A Formal Correctness Proof of Edmonds' Blossom Shrinking Algorithm

Mohammad Abdulaziz[0000-0002-8244-518X]

King's College London, London, United Kingdom.

Contributing authors: mohammad.abdulaziz@kcl.ac.uk;

Abstract

We present the first formal correctness proof of Edmonds' blossom shrinking algorithm for maximum cardinality matching. We focus on formalising the mathematical structures and properties that allow the algorithm to run in worst-case polynomial running time. We formalise Berge's lemma for matching, blossoms and their properties, and a mathematical model of the algorithm, showing that it is totally correct. We provide the first detailed proofs of many of these properties.

Keywords: Formal Mathematics, Algorithm Verification, Matching Theory, Graph Theory, Combinatorial Optimisation

1 Introduction

Maximum cardinality matching is a basic problem in computer science, operational research, graph theory, and combinatorial optimisation. In this problem, given an undirected graph, one is to find the largest, in terms of cardinality, subset of vertex disjoint edges of that graph.

We describe the first formal functional correctness proof of Edmonds' blossom shrinking algorithm [1], which is an algorithm to solve the maximum cardinality matching problem in general graphs. We do the proof in the theorem prover Isabelle/HOL. Developing a formal correctness proof for this algorithm presents substantial challenges. First, the correctness argument depends on substantial graph theory, including results like Berge's lemma [2]. Second, it includes reasoning about graph contractions, with complex case analyses associated with reasoning about contractions. Third, the algorithm's core procedure is an involved iterative search procedure that builds a complex forest data structure. Proving its total correctness depends on a large number of complex loop invariants and on the construction of a complex certificate.

Our contributions here include:

1. developing substantial formal libraries for: undirected graphs, including reasoning principles for performing mathematical induction and treating connected components; alternating paths, a necessary concept for reasoning about matchings and matching algorithms; and matching theory, including Berge's lemma,
2. methodology-wise, we use Isabelle/HOL's function package to model all iterative computation; Isabelle/HOL's locales to perform algorithmic to structure our proofs; and Isabelle/HOL's classical reasoning to automate much of our proofs, showing that standard tools of Isabelle/HOL already suffice to perform reasoning about some of the most complex algorithms in a relatively elegant fashion, and
3. mathematically, despite the existence of many established expositions [3–5], we (a) provide the first complete case analyses of two central results: the decades old Berge's lemma and the fact that *blossom shrinking* preserves *augmenting path* existence; and (b) provide the first complete list of invariants and the first detailed correctness proof of the core search procedure.

We note that parts of this work were presented in a preliminary form in an earlier invited conference paper [6].

The structure of the paper will be as follows: we first discuss the necessary background notions. Then we have three technical sections, each dedicated to a major algorithmic part and to the mathematics behind its correctness. In each of those sections, in addition to presenting the formalisation, we present our own informal proofs, which we believe provide necessary mathematical insights for those who are interested in the formalisation, as well as those who are interested in the algorithm's correctness generally. Then we finish with a discussion section.

Algorithm 1 FIND_MAX_MATCHING($\mathcal{G}$)

```
M := ∅
γ := AUG_PATH_SEARCH(G, M)
while γ is an augmenting path do
    M := M ⊕ γ
    γ := AUG_PATH_SEARCH(G, M)
end while
return M
```

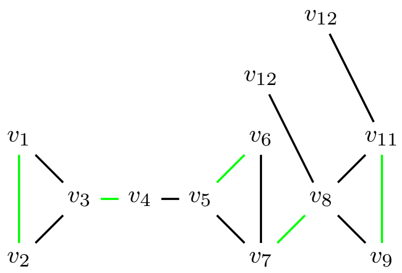

Fig. 1 An undirected graph. The green edges constitute a maximum cardinality matching.

The Algorithm

At a high-level, Edmond's blossom-shrinking algorithm [1] works as follows. The algorithm has a top loop that repeatedly searches for an augmenting path, i.e. a path whose edges alternate in terms of membership in the matching and that begins and ends at unmatched vertices. Initially, the current matching is empty. Whenever an augmenting path is found, augmentation of the matching using the found augmenting path increases the size of the matching by one. Augmentation is done by taking the symmetric difference between the matching and the edges in the augmenting path, which will always return a larger matching (see Fig. 2). If no augmenting path exists with respect to the current matching, the current matching has maximum cardinality.

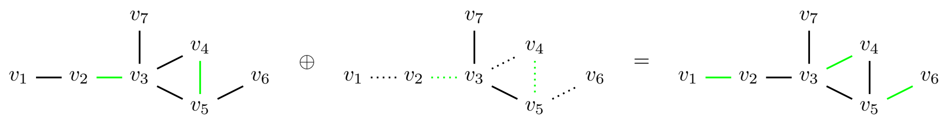

Fig. 2 A figure demonstrating how a matching can be augmented by an agumenting path. The current matching is labelled green. The dotted path is an augmenting path w.r.t that matching and the graph on the left. The new matching is on the right, with one more edge.

The previous loop is similar to the loops of many other algorithms for matching and, more generally, combinatorial optimisation problems. The main difficulty, however, for maximum cardinality matching in general graphs is that of searching for augmenting paths. For instance, one could think of using a modified breadth-first or depth-first search, which takes edges based on alternating membership in the matching, to find those paths. Although this works for finding augmenting paths in bi-partite graphs, it would not work for general graphs, as the presence of odd alternating cycles (henceforth, odd cycles) in the graph would necessitate keeping track of whether a given vertex has been visited through an edge in the matching, an edge not in the matching, or both types of edges. Furthermore, this has to be done for every path, which would make the algorithm run in an exponential worst-case running time.

Edmonds' blossom shrinking algorithm avoids this problem based on the following insight: *odd alternating cycles in the graph can be shrunk (i.e. contracted to one vertex) and the resulting graph has an augmenting path iff the original graph has one.*

This insight is then used to modify the general schema s.t. the search becomes for an augmenting path *or* an odd cycle. If the former is found, the matching is augmented. If the latter is found, the cycle is removed and the search continues. Crucially, this works for any odd cycle that is found, and thus no backtracking is needed, which preserves the polynomial worst-case running time. Having to prove that shrinking odd cycles preserves augmenting paths and the fact that the algorithm needs data structures to represent odd cycles, shrink them, and find them makes Edmond's blossom shrinking algorithm one of the harder algorithms to justify in the theory of combinatorial optimisation and efficient algorithms more generally.

Methodology

We implement our verification using Isabelle/HOL's *locale*s, which provide a mechanism to parametrically model algorithms and in a step-wise refinement approach. In this approach, at a given step, we define an algorithm and assume that some functions exist, along with desirable properties of those functions. Those properties are written as specifications. In the next step, we define a more detailed description of the assumed function and show that this description satisfies the specification. This is repeated until there are no significant assumptions left or, if there are any, only trivial ones from an algorithmic perspective are left, e.g. the existence of a function that chooses an arbitrary element of a finite set.

Availability

Our formalisation can be found online at https://github.com/mabdula/Isabelle-Graph-Library at a branch called blossom_jar.

2 Background

Isabelle/HOL Notation

Isabelle/HOL [7] is a theorem prover based on Higher-Order Logic. Isabelle's syntax is a variation of Standard ML combined with almost standard mathematical notation. Function application is written infix and functions are usually Curried (i.e. function f applied to arguments $x_1 \ \dots \ x_n$ is written as $f \ x_1 \ \dots \ x_n$ instead of the standard notation $f(x_1, \ \dots \ , x_n)$). We explain non-standard syntax in the paper where it occurs.

A central concept for our work is Isabelle/HOL *locale*s. A locale is a named context: definitions and theorems proved within the locale can refer to the parameters and assumptions declared in the heading of the locale. Here, for instance, we have the locale `graph_defs` fixing a graph `G` and the locale `graph_abs` that additionally assumes that the graph satisfies an invariant `graph_invar`. We extensively use locales, as shown in the rest of this paper, to structure the reasoning in our formalisation.

```
definition degree where
  "degree G v ≡ card' ({e. v ∈ e} ∩ G)"

definition degree where
  "degree G v ≡ card' ({e. v ∈ e} ∩ G)"

locale graph_defs =
  fixes G :: "'a set set"

definition "dblton_graph G ≡ (∀e∈G. ∃u v. e = {u, v} ∧ u ≠ v)"

abbreviation "graph_invar G ≡ dblton_graph G ∧ finite (Vs G)"

locale graph_abs =
  graph_defs +
  assumes graph: "graph_invar G"
```

Listing 1 Basic graph definitions: an undirected graph is formalised as a set of sets. We define a context `graph-abs`, where we fix a graph `G` and assume that it has the right type and every edge in it is a doubleton set and that it has a finite set of vertices.

Graphs

An *edge* is a set of vertices with size 2. A *graph* $\mathcal{G}$ is a finite set of edges. The *degree* of a vertex v in a graph $\mathcal{G}$, $d(\mathcal{G}, v)$, is $|\{e \mid e \in \mathcal{G} \wedge v \in e\}|$. Although there is a number of other formalisations of undirected graphs [8, 9], we started our own formalisation. We formalised the notion of an undirected graph as shown in Listing 1. The main reason we pursued this formalisation is its simplicity: we do not keep track of an explicit set of vertices for each graph. Instead, the graph's vertices are the union of the vertices of a graph's edges, $\bigcup \mathcal{G}$, denoted by $\mathcal{V}(\mathcal{G})$ (`Vs G`, in Isabelle). This representation has the advantage that one does not have to prove, with every change to the graph, that all graph's edges are incident on the new graph's associated set of vertices. Nonetheless, it has the disadvantage that it does not allow for graphs with isolated vertices. However, this constraint has not caused us any problems in this or in the many other projects in which we and others have used it [10–12].

```
context fixes G :: "'a set set" begin
inductive path where
  path0: "path []" |
  path1: "v ∈ Vs G ⟹ path [v]" |
  path2: "{v,v'} ∈ G ⟹ path (v'#vs) ⟹ path (v#v'#vs)"

definition "walk_betw G u p v
              ≡ (p ≠ [] ∧ path G p ∧ hd p = u ∧ last p = v)"

definition reachable where
  "reachable G u v = (∃p. walk_betw G u p v)"
```

Listing 2 A vertex path in an undirected graph is defined as an inductive predicate. We define based on that the notion of a walk between two vertices and the notion of reachability between two vertices.

A list of vertices $v_1v_2\dots v_n$ is a *path* w.r.t. a graph $\mathcal{G}$ iff every $\{v_i, v_{i+1}\} \in \mathcal{G}$. A path $v_1v_2\dots v_n$ is a *simple path* iff $v_i \neq v_j$, for every $1 \leq i \neq j \leq n$. We will denote the list of edges $\{v_1, v_2\}\{v_1, v_2\}\{v_1, v_2\}\dots\{v_{n-1}, v_n\}$ occurring in a path $v_1v_2\dots v_n$ by $E(v_1v_2\dots v_n)$. Paths are formally defined recursively in a straightforward fashion as shown in Listing 2. Simple paths in Isabelle are denoted by `distinct`, indicating that their vertices are pairwise distinct. A path $v_1v_2\dots v_n$ is called a *cycle* if $3 < n$ and $v_n = v_1$, and we call it an *odd cycle* if n is even. Note: in informal statements and proofs, we will overload set operations to lists in the obvious fashion.

```
definition connected_component where
  "connected_component G v = {v'. v' = v ∨ reachable G v v'}"

definition connected_components where
  "connected_components G =
     {vs. ∃v. vs = connected_component G v ∧ v ∈ (Vs G)}"

definition component_edges where
  "component_edges G C = {{x, y} | x y.  {x, y} ⊆ C ∧ {x, y} ∈ G}"

definition components_edges where
  "components_edges G =
     {component_edges G C| C. C ∈ connected_components G}"
```

Listing 3 Formal definitions of two different notions of connected components. The first is a connected component of vertices and the second is a connected component of edges.

The *connected component* of a vertex v, denoted by $\mathcal{K}(\mathcal{G}, v)$ is the set of vertices reachable from v in the graph $\mathcal{G}$. The *connected components* of a graph, denoted by $\mathfrak{K}(\mathcal{G})$, is $\{\mathcal{K}(\mathcal{G}, v) \mid v \in \mathcal{V}(\mathcal{G})\}$. We define a second notion, the *component edges*, which, for a set of vertices $\mathcal{V}$, denoted by $\mathcal{E}(\mathcal{V})$, is the set of edges incident on two vertices in $\mathcal{V}$. In Isabelle, these definitions are formalised as shown in Listing 3. The connection between the two notions, which is important for proofs that require the two perspectives on connected components, is characterised as follows.

Proposition 1. *For any graph $\mathcal{G}$, any two vertices v_1 and v_2, $\mathcal{E}(\mathcal{K}(\mathcal{G}, v_1)) \cap \mathcal{E}(\mathcal{K}(\mathcal{G}, v_2)) = \emptyset$, if $v_1 \neq v_2$.*

Another important property of connected components is the following case analysis.

Proposition 2. *For a graph $\mathcal{G}$ and an edge e ($= \{v, u\}$), if $K \in \mathfrak{K}(\{e\} \cup \mathcal{G})$, then one of the following holds:*

1. *$K \in \mathfrak{K}(\mathcal{G})$*
2. *$v \notin \mathcal{V}(\mathcal{G})$, $u \notin \mathcal{V}(\mathcal{G})$, and $K = \{v, u\}$,*
3. *$v \in \mathcal{V}(\mathcal{G})$ and $u \notin \mathcal{V}(\mathcal{G})$ and $K \in (\{u\} \cup \mathcal{K}(\mathcal{G}, v)) \cup (\mathfrak{K}(\mathcal{G}) \setminus \mathcal{K}(\mathcal{G}, v))$, or*
4. *$v \in \mathcal{V}(\mathcal{G})$, $u \in \mathcal{V}(\mathcal{G})$, $\mathcal{K}(\mathcal{G}, v) \neq \mathcal{K}(\mathcal{G}, u)$, and $K \in \{\mathcal{K}(\mathcal{G}, v) \cup \mathcal{K}(\mathcal{G}, u)\} \cup (\mathfrak{K}(\mathcal{G}) \setminus \mathcal{K}(\mathcal{G}, v) \setminus \mathcal{K}(\mathcal{G}, u))$.*

This case analysis is crucial for proving facts about connected components by induction on the graph, and it is generally reusable. One last property of connected components that we proved, and that is necessary for proving Berge's Lemma is the following.

