Trustworthy Graph Algorithms

Mohammad Abdulaziz
TU München

Kurt Mehlhorn
MPI for Informatics

Tobias Nipkow
TU München

Abstract

The goal of the LEDA project was to build an easy-to-use and extendable library of correct and efficient data structures, graph algorithms and geometric algorithms. We report on the use of formal program verification to achieve an even higher level of trustworthiness. Specifically, we report on an ongoing and largely finished verification of the blossom-shrinking algorithm for maximum cardinality matching.

2012 ACM Subject Classification Design and Analyis of Algorithms; Software Verification and Validation

Keywords and phrases Graph Algorithms, Correctness, Formal Methods, Software Libraries

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

This talk is a follow-up on two previous invited MFCS-talks given by the second author:

- *LEDA: A Library of Efficient Data Types and Algorithms* in MFCS 1989 [32], and
- *From Algorithms to Working Programs: On the Use of Program Checking in LEDA* in MFCS 1998 [34].

After a review of these papers, we discuss the further steps taken to reach even higher trustworthiness of our implementations.

- Formal correctness proofs of checker programs [5, 40], and
- Formal verification of complex graph algorithms [1].

The second item is the technical core of the paper: it reports on the ongoing and largely finished verification of the blossom-shrinking algorithm for maximum cardinality matching in Isabelle/HOL by the first author.

Personal Note by the Second Author: As this paper spans 30 years of work, the reader might get the impression that I followed a plan. This is not the case. As a science, in this case computer science, progresses, there are logical next steps. I took these steps. I did not know 30 years ago, where the journey would lead me.

2 Level One of Trustworthiness: The LEDA Library of Efficient Data Types and Algorithms

In 1989, Stefan Näher and the second author set out to build an easy-to-use and extendable library of correct and efficient data structures, graph algorithms and geometric algorithms. The project was announced in an invited talk at MFCS 1989 [32] and the library is available from Algorithmic Solutions GmbH [28]. LEDA, the library of efficient data types and

```
template <class NT>
void DIJKSTRA_T(const graph& G, node s, const edge_array<NT>& cost,
                  node_array<NT>& dist, node_array<edge>& pred)
{
  node_pq<NT>  PQ(G);   // a priority queue for the nodes of G
  node v; edge e;
  dist[s] = 0;          // distance from s to s is zero
  PQ.insert(s,0);       // insert s with value 0 into PQ
  forall_nodes(v,G) pred[v] = nil; // no incoming tree edge yet
  while (!PQ.empty())   // as long as PQ is non-empty
  { node u = PQ.del_min();   // let u be the node with minimum dist in PQ
    NT du = dist[u];         // and du its distance
    forall_adj_edges(e,u)    // iterate over all edges e out of u
    { v = G.opposite(u,e);   // makes it work for ugraphs
      NT c = du + cost[e];   // distance to v via u
      if (pred[v] == nil && v != s )            // v already reached?
        PQ.insert(v,c);                         // first path to v
      else if (c < dist[v]) PQ.decrease_p(v,c); // better path
           else continue;
      dist[v] = c;           // store distance value
      pred[v] = e;           // and incoming tree edge
    }
   }
  }
```

Figure 1 The LEDA implementation of Dijkstra's algorithm: Note that the executable code above is similar to a typical pseudo-code presentation of the algorithm.

algorithms, offers a flexible data type graph with loops for iterating over edges and nodes and arrays indexed by nodes and edges. It also offers the data types required for graph algorithms such as queues, stacks, and priority queues. It thus created a framework in which graph algorithms can be formulated easily and naturally, see Figure 1 for an example. The design goal was to create a system in which the difference between the pseudo-code used to explain an algorithm and what constitutes an executable program is as small as possible. The expectation was that this would ease the burden of the implementer and make it easier to get implementations correct.

3 Level Two of Trustworthiness: Certifying Algorithms

Nevertheless, some implementations in the initial releases were incorrect, in particular, the planarity test[1]; it declared some planar graphs non-planar. At around 1995, we adopted the concept of certifying algorithms [34, 31] for the library and reimplemented all algorithms [35]. A certifying algorithm computes for each input a easy-to-check certificate (witness) that demonstrates to the user that the output of the program for this particular input is correct; see Figure 2. For example, the certifying planarity test returns a Kuratowski subgraph if it

[1] Most of the implementations of the geometric algorithms were also incorrect in their first release as we had naïvely used floating point arithmetic to implement real arithmetic and the rounding errors invalidated the implementations of the geometric primitives. This lead to the development of the exact computation paradigm for geometric computing by us and others [21, 46, 14, 45, 33]. In this paper, we restrict to graph algorithms.

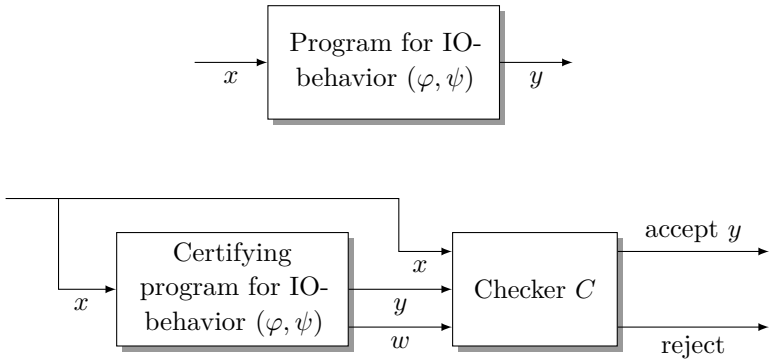

Figure 2 The top figure shows the I/O behavior of a conventional program for IO-behavior (φ, ψ); here φ is the precondition and ψ is the postcondition. The user feeds an input x satisfying $\varphi(x)$ to the program and the program returns an output y satisfying $\psi(x, y)$. A certifying algorithm for IO-behavior (φ, ψ) 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 postcondition $\psi(x, y)$, i.e., it proves $\psi(x, y)$. (reprinted from [5])

declares the input graph non-planar and a (combinatorial) planar embedding if it declares the input graph planar, and the maximum cardinality matching algorithm computes a matching and an odd-set-cover that proves its optimality; see Figures 3 and 4. The state of the art of certifying algorithms is described in [31]. We also implemented checker programs that check the witness for correctness and argued that the checker programs are so simple that their correctness is evident. From a pragmatic point of view, the goals of the project were reached by 2010. The library was easy-to-use and extendable, the implementations were efficient, and no error was discovered in any of the graph algorithms for several years despite intensive use by a commercial and academic user community.

Note that, most likely, errors would not have gone undiscovered because of the use of certifying algorithms and checker programs. Only if a module produced an incorrect output and hence an invalid certificate and the checker program missed to uncover the invalidity of the certificate would an error go unnoticed. Of course, the possibility is there and the phrase "most likely" in the preceding sentence has no mathematical meaning.

Alternative libraries such as Boost and LEMON [44, 29] are available now and some of their implementations are slightly more efficient than ours. However, none of the new libraries pays the same attention to correctness. For example, all libraries allow floating point numbers as weights and capacities in network algorithms, but only LEDA ensures that the intricacies of floating point arithmetic do not invalidate the implementations; see [6] and [35, Section 7.2].

4 Level Three of Trustworthiness: Formal Verification of Checkers

We stated above that the checker programs are so simple that their correctness is evident. Shouldn't they then be amenable to formal verification? Harald Ganzinger and the second author attempted to do so at around 2000 and failed. About 10 years later (2011 – 2014) Eyad Alkassar from the Verisoft Project [43], Sascha Böhme and Lars Noschinski from Tobias Nipkow's group at TU München, and Christine Rizkallah and the second author succeeded in formally verifying some of the checker programs [5, 40]. In order to be able to talk about formal verification of checker programs, we need to take a more formal look at certifying algorithms.

A *matching* in a graph G is a subset M of the edges of G such that no two share an endpoint.

An odd-set cover OSC of G is a labeling of the nodes of G with non-negative integers such that every edge of G (which is not a self-loop) is either incident to a node labeled 1 or connects two nodes labeled with the same i, $i \geq 2$.

