Halvard Hummel

Evaluation of the Iwata–Yokoi Algorithm

Master’s thesis in Computer Science
Supervisor: Magnus Lie Hetland
June 2020

Master’s thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Halvard Hummel

Evaluation of the Iwata–Yokoi Algorithm

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

Given an undirected multigraph, $G = (V, E)$, without self-loops, and a subset of terminals, $T \subseteq V$, a path between two distinct terminals, with all path-internal vertices being non-terminals, is called a T-path. Given this definition, Gallai formulated the edge-disjoint T-paths problem, later popularized by Mader, which is concerned with finding maximum-cardinality sets of pairwise edge-disjoint T-paths, given arbitrary graphs and subsets of terminals.

At SODA 2020, Iwata and Yokoi presented a new combinatorial algorithm for the edge-disjoint T-paths problem, employing a similar strategy to the one used by Edmonds in his blossom algorithm for the maximum matching problem. This master's thesis presents a handful of either slightly mishandled or unhandled edge cases discovered while working with and implementing the Iwata–Yokoi algorithm. The edge cases discussed in this master's thesis are all found in the augmentation procedure of the Iwata–Yokoi algorithm, concerning redundant self-loops, invalid shortcuts, illegal double usage of labeled edges and the appearance of cycles in the T-paths. Each of these edge cases are discussed and appropriate modifications to the algorithm, to mitigate each edge case individually, are presented and proven. The modifications and edge cases are also discussed in the context of the improvements suggested to the, then unpublished, Iwata–Yokoi algorithm in the preliminary project for this master's thesis. Additionally, some minor discussion on efficiency and suggestions for modifications based on predicted empirical efficiency are included.

Sammendrag

Gitt en urettet multigraf, $G = (V, E)$, uten løkker, og en delmengde terminaler, $T \subseteq V$, kalles en sti mellom to distinkte terminaler, hvor alle interne noder i stien ikke er terminaler, en T-sti. Gitt denne definisjonen formulerte Gallai kantdisjunkte-T-stier problemet, som senere ble popularisert av Mader. Dette problemet handler om, for arbitærer grafer og mengder av terminaler, å finne maksimale mengder, med hensyn til kardinalitet, av parvis kantdisjunkte T-stier.

På SODA2020 presenterte Iwata og Yokoi en ny kombinatorisk algoritme for kantdisjunkte-T-stier problemet. Denne algoritmen anvender en lignende strategi som Edmonds' blomstringsalgoritme for maksimal matching problemet. Denne masteroppgaven presenterer en håndfull spesialtilfeller, oppdaget under implementasjon av Iwata–Yokoi algoritmen, som enten er uhåndtert eller delvis feilhåndtert i algoritmen. Disse tilfellene finner alle sted i forøkningsprosedyren til algoritmen, og tar tak i overflødige løkker, ugyldige snarveier, ulovlig dobbel bruk av kanter samt opptredener av sykler i T-stier. Hvert av disse spesialtilfellene blir diskutert og passende endringer av algoritmen blir presentert og får sin korrekthet bevist. Endringene og de problematiske tilfellene blir også diskutert i sammenheng med endringene som ble foreslått til, den da upubliserte, Iwata–Yokoi algoritmen i forprosjektet til denne masteroppgaven. I tillegg inneholder masteroppgaven noen foreslåtte endringer og diskusjon knyttet til empirisk ytelse.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Given an undirected multigraph, $G = (V, E)$, without self-loops, and a subset of terminals, $T \subseteq V$, a path between two distinct terminals, with all path-internal vertices being non-terminals, is called a *T-path*. In 1961 Gallai posed a question regarding how many T-paths could be packed into an arbitrary graph with a given set of terminals [1]. There are two versions of the problem, the edge-disjoint T-paths problem and the vertex-disjoint T-paths problem. They regard finding, respectively, maximum-cardinality sets of pairwise edge-disjoint and vertex-disjoint T-paths. This thesis will focus on the edge-disjoint T-paths problem.

In 1978 Mader presented formulas for both versions of the T-paths problem [2, 3]. Both formulas apply a similar iterative approach, over all possible divisions of the given graph, counting the number of possible paths between the components of the divided graph. While Mader's formulas provided a way to calculate the correct number of edge-disjoint T-paths in a maximum-cardinality set, they did not offer a way to construct such sets. In the 40 years since the publication of Mader's papers, several different algorithms have been presented for the two versions of the problem, employing a wide variety of algorithmic concepts. While there exist many different algorithms for the problem, most solve generalized or slightly modified versions of the T-paths problems, e.g., minimum-cost. These algorithms can usually be directly applied to the T-paths problems through simple reductions.

The most recent algorithm, presented by Iwata and Yokoi at SODA 2020, is a deterministic algorithm employing a similar strategy to that of Edmonds' blossom algorithm [4, 5]. The Iwata–Yokoi algorithm achieves the best time complexity of all known deterministic solutions, $\mathcal{O}(VE^2)$. Additionally, there exists a randomized algorithm achieving better time complexity for non-dense graphs. The algorithm employs a reduction, discovered by Schrivjer, to the linear matroid parity problem [6]. Using Cheung, Lau and Leung's randomized algorithm for

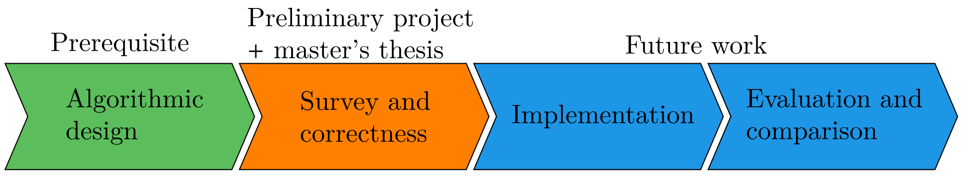


Figure 1.1: Project timeline for a practical approach to the T-paths problems

the linear matroid parity problem, it is possible to achieve a time complexity of $\mathcal{O}(E^{\omega})$, where ω is the matrix multiplication factor, currently at approximately 2.373 [7, 8]. For a more in-depth discussion on the different algorithms, including the ones not mentioned here, see the preliminary project for this master's thesis [9]. Alternatively, the introduction of Iwata and Yokoi's paper provides a short historical timeline of the state of the problem.

So far, all research on the T-paths problems has been completely theoretical, with no focus on comparison and evaluation of efficiency outside of complexity theory. Aiming at expanding the research on the topic with a practical approach, eventually performing empirical evaluation of the different algorithms, and comparison based on empirically measured efficiency, the preliminary project and this master's thesis focus on required initial surveying and evaluation of correctness. Figure 1.1 places these two technical reports on the timeline for the practical approach as a whole. The preliminary project contains a general survey of the edge-disjoint T-paths problem. Additionally, it contains an initial look at implementation details of the Iwata–Yokoi algorithm, with suggested changes to achieve a slightly better time complexity. This master's thesis was supposed to further the practical approach, looking at implementation and initial emperical evaluation of the Iwata–Yokoi algorithm. However, due to the discovery of several unhandled edge-cases in the Iwata–Yokoi algorithm, of varying degrees of non-triviality, the thesis has moved towards a more in-depth theoretical evaluation of the Iwata–Yokoi algorithm, inspecting its correctness. While the general concepts of the algorithm work properly, we will show a handful of edge-cases that are either not covered by or not correctly handled by Iwata and Yokoi. Appropriate modifications to mitigate these non-trivial edge-cases will be presented, accompanied by correctness proofs.

The thesis is structured as follows. Chapter 2 provides an introduction to the required knowledge about the edge-disjoint T-paths problem, as well as an in-depth guide to the Iwata–Yokoi algorithm. Additionally, it touches on the notation, terms and style of figures used throughout the thesis. Chapters 3 through 6 each focuses on either distinct parts of the algorithm or types of edge cases. Each of the chapters introduces a mis- or unhandled edge case, proving the occurrence of the edge case, and suggesting and proving correctness of modifications to the algorithm to mitigate the missing handling of the case. Chapter 3 discuss the appearance of so-called redundant self-loops. Chapter 4 describes two cases in which the shortcut-finding procedure discovers shortcuts when none is possible. Chapter 5 introduces a case in which the partial symmetric difference operation creates invalid augmenting walks. Finally, Chapter 6 shows the appearance of cycles in T-paths, providing two distinct removal approaches. Chapter 7 contains concluding remarks and discussion on future work.

Chapter 2

Background

This chapter covers the background information required to understand the main contents of this thesis. Section 2.1 contains information about the definitions and notation used in this thesis, along with an explanation of how to read the drawn graphs. Then, in Section 2.2, a general introduction to the edge-disjoint T-paths problem is given, with Section 2.3 containing a thorough explanation of the Iwata–Yokoi algorithm. This thesis is only concerned with this algorithm and thus does not touch on any of the other existing algorithms. For discussion on all known algorithms, as well as a general survey of the edge-disjoint T-paths problem, see the report from the preliminary project for this thesis [9].

2.1 Definitions, Notation and Figures

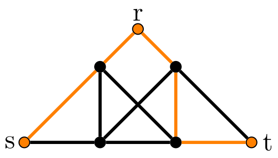
We define a *graph*, $G = (V, E)$, as a set of *vertices* and a set of *edges* connecting pairs of vertices. All graphs are assumed to be *undirected* unless otherwise stated, that is, all edges can be used in both directions. A graph is said to be *simple* if there is at most one edge between any pair of vertices, otherwise it is called a *multigraph*. The vertices an edge connects are called the *adjacent vertices* of the edge. If an edge only has one adjacent vertex it is called a *self-loop*, connecting a vertex to itself. In a graph, a *walk* is defined as an ordered list of edges and vertices, $v_0, e_1, v_1, \dots, e_n, v_n$, where edge e_i connects vertices v_{i-1} and v_i. A walk is called a *path* if each vertex in the graph appears at most once, that is, there are no cycles in the walk. All vertices on the walk except for v_0 and v_n are called the *internal vertices* of the walk. We say that e_i is walked in the direction from v_{i-1} to v_i. If a walk contains a cycle, we call the vertex visited twice in the cycle the *connecting vertex* of the cycle. A walk in a graph with vertices $a, b, \dots, z, 1, 2, 3, \dots$ can be denoted by $a-1-3-2-c$, the sequence of vertices visited by the walk, if it is unambiguous which edges are used by the walk. Not to be confused with the later introduced notation of *symbols* on a walk (introduced in Section 2.3), $abcbcb$.

A graph is said to be a *tree* if there is exactly one possible walk between any pair of vertices in the graph. A tree is said to be *rooted* if a single vertex in the tree has been designated as a *root node*. A graph consisting of components that are all trees is called a *forest*. Note that a single vertex forms a tree, so a forest can also be defined as a graph where there for any pair of vertices exists at most one walk between the vertices, i.e., an *acyclic graph*. In the context of expanding a rooted forest, an edge is called a *frontier edge* if inserting it into the forest results in combining a single-vertex tree with a rooted tree. In this context, all other edges are called *interior edges*.

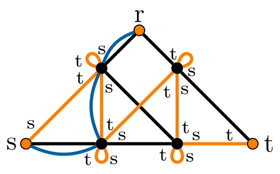
For T-paths and the concepts of the Iwata–Yokoi algorithm, we adopt a similar notation as the one used by Iwata and Yokoi. We define P to mean a set of edge-disjoint T-paths, if not otherwise specified. P_j is defined to be the j-th T-path in P. P_j may be used to denote an arbitrary T-path, without any references to P; in these cases it should follow from context what T-path P_j is referring to. Further notation related to the concepts of the Iwata–Yokoi algorithm will be introduced, together with the underlying concepts, in Section 2.3.

Illustrating the edge-disjoint T-paths problem requires drawing graphs with properties that are not found in standard graphs. We thus, in this thesis, employ a color-coding of edges and vertices to distinguish their properties. While normal vertices in a graph will be drawn in black, terminals will be drawn in orange. Both terminals and non-terminals may be labeled in a graph; in these cases terminals will always be labeled by letters, while non-terminals will be labeled by numbers. Edges are normally drawn in black; however, if they belong to a T-path they will be drawn in orange (light gray in black and white). For the case of the Iwata–Yokoi algorithm (see Section 2.3), an augmenting walk will have its edges drawn in blue (dark gray in black and white) besides the already drawn edges in the graph. If an edge is labeled in the Iwata–Yokoi algorithm, the labeling is given close to each end of each edge. The labeling will in some cases be omitted to simplify the figures. While the edge-disjoint T-paths problem deals with undirected graphs, edges in T-paths may at times be drawn as directed edges. This is to indicate the order of the edges in the T-paths when there are cycles present in them. Additionally, edges in T-paths may be drawn as dotted or dashed lines to differentiate between distinct T-paths. Dashed edges in the augmenting walk indicates that the augmenting walk continues outside of the drawn part of the graph. Figure 2.1 contains examples of graphs drawn in the style described above.

When referring to time complexity, as in $\mathcal{O}(E)$, E and V refer to, respectively, the number of edges and vertices in the given graph. Additionally, in this context



(a) Example for the edge-disjoint T-paths problem



(b) Example for the Iwata-Yokoi algorithm

Figure 2.1: Example graphs to show the notation used in this thesis

Q and P_j refer to the maximum length of, respectively, the augmenting walk and a T-path.

For pseudocode, a similar style to the one employed by Cormen, Leiserson, et al. in their introductory book on algorithms [10] will be used. Generally, indents will be used to separate logical scopes of code (if/else, for/while, etc.). Variables defined in one scope is available in all other scopes of a procedure, but not across procedures. Variables are either simple values (integer, boolean, byte, etc.) or treated as objects. An object is here a collection of variables, each accessible through dot notation, e.g., *e.src*. Accessing and slicing lists is considered in the same way that Julia does (similar to the pythonic way), using bracket notation, colon for slicing, **end** to mean the index of the last element in the list and 1-indexing. Finally, the functions ENUMERATE and FILTER are assumed to exist, respectively turning a list of values into a list of tuples, (index, value), and filtering a list on a given condition.

2.2 The Edge-Disjoint T-Paths Problem

Let $G = (V, E)$ be a multigraph without self-loops, and let $T \subseteq V$ be a set of terminals. Then a *T-path* in G with regards to T is a path in the graph, that starts and ends in distinct terminals, with each internal vertex of the path not in T. A set P of T-paths is said to be *edge-disjoint* if and only if the T-paths in the set are all pair-wise edge-disjoint. Given a multigraph $G = (V, E)$ without self-loops and a set of terminals, $T \subseteq V$, the edge-disjoint T-paths problem asks to find any maximum-cardinality set of edge-disjoint T-paths. For any such graph and set of terminals, there may exist more than one solution to the edge-disjoint T-paths problem.

While the edge-disjoint T-paths problem is concerned with finding a maximum-cardinality set of edge-disjoint T-paths, the initial work on a solution was made by Mader [2]. He derived a formula for finding the size of a maximum-cardinality set of edge-disjoint T-paths. While this thesis is only concerned with the Iwata–Yokoi algorithm, we will include Mader's theorem in addition to a short description of its workings. This theorem provides a different way of viewing the problem, which might help illustrate the complexity of the problem and its algorithms. Mader's theorem is repeated below, in Theorem 2.1. The formulation of the theorem here is the same as used in the preliminary project for this thesis, that is the aforementioned survey [9]. This formulation builds on the works of Schrivjer, Iwata and Yokoi, and Keisjper et al. [4, 6, 11].

Theorem 2.1 (Mader's theorem) *Given a multigraph $G = (V, E)$ and a subset $T \subseteq V$, then the maximum number of edge-disjoint T-paths for graph G and T, $\lambda(T;G)$, is given by*

$$\lambda(T;G) = \min_X \frac{1}{2}\left[\sum_{t\in T} d(X_t) - \operatorname{odd}(G\setminus X)\right]$$

where X is a collection of $|T|$ disjoint subsets, $X_t \subseteq V$ such that each X_t contains a single terminal $t \in T$. $d(S)$ denotes the number of edges going out of a subset $S \subseteq V$ to $G \setminus S$, and $\boldsymbol{odd}(G\setminus X)$ denotes the number of components C_i of $G \setminus X$ with $d(C_i)$ odd.

To understand what Mader's theorem does, one has to consider a graph as a set of components connected by edges, that is, a graph can be divided into disjoint sets of vertices in any number of ways. For the edge-disjoint T-paths problem, one can consider the graph to be divided by $|T|$ such disjoint sets of vertices, with each set containing a single terminal and zero or more non-terminals. For any such division of the graph, a T-path must use the edges between the components of the graph, with one for each of the two components with terminals it will visit. For visited components without terminals, an even number of these external edges must be used. What Mader's theorem does is to iterate over all such possible division of the graph, finding the most restrictive one based on the number of edges going between the components. This division of the graph finds the choke points for the T-paths, and thus the most restrictive division gives an approximation to the size of the maximum-cardinality sets of T-paths. The reasoning behind this, is that while the less restrictive division may seem to be able to pack more T-paths, the counting of edges does not take into consideration how many components a T-path must visit on its way between the components with terminals. A full proof of the theorem will not be given here. The interested reader can refer to any of the four aforementioned works for a detailed proof.

2.3 The Iwata–Yokoi Algorithm

Here, an in-depth description of the Iwata–Yokoi algorithm is given. The entire algorithm will be covered, but the level of detail of the different components will differ based on what and how much is required in order to understand the later discussion.

In 2020 Iwata and Yokoi published a paper detailing a new deterministic algorithm for the edge-disjoint T-paths problem, building on a similar scheme to that of Edmonds' famous blossoming algorithm for the maximum matching problem [4, 5]. While their algorithm was not given a specific name, we will in this thesis refer to the algorithm as the Iwata–Yokoi algorithm, as was also done in the preliminary project. The Iwata–Yokoi algorithm is an iterative algorithm, where each iteration takes a set of n edge-disjoint T-paths as input, and either returns a set of $n+1$ edge-disjoint T-paths or guarantees that the set of n edge-disjoint T-paths is a set of maximum cardinality.

A central concept in the Iwata–Yokoi algorithm is the so-called *labeled graph*. For an instance of the edge-disjoint T-paths problem, the labeled graph is a modified copy of the underlying graph, where the modifications are dependent on the current state of the algorithm, that is, the modifications depend on the currently found T-paths. A labeled graph is thus defined on a graph, G, and a set of T-paths, P. In the labeled graph, each terminal is given a unique label. Going from terminal s to terminal t on a T-path, the first end of each edge is labeled by s, while the second end is labeled by t. Additionally, a self-loop is created on each of the internal vertices of the T-path. These self-loops are labeled in the same way as the normal edges. Note that the order in which the symbols are added to a self-loop does not matter, as long as the order is fixed after creation. Both the self-loops and the edges of a T-path are said to be *labeled*, while all other edges are called *free*. An example of this can be seen in Figure 2.1b, where an example of a labeled graph, with a single T-path, is shown.

For any walk, W, in the labeled graph, the *symbols* on the walk, denoted by $\gamma(W)$, are the ordered string of symbols that appear on the vertices and edges of the walk, going from v_0 to v_n. Using this definition of symbols on a walk, another central concept of the Iwata–Yokoi algorithm can be introduced. An *augmenting walk* is a walk in the labeled graph, where the following properties hold.

(i) The walk is between two terminals (v_0 and v_n), with all internal vertices of the walk being non-terminals.

(ii) In the symbols on the walk, there are no consecutive appearances of the same symbol.

(iii) The walk uses each free edge and self-loop at most once, with all other labeled edges being used at most once in each direction.

While we will not cover the reasoning behind these properties in detail, note that when there are no labeled edges in an augmenting walk, the augmenting walk forms a T-path that is pair-wise edge-disjoint with all other T-paths in the labeled graph. In general, the current augmenting walk in the algorithm will be denoted by Q. Also, note that it can be shown that a set of edge-disjoint T-paths has maximum cardinality if and only if it is impossible to construct any augmenting walk in the labeled graph.

Each iteration of the Iwata–Yokoi algorithm consists of two procedures, the search procedure and the augmentation procedure. The search procedure tries to find an augmenting walk, guaranteeing that if it does not find an augmenting walk, it is impossible to construct one in the labeled graph. If the search procedure finds an augmenting walk, then the augmentation procedure applies a set of sub-procedures iteratively modifying the augmenting walk and T-paths until the augmenting walk no longer contains any labeled edges. Then, the algorithm creates a new set of T-paths, containing the modified T-paths and the augmenting walk. By iterating until the search procedure does not return an augmenting walk, the algorithm expands a set of edge-disjoint T-paths, guaranteeing that the found set has maximum cardinality at the termination of the algorithm.

2.3.1 The Search Procedure

The search procedure tries to find an augmenting walk in the labeled graph. This is done through a forest expansion process, where a forest is iteratively expanded using vertices and edges from the labeled graph. An example of this process is shown in Figure 2.2. Initially, the forest consists of a set of single-vertex rooted trees, one for each terminal in the labeled graph. This can be seen in Figures 2.2a and 2.2b, where the underlying labeled graph and the initial forest are shown. Then, through adding edges (and vertices) to the graph, the rooted trees are expanded. The point here is to create a forest under certain restrictions. This, such that adding any interior edge in the forest that suffice certain properties will result in the interior edge along with the paths from the adjacent vertices of the interior edge to their respective terminals (root vertices) forms an augmenting walk. The algorithm guarantees this by limiting which edges can be used in the expansion of the forest.