Lemma 1. *If $K \in \mathfrak{K}(\mathcal{G})$ and, for every $v \in \mathcal{V}(\mathcal{G})$, $d(\mathcal{G}, v) \leq 2$, then there is a simple path γ s.t. γ has exactly all elements of K.*

Proof sketch. The proof is by induction on $\mathcal{G}$. Let all the variable names in the induction hypothesis (I.H.) be barred, e.g. the connected component is $\overline{K}$. The base case has an empty graph is straightforward. For the step case, we have as an assumption that $K \in \mathfrak{K}(\{e\} \cup \mathcal{G})$, for some e, where there are two vertices s.t. $e = \{v, u\}$. From Proposition 1, we have to consider the following four cases.

Case 1. In this case, we can immediately apply the I.H. with $\overline{K}$ assigned to K, and obtain $\overline{\gamma}$ that is a simple path w.r.t. $\mathcal{G}$ and that has all the vertices of K. Since $\mathcal{G} \subseteq \{e\} \cup \mathcal{G}$, $\overline{\gamma}$ is the required witness. □

Case 2. In this case, the required path is vu. □

Case 3. First, we apply the I.H. to $\mathcal{G}$, where $\overline{K}$ is instantiated with $\mathcal{K}(\mathcal{G}, v)$. We obtain $\overline{\gamma}$ that is a simple path w.r.t. $\mathcal{G}$. From the premises of the induction, we know that $d((\{e\} \cup \mathcal{G}), u) \leq 2$. That means that $d(\mathcal{G}, v) \leq 1$, which means that $\overline{\gamma}$ starts or ends with v. Thus u can be appended to either end of $\overline{\gamma}$ (the end at which v is located) and the resulting list of vertices is the required witness.

Case 4. In this case, we apply the I.H. twice, once to $\mathcal{K}(\mathcal{G}, v)$ and $\mathcal{G}$ and another to $\mathcal{K}(\mathcal{G}, u)$ and $\mathcal{G}$. We obtain two paths $\overline{\gamma}_v$ and $\overline{\gamma}_u$, where both are simple paths w.r.t. $\mathcal{G}$ and where the first has the vertices of $\mathcal{K}(\mathcal{G}, v)$ and the second has those of $\mathcal{K}(\mathcal{G}, u)$. Also each of the two paths has all the vertices of the corresponding connected component. Following a similar argument to Case 3, we have that v is either at beginning or at the end of $\overline{\gamma}_v$, and the same is true for u and $\overline{\gamma}_u$. The required witness path is $\overline{\gamma}_v \frown \overline{\gamma}_u$, s.t. v and u are adjacent. □

□

Remark 1. The proof of the above lemma is an example of proofs about connected components that use the case analysis implied by Proposition 2. It also has a theme that recurs often in the context of this present formalisation and other formalisations concerned with graph algorithms, namely, the occurrence of symmetries in proofs. Here, for instance 1. the third case has two symmetric cases, namely, whether v occurs at the beginning or at the end of $\overline{\gamma}$, and 2. the fourth case has four symmetric cases, where v and u occur in $\overline{\gamma}_v$ and $\overline{\gamma}_v$. In the formal setting, despite spending effort on devising lemmas capturing them, these symmetries caused the formal proofs to be much longer than the informal (e.g. both Proposition 2 and Lemma 1 required two hundred lines of formal proof scripts each).

```
definition matching where
  "matching M ⟷
     (∀e1 ∈ M. ∀e2 ∈ M. e1 ≠ e2 ⟹ e1 ∩ e2 = {})"
```

Listing 4 Formal definition of matchings.

Matchings

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 modelled as shown in Listing 4. In almost all relevant cases, a matching is a subset of a graph, in which case we call it a matching w.r.t. that graph. Given a matching $\mathcal{M}$, we say vertex v is (un)matched $\mathcal{M}$ iff $v \in \mathcal{V}(\mathcal{M})$ (does not hold). 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 $|\mathcal{M}'| \leq |\mathcal{M}|$.

```
inductive alt_list where
"alt_list P1 P2 []" |
"P1 x ⟹ alt_list P2 P1 l ⟹ alt_list P1 P2 (x#l)"

definition matching_augmenting_path where
  "matching_augmenting_path M p ≡
    (length p ≥ 2) ∧
     alt_list (λe. e ∉ M) (λe. e ∈ M) (edges_of_path p) ∧
     hd p ∉ Vs M ∧ last p ∉ Vs M"

abbreviation "graph_augmenting_path E M p ≡
  path E p ∧ distinct p ∧ matching_augmenting_path M p"
```

Listing 5 Formal definition of alternating paths.

Augmenting Paths

A list of vertices $v_1 v_2 \dots v_n$ is an *alternating path* w.r.t. a set of edges E iff for some E' we have that 1. $E' = E$ or $E' = \{e \mid e \notin E\}$, 2. $\{v_i, v_{i+1}\} \in E'$ holds for all even numbers i, where $1 \leq i < n$, and 3. $\{v_i, v_{i+1}\} \notin E'$ holds for all odd numbers

i, where $1 \leq i \leq n$. We call a list of vertices $v_1 v_2 \dots v_n$ an *augmenting path* w.r.t. a matching $\mathcal{M}$ iff $v_1 v_2 \dots v_n$ is an alternating path w.r.t. $\mathcal{M}$ and $v_1, v_n \notin \mathcal{V}(\mathcal{M})$. We call γ an augmenting path w.r.t. to the pair $\langle \mathcal{G}, \mathcal{M} \rangle$ iff it is an augmenting path w.r.t. to a matching $\mathcal{M}$ and is also a simple path w.r.t. a graph $\mathcal{G}$. 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.

```
lemma induct_alt_list012:
  assumes "alt_list P1 P2 l"
  assumes "T []"
  assumes "⋀x. P1 x ⟹ T [x]"
  assumes "⋀x y zs. P1 x ⟹ P2 y ⟹ T zs ⟹ T (x#y#zs)"
  shows "T l"

lemma alternating_length_balanced:
  assumes "alt_list P1 P2 l" "∀x∈set l. P1 x ⟷ ¬ P2 x"
  shows "length (filter P1 l) = length (filter P2 l) ∨
         length (filter P1 l) = length (filter P2 l) + 1"

lemma alternating_eq_iff_even:
  assumes "alt_list P1 P2 l" "∀x∈set l. P1 x ⟷ ¬ P2 x"
  shows "length (filter P1 l) = length (filter P2 l) ⟷ even
     (length l)"

lemma alternating_eq_iff_odd:
  assumes "alt_list P1 P2 l" "∀x∈set l. P1 x ⟷ ¬ P2 x"
  shows "length (filter P1 l) = length (filter P2 l) + 1 ⟷
     odd (length l)"
```

Listing 6 Basic principles of reasoning about alternating lists.

Alternating paths are formalised as shown Listing 5. We first define an inductive predicate characterising what it means for a list to alternate w.r.t. two predicates, and based on that define augmenting paths. We note that, since, as far as we are aware, this is the first formalisation of a substantial result involving alternating lists, it is worthwhile to display here the reasoning principles needed to reason about alternating lists. The reasoning principles are the induction principle `induct_alt_list012` and the other three lemmas in Listing 6 that relate the length of an alternating list to the predicate that holds for the last element of the alternating list. We note that these reasoning principles are enough to derive all facts we needed in the context of Edmonds' blossom algorithm as well other matching algorithms [10].

```
locale create_vert =
  fixes create_vert::"'a set ⇒ 'a"
  assumes create_vert_works: "finite vs ⟹ create_vert vs ∉ vs
    "

locale choose =
  fixes sel
  assumes sel: "⟦finite s; s ≠ {}⟧ ⟹ (sel s) ∈ s"
```

Listing 7 Two non-deterministic functions that we assume: one to create new vertices and the other to choose vertices.

```
definition
  "sel_edge G =(
     let v1 = sel (Vs G);
         v2 = sel (neighbourhood G v1)
     in
        {v1,v2})"

lemma sel_edge:
  assumes "graph_invar G" "G ≠ {}"
  shows "sel_edge G ∈ G"
```

Listing 8 Two non-deterministic functions that we assume: one to create new vertices and the other to choose vertices.

Nondeterminism

Edmonds' blossom shrinking algorithm has multiple computational steps that are most naturally modelled nondeterministically. In a locale-based approach to nondeterminism, an approach to model nondeterminism is to assume functions that perform nondeterministic computation steps. If the algorithm is to executed, those functions are to be instantiated with executable implementations. This includes functions to non-deterministically choose edges from a graph or a matching, a neighbour of a vertex, etc. In our approach to model that nondeterminism, however, we aspire to limit nondeterminism to a minimum, i.e. we wanted to assume a minimal number of nondeterministic functions in the verified algorithm. The only place where there is nondeterminism in the way we modelled the algorithm are in the locales `choose` and `create_vert`, shown in Listing 7. There, we assume the presence of a function that can choose an arbitrary vertex from a finite set of vertices. Based on those functions, we define all other nondeterministic functions used throughout the algorithm. Listing 8, for instance, shows how we define a function that chooses an arbitrary edge from a set of edges. Concentrating all nondeterminism that way aims at making the generation of an executable implementation a more straightforward process, requiring minimal modifications to our current formalisation.

3 The Top Loop

```
locale find_max_match = graph_abs G for G +
  fixes aug_path_search::"'a set set ⇒ 'a set set ⇒ ('a list)
      option"
  assumes
    aug_path_search_complete:
    "⟦matching M; M ⊆ G; finite M;
      (∃p. graph_augmenting_path G M p)⟧ ⟹
        (∃p. aug_path_search G M = Some p)" and
    aug_path_search_sound:
    "⟦matching M; M ⊆ G; finite M;
      aug_path_search G M = Some p⟧ ⟹
        graph_augmenting_path G M p"
```

Listing 9 An Isabelle/HOL locale showing the parameters on which the top loop of Edmonds' blossom algorithm is parameterised. Most notably, it assumes the presence of a function that returns an augmenting path, if one exists.

```
function (domintros) find_max_matching where
  "find_max_matching M =
     (case aug_path_search G M of Some p ⇒
        (find_max_matching (M ⊕ (set (edges_of_path p))))
      | _ ⇒ M)"
```

Listing 10 A recursive function modelling the top loop of Edmonds' blossom shrinking algorithm.

In Isabelle/HOL, Algorithm 1 is formalised in a parametric fashion within the Isabelle locale `find_max_match` whose header is shown in Listing 9. The algorithm itself is formalised as the recursive function shown in Listing 10. Recall that Algorithm 1 is parameterised over the function AUG_PATH_SEARCH, which is a function that searches for augmenting paths, if any exists. To formalise that, we identify `aug_path_search` as a parameter of the locale `find_max_match`, corresponding to the function AUG_PATH_SEARCH. The function `aug_path_search` should take as input a graph and a matching. It should return an `('a list) option` typed value, which would either be `Some p`, if a path `p` is found, or `None`, otherwise. 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` that, given a term of type `'a option`, returns `x`, if the given term is `Some x`, and which is undefined otherwise.

Functions defined within a locale are parameterised by 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 a function which computes augmenting paths as a parameter.

Similarly, theorems proven within a locale implicitly have the assumptions of the locale. So if we use the lemma `find_max_matching_works` (Listing 13) outside of the locale, 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 correct procedure for computing augmenting paths. The use of locales for performing gradual refinement of algorithms allows us to focus on the specific aspects of the algorithm relevant to a refinement stage, with the rest of the algorithm abstracted away.

3.1 Correctness

The correctness of Algorithm 1 depends on the assumed behaviour of Aug_Path_Search, i.e. Aug_Path_Search has to conform to the following specification in order for Algorithm 1 to be correct.

Specification 1. 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.* $\langle \mathcal{G}, \mathcal{M} \rangle$*.*

In the formalisation, this specification corresponds to the assumption on `find_aug_path` in the locale `find_max_match` shown in Listing 9.

The correctness proof of Algorithm 1 is mainly constituted of Berge's lemma [2], which justifies the total correctness of most maximum cardinality matching algorithms.

Lemma 2. *Consider two matchings w.r.t.* $\mathcal{G}$*,* $\mathcal{M}$*, and* $\mathcal{M}'$*. For a connected component* $\mathcal{K}$ *of the graph* $\mathcal{M} \oplus \mathcal{M}'$*, if* $|\mathcal{M}' \cap \mathcal{E}(\mathcal{K})| > |\mathcal{M} \cap \mathcal{E}(\mathcal{K})|$*, then* $\mathcal{E}(\mathcal{K})$ *can always be arranged into* γ*, a simple path w.r.t.* $\mathcal{M} \oplus \mathcal{M}'$*.*

Proof sketch. We perform a case analysis on $|\mathcal{K}|$. If $|\mathcal{K}| \leq 1$, the proof is trivial. For the case when $|\mathcal{K}| > 1$, we prove the theorem by contradiction. We consider the following two cases.

Case 1 (γ is a cycle). First, from the fact that $|\mathcal{M}' \cap \mathcal{E}(\mathcal{K})| > |\mathcal{M} \cap \mathcal{E}(\mathcal{K})|$ and since γ is an alternating path, we know that $\mathsf{hd}(\gamma) \in \mathcal{M}'$ and $\mathsf{last}(\gamma) \in \mathcal{M}'$, and the first and last edges of γ are both members of $\mathcal{M}'$. From the case assumption, we know that $\mathsf{hd}(path) = \mathsf{last}(\gamma)$. This implies that $2 \leq d(\mathcal{M}', \mathsf{last}(\gamma))$, which is a contradiction since $\mathcal{M}'$ is a matching.