Let n_i be the number of nodes labeled i and consider any matching N. For i, $i \geq 2$, let N_i be the edges in N that connect two nodes labeled i. Let N_1 be the remaining edges in N. Then $|N_i| \leq \lfloor n_i/2 \rfloor$ and $|N_1| \leq n_1$ and hence

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

for any matching N and any odd-set cover OSC. It can be shown that for a maximum cardinality matching M there is always an odd-set cover OSC with

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

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

list<*edge*> MAX_CARD_MATCHING(*graph* G, *node_array*<*int*>& OSC)

computes a maximum cardinality matching M in G and returns it as a list of edges. The algorithm ([12], [15]) has running time $O(nm \cdot \alpha(n, m))$.
An odd-set cover that proves the maximality of M is returned in OSC.

bool CHECK_MAX_CARD_MATCHING(*graph* G, *list*<*edge*> M, *node_array*<*int*> OSC)

checks whether M is a maximum cardinality matching in G and OSC is a proof of optimality. Aborts if this is not the case.

Figure 3 The LEDA manual page for maximum cardinality matchings (reprinted from [34]).

We consider algorithms which take an input from a set X and produce an output in a set Y and a witness in a set W. The input $x \in X$ is supposed to satisfy a precondition $\varphi(x)$, and the input together with the output $y \in Y$ is supposed to satisfy a postcondition $\psi(x, y)$. A *witness predicate* for a specification with precondition φ and postcondition ψ is a predicate $\mathcal{W} \subseteq X \times Y \times W$, where W is a set of witnesses with the following *witness property*:

$$\varphi(x) \wedge \mathcal{W}(x, y, w) \longrightarrow \psi(x, y). \tag{1}$$

The checker program C receives a triple[2] (x, y, w) and is supposed to check whether it fulfills the witness property. If $\neg\varphi(x)$, C may do anything (run forever or halt with an arbitrary output). If $\varphi(x)$, C must halt and either accept or reject. It is required to accept if $\mathcal{W}(x, y, w)$ holds and is required to reject otherwise. This results in the following proof obligations.

Checker Correctness: We need to prove that C checks the witness predicate assuming that the precondition holds, i.e., on input (x, y, w):

(i) If $\varphi(x)$, C halts.
(ii) If $\varphi(x)$ and $\mathcal{W}(x, y, w)$, C accepts (x, y, w), and if $\varphi(x)$ and $\neg\mathcal{W}(x, y, w)$, C rejects the triple.

[2] We ignore the minor complication that X, Y, and W are abstract sets and programs handle concrete representations.

```
static bool return_false(string s)
{ cerr << "CHECK_MAX_CARD_MATCHING: " << s << "\n"; return false; }

bool CHECK_MAX_CARD_MATCHING(const graph& G, const list<edge>& M,
                                    const node_array<int>& OSC)
{ int n = Max(2,G.number_of_nodes());
  int K = 1;
  array<int> count(n);
  for (int i = 0; i < n; i++) count[i] = 0;
  node v; edge e;

  forall_nodes(v,G)
  { if ( OSC[v] < 0 || OSC[v] >= n )
     return_false("negative label or label larger than n - 1");
    count[OSC[v]]++;
    if (OSC[v] > K) K = OSC[v];
  }

  int S = count[1];
  for (int i = 2; i <= K; i++) S += count[i]/2;
  if ( S != M.length() )
    return_false("OSC does not prove optimality");

  forall_edges(e,G)
  { node v = G.source(e); node w = G.target(e);
    if ( v == w || OSC[v] == 1 || OSC[w] == 1 ||
            ( OSC[v] == OSC[w] && OSC[v] >= 2) ) continue;
    return_false("OSC is not a cover");
  }
  return true;
}
```

Figure 4 The checker for maximum cardinality matchings (reprinted from [34]).

Witness Property: We need to prove implication (1).

In case of the maximum cardinality matching problem, the witness property states that an odd-set cover OSC as defined in Figure 3 with $|M| = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor$ proves that the matching M has maximum cardinality. Checker correctness amounts to the statement that the program shown in Figure 4 is correct.

We proved the witness property using Isabelle/HOL [38]. For the checker correctness, we used VCC [9] and later Simpl [42] and AutoCorres [16]. The latter approach has the advantage that the entire verification can be performed within Isabelle. Simpl is a generic imperative programming language embedded into Isabelle/HOL, which was designed as an intermediate language for program verification. We implemented checkers both in Simpl and C. Checkers written in Simpl were verified directly within Isabelle. For the checkers written in C, we first translated from C to Isabelle using the C-to-Isabelle parser that was developed as part of the seL4 project [22], and then used the AutoCorres tool developed at NICTA that simplifies reasoning about C in Isabelle/HOL. Christine spent several months at NICTA to learn how to use the tool. We verified the checkers for connectivity, maximum cardinality matching, and non-planarity. In particular, for the non-planarity checker it was essential that Lars Noschinski in parallel formalized basic graph theory in Isabelle [39].

A disclaimer is in order here. We did not verify the C++ program shown in Figure 4. Rather we verified a manual translation of this program into Simple or C, respectively. For this translation, we assumed a very basic representation of graphs. The nodes are numbered from 0 to $n-1$, the edges are numbered from 0 to $m-1$ with the edges incident to any vertex numbered consecutively and arrays of the appropriate dimension are used for cross-referencing and for encoding adjacency lists.

The verification attempt for the maximum cardinality checker shown in Figure 4 discovered a flaw. Note that the program does not check whether the edges in M actually belong to G. When we wrote the checker, we apparently took this for granted. The verification attempt revealed the flaw.

We also considered going further and briefly tried to verify the LEDA maximum cardinality matching algorithm [35, Section 7.7]. The program has 330 lines of code and the description of the algorithm, its implementation and its correctness proof spans over 20 pages. We found the task too daunting and, extrapolating from the effort required for the verification of the checkers, estimated the effort as several man-years.

5 Level Four of Trustworthiness: Formal Verification of Complex Algorithms

A decade later, we perform the formal verification of the blossom-shrinking algorithm for maximum cardinality. We give a short account of the verification which will be described in detail in our forthcoming publication [1]. On a high-level Edmond's blossom-shrinking algorithm [12] works as follows. The algorithm repeatedly searches for an augmenting path with respect to the current matching. Initially, the current matching is empty. Whenever an augmenting path is found, augmentation of the path increases the size of the matching by one. If no augmenting path exists with respect to the current matching, the current matching has maximum cardinality.

The search for an augmenting path is via growing alternating trees rooted at free vertices, i.e. vertices not incident to an edge of the matching. The search is initialised by making each free vertex a root of an alternating tree; the matched nodes are in no tree initially. In an alternating tree, vertices at even depth are entered by a matching edge, vertices at odd depth are entered by a non-matching edge, and all leaves have even depth. In each step of the growth process, one considers a vertex, say u_1, of even depth that is incident to an edge $\{u_1, u_2\}$ not considered before. If u_2 is not in a tree yet, then one adds u_2 (at odd level) and its mate (at even level) under the current matching to the tree. If u_2 is already in a tree and has odd level then one does nothing as one simply has discovered another odd length path to u_2. If u_2 is already in a tree and has even level then one has either discovered an augmenting path (if u_2 is in a different tree than u_1) or a blossom (if u_2 and u_1 are in the same tree). In the latter case, consider the tree paths from u_2 and u_1 back to their common root and let u_3 be the lowest common ancestor of u_2 and u_1. The edge $\{u_1, u_2\}$ plus the tree paths from u_1 and u_2 to u_3 form an odd length cycle. One collapses all nodes on the cycle into a single node and repeats the search for an augmenting path in the quotient (= shrunken) graph. If an augmenting path is found in the quotient graph, it is lifted (refined) to an augmenting path in the original graph. If no augmenting path exists in the quotient graph, no augmenting path exists in the original graph. In this section, we describe in detail the algorithm outlined above, and the process of formalising and verifying it in Isabelle/HOL.