The forest expansion consists of a two step process, repeated until either an augmenting walk is found or it runs out of edges. For any part of the process, an edge is only considered if for each of its adjacent vertices, the symbol on the edge closest to the vertex is distinct from the label of the vertex, or if the vertex does not have a label the first symbol on the walk from the vertex to root. The reason

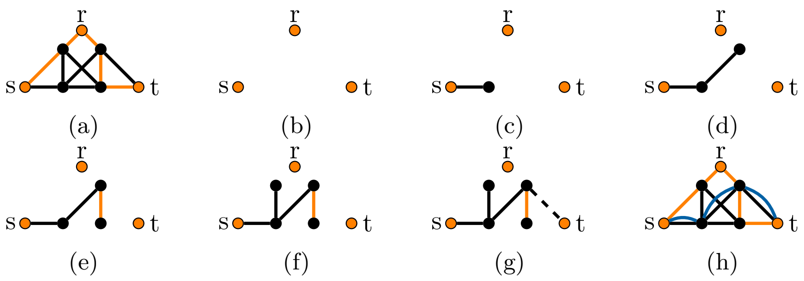


Figure 2.2: Example of the search procedure

behind this, is that to able to find an augmenting walk, one must guarantee that property (ii) holds. Thus the forest is expanded in such a way that this property holds for the walks from any vertex to its respective root vertex.

The first step of the forest expansion is the addition of frontier edges. Here, frontier edges from the labeled graph, that suffice the above property, are added until the process runs out of frontier edges. The edges can be chosen in any order. Figures 2.2c, 2.2d, 2.2e and 2.2f show an example of this part of the expansion. Then, in step two, a single interior edge is selected from the set of possible interior edges. This edge may fall into one of three categories: (1) it connects two vertices in distinct trees, (2) it connects two vertices in the same tree that do not share any edges on their walks to the root vertex, and (3) it connects two vertices in the same tree that share some sub-path on their walks to the root vertex. For cases (1) and (2), this interior edge forms an augmenting walk in the aforementioned way, and the search procedure can return this augmenting walk. In case (3) it does not form an augmenting walk, as, due to the restrictions set for the added edges, the shared edge closest to the root vertex will be a free edge. Thus (iii) would not hold for the constructed walk. Figure 2.2g shows an example of an interior edge, dashed in the figure, added to the forest. This edge is in category (1) and Figure 2.2h shows the resulting augmenting walk in the labeled graph.

To handle case (3) and be able to guarantee that the search procedure finds an augmenting walk when possible, the concept of a *blossom* is introduced. A blossom is a structure in the graph, consisting of a cycle and possibly a path outside of this cycle. The usage of the concept here is similar to how Edmonds used the concept in his blossom algorithm, where for an arbitrary augmenting walk one can always find a walk through the blossom that fits into the augmenting walk without breaking (ii) and (iii). That is, an entire blossom can be treated as a single wildcard vertex that no matter the symbols around it can be expanded in such a way that a valid augmenting walk can be created.

In the Iwata–Yokoi algorithm, a blossom is created every time an interior edge

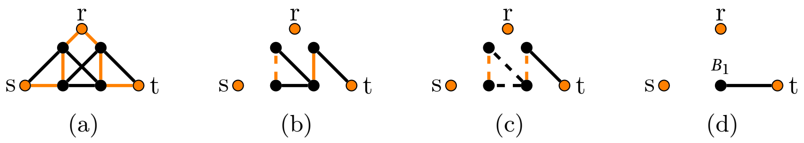


Figure 2.3: Example of the creation of a blossom

that falls into category (3) is selected in the second step of the procedure. The blossom consists of the cycle created by the interior edge and the distinct initial parts of the walks to the root vertex from the adjacent vertices of the interior edge. Additionally, labeled edges are added to blossom from the start of the common sub-path to the root vertex until a free edge is found. So, depending on how this common sub-path looks, anywhere from zero to all but one of its edges might be added to the blossom. When a blossom is created, the vertices in the blossom are contracted into a single vertex in the graph with a special label, and the search procedure jumps back to step 1. The contraction creates copies of the edges connecting vertices in the blossom to the rest of the graph, now connecting the blossom vertex to the rest of the graph. Figure 2.3 shows an example of the creation of a blossom, where the underlying labeled graph is shown in Figure 2.3a. Figure 2.3b shows the search procedure after selecting an interior edge in category (3), dashed in the figure. In Figure 2.3c, the edges of the resulting blossom are shown as dashed, with Figure 2.3d showing the state of the forest after the creation of the blossom.

If the second step returns an augmenting walk containing one or more blossoms, each blossom can be expanded by first trying to take the direct route out of the blossom, that is, going straight towards the root vertex. If the symbol preceding the entry to the blossom does not allow for this, the requirements set earlier for frontier edges guarantees that walking around the cycle in the blossom does not result in consecutive appearances of symbols in the walk. Since the only part of the blossom that may be used twice is the edges in the common sub-path, (iii) must hold in all such situations as these edges are all labeled. Figure 2.4 shows two examples of expansion of a blossom in an augmenting walk. In Figure 2.4a, the symbol on the augmenting walk preceding the blossom is s. Since it is possible to go directly up the blossom without breaking (ii), this direction is chosen. In the case of Figure 2.4b, the preceding symbol, u, makes it impossible to go directly up the blossom and the long way around is taken instead.

Using this blossom concept, Iwata and Yokoi showed that either an augmenting walk is found in the search procedure or there are none in the labeled graph. There are a couple of important properties of this returned augmenting walk that will be needed for the later discussion. First, the returned augmenting walk consists of at most two visits to each vertex; as such, it has a length of less than

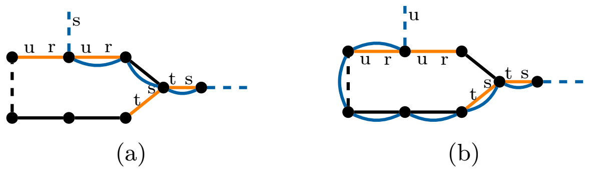


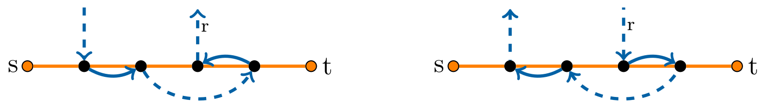
Figure 2.4: Example of a blossom expansion. The interior edge that created the blossom is dashed.

$2|V|$. Second, due to the chosen expansion method for blossoms, all self-loops that appear in the augmenting walk are necessary to maintain (ii).

2.3.2 The Augmentation Procedure

The augmentation procedure takes the augmenting walk outputted by the search procedure and tries to simplify the augmenting walk such that it becomes pairwise edge-disjoint with the T-paths. It does this by performing a set of operations that can modify both the augmenting walk and the T-paths. These operations aim at reducing the overlap between the edges used in the augmenting walk and the T-paths. By guaranteeing that each operation results in a decrease in this overlap, while maintaining the properties of the augmenting walk and T-paths, the procedure will in the end result in the augmenting walk having no overlap with the T-paths.

When talking about overlaps between the augmenting walk and the T-paths, the overlaps are not generally considered as overlaps of single edges, but rather as sequences of overlapping edges, that is, a so-called concept of *P-segments* is used. A P-segment is defined as a continuous part of the augmenting walk consisting entirely of edges from a single T-path, with the preceding and succeeding edges in the augmenting walk not belonging to the T-path. While a P-segment is any such overlap, an overlap is called a P_j-segment if it is an overlap of the j-th T-path. Given this definition of P-segments, the augmentation procedure aims to continuously reduce the number of P-segments in the augmenting walk. That is, the number of P-segments has to decrease every iteration of the procedure, while the number of overlapping edges does not matter in itself. The number of P-segments in the augmenting walk may be denoted by $\mu(Q)$. A P-segment is called *st-directed* if it belongs to a T-path which has end-vertices labeled s and t, and the edges in the P-segment are used in the direction going from s to t on the T-path. This is a general concept, where the labels of the terminals of a T-path are normally assumed to be s and t. Similarly, given an st-directed P_j-segment, a P_j-segment is called *ts-directed* if it uses the edges of the T-path in the opposite direction.

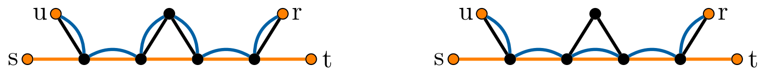


(a) Type 3 (b) Type 4

Figure 2.5: Examples of shortcut types 3 and 4

The augmentation procedure performs two different operations, a shortcut operation and a partial symmetric difference operation, depending on the current state of the augmenting walk. For both of the operations, only the P_j-segments of the T-path of first P-segment in the augmenting walk is considered. Thus, when talking about these operations and P_j-segments, P_j refers to this T-path. In a single iteration of the augmentation procedure the shortcut operation is always performed if the augmenting walk allows for it, with the partial symmetric difference operation performed only when it is not possible to perform the shortcut operation. This is due to the shortcut operation establishing certain properties for the interaction between the augmenting walk and the T-path when there are no more shortcuts possible. These properties are required in order to prove that the partial symmetric difference operation results in an augmenting walk and valid T-paths.

The shortcut operation tries to find two P_j-segments such that replacing the part of the augmenting walk between the start of the first and end of the second with the part of the T-path between the two results in the augmenting walk staying valid. In some cases, the start and end of the P_j-segments may be the same vertex; in such cases, a self-loop on the given vertex may be inserted, but only when it is necessary to suffice (ii). Iwata and Yokoi classify four types of possible shortcuts, classified based on the order of the P_j-segments in both the augmenting walk and the T-path. This thesis will require knowledge of two of these, namely type 3 and 4. Type 3 shortcuts are shortcuts between an st-directed and a ts-directed P_j-segment, where the st-directed P_j-segment appears before the ts-directed one in the augmenting walk and has its initial vertex being either the same vertex as the end vertex of the ts-directed P_j-segment or appearing on the s-side of this vertex. Additionally, the symbol succeeding the ts-directed P_j-segment must not be a t. Refer to Figure 2.5a for an example of such a situation. A type 4 shortcut requires the same set of one st-directed and one ts-directed P_j-segment. In this type, the end vertex of the ts-directed P_j-segment is either the same as the initial vertex of the st-directed one or appears s-side of this vertex. Additionally, the symbol preceding the st-directed P_j-segment must not be t. Refer to Figure 2.5b for an example of a type 4 shortcut situation. If the reader is interested in the remaining types of shortcuts, refer to Iwata and Yokoi's



(a) Before (b) After

Figure 2.6: Example of the shortcut operation

paper.

In their paper, Iwata and Yokoi outline a five step process for finding possible shortcuts. Step 0 builds data structures of information about the augmenting walk and its P_j-segments. The step consists of an iteration over the edges in the augmenting walk, setting p_{st} or p_{ts} to x for a given edge if the edge belongs to the x-th P_j-segment and it is, respectively, *st*-directed or *ts*-directed. Additionally, for each P_j-segment found, *enter(x)* and *leave(x)* is set if, respectively, the preceding and succeeding symbols in the augmenting walk are neither s nor t. The remaining four steps each find shortcuts of their respective type. Since they are not needed, steps 1 and 2 are not explained here. The only required knowledge about these steps is that if they do not find a shortcut, then the values of p_{st} and p_{ts} decrease walking the T-path from, respectively, s and t. Additionally, they guarantee that P_j-segments with the same direction are vertex-disjoint.

Step 3 finds shortcuts of type 3 by iterating over the edges and self-loops on the T-path from s to t. Each time an edge with p_{st} set is found p^*_{st} is updated to this value. Finding an edge for which p_{ts} is greater than p^*_{st} and *leave* is set, a shortcut with the two P_j-segments is returned. The step also contains a second iteration, with the roles of st and ts reversed, iterating over the T-path in the opposite order. In a similar fashion, step 4 finds shortcuts of type 4. It also performs this double iteration over the T-path, with the roles of st and ts reversed in the second iteration. Differentiating from step 3, step 4 keeps track of the last seen *ts*-directed P_j-segment, p^*_{ts}, comparing found *st*-directed P_j-segments to this p^*_{ts} value, checking if they have a lower p_{st} value. Additionally, *enter* is used instead of *leave*.

A simple example of the execution of the shortcut operation can be seen in Figure 2.6. There, in Figure 2.6a, an augmenting walk with two P-segments belonging to the same T-path can be seen. The setup shows an example of the simplest type of shortcut, type 1, where the two P_j-segments are equally directed and appear in the same order in both the augmenting walk and the T-path. In such a case, the augmenting walk can be simplified as shown in Figure 2.6b, where the part of the augmenting walk between the two P_j-segments has been changed to the part of the T-path between the two. Note that while there are now three edges that overlap between the augmenting walk and T-paths, as opposed to the earlier two, the number of P-segments has decreased by 1. Iwata and Yokoi showed that the shortcut operation always reduces the number of P-segments by

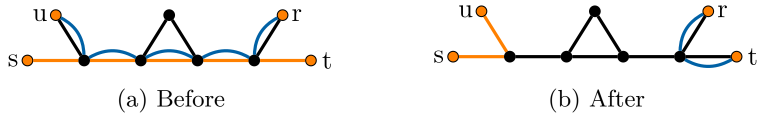


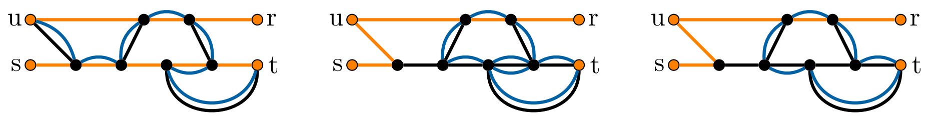
Figure 2.7: Example of the partial symmetric difference operation

at least 1, more if there were other P-segments in the augmenting walk between the two.

In the example of Figure 2.6 the length of the augmenting walk decreases. This is not always the case, and the length of the augmenting walk may increase by almost the entire length of the T-path. Additionally, while not shown in the figure, the new edges inserted into the augmenting walk may already be in use in the augmenting walk. If this is the case, to make sure that (iii) is kept intact, one of the occurrences and the edges between the two occurrences are removed from the augmenting walk.

The partial symmetric difference operation performs a special interchange of the edges in the T-path and the augmenting walk. The new T-path is formed by concatenating the edges in the T-path before the P_j-segment with the edges of the augmenting walk before the P_j-segment. In a similar fashion, the new augmenting walk is formed by concatenation of the edges after the P_j-segment in the T-path and the edges after the P_j-segment in the augmenting walk. An example of this procedure is shown in Figure 2.7, wherein Figure 2.7a an augmenting walk with a single P-segment is shown. While there is no direction shown in the figure, there is an assumption here that the augmenting walk is considered to go from u to r. Given this, the result of the partial symmetric difference operation can be seen in Figure 2.7b. While the length of both the T-path and the augmenting walk decreased in the example of Figure 2.7, this is not always the case. The combined length of the two will always decrease, while the interchange of edges may result in the length of one of them increasing.

The shortcut operation reduces the number of P_j-segments in the augmenting walk. Nonetheless, there is no guarantee that there is only one P_j-segment in the augmenting walk when the partial symmetric difference operation is performed. This, as there are some combinations of P_j-segments where the symbols on the augmenting walk do not allow for shortcuts. Thus, there may be P_j-segments both on the s-side and t-side of the P_j-segment removed in the partial symmetric difference operation. For the ones on the s-side, the shortcut operation guarantees that the possible change in symbols on the T-path does not result in (ii) being invalidated. On the t-side, on the other hand, the edges of the T-path are now free edges, and (iii) may be broken from dual usage of these free edges. As such, a second operation, the uncrossing operation, is performed at the end of the partial



(a) Before (b) After interchange (c) After uncrossing

Figure 2.8: Example of the partial symmetric difference operation with uncrossing

symmetric difference operation.

The uncrossing operation removes all dual usages of the now free, previously labeled, edges in the augmenting walk. The shortcut operation guarantees that these dual usages are both equally directed. Since one of the usages did not appear in the augmenting walk before the partial symmetric difference operation, simply removing them in the way that was done in the shortcut operation would risk invalidating (ii). Instead Iwata and Yokoi found that the guarantees of the shortcut operation allow the two usages to both be removed when the edges between them are flipped (used in the opposite direction and order).

Figure 2.8 shows an example situation where the uncrossing operation must be performed as part of the partial symmetric difference operation. Figure 2.8a contains the initial situation with an augmenting walk containing three P-segments, two belonging to the same T-path. It is not possible to perform a shortcut operation here, as that would result in the augmenting walk having a repetition of symbols. Thus, the partial symmetric difference operation is performed. Figure 2.8b shows the result of performing the interchange of the edges. There, the edge of what was the third P-segment is used twice. While it may not be immediately obvious from the figure, the edge is used twice in the same direction. Performing the uncrossing operation results in the situation of Figure 2.8c, where the two usages of the free edge are removed.

The partial symmetric difference operation has several interesting properties. First, it can be seen that the operation deletes the edges of a P-segment, reducing the number of P-segments by at least one. Especially, it can be shown that the new edges in the augmenting walk can not form any P-segments. By this, it also follows that the number of overlapping edges decreases. Moreover, the T-path changed in the operation will remain a valid T-path since the edges gained from the augmenting walk where all free edges prior to the operation.

In many ways, the shortcut operation and partial symmetric difference operation can be considered as two operations that in combination push the occurrences of P-segments in the augmenting walk backwards. The shortcut operation declutters the internals of the augmenting walk, allowing the partial symmetric difference operation to push the boundary of where the P-segments may occur in the augmenting walk.

2.3.3 Time Complexity

While the exact time complexity of the Iwata–Yokoi algorithm is not important for the discussion here, the time complexities of the sub-procedures are needed to show that the later modifications to the algorithm do not result in an increase in time complexity. We thus give a short introduction to the time complexity of the algorithm.

As previously mentioned, the Iwata–Yokoi algorithm consists of a set of iterations, each finding one additional T-path. As such, there can be a maximum of $|E|$ iterations. Each iteration consists of one application of the search procedure, which Iwata and Yokoi showed to have a time complexity of $\mathcal{O}(VE)$. Additionally, the augmentation procedure consists of iterations where either the shortcut operation or the partial symmetric difference operation is performed. Each of these operations was shown to have a time complexity of $\mathcal{O}(E)$. Since both operations reduce the number of P-segments by at least one, and the number of P-segments after the search procedure can not be larger than $\mathcal{O}(V)$, due to the limits on the length of the augmenting walk, Iwata and Yokoi showed that there are at most $\mathcal{O}(V)$ iterations in the augmentation procedure. As such, the time complexity of the augmentation procedure was shown to be $\mathcal{O}(VE)$. In conclusion, this gives the Iwata–Yokoi algorithm a time complexity of $\mathcal{O}\left(VE^2\right)$.

2.3.4 Improvements From the Preliminary Project

As part of the preliminary project for this thesis, possible improvements, in regards to time complexity, were found for the Iwata–Yokoi algorithm. These improvements resulted in a reduction in the time complexity of the algorithm to $\mathcal{O}\left(V^2E+E^2\right)$. As the changed algorithm does not greatly differ from the Iwata–Yokoi algorithm, the later discussed modifications to the Iwata–Yokoi algorithm should be shown to also work properly for these improvements without reducing the improvements to the time complexity. Thus, the changes will be briefly covered here.

The improvements suggested are twofold. For the search procedure a change in data structures was suggested. An edge should refer to the position of its adjacent vertices in an array rather than the vertices itself. Using this, it was shown that blossom creation had a time complexity of $\mathcal{O}(V)$ rather than the $\mathcal{O}(E)$ time complexity of a standard contraction in the graph. This, as blossom creation, and thus the contraction of the graph, could be done without modifications to the edges by simply replacing the vertices of the blossom in the vertex array. This resulted in the search procedure having a time complexity of $\mathcal{O}\left(V^2+E\right)$.

For the augmentation procedure the time complexity is mainly bound by the size of the augmenting walk. As previously mentioned, the length of the augmenting walk may increase from both the shortcut operation and the partial symmetric difference operation. This results in the length of the augmenting walk not being bound by $\mathcal{O}(V)$, but rather by $\mathcal{O}(E)$. Since the augmenting walk initially has at most two visits to each vertex, it was showed that both operations result in at most one additional visit to any vertex. Thus, exploiting the properties of the augmenting walk, a procedure, Simplify, was designed to remove at least one visit from each vertex that was visited thrice. This resulting in the augmenting walk having a length of $\mathcal{O}(V)$, and the augmentation procedure having a time complexity of $\mathcal{O}\left(V^2\right)$.

