computation of $P$ on input $x$ and $y$ is the output of this computation". If $P$ is correct, then $\mathcal{W}$ is a witness predicate. This predicate certainly satisfies the checkability requirement. However, it does not satisfy the simplicity requirement, since a proof of the witness property is tantamount to proving the correctness of $P$.
2. Another extreme is to define $\mathcal{W}(x, y, w)$ as "$\psi(x, y)$". For this predicate simplicity is trivially given. However, deciding $\mathcal{W}$ amounts to a solution of the original problem.

As our definition is not and cannot be made mathematically stringent, whether an algorithm should be accepted as certifying is a matter of taste. However, if we drop our non-mathematical requirement of "easiness to check", we can ask formal questions on the existence of certifying algorithms.

**Question 1.** *Does every computable function have a certifying algorithm.*

A more stringent version of this question asks, in addition, about the resource requirements of the certifying program.

**Question 2.** *Does every program $P$ have an efficient strongly certifying or certifying or weakly certifying counterpart, i.e., a counterpart with essentially the same running time?*

More precisely, let $P$ be a program with I/O behavior $(\varphi, \psi)$. An efficient strongly certifying counterpart would be a program Q and a predicate $\mathcal{W}$ such that

1. $\mathcal{W}$ is a strong witness predicate for $(\varphi, \psi)$.

2. On input $x$, program $Q$ computes a triple $(x, y, w)$ with $\mathcal{W}(x, y, w)$.
3. On input $x$, the resource consumption (time, space) of $Q$ on $x$ is at most a constant factor larger than the resource consumption of $P$.

For an ordinary certifying counterpart, we would replace strong witness predicate by witness predicate in the first condition. For a weakly certifying counterpart, we would additionally replace the second condition by the following: if $Q$ halts on $x$, it computes a triple $(x, y, w)$ with $\mathcal{W}(x, y, w)$, and if $\varphi(x)$ then $Q$ halts on $x$. We address these questions in the next two subsections.

### 5.6. *Deterministic programs with trivial preconditions*

We show that every deterministic program that has a trivial precondition $\varphi(x) = T$ for all $x$ can be made certifying with only a constant overhead in running time. This sounds like a very strong result. It is not really; the argument formalizes the intuition that a formal correctness proof is a witness of correctness. The construction shows some resemblance to proof-carrying code [48].

Let $\psi$ be a postcondition and let $P$ be a program (in a programming language $L$ with well-defined semantics) with I/O behavior $(T, \psi)$. We assume that we have a proof (in some formal system $S$) of the fact that $P$ realizes $(T, \psi)$, i.e., a proof of the statement[4]

$$\forall x \quad P \text{ halts on input } x \text{ and } \psi(x, P(x)). \tag{6}$$

We use $w_2$ to denote the proof.

We extend $P$ to a program $Q$ which on input $x$ outputs $P(x)$ and a witness $w = (w_1, w_2, w_3)$ where $w_1$ is the program text $P$, $w_2$ is as above, and $w_3$ is the computation of $P$ on input $x$.

The witness predicate $\mathcal{W}(x, y, w)$ holds if $w$ has the required format, i.e., $w = (w_1, w_2, w_3)$, where $w_1$ is the program text of some program $P$, $w_2$ is a proof in $S$ that $P$ realizes I/O behavior $(T, \psi)$, $w_3$ is the computation of $P$ on input $x$, and $y$ is the output of $w_3$. The following statements show that algorithm $Q$ is an efficient certifying algorithm:

1. $\mathcal{W}$ is a strong witness predicate for I/O behavior $(T, \psi)$:
   - *Checkability:* The check whether a triple $(x, y, w)$ satisfies the witness predicate is easy. We use a proof checker for the formal system $S$ to verify that $w_2$ is a proof for statement (6). We use an interpreter for the programming language $L$ to verify that $w_3$ is the run of $P$ on input $x$ and that $y = P(x)$.
   - *Strong witness property and simplicity:* The proof of the implication $\mathcal{W}(x, y, w) \Rightarrow \psi(x, y)$ is elementary: Assume $\mathcal{W}(x, y, w)$. Then $w = (w_1, w_2, w_3)$, where $w_1$ is the program text of some program $P$, $w_2$ is a proof (in system $S$) that $P$ realizes I/O behavior $(T, \psi)$, $w_3$ is the computation of $P$ on input $x$, and $y$ is the output of $w_3$. Thus $y = P(x)$ and $\psi(x, P(x))$.
2. For every input $x$ algorithm $Q$ computes a witness $w$ with $\mathcal{W}(x, P(x), w)$. This follows from the definition of $Q$.
3. The running time of $Q$ is asymptotically no larger than the running time of $P$. The same holds true for the space complexity. Observe that $Q$ produces a large output; however, the workspace requirement is the same as for $P$. The same is true for the checker described in the checkability argument.

We summarize the discussion.

**Theorem 5.** *Every deterministic program for I/O-specification $(T, \psi)$ has an efficient strongly certifying counterpart. This assumes that a proof for (6) in a formal system is available.*

We admit that the construction above leaves much to be desired. It is not a practical way of constructing certifying algorithms. After all, certifying programs are an approach to correctness when a formal correctness proof is out of reach. A frequent reaction to Theorems 5 and 7 is that they contradict intuition. In fact, we also started out wanting to prove the opposite. In an attempt to prove that some algorithms cannot be certifying without loss of efficiency, we discovered Theorem 5. We come back to this point in Section 14.

However, the construction is also quite assuring and gives strong moral support. Every deterministic program can be made certifying with only a constant loss in time and space. *So, when searching for a certifying algorithm we only have to try hard enough; we are guaranteed to succeed.* The construction also captures the intuition that certification is no harder than a formal correctness proof of a program.

### 5.7. *Non-trivial preconditions*

For non-trivial preconditions the situation is more subtle. We will see that there are I/O-behaviors that can be realized by a non-certifying algorithm but cannot be realized by a certifying algorithm. However, every algorithm can be made weakly certifying.

*A program without a certifying counterpart:* On an input $x$ not satisfying the precondition, a non-certifying program may do anything. In particular, it may diverge or return nonsense. The requirements for a certifying algorithm are more stringent. On input $x$, it is supposed to either return a $y$ and a witness which proves $\psi(x, y) \vee \neg\varphi(x)$ or output $\perp$ and a witness which proves $\neg\varphi(x)$. We will next show that some I/O-behaviors resist certification.

**Theorem 6.** *Consider the following task. The input consists of two strings $s$ and $x$. The precondition states that the string is the description of a Turing machine which halts on $x$. The postcondition states that the output is the result of running $s$ on $x$. This behavior can be realized algorithmically, however it cannot be realized by a certifying algorithm.*

**Proof.** The behavior is easy to implement, take any universal Turing machine. It is even conceivable to prove the correctness of the implementation. After all, universal Turing machines are quite simple programs.

However, there is no certifying algorithm implementing this behavior. What would a certifying algorithm have to do on input $s$ and $x$? It either outputs $\perp$ and proves that $s$ is not the description of a Turing machine halting on $x$, or

[4] We use $P(x)$ to denote the output of $P$ on input $x$.

it provides an output $y$, and then proves that $s$ is not the description of a Turing machine halting on $x$ or that the output of the Turing machine described by $s$ on $x$ is $y$. By standard diagonalization we can show that such an algorithm (which would essentially have to solve the halting problem) cannot exist: Suppose $H(s, x)$ is a program that always halts, and, whenever $s$ encodes a Turing machine halting on input $x$ with some output, then $H$ outputs the same output. Consider the program $P$, that on input $s'$ calls $H(s', s')$ and outputs something differing from $H(s', s')$. Since $H$ always halts, $P$ always halts. If $p$ is a description of $P$ then $P(p) \neq H(p, p) = P(p)$, a contradiction. □

Note that there exists a weakly certifying algorithm that solves the problem. The reason is that weakly certifying algorithms do not have to halt when the precondition is not fulfilled. In fact every program can be made weakly certifying, as we show next.

*Every program can be made weakly certifying:* We modify the argumentation of Section 5.6. Let $(\varphi, \psi)$ be an I/O-specification and let $P$ be a program (in a programming language $L$ with well-defined semantics) with I/O behavior $(\varphi, \psi)$. We assume that we have a proof (in some formal system $S$) that $P$ realizes $(\varphi, \psi)$, i.e., a proof of the statement

$$\varphi(x) \Longrightarrow P \text{ halts on input } x \text{ and } P(x) \in Y \quad \text{and} \quad \text{if } P \text{ halts on } x \text{ then } \psi(x, P(x)). \tag{7}$$

We use $w_2$ to denote the proof.

We extend $P$ to a program $Q$ which on input $x$ does the following: If $P$ halts on input $x$, $Q$ outputs $P(x)$ and a witness $w = (w_1, w_2, w_3)$ where $w_1$ is the program text $P$, $w_2$ is as above, and $w_3$ is the computation of $P$ on input $x$. This construction is akin to proof-carrying code [48,49].

The witness predicate $\mathcal{W}(x, y, w)$ holds if $w$ has the required format, i.e., $w = (w_1, w_2, w_3)$, where $w_1$ is the program text of some program $P$, $w_2$ is a proof for (7), $w_3$ is the computation of $P$ on input $x$, and $y$ is the output of $w_3$. The following statements show that $Q$ is an efficient weakly certifying algorithm:

1. $\mathcal{W}$ is a witness predicate for I/O behavior $(\varphi, \psi)$:
   - *Checkability:* The check whether a triple $(x, y, w)$ satisfies the witness predicate is again easy. We use a proof checker for the formal system $S$ to verify that $w_2$ is a proof for statement (7). We use an interpreter for the programming language $L$ to verify that $w_3$ is the run of $P$ on input $x$ and that $y = P(x)$. Note that this checker always halts.
   - *Witness property and simplicity:* Assume $\mathcal{W}(x, y, w)$. Then $w = (w_1, w_2, w_3)$, where $w_1$ is the program text of some program $P$, $w_2$ is a proof for (7), $w_3$ is the computation of $P$ on input $x$, and $y$ is the output of $w_3$. Thus $y = P(x)$ and either $\neg\varphi(x)$ or $\psi(x, P(x))$. This proof is elementary.
2. For every input $x$ with $\varphi(x)$, algorithm $Q$ computes a witness $w$ with $\mathcal{W}(x, P(x), w)$. This follows from the definition of $Q$.
3. The argument from Section 5.6 applies.

We summarize the discussion.

**Theorem 7.** *Every deterministic program for* I/O*-specification has an efficient weakly certifying counterpart. This assumes that a proof for (7) in a formal system is available.*

Theorem 5 follows as a corollary by setting $\varphi = T$. The remarks following Theorem 5 apply also to the construction of this paragraph.

### 5.8. *An objection*

Several colleagues suggested to restrict the length of and the computation time required to check the witness, e.g., to a polynomial in the length of the input. Following the suggestion would have the undesirable consequence that only problems in $NP \cap coNP$ could have certifying algorithms.

**Lemma 5.** *Let $f : X \to \{0, 1\}$ and assume that $f$ has a witness predicate $W$ with polynomial size witnesses and that $W$ can be evaluated in polynomial time. Then $f \in NP \cap coNP$.*

**Proof.** We guess the output $y$ and the witness $w$ and compute $W(x, y, w)$. In this way, we obtain a polynomial-time nondeterministic algorithm for the yes- as well as the no-instances of $f$. Thus $f \in NP \cap coNP$. □

There is no reason to restrict certification to problems in $NP \cap coNP$. In fact, certification has been used successfully to verify optimal solutions to $NP$-complete optimization problems; see Section 9.3. Even more so, the concept of certifying algorithms is applicable to the whole spectrum of complexity classes.

## 6. Checkers

A *checker* for a witness predicate $\mathcal{W}$ is an algorithm $C$ that on input $(x, y, w)$ returns $\mathcal{W}(x, y, w)$. Fig. 1 in the introduction compares the I/O behavior of a non-certifying program with a certifying program and demonstrates how a checker is used to ensure that the witness $w$ certifies that $y$ is the correct output for input $x$. When designing checkers, there are several aspects that we are interested in:

1. Correctness: Checkers must be correct.
2. Running time: Ideally, the running time of a checker is linear in the size of its input, i.e., the size of the triple $(x, y, w)$.
3. Logical complexity: Checkers should be simple programs.
4. Required randomness (see Section 10): Checkers may use randomness. Most users will prefer deterministic checkers over randomized checkers.

Correctness is the crucial issue. The concept of certifying algorithm relies on our ability to write correct checkers. What approaches do we have? We may take the pragmatic approach or use formal verification for checkers. We discuss these options next.

### 6.1. *The pragmatic approach*

The pragmatic answer is that the task of checking a witness should be so simple that the question of having a correct checking program is not really an issue. The checking program is short and has a simple logical structure and hence every programmer should be able to get it right. LEDA

**Table 1 – The length (in lines of code (LoC)) of two modules and the corresponding checkers in LEDA. The second line refers to the module that computes maximum matchings in graphs. The module has 280 LoC, the checker has 26 LoC. It verifies that an odd-set cover proves the optimality of a matching in a general graph, see Section 2.5. The third line refers to the planarity test module. It computes combinatorial planar embeddings of planar graphs and exhibits Kuratowski subgraphs in non-planar graphs and has 900 LoC. The corresponding checkers verify that a combinatorial embedding satisfies Euler's relation, see Section 2.6, and that a list of edges in a graph G forms a Kuratowski subgraph. The former checker hat 35 LoC, the latter checker has 95 LoC and is the longest checker in LEDA.**

| Problem | LoC(P) | LoC(C) | Reference |
|---|---|---|---|
| Max Cardinality Matching | 280 | 26 | [20, Section 7.7] |
| Planarity | 900 | 130 | [20, Section 8.7] |

followed the pragmatic approach; Table 1 shows the length (in lines of code) of some checkers in LEDA.

What kind of primitives can one use in a checker? We have to assume that there is a basic layer of correct primitives, e.g., simple data structures such as linear lists, graphs, union-find and simple algorithms such as connectivity of graphs and minimum spanning trees.

We can also use non-trivial primitives as long as they are certifying. Assume that you want to use a function $f$ in a checker $C$ and that you have a certifying algorithm $Q_f$ and a correct checker $C_f$ for it. When you need to compute $f(x')$ in $C$, you call $Q_f$ and obtain $y'$ and a witness $w'$. Then you use $C_f$ to check the triple $(x', y', w')$. If $C_f$ accepts the triple, you know that $y'$ is equal to $f(x')$ and you proceed. If $C_f$ rejects the triple, $C$ gives up and rejects.

The checker could also require that the triple $(x', y', w')$ is provided to as part of the witness. This would simplify the checker and is useful, whenever the checker has to operate under limited resources or when one wants to formally verify the checker.

### 6.2. *Manipulation of the input*

The checker uses the input data. Since a program could tamper with the input, precautions must be taken, to ensure that the witness is checked against an unmanipulated input. This issue is also addressed in Fig. 1 in the introduction, which shows that the checker accesses the original input.

As an example of a common pitfall, observe that in order to check the output of a sorting algorithm, it does not suffice to verify that the output list is sorted. Rather, one also needs to check that the output list contains the same elements as the input list.

Two methods can be applied to resolve the issue with manipulation of the input. In the first method the checker withholds a private copy of the input, to which the certifying algorithm has no access to. When done so, the witness has to be written in a way that allows for it to be checked using the copy of the input.

In the second method the checker prevents (or monitors) alterations of the input. Recall that in the case of binary search (see Section 5.2) the second method has to be applied to prevent dramatic increase of the running time. In the case of sorting, alterations of the input may for example be monitored by using only a trusted version of swap (see Section 11).

### 6.3. *Formal verification of checkers*

Since checkers are so simple, we might be able to prove them correct. There is an obvious objection. Only programs written in a programming language with a formally defined semantics can be proven correct, but most algorithms are written in languages, e.g., C, C++ or Java, whose semantics are only informally defined.

However, there is no need to write the checker in the same language as the program computing the solution and the witness. We may write the checker in a language that supports verification. Since checkers are simple programs, this should not be a big obstacle. Also since the time complexity of the checking task is frequently much smaller than the complexity of computing the solution, the efficiency of the language is not an obstacle.

We next turn to a discussion of the advantages of certifying algorithms.

## 7. Advantages of certifying algorithms

Assume that we have a (strongly, ordinary, weakly) certifying program $P$ for an I/O behavior $(\varphi, \psi)$ and a correct checker $C$. What do we gain?

*Instance correctness:* If the checker accepts $(x, y, w)$, $w$ proves that either $\neg\phi(x)$ or $\psi(x, y)$. In the case of a strongly certifying algorithm, we also know which alternative holds. We are certain that $P$ worked correctly on the instance $x$. We emphasize that we do not know that the program will work correctly for all inputs, we only know it for instance $x$. If the checker rejects $(x, y, w)$, $w$ is not a valid certificate and we know that $P$ erred either in the computation of $y$ or in the computation of the certificate $w$.

*Testing on all inputs:* Testing is the most popular method to ensure correctness of implementations. The implementation is run on inputs for which the correct result is already known and only released if the correct output is produced for all test inputs. The standard objection to testing is, of course, that it can only prove the presence of errors and not the absence of errors. There is also a second objection: *one can only test on inputs for which the answer is already known.* What if your algorithm is the first for a certain task and you know the answer only on a very small set of inputs or if your algorithm is much more efficient than all previous algorithms and hence you can run the other algorithms only for small inputs? Using certifying algorithms, we can now test our program on every input $x$ and not just on inputs for which we already know the right answer by other means. We can even test our program whenever it is executed.

*Confinement of error:* Whenever a certifying program fails on an input, the checker catches the error. In a system consisting of certifying components, errors are caught by the failing

module and do not spread over to other modules. This greatly simplifies the task of finding the source of the error.

*More effective program development:* Program modules are usually incorrect or incomplete during much of their development phase. Confinement of error is particularly important during this phase.

*Trust with minimal intellectual investment:* Which intellectual investment does a user have to make in order to trust a certifying program? Not much. First, he has to understand, why a witness proves the correctness of the program on a particular instance, and second he has to verify that the checker is correct.

*Remote computation:* Certification allows a user to locally (e.g., on a mobile device) verify an answer that has been computed at some distant location (e.g., on a fast remote server), even if the software used for computation is untrusted, or the channel by which the result is transferred is noisy. This allows the usage of untrusted fast remote servers.

*Verified checkers:* As we will see in the examples, checkers are frequently so simple, that their formal verification is feasible. Also, they are frequently so efficient compared to the program itself, that we may be willing to write them in a less efficient programming language which eases verification.

*Black-box programs:* In order to develop trust in a certifying program there is no need to have access to the source text of the program. It suffices to have access to the source of the checker since only its correctness needs to be verified. In other words, the intellectual property can be kept secret and yet trust can be developed.

This argument is somewhat compromised by our theoretical constructions in Section 5. There, code (either in source or in binary form) and correctness proof are part of the witness and cannot be kept secret. Zero-knowledge proofs might allow to overcome this problem.

*Tamperproofness:* Certifying algorithms are resistant to tampering. If a triple $(x, y, w)$ does not pass the witness predicate, the checker rejects it. If it satisfied the witness predicate despite the fact that the program was tampered with, the user receives a correct result and does neither notices nor cares about the tampering.

*Efficiency of checking:* In all examples discussed in this paper, the checker runs in time linear in the size of the triple $(x, y, w)$ and, in the case of algorithms with super-linear running time the length of $w$ is sublinear in the running time of the algorithm.

On the contrary, a program that only returns its answer and nothing else cannot be checked in sub-computing time or space (at least if its answers belongs to some finite set). Otherwise, we would simply present the input with all possible answers to the checker. Exactly one answer will be accepted and so we return it as the answer to the input.

*Certifying algorithms, a challenge for algorithmics:* Most existing algorithms are non-certifying. It is a challenge for algorithmics to find certifying algorithms which are as efficient as the existing non-certifying ones. The design of certificates frequently leads to deeper insight into the problem structure.

*Better programming:* Turning a correct algorithm into a correct program is an error-prone task. An algorithm is the description of a problem solving method intended for human consumption. It is usually described in natural language, pseudo-code, mathematical notation or a mixture thereof. A program is the description of a problem solving method intended for execution by a machine. It is written in a programming language and usually much more detailed than the algorithm. Certifying algorithms are easier to implement correctly than non-certifying algorithms because they can be tested on all inputs.

*Practical experience:* Mehlhorn and Näher adopted the concept of certifying algorithms a design principle for the LEDA [20,47] library of efficient data types and algorithms, the name "certifying algorithm" was coined in [6]. They started to build LEDA in 1989 [4,56]. They implemented conventional algorithms and some checks for assertions and postconditions, and tested extensively. Nevertheless at least two major programs were incorrect when first released: the planarity test, see Sections 2.6 and 3, and the convex hull algorithm in arbitrary dimensional spaces, see Section 9.1. In the attempt to correct the errors in these programs, Mehlhorn and Näher adopted the concept of certifying programs and reported about it in [8,9]. For the LEDA book [20], many algorithms were reimplemented and a large number of the algorithms in LEDA were made certifying, in particular, the graph and geometry algorithms. However, there are still large parts which are non-certifying, e.g., all algorithms working on the number types of LEDA.

*Hidden assumptions:* A checker can only be written if the problem at hand is rigorously defined. Mehlhorn and Näher noticed that some of their specifications in LEDA contained hidden assumptions which were revealed during the design of the checker. For example, an early version of the biconnected components algorithm assumed that the graph contains no isolated nodes.

## 8. General techniques

There are several general techniques, that facilitate the design of certifying algorithms. We start their discussion by considering reductions that preserve witnesses.

### 8.1. *Reduction*

Reduction is a powerful problem solving method. In order to solve a problem, we reduce it to a problem for which we already know a solution. More precisely, we want to solve problem $P$ using an algorithm $A'$ for a problem $P'$ and two transformations. Transformation $f$ translates problem instances of problem $P$ into problem instances of problem $P'$ which are then solved by means of algorithm $A'$. The result of $A'$ is translated back to an output of $P$ by means of a transformation[5] $g$. Thus $f : X \to X'$, $g : X \times Y' \to Y$ and $A(x) = g(x, A'(f(x)))$ is an algorithm for $P$.

[5] This transformation has inputs $y'$, the output of $A'$, and $x$, the instance of $P$ to be solved. The input $x$ is needed so that $y'$ can be interpreted for it.

In this section, we show how to use reductions in the context of certifying algorithms. We will first discuss an example – a reduction of maximum cardinality bipartite matching to maximum flow – and then give a general formulation. The main additional requirement is the availability of a transformation that transforms witnesses for $P'$ into witnesses for $P$.

### 8.1.1. An example

A matching in a graph is a set of edges no two of which share an endpoint. A maximum cardinality matching or maximum matching is a matching of maximum cardinality. A node cover $C$ is a set of nodes covering all edges in $G$, i.e., for each edge of $G$ at least one endpoint is in $C$. The following Lemma is a special case of the discussion in Section 2.5 (Fig. 9).