5.1 Isabelle/HOL

Isabelle/HOL [41] is a theorem prover for classical Higher-Order Logic. Roughly speaking, Higher-Order Logic can be seen as a combination of functional programming with logic. Isabelle's syntax is a variation of Standard ML combined with (almost) standard mathematical notation. Application of function f to arguments $x_1\ \ldots\ x_n$ is written as $f\ x_1\ \ldots\ x_n$ instead of the standard notation $f(x_1,\ \ldots\ , x_n)$. We explain non-standard syntax in the paper where it occurs.

Isabelle is designed for trustworthiness: following the LCF approach [36], a small kernel implements the inference rules of the logic, and, using encapsulation features of ML, it guarantees that all theorems are actually proved by this small kernel. Around the kernel, there is a large set of tools that implement proof tactics and high-level concepts like algebraic data types and recursive functions. Bugs in these tools cannot lead to inconsistent theorems being proved since they all rely on the kernel only, but only to error messages when the kernel refuses a proof. Isabelle/HOL comes with a rich set of already formalized theories, among which are natural numbers and integers as well as sets and finite sets.

5.2 Preliminaries

An edge is a set of vertices with size 2. A graph $\mathcal{G}$ is a set of edges. A set of edges $\mathcal{M}$ is a matching iff $\forall e, e' \in \mathcal{M}.\ e \cap e' = \emptyset$. In Isabelle/HOL that is represented as follows:

```
matching M ⟷ (∀ e1 ∈ M. ∀ e2 ∈ M. e1 ≠ e2 ⟶ e1 ∩ e2 = {})
```

In may cases, a matching is a subset of a graph, in which case we call it a matching w.r.t. the graph. For a graph $\mathcal{G}$, $\mathcal{M}$ is a maximum matching w.r.t. $\mathcal{G}$ iff for any matching $\mathcal{M}'$ w.r.t. $\mathcal{G}$, we have that $|\mathcal{M}'| \leq |\mathcal{M}|$.

5.3 Formalising Berge's Lemma

A list of vertices $u_1u_2\ldots u_n$ is a path w.r.t. a graph $\mathcal{G}$ iff every $\{u_i, u_{i+1}\} \in \mathcal{G}$. A path $u_1u_2\ldots u_n$ is a simple path iff for every $1 \leq i \neq j \leq n$, $u_i \neq u_j$. A list of vertices $u_1u_2\ldots u_n$ is an alternating path w.r.t. a set of edges E iff for some E' (i) $E' = E$ or $E' = \{e \mid e \notin E\}$, (ii) $\{u_i, u_{i+1}\} \in E'$ holds for all even numbers i, where $1 \leq i < n$, and (iii) $\{u_i, u_{i+1}\} \notin E'$ holds for all odd numbers i, where $1 \leq i \leq n$. We call a list of vertices $u_1u_2\ldots u_n$ an augmenting path w.r.t. a matching $\mathcal{M}$ iff $u_1u_2\ldots u_n$ is an alternating path w.r.t. $\mathcal{M}$ and $u_1, u_n \notin \bigcup \mathcal{M}$. It is often the case that an augmenting path γ w.r.t. to a matching $\mathcal{M}$ is also a simple path w.r.t. a graph $\mathcal{G}$, in which case we call the path an augmenting path w.r.t. to the pair $\langle \mathcal{G}, \mathcal{M} \rangle$. Also, for two sets s and t, $s \oplus t$ denotes the symmetric difference of the two sets. We overload $\oplus$ to arguments which are lists in the obvious fashion.

▷ Theorem 1 (Berge's Lemma). For a graph $\mathcal{G}$, a matching $\mathcal{M}$ is maximum w.r.t. $\mathcal{G}$ iff there is not an augmenting path γ w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$.

Our proof of Berge's lemma is shorter than the standard proof. The standard proof consists of three steps. First, for any two matchings $\mathcal{M}$ and $\mathcal{M}'$, every connected component of the graph $\mathcal{M} \oplus \mathcal{M}'$ is either (i) a singleton vertex, (ii) a path, or (iii) a cycle. Second, for a set of edges $C \subseteq \mathcal{M} \oplus \mathcal{M}'$ s.t. $|C \cap \mathcal{M}| < |C \cap \mathcal{M}'|$, the edges from C form a path. Thirds, such a set C of edges exists, if $|\mathcal{M}| < |\mathcal{M}'|$. We observe that it is much easier to directly show that such a C exists and that all its edges can be arranged in a path, without having to prove the first step about all connected components. We found this different proof during the process of formalising the theorem, and finding this shorter proof was primarily motivated

by making the formalisation shorter and more feasible. The discovery of simpler proofs or more general theorem statements is one potential positive outcome of verifying algorithms, and mathematics in general, in interactive theorem provers [3, 2, 10].

Algorithm 1: FIND_MAX_MATCHING($\mathcal{G}, \mathcal{M}$)

$\gamma :=$ AUG_PATH_SEARCH($\mathcal{G}, \mathcal{M}$)
if γ is some augmenting path
  return FIND_MAX_MATCHING($\mathcal{G}, \mathcal{M} \oplus \gamma$)
else
  return $\mathcal{M}$

Now consider Algorithm 1. Berge's lemma implies the validity of that algorithm as a method to compute maximum matchings in graphs. The validity of Algorithm 1 is stated in the following corollary.

▷ Corollary 1. Assume that AUG_PATH_SEARCH($\mathcal{G}, \mathcal{M}$) is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$, for any graph $\mathcal{G}$ and matching $\mathcal{M}$, iff $\mathcal{G}$ has an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$. Then, for any graph $\mathcal{G}$, FIND_MAX_MATCHING($\mathcal{G}, \emptyset$) is a maximum matching w.r.t. $\mathcal{G}$.

As shown in Corollary 1, Algorithm 1 depends on the function AUG_PATH_SEARCH which is a sound and a complete procedure to compute augmenting paths in graphs.

In Isabelle/HOL, the first step is to formalise the path concepts from above. Paths and alternating paths are defined recursively in a straightforward fashion. An augmenting path is defined as follows:

```
augmenting_path M p ≡ (length p ≥ 2) ∧ alt_path M p
                      ∧ hd p ∉ Vs M ∧ last p ∉ Vs M
```

The formalised statement of Berge's lemma is as follows:

```
theorem Berge:
  assumes
    finite M and matching M and M ⊆ E
    and
    (∀e∈E. ∃u v. e = {u,v} ∧ u ≠ v) and finite (Vs E)
  shows (∃p. augmenting_path M p ∧ path E p ∧ distinct p) ⟷
          (∃M' ⊆ E. matching M' ∧ card M < card M')
```

Note that in the formalisation when the paths need to be simple, such as in Berge's lemma above, we have the additional assumption that all vertices are pairwise distinct, denoted by the Isabelle/HOL predicate `distinct`. Just to clarify Isabelle's syntax: the lemma above has two sets of assumptions, one on the matching and the other on the graph. The matching has to be a finite set, which is a matching w.r.t. the given graph. The graph has to have edges which only have two vertices, and its set of vertices has to be finite.