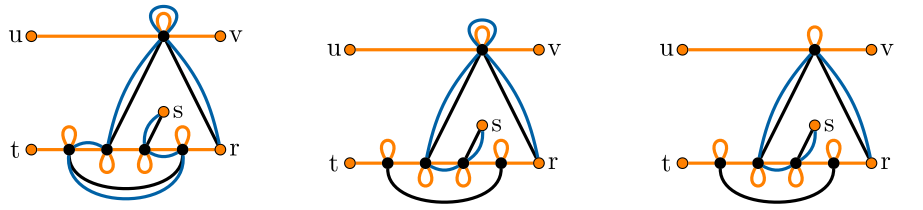
Chapter 3

Redundant Self-Loops

In their paper, Iwata and Yokoi make an assumption regarding so-called *redundant* self-loops, that is, self-loops in the augmenting walk that can be deleted without resulting in the appearance of consecutive symbols on the augmenting walk. The authors make the assumption that without loss of generality one can assume the augmenting walk to have no such self-loops. The usage of this assumption seems to mainly be related to the definition of a P-segment. This, as with the assumption it is guaranteed that a P-segment is either a single self-loop or a sub-path of a T-path. For this guarantee, the given reliance on no redundant self-loops is stronger than necessary. This, as a self-loop may be redundant even when it does not appear directly before or after an edge of the same T-path in the augmenting walk. As will be seen shortly, the algorithm will in some cases be able to break the assumption above and produce what we will call a *standalone* redundant self-loop, that is, a redundant self-loop that forms a P-segment containing only itself.

We have not been able to prove or disprove the occurrence of non-standalone self-loops. It seems likely that the occurrence of these is not possible in the standard operational space of the Iwata–Yokoi algorithm where the interactions between the different operations seem to mitigate any such possibilities. It is, however, possible for the operations of the algorithm, with valid situations that have been crafted directly instead of being a product of the algorithm itself, to produce such non-standalone redundant self-loops. Thus, while it may possible to prove the non-existence through an intricate proof of the interactions between the operations, such a proof gives little worth. As for any minor change to the algorithm, such as all the changes in this thesis, the proof has to be repeated with appropriate changes.

While proving the non-existence of non-standalone self-loops may make it possible to ignore redundant self-loops all together, there is no guarantee that



(a) Labeled graph with an augmenting walk

(b) After execution of the shortcut operation

(c) Removal of redundant self-loop

Figure 3.1: Example of a situation where the shortcut operation can result in the augmenting walk having a redundant self-loop.

Iwata and Yokoi have not relied on the property in other parts of the paper. Given this and the resulting simplifications to the later modifications, we will rather present a procedure to eliminate redundant self-loops in the augmenting walk. This is especially rooted in the fact that the later presented modifications increase the chance of redundant self-loops occurring. In the later introduced procedure it will be assumed that the redundant self-loops can both be standalone and non-standalone.

Before introducing a procedure that removes any redundant self-loops in the augmenting walk, two examples of situations in which redundant self-loops are created will be shown, one for the shortcut operation and one for the partial symmetric difference operation. It can be verified that the augmenting walks of the examples can be created by the search procedure, depending on the order in which the edges are chosen. For the search procedure, it follows directly from the argumentation and proofs of Iwata and Yokoi that the procedure will not result in any redundant self-loops. This, due to how the blossom expansion is performed and the criteria chosen for the validity of self-loops as interior edges in the forest expansion.

In the shortcut operation a new self-loop may be added to the augmenting walk, but only when it is absolutely needed, that is, the self-loop is not added to the augmenting walk unless the symbols on the augmenting walk require it. As such, this self-loop is never redundant. The shortcut operation may still result in the augmenting walk containing redundant self-loops, as the symbols on the walk may change, and thus already existing self-loops may turn from non-redundant to redundant. Figure 3.1 shows an example of such a situation. There, in Figure 3.1a, the augmenting walk has the symbols $strtruvr$. The self-loop (uv) is surrounded by r's, meaning it is non-redundant. After application of the shortcut operation, as seen in Figure 3.1b, the symbols are $srtuvr$. This means that the self-loop has become redundant and can be removed as seen in

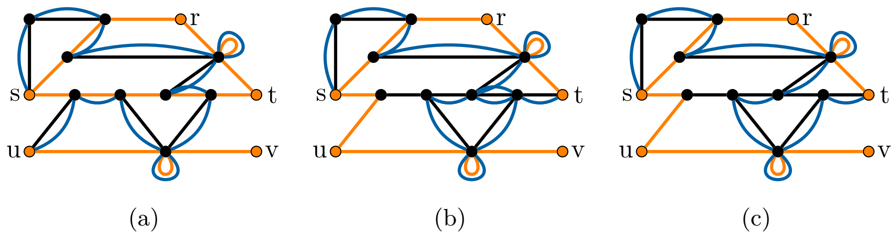


Figure 3.2: Example of a situation where the partial symmetric difference operation results in redundant self-loops in the augmenting walk.

Figure 3.1c. Then, the symbols on the augmenting walk are $srtr$ which suffice (ii).

Since the partial symmetric difference operation will not be run unless there are no shortcuts available, any situations in which the partial symmetric difference operation produce redundant self-loops will necessarily be more complex than for the shortcut operation. An example of this can be seen in Figure 3.2. There, in Figure 3.2a, an augmenting walk containing five P-segments, of which two are non-redundant self-loops, can be seen. We will assume that the symbols on the self-loop on the T-path from r to t are tr when going from u to s in the augmenting walk, as otherwise a shortcut is possible. This is a possible situation, as the order of the symbols on self-loops is determined prior to the search procedure and can take any of the two possible orders. If this is the case, the symbols on the augmenting walk will be $ustuvtstrsrs$. This means that a shortcut operation can not be performed on the two P_j-segments of the T-path going from s to t. As such, Figures 3.2b and 3.2c show the execution of the partial symmetric difference operation on the given augmenting walk. Both of the self-loops in the original augmenting walk are kept in the new augmenting walk. Since the symbols of the new augmenting walk are $tvutrsrs$, the self-loop on the T-path from u to v is non-redundant while the other self-loop is redundant. Removing the redundant self-loop results in the symbols on the augmenting walk becoming $tvusrs$. The remaining self-loop is now also redundant and can be removed.

The example from the partial symmetric difference operation shows an important situation that must be handled when removing redundant self-loops, that is, the augmenting walk may contain more than one redundant self-loop, and removing one redundant self-loop may result in another self-loop becoming redundant. While it seems unlikely that the shortcut operation may result in this occurring in the standard Iwata–Yokoi algorithm, some of the changes suggested later in this thesis, for example one of the ways to handle cycles in the T-paths, may also result in the occurrence of more than one successive self-loop. If there are more than one successive self-loop in the augmenting walk, the shortcut operation may result in one or more of these becoming redundant.

We will now present a procedure, Remove-Redundant-Self-Loops, which through simple iteration of the augmenting walk removes redundant self-loops. Since a self-loop is redundant only when the symbols preceding and succeeding it are not equal, the procedure creates a list of the symbols on the augmenting walk to allow for constant time referencing. The procedure then creates a new augmenting walk through iteration over the old augmenting walk. All edges that are not redundant self-loops are added to the new augmenting walk. A self-loop is in this case classified as redundant if the last symbol in the new augmenting walk is not equal to the next symbol in the old augmenting walk. This, as will be shown later, guarantees that the new walk suffice (ii). It is worth noting that there may still be redundant self-loops in the resulting augmenting walk. This, as for a situation with several consecutive self-loops, with only the last being classified as redundant, the chosen one-pass strategy of the procedure does not allow the changes from the removal of the last self-loop to propagate to the previous self-loops. Thus, a procedure Remove-All-Redundant-Self-Loops is created. This procedure calls the Remove-Redundant-Self-Loops procedure until no redundant self-loops are deleted in a single application.

```
Remove-Redundant-Self-Loops($Q$)
 1  $\gamma = \gamma(Q)$
 2  $p = \gamma[1]$
 3  $p_n = 2$
 4  $deleted$ = FALSE
 5  $Q'$ = empty path
 6  **for** $e$ **in** $Q.edges$
 7        **if** $e.labeled$
 8              $p_n$ += 2
 9        **if** $e.src \neq e.dst$ **or** $p == \gamma[p_n]$
10              **add** $e$ to $Q'$
11              **if** $e.labeled$
12                    $p = e.dst\text{-}symbol$
13        **else** $deleted$ = TRUE
14  **return** $Q'$, $deleted$
```

We will now show and prove the validity and time complexity of the Remove-Redundant-Self-Loops procedure.

Lemma 3.1 Remove-Redundant-Self-Loops *returns an augmenting walk. The resulting augmenting walk may only differ from the input augmenting walk by one or more deleted self-loops.*

Proof: An edge is never removed after being added to Q'. Thus, for the difference between Q and Q' to only be the deletion of self-loops, it must be shown that all normal edges are added to Q' and all edges in Q' are added in their original order. The only time an edge is added to Q' is on line 10. For line 10 to occur, at least one of the conditions on line 9 must be valid. Any non-self-loop edge suffice the first condition, and thus line 10 runs for all non-self-loop edges. Moreover, since the loop of line 6 walks Q from start to finish, the edges in Q' must follow the same order as in Q. Thus, the only difference between Q and Q' may be missing self-loops.

Since only self-loops may be deleted, it follows that Q' is a walk. Additionally, since no new edges are added, the properties (i) and (iii) of augmenting walks must be fulfilled. For an edge, e, if both of the conditions on line 9 are false, then e is a self-loop and the symbol of the next labeled edge in Q is different from the last symbol in Q'. Thus deleting such e does not break property (ii) of an augmenting walk. Especially, even though the check on line 9 checks whether or not each edge can be removed with no regards to the removal of later edges in Q, (ii) must hold since an edge is only removed if it is guaranteed that Q' concatenated with the remainder of Q suffice (ii). ■

Lemma 3.2 REMOVE-REDUNDANT-SELF-LOOPS *either removes at least one redundant self-loop and returns* TRUE *for "deleted" or there are no redundant self-loops in the augmenting walk, the augmenting walk is not change and* FALSE *is returned for "deleted".*

Proof: Assume that Q has at least one redundant self-loop. Then, when e is the first redundant self-loop of Q, neither of the conditions on line 9 may hold. e is of course a self-loop, and since e is a redundant self-loop the symbol preceding e, p, must differ from the symbol succeeding e, $\gamma[p_n]$. Thus, line 10 is never ran and e is not added to the new augmenting walk. Moreover, since none of the conditions on line 9 hold, line 13 must be run and *deleted* will be set to TRUE.

Assume the opposite, namely that Q has no redundant self-loops. Then, at least one of the conditions on line 9 must hold for all edges e, as either e is not a self-loop, or e is a non-redundant self-loop and thus the symbol before e, p, is equal to the symbol after e, $\gamma[p_n]$. Thus, line 13 will never be run and *deleted* will always remain FALSE. It follows directly from Lemma 3.1 that the augmenting walk does not change. ■

Lemma 3.3 REMOVE-REDUNDANT-SELF-LOOPS *has a time complexity of* $\mathcal{O}(Q)$

Proof: Line 1 is a simple iteration over the augmenting walk and can be performed in $\mathcal{O}(Q)$. Lines 2 through 5 and 14 can be performed in constant time. The loop on line 6 is an iteration over the augmenting walk and thus consists of

$\mathcal{O}(Q)$ iterations. Each of the operations on lines 7 through 13 can be performed in constant time. Thus, the loop of line 6 has a time complexity of $\mathcal{O}(Q)$. It follows that the whole procedure has a time complexity of $\mathcal{O}(Q)$. ■

The Remove-All-Redundant-Self-Loops procedure will now be presented with proofs for the validity and time complexity. As previously mentioned, this procedure simply applies the Remove-Redundant-Self-Loops procedure until no self-loops have been deleted in a single application.

```
Remove-All-Redundant-Self-Loops($Q$)
1  *deleted* = TRUE
2  **while** *deleted*
3        $Q$, *deleted* = Remove-Redundant-Self-Loops($Q$)
4  **return** $Q$
```

Lemma 3.4 *Let Q and Q' be the augmenting walk, respectively, before and after the application of the* Remove-All-Redundant-Self-Loops *procedure. Then, the number of applications performed of the* Remove-Redundant-Self-Loops *procedure is smaller than or equal to $\mu(Q) - \mu(Q') + 2$.*

Proof: Assume that there are no redundant self-loops in the augmenting walk, then there is only one application of the procedure, and there is no change in the number of P-segments. Since the number of applications is less than the constant 2, this case holds.

Now assume that there is at least one redundant self-loop in the augmenting walk. Then, there are three possible classifications for the self-loops that will be removed from the augmenting walk. The self-loops may all be standalone, a mix of standalone and non-standalone or non-standalone. In all of these cases, Lemma 3.2 guarantees that the number of applications of the Remove-Redundant-Self-Loops procedure will at most be an equal amount of times to the number of self-loops removed.

For the case of only standalone self-loops, an entire P-segment is removed for each removed self-loop. Thus, the number of applications can not be higher than $\mu_P(Q) - \mu_P(Q') + 1$, with the 1 being a result of having to perform an extra application in the end to check that there are no remaining redundant self-loops.

For the case of only non-standalone self-loops, one can see that (1) removing a non-standalone self-loop does not change whether or not the previous or next self-loop is redundant and (2) all non-standalone self-loops are redundant and can be removed in a single application of the procedure. For (1) it suffices to see that a non-standalone self-loop is either followed by or preceded by an edge from the same T-path with the same symbols in the same order as the self-loop. This follows directly from the definition of an augmenting walk. Thus, removing the non-standalone self-loop does not change the symbols before or after any other self-loop. For (2) one can see that due to the edge preceding or succeeding the self-loop having the same symbols as the self-loop, the symbol before and after a non-standalone self-loop can never be the same. Thus, they will always be removed by the procedure. A total of 2 applications are thus needed. This is equal to the constant above. Especially, the number of P-segments does not change when removing a non-standalone self-loop, since, by definition, at least one edge remains in each of the P-segments.

For the last case, removing a redundant self-loop reduces the number of P-segments by either (a) 2, (b) 1 or (c) 0. Case (a) occurs when a standalone self-loop is removed and there is a self-loop directly preceding or succeeding it that goes from being a standalone self-loop to a non-standalone self-loop. Thus, the P-segment of the deleted self-loop is removed and the P-segment of the other self-loop is combined with another P-segment and thus a reduction by 2 in the number of P-segments is achieved. Case (b) may occur if a standalone self-loop is deleted and case (a) is not valid. Then, a P-segment is simply removed from the augmenting walk. Case (c) occurs when a non-standalone self-loop is removed, as then only an edge from a P-segment is removed, not an entire P-segment. Using this, it follows that the number of P-segments must be reduced in each iteration, unless only non-standalone self-loops are removed. Using (1) from above, this can only happen if there are no more standalone self-loops that will be removed. By (2) this can only happen once, as all such self-loops are removed in the same application. Thus, for all but a maximum of two applications of the procedure, the number of P-segments is reduced by at least one and it follows that the number of applications is at most this reduction in P-segments plus two. ∎

Theorem 3.5 *Applying the* Remove-All-Redundant-Self-Loops *procedure at the end of each iteration of the augmentation procedure does not increase the time complexity of the algorithm, and results in a valid augmenting walk with no redundant self-loops.*

Proof: By Lemma 3.1 it follows that the augmenting walk returned from the Remove-Redundant-Self-Loops procedure is valid. Since this is the only change made to the augmenting walk in the procedure, the resulting augmenting walk must be valid. By Lemma 3.2 and the fact that the procedure is applied until *deleted* is **false**, there can not be any redundant self-loops in the resultant augmenting walk, as otherwise *deleted* would be **true**.

As for the time complexity of the augmentation procedure, the augmentation procedure consists of a number of iterations where each iteration must be performed in $\mathcal{O}(E)$ operations. The number of iterations is bound by the number of P-segments in the augmenting walk. By Lemma 3.4, it follows that there is a constant number of applications of the Remove-Redundant-Self-Loops procedure, in addition to a number of applications bound by the reduction in the number of P-segments. By Lemma 3.3, the time complexity of each application of the procedure is $\mathcal{O}(Q)$. Since Q is bound by $\mathcal{O}(E)$, the constant number of applications disappear in the time complexity of one iteration of the augmentation procedure. For the non-constant number of applications, the maximum number of iterations of the augmentation procedure decreases by an equal or larger amount. Thus, since the time complexity of one application of the Remove-Redundant-Self-Loops procedure is the same as for one iteration of the augmentation procedure, the time complexity does not increase. ■

We note that the improvements made to the algorithm in the preliminary project do not either see an increase in time complexity. While each iteration of the augmentation procedure has a time complexity of $\mathcal{O}(V)$ in the improvement, the augmenting walk is also bound by $\mathcal{O}(V)$. Thus, the Remove-Redundant-Self-Loops procedure is bound by $\mathcal{O}(V)$, and the same logic as in the proof of Theorem 3.5 can be employed.

Chapter 4

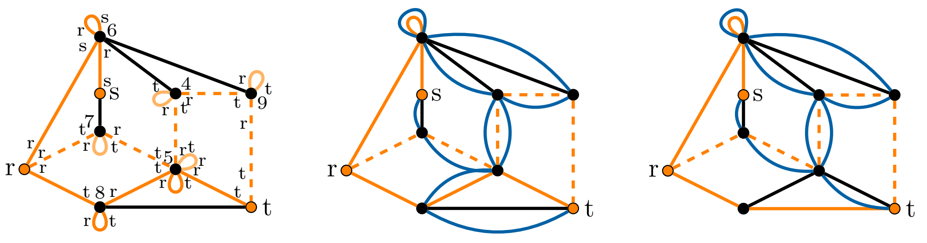
The Shortcut Operation

In their paper, Iwata and Yokoi outlined a procedure for finding possible shortcuts in $\mathcal{O}(E)$ time complexity. We have found that some parts of the procedure may find a shortcut of a given type, even when the pair of P_j-segments does not constitute to a shortcut of this type. In fact, the procedure seems to find shortcuts in the augmenting walk where there are no possible shortcuts. This chapter will show examples of situations where the procedure finds these invalid shortcuts. The shortcut-finding procedure will be modified so as to mitigate this problem. Additionally, Section 4.3 introduces a non-required modification to the shortcut-finding procedure. This modification is meantto simplify the handling of self-loops in implementations of the algorithm.

4.1 Sample situations

An example of a problematic situation with the shortcut operation can be seen in Figure 4.1, where a situation with three T-paths and an augmenting walk is depicted. We will denote the T-paths described by $s-6-r$, $t-5-8-r$ and $r-7-5-4-9-t$ as P_1, P_2 and P_3, respectively. Note that the augmenting walk is described by $s-7-5-4-9-6-6-4-5-8-t$, where the $7-5-4-9$ part is a P_3-segment, the $6-6$ part is a P_1-segment, the $4-5$ part is a P_3-segment and the $5-8$ part is a P_2-segment. One can also, using Figure 4.1, easily verify that the three T-paths and the augmenting walk are all valid, with the augmenting walk having the following symbols $srtrtrtsrtrtrt$.

When applying the augmentation procedure to the given augmenting walk, the first P-segment to be considered is the P_3-segment $7-5-4-9$. It can be checked that no shortcut operation can be performed here, as that would require a shortcut operation to be performed with the only other P_3-segment, $4-5$.



(a) Labeled graph (b) Step 4 example (c) Step 3 example

Figure 4.1: An example of a situation where the algorithm for finding shortcuts in the Iwata-Yokoi algorithm fails.

Performing a shortcut with the P_3-segments would result in changing the part of the augmenting walk, $7-5-4-9-6-6-4-5$, to the part of P_3 going from vertex 7 to vertex 5. That is, replacing it with the edge $7-5$. This would result in the invalid augmenting walk $s-7-5-8-t$, which has two consecutive t's in its symbols, $srttrt$. As such, it is not possible to perform the shortcut operation on the augmenting walk.

Given this, one would expect the 5 step shortcut-finding procedure to find no possible shortcuts in the augmenting walk. As will be seen shortly, this is not the case. Going through step 0 of the procedure, the following values are set for p_{st} and p_{ts}.

$$p_{st}(7-5) = p_{st}(5-4) = p_{st}(4-9) = 1 \tag{4.1}$$
$$p_{ts}(5-4) = 2 \tag{4.2}$$

For all other edges p_{st} and p_{ts} are both not set. Additionally, for the two P_j-segments the following values are found for *enter* and *leave*.

$$enter(1) = 1 \qquad leave(1) = 1 \tag{4.3}$$
$$enter(2) \neq 1 \qquad leave(2) \neq 1 \tag{4.4}$$

Going through steps 1 through 3 no possible shortcuts are found, as expected. However, in step 4, following the exact process outlined, a possible shortcut is returned. Table 4.1 show the process in step 4 with all values and computations at each edge, e, (safely ignoring self-loops) as we walk P_3 from r to t as outlined in the step. When reaching edge $5-4$, p^*_{ts} is set to 2 and as this edge also has $p_{st} = 1 < 2 = p_{ts}*$ and enter(1) = 1, a possible shortcut has been found. However, Iwata and Yokoi have not specified whether p^*_{ts} should be updated before or after the checks for possible shortcuts. For the former, it will result in an invalid

e	p^*_{ts}	$p_{st}(e)$	enter$(p_{st}(e)) = 1$	$p_{st}(e) < p_{ts}*$
$c-7$	-	-	-	-
$7-5$	-	1	YES	-
$5-4$	2	1	YES	?
$4-9$	2	1	YES	YES
$9-b$	2	-	-	-

Table 4.1: Table of data in step 4 in the shortcut-finding procedure

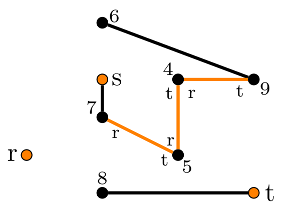
shortcut being found for this edge. For the latter, one can show that in other situations the procedure may miss possible shortcuts. Nonetheless, for the next edge, $4-9$, the same conditions hold, with $p_{st} = 1 < 2 = p^*_{ts}$ and enter$(1) = 1$. Thus, no matter the order in which the check was meant to be performed, the procedure finds an invalid shortcut with the two P_3-segments. As already seen, this shortcut returned by the shortcut-finding procedure results in an invalid augmenting walk.