**Lemma 6.** *Let $G$ be a bipartite graph, $M$ be any matching in $G$, and $C$ be any node cover. Then $|M| \leq |C|$. If $|M| = |C|$, $M$ is a maximum cardinality matching in $G$.*

**Proof.** We define a mapping from $M$ to $C$. For any edge in the matching at least one endpoint must be in the node cover. We can therefore map any edge in the matching to a node in the cover. This mapping is injective, since edges in a matching do not share endpoints. Thus $|M| \leq |C|$. □

A network is a directed graph $G = (V, E)$ with a nonnegative capacity function *cap* defined on the edges and two designated nodes $s$ and $t$. A flow is a function $f$ defined on the edges that observes the capacity constraints, i.e., $0 \leq f(e) \leq cap(e)$ for any edge $e$, and observes the flow conservation constraints, i.e., for any node $v$ different from $s$ and $t$, the flow out of $v$ is the same as the flow into $v$, i.e., $excess(v) = \sum_{e;e=(v,w)} f(e) - \sum_{e;e=(u,v)} f(e) = 0$. The value of the flow is the excess of $s$, i.e., $val(f) = excess(s)$. An $(s,t)$-cut $(S,T)$ is a partition of $V$ into two sets $S$ and $T$ such that $s \in S$, $t \in T$, $V = S \cup T$ and $S \cap T = \emptyset$. The capacity of a cut $(S,T)$ is the total capacity of the edges going from $S$ to $T$, i.e, $cap(S,T) = \sum_{e \in S \times T} cap(e)$.

**Lemma 7.** *Let $G$ be a network with source $s$ and sink $t$, $f$ an $(s,t)$-flow and $(S,T)$ an $(s,t)$-cut. Then $val(f) \leq cap(S,T)$. If $val(f) = cap(S,T)$, then $f$ is a flow of maximum value and $f(e) = cap(e)$ for all $e \in S \times T$ and $f(e) = 0$ for all $e \in T \times S$.*

**Proof.** We have

$$\begin{aligned} val(f) &= excess(s) \\ &= \sum_{v \in S} excess(v) \\ &= \sum_{v \in S} \left( \sum_{e;e=(v,w)} f(e) - \sum_{e;e=(u,v)} f(e) \right) \\ &= \sum_{e \in S \times T} f(e) - \sum_{e \in T \times S} f(e) \\ &\leq cap(S,T). \quad \square \end{aligned}$$

The reduction from maximum bipartite matching to maximum flow is as follows: Let $G = (U \cup W, E)$ be a bipartite graph. We construct an auxiliary graph $G'$ with node set $V = U \cup W \cup \{s,t\}$, where $s$ and $t$ are new nodes. We have edges from $s$ to all nodes in $U$, direct the edges in $E$ from $U$ to $W$, and edges from all nodes in $W$ to $t$. All edges have capacity one.

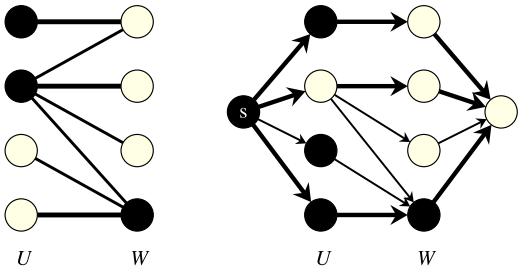



**Fig. 9 – The figure on the left shows a bipartite graph $G$, a maximum matching $M$ (=the heavy edges) and a node cover $C$ (=the filled vertices). The figure on the right shows the corresponding flow network (all edges are directed from left to right and have capacity one), a maximum flow (=the heavy edges), and an $(s,t)$-cut $(S,T)$; $S$ consists of the filled vertices. The maximum flow and the $(s,t)$-cut induce a maximum matching and a node cover. The matching consists of the saturated edges and $C$ consists of the vertices in $T \cap U$ plus the vertices in $S \cap W$ plus the vertices in $U \cap S$ that have an edge to a vertex in $T \cap W$.**

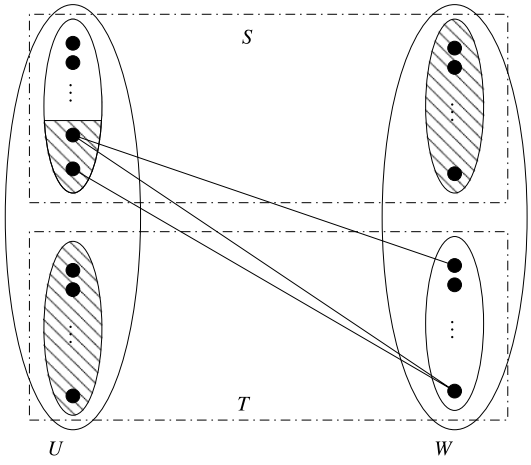



**Fig. 10 – The figure demonstrates how to obtain a node cover from a cut $(S,T)$. The cover contains the nodes in $T \cap U$, $S \cap W$ and the source nodes of the edges in $(S \cap U) \times (T \cap W)$. Every edge has an endpoint in a shaded region, which shows that the set of vertices in the shaded region is a node cover.**

Let $f_0$ be an integral maximum flow. We construct a matching $M_0$ in $G$ from it by putting into $M_0$ precisely the edges in $E$ that carry a flow of one.

**Lemma 8.** *$M_0$ is a matching and $|M_0| = val(f_0)$.*

**Proof.** Since the edges from $s$ to any node in $U$ and from any node in $W$ to $t$ have capacity one and our flow is integral, the flow out of a node in $U$ is at most one and the flow into a node in $W$ is at most one. Thus $M_0$ is a matching and $|M_0| = val(f_0)$. □

We next show how to translate cuts into node covers. Fig. 10 illustrates this construction. Let $(S,T)$ be an $(s,t)$-cut.

We define

$$C := (T \cap U) \cup (S \cap W) \cup \{\, u \mid \text{there is an edge } e = (u, w) \in (S \cap U) \times (T \cap W) \,\}.$$

**Lemma 9.** *If $(S, T)$ is an $(s, t)$-cut, then $C$ is a node cover.*

**Proof.** Let $e = (u, w)$ be any edge of our bipartite graph. If either $u \in T$ or $w \in S$, $e$ is clearly covered. So assume $u \in S$ and $w \in T$. Then $e$ is covered by the third term in the definition of $C$. Thus $C$ is a node cover. □

**Lemma 10.** *Let $(S_0, T_0)$ be an $(s, t)$-cut with $val(f_0) = cap(S_0, T_0)$ and let $M_0$ and $C_0$ be the corresponding matching and node cover. Then $|M_0| = |C_0|$.*

**Proof.** We have

$$\begin{aligned} |M_0| &= val(f_0) \\ &= cap(S_0, T_0) \\ &= |S_0 \cap W| + |T_0 \cap U| + |\{e = (u, w) \in (S_0 \cap U) \times (T_0 \cap W)\}| \\ &\le |S_0 \cap W| + |T_0 \cap U| + |\{\, u \mid \text{there is an edge } e = (u, w) \in (U \cap S_0) \times (W \cap T_0) \,\}| \\ &= |C_0|, \end{aligned}$$

where the second equality follows from the fact that there are $|S_0 \cap W|$ edges $(w, t)$ in $S_0 \times T_0$ and $|T_0 \cap U|$ edges $(s, u)$ in $S_0 \times T_0$ and the inequality follows from the fact that all edges in $S_0 \times T_0$ are saturated and hence counting endpoints in $U$ is equivalent to counting the edges in $(S_0 \cap U) \times (T_0 \cap W)$. □

The example demonstrates how to design a certifying algorithm via a reduction to a different problem, for which a certifying algorithm is already known. After we describe the general approach to reductions, we explain why and how the previous example is a special case.

#### 8.1.2. *The general approach*

We now describe a general approach for obtaining certifying algorithms via reductions.

**Theorem 8.** *Let $(\varphi, \psi)$ and $(\varphi', \psi')$ be I/O-specifications and let $\mathcal{W}$ and $\mathcal{W}'$ be corresponding (strong) witness predicates. Let $A'$ be a (strongly, weakly) certifying algorithm for I/O-specification $(\varphi', \psi')$ and witness predicate $\mathcal{W}'$. The translations $f : X \to X'$, $g : X \times Y' \to Y$, $h : X \times Y' \times W' \to W$ transform inputs, outputs, and witnesses respectively. Assume further*

$$\mathcal{W}'(f(x), y', w') \Longrightarrow \mathcal{W}(x, g(x, y'), h(x, y', w')). \tag{8}$$

*If $A'$ is weakly certifying, also assume*

$$\varphi(x) \Longrightarrow \varphi(f(x)).$$

*Then the following algorithm $A$ is a (strongly, weakly) certifying algorithm for I/O-specification $(\varphi, \psi)$ and witness predicate $\mathcal{W}$.*

1. *translate $x$ into $f(x)$*
2. *run $A'$ on $f(x)$ to obtain $y'$ and $w'$*
3. *translate $y'$ into $y = g(x, y')$ and $w'$ into $w = h(x, y', w')$ and return $(y, w)$.*

**Proof.** Assume first that $\mathcal{W}$ and $\mathcal{W}'$ are (strong) witness predicates and $A'$ is a (strongly) certifying algorithm. Let $x \in X$ be arbitrary and let $x' = f(x)$ be the corresponding input for $A'$. Algorithm $A'$ terminates on $x'$ and returns a pair $(y', w')$ with $\mathcal{W}'(x, y', w')$. Then $\mathcal{W}(x, y, w)$ by implication (8). Thus $A$ is a (strongly) certifying algorithm for I/O-specification $(\varphi, \psi)$ and witness predicate $\mathcal{W}$.

We come to the case that $A'$ is weakly certifying. Let $x \in X$ satisfy $\varphi(x)$. Then $\varphi'(x')$ and hence $A'$ terminates and returns a pair $(y', w')$. Thus $A$ terminates and $\mathcal{W}(x, y, w)$. □

It may seem strange that the implication $\varphi(x) \Longrightarrow \varphi(f(x))$ is only needed in the case of weakly certifying algorithms. Note however, that even in this case, it is only used to conclude that $A'$ terminates on $x'$. Since (strongly) certifying algorithms are total, their termination is given for free. So assume that $f$ translates an input $x$ with $\varphi(x)$ into an $x'$ with $\neg\varphi'(x')$ and $A'$ produces a witness $w'$ that proves that $x'$ violates the precondition. In such a situation, it is unlikely that one can prove the implication $\mathcal{W}'(f(x), y', w') \Longrightarrow \mathcal{W}(x, g(x, y'), h(x, y', w'))$. It is not impossible, e.g., if the I/O-specification $(\varphi, \psi)$ is trivial.

Let us illustrate Theorem 8 on our previous example. We have

- $X$ = the set of all bipartite graphs $G$.
- $X'$ = the set of all directed networks $G'$ with designated nodes $s$ and $t$.
- $f$ translates a bipartite graph $G = (U \cup W, E)$ into a directed network.
- $Y$ = the set of matchings in $G$ (alternatively $Y$ = all subsets of $E$).
- $Y'$ = the set of integral $(s, t)$-flows in $G'$.
- $\psi'(G', f') = T$ iff $f'$ is a maximum flow in $G'$.
- $\psi(G, M) = T$ iff $M$ is a maximum matching in $G$.
- $W'$ = the set of $(s, t)$-cuts.
- $W$ = the set of node covers in $G$.
- $\mathcal{W}'(G', f', (S', T')) = T$ iff $val(f') = cap(S', T')$.
- $\mathcal{W}(G, M, C) = T$ iff $|M| = |C|$.
- $g$ translates an integral flow into a matching.
- Lemma 8 proves that $g$ does as required.
- $h$ translates an $(S, T)$-cut into a cover.
- Lemma 9 shows that $h$ does as desired.
- Lemma 10 proves implication (8).

This shows that the example of the reduction from maximum cardinality bipartite matching to maximum flow is indeed a special case of the general scheme of reductions.

### 8.2. *Linear programming duality*

Linear programming duality is a general technique for certifying algorithms. Linear programming is about optimizing linear objective functions in the presence of linear constraints. The dual of a linear program can provide a witness of optimality.

For $n$ nonnegative real variables $x = (x_1, \ldots, x_n)$, the goal is to maximize the linear function $c^T x = \sum_{1 \le j \le n} c_j x_j$ subject to $m$ linear constraints. The $i$th constraint has the form $\sum_{1 \le j \le n} A_{ij} x_j \le b_i$. In matrix notation, a linear program is

defined by an $m \times n$ real matrix $A$ and real vectors $c$ and $b$ of dimension $n$ and $m$, respectively. The goal is to solve:

$$\max c^T x \quad \text{subject to} \quad Ax \leq b \text{ and } x \geq 0.$$

The dual linear program is a linear program in $m$ nonnegative variables $y = (y_1, \ldots, y_m)$. There is one variable for each constraint of the primal linear program. The objective function to be minimized is $y^T b = \sum_{1 \leq i \leq m} y_i b_i$. There is one constraint for each variable of the primal. The $j$th constraint is $\sum_{1 \leq i \leq m} y_i A_{ij} \leq c_i$. Thus, in matrix notation the dual program can be formulated as:

$$\min y^T b \quad \text{subject to} \quad y^T A \geq c^T \text{ and } y \geq 0.$$

**Lemma 11** (*Linear Programming Duality*). *For linear programs, the following holds:*

(a) *Weak duality: If $x^*$ and $y^*$ are solutions to a linear program and its dual, respectively, then $c^T x^* \leq y^{*T} b$.*
(b) *Complementary slackness: Assume $x^*$ and $y^*$ are solutions to a linear program and its dual, respectively, with $c^T x^* = y^{*T} b$. Let $A_{(\cdot,i)}$ be the $i$th column of $A$ and let $A_{(j,\cdot)}$ be the $j$th row of $A$. Then $c_i = y^{*T} A_{(\cdot,i)}$ whenever $x_i^* > 0$ and $b_j = A_{(j,\cdot)} x^*$ whenever $y_j > 0$.*
(c) *Strong duality: If both programs are feasible, then there are solutions $x^*$ and $y^*$ with $c^T x^* = y^{*T} b$.*

**Proof.** We only prove weak duality and complementary slackness. For strong duality, we refer the reader to any textbook on linear programming, e.g., [57]. Weak duality is easy to prove. We have

$$c^T x^* \leq y^{*T} A x^* \leq y^{*T} b.$$

The first inequality follows from $c^T \leq y^{*T} A$ and $x^* \geq 0$ and the second inequality follows from $Ax^* \leq b$ and $y^* \geq 0$.

Assume now that $c^T x^* = y^{*T} b$. Then both inequalities in the equation above must be equalities, i.e.,

$$c^T x^* = y^{*T} A x^* = y^{*T} b.$$

In particular for any $i$ with $x_i^* > 0$, we must have $c_i = y^{*T} A_{(\cdot,i)}$ and for any $j$ with $y_j^* > 0$, we must have $b_j = A_{(j,\cdot)} x^*$, showing the complementary slackness. □

So a certifying algorithm for solving linear programs outputs a primal solution $x^*$ and a dual solution $y^*$ with $c^T x^* = y^{*T} b$. Observe, that it is trivial to check whether $x^*$ is a solution for the primal, that $y^*$ is a feasible solution for the dual, and that $c^T x^* = y^{*T} b$. Also the proof that this certifies optimality is trivial as we have seen above. It is only the existence of $x^*$ and $y^*$ (i.e., the strong duality) which is hard to prove, but this is not needed to be convinced.

The previous section, that deals with the reduction of bipartite matching to the computation of a maximum flow, shows an example of linear programming duality. The dual problem to the maximum flow problem is the minimum $(s, t)$-cut problem. As a second example we now consider the minimum spanning tree problem.

*Minimum spanning trees:* Let $G = (V, E)$ be an undirected connected graph and let $w : E \to \mathbb{R}_{\geq 0}$ be a nonnegative weight function on the edges. The goal is to find a spanning tree $T \subseteq E$ of minimum weight $w(T) = \sum_{e \in T} w(e)$.

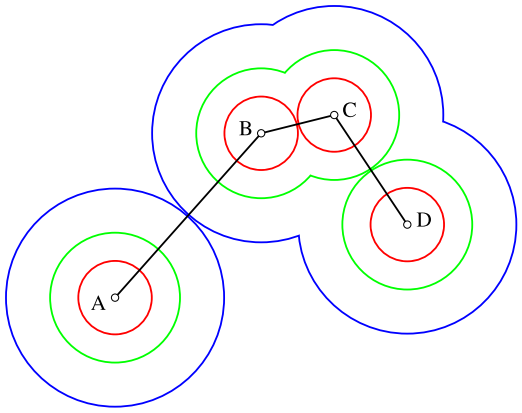



**Fig. 11 – The figure shows the minimum Euclidean spanning tree of points A, B, C, and D. The four points define a complete graph on four vertices; the weight of an edge is the Euclidean distance between its endpoints. The edges shown form a minimum spanning tree of this graph. The figure also shows a proof of optimality of this spanning tree in the form of three moats of radii $r_{\text{red}} < r_{\text{green}} < r_{\text{blue}}$. Any spanning tree $T'$ must have three edges crossing the red moat (=the union of the three red circles) and hence accrues a cost of $3 \cdot 2 \cdot r_{\text{red}}$ within the red moat. Similarly, it must have two edges crossing the green moat (=the region within the green circles, but outside of the red circles) and hence accrues a cost of at least $2 \cdot 2 \cdot (r_{\text{green}} - r_{\text{red}})$ within the green moat. Finally, it must have at least one edge crossing the blue moat (=the region within the blue circles but outside the green circles) and hence accrues a cost of at least $1 \cdot 2 \cdot (r_{\text{blue}} - r_{\text{green}})$ in the blue moat. The tree shown accrues exactly these costs. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)**

What constitutes a proof of optimality? Fig. 11 illustrates such a proof; the drawing and the usage of the word "moat" is inspired by [58]. For a partition $\pi$ of the vertex set let $\#(\pi)$ be the number of blocks of $\pi$ and let $\delta(\pi)$ be the set of edges whose endpoints belong to distinct blocks of $\pi$. Assume we have nonnegative values $y_\pi$, one for each partition $\pi$ of the vertex set such that

$$\sum_{\pi} y_\pi (\#(\pi) - 1) = \sum_{e \in T} w_e \tag{9}$$

$$\sum_{\pi; e \in \delta(\pi)} y_\pi \leq w(e) \quad \text{for all } e \in E \tag{10}$$

$$y_\pi \geq 0 \quad \text{for all } \pi, \emptyset \subset S \subset V. \tag{11}$$

Then the values $y_\pi$ certify optimality of $T$.

**Lemma 12.** *If (9)–(11) hold, $T$ is a minimum weight spanning tree.*

**Proof.** Let $T'$ be any spanning tree. For any partition $\pi$, the number of edges in $\delta(\pi) \cap T'$ must be at least $\#(\pi) - 1$ and hence

$$\begin{aligned} w(T') &= \sum_{e \in T'} w(e) \quad \text{definition of } w(T') \\ &\geq \sum_{e \in T'} \sum_{\pi; e \in \delta(\pi)} y_\pi \quad \text{by inequality (10)} \\ &= \sum_{\pi} \sum_{e \in T' \cap \delta(\pi)} y_\pi \quad \text{change of order of summation} \end{aligned}$$

$$\geq \sum_{\pi} y_\pi(\#(\pi)-1) \quad \text{since } |T' \cap \delta(\pi)| \geq \#(\pi)-1 \text{ and } y_\pi \geq 0$$
$$= w(T) \quad \text{by Eq. (9).}$$

Thus $T$ is a minimum spanning tree. □

We next show how to compute the values $y_\pi$; the construction will associate nonzero values $y_\pi$ only with $n-1$ partitions. Let $T$ be an alleged minimum spanning tree and let $e_1, e_2, \ldots, e_{n-1}$ be the edges of $T$ in increasing order of weight; ties are broken arbitrarily. Let $\pi_1$ be the partition consisting of $n$ singleton blocks and, for $i \geq 2$, let $\pi_{i+1}$ be obtained from $\pi_i$ by uniting the blocks containing the endpoints of $e_i$. Observe that the endpoints must be in distinct blocks since $T$ is a tree. We define $y_\pi = 0$ for any $\pi$ that is not in $\{\pi_1, \ldots, \pi_{n-1}\}$. For simplicity, write $y_i$ instead of $y_{\pi_i}$ and define[6]

$$y_i = \begin{cases} w_{e_1} & \text{if } i = 1 \\ w_{e_i} - w_{e_{i-1}} & \text{if } i > 1. \end{cases}$$

**Lemma 13.** *(9)–(11) hold for the values $y_\pi$ as defined above.*

**Proof.** All values $y_\pi$ are nonnegative. The blocks of $\pi_i$ are exactly the connected components of the graph $(V, \{e_1, \ldots, e_{i-1}\})$. Thus $e_i$ has its endpoints in different blocks for all partitions $\pi_1, \pi_2, \ldots, \pi_i$ and has both endpoints in the same block of $\pi_j$ for $j > i$ and hence

$$w(e_i) = \sum_{2 \leq j \leq i} (w(e_j) - w(e_{j-1})) + w(e_1) = \sum_{j \leq i} y_i = \sum_{\pi; e \in \delta(\pi)} y_\pi.$$

Consider next a non-tree edge $e$ and let $i$ be maximal such that the endpoints of $e$ are in distinct blocks of $\pi_i$. Then $e$ must connect the same blocks of $\pi_i$ as $e_i$ does; otherwise the endpoints of $e$ would be in distinct blocks of $\pi_{i+1}$. Thus $e$ and $e_i$ lie on a common cycle of $(V, \{e_1, \ldots, e_i, e\})$ and hence $w(e) \geq w(e_i)$. Thus (10) holds.

Finally, inspecting the proof of Lemma 12 with $T' = T$, all inequalities must be equalities and hence (9) holds. □

How does one arrive at this certificate? Linear Programming is the key [59]. The fractional minimum weight spanning subgraph problem is easily formulated as a linear program. We have a nonnegative variable $x_e$ for each edge. The value of $x_e$ designates the fraction with which $e$ belongs to the spanning subgraph. The goal is to minimize $\sum_{e \in E} w_e x_e$. We have a constraint for each partition $\pi$ of $V$, namely that $\sum_{e \in \delta\pi} x_e \geq \#(\pi) - 1$, i.e., for each partition, we must pick at least $\#(\pi) - 1$ edges connecting vertices in different blocks of the partition. We may do so by picking edges fractionally. We obtain the following formulation as a linear program.

$$\begin{aligned} \text{minimize} \quad & \sum_e w_e x_e \\ \text{subject to} \quad & \sum_{e \in \delta(\pi)} x_e \geq \#(\pi) - 1 \quad \text{for all partitions } \pi \text{ of } V \\ & x_e \geq 0 \quad \text{for all } e \in E. \end{aligned}$$

It is not obvious that this linear program always has an integral optimal solution. The dual linear program has a variable $y_\pi$ for every partition $\pi$ and reads:

$$\begin{aligned} \text{maximize} \quad & \sum_\pi (\#(\pi) - 1) y_\pi \\ \text{subject to} \quad & \sum_{\pi; e \in \delta(\pi)} y_\pi \leq w_e \quad \text{for every edge } e \\ & y_\pi \geq 0 \quad \text{for all } \pi. \end{aligned}$$

[6] The radii of the moats shown in Fig. 11 are half of these $y_i$'s.

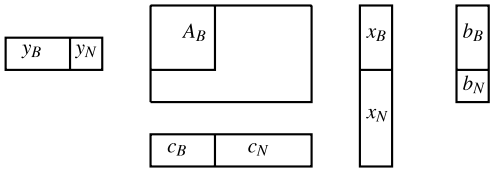



**Fig. 12 – The decomposition into basic and non-basic variables. For simplicity, we assumed that $A_B$ is the left upper corner of the constraint matrix.**

A spanning tree is an integral solution to the primal and Lemma 13 shows that there is a dual solution with the same objective value. Thus a minimum spanning tree is an optimal solution to the primal and the dual solution proves its optimality. Lemma 12 is a proof of weak duality for this special case.