Case 2 (There is $v \in \gamma \setminus \{\mathsf{hd}(\gamma), \mathsf{last}(\gamma)\}$ s.t. $3 \leq d(\mathcal{M} \oplus \mathcal{M}', v)$). Since $v \in \mathcal{V}(\mathcal{M} \oplus \mathcal{M}')$, then we have that either $2 \leq d(\mathcal{M}, v)$ or $2 \leq d(\mathcal{M}', v)$. In both cases we have a contradiction, since both $\mathcal{M}$ and $\mathcal{M}'$ are matchings.

□

Theorem 1 (Berge 1957 [2])**.** *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.* $\langle \mathcal{G}, \mathcal{M} \rangle$*.*

Proof sketch. ($\Rightarrow$) Assume there is a matching $\mathcal{M}'$, s.t. $|\mathcal{M}'| > |\mathcal{M}|$. Then there is a connected component $\mathcal{K} \in \mathfrak{K}(\mathcal{M} \oplus \mathcal{M}')$ s.t. $|\mathcal{M}' \cap \mathcal{E}(\mathcal{K})| > |\mathcal{M} \cap \mathcal{E}(\mathcal{K})|$. From Lemma 2, we have that all edges in $\mathcal{K}$ can be arranged in a path γ, s.t. the edges in γ alternate in membership of $\mathcal{M}$ and $\mathcal{M}'$. Also, the path will have more edges from $\mathcal{M}'$ than it does from $\mathcal{M}$. That means that γ starts and ends at vertices that are not in $\mathcal{M}$. Then γ is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$.

($\Leftarrow$) Suppose there is an augmenting path γ w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$. Then $\gamma \oplus \mathcal{M}$ is a matching w.r.t. $\mathcal{G}$ and $|\gamma \oplus \mathcal{M}| > |\mathcal{M}|$. We prove that by induction on γ. The proof is straightforward for the cases where $|\gamma| \leq 2$. For $|\gamma| \leq 2$, we want to show that the theorem holds for a path $v_1 v_2 v_3 \frown \gamma$. We apply the induction hypothesis with $\overline{\mathcal{M}} \equiv (\mathcal{M} \setminus \{\{v_2, v_3\}\}) \cup \{\{v_1, v_2\}\}$. Since $v_1 v_2 v_3 \frown \gamma$ is an alternating path w.r.t. $\mathcal{M}$, then $v_3 \frown \gamma$ is also an alternating path w.r.t. $\overline{\mathcal{M}}$. Also recall that $v_3 \notin \mathcal{V}(\overline{\mathcal{M}})$, by definition of $\overline{\mathcal{M}}$. Also $\mathsf{last}(v_3 \frown \gamma) \notin \mathcal{V}(\overline{\mathcal{M}})$, from the induction premises and by definition of $\overline{\mathcal{M}}$. Thus, by applying the induction hypothesis, we have that $\overline{\mathcal{M}} \oplus (v_3 \frown \gamma)$ is a matching with more edges than $\overline{\mathcal{M}}$. Note that, since $\overline{\mathcal{M}} = \mathcal{M} \oplus \{\{v_1, v_2\}, \{v_2, v_3\}\}$, which follows from the definition of $\overline{\mathcal{M}}$, we have that $\overline{\mathcal{M}'} \oplus (v_3 \frown \gamma) = \mathcal{M} \oplus (v_1 v_2 v_3 \frown \gamma)$. This, together with the facts that $\overline{\mathcal{M}} \oplus (v_3 \frown \gamma)$ is a matching with more edges than $\overline{\mathcal{M}}$, finish our proof. □

Formalisation

The formalised statement of Berge's lemma is shown in Listing 11.

```
theorem Berge:
  assumes matching: "finite M" "matching M" "M ⊆ G"
  shows
   "(∃p. matching_augmenting_path M p ∧ path G p ∧ distinct p)
         =  (∃M' ⊆ G. matching M' ∧ card M < card M')"
```

Listing 11 Statement of Berge's lemma in our formalisation.

Remark 2. Our proof of Berge's lemma is similar to the exposition in Bondy and Morty's textbook [13, Chapter 16], which is a standard textbook on graph theory. However, there is a significant difference when it comes to the proof Lemma 2, where in the formal proof we have the extra task of showing that the case analysis performed within that lemma is exhaustive, while all informal treatments this is not considered. Informally, it is obvious that if a set of edges cannot be arranged into a path, it is either a cycle or there is a vertex on which two edges are incident. In the formalisation, on the other hand, that took a majority of the effort of proving Lemma 2. This is an example of a theme that is recurring in formalising reasoning that uses appeals to graphical or geometric intuition, and was documented by a number of authors, including us [10, 11, 14, 15].

The functional correctness of Algorithm 1 is stated in the following corollary.

Corollary 1. *Assume that* AUG_PATH_SEARCH$(\mathcal{G}, \mathcal{M})$ *satisfies Specification 1. Then, for any graph* $\mathcal{G}$*,* FIND_MAX_MATCHING$(\mathcal{G}, \emptyset)$ *is a maximum matching w.r.t.* $\mathcal{G}$*.*

Proof sketch. The statement follows from Theorem 1 and the fact that AUG_PATH_SEARCH$(\mathcal{G}, \mathcal{M})$ satisfies Specification 1. □

```
lemma find_max_matching_dom:
  assumes "matching M"" M ⊆ G"" finite M"
  shows "find_max_matching_dom M"
```

Listing 12 The termination theorem of the top loop of the algorithm. An implicit assumption here is that `aug_path_search` conforms to the assumptions in the locale header.

```
lemma find_max_matching_works:
  shows "(find_max_matching {}) ⊆ G"
    "matching (find_max_matching {})"
    "∀M. matching M ∧ M ⊆ G ⟹
           card M ≤ card (find_max_matching {})"
```

Listing 13 The functional correctness theorem of the top loop of the algorithm. Similarly to the termination theorem, we assume `aug_path_search` conforms to the assumptions in the locale header.

The formalised functional correctness theorem is shown in Listing 13. The theorem has three conclusions: the algorithm returns a subset of the graph, that subset is a matching, and the cardinality of any other matching is bounded by the size of the returned matching. Note that since it is proved within the locale `find_max_match`, it has an implicit assumption that `find_aug_path` satisfies the specification `find_aug_path_spec`. Also note that the algorithm is initialised with the empty matching.

The formal proof of Corollary 1 is done by computation induction using the induction principle that results from the termination proof of that recursive function. We use this methodology that is based on Isabelle/HOL's function package [16] for modelling and reasoning about all the major algorithms that we consider in this paper: we model them as recursive functions and prove facts about them using computation induction. For such recursive functions, the induction principle as well as the defining equations are conditional on the input being one on which the function is well-defined (e.g. inputs for which the predicate `find_max_matching_dom` in Listing 12 applies). The termination proof of the algorithm is based on showing that $|\mathcal{G} \setminus \mathcal{M}|$ decreases with every recursive call. The termination theorem is shown in Listing 12, where it is shown that the algorithm terminates, if it starts with a finite matching.

4 Handling Odd Cycles

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

1: **if** BLOSSOM_SEARCH($\mathcal{G}, \mathcal{M}$) is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$ **then**
2:     **return** BLOSSOM_SEARCH($\mathcal{G}, \mathcal{M}$)
3: **else if** BLOSSOM_SEARCH($\mathcal{G}, \mathcal{M}$) is a blossom $\langle \gamma, C \rangle$ w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$ **then**
4:     **return** refine(AUG_PATH_SEARCH($\mathcal{G}/P_C, \mathcal{M}/P_C$))
5: **else**
6:     **return** no augmenting path found
7: **end if**

In this step we refine AUG_PATH_SEARCH, which is the function that computes augmenting paths, into a more detailed description. In our exposition, AUG_PATH_SEARCH, refined as Algorithm 2, is a function that handles odd cycles found in the graph by removing them, which is the main insight underlying Edmonds' blossom shrinking algorithm. It is again parametrically defined, where it depends on the function BLOSSOM_SEARCH. AUG_PATH_SEARCH either (i) returns an augmenting path if BLOSSOM_SEARCH finds one (Line 2), (ii) removes (more on that later) an odd cycle from the graph, by contracting it (Line 4) and then recursively continues searching for augmenting paths, or (iii) reports that no augmenting paths exists, if BLOSSOM_SEARCH finds no odd cycles or augmenting paths (Line 6).

An important element here is how odd cycles are manipulated. Odd cycles found by BLOSSOM_SEARCH are returned in the form of *blossoms*, which is a central concept in Edmonds' algorithm. A blossom is a alternating path starting with an unmatched vertex that is constituted of two parts: (i) the *stem*, which is a simple path and, (ii) an odd cycle, which is a path with an odd number of edges, starting and ending at the same vertex. For instance, vertices v_1, v_2, v_3, v_4, v_5, and v_3 in Figure 2 constitute a blossom.

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

definition match_blossom where
  "match_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])))"

abbreviation "blossom G M stem C ≡
    path G (stem @ C) ∧ match_blossom M stem C"
```

Listing 14 The definition of a blossom. Note: `edges_of_path` is a function which, given a path, returns the list of edges constituting the path

Further Notation. A pair $\langle v_1v_2 \dots v_{i-1}, v_iv_{i+1} \dots v_n\rangle$ is a *blossom* w.r.t. a matching $\mathcal{M}$ iff 1. $v_iv_{i+1} \dots v_n$ is an odd cycle, 2. $v_1v_2 \dots v_n$ is an alternating path w.r.t. $\mathcal{M}$, and 3. $v_1 \notin \mathcal{V}(\mathcal{M})$. We refer to $v_1v_2 \dots v_i$ as the *stem* of the blossom and v_i as the *base* of the blossom. In many situations we have a pair $\langle v_1v_2 \dots v_{i-1}, v_iv_{i+1} \dots v_n\rangle$ which is a blossom w.r.t. a matching $\mathcal{M}$ where $v_1v_2 \dots v_{i-1}v_iv_{i+1} \dots v_{n-1}$ is also a simple path w.r.t. a graph $\mathcal{G}$ and $\{v_{n-1}, v_n\} \in \mathcal{G}$. In this case we call it a blossom w.r.t. $\langle \mathcal{G}, \mathcal{M}\rangle$. The formalisation of the notion of a blossom is shown in Listing 14. Furthermore, for a function f and a set s, let $f(\!|s|\!)$ denote the image of f on s. For a graph $\mathcal{G}$, and a function f, the *quotient* $\mathcal{G}/f$ is the set $\{f(\!|e|\!) \mid e \in \mathcal{G}\}$. The formalisation of this notion of quotient is shown in Listing 15.

```
definition quot_graph where
  "quot_graph P G = {e'. ∃e∈G. e' = P ` e}"