In Isabelle/HOL Algorithm 1 is formalised within the following *locale*.

```
locale find_max_match =
  fixes aug_path_search::'a set set ⇒ 'a set set ⇒ ('a list) option and
    E
  assumes
    aug_path_search_complete:
    matching M ∧ M ⊆ E ∧ finite M ∧
      (∃p. path E p ∧ distinct p ∧ augmenting_path M p)
```

```
      ⟹ (∃p. aug_path_search E M = Some p)
  and
  aug_path_search_sound:
  matching M ∧ M ⊆ E ∧ finite M ∧ aug_path_search E M = Some p ⟹
          path E p ∧ distinct p ∧ augmenting_path M p
  and
  graph: ∀e∈E. ∃u v. e = {u, v} ∧ u ≠ v finite (Vs E)
```

A locale is a named context: definitions and theorems proved within locale `find_max_match` can refer to the parameters and assumptions declared there. In this case, we need the locale to identify the parameter `aug_path_search` of the locale, corresponding to the function Aug_Path_Search, which is used in Algorithm 1. The function `aug_path_search` should take as input a graph and a matching. It should return an `('a list) option` typed value. Generally speaking, the value of an `'a option` valued term could be in one of two forms: either `Some x`, or `None`, where `x` is of type `'a`. In the case of `aug_path_search`, it should return either `Some p`, where `p` is a path in case an augmenting path is found, or `None`, otherwise. There is also the function `the`, which given a term of type `'a option`, returns `x`, if the given term is `Some x`, and which is undefined otherwise. Within that locale, the definition of Algorithm 1 and its verification theorem are as follows. Note that the correction theorem has four conclusions: the algorithm returns a subset of the graph, that subset is a matching, that matching is finite and the cardinality of any other matching is bounded by the size of the returned matching.

```
find_max_matching M =
   (if (∃p. aug_path_search E M = Some p) then
      (find_max_matching (M ⊕ (set (edges_of_path (the (aug_path_search E M))))))
    else M)

lemma find_max_matching_works:
  shows (find_max_matching {}) ⊆ E
    matching (find_max_matching {})
    finite (find_max_matching {})
    ∀M. matching M ∧ M ⊆ E ∧ finite M ⟶ card M ≤ card (find_max_matching {})
```

Functions defined within a locale are parameterised on the constants which are declared in the locale's definition. When a function is used outside a locale, these parameters must be specified. So, if `find_max_matching` is used outside the locale above, it should take as a parameter a function which computes augmenting paths. Similarly, theorems proven within a locale implicitly have the assumptions of the locale. So if use the lemma `find_max_matching_works`, we would have to prove that the functional argument to `find_max_matching` satisfies the assumptions of the locale, i.e. that argument is a sound and complete procedure for computing augmenting paths. The way theorems from locales are used will be clearer in the next section when we refer to the function `find_max_matching` and use the lemma `find_max_matching_works` outside of the locale `find_max_match`. The use of locales for performing gradual refinement of algorithms allows to focus on the specific aspects of the algorithm relevant to a refinement stage, with the rest of the algorithm abstracted away.

5.4 Verifying that Blossom Contraction Works

In Corollary 1, which specifies the soundness of Find_Max_Matching, we have not explicitly specified the function Aug_Path_Search. Indeed, we have only specified what its

output has to conform to. We now refine that specification and describe Aug_Path_Search algorithmically.

Firstly, for a function f and a set s, let $f(\!|s|\!)$ denote the image of f on s. Also, for a set of edges E, and a function f, the quotient E/f is the set $\{f(\!|e|\!) \mid e \in E\}$. We now introduce the concepts of a *blossom*. A list of vertices $u_1u_2 \dots u_n$ is called a cycle if $3 < n$ and $u_n = u_1$, and we call it an odd cycle if n is even. A pair $\langle u_1u_2 \dots u_{i-1}, u_iu_{i+1} \dots u_n \rangle$ is a blossom w.r.t. a matching $\mathcal{M}$ iff (i) $u_iu_{i+1} \dots u_n$ is an odd cycle, (ii) $u_1u_2 \dots u_n$ is an alternating path w.r.t. $\mathcal{M}$, and (iii) $u_1 \notin \bigcup \mathcal{M}$. We also refer to $u_1u_2 \dots u_i$ as the stem of the blossom. In many situations we have a pair $\langle u_1u_2 \dots u_{i-1}, u_iu_{i+1} \dots u_n \rangle$ which is a blossom w.r.t. a matching $\mathcal{M}$ where $u_1u_2 \dots u_{i-1}u_iu_{i+1} \dots u_{n-1}$ is also a simple path w.r.t. a graph $\mathcal{G}$ and $\{u_{n-1}, u_n\} \in \mathcal{G}$. In this case we call it a blossom w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$.

Based on the above definitions, we prove that contracting (i.e. shrinking) the odd cycle of a blossom preserves the existence of an augmenting path, which is the second main result needed to prove the validity of the blossom-shrinking algorithm, after Berge's lemma.

▷ Theorem 2. Consider a graph $\mathcal{G}$ and a vertex $u \notin \bigcup \mathcal{G}$. Let for a set s, the function P_s be defined as $P_s(x) = \mathsf{if}\ x \in s\ \mathsf{then}\ u\ \mathsf{else}\ x$. Then, for a blossom $\langle \gamma, C \rangle$ w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$, if s is the set of vertices in C, then we have an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$ iff there is an augmenting path w.r.t. $\langle \mathcal{G}/P_s, \mathcal{M}/P_s \rangle$.

Theorem 2 is used in most expositions of the blossom-shrinking algorithm. In our proof for the forward direction (if an augmenting path exists w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$, then there is an augmenting path w.r.t. $\langle \mathcal{G}/P_s, \mathcal{M}/P_s \rangle$, i.e. w.r.t. the quotients), we follow a standard textbook approach [23]. In our proof for the backward direction (an augmenting path w.r.t. the quotients can be lifted to an augmenting path w.r.t. the original graph) we define an (almost) executable function refine that does the lifting.[3] We took the choice of explicitly defining that function with using it in the final algorithm in mind. This is similar to the approach used in the informal proof of soundness of the variant of the blossom-shrinking algorithm used in LEDA [35].

Now, using Theorem 2, one can show that Algorithm 2 is a sound and complete procedure for computing augmenting paths.

Algorithm 2: Aug_Path_Search$(\mathcal{G}, \mathcal{M})$

```
if Compute_Blossom(G, M) is a blossom ⟨γ, C⟩ w.r.t. ⟨G, M⟩
  return refine(Aug_Path_Search(G/P_C, M/P_C))
else if Compute_Blossom(G, M) is an augmenting path w.r.t. ⟨G, M⟩
  return Compute_Blossom(G, M)
else
  return no augmenting path found
```

The soundness and completeness of this algorithm assumes that Compute_Blossom can successfully compute a blossom or an augmenting path in a graph iff either one exists. This is formally stated as follows.

▷ Corollary 2. Assume that, for a graph $\mathcal{G}$ and a matching $\mathcal{M}$ w.r.t. $\mathcal{G}$, there is a blossom or an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$ iff Compute_Blossom$(\mathcal{G}, \mathcal{M})$ is a blossom or an augmenting

[3] The function refine, as defined later, is executable except for a choice operation.

path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$. Then for any graph $\mathcal{G}$ and matching $\mathcal{M}$, Aug_Path_Search$(\mathcal{G}, \mathcal{M})$ is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$ iff there is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$.

To formalise that in Isabelle/HOL, an odd cycle and a blossom are defined as follows:

```
odd_cycle p ≡ (length p ≥ 3) ∧ odd (length (edges_of_path p)) ∧ hd p = last p

blossom M stem C ≡ alt_path M (stem @ C) ∧
  distinct (stem @ (butlast C)) ∧ odd_cycle C ∧ hd (stem @ C) ∉ Vs M ∧
  even (length (edges_of_path (stem @ [hd C])))
```

In the above definition `@` stands for list concatenation and `edges_of_path` is a function which, given a path, returns the list of edges constituting the path.

To define the function refine that refines a quotient augmenting path to a concrete one or to formalise the theorems showing that contracting blossoms preserves augmenting paths we first declare the following locale:

```
locale quot =
  fixes P s u
  assumes ∀v∈s. P v = v and u∉s and (∀v. v∉s ⟶ P v = u)
```

That locale fixes a function `P`, a set of vertices `s` and a vertex `u`. The function `P` maps all vertices from `s` to the given vertex `u`.