While the above example only shows a fault in step 4 when walking a P_j-segment from *s*- to *t*-side, it is trivial to show that the procedure inhibits the same fault for the other part of step 4, that is, walking from *t*- to *s*-side with the roles of p_{ts} and p_{st} changed. In fact, if we considered P_3 in reverse order, $t-9-4-5-7-r$, with the other T-paths and the augmenting walk as is, the invalid shortcut would be found in this part of step 4 instead.

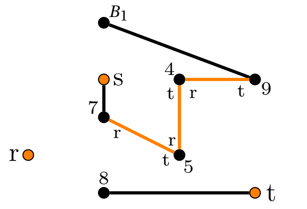
Notice that if the symbols on the augmenting walk allowed the shortcut to be valid, the found invalid shortcut would actually be of type 3. This, as the initial vertex of the *st*-directed P_j-segment appears on the *s*-side of the end vertex of the *ts*-directed P_j-segment in the T-path. Given that step 4 is only supposed to find shortcuts of type 4, this is another point that has to be considered in the modifications to the shortcut-finding procedure.

In a similar fashion to the one laid out in detail above, it is possible to construct a case in which step 3 of the shortcut-finding procedure finds an invalid shortcut due to a similar weakness. Invalid shortcuts found by step 3 would in the same way as above be of another type, namely type 4, if the symbols on the augmenting walk allowed the shortcut to be valid. It seems like steps 3 and 4 exhibits the same weakness, as the checks for shortcuts only considers if an edge belongs to the given P_j-segment, rather than where in the P_j-segment the edge appears. Since the shortcut types are defined based on where the P_j-segments start and end in the T-path, this is not enough for the cases where the two P_j-segments partially overlap.

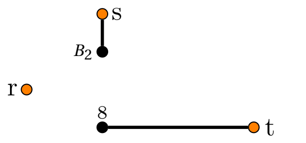
An example of a situation in which step 3 of the procedure finds an invalid shortcut can be seen in Figure 4.1c. This situation is similar to the one earlier discussed for step 4 with P_2 changed to $t-8-r$ and the augmenting walk being $t-5-4-6-6-9-4-5-7-s$. It can again be checked that the given augmenting



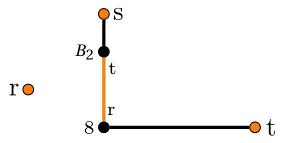
(a) Frontier edge exploration



(b) Blossom B_1 from using the self-loop of vertex 6 in P_1. Consists of vertex 6



(c) Blossom B_2 from the edge $4-6$. Consists of the vertices B_1,$9,4,5,7$



(d) The edge $B_2(5)-8$ produces a compacted augmenting walk

Figure 4.2: A possible forest expansion in the search procedure

walk is valid, having the symbols $trtsrtrtrtrs$, and that no possible shortcuts exist for the P_3-segments. However, going through step 3 a shortcut is returned either on the edge $5-4$ or the edge $4-9$, again depending on the order in which p^*_{st} is updated and the checks for shortcuts are performed.

As with the previously discussed non-standalone redundant self-loops, it is interesting to consider if these situations may actually occur in the Iwata–Yokoi algorithm before deciding on handling them. If it can be shown that no such cases can occur, then it is not necessary to modify the algorithm to mitigate the shown problems. This is not the case for the shortcut-finding procedure, where for some labeled graphs the search procedure can only find augmenting walks for which either step 3 or 4 returns an invalid shortcut. The situation shown in Figure 4.1 form one such example.

Figure 4.2 shows one possible forest expansion for the labeled graph shown in Figure 4.1a. First, after all the frontier edges have been exhausted in Figure 4.2a, the only interior edge, the self-loop of P_1 on vertex 6, is used to create blossom B_1 in Figure 4.2b. This allows for the edge $4-6$ to become a valid interior edge, which is used in Figure 4.2c, creating a blossom B_2 consisting of all but one non-terminal. Finally, this results in the edge $5-8$ becoming a valid interior edge. In Figure 4.2d, this edge is used to produce a compacted augmenting walk. While Figure 4.2 does not show the expansion of blossoms B_1 and B_2, it can be shown that the augmenting walk must walk the long way around the blossoms, and thus produce the augmenting walk shown in Figure 4.1b.

The example discussed above shows one possible forest expansion in the labeled graph. It is possible to show that there are a total of six possible distinct forests that can be created depending on which edges are used as frontier edges. The forests are a result of the choice between edges $7-5$ and $8-5$, as well as which two of three edges connecting vertices 4, 6 and 9 are used. All of these forests inhibit the same property, as was mentioned for the expansion showed in Figure 4.2, where given the selection of the frontier edges only one valid interior edge can be found at each step of the search procedure. Thus, a total of six different expansions can be found, producing four distinct augmenting walks. These four augmenting walks form pairs of reversed walks. Two of these, the one shown for step 4 and its reverse, always result in invalid shortcuts being found, while the remaining two only results in invalid shortcuts if the update of p^*_{st} and p^*_{ts} happens before the check for a shortcut.

As seen, it is non-trivial to design a way to mitigate the occurrence of situations in which an invalid shortcut operation may be found. If the updating of p^*_{ts} (p^*_{st}) occurs before the checks in steps 3 and 4, then for the labeled graph discussed above there is no way to mitigate this problem locally in the search procedure. This would mean that such a mitigation would need to be on a more global level in the algorithm.

4.2 Changing the Shortcut-Finding Procedure

We will now look at what changes need to be made to mitigate these issues in the shortcut-finding procedure. The changes aim to modify the shortcut operation the least. This, as there does not seem to be any better strategies than the ones already employed by Iwata and Yokoi, and there is no reason to change the parts that are already working unless they can be simplified or optimized. For all the changes discussed below, only the changes made going from s- to t-side are given. The changes going from t- to s-side are exactly the same, only with the roles of st and ts interchanged.

As was previously mentioned, both of the problematic steps in the procedure seemed to find invalid shortcuts only when the invalid shortcut was of another type than what the step was supposed to find. That is, it seems like the invalid shortcuts are a result of the steps not considering what part of the P_j-segment a given edge belonged to when comparing p^*_{st} and p^*_{ts} to the equivalent values of the given edge. Shortcuts of type 3 and 4 require a certain order of the appearance of the initial vertex of the first P_j-segment and the last vertex of the second P_j-segment in the T-path. For type 3 and 4, this order is roughly opposite and thus step 3 finding type 4 shortcuts and step 4 finding type 3 shortcuts indicate that this order has to be checked.

The first change that will be made, is in step 0. In this step, there is already a process for setting $enter(x)$ and $leave(x)$ based on the preceding and succeeding symbol on the augmenting walk for the x-th P_j-segment. In a similar fashion, set $first(x)$ and $last(x)$ to respectively the first and last edge in the x-th P_j-segment. This change introduces a data structure that makes it possible to later find the start and end of a P_j-segment. It is possible to do this without the help of this data structure, , depending on which direction the T-path is walked, the first edge of a P_j-segment seen is either the first or the last edge of the P_j-segment. However, since step 0 already builds data structures that contain content for each P_j-segment, having $first$ and $last$ does not result in any change in time complexity and only adds a slight memory and computational cost. Especially, it somewhat simplifies the later checks.

To mitigate the issues in step 3, the following changes can be made to the step. First, make sure that the update of p^*_{st} is performed before the check for a shortcut. Second, add the check $last(p_{ts}(e')) = e'$, that is, for a given edge e' the check becomes $p_{ts}(e') > p^*_{st}$, $leave(p_{ts}(e')) = 1$ and $last(p_{ts}(e')) = e'$. The reason for the specification of the order in which the update and check should be performed, is that if the same edge is used by both an *st*-directed and a *ts*-directed P_j-segment, then if the edge is the first in the *st*-directed P_j-segment and the last in the *ts*-directed P_j-segment, the start and end of the two P_j-segments overlap and thus there can be a shortcut of type 3. To be pedantic, one could update p^*_{st} only when an edge, e, with $first(p_{st}(e)) = e$ is visited. However, going from *s*- to *t*-side the first edge found in an *st*-directed P_j-segment will always have this property, making the check redundant.

Theorem 4.1 *The updated step 3 in the shortcut-finding procedure finds only shortcuts of type 3 and finds all such shortcuts.*

Proof: This proof will only show the part of step 3 where the iteration is from *s*- to *t*-side. The proof in the other direction is exactly the same except for all references to s and t being interchanged.

First, assume that a shortcut was found by step 3, it must then be shown that the shortcut is of type 3, that is, the three conditions for a shortcut of type 3 are sufficed. Thus, the two returned P_j-segments must be directed in opposite directions and it must hold that (1) the *st*-directed P_j-segment appears on the augmenting walk before the *ts*-directed one, (2) the first vertex of the *st*-directed P_j-segment appears *s*-side of the last vertex in the *ts*-directed one or they are equal, and (3) the first symbol after the *ts*-directed P_j-segment is not t.

Since the step returns one P_j-segment for which p_{st} is set and one for which p_{ts} is set, the two P_j-segments must be oppositely directed. Given this (1) must hold, as if a shortcut was found then $p_{ts}(e') > p^*_{st}$, so the *ts*-directed P_j-segment must by the definition of the values for p_{ts} and p^*_{st} appear after the *st*-directed one in the augmenting walk. Similarly, $leave(p_{ts}(e'))$ is set to 1, so the next symbol after the *ts*-directed P_j-segment is not t, and (3) holds.

The first vertex of the *st*-directed P_j-segment is the adjacent vertex on the *s*-side of the first edge in the P_j-segment. Similarly, for the *ts*-directed P_j-segment, the last vertex of the P_j-segment is the adjacent vertex on the *s*-side of the last edge in the P_j-segment. Thus, for (2) to hold, the first edge of the *st*-directed P_j-segment must either appear on the *s*-side of the last edge in the *ts*-directed P_j-segment, or the two edges must be equal. Since there are no cycles in a T-path, the first edge seen of the *st*-directed P_j-segment and the *ts*-directed P_j-segment, must, respectively, be the first and last edge of the P_j-segments. Thus, since p^*_{st} is set to the value of the *st*-directed P_j-segment of the given edge e' before checking the edge, and $p_{ts}(e') > p^*_{st}$, the first edge of the *st*-directed P_j-segment must either be e' or appear *s*-side of e' and thus (2) hold. Thus, for any shortcut returned, all the properties for a shortcut of type 3 hold, and it must be a valid shortcut of type 3.

For the opposite, assume that there exists at least one shortcut of type 3 in the augmenting walk. Then, there exists a pair of P_j-segments, where one is *st*-directed and the other is *ts*-directed, the *st*-directed P_j-segment is the one that appears closest to the *ts*-directed one on its *s*-side, and (1), (2) and (3) from above hold. Since (2) holds, then, following the same logic as above, the first edge seen of the *st*-directed P_j-segment must appear before or at the same time as the first edge seen of the *ts*-directed P_j-segment when walking from *s*- to *t*-side. Thus, p^*_{st} must be updated to the value of the *st*-directed P_j-segment before any of the edges of the *ts*-directed P_j-segment are checked. Thus, by (2) $p_{ts}(e') > p^*_{st}$ for all edges in the *ts*-directed P_j-segment and due to (3) it must also hold for each edge in the P_j-segment that $leave(p_{ts}(e')) = 1$. Since $last(p_{ts}(e')) = e'$ holds for exactly one edge in the P_j-segment and all the other required checks hold for this edge, a shortcut will be returned. Thus, if there exist any shortcuts of type 3, one will be found by step 3. ■

To mitigate the issues in step 4, a similar approach to the one used for step 3 can be taken. As for p^*_{st} in step 3, p^*_{ts} should be updated before the check is performed. For the check, the additional condition of $first(p_{st}(e')) = e'$ is added. Resulting in the entire check becoming $p_{st}(e') < p^*_{ts}$, $enter(p_{st}(e') = 1$ and $first(p_{st}(e')) = e'$. Again, to be pedantic one could check $leave(p_{ts}(e)) = e$ before updating p^*_{st}, but this does not change anything for the same reasons as in step 3.

Theorem 4.2 *The updated step 4 in the shortcut procedure finds only shortcuts of type 4 and finds all such shortcuts.*

Proof: As in the proof of Theorem 4.1, this proof will only show the part of step 4 where the iteration is from *s*- to *t*-side. For the other direction, simply interchange all references to s and t.

First, assume that a shortcut was found by step 4, it must then be shown that the shortcut is of type 4, that is, the three conditions for a shortcut of type 4 must be sufficed. Thus, the two returned P_j-segments must be oppositely directed and it must hold that (1) the *st*-directed P_j-segment appears on the augmenting walk before the *ts*-directed one, (2) the last vertex of the *ts*-directed P_j-segment is either equal to or appear *s*-side of the first vertex of the *st*-directed P_j-segment and (3) the preceding symbol of the *st*-directed P_j-segment is not t.

Since step 4 returns one P_j-segment with p_{st} set and one with p_{ts} set, the two P_j-segments must be oppositely directed. Since $p_{st}(e') < p^*_{ts}$ holds for all returned shortcuts, (1) must hold. Moreover, since $enter(p_{st}(e')) = 1$ for all returned shortcuts, (3) must hold.

By the same logic as in the proof of Theorem 4.1, the first vertex of the *st*-directed P_j-segment lies directly *s*-side of the first edge in the P_j-segment, and this is the first edge of the P_j-segment reached when walking from *s*- to *t*-side. Similarly, the last vertex in the *ts*-directed P_j-segment must appear directly *s*-side of the last edge in the P_j-segment, and this is the first edge of the P_j-segment reached when walking from *s*- to *t*-side. When reaching the edge, e', for which the given shortcut was found, at least one edge from the *ts*-directed P_j-segment has been visited. Thus, by the above logic, we must already have passed the last vertex in the *ts*-directed P_j-segment or it is the *s*-side vertex of e'. Then, since $first(p_{st}(e')) = e'$, the first vertex of the *st*-directed P_j-segment must be either the same as the last vertex of the *ts*-directed P_j-segment or *t*-side of this vertex. Thus, (2) must also hold, and the given shortcut is of type 4.

For the opposite, assume that there exists at least one shortcut of type 4. Then, there exists a pair of P_j-segments where one is *st*-directed and the other *ts*-directed, and the *ts*-directed P_j-segment is the one that appears closest to the *st*-directed one on its *t*-side. Additionally, (1), (2) and (3) hold for this pair of P_j-segments. Let e' be the first edge in the *st*-directed P_j-segment, then, by (2), e' is either the last edge in the *ts*-directed P_j-segment or the last edge appears *s*-side of e'. Thus p^*_{ts} will be set to the *ts*-directed P_j-segment before the check for a shortcut on e'. Due to (1), it then follows that $p_{st}(e') < p^*_{ts}$ is true, and, due to (3), it must be true that $enter(p_{st}(e')) = 1$. Especially, by the definition of e' it must be true that $first(p_{st}(e')) = e'$. Thus, when e' is reached in step 4, the given shortcut is found, and step 4 must find a shortcut of type 4 if at least one such shortcut exists. ■

4.3 Shortcuts and Self-loops

In this section we will introduce changes to the shortcut-finding operation in regards to self-loops. These changes are not needed, and the shortcut-finding operation should work perfectly fine with only the changes described in Section 4.2. The changes described here are not meant to handle any edge case, but rather meant to simplify and optimize the handling of self-loops in the entire algorithm.

As has been described earlier, P-segments can be divided into two categories, that is, P-segments that are equal to a sub-path of a T-path and P-segments that are a single self-loop on a vertex of a T-path. The self-loops are special edges in the labeled graph that must be created and removed whenever edges are added to or removed from a T-path. Since all possible changes to a T-path require some form of iteration over the T-path, keeping and updating references to these self-loops in the T-path does not result in an increase in time complexity. As was touched on in the preliminary project, updating the labeled graph is, on the other hand, hard since it requires a data structure that supports constant time access and deletion of arbitrary elements (edges). As such, the self-loops, at the very least; have to be reconstructed in the labeled graph between each iteration of the algorithm. Or, as is the solution of Iwata and Yokoi, the whole labeled graph is reconstructed between each iteration of the algorithm. In either case, the need for keeping and updating references to self-loops of a T-path depends on the need for such information during the augmentation procedure. This, as the search procedure does not change the T-paths and is only concerned with the edges in the labeled graph, not the T-paths themselves. Thus, the search procedure works properly with the initial state from the reconstructed labeled graph, not requiring a T-path to know about its self-loops. Three situations can be identified, in the augmentation procedure, in which a T-path may need to know about its self-loops.

The first case is in the shortcut operation where a self-loop on a given vertex of the selected T-path may need to be inserted into the augmenting walk. In this case there is no concern about if the self-loop is already used in the augmenting walk, since duplicate use of the self-loop in this situation can not occur as a result of the algorithm's design. Based on this, a new self-loop of the T-path on the specific vertex could be created, instead of having to find the self-loop in the T-path. This, as the previously created self-loop is no longer in use.

The second case is in the partial symmetric difference operation where all self-loops of the T-path that appear in the augmenting walk on vertices visited by the now removed part of the T-path must be removed from the augmenting walk. Here, one could work around the need for a list of the self-loops removed by keeping track of the vertices that were visited in the removed part of the T-

path. Since a T-path does not have any cycles, any self-loops in the augmenting walk on any of those visited vertices that belonged to the given T-path can be removed. This operation is within the time complexity of the algorithm, since it is bound by the number of vertices, the length of the T-path and the length of the augmenting walk, all of which are within the time complexity of the partial symmetric difference operation, $\mathcal{O}(E)$.

This leaves the third and hardest case, that is, the need for the knowledge of the self-loops in the shortcut-finding procedure. Here, each iteration of the T-path requires a combined iteration of the edges and self-loops of the T-path so that each edge and self-loop is considered in the same order they appear in on the T-path. As such, this operation requires all the cases above to use the correct self-loops, as well as any changes to the T-path to correctly update the references to the self-loops. Here, we will suggest an alternative approach to self-loops in the shortcut-finding operation so that it does not depend on a T-path having knowledge about its own self-loops, and thus removing the need to keep and update references to the self-loops in the T-paths. Do note that the suggested changes will build on the assumption that each self-loop knows about which T-path it belongs to. This assumptions is anyways required in the standard Iwata–Yokoi algorithm.

To be able to remove the need for knowledge about a T-paths self-loops in the shortcut-finding procedure, certain properties regarding self-loops and shortcuts must be shown. The following theorem is essential in this regard.

Theorem 4.3 *Assume an augmenting walk has no redundant self-loops. Then, if there exists a pair of two self-loops in the augmenting walk from the same T-path, they form a valid shortcut.*

Proof: If a self-loop is not redundant in the augmenting walk, then the symbols on the self-loop are different from those directly preceding and succeeding the self-loop in the augmenting walk. This, as the self-loop is non-redundant only when the directly preceding and succeeding symbols are equal. Replacing the part of the augmenting walk between the first self-loop and the second self-loop with the part of the T-path between the vertices of these self-loops can therefore not break property (ii) since the symbols preceding and succeeding this new P_j-segment in the augmenting walk are different from the symbols of the T-path. Since the shortcut operation guarantees for (i) and (iii) to hold, a shortcut must exist as it is possible to perform the shortcut operation and have a valid augmenting walk. ■

Using the results of Theorem 4.3, one can design a simple procedure that finds any shortcut between pairs of self-loops. The procedure FIND-SELF-LOOPS-SHORTCUT, below, is an example of such a procedure. There the augmenting walk is simply iterated, keeping track of the last seen self-loop of the given T-path. Then, either another self-loop on the T-path is seen and a valid shortcut is returned or there are either zero or one self-loops of the T-path in the augmenting walk. If there is only one such self-loop in the augmenting walk, it is returned.

```
Find-Self-Loops-Shortcut($Q, P_j$)
l = NIL
for e in Q.edges
    if $e \in P_j$ and $e.src == e.dst$
        if $l ==$ NIL
            $l = e$
        else return $l$, $e$
return $l$
```

Corollary 4.4 *For an augmenting walk with no redundant self-loops,* Find-Self-Loops-Shortcut *either finds a valid shortcut between two self-loops of the given T-path or there are less than two self-loops from the T-path in the augmenting walk. The procedure does not increase the time complexity of the shortcut operation.*

Since the Find-Self-Loops-Shortcut procedure requires a single iteration over the augmenting walk, the procedure does not increase the time complexity of the shortcut operation. It does, however, add additional computational costs to the shortcut operation. Thus, we note that the Find-Self-Loops-Shortcut procedure can be combined with step 0 in the shortcut operation to reduce the number of iterations required. This, as the procedure does not require any of the data that is initialized in step 0. The procedure will still add a minuscule number of operations, but this increase should be unnoticeable.