abbreviation "quotG G ≡ (quot_graph P G) - {{u}}"
```

Listing 15 The formalisation of a quotient of a graph. Note: `f ` s` is the image of function `f` on a `s`.

4.1 Blossom Contraction: A Detailed Proof

We prove that contracting (i.e. shrinking) the odd cycle of a blossom preserves the existence of an augmenting path, which is the second core result needed to prove the validity of the blossom-shrinking algorithm, after Berge's lemma. This result is the core idea behind this algorithm and it is why the algorithm can compute maximum cardinality matchings in polynomial time.

Theorem 2. *Consider a graph $\mathcal{G}$, a matching $\mathcal{M}$ w.r.t. $\mathcal{G}$, a blossom $\langle stem, C\rangle$ w.r.t. $\langle \mathcal{G}, \mathcal{M}\rangle$, and a vertex $u \notin \mathcal{V}(\mathcal{G})$. Let the function P be defined as $P(v) \equiv \mathsf{if}\ v \in s\ \mathsf{then}\ v\ \mathsf{else}\ u$, where $s \subset \mathcal{V}(\mathcal{G})$. Then we have an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M}\rangle$ iff there is an augmenting path w.r.t. $\langle \mathcal{G}/P, \mathcal{M}/P\rangle$.*

Proof sketch. We prove the directions of the bi-implication separately.

($\Rightarrow$) Let γ be the augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M}\rangle$. We prove this direction by considering two main cases.

Case 1 (stem $= \emptyset$ (i.e. $\mathsf{hd}(C) \notin \mathcal{M}$)). We have three further cases, which are clearly exhaustive.

Case 1.i ($|\gamma \cap C| = 0$). This case is trivial, since γ would also be an augmenting path w.r.t. $\langle \mathcal{G}/P, \mathcal{M}/P\rangle$.

Case 1.ii ($|\gamma \cap C| = 1$). In this case, we have three further cases, each representing a possible position of the odd cycle w.r.t. the augmenting path.

Case 1.ii.a ($\mathsf{hd}(C) \in \gamma \setminus \{\mathsf{hd}(\gamma), \mathsf{last}(\gamma)\}$). In this case, we have a contradiction because the base of the blossom will be in the matching, contradicting the assumption of Case 1.

Case 1.ii.b ($\mathsf{hd}(C) = \mathsf{last}(\gamma)$). Let γ' be γ without $\mathsf{last}(\gamma)$, i.e. $\gamma = \gamma' \frown \mathsf{last}(\gamma)$. In this case, the cycle is contracted to u. Also $u \notin \mathcal{V}(\mathcal{M}/P)$. Thus $\gamma' \frown u$ is an augmenting path w.r.t. $\langle \mathcal{G}/P, \mathcal{M}/P \rangle$.

Case 1.ii.c ($\mathsf{hd}(C) = \mathsf{hd}(\gamma)$). This case is symmetric with Case 1.ii.b.

Case 1.ii.d ($\mathsf{hd}(\gamma) \in (C \setminus \{\mathsf{hd}(C)\})$). This leads to a contradiction because $\mathsf{hd}(\gamma) \notin \mathcal{M}$, from the assumption that γ is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$, and every vertex in $C \setminus \mathsf{hd}(C)$ is in $\mathcal{V}(\mathcal{M})$.

Case 1.ii.e ($\mathsf{last}(\gamma) \in (C \setminus \{\mathsf{hd}(C)\})$). This case is symmetric to Case 1.ii.d.

Case 1.ii.f ($(\gamma \setminus \{\mathsf{hd}(\gamma), \mathsf{last}(\gamma)\}) \cap (C \setminus \mathsf{hd}(C) \neq \emptyset)$). Note that, from the definition of a blossom, every vertex in the cycle, except for the base, is in $\mathcal{M}$. From the assumption of Case 1.ii, we have that $2 \leq d(\mathcal{M}, v)$, for some $v \in C \setminus \{\mathsf{hd}(C)\}$. This is a contradiction because $\mathcal{M}$ is a matching.

Case 1.iii ($1 < |\gamma \cap C|$). From the case assumption, there must be a vertex v s.t. $v \in (C \setminus \{\mathsf{hd}(C)\})$ and $v \in \gamma$. We will then have one of the Cases 1.ii.d, 1.ii.e, or 1.ii.f, which all lead to a contradiction.

Case 2 ($\mathrm{stem} \neq \emptyset$). From the fact that γ is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$, we have that $\mathcal{M} \oplus \gamma$ is a matching and $|\mathcal{M}| < |\mathcal{M} \oplus \gamma|$. We also have that $\mathsf{hd}(\mathrm{stem}) \notin \mathcal{M}$ and that $\mathsf{last}(\mathrm{stem}) \in \mathcal{M}$, from the definition of blossom. Thus $\mathcal{M} \oplus \mathrm{stem}$ is a matching too, but $|\mathcal{M} \oplus \mathrm{stem}| = |\mathcal{M}|$, and thus, from Berge's lemma, there must be γ' that is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \oplus \mathrm{stem} \rangle$. Accordingly, we can apply Case 1 to γ' and to the matching $\mathcal{M} \oplus \mathrm{stem}$.

($\Leftarrow$) Let γ be the augmenting path w.r.t. $\langle \mathcal{G}/P, \mathcal{M}/P \rangle$. We have two cases.

Case 1 ($u \notin \gamma$). In this case we have that γ is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$, which finishes our proof.

Case 2 ($u \in \gamma$). From the assumption of Case 2, there are paths γ_1 and γ_2, and a vertex u, s.t. $\gamma = \gamma_1 \frown u \frown \gamma_2$. Since γ is an augmenting path w.r.t. $\langle \mathcal{G}/P, \mathcal{M}/P \rangle$, then exactly one of two edges incident to u belongs to $\mathcal{M}/P$, which gives rise to the following two cases.

Case 2.i ($\{\mathsf{last}(\gamma_1), u\} \in \mathcal{M}/P$). From the case assumption (namely, $\{\mathsf{last}(\gamma_1), u\} \in \mathcal{M}/P$) and from the definition of the quotient operation on graphs, we know that there is some vertex $v_1 \in C$ s.t. $\{\mathsf{last}(\gamma_1), v_1\} \in \mathcal{M}$. Since $\mathsf{last}(\gamma_1) \notin C$ and $\{\mathsf{last}(\gamma_1), v_1\} \in \mathcal{M}$, then $v_1 = \mathsf{hd}(C)$. We also know that since $\{u, \mathsf{hd}(\gamma_2)\} \in \mathcal{G}/P$, there must be a vertex v_2 s.t. $v_2 \in C$ and $\{v_2, \mathsf{hd}(\gamma_2)\} \in \mathcal{G}$. This means there are C_1 and C_2 s.t. $C = C_1 \frown v_2 \frown C_2$. We have two cases.

Case 2.i.a ($\{\mathsf{last}(C_1), v_2\} \in \mathcal{M}$). In this case, we have that $\gamma_1 \frown \mathrm{rev}(C_2) \frown \gamma_2$ is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$, which finishes our proof.

Case 2.i.b ($\{v_2, \mathsf{hd}(C_2)\} \in \mathcal{M}$). In this case, we have that $\gamma_1 \frown C_1 \frown \gamma_2$ is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$, which finishes our proof.

Case 2.ii ($\{\mathsf{hd}(\gamma_2), \mathsf{hd}(C)\} \in \mathcal{M}/P$). This is symmetric with case Case 2.i.

□

Formalisation

To formalise Theorem 2, we first declare the locale `quot` shown in Listing 16. This locale makes clear the assumptions, under which odd cycles can be contracted: the odd cycle has to be non-empty, and the odd cycle is contracted to a vertex that does not occur in the graph outside of the odd cycle. This allows the odd cycle to be contracted to a representative vertex from the odd cycle, which would allow for one of the more efficient implementations of the algorithm.

```
locale pre_quot = choose sel + graph_abs E
  for sel::"'a set ⇒ 'a" and E::"'a set set"

locale quot = pre_quot sel E for sel E +
  fixes s::"'a set" and u::'a
  assumes good_quot_map: "u ∉ s" "s ⊂ Vs E"

abbreviation "P v ≡ (if v ∈ s then v else u)"
```

Listing 16 A locale fixing assumptions on the representative vertex `u` for contraction.

Now, recall the function refine that refines a quotient augmenting path to a concrete one. Its formalisation is shown in Listing 17. 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`. `stem2vert_path` with last argument `hd p2` uses `choose_con_vert` to find a neighbour of `hd p2` on the cycle C. It splits the cycle at this neighbour and then returns the path leading to the base of the blossom starting with a matching edge. Finally, `replace_cycle` concatenates together p_1, p_2 and either `stem2p2` and `p12stem` to obtain an augmenting path in the concrete graph. This function has many possible execution paths, each equivalent to the case analysis in backward direction of the proof of Theorem 2.

```
fun find_pfx::"('b ⇒ bool) ⇒ 'b list ⇒ 'b list" where
 "find_pfx Q [] = []" |
 "find_pfx Q (h # l) =
   (if (Q h) then [h] else h # (find_pfx Q l))"

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

definition replace_cycle where
  "replace_cycle C M p1 p2 ≡
   let stem2p2 = stem2vert_path C M (hd p2);
       p12stem = stem2vert_path C 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)))"

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

Listing 17 The formalisation of refine.

The formal statement of Theorem 2 is shown in Listing 18. Similar to what we mentioned in Remark 1, a lot of the effort to formally prove this theorem is dominated by managing symmetries. One way to handle these symmetries is to devise lemmas capturing them, like lemma path_u_p3 shown in Listing 19. That lemma covers a major part of proving the four cases arising in Cases 2.i and 2.ii in the proof of Theorem 2. However, despite devising many of these lemmas, the formal proof of this theorem was around than 5K lines of proof script. This is because, although lemmas like that save a lot of repeated reasoning, it can still be cumbersome to prove that their preconditions hold for all cases (e.g. when `C` is assigned with C_1, C_2, $\text{rev}(C_1)$, and $\text{rev}(C_2)$ from Cases 2.i and 2.ii).

```
theorem refine:
  assumes cycle: "odd_cycle C" "alt_path M C"
                 "distinct (tl C)" "path E C" and
    quot_aug_path: "graph_augmenting_path (quotG E) (quotG M)
       p'" and
    matching: "matching M" "M ⊆ E" and
    quot: "s = (Vs E) - set C"
  shows "graph_augmenting_path E M (refine C M p')"

theorem aug_path_works_in_contraction:
  assumes match_blossom: "blossom E M stem C" and
    aug_path: "graph_augmenting_path E M p" and
    matching: "matching M" "M ⊆ E" "finite M" and
    quot: "s = (Vs E) - set C" "u ∉  Vs E"
  shows "∃p'. graph_augmenting_path (quotG E) (quotG M) p'"
```

Listing 18 The formal statements of the two directions of Theorem 2.

```
have path_u_p3: "path (quotG E) (u # p3)"
  if wx: "p = p1 @ x # p3" "x ∈ set C"
         "∀x∈set p3. x ∉ set C" and
    aug_path: "path E p" and
    p3_subset_s: "set p3 ⊆ s" and
    p3_nempty: "p3 ≠ []"
  for p1 x p3 p
```

Listing 19 A lemma devised to prove symmetric cases in the proof of `aug_path_works_in_contraction`.

4.2 The Algorithm

Finally, we formalise Algorithm 2 parametrically as shown in Listing 21. We model the algorithm in Isabelle/HOL using a locale which parameterises the algorithm over the function `blos_search` which performs the search for blossoms or augmenting paths. The function either returns an augmenting path or a blossom, which is formalised as the algebraic data type `match_blossom_res`. In addition to being parameterised over the function `blos_search`, it is also parameterised over the two functions `create_vert` and `sel`, shown in Listing 7. Note that we instantiate both arguments `P` and `s` of the locale `quot` to obtain the quotienting function `quotG` and the function for refining augmenting paths `refine`.

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

locale find_aug_path = choose + create_vert +
  fixes blos_search::"'a set set ⇒ 'a set set ⇒  ('a
      match_blossom_res) option"
  assumes
    bloss_algo_complete:
    "⟦graph_invar G; matching M; M ⊆ G;
        (∃p. graph_augmenting_path G M p)⟧
     ⟹ (∃blos_comp. blos_search G M = Some blos_comp)" and
    bloss_algo_sound:
    "⟦graph_invar G; matching M; M ⊆ G; blos_search G M = Some
        (Path p)⟧ ⟹
     graph_augmenting_path G M p"
    "⟦graph_invar G; matching M; M ⊆ G; blos_search G M = Some
        (Blossom stem C)⟧ ⟹
     blossom G M stem C"
```

Listing 20 The locale parameterising the formalisation of the algorithm `find_aug_path`. Note: `@` denotes list concatenation.

```
function find_aug_path where
  "find_aug_path G M =
     (case blos_search G M of Some match_blossom_res ⇒
        case match_blossom_res of Path p ⇒ Some p
        | Blossom stem cyc ⇒
            let u = create_vert (Vs G);
                s = Vs G - (set cyc);
                quotG = quot.quotG s (create_vert (Vs G));
                refine = quot.refine sel G s (create_vert (Vs
                    G)) cyc M
            in (case find_aug_path (quotG G) (quotG M) of Some
                p' ⇒ Some (refine p')
                | _ ⇒ None)
      | _ ⇒ None)"
```

Listing 21 Formalisation of Algorithm 2.

Correctness

To prove that Algorithm 2 is correct, we first precisely specify the properties expected of Blossom_Search, on which Algorithm 2 is parameterised.

Specification 2. *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* Blossom_Search$(\mathcal{G},\mathcal{M})$ *is a blossom or an augmenting path w.r.t. $\langle\mathcal{G},\mathcal{M}\rangle$.*

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

if $\exists e \in \mathcal{G}. e \cap \mathcal{V}(\mathcal{M}) = \emptyset$ **then**
  return Augmenting path choose $\{e \mid e \in \mathcal{G} \wedge e \cap \mathcal{V}(\mathcal{M}) = \emptyset\}$
else if compute_alt_path($\mathcal{G}, \mathcal{M}$) $= \langle \gamma_1, \gamma_2 \rangle$ **then**
  if last $\gamma_1 \neq$ last γ_2 **then**
    return Augmenting path (rev γ_1)$\frown\gamma_2$
  else
    $\langle \gamma_1', \gamma_2' \rangle$ = longest_disj_pref(γ_1, γ_2)
    return Blossom $\langle$rev(drop $(|\gamma_1'| - 1)$ γ_1), (rev γ_1')$\frown\gamma_2'\rangle$
  end if
else
  return No blossom or augmenting path found
end if

The functional correctness Algorithm 2 is stated as follows.

Corollary 2. *Assume* BLOSSOM_SEARCH($\mathcal{G}, \mathcal{M}$) *satisfies Specification 2. Then* AUG_PATH_SEARCH *satisfies Specification 1.*

Proof sketch. From Theorem 2 and by computation induction. □

Remark 3. 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. $\langle \mathcal{G}/P, \mathcal{M}/P \rangle$, i.e. w.r.t. the quotients), we follow a standard textbook approach (e.g. Lemma 10.25 in Korte and Vygen's book [4]). Our proof is, nonetheless, the only one we are aware of that explicitly pins down the cases, at least among standard textbooks [3–5] and lecture notes available online. One particular approach that is worth mentioning is that taken in LEDA [3] by Mehlhorn and Näher. In their approach, they skip showing this direction completely, due to the complexity of the case analysis and the fact that it was not fully performed in other expositions. Instead, they replaced it with a claim, presumed to be much easier to prove, that, if we can construct an odd set cover for $\mathcal{G}/P$, i.e. a certificate can be constructed showing that there is not an augmenting path w.r.t. $\langle \mathcal{G}/P, \mathcal{M}/P \rangle$, then there is a certificate showing that there is not an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$. Nonetheless, it turned out, when we tried to formalise that approach that we need a case analysis that is more complex than the one we perform in the proof of Theorem 2.

Remark 4. 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. We took the choice of explicitly defining that function with using it in an implementation of the 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 [3].

5 Computing Blossoms and Augmenting Paths

Here we take one further step in our refinement of the algorithm's description, where we give a more detailed description of the function BLOSSOM_SEARCH (see

Algorithm 3), which can compute augmenting paths or blossoms, if any exist in the graph. The algorithm takes as input two alternating paths and returns either an augmenting path or a blossom. The two given alternating paths have to satisfy a number of conditions, and those conditions describe any such alternating paths that result from the core search procedure of the algorithm. The algorithm is parameterised over two functions: compute_alt_path and longest_disj_pref. The former is the core search procedure of Edmonds' blossom algorithm and the latter is a function that takes the output of the former and uses it to return the stem of a blossom, if the two alternating path returned by compute_alt_path represent a blossom.

In Isabelle/HOL, the definition of Blossom_Search is shown in Listing 22. It depends on the function `longest_disj_pfx`, whose definition as well as its correctness statement are in Listing 23. It also depends on the function compute_alt_path. We use a locale again to formalise Algorithm 3. That locale parameterises Blossom_Search on the function compute_alt_path that searches for alternating paths and poses the soundness and completeness assumptions for that alternating path search function. The locale assumptions assert that compute_alt_path conforms to Specification 3.