Now, we formalise the function refine which lifts an augmenting path in a quotient graph to an augmenting path in the concrete graph. The function refine takes an augmenting path p in the quotient graph and returns it unchanged if it does not contain the vertex u and deletes u and splits p into two paths p_1 and p_2 otherwise. In the latter case, p_1 and p_2 are passed to `replace_cycle`. This function first defines two auxiliary paths `stem2p2` and `p12stem` using the function `stem2vert_path`. Let us have a closer look at the path `stem2p2`. `stem2vert_path` with last argument `hd p2` uses `choose_con_vert` to find a neighbor of `hd p2` on the cycle C. It splits the cycle at this neighbor and then returns the path leading to the base of the blossom starting with a matching edge. Finally, `replace_cycle` glues together p_1, p_2 and either `stem2p2` and `p12stem` to obtain an augmenting path in the concrete graph.

```
choose_con_vert vs E v ≡ (SOME v'. v' ∈ vs ∧ {v, v'} ∈ E)

stem2vert_path C E M v ≡
let find_pfx' = (λC. find_pfx ((=) (choose_con_vert (set C) E v)) C) in
  if (last (edges_of_path (find_pfx' C)) ∈ M) then
    (find_pfx' C)
  else
    (find_pfx' (rev C))

replace_cycle C E M p1 p2 ≡
 let stem2p2 = stem2vert_path C E M (hd p2);
     p12stem = stem2vert_path C E M (last p1) in
 if p1 = [] then
   stem2p2 @ p2
 else
   (if p2 = [] then
      p12stem @ (rev p1)
    else
     (if {u, hd p2} ∉ quotG M then
```

```
        p1 @ stem2p2 @ p2
      else
        (rev p2) @ p12stem @ (rev p1)))

  refine C E M p ≡
   if (u ∈ set p) then
     (replace_cycle C E M (fst (pref_suf [] u p)) (snd (pref_suf [] u p)))
   else p
```

In Isabelle/HOL the two directions of the equivalence in Theorem 2 are formalised as follows:

```
theorem quot_apath_to_apath:
  assumes
    odd_cycle C and alt_path M C and distinct (tl C) and path E C
    and
    augmenting_path (quotG M) p' and distinct p' and path (quotG E) p'
    and
    matching M and M ⊆ E
    and
    s = (Vs E) - set C
    and
    ∀e∈E. ∃u v. e = {u, v} ∧ u ≠ v
  shows augmenting_path M (refine C E M p') ∧ path E (refine C E M p') ∧
        distinct (refine C E M p')

theorem aug_path_works_in_contraction:
  assumes
    path E (stem @ C) and blossom M stem C
    and
    augmenting_path M p and path E p and distinct p
    and
    matching M and M ⊆ E and finite M
    and
    s = (Vs E) - set C and u ∉  Vs E
    and
    ∀e∈E. ∃u v. e = {u, v} ∧ u ≠ v and finite (Vs E)
  shows ∃p'. augmenting_path (quotG M) p' ∧ path (quotG E) p' ∧ distinct p'
```

A main challenge with formalising Theorem 2 in Isabelle/HOL is the lack of automation for handling symmetries in its proof.

To formalise Algorithm 2 we use a locale to assume the existence of the function which computes augmenting paths or blossoms, iff either one exist. That function is called *blos_search* in the locale declaration. Its return type and the assumptions on it are as follows:

```
datatype 'a blossom_res =
    Path (aug_path: "'a list") | Blossom (stem_vs: "'a list") (cycle_vs: "'a list")

bloss_algo_complete:
  (((∃p. path E p ∧ distinct p ∧ augmenting_path M p)
        ∨ (matching M ∧ (∃stem C. path E (stem @ C) ∧ blossom M stem C))))
     ⟹ (∃blos_comp. blos_search E M = Some blos_comp)

bloss_algo_sound:
```

```
(∀e∈E. ∃u v. e = {u, v} ∧ u ≠ v) ∧ blos_search E M = Some (Path p)
    ⟹ (path E p ∧ distinct p ∧ augmenting_path M p)
blos_search E M = Some (Blossom stem C)
    ⟹ (path E (stem @ C) ∧ (matching M ⟶ blossom M stem C))
```

The locale also fixes a function *create_vert* which creates new vertex names to which vertices from the odd cycle are mapped during contraction. Within that locale, we define Algorithm 2 and prove its soundness and completeness theorems, which are as follows:

```
quotG E ≡ (quot_graph P E) - {{u}}

find_aug_path E M =
   (case blos_search E M of Some blossom_res ⇒
      case blossom_res of Path p ⇒ Some p
      | Blossom stem cyc ⇒
          let u = create_vert (Vs E);
              s = Vs E - (set cyc);
              quotG = quot.quotG (quot_fun s u) u;
              refine = quot.refine (quot_fun s u) u cyc E M
          in (case find_aug_path (quotG E) (quotG M) of Some p' ⇒ Some (refine p')
              | _ ⇒ None)
    | _ ⇒ None)
```

lemma *find_aug_path_sound:*
  assumes
    matching M **and** *M ⊆ E* **and** *finite M*
    and
    ∀e∈E. ∃u v. e = {u, v} ∧ u ≠ v **and** *finite (Vs E)*
    and
    find_aug_path E M = Some p
  shows *augmenting_path M p ∧ path E p ∧ distinct p*

lemma *find_aug_path_complete:*
  assumes
    augmenting_path M p **and** *path E p* **and** *distinct p*
    and
    matching M **and** *M ⊆ E* **and** *finite M*
    and
    ∀e∈E. ∃u v. e = {u, v} ∧ u ≠ v **and** *finite (Vs E)"*
  shows *∃p'. find_aug_path E M = Some p'*

Note that in *find_aug_path*, we instantiate both arguments *P* an *s* of the locale *quot* to obtain the quotienting function *quotG* and the function for refining augmenting path *refine*.

Lastly, what follows shows the validity of instantiating the functional argument of *find_max_matching* with *find_aug_path*, which gives us the following soundness theorem of the resulting algorithm.

lemma *find_max_matching_works:*
  assumes
    finite (Vs E) **and** *∀e∈E. ∃u v. e = {u, v} ∧ u ≠ v*
  shows
    find_max_match.find_max_matching find_aug_path E {} ⊆ E
    matching (find_max_match.find_max_matching find_aug_path E {})
    finite (find_max_match.find_max_matching find_aug_path E {})
    ∀M. matching M ∧ M ⊆ E ∧ finite M
        ⟶ card M ≤ card (find_max_match.find_max_matching find_aug_path E {})

5.5 Computing Blossoms and Augmenting Paths

Until now, we have only assumed the existence of the function COMPUTE_BLOSSOM, which can compute augmenting paths or blossoms, if any exist in the graph. We now refine that to an algorithm which, given two alternating paths resulting from the ascent of alternating trees, returns either an augmenting path or a blossom.

We first introduce some notions and notation. For a list l, let $|l|$ be the length of l. For a list l and a natural number n, let $\mathsf{drop}\ n\ l$ denote the list l, but with the first n elements dropped. For a list l, let $h :: l$ denote adding an element h to the front of a list l. For a non-empty list l, let $\mathsf{first}\ l$ and $\mathsf{last}\ l$ denote the first and last elements of l, respectively. Also, for a list l, let $\mathsf{rev}\ l$ denote its reverse. For two lists l_1 and l_2, let $l_1 \frown l_2$ denote their concatenation. Also, let $\mathsf{longest\_disj\_pref}\ l_1\ l_2$ denote the pair of lists $\langle l_1', l_2' \rangle$, where l_1' and l_2' are the longest disjoint prefixes of $l1$ and l_2, respectively, s.t. $\mathsf{last}\ l_1' = \mathsf{last}\ l_2'$. Note: $\mathsf{longest\_disj\_pref}\ l_1\ l_2$ is only well-defined if there is are l_1', l_2', and l s.t. $l_1 = l_1' \frown l$ and $l_2 = l_2' \frown l$, and if both l_1' and l_2' are disjoint except at their endpoints.

We now are able to state the following two lemmas concerning the construction of a blossom or an augmenting path given paths resulting from alternating trees search.