*Verifying linear programs:* Linear programming duality is a great method for checking optimality of solutions to linear programs. Given a feasible solution $x^*$ to the primal program linear program

$$\text{maximize } c^T x \text{ subject to } Ax \leq b \text{ and } x \geq 0$$

and a feasible solution $y^*$ to the corresponding dual linear program

$$\text{minimize } y^T b \text{ subject to } y^T A \geq c^T \text{ and } y \geq 0$$

the equality

$$c^T x^* = y^{*T} b \tag{12}$$

implies optimality of both solutions. Unfortunately, linear programming solvers [60,61] for general linear programs are numerical procedures and yield only approximate solutions. So we cannot hope that the computed approximate solutions satisfy equation (12). Fortunately, linear programming solvers also return a combinatorial description of the optimal solution in terms of a basis.

A basis is a square non-singular sub-matrix $A_B$ of the constraint matrix $A$. We call the primal variables corresponding to columns of $A_B$ basic and denote them by $x_B$; the other primal variables are called non-basic and denoted by $x_N$. Similarly, we call the dual variables corresponding to rows of $A_B$ basic and denote them by $y_B$; the other dual variables are called non-basic and denoted by $y_N$. Analogously, we split $b$ into $b_B$ and $b_N$ and $c$ into $c_B$ and $c_N$, see Fig. 12. A basis induces a primal solution $(\hat{x}_B^T, \hat{x}_N^T)$ (not necessarily feasible) and a dual solution $(\hat{y}_B^T, \hat{y}_N^T)$ (not necessarily feasible) by way of:

$$A_B \hat{x}_B = b_B \quad \text{or} \quad \hat{x}_B = A_B^{-1} b_B \quad \text{and} \quad \hat{x}_N = 0 \tag{13}$$

$$\hat{y}_B^T A_B = c_B^T \quad \text{or} \quad \hat{y}_B^T = c_B^T A_B^{-1} \quad \text{and} \quad \hat{y}_N = 0. \tag{14}$$

The objective value of these solutions is equal. Indeed,

$$(c_B^T, c_N^T) \begin{pmatrix} \hat{x}_B \\ \hat{x}_N \end{pmatrix} = c_B^T \hat{x}_B = c_B^T A_B^{-1} b_B = \hat{y}_B b_B = (\hat{y}_B^T, \hat{y}_N^T) \begin{pmatrix} b_B \\ b_N \end{pmatrix}.$$

A basis is primal feasible if $\hat{x}_B \geq 0$ and is dual feasible if $\hat{y}_B \geq 0$.

**Lemma 14.** *If a basis is primal and dual feasible, the corresponding primal $(\hat{x}_B^T, \hat{x}_N^T)$ and dual solution $(\hat{y}_B^T, \hat{y}_N^T)$ are optimal.*

**Proof.** $(\hat{x}_B^T, \hat{x}_N^T)$ is a solution to the primal linear program, $(\hat{y}_B^T, \hat{y}_N^T)$ is a solution to the dual linear program, and their objective values are equal. Thus, the solutions are optimal by Lemma 11. □

Optimality of a basis $B$ can now be checked as follows [29]. Eqs. (13) and (14) are used to compute the primal and dual solutions corresponding to the basis; the computation is carried out in exact rational arithmetic. If both solutions are feasible, the basis is optimal. In this way, the speed of floating point arithmetic is used to find the optimal basis and the exactness of rational arithmetic is used to certify the solution. If the basis is not optimal but "close to optimal", it can be taken as the starting basis for an exact primal or dual Simplex algorithm. The use of exact rational arithmetic can be replaced by the use of high-precision floating point arithmetic [30].

### 8.3. *Characterization theorems*

Within characterization theorems sometimes lies the potential to certify an output. We have already seen examples of this: A graph is not planar if and only if it contains a Kuratowski subgraph (see Section 2.6), and a graph is not 3-connected if and only if it contains a separating pair (see Section 5.4).

Interestingly these characterizations follow a certain pattern: One direction of the proof of the characterization is easy, and this side corresponds to required simplicity of the witness. The more difficult direction is the one required to establish the existence of a witness.

To clarify this statement we provide another example: A graph $G$ is *perfect* if the chromatic number of every induced subgraph $H$ of $G$ is equal to the size of the largest clique of $H$.

An *odd hole* in a graph is an induced odd cycle of length at least 5, an *odd anti-hole* is an induced subgraph isomorphic to the complement of an odd hole. The strong perfect graph theorem [62] says that a graph is perfect if and only if it contains neither an odd hole nor an odd anti-hole.

The chromatic number of odd holes and of odd anti-holes is not equal to the size of their largest clique, so they cannot be contained in a perfect graph. This easy part of the characterization shows that odd holes and odd anti-holes certify a graph to be not perfect. The second part of the strong perfect graph theorem, the existence of an odd hole or an odd anti-hole in a non-perfect graph, resolves a conjecture that had been open for more than 40 years. The polynomial-time algorithm for the recognition of perfect graphs [63] detects an odd hole or an odd anti-hole in a perfect graph, and thereby certifies a graph to be not perfect.

### 8.4. *Approximation algorithms and problem relaxation*

Approximation algorithms compute nearly-optimal solutions of optimization problems. They come with a guarantee for the quality of approximation. We may assume w.l.o.g. that we deal with a minimization problem. In the proof of correctness the quality of the solution is usually measured against an easy to compute lower bound for the problem. A certifying approximation algorithm should output (in a certifying way) a lower bound in addition to the solution. This can either be the lower bound used in the proof of correctness or another lower bound. The general technique for obtaining lower bounds is problem relaxation, i.e., enlarging the set of feasible solutions. We give an example.

The traveling salesman problem asks for a shortest cycle visiting all vertices of a edge-weighted undirected graph $G = (V, E, c)$. The cost $c(C)$ of a cycle $C$ is the sum of the costs of the edges contained in $C$. We assume that $c$ satisfies the triangle inequality, i.e, $c(uv) \leq c(uw) + c(wv)$ for any triple of vertices $u$, $v$ and $w$. There are approximation algorithms which produce a solution whose cost is at most 1.5 times the optimum [64]. We discuss two lower bounds for the traveling salesman problem: 1-trees and the subtour elimination linear program. Both approaches yield the same value, but use very different principles [59].

*Lower bounds via 1-trees* A *1-tree anchored at a vertex* $v$ [65] is a spanning tree of $G \setminus v$ plus two edges incident to $v$. A *1-tree* is a 1-tree anchored at some vertex $v$ of $G$. As for cycles, the cost of a 1-tree is the sum of the costs of the edges in the 1-tree. A minimum 1-tree is a 1-tree of minimum cost. Minimum 1-trees are readily computed by $n$ minimum spanning tree computations since the minimum 1-tree anchored at a vertex $v$ is simply a minimum spanning tree of $G \setminus v$ plus the two cheapest edges incident to $v$.

Minimum 1-trees can be used to lower bound the cost of any traveling salesman tour. For this let $\pi$ be any real-valued function defined on the vertices of $G$ and consider the modified cost function $c^\pi(uv) = c(uv) + \pi(u) + \pi(v)$. We call $\pi$ a potential function. The cost of a one-tree $T$ under the cost function $c^\pi$ is defined as $c^\pi(T) = \sum_{uv \in T} c^\pi(uv)$. Accordingly, a minimum 1-tree with respect to the modified cost $\pi$ is a 1-tree with minimal modified cost.

**Lemma 15.** *Let $C$ be a traveling salesman tour, let $\pi$ be a potential function, and let $T$ be a minimum 1-tree with respect to $\pi$. Then*

$$c^\pi(T) - 2\sum_{v \in V} \pi(v) \leq c(C).$$

**Proof.** Since $c$ satisfies the triangle inequality, there is a tour $D$ visiting every vertex exactly once and having cost no more than $C$. This tour $D$ is a 1-tree (with respect to any anchor) and hence

$$c^\pi(T) \leq c^\pi(D).$$

Since $D$ uses exactly two edges incident to any vertex,

$$c(D) = c^\pi(D) - 2\sum_{v \in V} \pi(v).$$

Combining the equalities and inequalities, we obtain

$$c(C) \geq c(D) = c^\pi(D) - 2\sum_{v \in V} \pi(v) \geq c^\pi(T) - 2\sum_{v \in V} \pi(v). \quad \square$$

A certifying approximation algorithm for the traveling salesman problem outputs a tour $C$, a potential function $\pi$ and a minimum 1-tree $T$ for $c^\pi$, and a proof of optimality of $T$. The proof of optimality reduces to the minimum spanning tree problem, for which we have discussed a certifying algorithm in Section 8.2.

A good potential function $\pi$ can be found by an iterative process [65,66]. Observe that

$$c^{\pi}(T) - 2\sum_{v\in V}\pi(v) = c(T) + \sum_{v\in V}\deg_T(v)\pi(v) - 2\sum_{v\in V}\pi(v)$$
$$= c(T) + \sum_{v\in V}(\deg_T(v) - 2)\pi(v),$$

where $\deg_T(v)$ is the number of edges of $T$ incident to $v$. We conclude that the vector $(\deg_T(v)-2)_{v\in V}$ is the gradient of the expression $c^{\pi}(T) - 2\sum_{v\in V}\pi(v)$ viewed as function of $\pi$.

We start an iterative process with $\pi(v) = 0$ for all $v$. In an iteration, we first compute the minimum 1-tree $T$ with respect to the current modified cost function $c^{\pi}$. If $T$ is a tour, i.e., $\deg_T(v) = 2$ for all $v$, we stop; $T$ is an optimal tour. Otherwise, we update $\pi$ to $\pi'$ as follows:

$$\pi'(v) = \pi(v) + \epsilon(\deg_T(v) - 2) \quad \text{for all } v \in V,$$

where $\epsilon > 0$ is a small value; this is a small step in the direction of the gradient and increases the potential value of vertices of degree three or higher and decreases the potential value of vertices of degree one.[7] We set $\pi = \pi'$ and repeat. The iterative process produces a sequence of lower bounds. We remember the best lower bound computed in this way and use it to produce a lower bound for our problem, as described above.

*Lower bounds via linear programming* An integer linear programming formulation of the traveling salesman problem is as follows. We have a decision variable $x_e$ for each edge of the graph with the intention that the edges with $x_e = 1$ comprise an optimal solution. A tour contains two edges incident to every vertex and for every non-empty proper subset $S$ of $V$ there must be at least two edges in any tour with exactly one endpoint in $S$. We obtain the following formulation as an integer linear program.

$$\begin{aligned}
\text{minimize} \quad & \sum_e c_e x_e & & \\
\text{subject to} \quad & \sum_{e\in\delta(v)} x_e & = 2 & \quad \text{for all } v \in V \\
& \sum_{e\in\delta(S)} x_e & \geq 2 & \quad \text{for all } S \text{ with } \emptyset \neq S \neq V \\
& x_e & \in \{0,1\} & \quad \text{for all } e \in E.
\end{aligned}$$

Here we use $\delta(S)$ to the denote the set of edges with exactly one endpoint in $S$. The equality constraint for vertex $v$ is called a *degree constraint* and the inequality for subset $S$ is called a *subtour elimination constraint*. Consider any solution of this system and let $T$ be the edges picked; $e \in T$ iff $x_e = 1$. Then $T$ contains two edges incident to every vertex and hence is a collection of cycles. Assume for the sake of a contradiction, that the collection consists of more than one cycle and let $S$ be the vertex set of one of the cycles. Then $T$ contains no edge in $\delta(S)$ and hence violates the subtour elimination constraint for $S$.

The subtour LP is obtained from the ILP by replacing the constraint $x_e \in \{0,1\}$ by the weaker linear constraint $0 \leq x_e \leq 1$. Thus the subtour LP is a relaxation of the traveling salesman problem and provides a lower bound. Fig. 13 gives an example. In this example, the cost of an optimum tour is 10 and the objective value of the subtour LP is 9. The gap may be as large as a factor of two [59].

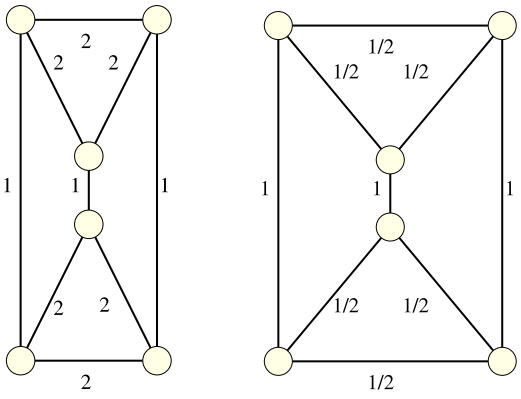



**Fig. 13 – The figure on the left shows the edge costs; there are six edges of cost two and three edges of cost one. The figure on the right shows an optimal solution to the subtour LP. The decision variables corresponding to the edges of cost two have value 1/2 and the decision variables corresponding to the edges of cost one have value 1. The solution has cost 9. The optimal tour has cost 10.**

The subtour LP has an exponential number of constraints, one for each non-empty proper subset of the vertices. It can be solved in polynomial time by means of the ellipsoid method [37]. In practice, one uses the simplex method and a judiciously chosen subset of the subtour elimination constraints. The subset is determined dynamically by a technique called *separation*. Let $(x_e^*)$ be a solution to an LP comprising the degree constraints, the bounding constraints $0 \leq x_e \leq 1$, and some of the subtour elimination constraints. Consider an auxiliary graph with vertex set $V$, edge set $E$, and set the capacity of the edge $e$ to $x_e^*$. Then $(x_e^*)$ violates a subtour elimination constraint if and only if this auxiliary graph has a cut of capacity less than two. Such a cut can be found by a minimum cut computation [67]. If a violated subtour elimination constraint is found, it is added to the LP, and the LP is resolved.

In Section 8.2 we learned how to verify solutions to linear programs. Now we want to certify a lower bound and this can be done by a simple rounding procedure [68]. Consider the dual for the subtour LP. It has an unconstrained variable $\pi_v$ for each vertex $v$, a nonnegative variable $y_S$ for each non-empty proper subset $S$ of $V$, and a nonnegative variable $z_e$ for each edge $e$. The variable $z_e$ corresponds to the upper bound

[7] For an edge $uv$, we have $c^{\pi'}(uv) = c^{\pi}(uv) + \epsilon(\deg_T(u) + \deg_T(v) - 4)$. Also,

$$c^{\pi'}(T) - 2\sum_{v\in V}\pi'(v) = c^{\pi}(T) - 2\sum_{v\in V}\pi(v) + \epsilon\sum_{v\in V}(\deg_T(v) - 2)^2.$$

Thus the cost of the 1-tree $T$ increases by the change of the potential function. However, at $\pi'$ a different 1-tree may be minimal and hence it is not guaranteed that the iteration produces better and better lower bounds. In fact, in general, it does not do so.

constraint $x_e \leq 1$. The goal is to

$$\begin{aligned} \text{maximize} \quad & 2\sum_v \pi_v + 2\sum_S y_S - \sum_e z_e \\ \text{subject to} \quad & \pi_u + \pi_v + \sum_{S;e\in\delta(S)} y_S - z_e \leq c_e \quad \text{for } e = (u,v) \in E \\ & y_S \geq 0 \quad \text{for all } S \text{ with } \emptyset \neq S \neq V \\ & z_e \geq 0 \quad \text{for all } e. \end{aligned}$$

If the primal or dual LP is solved by an LP-solver, the basis returned is not necessarily optimal. Also, primal feasibility and dual feasibility are not guaranteed. However, the solutions are usually close to optimal. In the case of the subtour LP, this can be exploited as follows. Consider any (not necessarily feasible) dual solution $(\pi_v)$, $(y_S)$, and $(z_e)$. We first replace any negative $y_S$ by zero and we then choose the $z_e$ large enough so that all dual constraints are satisfied. In this way, we obtain a feasible dual solution and hence a lower bound for the traveling salesman problem.

### 8.5. *Composition of programs*

Suppose that we have certifying algorithms for I/O-behaviors $(\varphi_1, \psi_1)$ and $(\varphi_2, \psi_2)$. How can we obtain a certifying algorithm for the composed behavior? That is easy.

Let $Q_1$ and $Q_2$ be certifying algorithms for the two I/O-behaviors, respectively, and let $C_1$ and $C_2$ be the corresponding checkers. A certifying algorithm for the composed I/O behavior works as follows: Assume that our input is $x$.

Run $Q_1$ on $x$. This produces $y$ and a witness $w_1$.
**if** $(y = \perp)$ **then**
  Output $\perp$ and the witness $(w_1, \perp, \perp)$
**else**
  Run $Q_2$ on $y$. This produces $z$ and a witness $w_2$.
  Output $z$ and the witness $w = (w_1, y, w_2)$.
**end if**

The checker $C$ for the composed behavior accepts $(w', y, w'')$ as a witness if

$$y = \perp \quad \text{and} \quad C_1 \text{ accepts } (x, y, w') \text{ or} \tag{15}$$

$$C_1 \text{ accepts } (x, y, w') \quad \text{and} \quad C_2 \text{ accepts } (y, z, w''). \tag{16}$$

Two I/O-behaviors should only be composed if the postcondition of the first behavior implies the precondition of the second behavior, i.e., $\psi_1(x, y) \implies \varphi_2(y)$. If $Q_2$ is strongly certifying, it can discover a misuse of composition: Assume that the $y$ output by $Q_1$ does not satisfy $\varphi_2$. Then $Q_2$ will either produce a $z$ with $\psi_2(y, z)$ and a proof that it did so or a proof for $\neg\varphi_2(y)$. In the former case, $Q_2$ could handle $y$ although it did not have to do so, in the latter case $Q_2$ states that its precondition is violated.

## 9. Further examples

In the introductory Section 1 we have discussed the examples of bipartition, connected components, shortest paths, greatest common divisors, maximum cardinality matchings and planarity. In Section 5.2 we discussed a strongly certifying algorithm that five-colors a planar graph and in Section 5.4 we described a simple way to certify triconnectedness. We now discuss further illustrative examples, demonstrating the broad applicability of certification.

### 9.1. *Convexity of higher-dimensional polyhedra and convex hulls*

The convex hull of a finite set $S$ of points in $d$-dimensional space is the smallest convex set containing $S$. Its boundary is a piecewise linear hyper-surface. There are many algorithms for higher-dimensional convex hulls [69–73] and implementations of some [74,75,33]. In 2 and 3 dimensions the output of a geometric algorithm can be visualized and this helps debugging geometric programs. In higher dimensions, visualization is not possible. How can one certify the output of a convex hull algorithm?

What is the output? All algorithms output the boundary of the convex hull as a simplicial piecewise linear hyper-surface $\mathcal{F}$. We will define this term below. In 3-dimensional space the boundary is given as a set of triangles (in 3-space) that are glued together at their edges.

**Task 1.** Given a set $S$ of points and a hyper-surface $\mathcal{F}$ verify that $\mathcal{F}$ is the boundary of the convex hull of $S$.

We split this task into two subtasks.

**Subtask 1.** Given a piecewise linear simplicial hyper-surface $\mathcal{F}$ in $d$-dimensional space verify that $\mathcal{F}$ is the surface of a convex polytope.

Assume that $\mathcal{F}$ is the surface of a convex polytope and let $P$ be the convex polytope whose boundary is $\mathcal{F}$.

**Subtask 2.** Verify that

- every vertex of $P$ is a point in $S$ and that
- every point of $S$ is contained in $P$.

We discuss the two subtasks in turn. This section is based on [7, Section 2.3]. An alternative solution can be found in [76]. We first deal with the Subtask 1, whether a simplicial piecewise linear hyper-surface $\mathcal{F}$ without boundary in $d$-dimensional space is the boundary of a convex polytope. We assume that the hyper-surface is given by its facet graph. The facet graph is a $d$-regular graph whose nodes are labeled by $d$-tuples of affinely independent points, i.e., each node corresponds to an oriented $(d-1)$ simplex (=a facet of the surface). The hyperplane supporting a facet divides $d$-space into a positive and a negative halfspace. Neighboring nodes differ in their labeling by exactly one point and for everyone of the $d$ vertices of a facet there must be such a neighboring facet. In other words, edges correspond to $(d-2)$-simplices shared by two facets. Neighboring nodes must be labeled consistently, i.e., the negative halfspace corresponding to adjacent facets must agree locally.

Let us interpret this definition in 3-space. Every node of the facet graph corresponds to an oriented triangle in 3-space. Oriented means that the two sides of the triangle are distinguished, one is "outside" and one is "inside" (in the paragraph above, inside and outside are called negative and positive, respectively). Adjacent triangles share two vertices and differ in one. Every triangle has three neighbors and the two sides of adjacent triangles are locally consistent.

For smooth surfaces, already Hadamard described a test for convexity.

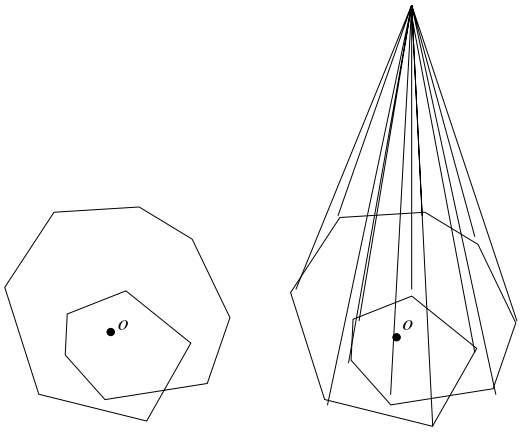



**Fig. 14 – Local convexity at ridges does not suffice: The figure shows a locally convex yet self-intersecting polygon in the $x$, $y$-plane. A smooth version of this curve demonstrates that the condition $d > 2$ is necessary in Theorem 9. Extending the polygon to a bipyramid by adding two points, one each on the negative and positive $z$-axis, and constructing two triangles for each polygon edge yields a simplicial surface that is locally convex at every ridge but has self-intersections; in the figure only the upper half of the bipyramid is shown. The point $o$ shown is on the negative side of all facets of the surface.**

**Theorem 9** (*Hadamard*). *Let $\mathcal{F}$ be a smooth compact surface in $\mathbb{R}^d$ without boundary and let $d > 2$. If $\mathcal{F}$ is locally convex at everyone of its points then $\mathcal{F}$ is the surface of a convex body.*

This theorem suggests that it suffices to check local convexity at every ridge of a simplicial surface. Although this is clearly a necessary condition for global convexity it is not sufficient as Fig. 14 shows, i.e., the non-smoothness of simplicial surfaces complicates matters. The following theorem is the proper formulation for the polyhedral case:

**Theorem 10** ([7]). *Let $\mathcal{F}$ be a simplicial $(d-1)$-dimensional surface without boundary in $\mathbb{R}^d$ that is consistently oriented, let $o$ be center of gravity of all the vertices of surface $\mathcal{F}$ and let $p$ be the center of gravity of some facet of $\mathcal{F}$. Then $\mathcal{F}$ is the surface of a convex body iff*

- *$\mathcal{F}$ is locally convex at all its ridges,*
- *$o$ is on the negative side of all its facets, and*
- *the ray emanating from $o$ and passing through $p$ intersects only the facet containing $p$.*

We refer the reader to [7] for a proof of this result. Fig. 14 illustrates it: Let $o$ be any point in the $x, y$-plane that is on the negative side of every facet of the surface shown. All but two rays emanating from $o$ intersect the surface twice and hence witness the non-convexity of the surface. The two exceptional rays go through the two tips of the bipyramid, i.e., pass through a lower dimensional feature of the surface. The key insight underlying the criterion is that this observation is generally true.