```
definition compute_match_blossom where
  "compute_match_blossom ≡
   (if (∃e. e ∈ unmatched_edges) then
         let singleton_path = sel_unmatched in
           Some (Path singleton_path)
    else
     case compute_alt_path
       of Some (p1,p2) ⇒
         (if (set p1 ∩ set p2 = {}) then
            Some (Path ((rev p1) @ p2))
          else
            (let (pfx1, pfx2) = longest_disj_pfx p1 p2 in
              (Some (Blossom
                       (rev (drop (length (the pfx1)) p1))
                       (rev (the pfx1) @ (the pfx2))))))
       | _ ⇒ None)"
```

Listing 22 Formalisation of Algorithm 3.

```
fun longest_disj_pfx where
  "longest_disj_pfx l1 [] = (None,None)"
| "longest_disj_pfx [] l2 = (None,None)"
| "longest_disj_pfx l1 (h#l2) =
    (let l1_pfx = (find_pfx ((=) h) l1) in
       if (last l1_pfx = h) then
         (Some l1_pfx,Some [h])
       else (let
              (l1_pfx,l2_pfx) = (longest_disj_pfx l1 l2)
             in
               case l2_pfx of Some pfx2 ⇒
                                (l1_pfx,Some (h#pfx2))
                              | _ ⇒ (l1_pfx, l2_pfx)))"

lemma common_pfxs_form_match_blossom':
  assumes
    pfxs_are_pfxs:
      "(Some pfx1, Some pfx2) = longest_disj_pfx p1 p2" and
    from_tree: "p1 = pfx1 @ p" "p2 = pfx2 @ p" and
    alt_paths:
      "alt_path M (hd p2 # p1)" "alt_path M (hd p1 # p2)"
      "last p1 ∉ Vs M" and
    hds_neq: "hd p1 ≠ hd p2" and
    odd_lens: "odd (length p1)" "odd (length p2)" and
    distinct: "distinct p1" "distinct p2" and
    matching: "matching M"
  shows
    "match_blossom M
                    (rev (drop (length pfx1) p1))
                    (rev pfx1 @ pfx2)"
```

Listing 23 An algorithm to find the longest disjoint prefix of two lists and its correctness statement.

Further Notation. We first introduce some notions and notation. For a list xs, let |xs| be the length of xs. For a list xs and a natural number n, let drop n xs denote the list xs, but with the first n elements dropped. For a list xs, let x :: xs denote adding an element x to the front of a list xs. For a non-empty list xs, let hd xs and last xs denote the first and last elements of xs, respectively. Also, for a list xs, let rev xs denote its reverse. For two lists xs_1 and xs_2, let $\mathrm{xs}_1 \frown \mathrm{xs}_2$ denote their concatenation. Also, let longest_disj_pref xs_1 xs_2 denote the pair of lists $\langle \mathrm{xs}_1', \mathrm{xs}_2' \rangle$, s.t. $\mathrm{xs}_1 = \mathrm{xs}_1' \frown \mathrm{xs}$ and $\mathrm{xs}_2 = \mathrm{xs}_2' \frown \mathrm{xs}$, and if both xs_1' and xs_2' are disjoint except at their endpoints. Listing 23 shows an implementation of the function. Note: this function is not always well-defined, but it is always well-defined if both lists are paths in a tree starting at the root, which is always the case for its inputs in our context here.

Correctness

The hard part of reasoning about the correctness of BLOSSOM_SEARCH is the specification of the properties of the functions on which it is parameterised. For two paths γ_1 and γ_2, a graph $\mathcal{G}$, and a matching $\mathcal{M}$ consider the following properties:

P1 γ_1 and γ_2 are simple paths w.r.t. $\mathcal{G}$.
P2 γ_1 and γ_2 alternating paths w.r.t. $\mathcal{M}$.
P3 γ_1 and γ_2 are of odd length.
P4 $\mathsf{last}\ \gamma_1 \notin \mathcal{V}(\mathcal{M})$.
P5 $\mathsf{last}\ \gamma_2 \notin \mathcal{V}(\mathcal{M})$.
P6 $\{\mathsf{hd}\ \gamma_1, \mathsf{hd}\ \gamma_2\} \in \mathcal{G}$.
P7 $\{\mathsf{hd}\ \gamma_1, \mathsf{hd}\ \gamma_2\} \notin \mathcal{M}$.

These properties are formalised in Listing 24.

Specification 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 satisfy properties P1-P7 iff two lists of vertices satisfying those properties exist.*

For the second function, longest_disj_pref we directly define it and prove it correct rather than devising a specification, mainly due its simplicity, as shown in Listing 23.

Lemma 3. *Assume* γ_1 *and* γ_2 *satisfy properties P1-P7 and are both are disjoint, and we have that* $\mathsf{last}\ \gamma_1 \neq \mathsf{last}\ \gamma_2$. *Then* $(rev\ \gamma_1)\frown\gamma_2$ *is an augmenting path w.r.t.* $\langle \mathcal{G}, \mathcal{M} \rangle$.

Proof sketch. The lemma follows from the following two facts.

Lemma 4. $(rev\ \gamma_1)\frown\gamma_2$ *is an alternating path w.r.t.* $\mathcal{M}$.

Proof. From P4, we have that $\mathsf{last}\ E(\gamma_1) \notin \mathcal{M}$, and thus we have that $\mathsf{hd}\ E(\mathrm{rev}\ \gamma_1) \notin \mathcal{M}$. Also, from P2, P3, and P5, we have that $\mathsf{hd}\ E(\gamma_2) \in \mathcal{M}$. From that, in addition to P2 and P7, we finish the proof. Also, from P4, and P5, we have that the first and last vertices of $(\mathrm{rev}\ \gamma_1)\frown\gamma_2$ are unmatched. Accordingly, we have that $(\mathrm{rev}\ \gamma_1)\frown\gamma_2$ is an augmenting path w.r.t. $\mathcal{M}$. □

Lemma 5. $(rev\ \gamma_1)\frown\gamma_2$ *is a simple path w.r.t.* $\mathcal{G}$.

Proof. This follows from P1, P6, and since we assume that γ_1 and γ_2 are disjoint. □

□

Lemma 6. *If* γ_1 *and* γ_2 *are both 1. simple paths w.r.t.* $\mathcal{G}$, *2. alternating paths w.r.t.* $\mathcal{M}$, *and 3. of odd length, and if we have that 4.* $\mathsf{last}\ \gamma_1 = \mathsf{last}\ \gamma_2$, *5.* $\mathsf{last}\ \gamma_1 \notin \mathcal{V}(\mathcal{M})$, *6.* $\{\mathsf{hd}\ \gamma_1, \mathsf{hd}\ \gamma_2\} \in \mathcal{G}$, *7.* $\{\mathsf{hd}\ \gamma_1, \mathsf{hd}\ \gamma_2\} \notin \mathcal{M}$, *and 8.* $\langle \gamma_1', \gamma_2' \rangle = \mathsf{longest\_disj\_pref}(\gamma_1, \gamma_2)$, *then* $\langle rev(\mathsf{drop}\ (|\gamma_1'| - 1)\ \gamma_1), (rev\ \gamma_1')\frown\gamma_2' \rangle$*is a blossom w.r.t.* $\langle \mathcal{G}, \mathcal{M} \rangle$.

Proof sketch. The proof here is done using a similar construction to what we did in the proof of Lemma 3. □

Finally, the following Theorem shows BLOSSOM_SEARCH is correct.

Theorem 3. *Assume that* compute_alt_path *satisfies Specification 3. Then* BLOSSOM_SEARCH *satisfies Specification 2.*

Proof sketch. The theorem follows from Lemma 6 and Lemma 3 and the definitions of Specification 3 and BLOSSOM_SEARCH. □

```
definition compute_alt_path_spec where
  "compute_alt_path_spec G M compute_alt_path ≡
   (∀p1 p2 pref1 x post1 pref2 post2.
       compute_alt_path = Some (p1, p2) ⟹
         p1 = pref1 @ x # post1 ∧ p2 = pref2 @ x # post2
          ⟹ post1 = post2) ∧
   (∀p1 p2. compute_alt_path = Some (p1, p2) ⟹
         alt_path M (hd p1 # p2)) ∧
   (∀p1 p2. compute_alt_path = Some (p1, p2) ⟹
        alt_path M (hd p2 # p1)) ∧
   (∀p1 p2. compute_alt_path = Some (p1, p2) ⟹
        last p1 ∉ Vs M) ∧
   (∀p1 p2. compute_alt_path = Some (p1, p2) ⟹
         last p2 ∉ Vs M) ∧
   (∀p1 p2. compute_alt_path = Some (p1, p2) ⟹
         hd p1 ≠ hd p2) ∧
   (∀p1 p2. compute_alt_path = Some (p1, p2) ⟹
         odd (length p1)) ∧
   (∀p1 p2. compute_alt_path = Some (p1, p2) ⟹
         odd (length p2)) ∧
   (∀p1 p2. compute_alt_path = Some (p1, p2) ⟹
         distinct p1) ∧
   (∀p1 p2.
       compute_alt_path = Some (p1, p2) ⟹ distinct p2) ∧
   (∀p1 p2. compute_alt_path = Some (p1, p2) ⟹ path G p1) ∧
   (∀p1 p2. compute_alt_path =  Some(p1, p2) ⟹ path G p2) ∧
   (∀p1 p2.
      compute_alt_path = Some (p1, p2) ⟹ {hd p1, hd p2} ∈ G)"

locale match = graph_abs G for G +
  fixes M
  assumes matching: "matching M" "M ⊆ G"

locale compute_match_blossom' = match G M + choose sel
   for sel::"'a set ⇒ 'a" and G M ::"'a set set" +

fixes compute_alt_path:: "(('a list × 'a list) option)"
assumes
compute_alt_path_spec:
"compute_alt_path_spec G M compute_alt_path" and
compute_alt_path_complete:
"(((∃p. path G p ∧ distinct p ∧
        matching_augmenting_path M p)))
         ⟹ (∃blos_comp. compute_alt_path = Some blos_comp)"
```

Listing 24 The specification of the correctness of the core search procedure.

Remark 5. The formal proofs of the above lemmas are largely straightforward. The main difficulty is coming up with the precise properties, e.g. in Specification 3,

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

ex $:= \emptyset$ // Set of examined edges
for $v \in \mathcal{V}(\mathcal{G})$ **do**
  label $v :=$ None
  parent $v :=$ None
end for
for $v \in \mathcal{V}(\mathcal{G}) \setminus \mathcal{V}(\mathcal{M})$ **do**
  label $v := \langle u, \text{even} \rangle$
end for
while $(\mathcal{G} \setminus \text{ex}) \cap \{e \mid \exists v \in e, r \in \mathcal{V}(\mathcal{G}).\text{label } v = \langle r, \text{even} \rangle\} \neq \emptyset$ **do**
  // Choose a new edge and label it examined
  $\{v_1, v_2\} :=$ choose $(\mathcal{G} \setminus \text{ex}) \cap \{\{v_1, v_2\} \mid \exists r.\text{label } v_1 = \langle r, \text{even} \rangle\}$
  ex $:=$ ex $\cup \{\{v_1, v_2\}\}$
  if label $v_2 =$ None **then**
    // Grow the discovered set of edges from r by two
    $v_3 :=$ choose $\{v_3 \mid \{v_2, v_3\} \in \mathcal{M}\}$
    ex $:=$ ex $\cup \{\{v_2, v_3\}\}$
    label $v_2 := \langle r, \text{odd} \rangle$
    label $v_3 := \langle r, \text{even} \rangle$
    parent $v_2 := v_1$
    parent $v_3 := v_2$
  else if $\exists s \in \mathcal{V}(\mathcal{G}).\text{label } v_2 = \langle s, \text{even} \rangle$ **then**
    // Return two paths from current edge's tips to unmatched vertex(es)
    return $\langle$follow parent v_1, follow parent $v_2\rangle$
  end if
end while
return No paths found

which required many iterations between the correctness proof of Algorithm 3 and Algorithm 4, which implements the assumed function compute_alt_path.

6 Searching for an Augmenting Path or a Blossom

Lastly, we refine the function compute_alt_path to a detailed algorithmic description (see Algorithm 4). This algorithm performs an *alternating tree search*. The search aims to either find an augmenting path or a blossom. It is done via growing alternating trees rooted at unmatched vertices. The search is initialised by making each unmatched vertex a root of an alternating tree; the matched nodes are in no tree initially. In an alternating tree, vertices at an even depth are entered by a matching edge, vertices at an odd depth are entered by a non-matching edge, and all leaves have even depth. In each step of the search, one considers a vertex v_1 of even depth that is incident to an edge $\{v_1, v_2\}$ that was not examined yet, s.t. there is $\{v_2, v_3\} \in \mathcal{M}$. If v_2 is not in a tree yet, then one adds v_2 (at an odd level) and v_3 (at an even level). If v_2 is already in a tree and has an odd level then one does nothing as one simply has discovered another odd length path to v_2. If v_2 is already in a tree and has an even level then one has either discovered an augmenting path (if v_1 and v_2 belong to trees with different roots) or a blossom (if v_1 and v_2 belong to the same tree). If the function positively terminates, i.e. finds two vertices with even labels, it returns two alternating paths by ascending the two alternating trees to which the two vertices belong, where both paths satisfy Properties P1-P7. This tree ascent is performed by the function follow . That function takes a higher-order argument and a vertex. The higher-order argument is a function that maps every vertex to another vertex, which is intended to be its parent in a tree structure.

To formalise Algorithm 4 in Isabelle/HOL, we first formally define the function follow , which follows a vertex's parent, as shown in Listing 25. Again, we use a locale to formalise that function, and that locale fixes the function `parent` mapping every vertex to its parent in its respective tree. Note that the function `follow` 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.