▷ Lemma 1. If γ_1 and γ_2 are both (i) simple paths w.r.t. $\mathcal{G}$, (ii) alternating paths w.r.t. $\mathcal{M}$, and (iii) of odd length, and if we have that (iv) $\mathsf{last}\ \gamma_1 = \mathsf{last}\ \gamma_2$, (v) $\mathsf{last}\ \gamma_1 \notin \bigcup \mathcal{M}$, (vi) $\{\mathsf{first}\ \gamma_1, \mathsf{first}\ \gamma_2\} \in \mathcal{G}$, (vii) $\{\mathsf{first}\ \gamma_1, \mathsf{first}\ \gamma_2\} \notin \mathcal{M}$, and (viii) $\mathsf{longest\_disj\_pref}\ \gamma_1\ \gamma_2$ is well-defined and $\langle \gamma_1', \gamma_2' \rangle = \mathsf{longest\_disj\_pref}\ \gamma_1\ \gamma_2$, then $\langle \mathsf{rev}(\mathsf{drop}\ (|\gamma_1'| - 1)\ \gamma_1), (\mathsf{rev}\ \gamma_1') \frown \gamma_2' \rangle$ is a blossom w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$.

▷ Lemma 2. If γ_1 and γ_2 are both (i) simple paths w.r.t. $\mathcal{G}$, (ii) alternating paths w.r.t. $\mathcal{M}$, (iii) of odd length, and (iv) disjoint, and if we have that (v) $\mathsf{last}\ \gamma_1 \notin \bigcup \mathcal{M}$, (vi) $\mathsf{last}\ \gamma_2 \notin \bigcup \mathcal{M}$, (vii) $\mathsf{last}\ \gamma_1 \neq \mathsf{last}\ \gamma_2$, (viii) $\{\mathsf{first}\ \gamma_1, \mathsf{first}\ \gamma_2\} \in \mathcal{G}$, and (ix) $\{\mathsf{first}\ \gamma_1, \mathsf{first}\ \gamma_2\} \notin \mathcal{M}$, then $(\mathsf{rev}\ \gamma_1) \frown \gamma_2$ is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$.

Based on the above lemmas we refine the algorithm COMPUTE_BLOSSOM as shown in Algorithm 3.

Algorithm 3: COMPUTE_BLOSSOM$(\mathcal{G}, \mathcal{M})$

```
if ∃e ∈ G.e ∩ ⋃M = ∅
  return Augmenting path choose {e | e ∈ G ∧ e ∩ ⋃M = ∅}
else if compute_alt_path(G, M) = ⟨γ1, γ2⟩
  if last γ1 ≠ last γ2
    return Augmenting path (rev γ1)⌢γ2
  else
    ⟨γ1', γ2'⟩ = longest_disj_pref γ1 γ2
    return Blossom ⟨rev(drop (|γ1'| − 1) γ1), (rev γ1')⌢γ2'⟩
else
  return No blossom or augmenting path found
```

The following corollary shows the conditions under which COMPUTE_BLOSSOM works.

▷ Corollary 3. Assume the function $\mathsf{compute\_alt\_path}(\mathcal{G}, \mathcal{M})$ returns two lists of vertices $\langle \gamma_1, \gamma_2 \rangle$ s.t. both lists are (i) simple paths w.r.t. $\mathcal{G}$, (ii) alternating paths w.r.t. $\mathcal{M}$, and (iii) of odd length, and also (iv) $\mathsf{last}\ \gamma_1 \notin \bigcup \mathcal{M}$, (v) $\mathsf{last}\ \gamma_2 \notin \bigcup \mathcal{M}$, (vi) $\{\mathsf{first}\ \gamma_1, \mathsf{first}\ \gamma_2\} \in \mathcal{G}$, and (vii) $\{\mathsf{first}\ \gamma_1, \mathsf{first}\ \gamma_2\} \notin \mathcal{M}$, iff two lists of vertices with those properties exist. Then there is

a blossom or an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$ iff COMPUTE_BLOSSOM$(\mathcal{G}, \mathcal{M})$ is a blossom or an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$.

In Isabelle/HOL, to formalise the function COMPUTE_BLOSSOM, we firstly defined a function, *longest_disj_pfx*, which finds the longest common prefix in a straightforward fashion with a quadratic wort-case runtime. The formalised versions of Lemma 1 and 2, which show that the output of *longest_disj_pfx* can be used to construct a blossom or an augmenting path are as follows:

```
lemma common_pfxs_form_blossom:
  assumes
    (Some pfx1, Some pfx2) = longest_disj_pfx p1 p2"
    and
    p1 = pfx1 @ p and p2 = pfx2 @ p"
    and
    alt_path M p1 and alt_path M p2 and last p1 ∉ Vs M and {hd p1, hd p2} ∈ M"
    and
    hd p1 ≠ hd p2"
    and
    even (length p1) and even (length p2)
    and
    distinct p1 and distinct p2
    and
    matching M
  shows blossom M (rev (drop (length pfx1) p1)) (rev pfx1 @ pfx2)

lemma construct_aug_path:
  assumes
    set p1 ∩ set p2 = {}
    and
    p1 ≠ [] and p2 ≠ []
    and
    alt_path M p1 and alt_path M p2 and last p1 ∉ Vs M and last p2 ∉ Vs M
    and
    {hd p1, hd p2} ∈ M and
    and
    even (length p1) and even (length p2)
  shows augmenting_path M ((rev p1) @ p2)
```

The function COMPUTE_BLOSSOM is formalised as follows:

```
"compute_blossom G M ≡
 (if (∃e. e ∈ unmatched_edges G M) then
       let
         singleton_path =
            (SOME p. ∃v1 v2. p = [v1 ,v2] ∧ {v1, v2} ∈ unmatched_edges G M)
       in
         Some (Path singleton_path)
  else
   case compute_alt_path G M
     of Some (p1,p2) ⇒
       (if (set p1 ∩ set p2 = {}) then
          Some (Path ((rev p1) @ p2))
        else
          (let
```

```
          (pfx1, pfx2) = longest_disj_pfx p1 p2;
          stem = (rev (drop (length (the pfx1)) p1));
          cycle = (rev (the pfx1) @ (the pfx2))
        in
         (Some (Blossom stem cycle))))
  | _ ⇒ None)"
```

We use a locale again to formalise that function. That locale parameterises it on a function that searches for alternating paths and poses the soundness and completeness assumptions for that alternating path search function. This function is equivalent to the unspecified function compute_alt_path in Corollary 3 and locale's assumptions on it are formalised statements of the seven assumptions on compute_alt_path in Corollary 3.

5.6 Computing Alternating Paths

Lastly, we refine the function compute_alt_path to an algorithmic specification. The algorithmic specification of that function performs the alternating tree search, see Algorithm 4. If the function positively terminates, i.e. finding two vertices with even labels, returns two alternating paths by ascending the two alternating trees to which the two vertices belong. This tree ascent is performed by the function follow . That function takes a functional argument f and a vertex, and returns the singleton list $[u]$ if $f(u) = \text{None}$, and $u :: (\text{follow } f\ (f(u)))$ otherwise.

Algorithm 4: compute_alt_path$(\mathcal{G}, \mathcal{M})$

```
ex = ∅ // Set of examined edges
foreach u ∈ ⋃𝒢
    label u = None
    parent u = None
U = ⋃𝒢 \ ⋃ℳ // Set of unmatched vertices
foreach u ∈ U
    label u = ⟨u, even⟩
while (𝒢 \ ex) ∩ {e | ∃u ∈ e, r ∈ ⋃𝒢.label u = ⟨r, even⟩} ≠ ∅
    // Choose a new edge and labelled it examined
    {u1, u2} = choose (𝒢 \ ex) ∩ {{u1, u2} | ∃r.label u1 = ⟨r, even⟩}
    ex = ex ∪ {{u1, u2}}
    if label u2 = None
        // Grow the discovered set of edges from r by two
        u3 = choose {u3 | {u2, u3} ∈ ℳ}
        ex = ex ∪ {{u2, u3}}
        label u2 = ⟨r, odd⟩; label u3 = ⟨r, even⟩; parent u2 = u1; parent u3 = u2
    else if ∃s ∈ ⋃𝒢.label u2 = ⟨s, even⟩
        // Return two paths from current edge's tips to unmatched vertex(es)
        return ⟨follow parent u1, follow parent u2⟩
return No paths found
```

The soundness and completeness of Algorithm 4 is stated as follows.