The conditions listed in Theorem 10 are clearly necessary. Also, if every ray emanating from $o$ intersects $\mathcal{F}$ only once, $\mathcal{F}$ is the surface of a convex body. It is somewhat surprising, that it suffices to compute the number of intersections for a single ray. The verification is easy to program.

- Check local convexity at every ridge. If local convexity does not hold at some ridge declare $\mathcal{F}$ non-convex.
- Set $o$ to the center of gravity of the vertices of $\mathcal{F}$ and check whether $o$ is on the negative side of all facets. If not, declare $\mathcal{F}$ non-convex.
- Choose any facet and let $p$ be the center of gravity of its vertices. Let $r$ be the ray emanating from $o$ and passing through $p$. If $r$ intersects the closure of any other facet of $\mathcal{F}$ declare $\mathcal{F}$ non-convex.
- If $\mathcal{F}$ passes all three tests declare it the surface of a convex polytope.

We next turn to the Subtask 2. Assume that $\mathcal{F}$ passed the convexity test and let $P$ be the convex polyhedron with boundary $\mathcal{F}$. We need to verify that

- every vertex of $P$ is a point in $S$ and that
- every point of $S$ is contained in $P$.

The first item is fairly easy to check. If the vertices of $P$ are equipped with pointers to elements in $S$, the check is trivial. If the vertices of $P$ are specified by their coordinate tuples, the check involves a dictionary lookup.

The second condition is much harder to check. In fact, without additional information (=the witness), there seems to be no efficient way to verify it. A simple method would be to check every point of $S$ against every facet of $\mathcal{F}$. However, the complexity of this method is an order of magnitude larger that the complexity of the most efficient convex hull programs.[8] An alternative method is to use linear programming to check that all non-vertices are non-extreme.[9] For fixed dimension the alternative method is quadratic in the number of vertices. For variable dimension one might hope that a simplex-based verification procedure has good expected running time. Nevertheless, both approaches essentially resolve the original problem. We conclude that convex hull programs that output the hull as a convex polytope are hard to check. The "gift-wrapping" algorithm [69] falls in this category.

What is an appropriate witness that makes checking easy? Here is one. Arrange the points in $S$ in a linear order and for each point $p$ in $S$ that is not a vertex of $P$ indicate a set of $d+1$ points that come later in the ordering and that contain $p$ in their convex hull. We call such an ordering an *admissible ordering* of $S$.

**Lemma 16.** *An admissible ordering of $S$ proves that every point of $S$ is contained in $P$.*

[8] Algorithms based on randomized incremental construction [72,73,77] run in time related to the size of the output and the size of intermediate hulls and the algorithm of [71] is guaranteed to construct the hull in logarithmic time per face.

[9] The linear program has $d$ variables corresponding to the coefficients of a linear function. For each vertex of $\mathcal{F}$ there is a constraint stating that the function value at the vertex is negative. For each non-vertex consider the linear program that maximizes the function value at this point.

**Proof.** Let $q_1$ to $q_n$ be an admissible ordering of $S$. We show that each $q_i$ is a convex combination of vertices of $P$. We use induction on $i$ starting at $n$ and going down to 1. Consider any $i$. If $q_i$ is a vertex of $P$, there is nothing to show. Otherwise, $q_i$ is a convex combination of points that come later in the ordering. By induction hypothesis, these points are convex combinations of the vertices of $P$. Thus $q_i$ is a convex combination of the vertices of $P$. □

The witness is easily checked. For each point that is claimed to be a convex combination of points later in the ordering, one needs to solve a linear system.

The algorithms based on randomized incremental construction [72–75,33] can be modified to compute this witness. They compute a simplicial complex[10] comprising the hull, i.e., a set of simplices whose union is $P$. They do so incrementally. They start with $d+1$ points of $S$ spanning a simplex and then add point after point. If a new point $p$ is contained in the current hull, they determine a simplex in the current simplicial complex containing $p$. The vertices of this simplex are the witnesses for $p$. If the new point $p$ is outside the current hull, they determine all facets of the current hull visible from the new point. For each such facet $F$ they add a simplex $S(F, p)$ with base $F$ and tip $p$ to the simplicial complex.

Assume now that the algorithm is rerun: first the vertices of $P$ are inserted (in random order) and then the non-vertices (in any order). In this way, all simplices in the simplicial complex have their vertices among the vertices in $P$ and each non-vertex in $S$ is placed in a simplex spanned by vertices of $P$.

### 9.2. Solving linear systems of equations

We consider a system $Ax = b$ of $m$ linear equations in $n$ unknowns $x$; here $A$ is a $m$ by $n$ matrix of real numbers and $b$ is an $m$-vector. We want to know whether there is a solution to this system. Again, a conventional algorithm would just return a single bit, telling whether the system is solvable or not.

A certifying algorithm would do a lot more. If it declares the system solvable, it would return a solution, i.e., an $n$-vector $x_0$ such that $Ax_0 = b$. Observe that it is very easy to verify whether a given $x_0$ is a solution. We simply plug $x_0$ into the equation.

If it declares the system non-solvable, it could return an $m$-vector $c$ such that $c^TA = 0$ and $c^Tb \neq 0$. Observe that such a vector witnesses non-solvability. Indeed, assume the existence of a solution $x_0$. Then $c^TAx_0 = (c^TA)x_0 = 0^Tx_0 = 0$ and $c^TAx_0 = c^T(Ax_0) = c^Tb \neq 0$, a contradiction. Thus there is no solution.

Why does such a $c$ exist? If $Ax = b$ has no solution, $b$ does not belong to the space spanned by the columns of $A$. Thus we can write $b$ as $b = b' + b''$, where $b'$ is in the span of the columns of $A$ and $b''$ is orthogonal to all columns of $A$. We can take $c = b''$. Then $c^TA = 0$ and $c^Tb = b''^Tb'' \neq 0$.

How can we compute such a $c$? We use Gaussian elimination. It is well known that it returns a solution when given a solvable system. It is less well known that it also returns a witness for non-solvability when given an unsolvable system. We describe Gaussian elimination as a recursive procedure. If all entries of $A$ and $b$ are zero, the zero vector $0^n$ is a solution. If all entries of $A$ are zero and $b$ is nonzero, say $b_i \neq 0$, the $m$-vector $e_i$ having a one in position $i$ and zero everywhere else witnesses non-solvability.

So assume that $A$ has a nonzero entry, say $A_{ij} \neq 0$. We subtract a suitable multiple of the $i$th equation from all other equations (i.e., we subtract $A_{lj}/A_{ij}$ times the $i$th equation from the $l$th equation for $1 \leq l \leq m$ and $l \neq i$) so as to eliminate the $j$th variable from the other equations. The transformation yields a system with $m-1$ equations in $n-1$ unknowns, say $A'x' = b'$. Here $A'_{lk} = A_{lk} - (A_{lj}/A_{ij})A_{ik}$ and $b'_l = b_l - (A_{lj}/A_{ij})b_i$ for $l \neq i$ and all $k$. Also, row index $i$ and column index $j$ are deleted from the index set of the reduced system. Assume first that the reduced system is solvable and $x'_0$ is a solution. We plug $x'_0$ into the $i$th original equation, solve for $x_j$ and obtain a solution to the original system. Assume next that the reduced system is unsolvable and that $c'$ witnesses it, i.e., $c'^TA = 0$ and $c'^Tb' \neq 0$. We define the $m$-vector $c$ by $c_l = c'_l$ for $l \neq i$ and $c_i = -\sum_{l \neq i}(A_{lj}/A_{ij})c'_l$. Then for any $k$, $1 \leq k \leq n$,

$$\begin{aligned}
\sum_l c_l A_{lk} &= c_i A_{ik} + \sum_{l \neq i} c_l A_{lk} \\
&= c_i A_{ik} + \sum_{l \neq i} c'_l (A'_{lk} + (A_{lj}/A_{ij})A_{ik}) \\
&= \left( c_i + \sum_{l \neq i} c'_l (A_{lj}/A_{ij}) \right) A_{ik} + \sum_{l \neq i} c'_l A'_{lk} \\
&= \sum_{l \neq i} c'_l A'_{lk} \\
&= 0.
\end{aligned}$$

An analogous computation shows that $c^Tb = c'^Tb'$ and hence $c^Tb \neq 0$. We have now shown that Gaussian elimination easily turns into a certifying solver for linear systems of equations.

### 9.3. NP-complete problems

Branch-and-Bound and Brunch-and-Cut are powerful methods for computing optimal solutions to NP-complete problems [78]. The algorithms use a heuristic for computing a feasible solution and compute a matching lower bound (we again assume a minimization problem) for the objective value of any feasible solution to prove optimality. For the latter, they partition the search space and compute a lower bound for each cell of the partition. The heuristic solution is optimal if its objective value matches the lower bound.

In the case of the Traveling Salesman Problem (see Section 8.4) the partition is usually by inclusion and exclusion of edges. For example, we might divide the search space into two parts, by considering all tours containing a particular edge $uv$ and all tours not containing this edge. Fig. 15 shows an example. The two cells obtained in this way can be subdivided further using the same strategy recursively. In each cell a lower bound is computed, e.g., using the methods of Section 8.4. In summary, the approach is as follows.

- Use a heuristic to find the optimum solution. Verify that the solution is feasible.

[10] A simplicial complex is a set of simplices the intersection of any two is a face of both.

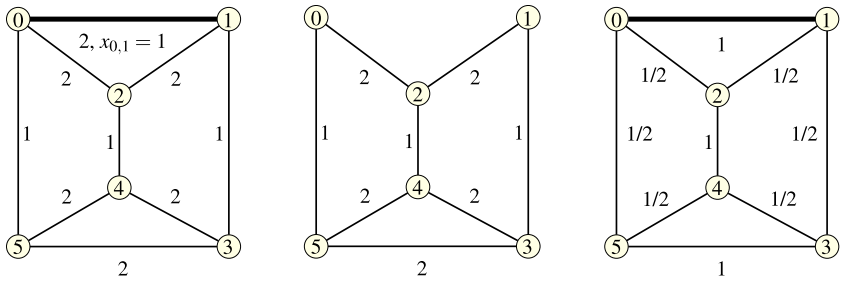



**Fig. 15 – In the example of Fig. 13, the optimal solution to the subtour LP is fractional; $x_{0,1}$ has value 1/2. We generate two subproblems. In the first subproblem, we set $x_{0,1}$ to 1 and force the edge into the tour. In the second subproblem, we set $x_{0,1}$ to 0; this is tantamount to deleting the edge. For both subproblems, the subtour LP (in fact, the LP with only the degree constraints) has objective value 10. For the first subproblem there is a non-integral solution of value 10 as shown on the right, there is also an integral solution of value 10.**

- Partition the space of feasible solutions. Verify that the partition is indeed a partition.
- Compute for each cell of the partition a lower bound on the objective value of the feasible solutions in this cell. Verify that the lower bound computed for each cell is indeed a lower bound and has at least a value equal to the cost of the heuristic solution computed in the first step.

The first two steps are typically simple. For the last step, one uses the techniques discussed in Section 8.4. In [68], Applegate et. al. report about the certification of an optimal TSP tour through 85,900 cities. The tour was obtained by a heuristic [79] and then verified by the approach outlined above.

As a second example consider the satisfiability problem of propositional logic. Let $\varphi$ be a boolean formula. A satisfying assignment is a witness of satisfiability. A resolution proof is a witness of non-satisfiability. The resolution proof may have exponential length; it is however, easy to check.

### 9.4. *Maximum weight independent sets in interval graphs*

Given a collection of weighted intervals, the goal is to find an independent set of maximum weight. We use $i$ to denote a generic interval and $I$ to denote an independent set. The goal is then to find an independent set $I$ of intervals of maximal weight $\sum_{i\in I} w_i$, where $w_i > 0$ is the weight of interval $i$. This problem is one of the introductory examples in the textbook of Kleinberg and Tardos [80].

The standard algorithm for this problem uses dynamic programming. Assume that the intervals are numbered in the order of their left endpoint. The optimal solution either contains interval 1 or it does not. Thus

$$Opt(1, n) = \max(Opt(2, n), w_1 + Opt(j, n))$$

where $j$ is minimal such that the $j$th interval is independent of the first, i.e., its left endpoint is to the right of the right endpoint of interval 1. The algorithm has linear running time by use of memoization.

We will next derive a linear time certifying algorithm. A clique is a set of intervals that intersect pairwise. A maximal clique is one that is not contained in any other clique. Consider the sorted list of all interval endpoints. Cliques correspond to the elementary intervals. A clique is maximal if the left endpoint of the corresponding elementary interval is the left endpoint of an interval and the right endpoint of the corresponding elementary interval is the right endpoint of an interval.

We compute an independent set $I^*$ and nonnegative values $y_C$ and $w_{iC}$ for each maximal clique $C$ and interval $i \in C$ such that (see Fig. 16)

$$w_i = \sum_{C;i\in C} w_{iC} \quad \text{and} \quad w_{iC} \le y_C \quad \text{and} \quad \max_{i\in I^*\cap C} w_{iC} = y_C.$$

Consider now any independent set $I$ of intervals. Then

$$\sum_{i\in I} w_i = \sum_{i\in I} \sum_{C:i\in C} w_{iC} = \sum_C \sum_{i\in I\cap C} w_{iC} = \sum_C \max_{i\in I\cap C} w_{iC} \le \sum_C y_C,$$

where the third equality follows from the fact that each clique can contain at most one element of $I$. For $I = I^*$, the inequality is an equality. Thus $w(I) \le w(I^*)$ and $I^*$ is a maximum weight independent set.

For an interval $i$, let $L_i$ be the leftmost maximal clique containing $i$. A simple greedy algorithm determines the $y_C$ and $w_{iC}$ values. We process the maximal cliques in order (say from right to left) and assign to each clique a value $y_C$. We also maintain reduced weights $w'_i$ for all intervals $i$. Initially, $w'_i = w_i$ for all $i$. Let $C$ be the current maximal clique (initially the rightmost maximal clique). We set

$$y_C = \max\left\{ w'_i \mid L_i = C \right\},$$

i.e., we set $y_C$ to the maximal reduced weight of any interval having $C$ as its leftmost maximal clique. Maximality of $C$ guarantees that there is at least one such interval. Also, for any $i$ contained in $C$, we set $w_{iC}$ to the minimum of $y_C$ and $w'_i$ and reduce $w'_i$ by $w_{iC}$.

An interval $i$ is called *defining* for $C$ if $C = L_i$ and $y_C = w_{iC}$ and $y_C > 0$. A clique $C$ with $y_C = 0$ has no defining interval. An interval is called tight for $C$ if $w_{iC} = y_C$. The following Lemma is key (and also obvious).

**Lemma 17.** *If $i$ is defining for $L_i$ then $i$ is tight for all cliques containing it.*

**Proof.** If $i$ is defining for $L_i$, $w_{iL_i} = y_{L_i} > 0$. This implies $w_{iC} = y_C$ for all maximal cliques containing $i$. □

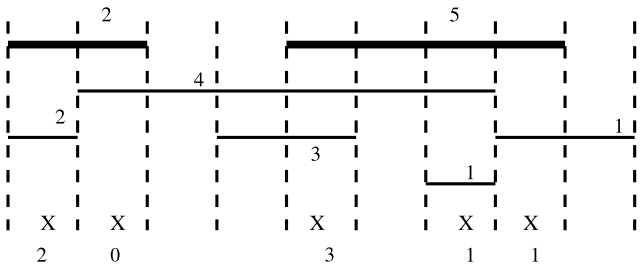



**Fig. 16 – An instance of the maximum weight independent set problem: the interval are indicated as horizontal lines. The weight of each interval is indicated near the interval. The two intervals drawn as heavy lines form a maximum weight independent set. There are five maximal cliques indicated by X, their $y_C$-values are indicated below the X's.**

We next construct the independent set $I^*$. Let $C$ be the leftmost clique. If its value $y_C$ is zero, we move on to the next clique. Otherwise, let $i$ be defining for $C$. We add $i$ to $I^*$ and remove all cliques containing $i$ and all intervals intersecting with it from the problem. Observe that, by the preceding Lemma, $w_{iD} = y_D$ for all cliques $D$ removed and that the intervals removed have their leftmost maximal clique among the removed cliques. In other words, the remaining cliques keep their defining intervals. We continue with the leftmost remaining clique. In this way, we have for every $C$ with positive $y_C$ an interval $i$ in $I^*$ with $w_{iC} = y_C$.

### 9.5. *String matching*

Given a text string $T_0T_1 \dots T_{n-1}$ and a pattern $P_0 \dots P_{m-1}$, the string matching problem is to decide whether the pattern occurs in the text string, i.e., whether there is a position $i$ such that $T_{i+j} = P_j$ for $0 \le j < m$. In such a situation, we say that the pattern occurs with shift i.

For the purpose of certification, if the pattern is found the position is output. A certificate for the contrary case may be given in the form of an array $w$ that indicates for each shift a specific character mismatch. The array $w$ that satisfies $w[i] = \min\{j \mid T_{i+j} \ne P_j\}$ provides exactly that: placing the pattern into the text with a shift of $i$ creates a mismatch at position $i + w[i]$. To verify the validity of the certificate it suffices to check $w[i] \in \{0, \dots, m-1\}$ and $T_{i+w[i]} \ne P_{w[i]}$ for all $i \in \{0, \dots, n-m\}$.

We now show how the Knuth–Morris–Pratt algorithm can be modified to provide such a certificate with its answer. Surprisingly, we could not find this modification in the literature. As is customary, we extend the pattern by a character \$ that does not match any other character, i.e. $P_m =$ \$ and $T_i \ne \$ \ne P_j$ for $0 \le i < n$ and $0 \le j < m$. Assume for the time being that we have at our disposal a function $w'$, where

$$w'(i) = \min\left\{ j \mid P_{i+j} \ne P_j \right\}$$

for $1 \le i \le m$. It will be computed along with the prefix function $\pi$, where

$$\pi(q) = \max\left(\{-1\} \cup \left\{ h \mid h < q \text{ and } P_0 \dots P_h = P_{q-h} \dots P_q \right\}\right).$$

The functions $w'$ and $\pi$ are related. For all $q$ and all $\ell$ with $1 \le \ell < q - \pi(q)$ we have $\ell + w'(\ell) \le q$. Indeed, $P_0 \dots P_{\pi(q)}$ matches $P_{q-\pi(q)} \dots P_q$ and $\pi(q)$ is maximal with this property. Thus if we place $P$ at position $\ell$ of $P$ with $1 \le \ell < q - \pi(q)$, we must have a mismatch before position $q$. Thus $\ell + w'(\ell) \le q$.

$$\begin{array}{cccccccccc} T_0 & T_1 & \dots & T_t & \dots & T_{t+q-\pi(q)-1} & T_{t+q-\pi(q)} & \dots & T_{t+q} & T_{t+q+1} \\ & & & = & \dots & = & = & \dots & = & \ne \\ & & & P_0 & \dots & P_{q-\pi(q)-1} & P_{q-\pi(q)} & \dots & P_q & P_q+1 \\ & & & & & & = & \dots & = & \\ & & & & & & P_0 & \dots & P_{\pi(q)} & \end{array}$$

**Fig. 17 – A typical situation during string matching.**

Assume now the longest prefix of the pattern that matches the substring of the text starting from position $t$ is of length $q$, see Fig. 17. We claim that in this situation we can easily compute the values $w(t+\ell)$ for $\ell \in \{0, 1, \dots, q-\pi(q)-1\}$. First we observe that $w(t) = q+1$ by definition. For $\ell \in \{1, \dots, q-\pi(q)-1\}$ we claim $w(t+\ell) = w'(\ell)$. Indeed aligning $P$ at position $t+\ell$ of the text is the same as aligning it at position $\ell$ of the pattern, at least for the next $q+1$ characters. The mismatch with $P$ occurs at position $\ell + w'(\ell)$ and since this number is at most $q$, the mismatch with $T$ occurs at $t+\ell+w'(\ell)$. Thus $w(t+\ell) = w'(\ell)$.

Similar to the computation of the $\pi$ function, the computation of $w'$ can be done by employing the same algorithm to match the pattern against itself. Recursive calls to $w'$ values will only invoke positions that are smaller and have already been computed.

In the analysis of the running time we would see that for every position $i$ there is exactly one assignment for $w(i)$ whose right hand side involves only a number or a previously known $w$ value. The corresponding statement for $w'$ holds equally, therefore the total running time increases by at most $O(m+n)$.

### 9.6. *Chordal graphs*

A *chord* on a simple cycle is an edge $uv$ that is not an edge of the cycle but whose endpoints are vertices on the cycle. A graph is *chordal* if every simple cycle of length at least four has a chord. The ability to efficiently recognize chordal graphs played a key role in linear time algorithms for recognizing interval graphs, which are chordal, and is discussed at length in [81] and the introductory textbook [80].

To certify that a graph is not chordal, it suffices to point out a chordless cycle. The checker must check that this cycle is, indeed, a cycle of the graph, and it must check that this cycle is chordless. This can be done by marking the vertices and then cycling through the edges to make sure that no edge that is not part of the cycle has two marked endpoints.

**Algorithm 1** KMP-MATCHER

```
1: n ← length[T]; m ← length[P]
2: π ← COMPUTE-PREFIX-FUNCTION(P)
3: q ← −1
4: for i = −1 to n − 2 do
5:    // We have T[i − q] ... T[i] = P[0] ... P[q] and i − q plays the
      role of t in Fig. 17
6:    while q ≥ 0 and P[q + 1] ≠ T[i + 1] do
7:       w(i − q) ← q + 1
8:       for ℓ = 1 to q − π(q) − 1 do
9:          w(i − q + ℓ) ← w′(ℓ)
10:      end for
11:      q ← π[q]
12:   end while            // either q = −1 or P[q + 1] = T[i + 1]
13:   if P[q + 1] = T[i + 1] then
14:      q ← q + 1
15:   else
16:      w(i) ← 0
17:   end if               // T[i + 1 − q] ... T[i + 1] = P[0] ... P[q]
18:   if q = m − 1 then
19:      print "Pattern occurs with shift i − m"
20:   end if
21: end for
```

**Algorithm 2** COMPUTE-PREFIX-FUNCTION

```
1: m ← length[P]
2: π[0] ← −1; q ← −1
3: for i = 0 to m − 1 do          // We have π[i] = q and hence
   P[i − q] ... P[i] = P[0] ... P[q]
4:    while q ≥ 0 and P[q + 1] ≠ P[i + 1] do
5:       w′(i − q) ← q + 1
6:       for ℓ = 1 to q − π(q) − 1 do
7:          w′(i − q + ℓ) ← w′(ℓ)
8:       end for
9:       q ← π[q]
10:   end while            // either q = −1 or P[q + 1] = P[i + 1]
11:   if P[q + 1] = P[i + 1] then
12:      q ← q + 1
13:   else
14:      w′(i) ← 0
15:   end if               // P[i + 1 − q] ... P[i + 1] = P[0] ... P[q]
16:   π(i + 1) = q
17: end for
```

The certificate that a graph $G$ is chordal is similar to the topological sort as a certificate that a graph is a directed acyclic graph. If $(v_1, v_2, \ldots, v_n)$ is an ordering of the vertices of $G$, then the *rightward neighbors* of $v_i$ are those neighbors of $v_i$ that lie to its right in the ordering. The ordering is a *perfect elimination ordering* if the rightward neighbors of every $v_i$ induce a complete subgraph.

A graph is chordal if and only if it has a perfect elimination ordering. Thus, an elimination order can serve as a witness that a graph is chordal.