Given that Find-Self-Loops-Shortcut or a similar approach in step 0 has been run and did not return a shortcut, the procedure returned either a single self-loop or nothing at all. In the latter case, there are no more self-loops of the given T-path in the augmenting walk and thus the remaining part of the shortcut-finding procedure does not need to consider self-loops of the T-path. For the case where a single self-loop of the T-path remains in the augmenting walk, there may exist possible shortcuts between the self-loop and other P_j-segments. As such, the self-loop must be considered in the remaining part of the shortcut-finding procedure. There are two possible ways to handle this self-loop without requiring the T-path to know about its self-loops. Both exploit the fact that since the T-path is a path, and thus does not contain any cycles, iterating over the T-path the self-loop can be inserted when its vertex is reached.

The first option is to design an additional procedure to be run between step 0 and 1. This procedure walks the T-path from s- to t-side, checking whether or not the seen P_j-segments can form shortcuts with the self-loop. For such a procedure, one could exploit the non-redundant property of the self-loop to make the procedure require only a single iteration of the T-path while at the same time detecting any possible shortcuts between the self-loop and the other P_j-segments. This would require the procedure to perform a combination of the checks in steps 1 through 4 of the standard shortcut-finding procedure with some simplifications, that is, in the same way Theorem 4.3 was proven the checks for *enter* and *leave* of the self-loop P_j-segment can be ignored. Thus, the procedure is simply a combination of the checks provided by Iwata and Yokoi, and presenting such a procedure does not provide anything new to the discussion and will be forgone.

The other option is to simply perform steps 1 through 4 as normal, inserting the self-loop at the appropriate place in the T-path and treating it as Iwata and Yokoi do in their original shortcut-finding procedure. This approach can be thought of as simply performing the remaining algorithm as normal, skipping all self-loops that are not used in the augmenting walk when the T-path is iterated.

The previously described way of handling self-loops in the shortcut-finding procedure may require additional iterations over the augmenting walk and/or the T-path. While these iterations do not increase the time complexity, as the procedure already iterates both the augmenting walk and the T-path, the point of these modifications was to simplify the handling of self-loops in the algorithm as to reduce the need for updates, that is, as a possible way to make the algorithm more efficient. Adding more operations one place in the algorithm to reduce the number of operations in another place does not necessarily result in a more efficient algorithm. For the modifications above, it can show that they result in a reduction in the number of edges considered in the shortcut-finding procedure. They should thus, in theory, result in fewer operations even just inside the procedure. What is important here is that for a T-path the number of self-loops is one less than the number of standard edges in the T-path. Thus, removing the self-loops from the T-path should result in roughly half as many operations needed per iteration of the T-path. Since steps 1 through 4 total eight iterations over the T-path, the modifications of the shortcut-finding procedure should result in a reduction in the number of operations if the modifications require less than four iterations over the T-path and its self-loops. This, as the number of edges considered in steps 1 through 4 now equal roughly four iterations of the T-path with its self-loops. The FIND-SELF-LOOPS-SHORTCUT procedure, when incorporated into step 0, should at most require as many operations as one iteration over the T-path, as it consists of a simple check each time an edge from the T-path is found. Additionally, creating the new T-path for iteration, including the single self-loop, should at most require one simple iteration over the T-path. Thus, it seems like the modifications should result in fewer computations even just inside the procedure.

Chapter 5

The Partial Symmetric Difference Operation

In their paper Iwata and Yokoi prove Lemma 3.4. The Lemma states that the partial symmetric difference operation results in a valid set of T-paths and a valid augmenting walk in the new labeled graph. Their multi-page proof focuses on the lack of consecutive symbols in the new walk, as is a required property of an augmenting walk. It does, however, not give a decisive proof for property (iii), that is, the property that a labeled edge may be used twice in an augmenting walk, but only once in each direction. It seems natural that this should hold for the partial symmetric difference operation, as first new edges are added to the augmenting walk that are otherwise only used once in the augmenting walk, before the uncrossing operation removes any duplicate usage of these edges from the augmenting walk. However, in some specific cases it is possible, due to the uncrossing operation changing the direction of subsequences of the augmenting walk, for a labeled edge of another T-path to be used twice in the same direction. This results in the modified walk not being an augmenting walk. This may occur due to a labeled edge being used twice in the augmenting walk prior to the partial symmetric difference operation, with the uncrossing operation only flipping one of occurrence of the labeled edge. The shortcut operation guarantees that this may not occur for labeled edges from the T-path that the partial symmetric difference operation is performed against. Otherwise, there would be an st-directed P_j-segment on the t-side of the P_j-segment removed by the partial symmetric difference operation and the shortcut operation could be performed on these two P_j-segments.

An example of the discussed problem can be seen in Figure 5.1. The figure contains a graph with four valid T-paths and an augmenting walk. While the labels for the edges are not shown, one can verify that the augmenting walk in Figure

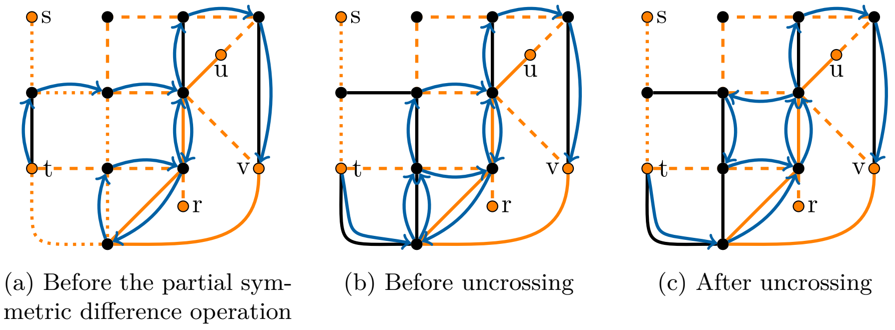
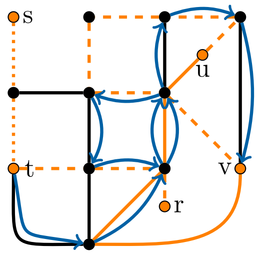


Figure 5.1: Example situation where the partial symmetric difference operation produces an invalid augmenting walk.

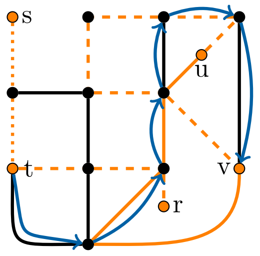
5.1a is valid. The symbols on the augmenting walk are $tstuvuvuvtstrvuvuv$, and all edges, except one labeled edge, are used once. The labeled edge used twice is used once in each direction. The figure shows the execution of a standard partial symmetric difference operation. First, the edges of the augmenting walk and the T-path are interchanged. The state after this part of the operation is shown in Figure 5.1b. At this stage of the operation there is never a guarantee that the walk is an augmenting walk. This is the reason for the uncrossing operation. Thus, the uncrossing operation is applied to the walk. The resulting walk is shown in Figure 5.1c. This walk suffices property (ii) of an augmenting walk, as the symbols on the walk are $tvuvuvutrvuvuv$. However, one of the labeled edges is used twice in the same direction, meaning that (iii) does not hold.

It is already known that the partial symmetric difference operation does not work properly for all augmenting walks, which is why the shortcut operation is used in the Iwata–Yokoi algorithm. As such, for the situation of Figure 5.1 to be a real problem in the Iwata–Yokoi algorithm, it must be shown that there is no possible shortcut operation for the T-path of the first P-segment. There are two P_j-segments in the augmenting walk. The first is an st-directed one and the second is a ts-directed one. Since the ts-directed one appears on the t-side of the st-directed one, any shortcut must be of type 3. However, type 3 shortcuts require the succeeding symbol on the augmenting walk of the ts-directed P_j-segment to not be t. In the given augmenting walk it is a t and thus there can not be a shortcut of type 3. This means that no shortcuts exist for this T-path, and, as such, this is a valid situation for the partial symmetric difference operation in the Iwata–Yokoi algorithm.

As with the case of the shortcut operation, one may wonder if this is something that may actually occur. Especially, considering that unlike in the case used for the shortcut operation, there exists many other, mostly simpler, augmenting



(a) After uncrossing



(b) After removing double usages of labeled edges in the same direction.

Figure 5.2: Example of the modification to the partial symmetric difference operation.

walks in the given labeled graph. Most of the possible augmenting walks do not inhibit the shown problem. Nonetheless, the augmenting walk shown in Figure 5.1 can be produced by the search procedure in several different ways. As such, the partial symmetric difference operation should be changed in such a way that the problem described above can no longer occur. Additionally, there is no guarantee that there does not exist a labeled graph where all augmenting walks inhibit this problem.

Solving the problem is rather trivial, as by Iwata and Yokoi's proof of Lemma 3.4 it is already shown that (ii) holds for the walk after the uncrossing operation. Thus, one can apply a similar strategy to the one described by Iwata and Yokoi to mitigate breaking (iii) in a similar fashion in the shortcut operation. The strategy used in the shortcut operation is to simply iterate the augmenting walk, checking if an edge inserted by the shortcut operation is used twice in the same direction. In such a case, the first occurrence of the edge and the edges between the two occurrences are removed from the augmenting walk. Since the two occurrences have the same symbols and (ii) holding before the removal of the edges, it is guaranteed that (ii) will hold afterwards. The only difference from the case of the shortcut operation is that the set of edges one has to check for duplicate visits is considerably larger. However, since both cases require a full iteration of the walk, the execution of this procedure will be no different than for the one used by Iwata and Yokoi in the shortcut operation. Thus, one can simply employ this strategy after the uncrossing step of the partial symmetric difference operation. Figure 5.2 shows the result of employing this strategy to the case of Figure 5.1.

Theorem 5.1 *Removing all duplicate usages of the same labeled edge in the same direction after the uncrossing operation by replacing the two occurrences* *and all edges between the two by one occurrence, results in the partial symmetric difference operation producing an augmenting walk.*

Proof: By Iwata and Yokoi's proof of Lemma 3.4, (ii) holds for the walk after the uncrossing operation. As Iwata and Yokoi mention, (i) also holds for the walk, as the initial part of the augmenting walk is exchanged for the last part of the T-path, and thus since (i) holds for T-paths, the interchange of edges does not break (i). For (iii), the uncrossing operation removes all duplicate usages of these newly added edges. Thus, the only way in which (iii) can be broken is if the usage of edges that existed in augmenting walk prior to the partial symmetric difference operation is now in a different direction. For free edges this does not matter, as the edges are used only once and thus the direction does not matter for the validity of (iii). For labeled edges, (iii) holds after duplicate occurrences have been removed. The outlined removal process removes a cycle in the walk. Since only internal vertices will be removed, (i) will not be broken by the operation. Since (ii) holds prior to the removal, the symbol preceding the first occurrence is different from the first symbol on the edge and the symbol succeeding the second occurrence is different from the second symbol on the edge. Thus, (ii) must also hold after the two occurrences and the edges between them have been replaced with a single occurrence, as this occurrence does not have the same symbols as the preceding nor succeeding symbols on the walk. Thus, (i), (ii) and (iii) all hold after the partial symmetric difference operation and the walk must be an augmenting walk. ■

Corollary 5.2 *The outlined operation does not change the time complexity of the partial symmetric difference operation.*

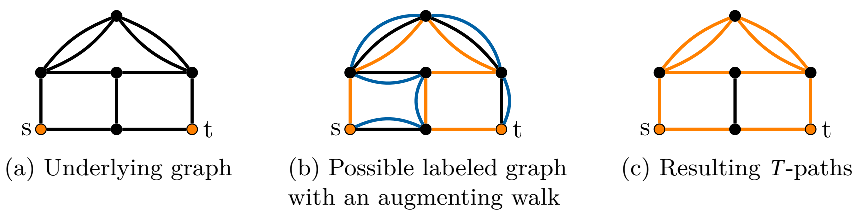
Proof: It follows from the description of the operation that it can be performed in a single iteration over the augmenting walk, with each step using a constant number of operations. As such, the operation should have a time complexity of $\mathcal{O}(Q)$. As the partial symmetric difference operation has a time complexity of $\mathcal{O}(E)$, including an iteration over the augmenting walk in the uncrossing operation stage, the time complexity of the partial symmetric difference operation can not increase by the outlined operation. ■

Chapter 6

Cycles in T-Paths

In their paper Iwata and Yokoi assume that all T-paths are of length $\mathcal{O}(V)$, as one would expect from the standard definition of a path. Since the length of a T-path may grow when it is involved in a partial symmetric difference operation, it is not guaranteed that this is the case. Especially, considering that the length of an augmenting walk is not bound by $|V|$, it seems unlikely that this is the case due to how the partial symmetric difference operation works. In fact, it is possible to show that in some cases a T-path may grow to a length of more than $|V|$. An example of such a case can be seen in Figure 6.1. There, a multigraph with seven vertices, of which two are terminals, is given. In Figure 6.1b, we can see a possible situation of a labeled graph with a single T-path and an augmenting walk. Performing the augmentation procedure two T-paths are found, as seen in Figure 6.1c. This is a valid set of T-paths, except for the fact that one of the T-paths contains 8 edges. Since the graph has only 7 vertices, this results in the T-path having at least one cycle. The case in the figure may naturally arise if a depth-first exploration strategy is used when selecting edges in the search procedure.

For the case of Figure 6.1, the increase in length does not result in the length exceeding $\mathcal{O}(V)$. This is mainly due to the simplicity of the graph used in the example. Consider instead a very large dense multigraph where more than $|V|$ T-paths may be found. A possible scenario that can occur is that in each of the first $|V|$ iterations of the Iwata–Yokoi algorithm an augmenting walk of length $|V|$ is found. If these augmenting walks contain a single P_1-segment in their second to last edge, with the edge of the P_1-segment being the second to last edge in P_1, then each iteration this would result in an increase in the length of the T-path by $|V|-2$ edges. After $|V|$ iterations, this would result in a T-path with a length of approximately $|V|^2-2|V|$, which is not $\mathcal{O}(V)$. Based on this, it seems like it may be enough to simply reduce the length of the T-paths between each iteration



(a) Underlying graph (b) Possible labeled graph with an augmenting walk (c) Resulting T-paths

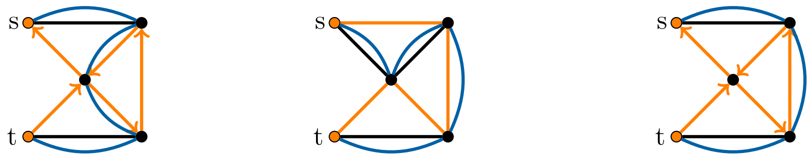
Figure 6.1: Example situation where cycles may occur on a T-path, and the length of a T-path may increase beyond the number of vertices.

of the algorithm. However, the length of a T-path may easily increase by more than $|V| - 2$ in each iteration of the Iwata–Yokoi algorithm. This is due to how the partial symmetric difference operation works. An augmenting walk may both give and take edges from the T-paths it interacts with. Thus, it can pick up edges from one T-path and deposit them into another. If this process was to repeat as many times as possible in a single application of the augmentation procedure, it could result in a T-path with a length of similar magnitude to that of the number of P-segments in the augmenting walk times $|V|$. Since the number of P-segments is bound by $\mathcal{O}(V)$, this would result in the length of a T-path possibly reaching $|V|^2$ in a single iteration.

All of this does not matter for the time complexity of the standard Iwata–Yokoi algorithm, as the only place where the length of a T-path affects the time complexity of the algorithm is in the augmentation procedure. There, the part of the time complexity that relies on the length of a T-path is also dependent on the length of the augmenting walk. Due to the edge-disjoint property of the T-paths, the combined length of all the T-paths may not exceed $|E|$. Thus, since the length of an augmenting walk is $\mathcal{O}(E)$, this increase in length does not increase the time complexity. For the improvements from the preliminary project, the length of the augmenting walk is bound by $\mathcal{O}(V)$. Thus, if the length of any T-path is not bound by $\mathcal{O}(V)$, the improvements do not result in an improved time complexity. Especially, the approach for reducing the size of the augmenting walk used in these improvements does not work properly when there are cycles in the T-paths.

While having T-paths of length $\mathcal{O}(E)$ would not result in any problems in regards to time complexity, ignoring the problem is not a viable solution. The algorithm is defined and proven with regards to the T-paths being without cycles, that is, being paths. As such, operations and definitions will not work properly with cycles in the T-paths.

The main problem with cycles is in regards to P-segments. The definition given by Iwata and Yokoi for a P-segment is that it is a sub-path of the augmenting walk that only contains edges that are either in a given T-path or self-loops of



(a) Problematic graph

(b) The partial symmetric difference operation

(c) Shortcut operation with new P-segment definition

Figure 6.2: Example of a problematic situations with cycles in T-paths

the T-path. Additionally, the preceding and succeeding edges in the augmenting walk must not be edges or self-loops of the T-path. As such, with the reliance on having no redundant self-loops, a P-segment is either a sub-path of the T-path or a self-loop. This property does not hold for T-paths with cycles. Then, as can be seen in Figure 6.2, one can have P-segments that consist of edges in the T-path, but does not form a sub-path of the T-path.

For a P-segment that is neither a sub-path nor a single self-loop of a T-path, there is no guarantee that the partial symmetric difference operation and the shortcut operation will work properly. For example, performing the partial symmetric difference operation in the labeled graph of Figure 6.2a will not give the expected result. Following the exact definition of the partial symmetric difference operation will result in the T-path and augmenting walk shown in Figure 6.2b. While there are modifications to both the augmenting walk and the T-path, the number of P-segments has not decreased and the resulting walk is not an augmenting walk, breaking property (ii) due to its symbols being $sstt$.

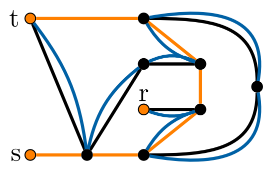
Since P-segments which are not sub-paths of the T-path cause problems, it may seem reasonable to define a P-segment to be either a consecutive part of the augmenting walk that is also a sub-path of the T-pathor a self-loop of the T-path. This definition would most likely work better with the partial symmetric difference operation. On the other hand, this definition does not work properly with the shortcut operation. Using the labeled graph in Figure 6.2a as an example, employing the shortcut operation a shortcut between the now two P-segments will be found. This, despite the fact that the shortcut is not valid. The result of the shortcut operation can be seen in Figure 6.2c, where the resulting walk is not an augmenting walk, breaking (ii) as earlier with the symbols $sstt$. The reason why a shortcut is found is that the shortcut operation, in step 1, checks for adjacent st-directed (and ts-directed) P_j-segments, as this would in a non-cyclic case indicate a shortcut of type 1. It may be possible to change the operation to ignore this specific case. Nonetheless, there are other problems in regards to time complexity with this definition of a P-segment. Finding all P-segments on an augmenting walk is no longer as simple as just iterating the augmenting walk.

All operations are defined based on the location of their vertices in the T-paths. This is in general problematic for cycles, where a vertex can appear more than once in a T-path. The problem should be easy enough to work around, as each edge is used at most once in a T-path, and thus the edges decide which occurrence of the vertex should be used. Nevertheless, the combination of all these problems, as well as the need to prove the entire algorithm anew, make it preferable to handle the situation by simply removing any cycles in the T-paths. Two viable solutions for removing cycles in the T-paths will be presented in the following sections. They differ in the time at which they remove the cycles, removing cycles either immediately after they appear or at the end of an application of the augmentation procedure. As such, they differ in complexity, with the immediate removal being more complex, but requiring no changes to the algorithm other than a procedure call.

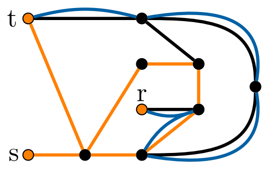
6.1 Immediate Cycle Removal

The most obvious solution to the cycle problem would be to simply remove any possible cycles from the T-path directly after an invocation of the partial symmetric difference operation. This way the T-paths will always be guaranteed to be without cycles. Removing the cycles can not be done with a standard cycle removal algorithm, as the cycles may contain a number of P_j-segments. If a cycle was removed without handling these P_j-segments, the result could be consecutive appearances of symbols in the walk, breaking property (ii) of an augmenting walk. Additionally, this could result in double usage of free edges, breaking property (iii) of an augmenting walk. Thus, an un-cycling procedure must be created so that each possible situation with P_j-segments in the cycle is properly handled. Figure 6.3 shows an example of a situation, where after the partial symmetric difference operation the T-path contains a cycle. In this case a single edge from the cycle is used in the augmenting walk. For this specific case, removing the cycle without consideration to the P_j-segment does not result in an invalid augmenting walk. This is only the case for this simple class of cases. For the other four nontrivial classes, which will be introduced later, modifications have to be made to the augmenting walk.