```
definition follow_invar'::"('a ⇒ 'a option) ⇒ bool" where
  "parent_spec parent ≡ wf {(x, y) |x y. (Some x = par y)}"

locale parent =
  fixes parent::"'a ⇒ 'a option" and
    parent_rel::"'a ⇒ 'a ⇒ bool"
  assumes parent_rel:
    "follow_invar' parent"

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

Listing 25 The definition of a function that ascends the search tree towards the root, returning the traversed path.

Based on that, compute_alt_path is formalised as shown in Listing 28. Note that we do not use a while combinator to represent the while loop: instead we formalise it as a recursive function. 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`.

```
definition if1 where
  "if1 flabel ex v1 v2 v3 r =
     ({v1, v2} ∈ G - ex ∧ flabel v1 = Some (r, Even) ∧
      flabel v2 = None ∧ {v2, v3} ∈ M)"

definition if1_cond where
  "if1_cond flabel ex =
     (∃v1 v2 v3 r. if1 flabel ex v1 v2 v3 r)"

definition if2 where
  "if2 flabel v1 v2 r r' =
     ({v1, v2} ∈ G ∧ flabel v1 = Some (r, Even) ∧
      flabel v2 = Some (r', Even))"

definition if2_cond where "if2_cond flabel =
   (∃v1 v2 r r'. if2 flabel v1 v2 r r')"

function compute_alt_path::
  "'a set set ⇒ ('a ⇒ 'a option) ⇒ ('a ⇒ ('a × label) option)
     ⇒ (('a list × 'a list) option)"
  where
  "compute_alt_path ex par flabel =
    (if if1_cond flabel ex then
       let
         (v1,v2,v3,r) = sel_if1 flabel ex;
         ex' = insert {v1, v2} ex;
         ex'' = insert {v2, v3} ex';
         par' = par(v2 := Some v1, v3 := Some v2);
         flabel' =
           flabel(v2 := Some (r, Odd), v3 := Some (r, Even));
         return = compute_alt_path ex'' par' flabel'
       in
         return
     else if if2_cond flabel then
        let
          (v1,v2,r,r') = sel_if2 flabel;
          return =
            Some (parent.follow par v1, parent.follow par v2)
        in
          return
     else
       let
          return = None
       in
         return)"
```

Listing 26 The definition of a function that constructs the search forest, which is the main search procedure of Edmonds' blossom shrinking algorithm. Note: `f(x := v)` denotes the point-wise update of a function in Isabelle/HOL.

Note that this function is also defined within a locale, shown in Listing 27. That locale assumes nothing but a choice function that picks elements from finite sets.

```
locale match = graph_abs G for G+
  fixes M
  assumes matching: "matching M" "M ⊆ G"

locale compute_alt_path = match G M + choose sel
  for G M::"'a set set" and sel::"'a set ⇒ 'a"
```

Listing 27 Functions on which compute_alt_path is parameterised.

One last aspect of our formalisation of compute_alt_path is how we model nondeterministic choice and selection. As mentioned earlier we aimed to arrive at a final algorithm with minimal assumptions on functions for nondeterministic computation. We thus implement all needed nondeterministic computation using the basic assumed nondeterminitic choice function. Listing 28 shows, as an example, how we nondeterministically choose the vertices in the first execution path of the while-loop (i.e. the path ending on Line 20).

```
definition
  "sel_if1 flabel ex =
     (let es =
        D ∩ {(v1,v2)| v1 v2. {v1,v2} ∈ (G - ex) ∧
                             (∃r. flabel v1 = Some (r, Even)) ∧
                             flabel v2 = None ∧ v2 ∈ Vs M};
         (v1,v2) = sel_pair es;
         v3 = sel (neighbourhood M v2);
         r = fst (the (flabel v1))
     in (v1,v2,v3,r))"

lemma sel_if1_works:
  assumes "if1_cond flabel ex"
          "(v1, v2, v3, r) = sel_if1 flabel ex"
  shows "if1 flabel ex v1 v2 v3 r"
```

Listing 28 The definition of a function that nondeterministically chooses vertices and a root that satisfy the conditions of the first execution branch of the while loop.

The functional correctness theorem of Algorithm 4, on the proof of which we spend the rest of this section, is stated as follows.

Theorem 4. compute_alt_path *satisfies Specification 3.*

6.1 Loop Invariants

Proving this theorem involves reasoning about a while-loop using loop invariants. Nonetheless, since the while-loop involves a large number of variables in the state, and those variables represent complex structures, e.g. parent, the loop invariants capturing the interactions between all those variables are extensive. During the development of the formal proof, we have identified the following loop-invariants to be sufficient to prove Theorem 4.

Invariant 1. *For any vertex v, if for some r,* label $v = \langle r, \mathsf{even}\rangle$*, then the vertices in the list* follow parent v *have labels that alternate between* $\langle r, \mathsf{even}\rangle$ *and* $\langle r, \mathsf{odd}\rangle$*.*

Invariant 2. *For any vertex v_1, if for some r and some l, we have* label $v_1 = \langle r, l\rangle$*, then the vertex list $v_1 v_2 \dots v_n$ returned by* follow parent v_1 *has the following property: if* label $v_i = \langle r, \mathsf{even}\rangle$ *and* label $v_{i+1} = \langle r, \mathsf{odd}\rangle$*, for some r, then* $\{v_i, v_{i+1}\} \in \mathcal{M}$*, otherwise,* $\{v_i, v_{i+1}\} \notin \mathcal{M}$*.*

Invariant 3. *The relation induced by the function* parent *is well-founded.*

Invariant 4. *For any* $\{v_1, v_2\} \in \mathcal{M}$*,* label $v_1 = $ *None iff* label $v_2 = $ *None.*

Invariant 5. *For any v_1, if* label $v_1 = $ *None then* parent $v_2 \neq v_1$*, for all v_2.*

Invariant 6. *For any v, if* label $v \neq$ *None, then* last (follow parent v) $\notin \mathcal{V}(\mathcal{M})$*.*

Invariant 7. *For any v, if* label $v \neq$ *None, then* label (last (follow parent v)) $= \langle r, \mathsf{even}\rangle$*, for some r.*

Invariant 8. *For any* $\{v_1, v_2\} \in \mathcal{M}$*, if* label $v_1 \neq$ *None, then* $\{v_1, v_2\} \in \mathsf{ex}$*.*

Invariant 9. *For any v,* follow parent v *is a simple path w.r.t. $\mathcal{G}$.*

Invariant 10. *For any* $\{v_1, v_2\} \in \mathcal{M}$*,* label $v_1 = \langle r, \mathsf{even}\rangle$ *iff* label $v_2 = \langle r, \mathsf{odd}\rangle$*.*

Invariant 11. *For all* $e \in \mathsf{ex}$*, there are* $v \in e$ *and r s.t.* label $v = \langle r, \mathsf{odd}\rangle$*.*

Invariant 12. *For all* $e \in \mathsf{ex}\setminus$*, if* $v \in e$*, then* label $v = $ *None.*

Invariant 13. *The set* $\{v \mid \exists r.\ \mathsf{label}\ v = \langle r, \mathsf{odd}\rangle\}$ *is finite.*

Invariant 14. $|\{v \mid \exists r.\ \mathsf{label}\ v = \langle r, \mathsf{odd}\rangle\}| = |\mathcal{M} \cap \mathsf{ex}|$.

Invariant 15. *For all* $v \in \mathcal{V}(\mathcal{G})$*, if* label $v = $ *None, then there is* $e \in \mathcal{G} \setminus \mathsf{ex}$ *s.t.* $v \in e$*.*

Invariant 16. *For all* $v \in \mathcal{V}(\mathcal{G})$*, if* label $v = \langle r, \mathsf{odd}\rangle$*, then there is* $e \in \mathcal{G} \cap \mathsf{ex}$ *s.t.* $v \in e$*.*

Proofs of those invariants require somewhat complex reasoning: they involve interactions between induction (e.g. well-founded induction on parent) and the evolution of the 'program state', i.e. the values of the variables as the while-loop progresses with its computation. We describe one of those formal proofs in some detail below to give the reader an idea of how we did those proofs.

Further Notation. In an algorithm, we refer to the value of a variable x 'after' executing line i with x^i.

Proof sketch of Invariant 1. The algorithm has only one execution branch where it continues iterating, namely, when the condition on Line 13 holds. We show that if Invariant 1 holds for label^{10} and parent^{10}, then it holds for label^{20} and parent^{20}. In particular, we need to show that if, for any vertex v, $\mathsf{label}^{20}\ v = \langle r, \mathsf{even}\rangle$, then the labels assigned by label^{20} to vertices of follow $\mathsf{parent}^{20}\ v$ alternate between $\langle r, \mathsf{even}\rangle$ and $\langle r, \mathsf{odd}\rangle$. The proof is by induction on follow $\mathsf{parent}^{20}\ v$. We have the following cases, two base cases and one step case.

Case 1 (follow $\mathsf{parent}^{20}\ v = \emptyset$). This case is trivial.

Case 2 ($\mathsf{follow}\ \mathsf{parent}^{20}\ v = u$, for some u). This case is also trivial since the $\mathsf{follow}\ \mathsf{parent}^{20}\ v$ has no edges.

Case 3 ($\mathsf{follow}\ \mathsf{parent}^{20}\ v = u_1 u_2 \frown \gamma$, for some u_1 and u_2). The proof can be performed by the following case analysis.

Case 3.i ($u_2 = {v_2}^{20}$). We further analyse the following two cases.

Case 3.i.a ($u_1 = {v_3}^{20}$). First, we have that $v = u_1$ from the definition of follow and from the assumption of Case 3. This together with the assumption of Case 3.i imply that $v = {v_3}^{20}$.

We also have that $\{{v_2}^{20}, {v_3}^{20}\} \in \mathcal{M}$, from Line 15, and the fact that neither v_2 and v_3 change between Lines 16-20. Also note that from Line 13, we have that $\mathsf{label}\ {v_2}^{20} = \text{None}$. This, together with Invariant 4, imply that $\mathsf{label}\ {v_3}^{20} = \text{None}$. Thus, from Invariant 5, we have that $\mathsf{parent}\ u \neq v_3$, for any u. Thus, $\{{v_3}^{20}, {v_2}^{20}\} \cap \mathsf{follow}\ \ \mathsf{parent}^{10} v_1 = \emptyset$.

From Line 9 we know that $\mathsf{label}\ {v_1}^{20} = \langle r, \mathsf{even}\rangle$, for some r. Thus, since Invariant 1 holds at Line 10, we know that the labels of $\mathsf{follow}\ \ \mathsf{parent}^{10}\ {v_1}^{20}$ alternate w.r.t. label^{10}. Since Lines 10-20 imply that $\mathsf{parent}^{10}\ v = \mathsf{parent}^{20}\ v$ and $\mathsf{label}^{10}\ v = \mathsf{label}^{20}\ v$, for all $v \notin \{{v_2}^{20}, {v_3}^{20}\}$, and since $\{{v_2}^{20}, {v_3}^{20}\} \cap \mathsf{follow}\ \ \mathsf{parent}^{10} v_1 = \emptyset$, then we have that $\mathsf{follow}\ \ \mathsf{parent}^{20}\ {v_1}^{20}$ alternate w.r.t. label^{20}. This, together with the assignments at Lines 17-20 imply that $\mathsf{follow}\ \ \mathsf{parent}^{20}\ {v_3}^{20}$ alternate w.r.t. label^{20}, which finishes our proof.

Case 3.i.b ($u_1 \neq {v_3}^{20}$). From the assumption of Case 3, we have that $\mathsf{parent}^{10}\ u_1 = {v_2}^{20}$, which is a contradiction from Invariant 5 and the condition in Line 13.

Case 3.ii ($u_1 = {v_2}^{20}$ and $u_2 = {v_1}^{20}$). This case is implied by Case 3.i since $\mathsf{follow}\ \ \mathsf{parent}^{20}\ {v_3}^{20} = {v_3}^{20} \frown \mathsf{follow}\ \ \mathsf{parent}^{20}\ {v_2}^{20}$.

Case 3.iii ($\{u_1, u_2\} \cap \{{v_2}^{20}\} = \emptyset$). We perform the following case analysis.

Case 3.iii.a (${v_3}^{20} \in \mathsf{follow}\ \mathsf{parent}^{20}\ v$). First, note that, from Line 20, we have that $\mathsf{parent}^{20}\ {v_3}^{20} = {v_2}^{20}$. Thus, if ${v_3}^{20} = u_1$, then we have that ${v_2}^{20} = u_2$, which is a contradiction from the assumption of Case 3.iii. Thus, we have that ${v_3}^{20} \in u_2 \frown \gamma$. Note that from Invariants 4 and 5, Lines 13 and 15, we have that $\mathsf{parent}^{10}\ v \neq {v_3}^{20}$, for all v. Thus, we have a contradiction.

Case 3.iii.b (${v_3}^{20} \notin \mathsf{follow}\ \mathsf{parent}^{20}\ v$). Note that from Invariant 5, and Lines 13 and 15, we have that, if there is $u \in u_1 u_2 \frown \gamma$ s.t. $\mathsf{parent}^{20}\ u$, then $u = {v_3}^{20}$, which is a contradiction. Thus, $v_2 \notin u_1 u_2 \frown \gamma$. Thus for any $u \in u_1 u_2 \frown \gamma$, we have that $\mathsf{parent}^{10}\ u = \mathsf{parent}^{20}\ u$ and $\mathsf{label}^{10}\ u = \mathsf{label}^{20}\ u$. This finishes our proof, since Invariant 1 holds for parent^{10} and label^{10}.

□

6.2 Total Correctness Proof

Below we describe in some detail our formal total correctness proof of the search algorithm. Although the algorithm has been treated by numerous authors, we believe that the following proof is more detailed that any previous exposition.