To understand why, let $G$ be a graph that is not chordal, and suppose $(v_1, v_2, \ldots, v_n)$ is a perfect elimination ordering. Since $G$ is not chordal, it has a chordless cycle $C$ of size at least four. Let $v_i$ be the leftmost vertex of $C$ in the ordering. Then its neighbors $v_j$ and $v_k$ on $C$ lie to its right, and since $C$ has no chord, $v_j$ and $v_k$ are non-adjacent, contradicting the assumption that $(v_1, v_2, \ldots, v_n)$ is a perfect elimination ordering.

It is harder to prove that every chordal graph has a perfect elimination ordering, but this implication is not needed to be convicted that the input graph is chordal.

A linear time algorithm to find a perfect elimination ordering is given in [82]. The obvious checker for the perfect elimination order takes longer to run than it takes to produce the witness. It must check that for each vertex $v_i$, the rightward neighbors of $v_i$ form a complete subgraph, which takes time that is quadratic in the number of these neighbors. Over all $v_i$, a single edge can be checked many times.

In [82] an algorithm is given that checks the witness in $O(n + m)$ time. The trick is to postpone the checks in such a way that each edge is checked only once.

The algorithm traverses the perfect elimination ordering $(v_1, v_2, \ldots, v_n)$. Inductively, $v_i$ has received a list $A(v_i)$ of vertices from its predecessors $(v_1, v_2, \ldots, v_{i-1})$. The algorithm ensures that $v_i$ is adjacent to these vertices, or else declares the witness as invalid. If it passes this test, it then determines the leftmost rightward neighbor $v_j$ of $v_i$ and appends the other rightward neighbors of $v_i$ to $A(v_j)$.

For the correctness, if $(v_1, v_2, \ldots, v_n)$ is not a perfect elimination ordering, then some $v_i$ has rightward neighbors $v_j$ and $v_k$ that are non-adjacent. Suppose without loss of generality that $j < k$. It is easily seen by induction from $i$ to $j$ that if the algorithm has not rejected the ordering by the time that $A(v_j)$ is checked, then $v_k$ is in $A(v_j)$. The algorithm will therefore reject the witness when it discovers that $A(v_j)$ contains a non-neighbor of $v_j$.

The algorithm can be implemented to run in $O(n+m)$ time using elementary methods, such as marking the neighbors of $v_i$ before checking $A(v_i)$, and then unmarking them before moving on to $v_{i+1}$.

The algorithm originally proposed for determining whether a graph is chordal included an algorithm that produces a perfect elimination ordering if a graph is chordal, and an imperfect elimination ordering if it is not [82]. The paper therefore included the above algorithm for checking whether an ordering is a perfect elimination ordering. It did not describe how to find a chordless cycle if the input graph is not chordal. The paper was then followed by a short addendum explaining how to do this [83].

### 9.7. *Numerical algorithms*

Numerical algorithms are usually implemented in floating point arithmetic. As floating point arithmetic incurs round-off error, numerical computations do not always yield good approximations of the true result. Almost every textbook in numerical analysis contains warning examples. The field of validated numerical computations [84,85] addresses this issue and develops methods that deliver rigorous results. It is beyond the scope of this paper to elaborate on validated numerical computations and so we confine ourselves with a simple example.

Let $a$ be a positive real number. We have some method for computing square roots, say the method returns $x_0$. How good is $x_0$? The following estimate is useful:

$$|\sqrt{a} - x_0| = \frac{a - x_0^2}{\sqrt{a} + x_0} \leq \begin{cases} \dfrac{a - x_0^2}{1 + x_0} & \text{if } a \geq 1 \\ \dfrac{a - x_0^2}{a + x_0} & \text{if } a < 1. \end{cases}$$

As a concrete example, let us estimate the distance of $\sqrt{2.5}$ and 1.58. We have

$$\left|\sqrt{2.5} - 1.58\right| \leq \frac{2.5 - 2.4964}{1 + 1.58} = \frac{0.0036}{2.58} \leq 0.0015.$$

### 9.8. *Guide to literature*

Frequently, algorithms research that is performed with efficiency in mind leads implicitly to methods suitable to certify the output. For various algorithmic problems however, specific algorithms that allow for certification had to be and have been designed. We briefly survey some examples.

*Graph recognition problems:* For various graph classes certified recognition algorithms exist. Among these classes are the interval and permutation graphs [6], the circular and unit circular arc graphs [86], the proper Helly circular arc graphs [87], the HHD-free graphs [88], and the co-graphs [89] (for which there is also a dynamic version [90]). Proper interval graphs are treated in [91] and [92], the latter also considers bipartite permutation graphs. Further certified recognition algorithms for several hereditary graph classes are given in [93].

Connectivity and two-connectivity are easily certified in linear time. A spanning tree certifies connectivity, a cut-vertex certifies non-two-connectivity, and every s-t-numbering, open ear decomposition and bipolar orientation certifies two-connectivity [94]. For triconnectedness of graphs, linear decision algorithms are known [26,27]. The fastest certifying algorithm runs in quadratic time (See Section 5.4 and [52]). A linear time certifying algorithm for graphs for which a Hamiltonian cycle is known is available [55]; this assumes that the Hamiltonian cycle is part of the input. For arbitrary $k$, the situation is as follows. In linear time any $k$-connected graph can be sparsified to a $k$-connected graph with $O(kn)$ edges [53]. Of course, certifying $k$-connectivity of the sparsification certifies $k$-connectivity of the original graph. Also, it is easy to check whether a vertex cut in the sparsification is also a vertex cut in the original graph. $k$-connectivity can be tested in time $O(m + \min(kn^2, k^2n^{3/2}))$ [95,96]; the algorithms are certifying. The certificates are flows. Linial et al. [97] gave a geometric characterization of $k$-connectivity for general $k$. A graph $G$ is $k$-connected if and only if, specifying any $k$ vertices of $G$, the vertices of $G$ can be represented as points in $\mathbb{R}^{k-1}$ so that no $k$ are on a hyperplane and each vertex is in the convex hull of its neighbors, except for the $k$ specified vertices. The characterization gives rise to an $O(n^{5/2} + nk^{5/2})$ time Monte Carlo algorithm for $k$-connectivity and an $O(kn^{5/2} + nk^{7/2})$ Las Vegas algorithm for $k$-connectivity. The algorithms are certifying.

*Permutation groups:* In [98] certifying algorithms for computational problems involving permutation groups given by generators are considered. More specifically, the problems considered are deciding membership, subgroups, computing orbits, the Schreier tree, stabilizers, bases, and computing the order of the given permutation group.

*Geometric problems:* In [99] a certifying algorithm for minimally rigid planar graphs is given. The papers [76] and [7] describe methods for certifying convexity of polyhedra and convex hulls (see Section 9.1) and various types of planar subdivisions, such as triangulations, Delaunay triangulations, and convex subdivisions. Another application of local to global principles for certification of convexity can be found in [100].

*Miscellaneous:* Feige and Krauthgamer [101] shows how to certify a large hidden clique in a semi-random graph and McConnell [102] explains how to certify whether a matrix has the consecutive ones property. Certification of various basic graph algorithms (with and without witnesses) are discussed in [103].

*Implementations:* Implementations of many certifying algorithms and the corresponding checkers are discussed in [20]. For the algorithm library LEDA the concept of certification has been and is one of the guiding principles.

## 10. Randomization

In the preceding sections, we considered deterministic certifying algorithms and deterministic checkers. In this section we consider randomization. We first explain that deterministic checkers turn Monte Carlo algorithms into Las Vegas algorithms. We then give three examples of randomized certification. The first two examples have a checker that does not require a witness. In the third example the provided witness allows for a faster running time of the randomized checker. Finally, we extend Section 5 to randomized algorithms.

### 10.1. *Monte Carlo algorithms resist deterministic certification*

Consider a Monte Carlo algorithm for a function $f$, i.e., a randomized algorithm which on input $x$ outputs $f(x)$ with probability at least 3/4 and may output anything otherwise. The running time is bounded by $T(|x|)$.

Assume now that there were an efficient certifying algorithm $Q$ with the same complexity: on input $x$ algorithm $Q$ outputs a triple $(x, y, w)$ passing the witness predicate $\mathcal{W}$ with probability at least 3/4 and may output anything else otherwise. It has running time $O(T(|x|))$. Further assume there is a deterministic checker $C$ that checks the triple $(x, y, w)$ in time $O(T(|x|))$. We can turn $Q$ into a Las Vegas algorithm for $f$ as follows: We first run $Q$. If the triple $(x, y, w)$ returned by $Q$ passes $\mathcal{W}$, we return $y$. Otherwise, we rerun $Q$.

The resulting randomized algorithm always returns the correct result $f(x)$ and has an expected running time in $O(T(|x|))$. Observe that each round has an expected running time in $O(T(|x|))$ since both the algorithm $Q$ and the checker $C$

run within this time. Also observe that the expected number of rounds is constant, since the success probability is at least 3/4.

Does this observation imply that Monte Carlo algorithms are inherently unsafe? No, it only says that the concept of deterministic certification does not apply to them. Program verification does apply and Monte Carlo algorithms are frequently quite simple so that verification may be(come) feasible. Another option is to deviate from the deterministic checkers by allowing randomization and error. We describe three examples in the next three subsections. The first two examples demonstrate use of randomized certification without the help of a witness, whereas the third example makes use of a witness to provide randomized certification of the output of a randomized algorithm.

### 10.2. *Integer arithmetic*

To check the result of a multiplication computation, the following method used to be taught in German high-schools and is already described by Al-Kharizmi in his book on algebra. It is known as "Neunerprobe" in German, "casting out nines" in English, and "preuve par neuf" in French.

We want to check whether $c = a \cdot b$. We form the repeated digit sums $s_a$, $s_b$ and $s_c$ of $a$, $b$ and $c$. The digit sum is formed by adding the digits of a number. If the result is a multi-digit number, the process is repeated, until one arrives at a one digit number. For example,

$4572 \to 18 \to 9$.

Then one multiplies $s_a$ and $s_b$ with a two-digit result; let $s$ be the digit sum of the result. If $s \neq s_c$, then $c$ is not equal to $a \cdot b$. If $s = s_c$, then $c$ may or may not be equal to $a \cdot b$.

The digit sum of a positive integer $a$ is nothing but the remainder of $a$ modulo 9 (with representatives 1 to 9 instead of 0 to 8). This follows from $10^k \bmod 9 = 1$ for all $k \geq 1$. Thus casting out nines rests on the statement

if $c = a \cdot b$ then $c \bmod 9 = ((a \bmod 9) \cdot (b \bmod 9)) \bmod 9$.

Of course, a similar statement holds for any integer $q$ instead of 9, i.e.,

if $c = a \cdot b$ then $c \bmod q = ((a \bmod q) \cdot (b \bmod q)) \bmod q$.

Let $s_a = a \bmod q$ and define $s_b$ and $s_c$ analogously. Let $s = (s_a \cdot s_b) \bmod q$. If $s \neq s_c$, then $c \neq a \cdot b$. If $s = s_c$, $q$ divides $c - a \cdot b$. The number of distinct prime divisors of $c - a \cdot b$ is bounded. Thus, if we choose $q$ from a sufficiently large set $P$ of primes, the test will show $c \neq a \cdot b$ with high probability. The following theorem quantifies these statements.

**Theorem 11.** *Let $a$, $b$, and $c$ be positive integers bounded by $2^{2^k}$, with $k \geq 5$. If $c \neq a \cdot b$ then*

1. *for any integer $d > 0$ the probability that a prime number $x$ taken uniformly at random from the primes within $\{2, \ldots, 2^{k+2d+3} - 1\}$ divides $c - a \cdot b$ is at most $1/2^d$,*
2. *the probability that an integer $x$ taken uniformly at random from $\{1, \ldots, 2^{k+5} - 1\}$ does not divide $c - a \cdot b$ is at least $1/(2 \cdot (k + 5) \ln 2)$, and*
3. *the probability that at least one of $L = 2k + 10$ integers $x_1, x_2, \ldots, x_L$ taken uniformly at random from $\{1, \ldots, 2^{k+5} - 1\}$ does not divide $c - a \cdot b$ is at least 1/2.*

**Proof.** (1) The absolute value of $c - ab$ is bounded by $2^{2^{k+1}+1}$. To show the statement, we will show that there are sufficiently many prime numbers in the interval $\{2, \ldots, 2^{k+3+2d} - 1\}$, and then, by bounding the number of prime factors of any number no larger than $2^{2^{k+1}+1}$, we will show that most of these prime numbers are not prime factors of $|c - ab|$.

There are at least $2^{k+3+2d}/((k+3+2d)\ln 2)$ prime numbers which are contained in the interval $\{2, \ldots, 2^{k+3+2d} - 1\}$. (This is in accordance with the prime number theorem that there are approximately $x/\ln(x)$ prime numbers smaller than $x$ and follows from Dusart's bound [104], since $2^{k+3+2d} > 599$). On the other hand for any integer $x$ that has $\ell$ distinct prime factors, it must be the case that $\ell! \leq n$. It suffices for us to show that the number of primes in $\{2, \ldots, 2^{k+3+2d} - 1\}$ is by a factor of $2^d$ larger than the number of distinct prime factors of $|c - ab|$. Thus it suffices to show that

$$\left(\frac{2^{k+3+2d}}{2^d \cdot (k+3+2d)\ln 2}\right)! \geq |c - ab|.$$

Since $2 \ln 2 < 2$, for $k \geq 1$ it is true that

$$2^{k+3+2d} \geq (2^{k+1} + 1) \cdot 2^{2d} \cdot (2 \ln 2),$$

thus

$$\frac{2^{k+3+2d}}{2^{2d} \cdot 2 \ln 2} \geq 2^{k+1} + 1.$$

By a simple expansion of the fraction it follows that

$$\frac{2^{k+3+2d}}{2^d \cdot (k+3+2d)\ln 2} \cdot \frac{(k+3+2d)}{2^d 2} \geq 2^{k+1} + 1. \tag{17}$$

Since $k \geq 1$ and $d \geq 1$, we have $(k + 3 + 2d)/(2^d 2) \leq \log(2^{k+3+2d}/(2^d \cdot (k+3+2d)e \ln 2))$: Indeed, the inequality holds for $k = 1$ and $d = 1$. Furthermore, for $k, d \in \mathbb{R}_{\geq 1}$, the partial derivatives with respect to $k$ and $d$ of the left side of the equation are smaller than the respective partial derivatives of the right side of the equation. We can thus replace the second factor on the left side of Eq. (17) and it follows that

$$\frac{2^{k+3+2d}}{2^d \cdot (k+3+2d)\ln 2} \cdot \log\left(\frac{2^{k+3+2d}}{2^d \cdot (k+3+2d)e \ln 2}\right) \geq 2^{k+1} + 1.$$

Stirling's inequality $x! \geq (x/e)^x$ implies $\log(x!) \geq x \log(x/e)$, applying this we proceed to obtain the desired inequality

$$\left(\frac{2^{k+3+2d}}{2^d \cdot (k+3+2d)\ln 2}\right)! \geq 2^{2^{k+1}+1} \geq |c - ab|.$$

(2) The second statement follows directly from the first by again noting that for $k \geq 11$ and $d = 1$ the number of primes in $\{2, \ldots, 2^{k+5} - 1\}$ is at least $2^{k+5}/((k+5)\ln 2)$.

(3) For any $i$, $1 \leq i \leq L$, the probability that $x_i$ divides $c - ab$ is at most $1 - 1/(2(k+5)\ln 2)$ by the second statement. Therefore, the probability that all $x_i$'s divide $c - ab$ is at most

$$\left(1 - \frac{1}{2(k+5)\ln 2}\right)^L = e^{L \cdot \ln\left(1 - \frac{1}{2(k+5)\ln 2}\right)} \leq e^{-\frac{L}{2(k+5)\ln 2}} \leq e^{-\ln 2} = \frac{1}{2}. \quad \square$$

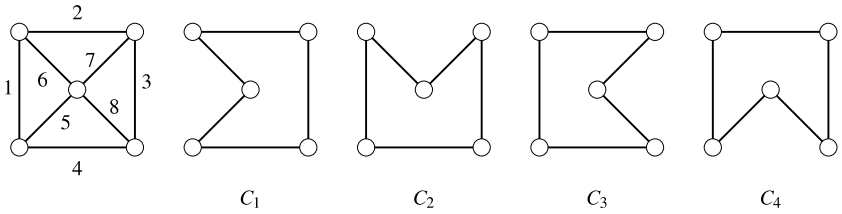



**Fig. 18 – The leftmost figure shows a graph with eight edges. The next four figures show four circuits in this graph. The circuit $C_1$ has the vector representation $C_1 = (0, 1, 1, 1, 1, 1, 0, 0)$. Let $D = (1, 1, 1, 1, 0, 0, 0, 0)$ be the circuit formed by the edges 1 to 4. Then $D = C_1 + C_2 + C_3 + C_4$. The set $\{C_1, C_2, C_3, C_4\}$ is a cycle basis. Its weight is equal to three times the sum of the weights of edges 1 to 4 plus 2 times the weight of edges 5 to 8.**

The asymptotic complexity of division is the same as the one of multiplication. Intuitively, multiplication is simpler than division. We can check a division $a/b = c$ by checking whether $a = c \cdot b$.

### 10.3. *Matrix operations*

Given three $n \times n$ matrices $A$, $B$, and $C$ over $\mathbb{Z}_2$, we want to verify that $AB = C$. There is an obvious way to check the equality: compute the product of $A$ and $B$ and compare the result entry by entry with $C$. This takes time $O(n^\omega)$, where $\omega$ is the exponent of matrix multiplication [105].

Already in 1977, Freiwalds [106] described a randomized algorithm for verifying matrix multiplication. The algorithm is simple. We generate a random vector $x \in \mathbb{Z}_2^n$ and compute $y = A(Bx) - Cx$. If $y$ is the zero vector, we accept the input, i.e., believe that $C = AB$, otherwise we state that $C \neq AB$. The computation of $y$ takes three matrix–vector products ($Bx$, $A(Bx)$, and $Cx$) and one vector addition and hence takes time $O(n^2)$.

Let $X = AB - C$. Then $y = Xx$. If $AB = C$ and hence $X = 0$, we have $y = 0$ for any choice of $x$. If $y \neq 0$, then $X \neq 0$ and hence $AB \neq C$. It remains to estimate the probability that $X \neq 0$ and $Xx = 0$.

**Lemma 18** ([106]). *Let $z$ be a nonzero $n$-vector over $\mathbb{Z}_2$. Then*

$$\text{prob}(z^T x \neq 0) = 1/2,$$

*where $x$ is a random $n$-vector over $\mathbb{Z}_2$.*

**Proof.** Since $z$ is nonzero, there is an $\ell$ with $z_\ell = 1$. Then

$$z^T x = x_\ell + \sum_{j \neq \ell} z_j x_j$$

and hence for any choice of the $x_j$'s, $j \neq \ell$, there is a exactly one choice for $x_\ell$ such that $z^T x \neq 0$. □

**Theorem 12** ([106]). *Let $X$ be a nonzero $n \times n$ matrix over $\mathbb{Z}_2$. Then*

$$\text{prob}(Xx \neq 0) \geq 1/2$$

*where $x$ is a random $n$-vector over $\mathbb{Z}_2$.*

**Proof.** Since $X$ is nonzero, it has at least one nonzero row. We apply the previous Lemma to this row. □

For further information on certification of matrix products see [107]. We now turn to an example that requires a witness to allow for the randomized certification.

### 10.4. *Cycle bases*

We discuss the certification of minimum weight cycle bases of undirected graphs. The fastest known algorithm for computing a minimum weight basis is a Monte Carlo algorithm with running time $O(m^\omega)$, where $\omega$ is the exponent of matrix multiplication [108]. The algorithm can be made certifying and the witness can be checked in Monte Carlo time $O(m^2)$. We describe the required modifications and show how to check the witness. We refer the reader to [109] for background information on cycle bases.

Let $G = (V, E)$ be a connected graph with $n$ vertices and $m$ edges and let $w : E \to \mathbb{R}_{>0}$ be a positive weight function on the edges of $G$. All results of this section also hold for nonnegative weight functions, but some of the arguments are shorter for positive weight functions. A *cycle* in $G$ is an even subgraph of $G$ and a *circuit* is an even connected subgraph with all vertices having degree two, see Fig. 18. The weight $w(C)$ of a cycle is the sum of the weights of its edges. We represent cycles as vectors in $\mathbb{Z}_2^E$; the coefficient corresponding to an edge is 1 if and only if the edge belongs to the cycle. The space of all cycles is then a vector space of dimension $\nu = m - (n - 1)$; observe that the addition of two cycles corresponds to the symmetric difference of their edge sets and hence yields a cycle. A set $\mathcal{B} = \{C_1, \ldots, C_\nu\}$ of cycles is a cycle basis if any cycle $C$ can be written as a linear combination $C = \sum_i \lambda_i C_i$ with $\lambda_i \in \mathbb{Z}_2$ for all $i$. Let $T$ be a spanning tree of $G$. For any non-tree edge $e$ let $C_e$ be the circuit formed by $e$ plus the path in $T$ connecting the endpoints of $e$. There are $m - n + 1$ such cycles. They are independent and form a basis; this is called the fundamental cycle basis. The weight of a basis is the sum of the weights of the cycles comprising the basis. A minimum weight basis is a basis of minimum weight.

The following lemma defines a certificate for minimum weight bases. For its proof, we first observe that every graph has a minimum cycle basis consisting only of circuits: Indeed, if there is a cycle $C$ in a basis $\mathcal{B}$ that is not a circuit, then $C$ is the union of two cycles $C'$ and $C''$ of smaller weight. Then $\mathcal{B} - C + C'$ and $\mathcal{B} - C + C''$ have smaller weight than $\mathcal{B}$ and at least one of them is a basis.

**Lemma 19** ([110]). *A set of cycles $\{C_1, \ldots, C_\nu\}$ is a minimum cycle basis if there are vectors $S_1, \ldots, S_\nu \in \{0, 1\}^E$ such that*[11]

(a) $\langle S_i, C_j \rangle = 0$ *for* $1 \leq i < j \leq \nu$,

[11] $\langle\ ,\ \rangle$ denotes the inner product of vectors.

(b) $\langle S_i, C_i\rangle = 1$,
(c) $C_i$ *is a shortest circuit with* $\langle S_i, C\rangle = 1$.

**Proof.** Let $k$ be maximal such that $\mathcal{B}_k = \{C_1, \ldots, C_k\}$ is contained in some minimum cycle basis and assume, for the sake of a contradiction, that $k < \nu$. Let $\mathcal{B}' = \{C_1, \ldots, C_k, D_{k+1}, \ldots, D_\nu\}$ be a minimum cycle basis extending $\mathcal{B}_k$. Then

$$C_{k+1} = \sum_{i\le k} \lambda_i C_i + \sum_{i>k} \lambda_i D_i \tag{18}$$

and hence

$$1 = \langle S_{k+1}, C_{k+1}\rangle = \sum_{i>k} \lambda_i \langle S_{k+1}, D_i\rangle.$$

Thus there must be an $\ell > k$ with $\lambda_\ell = \langle S_{k+1}, D_\ell\rangle = 1$. By (c), $w(C_{k+1}) \le w(D_\ell)$. Solving (18) for $D_\ell$ gives a representation of $D_\ell$ in terms of $\mathcal{B}'' = \mathcal{B}' \setminus \{D_\ell\} \cup \{C_{k+1}\}$. We conclude that $\mathcal{B}''$ is a basis of weight no larger than $\mathcal{B}'$. Thus there is a minimum basis extending $\mathcal{B}_{k+1}$. □

We next describe a probabilistic check of conditions (a), (b), and (c) that operates in time $O(m^2)$. The following observation will be useful.