To be able to design a procedure that removes cycles from a T-path, a definition of when a P_j-segment is in a cycle must be decided on. We will define this as a P_j-segment that contains either one or more edges in the cycle or one or more self-loops on the given T-path on a vertex in the cycle. For the self-loops this is required minimum, as the self-loops will be removed from the labeled graph when the edges of the cycle are removed from the T-path. For the edges on the cycle, one may consider restricting the definition to only include P_j-segments where all



(a) Before the partial symmetric difference operation



(b) After the partial symmetric difference operation

Figure 6.3: Example of a simple case with one P_j-segment in the resulting cycle.

the edges are in the cycle. This, as for any other P_j-segments at least one of the edges remain labeled after the removal of the cycle and thus (ii) would not be broken. Regardless, the labeled edges in the cycle may be used twice, and thus (iii) will not hold with this definition.

Given the above definition, the cases can be divided into five classes when the cycle on the T-path contains at least one P_j-segment. In addition, there is the trivial case of no P_j-segments in the cycle. The trivial case can mostly be ignored, as it does not require any changes to the augmenting walk. The trivial case and these five classes are given below, with general examples shown in Figure 6.4. When the descriptions refer to the first and last P_j-segments, the P_j-segments are considered by their appearance in the augmenting walk.

Type 0 There are no P_j-segments in the cycle.

Type 1 There is exactly one P_j-segment in the cycle.

(a) The symbols preceding and succeeding this P_j-segment are different.

(b) The symbols preceding and succeeding this P_j-segment are equal.

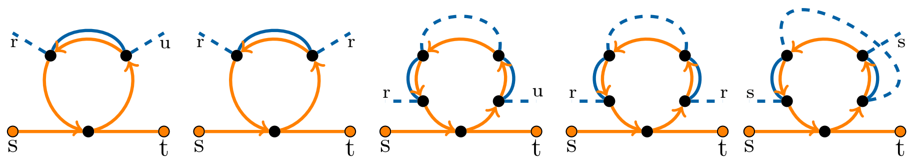
Type 2 There are two or more P_j-segments in the cycle.

(a) The symbols preceding and succeeding, respectively, the first and last P_j-segment are different.

(b) The symbols preceding and succeeding, respectively, the first and last P_j-segment are equal and neither s nor t.

(c) The first and last P_j-segment are directed in the opposite direction of each other and the symbols, respectively, preceding and succeeding them are equal and either s or t.

For these classes of cycles, one may wonder if the classes contain all possible cases. For both the cases of none and one P_j-segment in the cycle, it can easily



(a) Type 1a (b) Type 1b (c) Type 2a (d) Type 2b (e) Type 2c

Figure 6.4: General examples of the five nontrivial classes of cycles on a T-path

be identified that all possible cases have been covered. For two or more P_j-segments in the cycle, this is not as easy. Type 2a covers all cases where the symbol preceding and succeeding, respectively, the first and last P_j-segment are different. Thus types 2b and 2c must cover the cases where the symbols are equal. Since type 2b covers the cases where the symbols are equal and neither s nor t, type 2c must cover the remaining cases. Type 2c does in fact do this, as having the first and last P_j-segment in the cycle as oppositely directed is the only option in this case. If the two P_j-segments were equally directed with the preceding and succeeding symbols being equal and either s or t, then the augmenting walk would not suffice (ii). As such, these classes contain all possible cases.

The examples shown in Figures 6.4c, 6.4d and 6.4e are simplifications of the possible situations, as there can possibly be more than two P_j-segments in the cycle. Additionally, any of the P_j-segments in any of the five non-trivial classes may be self-loops. In reality, there can not be a self-loop in type 1a, as it would be a redundant self-loop, of which there is assumed to be none. Additionally, while not depicted in Figure 6.4, it is possible for there to be P_j-segments that overlap partially with the cycle and partially with the remaining parts of the T-path, as can be seen in Figure 6.7. While the properties of an augmenting walk limits the number of such P_j-segments to four, we will now show that under the assumption that neither the T-path nor the first free edges of the augmenting walk have any cycles in them before the partial symmetric difference operation, there can only be a maximum of two such P_j-segments in any given cycle. We will later design the un-cycling procedure in such a way that this property will always hold true.

Lemma 6.1 *Given a T-path with no cycles and an augmenting walk with no cycles in its initial free edges, any cycle created by the partial symmetric difference operation must contain both edges that were in the T-path before the operation and edges that were in the augmenting walk before the operation.*

Proof: Assume a cycle exists for which this is not true, then it must consist of either only free edges from the augmenting walk or only edges from the T-path. In either case, since the order of the edges from neither the augmenting walk nor the T-path change, this would mean that there must have been a cycle in either the T-path or the initial free edges of the augmenting walk. Thus, such a cycle can not exist. ■

Lemma 6.2 *Given a T-path with no cycles and an augmenting walk with no cycles in its initial free edges, after applying the partial symmetric difference operation there may only be a maximum of two P_j-segments in any cycle that are part of both the cycle and the other edges of the T-path. These P_j-segments will only use the edges of the T-path that belonged to the T-path before the partial symmetric difference operation.*

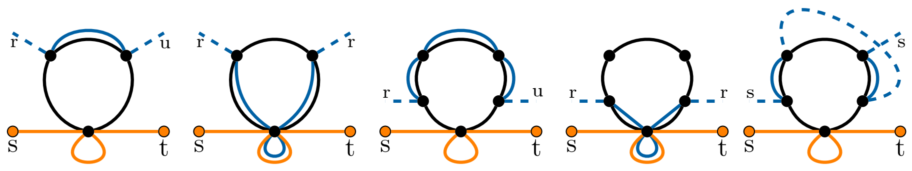
Proof: The partial symmetric difference operation modifies the T-path in such a way that it does not mix the edges that prior to the operation belonged to the T-path with those that prior to the operation belonged to the augmenting walk. Using this and Lemma 6.1, the T-path can be divided into two parts, one with the edges that belonged to the T-path prior to the operation and one with the edges from the augmenting walk. The point of division will be at a location in the T-path that is inside the cycle.

Since the edges that belonged to the augmenting walk were free, there can not be any P_j-segments using them right after the partial symmetric difference operation. Thus, there can only be P_j-segments on the part of the new T-path that belonged to the T-path prior to the partial symmetric difference operation. Given that the previously discussed division of the T-path into its two parts is performed somewhere on the cycle, this means that only one side of the cycle can have P_j-segments, and thus it follows that there can only be two P_j-segments that are both part of the cycle and the rest of the T-path. Additionally, by the above logic, these must use the edges of the T-path that belonged to the T-path prior to the partial symmetric difference operation. ■

Each of the previously discussed classes must be handled differently. We will for the moment assume that there are no P_j-segments in the cycle that are both part of the cycle and the remainder of the T-path. While these cases must be addressed at some point, they should be handled differently and will thus be discussed later. What follows is a description of the way that each of the different classes should be handled. Figure 6.5 shows the application of these procedures to each of the cases shown in Figure 6.4.

Type 0 Remove the cycle from the T-path.

Type 1 (a) Remove the cycle from the T-path. If the P_j-segment is a self-loop, remove it from the augmenting walk.



(a) Type 1a (b) Type 1b (c) Type 2a (d) Type 2b (e) Type 2c

Figure 6.5: Results of handling the different nontrivial classes of cycles on a T-path

(b) Remove the cycle from the T-path, and replace the P_j-segment in the augmenting walk by the edges in the cycle between the first vertex of the P_j-segment and the connecting vertex of the cycle, the self-loop of the T-path on the connecting vertex and then the edges in the cycle between the connecting vertex of the cycle and the last vertex of the P_j-segment.

Type 2 (a) Remove the cycle from the T-path, and replace the part of the augmenting walk between the start of the first P_j-segment to the end of the last P_j-segment by the part of the cycle between the first vertex of the first P_j-segment and the last vertex of the last P_j-segment.

(b) Remove the cycle from the T-path, and replace the part of the augmenting walk from the start of the first P_j-segment to the end of the last P_j-segment by the edges in the cycle between the first vertex of the first P_j-segment to the connecting vertex of the cycle, the self-loop of the T-path on the connecting vertex, and then the edges in the cycle between the connecting vertex of the cycle and the last vertex of the last P_j-segment.

(c) Assuming that the first P_j-segment is st-directed (change st-directed and ts-directed otherwise), find the last st-directed P_j-segment in the cycle. If the P_j-segment is not the same as the first one, then replace the part of the augmenting walk between the first vertex of the first and the last vertex of the second by the part of the cycle between them. Now find the first ts-directed P_j-segment in the cycle. If the P_j-segment is not the same as the last one in the cycle, then replace the part of the augmenting walk between the first vertex of the first and the last vertex of the second by the part of the cycle between them. If the two segments created by this process overlap, then replace the first by the edges between the first vertex of the first and the first vertex of the second, and the second by the edges between the last vertex of the first and the last vertex of the second. Then, also flip the edges on the part of the augmenting walk between the two. If after all of this any of the two segments is a self-loop, simply remove it from the augmenting walk. Finally, remove the cycle from the T-path. If some of the edges in the augmenting walk, outside of the cycle, were flipped and are now used twice in the same direction, replace the two appearances and the part of the augmenting walk between the two by one appearance (as done in Chapter 5 for the partial symmetric difference operation).

The ways chosen to handle the different classes of cycles aim at modifying the augmenting walk the least and in the simplest possible way. For types 1a, 1b, 2a and 2b, this is done by using parts of the cycle between the first entry and last exit from the cycle, inserting a self-loop of the T-path when required to guarantee (ii) holding in the resulting walk. This strategy does not work in case 2c, where the symbols do not allow for the usage of a self-loop on the T-path to guarantee this property. Instead a more intricate procedure is employed, exploiting the guarantees for symbols preceding and succeeding P_j-segments. We will now present a formal proof of the validity of these procedures.

Theorem 6.3 *Given an augmenting walk and a cycle on a T-path without any P_j-segments that are both part of the cycle and the remainder of the T-path, performing the actions above will result in a T-path with one or more removed cycles and an augmenting walk.*

Proof: It is easy to see that the T-path has one or more cycles removed, as the given cycle is removed as part of the procedure for all the different types. Here, at least one cycle is removed, more if there are overlapping cycles. Following, it will be shown on a case by case basis that the resulting walk is an augmenting walk.

For type 0, the augmenting walk does not use any edges in the cycle, and thus there are no changes to the walk nor its edges. As such, the resulting walk remains an augmenting walk.

For type 1a, the only possible change to the augmenting walk is the removal of a self-loop, so condition (i) still holds. Each of the labeled edges in the cycle is turned into free edges, but, since there is only one P_j-segment in the cycle, these edges are used at most once and thus (iii) still holds. Additionally, the symbols from the P_j-segment will disappear. Since the symbols preceding and succeeding the P_j-segment are different, the removal will not result in the consecutive appearance of symbols on the walk and (ii) holds. Thus, the resulting walk is an augmenting walk.

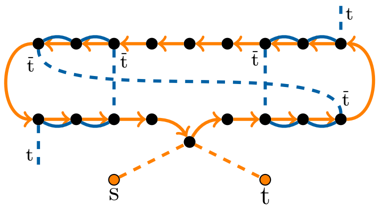
For type 1b, the edges of the P_j-segment are changed to the other edges of the cycle and a self-loop of the new T-path. None of the edges in the cycle is used in other places in the augmenting walk, as then there would be more than one P_j-segment in the cycle. Thus, the edges in the cycle are used only once. Additionally, the self-loop is on a vertex in the cycle and thus, by the above definition of a P_j-segment in a cycle, it is either not used beforehand or is the P_j-segment in the cycle. This means that it is used only once in the resulting augmenting walk. So, (iii) still holds. The symbol preceding and succeeding the P_j-segment can be neither s nor t, as the P_j-segment uses both of these symbols. So, the symbols from the introduced self-loop do not cause any consecutive appearance of symbols on the augmenting walk and separate the equal symbols that would otherwise result in consecutive appearances of a symbol. Thus, (ii) holds. Since some of the internal edges in the augmenting walk are replaced with some of the internal edges in the T-path, (i) holds, as none of the internal edges of the T-path can be adjacent to terminals.

For type 2a, the part of the augmenting walk that includes any P_j-segments in the cycle is replaced with parts of the cycle. Since this part has no symbols and the symbols preceding and succeeding the cycle are distinct, this results in no consecutive appearances of symbols, and thus (ii) holds. Similarly to 1a, all usages of edges in the cycle are replaced with a single usage of some of the edges, thus (iii) also holds. Finally, as in the case of type 1b, the internal edges of the augmenting walk are replaced with internal edges of the T-path, and thus (i) holds.

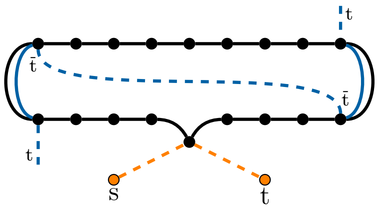
Type 2b replaces the part of the augmenting walk that contains the P_j-segments in the cycle with parts of the cycle and a self-loop. By the same logic as for type 1b, these parts of the cycle are only used once, and if the self-loop is already used, it will be removed in the operation and then used again. Thus, condition (iii) holds. Similarly to the earlier cases, internal edges of the augmenting walk are replaced with internal edges of the T-path, and thus (i) holds. Since neither the symbol preceding nor the one succeeding the new part of the augmenting walk is s or t, the self-loop can not result in any consecutive symbols, and (ii) holds.

Type 2c is a bit more complicated than the others. This proof assumes that the first P_j-segment is st-directed and the last is ts-directed. For the opposite case, the proof can be repeated in its exact form with only references to s and t interchanged. This assumption also means that the symbol preceding the first P_j-segment and the one succeeding the last P_j-segment is t.

The logic in type 2c follows from the observation that since the last symbol in an st-directed P_j-segment is t, the succeeding symbol in the augmenting walk must be something different from t. Thus, replacing part of the augmenting walk from the start of the first st-directed P_j-segment to the end of the last st-directed



(a) Before removing the cycle



(b) After removing the cycle

Figure 6.6: Handling type 2c

P_j-segment in the cycle with the parts of the cycle between the entrance of the first and exit of the last, will result in a valid string of symbols when removing the cycle from the T-path. This, as the symbol in the augmenting walk preceding the first st-directed P_j-segment is t, and since the cycle consists of free edges after being removed from the T-path, the succeeding symbol in the resulting walk must be the symbol succeeding the last st-directed P_j-segment, that is not a t. Especially, since all st-directed P_j-segments in the cycle are removed by this operation, the succeeding symbol cannot belong to a P_j-segment in the cycle. Thus, the succeeding symbol must exist after removing the cycle from the T-path. By the same logic, the case of replacing the remaining ts-directed P_j-segments in the cycle by parts of the cycle must result in a valid combination of symbols on the augmenting walk. For these, the symbol before the first ts-directed P_j-segment is not t and the one after the last is t. Thus, removing all symbols between does not break (ii). To help illustrate this see Figure 6.6.

After these combinations of P_j-segments, exactly two P_j-segments will remain in the cycle. These two P_j-segments are separated by a sub-walk of the augmenting walk which contains at least one symbol (more accurately a multiple of two symbols), whose first and last symbols are neither t. There is, however, no guarantee that the two P_j-segments do not overlap. Which after removal of the cycle from the T-path would result in (iii) not being satisfied. If this is the case, then the procedure change which vertices the two P_j-segments should start and end in. Due to the P_j-segments being on a cycle there are two ways to travel between the two vertices each P_j-segment connects. Thus, it is guaranteed that there is at least one way to create the P_j-segments, either before or after the change of start and end vertices, where the two created P_j-segments do not overlap. Since the sub-walk of the augmenting walk has t as neither its first nor last symbol, this can be done and still guarantee no consecutive appearances of symbols on the resulting walk after removal of the cycle from the T-path. Especially, this is due to the possibility of flipping the order and direction of usage of the edges in the augmenting walk. Finally, removing any of the P_j-segments if they are self-loops does not change anything after removing the cycle from the T-path.

Due to there being no overlap between the two P_j-segments, each edge in the cycle is used at most once. Since no other edges are added to the augmenting walk and all self-loops on the cycle are removed, the only way that (iii) can not hold is if two labeled edges are now used in the same direction. This may be the case when the edges in the sub-walk are flipped. In that case, the procedure employs the same strategy as introduced for the partial symmetric difference operation. This procedure was shown to reduce the number of usages in the same direction of a labeled edge from two to one, and thus after this procedure (iii) holds. As for (i), only internal edges of the T-path have been inserted into the augmenting walk, and thus (i) holds for the resulting walk. Finally, for (ii) it was shown that for this combination of P_j-segments, including the fix for the overlap, the first P_j-segment was preceded by t and succeeded by a symbol different from t, with the second P_j-segment preceded by a symbol different from t and succeeded by t. Since the sub-walk of the augmenting walk between the two P_j-segments has at least one symbol, this means that the after removing the cycle from the T-path, the symbols on the resulting walk are $\dots t\bar{t}\dots\bar{t}t\dots$. Where the dots represent zero or more symbols already on the augmenting walk that are guaranteed to not break (ii). This sequence of symbols can not contain consecutive appearances of symbols, and thus (ii) holds for the resulting walk. ■

Earlier, in Chapter 3, a procedure for removing redundant self-loops from the T-path was presented. As this procedure can be run after the un-cycling procedure, the un-cycling procedure does not need to guarantee no redundant self-loops in the augmenting walk. Nevertheless, it is interesting to see if the procedure may result in redundant self-loops. For the types 1b and 2b, the procedure creates a single self-loop. This newly created self-loop is not redundant, as both cases guarantee that the preceding and succeeding symbols of the self-loop are equal. As the solutions for the other cases do not create any self-loops, the un-cycling procedure can not insert any new redundant self-loops. It may, however, result in already existing self-loops becoming redundant. In all of the cases this may happen if the last labeled edge before the cycle or the first labeled edge after the cycle is a self-loop. As then the symbols either preceding or succeeding these self-loops may change. This may additionally happen in type 2c if there is a self-loop in the part of the augmenting walk between the two visits to the cycle. As a result, after any of these case the augmenting walk may contain up to several consecutive redundant self-loops.

Now that the un-cycling operations are known to produce valid states for the Iwata–Yokoi algorithm, it must be shown that using these operations do not increase the number of P-segments in the augmenting walk. This, as the augmentation procedure relies on reducing the number of P-segments to 0. An increase in P-segments may result in the augmentation procedure never finishing, or the time complexity of the algorithm increasing.

Theorem 6.4 *Given an augmenting walk and a cycle on a T-path without any P_j-segments that are both part of the cycle and the remainder of the T-path, performing the actions above will result in the number of P-segments in the augmenting walk staying constant for types 0 and 1b and decreasing for the remaining types.*

Proof: For type 0, no changes are made to the augmenting walk nor its edges, and thus the number of P-segments stays constant. For type 1b, a single P_j-segment is replaced by a self-loop. Each is counted as exactly one P-segment, and thus the number of P-segments in the augmenting walk stays constant. For types 1a and 2a, one or more P_j-segments are replaced by edges in the cycle. Since these edges are free after the cycle is removed from the T-path, one or more P-segment(s) are removed and no new are added. Thus, the number of P-segments in the augmenting walk decreases by one in case of type 1a and two or more in case of type 2a. For type 2b, two or more P_j-segments are replaced by cycle edges and a self-loop. Thus, by removing two or more P_j-segments and adding just a single P_j-segment, the number of P-segments in the augmenting walk decreases. For type 2c, two or more P_j-segments are replaced by cycle edges and the content between them in the augmenting walk. The content between them is already counted for in the augmenting walk, so no new P-segments are added and two or more are removed. This results in a decrease of P-segments in the augmenting walk. ■

Theorems 6.3 and 6.4 both assume that there are no P_j-segments that are partially in the cycle and partially outside. In some cases this is not a valid assumption, and the un-cycling procedure must thus be able to handle these as well. These can not be handled exactly like the other P_j-segments, as some of their edges remain labeled after the removal of the cycle. Thus, blindly applying the previous rules will not always result in an augmenting walk. The partially overlapping P_j-segments can be split two parts, one that is only in the cycle and one that is not in the cycle. Using this, the same rules as above can be applied. Figure 6.7 shows an example of a simple case with such a P_j-segment. In this situation, splitting the P_j-segment would result in one P_j-segment consisting of the edge between vertices 1 and 2 and another P_j-segment consisting of the edge between vertices 2 and 3. This would be equivalent to having a type 1 case, since the symbol preceding the P_j-segment in the cycle is t and the one succeeding this P_j-segment can not be t. Thus, the procedure for type 1 cases is applied, resulting in the augmenting walk and labeled graph shown in Figure 6.7b. Note that while the term split is used here, it is only used in a loose sense. There are

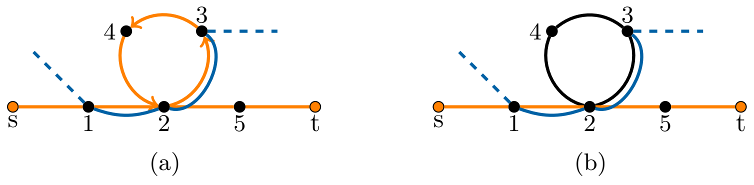


Figure 6.7: Example of a P_j-segment that partially overlaps with a cycle

no actual changes made to the augmenting walk, but rather the use of split here is in regards to the definition of a P_j-segment for this specific procedure.