Lemma 7. *Algorithm 4 always terminates.*

Proof sketch. The termination of the algorithm is based on showing that $|\mathcal{G} \setminus \mathsf{ex}|$ decreases with every iteration of the while loop. ☐

Lemma 8. *If Algorithm 4 returns two paths then they satisfy properties P1-P7.*

Proof sketch. First, from the definition of the algorithm, we know that the algorithm returns (at Line 23) the two lists $\mathsf{follow\ parent}^{23}\ {v_1}^{23}$ and $\mathsf{follow\ parent}^{23}\ {v_2}^{23}$. Invariant 3 implies that $\mathsf{follow\ \ parent}^{10}$ is well-defined for any vertex in the Graph, and thus $\mathsf{follow\ parent}^{23}\ {v_1}^{23}$ and $\mathsf{follow\ parent}^{23}\ {v_2}^{23}$ are both well-defined. We now show that they satisfy the properties P1-P7.

- P1 follows from Invariant 9 and since $\mathsf{follow\ parent}^{10}$ is well-defined.
- P2 follows from Invariants 1 and 2 and since $\mathsf{follow\ parent}^{10}$ is well-defined.
- From Line 11, we have that $\mathsf{label}^{10}\ {v_1}^{23} = \mathsf{even}$. From Invariant 7 and since $\mathsf{follow\ \ parent}^{10}$ is well-defined, we have that $\mathsf{label}^{10}(\mathsf{last}(\mathsf{follow\ \ parent}^{10}\ \ {v_1}^{23})) = \mathsf{even}$. Since $\mathsf{label}^{10} = \mathsf{label}^{23}$ and $\mathsf{parent}^{10} = \mathsf{parent}^{23}$, then $\mathsf{label}^{23}(\mathsf{last}(\mathsf{follow\ parent}^{23}\ {v_1}^{23})) = \mathsf{even}$. Also, from Invariant 1, we have that the vertices in $\mathsf{label}^{23}(\mathsf{last}(\mathsf{follow\ parent}^{23}\ {v_1}^{23}))$ alternate between labels of even and odd. From the properties of alternating lists, we know that if vertices of a list alternate w.r.t. a predicate (in this case $\mathsf{even}/\mathsf{odd}$), and the first and last vertex satisfy the same predicate (here even), then the length of this list is odd. Thus, the length of γ_1 is odd. Similarly, we show that the length of γ_2 is odd. This gives us P3.
- P4 and P5 follow from Invariant 6.
- P6 follow from Line 11.
- From Invariant 1, we have that, since $\mathsf{label}^{23}\ {v_1}^{23} = \mathsf{even}$, then the label of the vertex occurring after ${v_1}^{23}$ in $\mathsf{follow\ parent}^{23}\ {v_1}^{23}$, call it u_1, is labelled as odd. From Invariant 2, we thus have $\{{v_1}^{23}, u_1\} \in \mathcal{M}$. Similarly, we have that $\{{v_2}^{23}, u_2\} \in \mathcal{M}$, where u_2 is the vertex occurring in $\mathsf{follow\ parent}^{23}\ {v_2}^{23}$ after ${v_2}^{23}$. We thus have that $\{{v_1}^{23}, {v_2}^{23}\} \notin \mathcal{M}$, since no two matching edges can be incident to the same vertex, meaning that we have P7.

☐

Lemma 9. *If there are two paths satisfying properties P1-P7, then Algorithm 4 returns two paths.*

The proof of this lemma depends on showing that we can construct a certificate showing that no such paths exist, if the algorithm returns at Line 26. The certificate is an odd set cover, defined as follows. For a set $s \subseteq \mathcal{V}(\mathcal{G})$, s.t. $|s| = 2k + 1$, for some k, we define the *capacity* of s as follows:

$$\mathrm{cap}(s) = \begin{cases} 1 & \text{if } k = 0 \\ k & \text{otherwise.} \end{cases}$$

For a set of edges E, we say s covers E iff $s \cap e \neq \emptyset$, for each $e \in E$, and $k = 0$. Otherwise, s covers E iff $\bigcup E \subseteq s$. A set of sets OSC is an *odd set cover* of a graph $\mathcal{G}$ iff for every $s \in \mathrm{OSC}$, we have that $|s|$ is odd and that for every $e \in \mathcal{G}$, there is $s \in \mathrm{OSC}$ s.t. s covers e. We have the following standard property of odd set covers.

Proposition 3. *If OSC is an odd set cover for a graph $\mathcal{G}$, then, if $\mathcal{M}$ is a matching w.r.t. $\mathcal{G}$, we have that $|\mathcal{M}| \leq |OSC|$.*

Lemma 10. *Given a graph $\mathcal{G}$ and a matching $\mathcal{M}$ w.r.t. $\mathcal{G}$, if there is an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$, then are two paths satisfying properties P1-P7.*

Proof sketch. Let the augmenting path be called γ. First, note that any augmenting path has at least three edges. Thus, there must be a u_1, u_2, u_3, and γ_2, s.t. $\gamma = u_1 u_2 u_3 \frown \gamma_2$. Two paths that are the required witness are $u_3 u_2 v_1$ and $\text{rev}(\gamma_2)$. ☐

Lemma 11. *If Algorithm 4 returns at Line 26, then there is an odd set cover OSC for $\mathcal{G}$ and $|OSC| = |\mathcal{M}|$.*

Proof sketch. Let $\text{OSC} \equiv \{\{v\} \mid \mathsf{label}^{26}\ v = \mathsf{odd}\}$. First, we have that OSC is an odd set cover for ex, from the definition of odd set covers and from Invariant 11. Second, since no two vertices in an edge can have the same label of odd or even, we have that for any $e \in \mathcal{G}$, there is $v \in e$ s.t. $\mathsf{label}^{26}\ v = \text{None}$ or $\mathsf{label}^{26}\ v = \mathsf{odd}$. The rest of the proof is via the following case analysis.

Case 1 ($|\mathcal{M}| = |\mathsf{ex}|$). From the case assumption and Invariant 15, we have that $\mathsf{label}^{26}\ v \neq \text{None}$ holds for any $v \in \mathcal{V}(\mathcal{G})$. Thus, every $e \in \mathcal{G}$ has $v \in e$ s.t. $\mathsf{label}^{26}\ v = \mathsf{odd}$. Accordingly, OSC is an odd set cover for $\mathcal{G}$. From the case assumption, in addition to Invariant 14, we have that $|\text{OSC}| = |\mathcal{M}|$. Also, since, for every $s \in \text{OSC}$, we have that $|s| = 1$, Proposition 3 implies that $\mathcal{M}$ is a maximum cardinality matching. Thus, Lemma 10 finishes our proof.

Case 2 ($|\mathcal{M} \setminus \mathsf{ex}| = 1$). From the case assumption, there is u_1 and u_2 s.t. $\{u_1, u_2\} \in \mathcal{M} \setminus \mathsf{ex}$. Since $\{u_1, u_2\} \notin \mathsf{ex}$ and from Invariant 12, we have that $\mathsf{label}^{26}\ u_1 = \mathsf{label}^{26}\ u_2 = \text{None}$. Let OSC' be $\{\{u_1\}\} \cup \text{OSC}$.

Note 1. *For any $e' \in \mathcal{G} \setminus \mathsf{ex}$, there exists $u \in e'$ s.t. $\{u\} \in OSC'$.*

Proof sketch. We perform the proof by the following case analysis.

Case 2.i ($e' \in \mathcal{M}$). In this case, we have that $e' = \{u_1, u_2\}$ from the assumption of Case 2. Our witness u is thus u_1.

Case 2.ii ($e' \notin \mathcal{M}$). There must be a $u' \in e'$ s.t. $\mathsf{label}^{26}\ u' = \text{None}$ or $\mathsf{label}^{26}\ u' = \mathsf{odd}$. We perform the following case analysis.

Case 2.ii.a ($\mathsf{label}^{26}\ u' = \mathsf{odd}$). In this case, we have that $\{u'\} \in \text{OSC} \subseteq \text{OSC}'$, by definitions of OSC and OSC', which finishes our proof.

Case 2.ii.b ($\mathsf{label}^{26}\ u' = \text{None}$). There must be u'' s.t. $e' = \{u', u''\}$. We finish the proof by the following case analysis.

Case 2.ii.b.I ($\mathsf{label}^{26}\ u'' = \text{None}$). In this case, from the for loop at Line 6 and from the assumptions of Case 2.ii.b, we have that $\{u', u''\} \in \mathcal{M}$, which contradicts the assumption of Case 2.ii.

Case 2.ii.b.II ($\mathsf{label}^{26}\ u'' = \mathsf{odd}$). In this case, u'' is the required witness.

Case 2.ii.b.III ($\mathsf{label}^{26}\ u'' = \mathsf{even}$). This case leads to a contradiction, since it violates the termination assumption of the while loop at Line 9.

☐

From the above note, and since OSC is an odd set cover for ex, we have that OSC$'$ is an odd set cover for $\mathcal{G}$. Now, we focus on the capacity of OSC$'$. We have the following

$$\begin{aligned}
\text{cap}(\text{OSC}') &= |\text{OSC}'| && \text{since for all } s \in \text{OSC}', |s| = 1 \\
&= |\text{OSC}| + 1 && \text{by definition} \\
&= |\mathcal{M} \cap \mathsf{ex}| + |\mathcal{M} \setminus \mathsf{ex}| && \text{from Invariant 14} \\
&= |(\mathcal{M} \cap \mathsf{ex}) \cup (\mathcal{M} \setminus \mathsf{ex})| \\
&= |\mathcal{M}|.
\end{aligned}$$

This finishes our proof.

Case 3 ($2 \leq |\mathcal{M} \setminus \mathsf{ex}|$). From the case assumption, there must be u_1 and u_2, and $\mathcal{M}'$, s.t. $\mathcal{M} \setminus \mathsf{ex} = \{\{u_1, u_2\}\} \cup \mathcal{M}'$. First, note that $\{u_1, u_2\} \cap \mathcal{M}' = \emptyset$, since $\mathcal{M}'$ is also a matching and since $\{u_1, u_2\} \notin \mathcal{M}'$. Let OSC$'$ denote $\{\{u_1\}, \{u_2\} \cup \mathcal{V}(\mathcal{M}')\} \cup \text{OSC}$. We first show the following.

Note 2. *For any $e' \in \mathcal{G} \setminus \mathsf{ex}$, there exists $s \in OSC'$ s.t. $e' \subseteq s$ or there exists $u \in e'$ s.t. $\{u\} \in OSC'$.*

Proof sketch. Our proof is by case analysis.

Case 3.i ($e' = \{u_1, u_2\}$). In this case, our proof follows since $\{u_1\} \in \text{OSC}'$.

Case 3.ii ($e' \in \mathcal{M}'$). In this case, our proof follows since $\{u_1\} \cup \mathcal{V}(\mathcal{M}') \in \text{OSC}'$ and $e' \in \mathcal{M}'$.

Case 3.iii ($e' \notin \mathcal{M}$). This is a similar case analysis to Case 2.ii.

□

Based on the above note, and since OSC is an odd set cover for ex, we have that OSC$'$ is an odd set cover for $\mathcal{G}$. Finally, we consider the capacity of OSC$'$.

$$\begin{aligned}
\text{cap}(\text{OSC}') &= \text{cap}(\text{OSC}) + \text{cap}(\{\{u_1\}, \{u_2\} \cup \mathcal{V}(\mathcal{M}')\}) && \text{since } \bigcup\{\{u_1\}, \{u_2\} \cup \mathcal{V}(\mathcal{M}')\} \\
& && \text{and } \bigcup \text{OSC are disjoint} \\
&= \text{cap}(\text{OSC}) + \text{cap}(\{\{u_1\}\}) + && \text{since } u_1 \notin \bigcup\{\{u_2\} \cup \mathcal{V}(\mathcal{M}')\} \\
&\qquad \text{cap}(\{\{u_2\} \cup \mathcal{V}(\mathcal{M}')\}) \\
&= |\text{OSC}| + 1 + |\mathcal{M}'| && \text{by definition of capacity} \\
&= |\mathcal{M} \cap \mathsf{ex}| + |\mathcal{M} \setminus \mathsf{ex}| && \text{from Invariant 14} \\
&= |(\mathcal{M} \cap \mathsf{ex}) \cup (\mathcal{M} \setminus \mathsf{ex})| \\
&= |\mathcal{M}|.
\end{aligned}$$

Case 4 ($\mathcal{M} \subseteq \mathsf{ex}$). In this case, we have that for any $e \in \mathcal{M}$, there is u s.t. $\{u\} \in \text{OSC}$. Also, for any $e \in \mathcal{G} \setminus \mathsf{ex}$, there is u s.t. $\{u\} \in \text{OSC}$, using a case analysis like Case 2.ii. Thus, OSC is an odd set cover for $\mathcal{G}$. Since $\mathcal{M} \subseteq \mathsf{ex}$ and from Invariant 14, we have that $|\mathcal{M}| = |\text{OSC}|$. Since for all $s \in \text{OSC}$, $|s| = 1$, we have that $\text{cap}(\text{OSC}) = |\text{OSC}| = |\mathcal{M}|$, which finishes our proof.

□

Proof sketch of Lemma 9. We show the contrapositive, i.e. we show that if the algorithm returns no paths (i.e. terminate at Line 26), then there is not a path satisfying properties P1-P7. If the algorithm terminates at Line 26, based on Lemma 11, we can construct an odd set cover OSC, s.t. cap(OSC) = $|\mathcal{M}|$. From Proposition 3, we have that $\mathcal{M}$ is a maximum cardinality matching. Thus there is not an augmenting path w.r.t. $\langle \mathcal{G}, \mathcal{M} \rangle$, which, in addition to Lemma 10, finishes our proof. □

Proof sketch of Theorem 4. The theorem follows from Lemmas 7, 8, and 9. □

Remark 6. We note that, although Edmonds' blossom shrinking algorithm has many expositions in the literature, none of them, as far we have seen, have a detailed proof of the odd set cover construction (Lemma 9). We believe that our work is the first that provides a complete set of invariants and a detailed proof of the odd set cover construction. The closest exposition to our work, and which we use as an initial reference for our work, is the LEDA book by Näher and Mehlhorn. There, as we mentioned earlier, the authors assumed that the right-left direction of Theorem 2 is not needed and, accordingly, they only have invariants equivalent to Invariants 1-8. We thus needed to devise Invariants 9-16 from scratch, which are the invariants showing that the algorithm can construct an odd set cover if the search fails to find two paths or a blossom. Similar to Näher and Mehlhorn's book, we did not find a detailed proof of this construction in other standard textbooks [4, 5].