**Lemma 20** ([108]). *Let* $A_1, \ldots, A_k, S \in \mathbb{Z}_2^E$. *Then*

$$\langle A_i, S\rangle = 1 \text{ for some } i,\ 1 \le i \le k$$
$$\Longrightarrow \text{prob}\left(\left\langle \sum_{1\le i\le k} \lambda_i A_i, S\right\rangle = 1\right) = 1/2,$$

*where the* $\lambda_i$, $1 \le i \le k$, *are chosen independently and uniformly in* $\mathbb{Z}_2$.

**Proof.** Assume $\langle A_\ell, S\rangle = 1$. Then for any choice of the $\lambda_i$, $i \ne \ell$, there is exactly one choice for $\lambda_\ell$ such that $\langle \sum_{1\le i\le k} \lambda_i A_i, S\rangle = 1$, namely $\lambda_\ell = 1 + \langle \sum_{i\ne\ell} \lambda_i A_i, S\rangle$.

Alternatively, we observe that

$$\left\langle \sum_{1\le i\le k} \lambda_i A_i, S\right\rangle$$

is the inner product of a random vector $(\lambda_1, \ldots, \lambda_k)$ with the nonzero vector $(\langle A_1, S\rangle, \ldots, \langle A_k, S\rangle)$ and then appeal to Lemma 18. □

We first show how to verify properties (a) and (b). Let $A$ be matrix whose rows are the vectors $S_1^T$ to $S_\nu^T$ and let $B$ be the matrix whose columns are the cycles $C_1$ to $C_\nu$. We need to verify that $E := AB$ is a lower-diagonal matrix with ones on the diagonal. Property (b) is easily verified in time $O(\nu m) = O(m^2)$. We simply compute the products $\langle S_i, C_i\rangle$ for $1 \le i \le \nu$. We turn to property (a). The solution lies in Lemma 20. Consider the above-diagonal elements in the $j$th column, i.e., the elements $E_{i,j}$ for $i < j$. One of these elements is nonzero if and only if one of the vectors $S_1$ to $S_{j-1}$ is non-orthogonal to $C_j$. Lemma 20 is a probabilistic test for this property. We choose random numbers $\lambda_i \in \mathbb{Z}_2$, $1 \le i \le n$, and form the vectors

$$R_j = \sum_{i<j} \lambda_i S_i, \quad 1 \le j \le n.$$

These vectors can be computed in total time $O(nm)$. We next form the products

$$\langle R_j, C_j\rangle, \quad 1 \le j < n.$$

If one of these products is nonzero, $C$ is not a lower-diagonal matrix. Conversely, if $C$ is not a lower-diagonal matrix, then with probability at least one-half, one of these products is nonzero by Lemma 20.

We next turn to condition (c). A circuit $C$ is *isometric* if for any two vertices $u$ and $v$ in $C$, a shortest path connecting $u$ and $v$ is contained in $C$.

**Lemma 21** ([111]). *A minimum cycle basis* $\mathcal{B}$ *consists only of isometric circuits.*

**Proof.** We argue by contradiction. Suppose $C$ is not an isometric cycle but contained in a minimum cycle basis $\mathcal{B}$. Then there are vertices $u$ and $v$ in $C$ such that $C$ does not contain a shortest path connecting $u$ and $v$. Let $p$ be a shortest path connecting $u$ and $v$ and split $C$ at $u$ and $v$ into $C_1$ and $C_2$. Consider $C' = C_1 + p$ and $C'' = C_2 + p$. Both cycles are cheaper than $C$ and either $\mathcal{B} - C + C'$ or $\mathcal{B} - C + C''$ is a basis. □

The argument of the proof also shows that any cycle that fulfills condition (c) in Lemma 19 is an isometric circuit. To certify condition (c), we assume for simplicity from now on that the shortest path between every two vertices of the input graph is unique; such a situation can be simulated by adding a random infinitesimally small weight to every edge. For further details see [108].

**Lemma 22** ([108]). *Let* $C$ *be an isometric circuit. Then for each* $v \in C$ *there is an edge* $e = xy \in C$ *such that* $C$ *consists of the shortest paths from* $v$ *to* $x$ *and* $y$ *and the edge* $e$.

**Proof.** Consider any edge in $e = xy$ of $C$. Splitting $C \setminus e$ at $v$ gives us a path $q_{v,x}$ and a path $q_{v,y}$. There is a choice of $e$ (there might be two) such that both paths have weight at most $w(C)/2$. Since $C$ is isometric, it contains a shortest path connecting $v$ to $x$ and a shortest path connecting $v$ to $y$. These paths must be $q_{v,x}$ and $q_{v,y}$, respectively. □

We use the following witness for supporting the check of condition (c).

- For each vertex $v$, a shortest path tree $T_v$ rooted at $v$.
- A list $L = (D_1, \ldots, D_N)$ of circuits that allegedly contains all isometric circuits. The circuits are sorted by weight and the total size (=number of edges) of the circuits in $L$ is at most $nm$.
- For an edge $e = xy$ and a vertex $v$ let $C_{v,e} = p_{vx} + e + p_{vy}$, where $p_{vx}$ is the path from $v$ to $x$ in $T_v$. For each $e$ and $v$, the algorithm provides a link to a circuit $D \in L$ with $D = C_{v,e}$ or a proof that $C_{v,e}$ is non-isometric. The proof is a pair $(a, b)$ of vertices on $C_{v,e}$ such that the shortest path connecting $(a, b)$ is not part of $C_{v,e}$.

The algorithm in [108] computes such a list $L$ and the additional information stipulated above. Verification is as follows.

We first verify that the trees are shortest paths trees as described in Section 2.4; we then verify that all elements of $L$ are circuits, and that the total size of the circuits in $L$ is $nm$. We then choose for each edge $e$ a random label $\ell(e) \in \mathbb{Z}_p$ for a prime $p$ ($p = 2$ is sufficient) and precompute for each $v$ and each node $x$ of $T_v$, the sum of the labels of the edges on the path from $v$ to $x$ and for each $D$ in $L$ the sum of the labels of the edges in $D$. This takes time $O(nm)$. For each $e = xy$ and $v$ we perform the following test:

- If $C_{v,e}$ is linked to $D$ in $L$ we compute the label of $C_{v,e}$ (as the sum of $\ell(e)$ and the labels of the endpoints of $e$ in $T_v$) and compare it to the label of $D$. If the labels are different, we reject. The probability of failure to detect $C_{v,e} \neq D$ is bounded by $1/p$.[12] (This test was proposed in [112] as a general method for verifying equality of sets.)
- If $C_{v,e}$ is claimed to be non-isometric and $(a, b)$ is provided as a proof we verify that $a$ and $b$ lie on the paths $p_{v,x} \cup p_{v,y}$ where $e = xy$; say $a$ lies on the former path and $b$ lies on the latter. If $a$ and $b$ would lie on the same path, we reject, since subpaths of shortest paths are shortest. We then compute the lengths of the two paths in $C_{v,e}$ connecting $a$ and $b$ (one is $w(p_{v,a}) + w(p_{v,b})$ and one is $w(e) + w(p_{a,x}) + w(p_{b,y})$) and verify that these length are larger than $w(p_{a,b})$. If not, we reject.

We have now verified that $L$ contains all isometric circuits. It remains to verify that the circuits selected satisfy (c). Let $L = \{D_1, \ldots, D_N\}$ and recall that $L$ is sorted in order of increasing weight. We choose a random $\lambda_j \in \{0, 1\}$ for each $j$ and for each $\ell$, $1 \leq \ell \leq N$ form the sum $\sum_{j \leq \ell} \lambda_j D_j$. For each $i$, $1 \leq i \leq \mu$, let $C_i = D_{\pi(i)}$. Verify

$$\langle C_i, S_i \rangle = 1 \quad \text{and} \quad \left\langle \sum_{j < \pi(i)} \lambda_j D_j, S_i \right\rangle = 0.$$

By Lemma 20, the latter test fails with probability 1/2 if there is a $j < \pi(i)$ with $\langle D_j, S_i \rangle = 1$.

**Theorem 13.** *The witness for minimum cycle bases defined above can be checked in probabilistic time $O(m^2)$. The minimum cycle basis algorithm of* [108] *can compute this witness without loss of efficiency.*

### 10.5. Definitions

We extend the definitions and theorems of Section 5 to randomized algorithms. A *probabilistically checkable strong witness predicate* for an I/O-specification $(\varphi, \psi)$ is a predicate $\mathcal{W} : X \times Y^{\perp} \times W$ satisfying the following properties:

Strong witness property: Let $(x, y, w) \in X \times Y^{\perp} \times W$ satisfy the witness predicate. If $y = \perp$, $w$ proves that $x$ does not satisfy the precondition and if $y \in Y$, $w$ proves that $(x, y)$ satisfies the postcondition, i.e.,

$$\forall x, y, w \quad \begin{array}{l} (y = \perp \wedge \mathcal{W}(x, y, w)) \Longrightarrow \neg\varphi(x) \quad \text{and} \\ (y \in Y \wedge \mathcal{W}(x, y, w)) \Longrightarrow \psi(x, y). \end{array} \tag{19}$$

Randomized checkability: For a triple $(x, y, w)$ there is a trivial way to determine the value $\mathcal{W}(x, y, w)$ with high probability. I.e., there is a trivial randomized algorithm that computes $\mathcal{W}(x, y, w)$ correctly with probability at least 3/4 and this bound on the error probability is trivial to understand.

Simplicity: The implications (19) have a simple proof.

[12] Let $C$ and $D$ be subsets of $E$ with $C \neq D$. Let $e_0 \in C \oplus D$ be an edge in the symmetric difference of $C$ and $D$. Then for each choice of labels of the edges in $E \setminus e$, there is exactly one choice for $\ell(e_0)$ such that $\sum_{e \in C} \ell(e) = \sum_{e \in D} \ell(e)$.

The first and the last item are as in Section 5. Observe that the checker is allowed to reject a correct output and witness or accept a wrong output and witness with probability 1/4. The latter is important; otherwise the construction of Section 10.1 would apply.

A *randomized strongly certifying algorithm* for I/O-specification $(\varphi, \psi)$ and probabilistically checkable strong witness predicate $\mathcal{W}$ is an algorithm with the following properties:

- It halts for all inputs $x \in X$.
- On input $x \in X$ it outputs a $y \in Y^{\perp}$ and a $w \in W$ such that $\mathcal{W}(x, y, w)$ holds with probability at least 7/8.

The simplicity and checkability consideration of Section 5.5 apply.

The definition above captures all examples given in this section. We illustrate this for our last example: minimum weight cycle bases. The input $x$ is an edge-weighted undirected graph and $y$ is a set $\{C_1, \ldots, C_\nu\}$ of $\nu = m - n + 1$ circuits in $G$. The witness consists of vectors $S_1, \ldots, S_\nu$ of vectors in $\{0, 1\}^E$, a list $L = \{D_1, \ldots, D_N\}$ of circuits, and some more stuff. In order for the witness property to hold, the list $L$ must contain all isometric circuits, $\langle S_i, C_j \rangle = 0$ for $i < j$, and $C_i$ is a shortest circuit on $L$ whose inner product with $S_i$ is one. The checker may fail to verify the latter property (and the algorithm may fail to deliver a set of circuits with the latter property); in this situation, $\{C_1, \ldots, C_\nu\}$ is a basis, but not a minimum weight basis.

We next define the notions of a *randomized certifying algorithm* and *randomized weakly certifying algorithm*. A probabilistically checkable witness predicate for an I/O-specification $(\varphi, \psi)$ is a predicate $\mathcal{W} : X \times Y^{\perp} \times W$ satisfying the following properties.

Witness property: Let $(x, y, w) \in X \times Y^{\perp} \times W$ satisfy the witness predicate. If $y = \perp$, $w$ proves that $x$ does not satisfy the precondition and if $y \in Y$, $w$ proves that either $x$ does not satisfy the precondition or $(x, y)$ satisfies the postcondition, i.e.,

$$\forall x, y, w \quad \begin{array}{l} (y = \perp \wedge \mathcal{W}(x, y, w)) \Longrightarrow \neg\varphi(x) \quad \text{and} \\ (y \in Y \wedge \mathcal{W}(x, y, w)) \Longrightarrow \neg\varphi(i) \vee \psi(x, y). \end{array} \tag{20}$$

Randomized checkability: For a triple $(x, y, w)$ there is a trivial way to determine the value $\mathcal{W}(x, y, w)$ with high probability. I.e., there is a trivial randomized algorithm that computes $\mathcal{W}(x, y, w)$ correctly with probability at least 3/4 and this bound on the error probability is trivial to understand.

Simplicity: The implications (20) have a simple proof.

Items 1 and 3 are as in Section 5.

A randomized certifying algorithm for I/O-specification $(\varphi, \psi)$ is a randomized algorithm with the following property: For all inputs $x \in X$, it halts and outputs a $y \in Y^{\perp}$ and a $w \in W$ such that $\mathcal{W}(x, y, w)$ with probability at least 7/8.

A randomized weakly certifying algorithms for I/O-specification $(\varphi, \psi)$ is a randomized algorithm with the following properties:

- For inputs $x \in X$ satisfying the precondition, it halts with probability at least 7/8.

- If it halts on $x \in X$ for some choice of randomness, it halts on $x$ and outputs a $y \in Y^{\perp}$ and a $w \in W$ such that $\mathcal{W}(x, y, w)$ with probability at least 7/8.

An early version of the definition of randomized certification can be found in [113], where also complexity classes based on randomized certifiability are defined. For a restricted computation model it is shown that the gap in the best running times between a randomized and a deterministic certifying algorithm can be arbitrarily large.

We next generalize Theorem 7; Theorem 5 is the special case $\varphi = T$ and hence also generalizes. Let $(\varphi, \psi)$ be an I/O-specification and let $P$ be a randomized program (in a programming language $L$ with well-defined semantics) with I/O behavior $(\varphi, \psi)$.

We need to make an additional assumption on $P$, namely that $P$ computes a partial function. Let $f : X \to Y$ be a partial function. $P$ computes $f$ if for any $x \in X$, $f(x)$ is defined if and only if $P$ halts on $x$ for some choice of randomness. Moreover, if $f(x)$ is defined, $P$ outputs $f(x)$ with probability at least 7/8. We comment below, why the assumption that $P$ computes a function is needed. We assume that we have a proof (in some formal system $S$) of the following statement:

$$\begin{aligned}&\text{if } P \text{ halts on } x \text{ for some randomness,}\\ &\text{it halts on } x \text{ with probability at least 7/8 and}\\ &\text{there is a value } f(x) \text{ that is output with probability}\\ &\quad\text{at least 7/8 and satisfies } \neg\varphi(x) \vee \psi(x, f(x)) \quad \text{and}\\ &\varphi(x) \Longrightarrow P \text{ halts on input } x \text{ with probability at least 7/8.}\end{aligned} \tag{21}$$

We use $w_2$ to denote the proof. We extend $P$ to a program $Q$ which on input $x$ does the following: If $P$ halts on input $x$, $Q$ outputs $P(x)$ and a witness $w = (w_1, w_2)$ where $w_1$ is the program text $P$ and $w_2$ is a proof for (21).

The witness predicate $\mathcal{W}(x, y, w)$ holds if $w$ has the required format, i.e., $w = (w_1, w_2)$, where $w_1$ is the program text of some program $P$, $w_2$ is a proof for (21), and $y = f(x)$, where $f$ is the partial function defined by $P$.

The following randomized algorithm $C$ decides the witness predicate. On input $(x, y, w)$, it first checks that $w_2$ is a proof of (21) where $P$ is given by $w_1$. It then runs $P$ on $x$ for $k$ different choices of randomness (the $k$ runs are performed in parallel); $k$ will be fixed below. If $k/2+1$ of the runs return $y$, the checker stops and accepts. Otherwise, it rejects or runs forever.

1. $\mathcal{W}$ has the witness property: If $\mathcal{W}(x, y, w)$, (21) holds and hence $P$ computes a function; call it $f$. Also, $y = f(x)$ by the definition of $\mathcal{W}$.
2. $C$ decides $\mathcal{W}$: Since $C$ checks the proof $w_2$, we may assume that $P$ satisfies (21). If $P$ never halts on $x$, $C$ diverges. If $P$ halts on $x$ for some randomness, it halts on $x$ and returns $f(x)$ with probability at least 7/8. The probability that, among $k$ runs of $P$, $k/2$ or more produce an output different from $f(x)$ is at most

$$\sum_{i \geq k/2} \binom{k}{i} \left(\frac{1}{8}\right)^{i} \left(\frac{7}{8}\right)^{k-i} \leq \sum_{i} \binom{k}{i} \left(\frac{1}{8}\right)^{k/2}$$
$$\leq 2^{k} \left(\frac{1}{4}\right)^{k/2} \left(\frac{1}{2}\right)^{k/2} = \left(\frac{1}{2}\right)^{k/2} \leq 1/4$$

   where the last inequality holds for $k \geq 4$. Thus if $y = f(x)$, $\text{prob}(\mathcal{W}(x, y, w)) \geq 3/4$ and if $y \neq f(x)$, $\text{prob}(\mathcal{W}(x, y, w)) \leq 1/4$. Thus $C$ decides $\mathcal{W}$.
3. Simplicity: The arguments in the preceding items are straightforward.
4. Efficiency: The running times of $Q$ and $C$ are asymptotically no larger than the expected running time of $P$. Consider a particular $x$ on which $P$ halts for some choice of randomness. Then $P$ halts on $x$ with probability at least 7/8. Let $T$ be the expected running time of $P$ on $x$, where we average over the halting runs. Then only a fraction 1/10 of the halting runs can take more than $10T$ steps and hence $P$ halts on $x$ within $10T$ steps with probability at least $(9/10)/(7/8)$. A small adaption of the computation in (2) shows that sufficiently many runs of $P$ stop within $10T$ steps. The same argument holds true for the space complexity.

We summarize the discussion.

**Theorem 14.** *Every randomized program computing a function has an efficient weakly certifying counterpart. This assumes that a proof for (21) in a formal system is available.*

How important is it that $P$ computes a function? Consider the following randomized algorithm. It takes as an input the description of 10 Turing machines and produces 10 random bits. The postcondition is that for at least one $i$, the $i$th bit tells whether the $i$th Turing machine halts on empty input. This postcondition is satisfied with probability $1-1/2^{10}$. There seems to be now way to boost the success probability of this program to a higher value and there seems to be now way to check the output of this program.

## 11. Certification and verification

We give examples where certification and verification can support each other. Some properties of the algorithm are more easily verified and other are more easily certified.

*Sorting:* The example is due to Gerhard Goos (personal communication). The input to a sorting algorithm is a set of elements from a linearly ordered set. The output is the same set but in sorted order. Sortedness is easily checked and for a subclass of sorting algorithm integrity of the input is easily verified.

It is trivial to check that a sequence of elements is sorted, say in ascending order. We simply step through the sequence and check that we see successively larger elements. It is difficult to check that two sets agree. In fact, the most efficient way to check equality of sets is to sort them and check equality of the sorted sets.

General sorting algorithms frequently use a subroutine *swap* that exchanges the position of two elements: *swap* is an extremely short program (less than three lines in most programming languages) whose correctness is obvious (and is also easily proven). Any program which only uses (a correct) *swap* to move elements around, outputs a permutation of its input set.

*Minimum cut:* Our second example is more subtle. A cut in an undirected graph $G = (V, E)$ is any subset $S$ of the vertices which is neither empty nor the full set of vertices, i.e., $\emptyset \neq S \neq V$. Let $c$ be a cost function on the edges of $G$. The weight

of a cut $S$ is the total cost of the edges having exactly one endpoint in $S$, i.e.,

$$c(S) = \sum_{e;|e\cap S|=1} c(e).$$

A minimum cut is a cut of minimum weight. We do not know of a certificate for minimum cuts. There is a very efficient Monte Carlo algorithm for computing a minimum cut [114]. Here, we consider the following generic approach to computing a minimum cut.

The algorithm works recursively. It uses a subroutine, which does the following: it determines two vertices $s$ and $t$ ($s$ and $t$ are not input to the subroutine, but the subroutine determines them) and a cut $S$ which is minimal among all cuts containing $s$ but not $t$. Assume that we call the subroutine for $G$ and it returns $s$, $t$ and a cut $S$. There are two cases: either $S$ is a minimum cut in $G$ or the minimum cut in $G$ does not separate $s$ and $t$. In the former case we are done (however, we do not know this) and in the latter case, we may collapse $s$ and $t$ into a single vertex, i.e., remove $s$ and $t$ from the graph, add a new vertex $z$ and for any $v \in V \setminus \{s,t\}$ assign $c(s,v)+c(t,v)$ as cost of the edge[13] $(z,v)$, and determine a minimum cut in the reduced graph. The minimum cut in the reduced graph will also be a minimum cut in the original graph (after replacing $z$ by $\{s,t\}$). An iterative version of the algorithm is as follows:

**while** $G$ has at least two vertices **do**
  determine two vertices $s$ and $t$ in $G$ and a cut $S$ which is minimal among all cuts containing $s$ but not $t$;
  collapse $s$ and $t$.
**end while**
output the smallest cut found in any iteration;

Given the correctness of the subroutine, the algorithm is correct. Stoer and Wagner [67], simplifying an earlier algorithm of Nagamochi and Ibaraki [53], gave an efficient algorithm for the subroutine. Arikati and Mehlhorn [115] made the subroutine certifying. The certifying version computes $s$, $t$, the cut $S$ and a flow $f$ from $s$ to $t$ of value $c(S)$. The flow proves the minimality of $S$ by the min-cut-max-flow theorem (Lemma 7).

## 12. Reactive programs and data structures

Reactive programs run forever. They keep state information and receive inputs. They return outputs and update their internal state. In the algorithms community, reactive programs are called data structures and this is the terminology which we are going to use. The papers [116,117,1,2,38–41,118] discuss certification of data structures.

We distinguish between abstract data types (=specifications of the intended behavior of the data structure) and implementations (=programs exhibiting a certain behavior). The question is how to certify that a certain implementation implements a certain data type. Akin to certifying programs, we consider certification of data structures.

[13] Of course, if $v$ is not connected to either $s$ or $t$, there is no need for introducing an edge connecting $v$ and $z$.

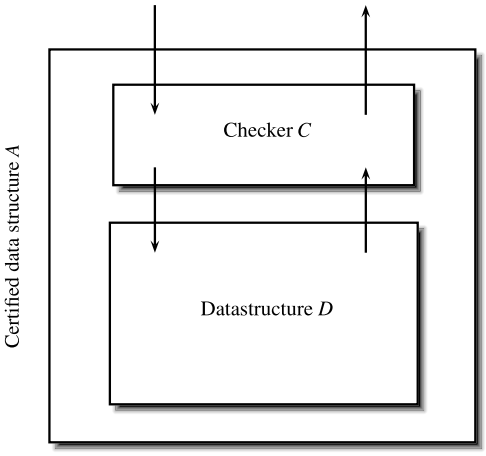



**Fig. 19 – $D$ is a data structure implementation and $C$ monitors its behavior. Any input is passed to $C$ which then forwards it, maybe in modified form to $D$. $D$ reacts to it, $C$ inspects the reaction of $D$ and returns an answer to the environment. If $D$ is correct, the combination of $C$ and $D$ realizes the abstract data type $A$. If $D$ is incorrect, $C$ catches the error.**

We explore the following scenario, see Fig. 19. We have an abstract data type $A$, a data type implementation $D$ allegedly realizing an abstract data type $A'$, and a monitor or checker $C$. The checker monitors the execution of the data structure $D$. We may also call it a watchdog. The pair $C+D$ is supposed to implement the behavior $A$ as follows: The checker $C$ interacts with the environment and offers the interface of the abstract data type $A$. It also interacts with $D$ through the interface of $A'$. The purpose of the checker is twofold:

- The checker offers the behavior $A$ to the environment. It makes use of $D$ to do so. Ideally, all the hard work is done by $D$ and $C$ is only a simple interface layer.
- It checks that $D$ correctly implements $A'$. If $D$ does, $C$ is supposed to keep quiet. If $D$ does not, $C$ must raise an exception. This may happen immediately or eventually. It the former case $C$ must raise an exception immediately after the first incorrect reaction by $D$, in the latter case, the exception may be delayed and only must come eventually.