Theorem 6.5 *Given an augmenting walk and a cycle on a T-path, performing the actions above while considering any P_j-segments that partially overlap the cycle using the splitting method will result in a T-path and an augmenting walk.*

Proof: Theorem 6.3 showed that this is valid when no P_j-segments partially overlap the cycle. So, assume that is at least one such partially overlapping P_j-segment in the augmenting walk. For the T-path there is no differences in whether or not there exists a partially overlapping P_j-segment, as the same changes are made to the T-path, and thus the properties for a T-path must hold. For the augmenting walk none of the properties directly care about which T-path a certain edge belongs to as long as the symbols on edge allow for property (ii) to be true. When considering the partially overlapping P_j-segment as two distinct P_j-segments, there is thus no difference between this and the case where the P_j-segment outside the cycle belongs to another T-path between s and t. Thus, it must follow from Theorem 6.3 that the operations result in an augmenting walk.
■

In Theorem 6.5, it was seen that considering each of the partially overlapping P_j-segments as two distinct P_j-segments results in a situation that is from the perspective of validity equivalent to there being no partially overlapping P_j-segments. However, removing only one of the two P_j-segments that represent one of these partially overlapping P_j-segments, will not result in a reduction in the number P-segments. Thus, it must be shown that the operations still result in a reduction in the number of P-segments or the number staying constant.

Theorem 6.6 *Given an augmenting walk and a cycle on a T-path where the cycle is produced by the partial symmetric difference operation on a cycle-free T-path and an augmenting walk without cycles in its initial free edges, performing the actions above while considering any P_j-segments that partially overlap the cycle using the splitting method will result in the number of P-segments in the augmenting walk either decreasing or staying constant.*

Proof: Theorem 6.4 showed that the statement holds true when there are no partially overlapping P_j-segments in the augmenting walk. Thus, only cases where there is at least one partially overlapping P_j-segment need to be considered here. Since type 0 does not contain any P_j-segments in the cycle, it does not need to be considered.

For type 1a, all P_j-segments in the cycle are removed, while no others are added. Thus, when there is one partially overlapping P_j-segment, the number of P-segments must stay constant. Similarly, types 2a and 2c replace the P_j-segments on the cycle by cycle edges and thus add no new P_j-segments. As such, these cases result in the number of P-segments staying constant or being reduced.

For type 1b, a single P_j-segment is replaced by another P_j-segment. This would mean that a partially overlapping P_j-segment would result in an increase in the number of P_j-segments. However, it is not possible for there to be any partially overlapping P_j-segments in type 1b. This, as by (ii), all partially overlapping P_j-segments have either the symbol preceding the cycle part of the P_j-segment be equal to the last symbol in the cycle part or the symbol succeeding the cycle part of the P_j-segment be equal to the first symbol in the cycle part. Thus, the symbol preceding the cycle part can not be equal to the symbol succeeding the cycle part, and thus type 1b suffices the statement above.

For type 2b, a single P_j-segment is created while two or more on the cycle are removed. For the cases where there are three or more P_j-segments in the cycle, Lemma 6.2 guarantee that there can not be an increase in the number of P-segments. This, as only a maximum of two of the P_j-segments can be partially overlapping P_j-segments, and thus at least one of the three or more P_j-segments is fully removed. For the remaining part of type 2b, where there are only two P_j-segments in the cycle, there are two possibilities, either both are partially overlapping P_j-segments or one is. For the latter case, the non partially overlapping P_j-segment is fully removed, and a single P_j-segment is created. Thus, the number of P_j-segments at worst stays constant. For the case in which there are two partially overlapping P_j-segments the first must be ts-directed and the second st-directed, as they share edges and thus must be oppositely directed. Additionally, if the first was st-directed and the second ts-directed, then the symbols preceding and succeeding the first and second P_j-segments in the cycle would both be t and thus the case would be of type 2c. This means that the part outside of the cycle of both P_j-segments will appear on the augmenting walk between the two parts of the P_j-segments on the cycle. Thus, the handling of type 2b will remove both parts of the partially overlapping P_j-segments and there will be a reduction in the number of P-segments by one. ■

It should be noted that while the proof of Theorem 6.6 only shows that the number of P-segments stays constant, it is possible to show that for types 2a, 2b and 2c the procedure always results in a reduction. Meaning that for all these cases, the number of remaining iterations is reduced.

In a simple cycle removal algorithm, where memory usage is not of concern, a walk W can have all its cycles removed in time complexity $\mathcal{O}(W+V)$ and space complexity $\mathcal{O}(V)$. This can be done creating a new path through iteration of the walk, checking if each vertex reached has already appeared in W. If a vertex has been visited before, the part of the new walk succeeding the last visit is removed.

The concept of such a simple cycle removal algorithm could be employed to this more complex cycle removal problem. However, this would result in either having to perform changes to the augmenting walk every time a cycle is removed or needing lots of bookkeeping to keep track of changes to the augmenting walk to coalesce them into one large change. Neither of these options can easily be used to achieve the required time complexity. Using the assumptions discussed early on in this section, it is possible to greatly reduce the amount of work required to remove all possible cycles in the T-path.

Theorem 6.7 *Given that there are no cycles in the T-path and no cycles in the free edges of the augmenting walk, all cycles in the T-path after the partial symmetric difference operation will be removed if the cycle for which its first edge appears first on the T-path is removed.*

Proof: Under the given conditions Lemmas 6.1 and 6.2 can be used. These say that any cycle in the T-path has two parts, one consisting of prior edges of the T-path and one consisting of edges from the augmenting walk. Since there is no mixing of the edges from the T-path and the augmenting walk, all cycles share at least two edges, one from the prior T-path and one from the augmenting walk, that is, all cycles contain the merging point of the edges from the prior T-path and the augmenting walk. As such, all cycles in the T-path at least partially overlap.

When walking the T-path in any of the directions, all the cycles must have started before the discussed merging point, and can not have ended before after the merging point. The assumptions of the theorem guarantee that there is at most one visit to each vertex on each side of the merging point. Thus, the removal of the cycle starting the earliest on one side of the merging point must remove the visits to all vertices visited on both sides of the merging point, from this side of the merging point. If this was not true, then there would be a cycle starting earlier in the T-path, since it must have one visit to its connecting vertex on each side of the merging point. Thus, the resulting T-path has at most one visit to each vertex and there can not be any cycles in the T-path. ■

To be able to have a procedure use the results discussed in this section, there must be a guarantee that the earlier assumptions about the lack of cycles in the T-paths and in the initial free edges of the augmenting walk hold. For the augmenting walk, a simple cycle removal algorithm on the sub-walk consisting of its initial free edges can guarantee this property. The partial symmetric difference operation is the only operation that makes any changes to the T-paths. Thus, earlier assumptions hold if the T-paths do not have any cycles before the start of the augmentation procedure. This, as the given procedure immediately removes any new cycles in the T-paths after the partial symmetric difference operation. Since the T-paths are not changed outside the augmentation procedure, a simple cycle removal algorithm applied to the augmenting walk at the end the augmentation procedure will guarantee all new T-paths to be without cycles.

Below, the pseudocode of the two procedures MODIFY-AUGMENTING-WALK and REMOVE-CYCLES-FROM-T-PATH is given. The MODIFY-AUGMENTING-WALK procedure modifies the augmenting walk in the way described for the aforementioned different types of cycles in the T-paths. REMOVE-CYCLES-FROM-T-PATH performs the entire removal procedure for the cycles in a T-path, using the MODIFY-AUGMENTING-WALK procedure to modify the augmenting walk.

```
REMOVE-CYCLES-FROM-T-PATH($Q, P_j$)
1  $cycle\text{-}start = cycle\text{-}end = |P_j|$
2  for $i, v$ in ENUMERATE($P_j.vertices$)
3      if $v.visited \neq$ NIL and $cycle\text{-}start \geq v.visited$
4          $cycle\text{-}start = v.visited$
5          $cycle\text{-}end = i$
6      else if $v.visited ==$ NIL
7          $v.visited = i$
8  for $v$ in $(P_j[1 : cycle\text{-}start - 1] + P_j[cycle\text{-}end + 1 : \textbf{end}]).vertices$
9      $v.visited =$ NIL
10 $cycle\text{-}segments = [\ ]$
11 for $p_j$ in $Q.P_j\text{-}segments$
12     $p'_j =$ FILTER($e$ in $p_j.edges$, $e.src.visited \neq$ NIL and $e.dst.visited \neq$ NIL)
13     if $|p'_j| > 0$
14         add $p'_j$ to $cycle\text{-}segments$
15 MODIFY-AUGMENTING-WALK($Q$, $P_j$, $cycle\text{-}segments$)
16 $P_j = P_j[1 : cycle\text{-}start - 1] + P_j[cycle\text{-}end : \textbf{end}]$
```

The REMOVE-CYCLES-FROM-T-PATH procedure iterates over the T-path, finding the cycle that starts the earliest in the T-path. This is done using a simple numbering scheme, numbering each vertex by the number of preceding vertices in the T-path. This way, when reaching a vertex with this number set, one can easily determine if the cycle found occurs before any previously found cycles. Next, the procedure removes the numbering of all but the vertices in the selected cycle. The remaining numbers can then be used to select P_j-segments that are in the given cycle and filtering these to only contain edges that are classified as in the cycle. Finally, the MODIFY-AUGMENTING-WALK procedure is used to modify the augmenting walk before removing the cycle from the T-path.

```
MODIFY-AUGMENTING-WALK($Q, P_j, segments$)
 1  $s1 = segments[1]$
 2  $s2 = segments[\textbf{end}]$
 3  if $|segments| == 1$
 4      if $s1.previous\text{-}symbol == s1.succeeding\text{-}symbol$
 5          replace $Q[s1.start : s1.end]$ by the other part of the cycle with an
 6                  inserted self-loop at the connecting vertex
 7      else if $s1$ is a self-loop
 8          remove $Q[s1.start : s1.end]$
 9  else if $|segments| > 1$
10      if $s1.previous\text{-}symbol \neq s2.succeeding\text{-}symbol$
11          replace $Q[s1.end : s2.start]$ by $P_j$ between $s1.end$ and $s2.start$
12          if $s1$ is a self-loop
13              remove $Q[s1.start : s1.end]$
14          if $s2$ is a self-loop
15              remove $Q[s2.start : s2.end]$
16      else if $s1.previous\text{-}symbol$ != $s$ and $s1.previous\text{-}symbol$ != $t$
17          replace $Q[s1.start : s2.end]$ by the other part of the cycle with an
18                  inserted self-loop at the connecting vertex
19      else
20          $s1n = s1$
21          $s2p = s2$
22          for $s$ in $segments$
23              if $s.direction == s1.direction$
24                  $s1n = s$
25                  $s2p = s2$
26              else if $s2p == s2$
27                  $s2p = s$
28          if $P_j[s1.start : s1n.end]$ and $P_j[s2p.start : s2.end]$ overlap
29              reverse $Q[s1n.end : s2p.start]$
30              replace $Q[s2p.start : s2.end]$ by $P_j$ between $S1n.end$ and $s2.end$
31              replace $Q[s1.start : s1n.end]$ by $P_j$ between $S1.start$ and $s2p.start$
32              REMOVE-DUPLICATE-USAGES-IN-SAME-DIRECTION($Q$)
33          else
34              replace $Q[s2p.start : s2.end]$ by $P_j$ between $S2p.start$ and $s2.end$
35              replace $Q[s1.start : s1n.end]$ by $P_j$ between $S1.start$ and $s1n.end$
```

The MODIFY-AUGMENTING-WALK procedure simply performs the modification actions to the augmenting walk for each of the possible classes of cycles outlined earlier. Thus, we will not cover the workings of the pseudocode in detail here. In the pseudocode, sN is used to denote a P_j-segment. Note that for an sN, $sN.direction$ is the direction of the P_j-segment in the augmenting walk. Also note that $sN.start$ and $sN.end$ denote the first and last vertex in the P_j-segment.

It remains to be shown that the suggested changes can be used in the Iwata–Yokoi algorithm without resulting in an increase in time complexity. The remainder of this section will focus on this.

Lemma 6.8 *The procedure* Modify-Augmenting-Walk *has a time complexity of* $\mathcal{O}(Q + P_j)$.

Proof: Any **replace** operation can be performed in $\mathcal{O}(Q + P_j)$, as a replace operation consists of iterating over the T-path to find which edges to add to the augmenting walk and creating a new augmenting walk from edges already in the augmenting walk and edges from T-path. These operations thus have time complexities of, respectively, $\mathcal{O}(P_j)$ and $\mathcal{O}(Q + P_j)$. Since the number of P_j-segments on the augmenting walk can not be greater than the length of the augmenting walk, the loop of lines 22 through 27 has a time complexity of $\mathcal{O}(Q)$, due to each iteration consisting of a small number of constant time operations. All conditional checks, except for line 28, are simple comparisons of values, and thus can be performed in constant time. For the conditional check of line 28, the overlap is a simple check of two lines overlapping. Overlap of two lines can be performed by iteration in $\mathcal{O}(P_j)$ time, due to each line being bound by $\mathcal{O}(P_j)$. Assuming the numbering from Remove-Cycles-From-T-Path exists, the check is equivalent to the overlap of two number ranges and can be performed in constant time. Additionally, it was previously shown that the Remove-Duplicate-Usages-In-Same-Direction procedure has a time complexity of $\mathcal{O}(Q)$. Thus, it follows that the whole procedure has a combined time complexity of $\mathcal{O}(Q + P_j)$. ■

Lemma 6.9 *The procedure* Remove-Cycles-From-T-Path *has a time complexity of* $\mathcal{O}(Q + P_j)$.

Proof: Lines 2 through 7 is a single loop over the T-path with a constant number of simple operations in each iteration. It thus has a time complexity of $\mathcal{O}(P_j)$. The same goes for the loop of lines 8 and 9. In each iteration of the loop of line 11, another loop, line 12, is performed. Since line 11, is an iteration over the P_j-segments in the augmenting walk, the total number of iterations on line 12 across all P_j-segments considered is bound by the length of the augmenting walk. Thus, line 12 has a time complexity of $\mathcal{O}(Q)$ in the whole procedure. Finding all the P_j-segments in an augmenting walk was shown by Iwata and Yokoi to have a $\mathcal{O}(Q)$ time complexity, and thus the loop of lines 11 through 14 must have a time complexity of $\mathcal{O}(Q)$. By Lemma 6.8, line 15 has a time complexity of $\mathcal{O}(Q + P_j)$.

Finally, line 16 is a simple copy instruction on some of the edges in the T-path and must therefore have a time complexity of $\mathcal{O}(P_j)$. This gives the procedure as a whole a combined time complexity of $\mathcal{O}(Q + P_j)$. ■

Given this guarantee on the time complexity of the un-cycling procedure, the augmentation procedure can be modified to use the un-cycling procedure. Below, a modified version, Augment*, of the standard Augment procedure from the Iwata–Yokoi algorithm is shown with the changes needed to handle cycles in T-paths. All the changes have been discussed and proven above, so only a proof for the time complexity of the Iwata–Yokoi algorithm with Augment* is given here.

```
Augment*($Q, P$)
1  while $\mu_P(Q) > 0$
2      if $Q$ contains a possible shortcut operation
3          perform the shortcut operation
4      else
5          un-cycle the first free edges of $Q$
6          perform the partial symmetric difference operation
7          call Remove-Cycles-From-T-Path
8  Un-Cycle($Q$)
9  add $Q$ to $P$
```

Theorem 6.10 *Exchanging* Augment *by* Augment* *does not increase the time complexity of the Iwata–Yokoi algorithm.*

Proof: The Augment* procedure adds an un-cycling of the first free edges of the augmenting walk and a call to Remove-Cycles-From-T-Path every time the partial symmetric difference operation is performed. The un-cycling was shown earlier to have a time complexity of $\mathcal{O}(V + Q)$, which is bound by $\mathcal{O}(E)$ since one can assume that $|V|$ is bound by $\mathcal{O}(E)$ (for the case of the edge-disjoint T-paths problem, as otherwise vertices could simply be removed). Lemma 6.9 showed that the time complexity of the Remove-Cycles-From-T-Path procedure is $\mathcal{O}(Q + P_j)$, which is again equivalent to $\mathcal{O}(E)$. Since the partial symmetric difference operation has a time complexity of $\mathcal{O}(E)$, these other operations do not change the time complexity of the augmentation procedure. Finally, the last un-cycling of the augmenting walk is, by the above logic, of time complexity $\mathcal{O}(E)$, and thus does not increase the $\mathcal{O}(VE)$ time complexity of Augment. Since the time complexity of Augment is not change, the time complexity of the Iwata–Yokoi algorithm does not change. ■

6.2 Delayed Cycle Removal

As seen in Section 6.1, removing the cycles in a T-path immediately after they occur requires a non-trivial procedure. Another possible solution is simply to wait until after the augmentation procedure. Then, by the definition of the algorithm, the augmenting walk does not have any P-segments and thus the cycles in the T-paths can be removed without consideration to the augmenting walk. This means that a simple un-cycling procedure can be used. As previously mentioned, cycles in the T-paths may result in a number of different problems, as such it must be shown that the augmentation procedure works properly with cycles in the T-paths as long as there are no cycles in the T-paths before starting the augmentation procedure. Theorem 6.11 is the main observation that allows for proving this.

Theorem 6.11 *Given a T-path that contained no cycles before the augmentation procedure, the part of the T-path containing P_j-segments will at any stage during the augmentation procedure be a sub-path of the original T-path.*

Proof: This property must hold at the start of the augmentation procedure, as then the T-paths and augmenting walk have not changed and thus the T-paths and the P-segments are equivalent to before the augmentation procedure. As such, it must be shown that for both the shortcut operation and the partial symmetric difference operation, the operation does not change the T-paths and the augmenting walk in such a way that the property no longer holds. Note that as long as the property holds, the sub-path containing P_j-segments is cycle-free. If only the cycle-free sub-path of the T-path is considered when finding where a vertex is on the T-path, all operations must function as proven by Iwata and Yokoi.

Assume the property holds before the shortcut operation is performed. Then in the shortcut operation two P_j-segments are replaced with a new P_j-segment in the augmenting walk. This new P_j-segment goes from the start of the first removed P_j-segment to the end of the second removed P_j-segment, using the part of the T-path between these two vertices. Since the new P_j-segment only uses the edges of the removed P_j-segments and the edges on the T-path between the two, the P_j-segment must be a sub-path of the original T-path. Otherwise, the edges between the two removed P_j-segments were not in the original T-path and thus the property did not hold. The shortcut operation may delete several of the occurrences of P_j-segments in the augmenting walk if they are now used twice in the same direction due to the new P_j-segment. This operation does not change the T-path, and may only remove P_j-segments from the augmenting walk. Thus, the sub-path of the T-path containing P_j-segments either stays the same or shrinks. As such, it must still be a sub-path of the original T-path. Given this, the property holds also after applying the shortcut operation.

Now assume the property holds before the start of the partial symmetric difference operation. The partial symmetric difference operation changes the T-path by removing some of the edges in one of its ends and replacing them with free edges from the augmenting walk. The edges that are removed from the T-path may have contained P_j-segments; however, since the edges are no longer in the T-path, they are no longer P_j-segments. Thus, the sub-path of the T-path containing P_j-segments will shrink. The new edges in the T-path were free edges, and thus no longer appear in the augmenting walk. As such they can not contain any P_j-segments, and the sub-path of the T-path that contains P_j-segments will not cover any of these new edges. Thus, since no new P_j-segments are added, and, especially, the new edges of the T-path does not contain any P_j-segments, the sub-path in the T-path containing P_j-segments must be a sub-path of the prior sub-path. If follows from this that the new sub-path is also a sub-path of the original T-path. Additionally, P_j-segments may be removed or have their direction or order of appearance changed during the uncrossing operation. Removal, as earlier shown, results in the sub-path either staying the same or shrinking. For the change of direction and order of usage of edges, the edges used do not change. Thus, the sub-path of the T-path containing P_j-segments stays constant, and the property must also hold after the partial symmetric difference operation.

Since the property holds at the start of the augmentation procedure, and the two operations performed in the augmentation procedure do not break the property when performed, the property must hold at each step of the augmentation procedure. ■

It follows directly from Theorem 6.11 that the delayed cycle removal strategy should work as well as any other strategy, as long as only the non-changed part of the T-path is considered when finding the position of a vertex in the T-path. This, as the theorem guarantees that all operations will only need to work on a non-cyclic part of the T-path, and thus if the operations do not work, then there must be something wrong with the operations. In that case, the problem with the operations would affect all other solutions to the cycle problem, including the earlier shown immediate cycle removal strategy.