Formalisation

```
fun alt_labels_invar where
  "alt_labels_invar flabel r [] = True"
| "alt_labels_invar flabel r [v] =
     (flabel v = Some (r, Even))"
| "alt_labels_invar flabel r (v1 # v2 # vs) =
     ((if (flabel v1 = Some (r, Even) ∧
            flabel v2 = Some (r, Odd)) then
         {v1,v2} ∈ M
       else if (flabel v1 = Some (r, Odd) ∧
                 flabel v2 = Some (r, Even)) then
         {v1, v2} ∉ M
       else undefined)
      ∧ alt_labels_invar flabel r (v2 #vs))"
```

Listing 29 A formalisation of Invariants 1 and 2.

The most immediate challenge to formalising the correctness proof of Algorithm 4 is to design the formalisation such that it is possible to effectively manage the proofs for the invariants we identified, which are numerous, and combine them into proving that the algorithm is totally correct. Formalising the invariants themselves is relatively straightforward, where we model invariants as predicates parameterised over the relevant variables in the algorithm. For instance, Listing 29 shows the formal predicate corresponding Invariants 1 and 2.

Proving that the invariants hold, on the other hand, is rather involved. We note that in much of the previous work the focus was on automatic verification condition generation from the invariants and the algorithm description. There one would state one invariant for the loop, involving all interactions between the state variables. Most verification conditions for that invariant are then automatically generated, and most of the generated conditions would be discharged automatically The ones which are not are either proved externally as lemmas or proved automatically by strengthening the invariant.

In our setting, the first difference is that many of the invariants need substantial abstract mathematical reasoning (see the proof of Invariant 1 for an example), which makes automatic discharging of verification conditions impossible in practice. Also, the standard approach would be infeasible here since the while-loop we consider is complex and the interactions between state variables are captured in 16 invariants. We thus structure the verification of invariants manually, where we try to prove every invariant independently, and assume, as we need, more of the other invariants that we previously identify, or create new invariants. Listing 30 shows the theorem we proved for `alt_labels_invar`, showing that it is preserved in the recursive execution path (i.e. the one ending in Line 20), and it shows the other invariants needed to prove `alt_labels_invar` that holds. Lastly, we note that we use computation induction to generate the verification conditions for us and Isabelle/HOL's standard automation to combine the invariants towards proving the final correctness theorem of the algorithm.

```
   lemma alt_labels_invar_pres:
 2   assumes
       ass: "if1 flabel ex v1 v2 v3 r" and
 4     invars:
         "⋀v r lab.
 6            flabel v = Some (r, lab) ⟹
                alt_labels_invar flabel r (parent.follow par v)"
 8       "⋀v r.
            flabel v = Some (r, Even) ⟹
10            alt_list (λv. flabel v = Some (r, Even))
                       (λv. flabel v = Some (r, Odd))
12                     (parent.follow par v)"
         "parent_spec par"
14       "flabel_invar flabel"
         "flabel_par_invar par flabel" and
16     inG: "∃lab. flabel' v = Some (r', lab)" and
       flabel':
18       "flabel' = (flabel(v2 ↦ (r, Odd), v3 ↦ (r, Even)))" and
       par': "par' = (par(v2 ↦ v1, v3 ↦ v2))"
20   shows "alt_labels_invar flabel' r' (parent.follow par' v)"
```

Listing 30 Lemma showing preservation of Invariants 1 and 2.

Other than proving that the invariants hold, we perform the rest of the formal proof of correctness of Algorithm 4 by computation induction on `compute_alt_path`, which is arduous but standard.

7 Discussion

Studying combinatorial optimisation from a formal perspective is a well-trodden path, with authors studying flows [11, 17, 18], linear programming [19–21], and online matching [10], to mention a few. The only previous authors to formally analyse maximum cardinality matching algorithms in general graphs, as far as we are aware, are Alkassar et al.[22], who formally verified a certificate checker for maximum matchings. In that work, the formal mathematical part amounted to formally proving Proposition 3, with the focus mainly on verifying an imperative implementation of the checker.

Studying matching has a deep history with the first results dating back to at least the 19th century, when Petersen [23] stated Berge's lemma. Since then, matching theory has been intensely studied. This is mainly due to the wide practical applications, like Kidney exchange [24], pupil-school matching, online advertising [25], etc.

In addition to practical applications, studying the different variants of matching has contributed immensely to the theory of computing. This includes, for instance, the realisation that polynomial time computation is a notion of efficient computation, which is the underlying assumption of computational complexity theory as well as the theory of efficient algorithms. This was first noted by Jack Edmonds in his seminal paper describing the blossom shrinking algorithm [1], where he showed that matchings have a rich mathematical structure that could be exploited to avoid brute-force search for maximum cardinality matchings in general graphs. Other important contributions of studying matchings is that it led to the design of the primal-dual paradigm [26], the Isolating lemma by Mulmuley et al. [27], the complexity class #P [28], and the notion of polyhedra for optimisation problems [29]. All of that makes it inherently interesting to study matching theory and algorithms from a formal mathematical perspective.

We have formally verified Edmonds' blossom shrinking algorithm, building a reasonably rich formal mathematical library on matching and graphs, and arriving at the first complete functional correctness proof of the algorithm, with all invariants stated and proved, and with all case analyses covered. From a formalisation perspective, we believe that tackling Edmonds' blossom shrinking algorithm is highly relevant. First, the algorithm has historical significance, as mentioned earlier, making studying it from a formal mathematical perspective inherently valuable. Second, as far as we are aware, the algorithm is conceptually more complex than any efficient (i.e. with worst-case polynomial running time) algorithm that was treated formally, thus formalising its functional correctness proof further shows the applicability as well as the utility, e.g. to come up with new proofs, of theorem proving technology to complex efficient algorithms.

There are other interesting avenues which we will pursue in the future. The first chiefly practically interesting direction is obtaining an efficient verified implementation of the algorithm. This could be obtained in a relatively straightforward manner by applying standard methods of refinement [30], either top-down using Lammich's framework [31] or bottom-up using Greenway et al.'s framework [32].

Another direction is formalising the worst-case running time analysis of an implementation of the algorithm. There is a number of challenges to doing that. A naive implementation would be $O(|\mathcal{V}(G)|^4)$. A more interesting implementation [33] using the union-find data structure can achieve $O(|\mathcal{V}(G)||G|\alpha(|G|,|\mathcal{V}(G)|))$, where α is the inverse Ackermann function. Achieving that would, in addition to deciding on a fitting methodology for reasoning about running times, need a much more detailed specification of the algorithm. Factors affecting the running time include: keeping and reusing the information from the search performed by Algorithm 4 after shrinking blossoms, and also carefully implementing the shrinking operation without the need to construct the shrunken graph from scratch.

The most interesting future direction is devising a formal correctness proof for the Micali-Vazirani [34] algorithm which has the fastest running time for maximum cardinality matching, and the correctness of which is not yet established, despite many trials at proving it [34–38]. Here, the algorithm achieves a running time of $O(\sqrt{|\mathcal{V}(G)|}|G|)$ by using shortest augmenting paths in phases, akin to Hopcroft-Karp's algorithm [39] for Maximum bipartite matching, as well as avoiding blossom shrinking altogether.

References

[1] Edmonds, J.: Paths, Trees, and Flowers. Canadian Journal of Mathematics (1965/ed)

[2] Berge, C.: Two theorems in graph theory. Proceedings of the National Academy of Sciences (1957)

[3] Mehlhorn, K., Näher, S.: LEDA: A Platform for Combinatorial and Geometric Computing, (1999). http://www.mpi-sb.mpg.de/%7Emehlhorn/LEDAbook.html Accessed 2023-10-09

[4] Korte, B., Vygen, J.: Combinatorial Optimization, (2012)

[5] Schrijver, A.: Combinatorial Optimization: Polyhedra and Efficiency, (2003)

[6] Abdulaziz, M., Mehlhorn, K., Nipkow, T.: Trustworthy graph algorithms (invited paper). In: The 44th International Symposium on Mathematical Foundations of Computer Science (MFCS) (2019)

[7] Paulson, L.C.: Isabelle: A Generic Theorem Prover, (1994)

[8] Noschinski, L.: A Graph Library for Isabelle. Math. Comput. Sci. (2015)

[9] Edmonds, C., Paulson, L.C.: Formal Probabilistic Methods for Combinatorial Structures using the Lovász Local Lemma. In: Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs And Proofs (2024)

[10] Abdulaziz, M., Madlener, C.: A Formal Analysis of RANKING. In: The 14th Conference on Interactive Theorem Proving (ITP) (2023)

[11] Abdulaziz, M., Ammer, T.: A Formal Analysis of Capacity Scaling Algorithms for Minimum Cost Flows. In: The 15th International Conference on Interactive Theorem Proving (ITP 2024) (2024)

[12] Abdulaziz, M.: https://github.com/mabdula/Isabelle-Graph-Library (2024). https://github.com/mabdula/Isabelle-Graph-Library Accessed 2024-10-15

[13] Bondy, J.A., Murty, U.S.R.: Graph Theory, (2008)

[14] Abdulaziz, M., Paulson, L.C.: An Isabelle/HOL Formalisation of Green's Theorem. Journal of Automated Reasoning (2019)

[15] Doorn, F., Massot, P., Nash, O.: Formalising the h-Principle and Sphere Eversion. In: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023 (2023)

[16] Krauss, A.: Automating recursive definitions and termination proofs in higher-order logic. PhD thesis, Technical University Munich (2009). http://mediatum2.ub.tum.de/doc/681651/document.pdf Accessed 2022-03-25

[17] Lee, G.: Correctnesss of ford-fulkerson's maximum flow algorithm1. Formalized Mathematics (2005)

[18] Lammich, P., Sefidgar, S.R.: Formalizing Network Flow Algorithms: A Refinement Approach in Isabelle/HOL. J. Autom. Reason. (2019)

[19] Maric, F., Spasic, M., Thiemann, R.: An Incremental Simplex Algorithm with Unsatisfiable Core Generation. Arch. Formal Proofs (2018). Accessed 2022-09-13

[20] Thiemann, R.: Duality of Linear Programming. Arch. Formal Proofs (2022). Accessed 2023-10-28

[21] Bottesch, R., Haslbeck, M.W., Thiemann, R.: Verifying an Incremental Theory Solver for Linear Arithmetic in Isabelle/HOL. In: Frontiers of Combining Systems - 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings (2019)

[22] Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: A Framework for the Verification of Certifying Computations. CoRR (2013) arXiv:1301.7462. Accessed 2024-12-01

[23] Petersen, J.: Die Theorie der regulären graphs (1891)

[24] Biró, P., van de Klundert, J., Manlove, D., Pettersson, W., Andersson, T., Burnapp, L., Chromy, P., Delgado, P., Dworczak, P., Haase, B., Hemke, A., Johnson,

R., Klimentova, X., Kuypers, D., Nanni Costa, A., Smeulders, B., Spieksma, F., Valentín, M.O., Viana, A.: Modelling and optimisation in European Kidney Exchange Programmes. European Journal of Operational Research (2021)

[25] Mehta, A., Saberi, A., Vazirani, U.V., Vazirani, V.V.: AdWords and generalized online matching. J. ACM (2007)

[26] Kuhn, H.W.: The Hungarian method for the assignment problem. Naval Research Logistics Quarterly (1955)

[27] Mulmuley, K., Vazirani, U.V., Vazirani, V.V.: Matching is as easy as matrix inversion. Comb. (1987)

[28] Valiant, L.G.: The complexity of computing the permanent. Theoretical Computer Science (1979)

[29] Edmonds, J.: Maximum matching and a polyhedron with 0,1-vertices. Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics (1965)

[30] Wirth, N.: Program Development by Stepwise Refinement. Commun. ACM (1971)

[31] Lammich, P., Tuerk, T.: Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In: Interactive Theorem Proving (2012)

[32] Greenaway, D., Andronick, J., Klein, G.: Bridging the Gap: Automatic Verified Abstraction of C. In: Interactive Theorem Proving (2012)

[33] Gabow, H.N.: An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs. J. ACM (1976)

[34] Micali, S., Vazirani, V.V.: An O(VE) algorithm for finding maximum matching in general graphs. In: 21st Annual Symposium on Foundations of Computer Science (FOCS) (1980)

[35] Blum, N.: A New Approach to Maximum Matching in General Graphs. In: Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings (1990)

[36] Vazirani, V.V.: A Theory of Alternating Paths and Blossoms for Proving Correctness of the O(\textbackslashsurdVE) General Graph Matching Algorithm. In: Proceedings of the 1st Integer Programming and Combinatorial Optimization Conference, Waterloo, Ontorio, Canada, May 28-30 1990 (1990)

[37] Vazirani, V.V.: A Theory of Alternating Paths and Blossoms for Proving Correctness of the \emphO(sqrt{\emphV E}) General Graph Maximum Matching

Algorithm. Comb. (1994)

[38] Vazirani, V.V.: A Theory of Alternating Paths and Blossoms from the Perspective of Minimum Length. Math. Oper. Res. (2024)

[39] Hopcroft, J.E., Karp, R.M.: An n\(ˆ\mbox5/2\) Algorithm for Maximum Matchings in Bipartite Graphs. SIAM J. Comput. (1973)