We call $C$ a monitor that realizes the abstract data type $A$ in terms of the abstract data type $A'$. Such a monitor is useful whenever the implementation of $A'$ is unknown or untrusted.

The abstract data type $A'$ should be at least as strong as $A$; otherwise, we would not speak of checking but of enhancing. In the interest of certifiability it may be necessary to make $A'$ stronger than $A$. The stronger behavior may be easier to check. This is akin to the situation for algorithms. In order to achieve certifiability, we solve a more general problem.

We denote operations performed on $D$ by $D.op(argumentlist)$ and operations performed on the abstract data type by $op(argumentlist)$.

We will present the following results: There is a checker for ordered dictionaries ($A = A' =$ ordered dictionary) that catches errors immediately and adds $O(1)$ overhead to each dictionary operation. Since ordered dictionaries are stronger than priority queues, we may also use the checker with

$A$ = priority queue and $A'$ = ordered dictionary. So ordered dictionaries are fairly easy to check.

For the second class of results we have $A = A' =$ priority queue. We will see that there is checker with $O(1)$ amortized overhead per operation that catches errors eventually [118]. We will also see that a checker that wants to catch errors immediately, must incur an amortized logarithmic overhead per operation. This is also the cost of priority queue operations. In other words, priority queues are hard to check and a checker that wants to catch errors immediately must essentially implement a priority queue by himself.

### 12.1. The dictionary problem

The dictionary problem for a universe $U$ and a set $I$ of information asks to maintain a set $S$ of pairs $(x, i) \in U \times I$ with pairwise-distinct keys (=first elements) under operations *insert*$(x, i)$, *delete*$(h)$, *find*$(x)$, *set_inf*$(h, i)$, *key*$(h)$ and *inf*$(h)$. Here, $h$ is a handle to (=pointer to) a pair in the dictionary.

*Insert*$(x, i)$ adds the pair $(x, i)$ to $S$ and returns a handle $h$ to the pair in the data structure. If there is already a pair in $S$ with key $x$, the information of the pair is replaced by $i$. *Delete*$(h)$ deletes the pair with handle $h$ and invalidates $h$; $h$ must be a valid handle of a pair in $S$. *Find*$(x)$ returns a handle to the pair with key $x$ (if any) and returns a special element *nil* otherwise, *set_inf*$(h, i)$ sets the information of the pair handled by $h$ to $i$, and *key*$(h)$ and *inf*$(h)$ return the key and information of the pair handled by $h$, respectively. If any of operations above is supplied with an invalid handle, the outcome of the operation and all subsequent operations is unspecified.

In the ordered dictionary problem, we assume that $U$ is linearly ordered and require the additional operations *locate*$(x)$ and *findmin*$(\,)$. The former returns a handle to the pair $(y, \,) \in S$ with minimal key $y \geq x$, if there is such a pair. Otherwise, the special value *nil* is returned. *Findmin*$(\,)$ returns a handle to the pair in $S$ with minimal key. If $S$ is empty, it returns *nil*.

*Checking ordered dictionaries with constant overhead:* We show that ordered dictionaries can be checked with constant overhead per operation. Errors are caught immediately.

Let $D$ be the alleged implementation of an ordered dictionary and let $S$ be the set stored in the dictionary. The checker maintains a doubly-linked list of triples $(x, i, h)$ sorted by key. There is one triple for each pair $(x, i) \in S$; $h$ is a handle to an item in $D$. The item handled by $h$ contains the pair $(x, r)$, where $r$ is a handle to the triple $(x, i, h)$ in $L$. In other words, corresponding pairs in $D$ and $L$ are linked to each other and $L$ contains the pairs in $S$ in sorted order. How do we make sure that $r$ is a handle into $L$ and not a handle to some uncontrolled location in memory which by accident points back to $D$. The standard solution for this problem is to store $L$ in the first $|L|$ entries of an array $LA$, one list item per entry of the array. The handle $r$ is accepted if it points into the first $|L|$ entries of $LA$. After a deletion from $L$, the element in position $LA[|L|]$ of $LA$ is moved to the position of the deleted element. The cross-pointers between $L$ and $D$ are changed accordingly; for the change in $D$, the operation $D$.*set_inf*$(h, \,)$ is used. We next discuss the implementations of the operations:

*Insert*$(x, i)$: The checker calls $D$.*locate*$(x)$ and $D$ returns a handle $h$ or *nil*. In the former case $h$ points to a pair $(y, r)$; $C$ uses $D$.*key*$(h)$ and $D$.*inf*$(h)$ to read out $y$ and $r$. If $r$ does not point into $L$, $C$ declares failure. Let $(z, j, h')$ be the entry of $L$ handled by $r$. If $h' \neq h$ or $z \neq y$, $C$ declares failure. Otherwise, it checks that $x \leq y$ and that $x$ is larger than the key in the triple preceding $(y, j, h)$. If not, the checker declares failure. If $y = x$, it replaces $j$ by $i$, and if $x < y$, it inserts a triple $(x, i, \,)$ into $L$ just before the triple with key $y$. Let $s$ be a handle to this triple. Next it inserts $(x, s)$ into $D$ and stores the returned handle in the triple with handle $s$. It returns $s$. If $D$ returns *nil*, $C$ checks that $x$ is larger that the key in the last item of $L$. If not, $C$ declares failure. If so, it inserts a triple $(x, i, \,)$ into $L$ after the last item of $L$. Let $s$ be a handle to this triple. Next, it inserts $(x, s)$ into $D$ and stores the returned handle in the triple with handle $s$. $C$ returns $s$.

*Delete*$(s)$: $s$ is a handle to an item $(x, i, h)$ in the list $L$. We remove the pair with handle $h$ from $D$ and we remove the item $(x, i, h)$ from $L$.

*findmin*$(\,)$: The checker calls $D$.*findmin* and obtains a handle $h$ to a pair $(x, r)$. As above the checker checks that the pair in $L$ handled by $r$ is equal to $(x, \,, h)$. It also checks that $r$ handles the first element of $L$. If so, it returns $r$.

We leave the implementation of the other operations to the reader.

Ordered dictionaries can be used to sort. Sorting $n$ items takes $\Omega(n \log n)$ comparisons. There is a nice division of labor. After the insertion of $n$ elements, the checker has the sorted list $L$ of the elements. However, it has performed only $O(n)$ comparisons. The hard work of locating a new element in the current list is done by $D$; then $C$ performs two comparisons to verify that $D$ did not lie. In the next section, we will show that the interface of priority queues is too narrow and does not allow a similar division of labor.

### 12.2. Priority queues

A priority queue offers the operations *insert*$(x, i)$ and *delmin*$(\,)$. Delmin returns and deletes pair with minimum key. The priority queue operations are a subset of the ordered dictionary operations and hence the construction of the preceding section is also a monitor that realizes priority queues in terms of ordered dictionaries.

In this subsection, we will discuss monitors that realize priority queues in terms of priority queues. We will show that there is a checker which reports errors with delay. The checker has constant overhead. We also show that any checker which catches errors with no delay, must incur logarithmic amortized cost per operation.

*A checker with delay:* We review a construction given in [118]; [20, Section 5.5.3] contains all implementation details. There is a simple but inefficient way for monitoring a priority queue. After every $D$.*findmin* operation, we simply check that the reported priority is the smallest of all priorities in the queue. This solution does the job but defeats the purpose as it adds linear overhead. We will reduce the overhead at the expense of catching errors delayed. When a $D$.*findmin* operation is performed, the checker will record that all items currently in the queue must have a priority at least

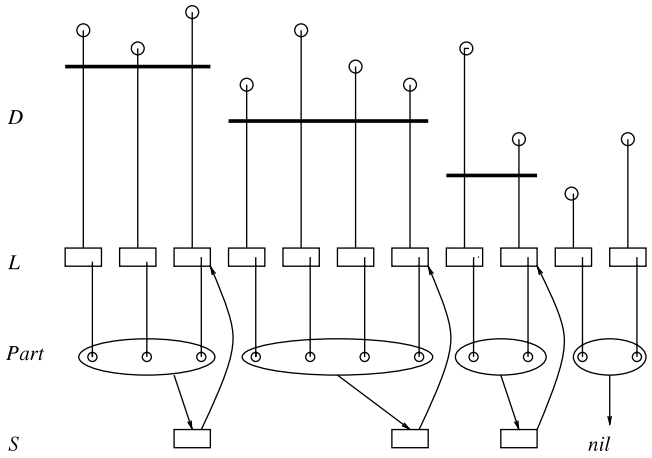



**Fig. 20 – In the top part of the figure the items in a priority queue $D$ are shown as circles in the $xy$-plane. The $x$-coordinate corresponds to the insertion time of the item and the $y$-coordinate corresponds to its priority. The lower bounds for the priorities are indicated as heavy horizontal lines. The lower bound for the last two items is $-\infty$. The lower part of the figure shows the data structures of the checker. The list $L$ has one item for each item in $D$, the list $S$ has one item for each step, and the union-find data structure *Part* connects each item in $L$ to the step to which it belongs. The blocks of *Part* are indicated as ellipses. Each element in $S$ knows the lower bound associated with the step and the last item in $L$ belonging to the step.**

as large as the priority reported. The actual checking is done later.

Consider Fig. 20. The top part of this figure shows the items in a priority queue from left to right in the order of their insertion time. The $y$-coordinate indicates the priority. With each item of the priority queue we have an associated lower bound. The *lower bound* for an item is the maximal priority reported by any *D.findmin* operation that took place after the insertion of the item. $D$ operates correctly if the priority of any item is at least as large as its lower bound. We can therefore check $D$ by comparing the priority of an item with its lower bound when the item is deleted from $D$.

How can we efficiently maintain the lower bounds of the items in the queue? We observe that lower bounds are monotonically decreasing from left to right and hence have the staircase-like form shown in Fig. 20. We call a maximal segment of items with the same lower bound a *step*. How does the system of lower bounds evolve over time? When a new item is added to the queue its associated lower bound is $-\infty$ and when a *D.findmin* operation reports a priority $p$ all lower bounds smaller than $p$ are increased to $p$. In other words, all steps of value at most $p$ are removed and replaced by a single step of value $p$. Since the staircase of lower bounds is falling from left to right this amounts to replacing a certain number of steps at the end of the staircase by a single step, see Fig. 21.

We can now describe the details of the checker. It keeps a linear list $L$ of items, one for each item in $D$. As in Section 12.1, the items in $L$ and the items in $D$ are cross-linked. The list $L$ is ordered by insertion time. The checker also keeps a list $S$ of steps and a union-find data structure *Part* for the items in $L$. The blocks of *Part* comprise the items in a step and a block has a pointer to the element in $S$ representing the step. An item in $S$ stores the lower bound associated with the step and a pointer to the last element in $L$ belonging to the step, see Fig. 20.

When an element $(x, i)$ is inserted into the queue, the checker forwards the insertion to $D$, adds an item to $L$, cross-links the two new items, and adds the new item in $L$ to the step with lower bound $-\infty$. If this step does not exist, it is created.

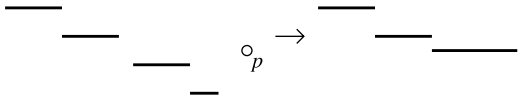



**Fig. 21 – Updating the staircase of lower bounds after reporting a priority of $p$. All steps whose associated lower bound is at most $p$ are replaced by a single step whose associated lower bound is $p$.**

When an element is deleted, the checker deletes the corresponding item in $L$, uses the union-find data structure to find the step to which the item belongs, and verifies that the priority of the item is at least as large as the lower bound stored with the step.

When a *D.findmin*( ) reports a priority $p$, all steps with lower bound at most $p$ are united into a single step. The relevant steps are found by scanning $S$ from its right end. The union-operations are initiated by going from the element in $S$ to the last item in $L$ that belongs to the block and then calling the union operation of the union-find data structure.

*A lower bound for on-line checkers:* We prove that any checker $C$ that catches errors with no delay, must incur logarithmic amortized cost per operation in the worst case. We will use the pair $C + D$ to sort $n$ elements by performing the sequence

$$\mathit{insert}(a_1); \ldots; \mathit{insert}(a_n); \mathit{delmin}( ); \ldots; \mathit{delmin}( ).$$

The pair $C+D$ must perform $\Omega(n \log n)$ comparisons to execute this sequence correctly. We will show that $C$ must perform $\Omega(n \log n)$ comparisons. We use the lower bound technique developed in [119]. We make the following assumptions about the checker: it cannot invent elements, i.e., only arguments of insert operations can be passed on to the data structure for insertion. However, not all arguments need to be passed on. Similarly, not all deletions need to be passed on. In

this way the set of elements stored in $D$ is a subset of the elements inserted by the environment. We use $S_D$ to denote the elements stored in $D$ and $S_A$ to denote the elements stored in $A$. The lower bound is through an adversary argument. The adversary fixes

1. the outcome of the comparisons made by $C$. We use $<_C$ for the partial order determined by $C$.
2. the outcome of *D.delmin*( ) operations.

Of course, any answer of the adversary must be consistent with previous answers. The checker interacts with the environment and with $D$. It has the following obligations.

1. With respect to the environment, it must implement a priority queue, i.e., every *delmin*( )-operation must return a handle to the minimum element in $S_A$.
2. With respect to the data structure, it must catch errors immediately, i.e., before the next interaction with the environment. In other words, after every *D.delmin*( )-operation, $C$ performs some comparisons and then either accepts or rejects the answer of $D$. Let $m$ be the priority returned by the *D.delmin*( )-operation. If $m$ is the minimum of $S_D$, $C$ must accept the answer, and if $m$ is not the minimum of $S_D$, $C$ must reject the answer.

There are two situations, where $C$ can readily decide. If $m$ is the minimum of $S_D$ under $<_C$, $C$ may accept, and if $m$ is not a minimal element of $S_D$ under $<_C$, $C$ must reject. We next show that $C$ cannot decide with less knowledge.

**Lemma 23.** *When $C$ accepts the answer $m$ of a D.delmin( ), $m$ must be the minimum of $S_D$ with respect to $<_C$, and if $C$ rejects, $m$ must not be a minimal element of $S_D$ with respect to $<_C$.*

**Proof.** Consider the first operation *D.delmin*( ), where the claim is false. Then $C$ accepted the outcomes of all previous *D.delmin*( )-operations and hence their answers were determined by $<_C$ at the time of their acceptance and hence by the current $<_C$.

Assume now that $m$ is a minimal element of $S_D$ with respect to $<_C$, but is not a minimum. Then there are linear extensions $<_1$ and $<_2$ of $<_C$, one having $m$ as the minimum and one not having $m$ as the minimum. Consider now the execution of our algorithm on $<_1$ and $<_2$. Since both orders extend $<_C$, the answers of all previous operations *D.delmin*( ) are still correct. However, under $<_1$, the answer to the current *D.delmin*( )-operation is correct and under $<_2$, the answer is incorrect. Thus $<_C$ does not contain sufficient information for $C$ to decide. □

The interactions with the environment are completely determined by the sequence of inserts, deletes, and the comparisons made by $C$. In particular, whenever an environment-*delmin*( ) is answered, $<_C$ must determine a unique minimal element in $S_A$. However, which comparisons are made by $C$ depends on the outcome of the *D.delmin*( ) operations. It is last sentence which forces us to continue the argument. It is conceivable, that the outcome of the *D.delmin*( ) operations guides the checker to the right comparisons. This was the case in the Section 12.1 where the checker needed only two comparisons to insert a new element at the right position. The interface of priority queue is too narrow to support a similar approach.

We next define the strategy of the adversary in a way similar to [119]. In [119], the adversary only needs to fix the outcome of comparisons made by $C$. It now also has to fix the outcome of *D.delmin*( ) operations. Since the adversary needs to stay consistent with itself, this will fix outcomes of future comparisons made by $C$. The strategy of [119] is as follows: The elements sit in the nodes of a binary tree; we use $v(x)$ to denote the node containing $x$ and $\ell(x)$ to denote the depth of the node $v(x)$. The number of comparisons made by $C$ will be $\Omega(\sum_x \ell(x))$. We say that an element $x$ sits left of an element $y$ if and only if there is a node $v$ such that $x$ sits in a left descendant of $v$ and $y$ sits in a right descendant of $v$. There are three possibilities for two elements $x$ and $y$. Either $x$ sits left of $y$ or $x$ sits right of $y$ or $x$ and $y$ sit on a common path, i.e., in nodes where one is an ancestor of the other. The partial order on the elements as follows: $x$ is smaller than $y$ if and only if $x$ sits left of $y$. If $x$ and $y$ sit on a common path, the order is still undecided. All linear orders compatible with the current partial order can be obtained by moving elements down the tree until no two elements sit on a common path. An insert puts the new element into the root.

When elements $x$ and $y$ are compared and

- $x$ and $y$ sit in the same node, we move one to the left and one to the right
- $x$ sits in a proper ancestor of $y$, $x$ is moved to the child which is not an ancestor of $y$,
- $x$ and $y$ sit in nodes of which neither is an ancestor of the other, no move is required.

Then the outcome of the comparison is as defined above.

**Lemma 24.** *If $x < y$ follows from the answers of the adversary then $x$ sits left of $y$.*

**Proof.** If $x < y$ follows from the answers of the adversary, there are elements $z_0, z_1, \ldots, z_k$ such that $x = z_0$, $y = z_k$, and $z_i$ was declared smaller than $z_{i+1}$ by the adversary for $0 \le i < k$. After $z_i$ was declared smaller than $z_{i+1}$, it sits left of $z_{i+1}$ in the tree. As elements only move down, it stays to the left of $z_{i+1}$. Thus $x$ sits left of $y$ in the tree. □

How does the adversary fix the outcome of a *D.delmin*( ) operation? By Lemma 23, the only logical constraint is that the outcome must be consistent with $<_C$. Consider the elements in $S_D$ and how they are distributed over the tree. If $S_D = \emptyset$, no action is required. Otherwise, we define a tree-path $p$ and call it the *D-min-path*. It starts in the root. Assume that we have extended $p$ up to a node $v$. If no elements in $S_D$ are stored in proper descendants of $v$, the path ends. If some are stored in the left subtree, we proceed to the left child, otherwise, we proceed to the right child. The elements of $S_D$ lying on $p$ are exactly the elements in $S_D$ which are minimal elements with respect to $<_C$. Observe that the D-min-path changes over time. Lemma 23 tells us that whenever $C$ accepts the outcome of a *D.delmin*( )-operation, the D-min-path must contain a single element in $S_D$.

The adversary fixes the outcome of *D.delmin*( ) as follows: it returns an element $m$ in $S_D$ in the highest (=closest to the root) non-empty node of $p$ for which the left child is not on $p$; non-empty means that the node contains an element of $S_D$. This node exists since the last node of $p$ is a non-empty node

for which the left child is not on $p$. If $m$ is the only element of $S_D$ stored on $p$, the adversary is done. Otherwise, it implicitly moves $m$ to the left child of $v$ and all elements of $S_D$ different from $m$ sitting in ancestors of $v$ and including $v$ to their right child. We say that the elements are moved implicitly, because these moves are hidden from $C$. The adversary commits to these moves but it does not make them yet. It makes them when the elements are involved in a comparison explicitly asked for by $C$. Let $R$ be the set of elements moved in this way. When an element in $R$ is involved in the next comparison, the adversary actually performs the hidden move and then follows the Brodal-et-al strategy. Let $k = |R| \geq 1$. We will show that $C$ has to make $\Omega(k)$ comparisons before it can accept the outcome of *D.delmin*( ).

Consider the comparisons made by $C$ between the call to *D.delmin*( ) and the commit to the answer $m$. By Lemma 23, $m$ must be the minimum of $S_D$ under $<_C$ when $C$ accepts $m$. Thus at the time of acceptance, the $D$-min-path contains a single element. We now distinguish cases: $R \setminus m$ is non-empty or $R = \{m\}$. In the former case, assume that there is an element in $R \setminus m$ that is not involved in a comparison made by $C$ before the time of commitment. Then this elements still sits in an ancestor of the node containing $m$ at the time of commitment and hence its order with respect to $m$ is still open. Thus $C$ cannot commit, a contradiction. We conclude that if the adversary performs $k$ hidden moves, we can charge them to at least $\lceil (k-1)/2 \rceil$ comparisons made by $C$ made before it accepts $m$.

If $k = 1$, we have to argue differently. In this case, no other element in $S_D$ is stored in an ancestor of or at the node containing $m$. However, in this case, the $D$-min-path extends beyond the node containing $m$ (otherwise, we would be in the situation that the D-min-path contains a single element of $S_D$) and hence $C$ must perform a comparison involving $m$ before it can accept $m$.

In this way, we charge $O(1)$ moves to a single comparison. More precisely, we charge at most five moves to a single comparison. The worst case occurs when $k = 3$. The adversary performs three hidden moves, $C$ performs one comparison and this results in two more moves.

It is now easy to complete the lower bound. We consider the sequence

*insert*($a_1$); ...; *insert*($a_n$); *delmin*( ); ...; *delmin*( );

This sequence sorts. When a delmin is answered, the element returned must be the unique minimum on the A-minimum path (this is the path containing the potential minima under $<_C$ in $S_A$). Thus there is no element stored in an ancestor of the element returned. By the definition of the adversary strategy, this is also true at the end of operation sequence. Let $\ell_i$ be the depth of the element returned by the $i$th delmin-operation at the end of the execution. Then[14] $\sum \ell_i \geq n \log n$. Since any comparison made by $C$ increases the sum $\sum_i \ell_i$ by at most five, we conclude that $C$ performs $(1/5)n \log n$ comparisons.

[14] Let $v_1, \ldots, v_n$ be $n$ nodes in an infinite binary tree, no two of which are ancestors of each other and let $\ell_i$ be the depth of $v_i$. Then $\sum \ell_i \geq n \log n$ as an easy induction shows. The claim is clear for $n = 1$. Assume now that $n > 1$ and that $n_1$ and $n_2$ elements lie in the left and right subtree, respectively. Then $\sum_{1 \leq i \leq n_1} (\ell_i - 1) \geq n_1 \log n_1$ and $\sum_{n_1+1 \leq i \leq n} (\ell_i - 1) \geq n_2 \log n_2$ by induction hypothesis. Thus

$$\sum_i \ell_i \geq n + n_1 \log n_1 + n_2 \log n_2$$
$$= n \log n + n \left(1 + \frac{n_1}{n} \log \frac{n_1}{n} + \frac{n_2}{n} \log \frac{n_2}{n}\right) \geq n \log n.$$

**Theorem 15.** *In the comparison model, any checker for priority queues which reports errors immediately, must perform* $(1/5)n \log n$ *comparisons.*

## 13. Teaching algorithms

The concept of certifying algorithm is easily incorporated into basic and advanced algorithm courses. We believe that it must be incorporated. In our own teaching we have used the following approach.