▷ Theorem 3. The function compute_alt_path$(\mathcal{G}, \mathcal{M})$ returns two lists of vertices $\langle \gamma_1, \gamma_2 \rangle$ s.t. both lists are (i) simple paths w.r.t. $\mathcal{G}$, (ii) alternating paths w.r.t. $\mathcal{M}$, and (iii) of

odd length, and also (iv) $\mathsf{last}\ \gamma_1 \notin \bigcup \mathcal{M}$, (v) $\mathsf{last}\ \gamma_2 \notin \bigcup \mathcal{M}$, (vi) $\{\mathsf{first}\ \gamma_1, \mathsf{first}\ \gamma_2\} \in \mathcal{G}$, and (vii) $\{\mathsf{first}\ \gamma_1, \mathsf{first}\ \gamma_2\} \notin \mathcal{M}$, iff two lists of vertices with those properties exist.

The primary difficulty with proving this theorem is identifying the loop invariants, which are as follows:

(i) For any vertex u, if for some r, $\mathsf{label}\ u = \langle r, \mathsf{even}\rangle$, then the vertices in the list $\mathsf{follow\ parent}\ u$ have labels that alternate between $\langle r, \mathsf{even}\rangle$ and even $\langle r, \mathsf{odd}\rangle$.
(ii) For any vertex u_1, if for some r and some l, we have $\mathsf{label}\ u_1 = \langle r, l\rangle$, then the list $\mathsf{follow\ parent}\ u_1$, made of the vertices list $u_1 u_2 \ldots u_n$, has the following property: if $\mathsf{label}\ u_i = \langle r, \mathsf{even}\rangle$ and $\mathsf{label}\ u_{i+1} = \langle r, \mathsf{odd}\rangle$, for some r, then $\{u_i, u_{i+1}\} \in \mathcal{M}$, otherwise, $\{u_i, u_{i+1}\} \notin \mathcal{M}$.
(iii) The relation induced by the function parent is well-founded.
(iv) For any $\{u_1, u_2\} \in \mathcal{M}$, $\mathsf{label}\ u_1 = \text{None}$ iff $\mathsf{label}\ u_2 = \text{None}$.
(v) For any u_1, if $\mathsf{label}\ u_1 = \text{None}$ then $\mathsf{parent}\ u_2 \neq u_1$, for all u_2.
(vi) For any u, if $\mathsf{label}\ u \neq \text{None}$, then $\mathsf{last}\ (\mathsf{follow\ parent}\ u) \notin \bigcup \mathcal{M}$.
(vii) For any u, if $\mathsf{label}\ u \neq \text{None}$, then $\mathsf{label}\ (\mathsf{last}\ (\mathsf{follow\ parent}\ u)) = \langle r, \mathsf{even}\rangle$, for some r.
(viii) For any $\{u_1, u_2\} \in \mathcal{M}$, if $\mathsf{label}\ u_1 \neq \text{None}$, then $\{u_1, u_2\} \in \mathsf{ex}$.
(ix) For any u, $\mathsf{follow\ parent}\ u$ is a simple path w.r.t. $\mathcal{G}$.
(x) Suppose we have two vertex lists γ_1 and γ_2, s.t. both lists are (i) simple paths w.r.t. $\mathcal{G}$, (ii) alternating paths w.r.t. $\mathcal{M}$, and (iii) of odd length, and also (iv) $\mathsf{last}\ \gamma_1 \notin \bigcup \mathcal{M}$, (v) $\mathsf{last}\ \gamma_2 \notin \bigcup \mathcal{M}$, (vi) $\{\mathsf{first}\ \gamma_1, \mathsf{first}\ \gamma_2\} \in \mathcal{G}$, and (vii) $\{\mathsf{first}\ \gamma_1, \mathsf{first}\ \gamma_2\} \notin \mathcal{M}$. Then there is at least an edge from the path $\mathsf{rev}\ \gamma_1 @ \gamma_2$ which is a member of neither $\mathcal{M}$ nor ex.[4]

To formalise Algorithm 4 in Isabelle/HOL, we first define the function which follows a vertex's parent as follows:

```
follow v = (case (parent v) of Some v' ⇒ v # (follow v') | _  ⇒ [v])
```

Again, we use a locale to formalise that function, and that locale fixes the function *parent*. Note that the above function is not well-defined for all possible arguments. In particular, it is only well-defined if the relation between pairs of vertices induced by the function *parent* is a well-founded relation. This assumption on *parent* is a part of the locale's definition.

Then, we then formalise compute_alt_path as follows:

```
compute_alt_path ex par flabel =
   (if (∃ v1 v2. {v1, v2} ∈ G - ex ∧ (∃ r. flabel v1 = Some (r, Even))) then
      let
        (v1,v2) = (SOME (v1,v2). {v1, v2} ∈ G - ex ∧
                                 (∃ r. flabel v1 = Some (r, Even)));
        ex' = insert {v1, v2} ex;
        r = (SOME r. flabel v1 = Some (r, Even))
      in
        (if flabel v2 = None ∧ (∃ v3. {v2, v3} ∈ M) then
          let
            v3 = (SOME v3. {v2, v3} ∈ M);
            par' = par(v2 := Some v1, v3 := Some v2);
            flabel' = flabel(v2 := Some (r, Odd), v3 := Some (r, Even));
```

[4] The hypothesis of this invariant is equivalent to the existence of an augmenting path or a blossom w.r.t. $\langle \mathcal{G}, \mathcal{M}\rangle$.

```
          ex'' = insert {v2, v3} ex';
          return = compute_alt_path ex'' par' flabel'
        in
          return
      else if ∃r. flabel v2 = Some (r, Even) then
         let
           r' = (SOME r'. flabel v2 = Some (r', Even));
           return = Some (parent.follow par v1, parent.follow par v2)
         in
           return
      else
        let
           return = None
        in
          return)
  else
    let
      return = None
    in
      return)
```

Note that we do not use a while combinator to represent the while loop: instead we formalise it recursively, passing the context along recursive calls. In particular, we define it as a recursive function which takes as arguments the variables representing the state of the while loop, namely, the set of examined edges `ex`, the parent function `par`, and the labelling function `flabel`.

5.7 Discussion

The algorithm in LEDA differs from the description above in one aspect. If no augmenting path is found, an odd-set cover is constructed proving optimality. Also the correctness proof uses the odd-set cover instead of the fact that an augmenting path exists in the original graph if and only if one exists in the quotient graph.

For an efficient implementation, the shrinking process and the lifting of augmenting paths are essential. The shrinking process is implemented using a union-find data structure and the lifting is supported by storing additional information with the edge that closes the cycle in a blossom [35].

6 Level Five of Trustworthiness: Extraction of Efficient Executable Code

In this section we examine the process of obtaining trustworthy executable and efficient code from algorithms verified in theorem provers. First we discuss the problem in general and then we examine our formalization of the blossom-shrinking algorithm.

Most theorem provers are connected to a programming language of some sort. Frequently, as in the case of Isabelle/HOL, that programming language is a subset of the logic and close to a functional programming language. The theorem prover will usually support the extraction of actual code in some programming language. Isabelle/HOL supports Standard ML, Haskell, OCaml and Scala.

To show that code extraction "works", here are some random non-trivial examples of verifications that have resulted in reasonably efficient code: Compilers for C [30] and for

ML [24], a SPIN-like model checker [13], network flow algorithms [27] and the Berlekamp-Zassenhaus factorization algorithm [11].

We will now discuss some approaches to obtaining code from function definitions in a theorem prover. In the ACL2 theorem prover all functions are defined in a purely functional subset of Lisp and are thus directly executable. In other systems, code generation involves an explicit translation step. The trustworthiness of this step varies. Probably the most trustworthy code generator is that of HOL4, because its backend is a verified compiler for CakeML [24], a dialect of ML. The step from HOL to CakeML is not verified once and for all, but every time it is run it produces a theorem that can be examined and that states the correctness of this run [37]. The standard code generator in Isabelle/HOL is unverified (although its underlying theory has been proved correct on paper [19]). There is ongoing work to replace it with a verified code generator that produces CakeML [20].

So far we have only considered purely functional code but efficient algorithms often make use of imperative features. Some theorem provers support imperative languages directly, e.g. Java [4]. We will now discuss how to generate imperative code from purely functional one. Clearly the code generator must turn the purely functional definitions into more imperative ones. The standard approach [7, 37] is to let the code generator recognize monadic definitions (a purely functional way to express imperative computations) and implement those imperatively. This is possible because many functional programming languages do in fact offer imperative features as well.

Just as important as the support for code extraction is the support for verified stepwise refinement of data types and algorithms by the user. Data refinement means the replacement of abstract data types by concrete efficient ones, e.g. sets by search trees. Algorithm refinement means the stepwise replacement of abstract high-level definitions that may not even be executable by efficient implementations. Both forms of refinement are supported well in Isabelle/HOL [17, 25, 26].

We conclude this section with a look at code generation from our formalization of the blossom-shrinking algorithm. It turns out that our formalization is almost executable as is. The only non-executable construct we used is `SOME x. P` that denotes some arbitrary `x` that satisfies the predicate `P`. Of course one can hide arbitrarily complicated computations in such a contruct but we have used it only for simple nondeterministic choices and it will be easy to replace. For example, one can obtain an executable version of function `choose_con_vert` (see Section 5.4) by defining a function that searches the vertex list `vs` for the first `v'` such that `{v, v'} ∈ E`. This is an example of algorithm refinement. To arrive at efficient code for the blossom-shrinking algorithm as a whole we will need to apply both data and algorithm refinement down to the imperative level. At least the efficient implementations referred to above, just before Section 5.1, are intrinsically imperative.

Finally let us note that instead of code generation it is also possible to verify existing code in a theorem prover. This was briefly mentioned in Section 4 and Charguéraud [8] has followed this approach quite successfully.

7 The Future

The state of the art in the verification of complex algorithms has improved enormously over the last decade. Yet there is still a lot to do on the path to a verified library such as LEDA. Apart from the shere amount of material that would have to be verified there is the challenge of obtaining trustworthy code that is of comparable efficiency. This requires trustworthy code generation for a language such C or C++, including the memory management. This is

a non-trivial task, but some of the pieces of the puzzle, like a verified compiler, are in place already.

References

1 Mohammad Abdulaziz, Kurt Mehlhorn, and Tobias Nipkow. A correctness proof of Edmonds' blossom shrinking algorithm for maximum matchings in graphs. forthcoming.

2 Mohammad Abdulaziz, Michael Norrish, and Charles Gretton. Formally verified algorithms for upper-bounding state space diameters. *J. Autom. Reasoning*, 61(1-4):485–520, 2018.

3 Mohammad Abdulaziz and Lawrence Paulson. An Isabelle/HOL formalisation of Green's theorem. *J. Autom. Reasoning, In Press*.

4 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

5 E. Alkassar, S. Böhme, K. Mehlhorn, and Ch. Rizkallah. A Framework for the Verification of Certifying Computations. *Journal of Automated Reasoning (JAR)*, 52(3):241–273, 2014. A preliminary version appeared under the title "Verification of Certifying Computations" in CAV 2011, LCNS Vol 6806, pages 67 – 82.

6 E. Althaus and K. Mehlhorn. Maximum Network Flow with Floating Point Arithmetic. *Information Processing Letters*, 66:109–113, 1998.

7 Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with Isabelle/HOL. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008*, pages 134–149, 2008.

8 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 418–430, 2011.

9 Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

10 Sander R Dahmen, Johannes Hölzl, and Robert Y Lewis. Formalizing the Solution to the Cap Set Problem. *arXiv:1907.01449*, 2019.

11 Jose Divasón, Sebastiaan J. C. Joosten, René Thiemann, and Akihisa Yamada. A verified implementation of the Berlekamp-Zassenhaus factorization algorithm. *J. Autom. Reasoning*, 2019. published online.

12 J. Edmonds. Maximum matching and a polyhedron with 0,1 - vertices. *Journal of Research of the National Bureau of Standards*, 69B:125–130, 1965.

13 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A fully verified executable LTL model checker. In *Computer Aided Verification - 25th International Conference, CAV 2013*, pages 463–478, 2013.

14 S. Fortune. Robustness issues in geometric algorithms. In *Proceedings of the 1st Workshop on Applied Computational Geometry: Towards Geometric Engineering (WACG'96)*, volume 1148 of Lecture Notes in Computer Science, pages 9–13, 1996.

15 H. N. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *Journal of the ACM*, 23:221–234, 1976.

16 David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of C. In *Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 99–115, 2012.

17 Florian Haftmann, Alexander Krauss, Ond*v*rej Kun*v*car, and Tobias Nipkow. Data refinement in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *LNCS Series*, pages 100–115, 2013.

18 Florian Haftmann and Tobias Nipkow. A code generator framework for Isabelle/HOL. *Theorem Proving in Higher Order Logics (TPHOLs 2007). Lecture Notes in Computer Science*, 4732:128–143, 2007.

19 Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.

20 Lars Hupel and Tobias Nipkow. A verified compiler from Isabelle/HOL to CakeML. In A. Ahmed, editor, *European Symposium on Programming (ESOP 2018)*, volume 10801 of *LNCS*, pages 999–1026. Springer, 2018.

21 L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom Examples of Robustness Problems in Geometric Computations. *Computational Geometry: Theory and Applications (CGTA)*, 40:61–78, 2008. a preliminary version appeared in ESA 2004, LNCS 3221, pages 702 – 713.

22 Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating-system kernel. *CACM*, 53(6):107–115, 2010.

23 B. Korte and J.Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 2012.

24 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, pages 179–192. ACM, 2014.

25 Peter Lammich. Automatic data refinement. In *Interactive Theorem Proving - 4th International Conference, ITP 2013*, pages 84–99, 2013.

26 Peter Lammich. Refinement to imperative HOL. *J. Autom. Reasoning*, 62(4):481–503, 2019.

27 Peter Lammich and S. Reza Sefidgar. Formalizing network flow algorithms: A refinement approach in isabelle/hol. *J. Autom. Reasoning*, 62(2):261–280, 2019.

28 LEDA (Library of Efficient Data Types and Algorithms). `www.algorithmic-solutions.com`.

29 LEMON graph library. COIN-OR project.

30 Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43:363–446, 2009.

31 R.M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

32 K. Mehlhorn and S. Näher. LEDA: A library of efficient data types and algorithms. In *MFCS'89*, volume 379 of Lecture Notes in Computer Science, pages 88–106, 1989.

33 K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *Proceedings of the 13th IFIP World Computer Congress*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

34 K. Mehlhorn and S. Näher. From algorithms to working programs: On the use of program checking in LEDA. In *MFCS'98*, volume 1450 of Lecture Notes in Computer Science, pages 84–93, 1998.

35 K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

36 Robin Milner. Logic for computable functions: description of a machine implementation. Technical report, Stanford University, 1972.

37 Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3):284–315, 2014.

38 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of Lecture Notes in Computer Science. Springer, 2002.

39 Lars Noschinski. A graph library for Isabelle. *Mathematics in Computer Science*, 9:22–39, 2015.

40 Lars Noschinski, Christine Rizkallah, and Kurt Mehlhorn. Verification of certifying computations through AutoCorres and Simpl. In *NASA Formal Methods*, volume 8430 of *Lecture Notes in Computer Science*, pages 46–61. 2014.

41 Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer, 1994.

42 Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

43 Verisoft. `http://www.verisoft.de/`, 2007.

44 Andrew Lumsdaine von Jeremy G. Siek, Lie-Quan Lee. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Professional, 2001.

45 C.-K. Yap. Towards exact geometric computation. *CGTA: Computational Geometry: Theory and Applications*, 7, 1997.

46 C.-K. Yap. Robust geometric computation. In J.E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41. CRC Press LLC, Boca Raton, FL, 2nd edition, 2003.