1. We present a certifying algorithm whenever possible. If no certifying algorithm is known, we present its design as a research problem. If only an inefficient certifying algorithm is known, we present it together with the more efficient non-certifying algorithm and present the design of an efficient certifying algorithm as a research problem.
2. When we discuss the first certifying algorithm, we spend time on motivation (Section 3) and usefulness (Section 7).
3. If the theme of integer arithmetic fits into the course, we explain that checking a division through multiplication and checking a multiplication through the method of "casting out nines" (see Section 10.2) are ancient examples of certifying the correctness of a computation.
4. If the computation of greatest common divisors fits into the course, we treat the basic and the extended Euclidean algorithm as examples of a non-certifying and a certifying algorithm (see Section 2.3).
5. In some advanced courses, we have discussed the theory of certifying algorithms (Section 5) and/or general approaches to certifying algorithms (Section 8).
6. In courses on linear programming, duality is discussed as a general principle of certification (Section 8.2).

As course material, we have used draft versions of this article. In the future, we will use this article. Also, the recent textbook by Mehlhorn and Sanders [120] uses the concept of certifying algorithms.

## 14. Future work

Open problems are numerous. Any algorithmic problem for which there is no certifying algorithm or only a certifying algorithm whose running time is of higher order than the known best non-certifying algorithm is an open problem. Our personal favorites are 3-connectivity of graphs (see Section 5.4), arrangements of algebraic curves, shortest paths in the plane or in space in the presence of obstacles, and the algebraic number packages in LEDA and CGAL.

We would also like to see advances in formal verification of certifying algorithms. For most algorithms mentioned in this paper, it should be feasible to give a formal proof

for the witness property and for the correctness of the checking program. Eyad Alkassar, Christine Rizkallah, Norbert Schirmer, and the second author have recently given such proofs for the maximum cardinality matching problem (see Section 2.5) in Isabelle [121] and VCC [122], respectively.

Also, our definition of certifying algorithm should be reconsidered. We mentioned that some fellow researchers feel that Theorems 5 and 7 should not hold. The task is then to find an appropriate restrictive definition of certifying algorithm.

## 15. Conclusions

Certifying algorithms are a preferred kind of algorithm. They prove their work and they are easier to implement reliably. Their widespread use would greatly enhance the reliability of algorithmic software.

## Acknowledgements

We want to thank many colleagues for discussions about aspects of this paper, in particular, Ernst Althaus, Peter van Emde Boas, Harry Buhmann, Arno Eigenwillig, Uli Finkler, Stefan Funke, Dieter Kratsch, Franco Preparata, Peter Sanders, Elmar Schömer, Raimund Seidel, Jeremy Spinrad, and Christian Uhrig.

## REFERENCES

[1] G.F. Sullivan, G.M. Masson, Using certification trails to achieve software fault tolerance, in: Brian Randell (Ed.), Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing, FTCS'90, IEEE, 1990, pp. 423–433.
[2] G.F. Sullivan, G.M. Masson, Certification trails for data structures, in: Proceedings of the 21st Annual International Symposium on Fault-Tolerant Computing, FTCS'91, IEEE Computer Society Press, Montreal, Quebec, Canada, 1991, pp. 240–247.
[3] M. Blum, S. Kannan, Designing programs that check their work, J. ACM 42 (1) (1995) 269–291. preliminary version in STOC'89.
[4] K. Mehlhorn, S. Näher, LEDA: a library of efficient data types and algorithms, in: MFCS'89, in: Lecture Notes in Computer Science, vol. 379, 1989, pp. 88–106.
[5] K. Mehlhorn, S. Näher, LEDA, a platform for combinatorial and geometric computing, Communications of the ACM 38 (1995) 96–102.
[6] D. Kratsch, R. McConnell, K. Mehlhorn, J. Spinrad, Certifying algorithms for recognizing interval graphs and permutation graphs, SIAM Journal on Computing 36 (2) (2006) 326–353. preliminary version in SODA 2003, pp. 158–167.
[7] Kurt Mehlhorn, Stefan Näher, Michael Seel, Raimund Seidel, Thomas Schilz, Stefan Schirra, Christian Uhrig, Checking geometric programs or verification of geometric structures, Computational Geometry 12 (1–2) (1999) 85–103. preliminary version in SoCG 96.
[8] K. Mehlhorn, S. Näher, C. Uhrig, The LEDA platform for combinatorial and geometric computing, in: Proceedings of the 24th International Colloquium on Automata, Languages and Programming, ICALP'97, in: Lecture Notes in Computer Science, vol. 1256, 1997, pp. 7–16.
[9] K. Mehlhorn, S. Näher, From algorithms to working programs: on the use of program checking in LEDA, in: MFCS'98, in: Lecture Notes in Computer Science, vol. 1450, 1998, pp. 84–93.
[10] A. Zeller, WHY PROGRAMS FAIL: A Guide to Systematic Debugging, Morgan-Kaufmann, 2005.
[11] M. Blum, H. Wasserman, Reflections on the Pentium division bug, IEEE Transaction on Computing 45 (4) (1996) 385–393.
[12] R. Floyd, Assigning meaning to programs, in: J.T. Schwarz (Ed.), Mathematical Aspects of Computer Science, AMS, 1967, pp. 19–32.
[13] C.A.R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (1969) 576–585.
[14] G. Gonthier, Formal proof–the Four-Color theorem, Notices of the American Mathematical Society 55 (11) (2008).
[15] J. Strother Moore, Qiang Zhang, Proof pearl: Dijkstra's shortest path algorithm verified with acl2, in: Joe Hurd, Thomas F. Melham (Eds.), Theorem Proving in Higher Order Logics, in: Lecture Notes in Computer Science, vol. 3603, Springer, 2005, pp. 373–384.
[16] Verisoft XT. http://www.verisoft.de/index_en.html.
[17] J.D. Bright, G.F. Sullivan, G.M. Masson, A formally verified sorting certifier, IEEE Transactions on Computers 46 (12) (1997) 1304–1312.
[18] J. Edmonds, Maximum matching and a polyhedron with 0, 1 — vertices, Journal of Research of the National Bureau of Standards 69B (1965) 125–130.
[19] J. Edmonds, Paths, trees, and flowers, Canadian Journal on Mathematics (1965) 449–467.
[20] K. Mehlhorn, S. Näher, The LEDA Platform for Combinatorial and Geometric Computing, Cambridge University Press, 1999.
[21] J.E. Hopcroft, R.E. Tarjan, Efficient planarity testing, Journal of the ACM 21 (1974) 549–568.
[22] A. Lempel, S. Even, I. Cederbaum, An algorithm for planarity testing of graphs, in: P. Rosenstiehl (Ed.), Theory of Graphs, International Symposium, Rome, 1967, pp. 215–232.
[23] K.S. Booth, G.S. Lueker, Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms, Journal of Computer and System Sciences 13 (1976) 335–379.
[24] S.G. Williamson, Depth-first search and Kuratowski subgraphs, Journal of the ACM 31 (4) (1984) 681–693.
[25] A. Karabeg, Classification and detection of obstructions to planarity, Linear and Multilinear Algebra 26 (1990) 15–38.
[26] J.E. Hopcroft, R.E. Tarjan, Dividing a graph into triconnected components, SIAM Journal of Computing 2 (3) (1973) 135–158.
[27] G.L. Miller, V. Ramachandran, A new graph triconnectivity algorithm and its parallelization, Combinatorica 12 (1) (1992) 53–76.
[28] C. Gutwenger, P. Mutzel, A linear time implementation of SPQR-trees, in: Graph Drawing, in: LNCS, vol. 1984, 2000, pp. 77–90.
[29] M. Dhiflaoui, S. Funke, C. Kwappik, K. Mehlhorn, M. Seel, E. Schömer, R. Schulte, D. Weber, Certifying and repairing solutions to large lps, how good are LP-solvers? in: SODA, 2003, pp. 255–256.
[30] David L. Applegate, William Cook, Sanjeeb Dash, Daniel G. Espinoza, Exact solutions to linear programming problems, Operations Research Letters 35 (6) (2007) 693–699.
[31] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, C. Yap, Classroom examples of robustness problems in geometric computations, Computational Geometry: Theory and Applications (CGTA) 40 (2008) 61–78. a preliminary version appeared in ESA 2004, LNCS, vol. 3221, pp. 702–713.

[32] C.-K. Yap, Robust geometric computation, in: J.E. Goodman, J. O'Rourke (Eds.), Handbook of Discrete and Computational Geometry, 2nd ed., CRC Press LLC, Boca Raton, FL, 2003 (Chapter 41).
[33] CGAL (Computational Geometry Algorithms Library). www.cgal.org.
[34] D. Halperin, E. Leiserowitz, Controlled perturbation for arrangements of circles, International Journal of Computational Geometry and Applications 14 (4) (2004) 277–310. preliminary version in SoCG 2003.
[35] C.A.R. Hoare, Proof of correctness of data representations, Acta Informatica 1 (1972) 271–281.
[36] V. Chvatal, Linear Programming, Freeman, 1993.
[37] A. Schrijver, Combinatorial Optimization (3 Volumes), Springer Verlag, 2003.
[38] J.D. Bright, G.F. Sullivan, On-line error monitoring for several data structures, in: Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing, FTCS'95, Pasadena, California, 1995, pp. 392–401.
[39] J.D. Bright, G.F. Sullivan, G.M. Masson, Checking the integrity of trees, in: Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing, FTCS'95, Pasadena, California, 1995, pp. 402–413.
[40] G.F. Sullivan, D.S. Wilson, G.M. Masson, Certification of computational results, IEEE Transactions on Computers 44 (7) (1995) 833–847.
[41] J.D. Bright, G.F. Sullivan, Checking mergeable priority queues, in: Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing, FTCS'94, IEEE Computer Society Press, Los Alamitos, CA, USA, 1994, pp. 144–153.
[42] M. Blum, S. Kannan, Designing programs that check their work, in: Proceedings of the 21th Annual ACM Symposium on Theory of Computing, STOC'89, 1989, pp. 86–97.
[43] Manuel Blum, Program result checking: a new approach to making programs more reliable, in: ICALP, 1993, pp. 1–14.
[44] Manuel Blum, Hal Wasserman, Program result-checking: a theory of testing meets a test of theory, in: FOCS, 1994, pp. 382–392.
[45] M. Blum, M. Luby, R. Rubinfeld, Self-testing/correcting with applications to numerical problems, in: Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, STOC'90, 1990, pp. 73–83.
[46] Hal Wasserman, Manuel Blum, Software reliability via run-time result-checking, Journal of ACM 44 (6) (1997) 826–849. preliminary version in FOCS'94.
[47] LEDA (Library of Efficient Data Types and Algorithms). www.algorithmic-solutions.com.
[48] G.C. Necula, P. Lee, Safe kernel extensions without run-time checking, SIGOPS Operating Systems Review 30 (1996) 229–243.
[49] George C. Necula, Proof-carrying code, in: POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 1997, pp. 106–119.
[50] Shafi Goldwasser, Silvio Micali, Charles Rackoff, The knowledge complexity of interactive proof systems, SIAM Journal on Computing 18 (1) (1989) 186–208.
[51] Robin A. Moser, A constructive proof of the Lovasz local lemma, in: STOC '09: Proceedings of the Fourty-First Annual ACM Symposium on Theory of Computing, ACM, New York, NY, USA, 2009 (in press).
[52] J.M. Schmidt, Construction sequences and certifying 3-connectedness, in: 27th International Symposium on Theoretical Aspects of Computer Science, STACS'10, Nancy, France, 2010.http://drops.dagstuhl.de/portals/extern/index.php?conf=STACS10.
[53] H. Nagamochi, T. Ibaraki, A linear-time algorithm for finding a sparse $k$-connected spanning subgraph of a $k$-connected graph, Algorithmica 7 (1992) 583–596.
[54] M. Kriesell, A survey on contractible edges in graphs of a prescribed vertex connectivity, Graphs and Combinatorics (2002) 1–33.
[55] A. Elmasry, K. Mehlhorn, J.M. Schmidt, A linear time certifying triconnectivity algorithm for Hamiltonian graphs. Available at the second author's web-page, March 2010.
[56] S. Näher, K. Mehlhorn, LEDA: a library of efficient data types and algorithms, in: ICALP'90, in: Lecture Notes in Computer Science, vol. 443, Springer, 1990, pp. 1–5.
[57] A. Schrijver, Theory of Linear and Integer Programming, Wiley, 1986.
[58] Michael Jünger, William R. Pulleyblank, Geometric duality and combinatorial optimization, in: S.D. Chatterji, B. Fuchssteiner, U. Kulisch, R. Liedl (Eds.), Jahrbuch Überblicke Mathematik, Vieweg, 1993, pp. 1–24.
[59] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, A. Schrijver, Combinatorial Optimization, John Wiley & Sons, Inc., 1998.
[60] CPLEX. www.cplex.com.
[61] SoPlex. www.zib.de/Optimization/Software/Soplex.
[62] Maria Chudnovsky, Neil Robertson, Paul Seymour, Robin Thomas, The strong perfect graph theorem, Ann. of Math. (2) 164 (1) (2006) 51–229.
[63] Maria Chudnovsky, Gérard Cornuéjols, Xinming Liu, Paul D. Seymour, Kristina Vuskovic, Recognizing Berge graphs, Combinatorica 25 (2) (2005) 143–186.
[64] E.L. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan, D.B. Shmoys, The Traveling Salesman Problem, Wiley, 1985.
[65] M. Held, R.M. Karp, The traveling-salesman problem and minimum spanning trees, Operations Research 18 (1970) 1138–1162.
[66] M. Held, R.M. Karp, The traveling-salesman problem and minimum spanning trees, part II, Mathematical Programming 1 (1971) 6–25.
[67] M. Stoer, F. Wagner, A simple min-cut algorithm, Journal of the ACM 44 (4) (1997) 585–591.
[68] D. Applegate, B. Bixby, V. Chvatal, W. Cook, D.G. Espinoza, M. Goycoolea, K. Helsgaun, Certification of an optimal TSP tour through 85,900 cities, Operations Research Letters 37 (1) (2009) 11–15.
[69] D.R. Chand, S.S. Kanpur, An algorithm for convex polytopes, J. ACM 17 (1970) 78–86.
[70] F.P. Preparata, M.I. Shamos, Computational Geometry: An Introduction, Springer, 1985.
[71] R. Seidel, Constructing higher-dimensional convex hulls at logarithmic cost per face, in: Proceedings of the 18th Annual ACM Symposium on Theory Computing, STOC'86, 1986, pp. 404–413.
[72] K.L. Clarkson, P.W. Shor, Applications of random sampling in computational geometry, II, Journal of Discrete and Computational Geometry 4 (1989) 387–421.
[73] K. Clarkson, K. Mehlhorn, R. Seidel, Four results on randomized incremental constructions, Computational Geometry: Theory and Applications 3 (1993) 185–212.
[74] C. Barber, D. Dobkin, H. Hudhanpaa, The quickhull program for convex hulls, ACM Transactions on Mathematical Software 22 (1996) 469–483.
[75] K. Mehlhorn, M. Müller, S. Näher, S.S. Schirra, M. Seel, C. Uhrig, J. Ziegler, A computational basis for higher-dimensional computational geometry and its applications, Computational Geometry: Theory and Applications 10 (1998) 289–303.
[76] O. Devillers, G. Liotta, F. Preparata, R. Tamassia, Checking the convexity of polytopes and the planarity of subdivisions, CGTA: Computational Geometry: Theory and Applications 11 (1998).

[77] C. Burnikel, K. Mehlhorn, S. Schirra, On degeneracy in geometric computations, in: Proceedings of the 5th Annual ACM–SIAM Symposium on Discrete Algorithms, SODA'94, 1994, pp. 16–23.
[78] D. Applegate, B. Bixby, V. Chvatal, W. Cook, The Traveling Salesman Problem: A Computational Study, Princeton University Press, 2006.
[79] K. Helsgaun, An effective implementation of K-opt moves for the Lin-Kernighan TSP heuristic, Technical Report 109, Roskilde University, 2006, Writings in Computer Science.
[80] J. Kleinberg, E. Tardos, Algorithm Design, Addison Wesley, 2005.
[81] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, 1980.
[82] D. Rose, R.E. Tarjan, G.S. Lueker, Algorithmic aspects of vertex elimination on graphs, SIAM Journal on Computing 5 (1976) 266–283.
[83] E.E. Tarjan, M. Yannakakis, Addendum: simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, SIAM Journal on Computing 14 (1985) 254–255.
[84] J. Herzberger (Ed.), Topics in Validated Computations—Studies in Computational Mathematics, Elsevier, 1994.
[85] S. Rump, Self-validating methods, Linear Algebra and its Applications (LAA) 324 (2001) 3–13.
[86] Haim Kaplan, Yahav Nussbaum, Certifying algorithms for recognizing proper circular-arc graphs and unit circular-arc graphs, in: Fedor V. Fomin (Ed.), WG, in: Lecture Notes in Computer Science, vol. 4271, Springer, 2006, pp. 289–300.
[87] Min Chih Lin, Francisco J. Soulignac, Jayme Luiz Szwarcfiter, Proper helly circular-arc graphs, in: Brandstädt et al. [123], pp. 248–257.
[88] Stavros D. Nikolopoulos, Leonidas Palios, An O(nm)-time certifying algorithm for recognizing hhd-free graphs, in: Franco P. Preparata, Qizhi Fang (Eds.), FAW, in: Lecture Notes in Computer Science, vol. 4613, Springer, 2007, pp. 281–292.
[89] Van Bang Le, H.N. de Ridder, Characterisations and linear-time recognition of probe cographs, in: Brandstädt et al. [123], pp. 226–237.
[90] C. Crespelle, C. Paul, Fully dynamic recognition algorithm and certificate for directed cograph, Discrete Applied Mathematics 154 (12) (2006) 1722–1741.
[91] Daniel Meister, Recognition and computation of minimal triangulations for at-free claw-free and co-comparability graphs, Discrete Applied Mathematics 146 (3) (2005) 193–218.
[92] Pavol Hell, Jing Huang, Certifying lexbfs recognition algorithms for proper interval graphs and proper interval bigraphs, SIAM J. Discret. Math. 18 (3) (2005) 554–570.
[93] Pinar Heggernes, Dieter Kratsch, Linear-time certifying recognition algorithms and forbidden induced subgraphs, Nordic J. of Computing 14 (1) (2007) 87–108.
[94] U. Brandes, Eager st-ordering, in: ESA, 2002, pp. 247–256.
[95] Shimon Even, An algorithm for determining whether the connectivity of a graph is at least k, SIAM Journal on Computing 4 (3) (1975) 393–396.
[96] Zvi Galil, Finding the vertex connectivity of graphs, SIAM Journal on Computing 9 (1) (1980) 197–199.
[97] N. Linial, L. Lovász, A. Wigderson, Rubber bands, convex embeddings, and graph connectivity, Combinatorica 8 (1) (1988) 91–102.
[98] Arjeh M. Cohen, Scott H. Murray, Martin Pollet, Volker Sorge, Certifying solutions to permutation group problems, in: Franz Baader (Ed.), CADE, in: Lecture Notes in Computer Science, vol. 2741, Springer, 2003, pp. 258–273.
[99] Sergey Bereg, Certifying and constructing minimally rigid graphs in the plane, in: SCG '05: Proceedings of the Twenty-first Annual Symposium on Computational Geometry, ACM, New York, NY, USA, 2005, pp. 73–80.
[100] Konstantin A. Rybnikov, An efficient local approach to convexity testing of piecewise-linear hypersurfaces, Computational Geometry 42 (2) (2009) 147–172.
[101] Uriel Feige, Robert Krauthgamer, Finding and certifying a large hidden clique in a semirandom graph, Random Struct. Algorithms 16 (2) (2000) 195–208.
[102] Ross M. McConnell, A certifying algorithm for the consecutive-ones property, in: J. Ian Munro (Ed.), SODA, SIAM, 2004, pp. 768–777.
[103] M. Metzler, Ergebnisüberprüfung bei Graphenalgorithmen, Master's Thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1997.
[104] Pierre Dusart, Autour de la fonction qui compte le nombre de nombres premiers, Ph.D. thesis, Universit'e de Limoges, Limoges, France, 1998.
[105] D. Coppersmith, S. Winograd, On the asymptotic complexity of matrix multiplication, SIAM Journal on Computing 11 (1982) 472–492.
[106] R. Freiwalds, Probabilistic machines can use less running time, in: Information Processing 77, Proceedings of IFIP Congress 77, 1977, pp. 839–842.
[107] Tracy Kimbrel, Rakesh Kumar Sinha, A probabilistic algorithm for verifying matrix products using $o(n2)$ time and $\log 2n + o(1)$ random bits, Inf. Process. Lett. 45 (2) (1993) 107–110.
[108] E. Amaldi, C. Iuliano, T. Jurkiewicz, K. Mehlhorn, R. Rizzi, Breaking through the $O(m^2 n)$ Barrier for minimum cycle bases, in: ESA 2009, in: LNCS, vol. 5757, 2009, pp. 301–312.
[109] T. Kavitha, Ch. Liebchen, K. Mehlhorn, D. Michail, R. Rizzi, T. Ueckerdt, K. Zweig, Cycle bases in graphs: characterization, algorithms, complexity, and applications, Computer Science Review 3 (2009) 199–243.
[110] J.C. de Pina, Applications of shortest path methods, Ph.D. thesis, University of Amsterdam, Netherlands, 1995.
[111] J.D. Horton, A polynomial-time algorithm to find the shortest cycle basis of a graph, SICOMP 16 (1987) 358–366.
[112] M.N. Wegman, J.L. Carter, New hash functions and their use in authentication and set equality, Journal of Computer and System Sciences 22 (3) (1981) 265–279.
[113] P. Schweitzer, Problems of unknown complexity: graph isomorphism and Ramsey theoretic numbers, Ph.D. Thesis, Universität des Saarlandes, Saarbrücken, Germany, July 2009.
[114] D.R. Karger, C. Stein, A new approach to the minimum cut problem, Journal of the ACM 43 (4) (1996) 601–640.
[115] S. Arikati, K. Mehlhorn, A Correctness certificate for the Stoer–Wagner mincut algorithm, Information Processing Letters 70 (1999) 251–254.
[116] N.M. Amato, M.C. Loui, Checking linked data structures, in: Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing, FTCS'94, 1994, pp. 164–173.
[117] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, Moni Naor, Checking the correctness of memories, Algorithmica 12 (2/3) (1994) 225–244.
[118] U. Finkler, K. Mehlhorn, Checking priority queues, in: Proceedings of the 10th Annual ACM–SIAM Symposium on Discrete Algorithms, SODA'99, 1999, pp. 901–902.
[119] G. Brodal, S. Chaudhuri, J. Radhakrishnan, The randomized complexity of maintaining the minimum, Nordic Journal of Computing 3 (4) (1996) 337–351.
[120] K. Mehlhorn, P. Sanders, Algorithms and Data Structures: The Basic Toolbox, Springer, 2008, 300 pages.
[121] Isabelle theorem prover. http://isabelle.in.tum.de/.
[122] VCC, a mechanical verifier for concurrent C programs. http://vcc.codeplex.com/.
[123] Andreas Brandstädt, Dieter Kratsch, Haiko Müller (Eds.), Graph-Theoretic Concepts in Computer Science, 33rd International Workshop, WG 2007, Dornburg, Germany, June 21–23, 2007. Revised Papers, in: Lecture Notes in Computer Science, vol. 4769, Springer, 2007.