It should be noted that employing this delayed cycle removal strategy requires that any modifications to any of the operations in the algorithm are made such that the modified operations do not break the premise of Theorem 6.11. In general, one can assume this to be true for all minor modifications, such as the ones in the previous chapters, where there are no new edges of the T-path added to the augmenting walk. This, as was used extensively in the proof of the theorem, is due to the fact that removing or changing the order/direction of edges that form P_j-segments in the augmenting walk, do not result in changes to the sub-path. Only operations that change the T-path and/or insert new edges into the augmenting walk can break the premise of the theorem. For more extensive changes and/or new operations one should in general repeat the proof of Theorem 6.11 for the affected parts of the algorithm.

We will now present a modified version of the augmentation procedure. The modified version, AUGMENT*, is based on the AUGMENT procedure in the standard Iwata–Yokoi algorithm. The algorithm assumes that each time a shortcut or partial symmetric difference operation is performed, the aforementioned sub-path of the given T-path is used to find the location of a vertex in the T-path. Keeping track of this sub-path for each T-path should be doable without any increase in time complexity. This can be done by simply starting each augmentation procedure with it set to the entire T-path and reducing the size each time the partial symmetric difference operation is performed on the T-path. This would generally not result in the sub-path being tight around the P_j-segments, however, it guarantees the sub-path to be a sub-path of the original T-path, and thus free of cycles. The sub-path can be reset during the un-cycling of the T-paths.

```
AUGMENT*($Q, P$)
 1  $P^*$ = empty list
 2  while $\mu_P(Q) > 0$
 3      if $Q$ contains a possible shortcut operation
 4          perform the shortcut operation
 5      else
 6          perform the partial symmetric difference operation
 7          add the changed $T$-path to $P^*$
 8  UN-CYCLE($Q$)
 9  for $P_j$ in $P^*$
10      UN-CYCLE($P_j$)
11  add $Q$ to $P$
```

In regards to the chosen un-cycling strategy, several different strategies can work within the time complexity of the augmentation procedure. There may be larger differences in the performance of the different strategies, especially considering that the different approaches result in the final un-cycling of the T-paths having different time complexities. In the AUGMENT* procedure a very simple approach is taken, consisting of adding each modified T-path to a list without consideration to duplicates. Then in the end, the list is iterated and each T-path is un-cycled. Due to possible duplicates in the list, each T-path can be attempted un-cycled several times. This is not needed and a non-duplicate approach might be better. The improvements of a non-duplicate approach are highly dependent on how often the partial symmetric difference operation is applied to a single T-path. Since, the shortcut operation reduces the number of P_j-segments in the augmenting walk, the partial symmetric difference operation is in most cases likely called once or twice for each T-path. Thus, the simple approach is chosen here.

Corollary 6.12 *Given a set of n T-paths without cycles and an augmenting walk, the* AUGMENT* *procedure produces a set of $n+1$ T-paths without cycles.*

Proof: It follows from Theorem 6.11 and the previous discussion that the operations should work as well here as in the standard case where there are no cycles. This means that a set of $n+1$ T-paths is produced at the end of the algorithm. Additionally, both the augmenting walk and all changed T-paths have their cycles removed in the end, and thus the new T-paths must be without cycles. ■

Given Corollary 6.12 and the fact that the search procedure does not change T-paths, it must follow that the Iwata–Yokoi algorithm works properly with the AUGMENT* operation. However, while touched on earlier, it remains to show that the time complexity does not increase as a result of the modifications made to the augmentation procedure.

Theorem 6.13 AUGMENT* *does not increase the time complexity of the Iwata–Yokoi algorithm.*

Proof: The AUGMENT* procedure differs from the standard augmentation procedure by adding T-paths to a list and at the end performing an un-cycling operation for each T-path in the list. Adding the T-paths to the list is a constant time operation. Furthermore, the number of T-paths in the list is limited by the number of P_j-segments in the augmenting walk, that is, bound by $\mathcal{O}(V)$. It was earlier showed that the un-cycling can be performed in $\mathcal{O}(V+P_j)$. A single T-path can be in P^* several times; however, each time following the first there are no cycles in the T-path and its length is less than $|V|$. Thus, the un-cycling of already un-cycled T-paths has a combined time complexity of $\mathcal{O}(V^2)$. For the first time un-cycling of each T-path, the combined length of all the T-paths can at most be $|E|$ due to the edge-disjoint property of the T-paths. Thus, all the first time un-cyclings have a combined time complexity of $\mathcal{O}(V^2+E)$. This, as there is at most V of these and P_j sum to E across them. This gives the un-cycling a total time complexity of $\mathcal{O}(V^2+E)$, which for the edge-disjoint T-paths problem is bound by $\mathcal{O}(VE)$. $\mathcal{O}(VE)$ is the time complexity of the augmentation procedure and thus the complexity of the algorithm as a whole does not increase. ■

The time complexity guarantee of Theorem 6.13 is only valid for the standard Iwata–Yokoi algorithm. For the improvements suggested in the preliminary project, using the AUGMENT* procedure would result in an increased time complexity. Additionally, the preliminary project made changes to the augmentation procedure that change the augmenting walk. The changes revolve around the introduction of a procedure, SIMPLIFY, that takes an augmenting walk with at most three visits to each vertex and reduces the number of visits to at most two. It does this by identifying cycles in the augmenting walk that can be removed while still keeping property (ii) intact. Since the procedure only removes edges from the augmenting walk, the procedure can not break the premise of Theorem 6.11. Thus the procedure itself can still be used in the delayed cycle removal strategy. The problem lies in the reliance on the augmenting walk visiting each vertex a maximum of three times.

To guarantee that there were always three or fewer visits to each vertex in the augmenting walk, the modifications of the preliminary project relied on the T-paths being without cycles. This meant that both the shortcut operation and the partial symmetric difference operation would to the augmenting walk add at most one visit to each vertex, with the SIMPLIFY procedure reducing this to two after each operation. For the delayed cycle removal strategy this is no longer the case. For the shortcut operation, Theorem 6.11 still guarantee this for the augmenting walk, as it adds edges from a cycle-free sub-path of the T-path. For the partial symmetric difference operation this is not the case, where the cycles in the T-path may result in the augmenting walk having more than three visits to a single vertex. Thus, modification to the improvements is required to achieve the time complexity using the delayed cycle removal approach.

Lemma 6.14 *Assume that all T-paths are cycle free before the augmentation procedure. In a single application of the augmentation procedure, the length of any T-path can only increase by at most twice the maximum possible length the augmenting walk may take in that application.*

Proof: The only situation in which the length of a T-path may change in the augmentation procedure is when the partial symmetric difference operation is applied to a P_j-segment of this T-path. Any such change is a result of one end of the T-path being changed for one part of the augmenting walk. As such, if the same end of a T-path is changed twice, then by Theorem 6.11 the edges added in the first change are all removed in the second change. This, as between the two changes no P_j-segment may have been created on the edges that were added in the first change. This means that any T-path may at the end of an application of the augmentation procedure contain, at most, edges from two changes. Thus, since each change may at most give the T-path all but one edge from the augmenting walk while only removing one edge from the T-path, two such changes may give the T-path up to four edges less than twice the maximum number of possible edges in the augmenting walk. ■

Using the result of Lemma 6.14, one can see that the length of any T-path is bound by the length of the augmenting walk when performing delayed cycle removal. Thus, it suffices to show that the length of the augmenting walk can be bound by $\mathcal{O}(V)$, and it must follow that the T-paths are also bound by $\mathcal{O}(V)$. We will use the SIMPLIFY procedure from the preliminary project in the modified augmentation procedure, although we can no longer rely on the T-path having at most three visits to each vertex. Thus, we note that it was shown that the procedure would reduce the number of visits to each vertex visited three or more times, by at least one. Allowing the procedure to be applied repeatedly to an augmenting walk in order to reduce the number of visits until there is at most two visits to each vertex. Before discussing the modifications to the augmentation procedure, two lemmas regarding the growth of the T-paths and the augmenting walk, will be shown. These are later needed to show the validity of the chosen modifications.

Lemma 6.15 *Assume that the augmenting walk has at most X visits to any vertex at all times before the partial symmetric difference operation is performed. Additionally, assume that all T-paths are cycle free before the start of the augmentation procedure. Then, all T-paths will at all points during the execution of the augmentation procedure, have at most have $2X+1$ visits to any vertex.*

Proof: Using the same logic as in Lemma 6.14, it follows that any change to a T-path removes any previously added edges during the augmentation procedure. As such, any T-path consists of at most one set of edges from a single state of the augmenting walk, the original edges of the T-path and another set of edges from a single state of the augmenting walk. For the edges from the augmenting walk it is guaranteed that each set visits any vertex at most X times. This means that the two sets combined visit each vertex at most $2X$ times. Additionally, the edges from the original T-path are without cycles and thus visit each vertex at most once. This restricts the number of visits to a single vertex to at most $2X+1$. ■

Lemma 6.16 *Assume that the augmenting walk has at most X visits to any vertex at all times before the partial symmetric difference operation is performed during an augmentation procedure. Additionally, assume that all T-paths are cycle free before the start of the augmentation procedure. Then, after the partial symmetric difference operation the augmenting walk will have at most $2X+1$ visits to any vertex.*

Proof: The partial symmetric difference operation gives the augmenting walk all edges of the T-path on one side of the given P_j-segment. By Lemma 6.15, especially its proof, it follows that the edges added to the augmenting walk from the T-path consist of at most $X+1$ visits to any vertex. As such the number of visits to any vertex in the resulting augmenting walk must be at most $2X+1$. ■

By Lemmas 6.15 and 6.16 it suffices to find a way to reduce the number of visits to each vertex in the augmenting walk from $2X+1$ to X in order to limit the length of an augmenting walk to $\mathcal{O}(V)$. Especially, by Theorem 6.11 the shortcut operation increases the number of visits by at most one, that is, it requires no changes from the original improvements made in the preliminary project. As previously mentioned, the SIMPLIFY method from the preliminary project can be used for this.

Iwata and Yokoi showed that augmenting walk returned from the search procedure had at most two visits to each vertex. As such, we will set X to two. Then, Lemma 6.15 guarantees that after the partial symmetric difference operation the number of visits to each vertex is at most five. Performing the SIMPLIFY procedure three times (or less if each vertex has at most two visits after an application), the number of visits will be reduced to two. Based on this, the following combination of the augmentation procedure from the preliminary project, the AUGMENT* procedure and the modifications discussed above, is suggested as the AUGMENT** procedure. As for the AUGMENT* procedure, it is assumed that when the shortcut operation and the partial symmetric difference operation are performed, only the unmodified sub-path of the T-path will be used to search for vertices.

```
AUGMENT**($Q,P$)
 1  $P^*$ = empty list
 2  **while** $\mu_P(Q) > 0$
 3      **if** $Q$ contains a possible shortcut operation
 4          perform the shortcut operation
 5          SIMPLIFY($Q$)
 6      **else**
 7          perform the partial symmetric difference operation
 8          **add** the changed $T$-path **to** $P^*$
 9          **while** $Q$ contains more than two visits to a vertex
10              SIMPLIFY($Q$)
11  UN-CYCLE($Q$)
12  **for** $P_j$ **in** $P^*$
13      UN-CYCLE($P_j$)
14  **add** $Q$ to $P$
```

Corollary 6.17 *Given a set of n T-paths without cycles and an augmenting walk, the* AUGMENT** *procedure produces a set of $n+1$ T-paths without cycles.*

Proof: It follows from Corollary 6.12 that the delayed cycle removal changes made to the augmentation procedure work properly. Additionally, it follows from the above discussion that the SIMPLIFY method does not break the premise of Theorem 6.11. Especially, it was proven in the priliminary project that the SIMPLIFY procedure resulted in the resultant walk being an augmenting walk. Based on this, the procedure will always have an augmenting walk and reduce the number of P_j-segments to zero. Thus, the procedure will produce $n+1$ T-paths. By the proof of Corollary 6.12, the T-paths will have any cycles removed and thus all will be cycle free. ■

Theorem 6.18 *The* AUGMENT** *procedure has a time complexity of $\mathcal{O}(V^2)$.*

Proof: By Theorem 6.11 and Lemmas 6.15 and 6.16, it follows that the length of the T-paths and augmenting walk are bound by $\mathcal{O}(V)$, as the SIMPLIFY procedure is used to reduce their lengths between operations. Since the SIMPLIFY procedure does not increase the number of P_j-segments in the augmenting walk, the number of iterations is the same as in the standard Iwata–Yokoi algorithm, that is $\mathcal{O}(V)$. It was shown in the preliminary project that all calls to the shortcut operation, the partial symmetric difference operation and the SIMPLIFY procedure are bound by $\mathcal{O}(V)$ when the T-paths and augmenting walk have lengths bound by $\mathcal{O}(V)$. Thus, it suffices to show that there is a constant number of calls to these procedures in a single iteration of the loop on line 2. Lines 4, 5 and 7 contain single calls to the procedures, while line 10 contains calls to the SIMPLIFY procedure in a loop. From Lemma 6.16, it follows that the number of visits to the augmenting walk is at most five at the start of this loop, and thus the number of calls is restricted to at most three. This means that the loop on lines 2 through 10 has a time complexity of $\mathcal{O}(V^2)$.

Lines 11 through 13 consists of a number of un-cyclings. For a walk, W, the un-cycling can be performed in $\mathcal{O}(V+W)$. Since all the T-paths and the augmenting walk are bound by $\mathcal{O}(V)$, this means that each un-cycling can be performed in $\mathcal{O}(V)$. The number of un-cycling operations required is bound by the number of P_j-segments in the initial augmenting walk, $\mathcal{O}(V)$, resulting in lines 11 through 13 have a time complexity of $\mathcal{O}(V^2)$. As such, the whole AUGMENT** procedure has a time complexity of $\mathcal{O}(V^2)$. ■

Since the augmentation procedure originally used in the improvements has a time complexity of $\mathcal{O}(V^2)$, AUGMENT** may be used instead without changes to the time complexity. As such, the improvements from the preliminary project may be employed together with the delayed cycle removal strategy.

6.3 Cycle Removal Efficiency

Having introduced two distinct methods for handling cycles in the T-paths, one may wonder which of the two strategies is the most effective. The answer is not straight forward and most likely depends on the specific class of graphs, that is, how likely are cycles to occur and especially cycles with P_j-segments.

The delayed cycle removal method is a much simpler method with each un-cycling requiring fewer checks than the immediate cycle removal procedure. Thus, one can expect that the delayed removal approach may result in fewer operations. Especially, considering that one in practice only needs to perform the un-cycling procedure once for each changed T-path, rather than once each time the T-path is changed. On the other hand, the immediate removal approach may reduce the number of P_j-segments in the augmenting walk when removing a cycle, and, as such, reduce the number of iterations required in the augmentation procedure. Additionally, the immediate removal of cycles immediately reduces the length of the T-paths and possibly the augmenting walk, which generally results in the other operations in the augmentation procedure requiring fewer operations per application. Note that the delayed cycle removal, with the improvements from the preliminary project, will also keep the T-paths and the augmenting walk short.

Additionally, it is worth noting that for the delayed removal approach it is possible to perform the un-cycling of the T-paths in parallel. This, as the T-paths are edge-disjoint, and thus are completely separated from each other. This is not possible in the case of the immediate removal approach, where at any point only one T-path may contain cycles. Thus, for a parallel computing approach, the delayed removal approach is highly preferred.

Chapter 7

Concluding Remarks and Future Work

Throughout this thesis a handful of mis- and/or unhandled edge cases have been presented. The existence of these edge cases results in the incorrectness of the algorithm, that is, the Iwata–Yokoi algorithm, as presented by Iwata and Yokoi in their paper, is not fully correct. Regardless, no problems were found with the underlying concepts of the algorithm. The problems, in regards to correctness, is these edge cases that can be handled by performing minor modifications or expansions to sub-procedures. This means that the underlying concepts of the Iwata–Yokoi algorithm seem to be correct, with only a few edge cases being missed. This is in itself not unlikely for complex algorithms such as this one. It is also a general result of the standard approach to proving correctness, which is human proofs. While well-written human proofs can generally be assumed to be true, there is always a slight chance that some small edge case has been missed or some seemingly insignificant assumption does not fully hold.

A more appropriate question to discuss is if the Iwata–Yokoi algorithm with the modifications suggested in this thesis is correct. In the same way that one generally assumes that algorithms with appropriate proven theorems are correct, there is nothing indicating that the modified version of the algorithm is not correct. During the work on this thesis several attempts have been made at finding other unhandled edge cases, but with the cases of Chapters 3 through 6 covered there seems to be no remaining unhandled edge cases. As such, one can in the same way as other algorithms are assumed to be correct, given their proofs, assume that the Iwata–Yokoi algorithm with the given modifications is correct.

To be pedantic one could perform a formal proof of the algorithm. This approach, which describes the algorithm and its theorems using a formal language, aims at showing the correctness through sequences of sentences in a formal language. A formal proof derives the stated properties and theorems from the description of the algorithm through these sequences of sentences. Due to its rigorous nature, deriving a formal proof is laborious and often requires extensive work. Due to the complexity of the Iwata–Yokoi algorithm, along with the theorems being tightly knit, we deem performing a formal proof as of little value. This is mainly due to the large amount of work required, even with a computer-assisted proof, in relation to the relatively small gain in certainty on the correctness.

Another approach that can be taken to better guarantee the correctness of the algorithm, and that ties well with the continuation of the project, is checking the algorithm using an implementation. By implementing the algorithm, it can be run over a large number of automatically generated inputs while checking the corresponding outputs for correctness. For the edge-disjoint T-paths problem, most of the properties of the output set can easily be checked, as the check for if the output is T-paths and if they are edge-disjoint are both easily verifiable. For the maximum-cardinality property of the set, this is not as easy, as checking if an outputted set has maximum cardinality requires the usage of Mader's theorem. Mader's theorem's brute force approach makes it very time consuming to check more than a small number of simple cases. Thus, a stopgap solution of comparing the size of the result of implementations of two distinct algorithms can be used. This approach, while not perfect, will either result in finding mistakes in one of the implementations (or both) or guarantee that if there are any mistakes in the implementations, they result in the same, wrong, results for two distinct algorithms. The latter is less likely than mistakes in one algorithm, and thus should provide a better guarantee for the correctness of the algorithms. This is also the way that one of the edge cases described in this thesis was found. Since the thesis was originally supposed to start performing empirical evaluation of the algorithms, this method was used when checking if the algorithms had been correctly implemented. This lead to finding a case in which cycles appeared in the T-paths.

Looking forward, there is still a lot of work required in order to establish a fully practical approach to the topic. Figure 7.1 shows the same timeline as in the introduction, where the preliminary project and this thesis have laid the foundation for implementation, evaluation and comparison of the different algorithms. The next step is to continue implementing the different algorithms, using these implementations to perform evaluation and comparison of the algorithms. Especially, one should try to implement the different variants of the Iwata–Yokoi algorithm suggested in this thesis and the preliminary project, measuring which of the variations result in increased efficiency and, if any, which results in the greatest increase in efficiency.

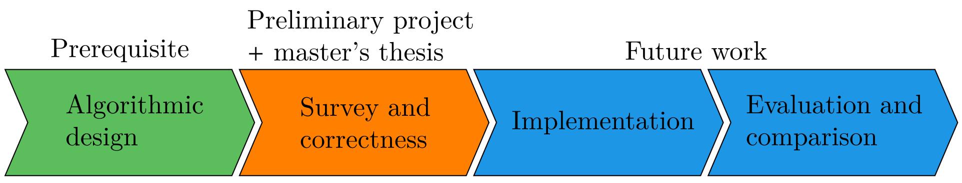


Figure 7.1: Project timeline for a practical approach to the T-paths problems

Bibliography

[1] T. Gallai. “Maximum-minimum sätze und verallgemeinerte faktoren von graphen”. In: *Acta Mathematica Hungarica* 12 (1961), pp. 131–173.

[2] W. Mader. “Über die Maximalzahl kantendisjunkter A-wege”. In: *Archiv der Mathematik* 30 (1 1978), pp. 325–336.

[3] W. Mader. “Über die Maximalzahl kreuzungsfreier H-wege”. In: *Archiv der Mathematik* 31 (1 1978), pp. 387–402.

[4] S. Iwata and Y. Yokoi. “A Blossom Algorithm for Maximum Edge-Disjoint T-Paths”. In: *Proceedings of the Thirty-first Annual ACM-SIAM Symposium on Discret Algorithms, SODA 2020, Salt Lake City, Utah, USA, January 5-8, 2020.* 2020.

[5] J. Edmonds. “Paths, Trees, and Flowers”. In: *Canadian Journal of Mathematics* 17 (1965), pp. 449–467.

[6] A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency.* Springer, 2003.

[7] H. Y. Cheung, L. C. Lau, and K. M. Leung. “Algebraic Algorithms for Linear Matroid Parity Problems”. In: *ACM Trans. Algorithms* 10.3 (2014), pp. 1–26.

[8] F. L. Gall. “Powers of Tensors and Fast Matrix Multiplication”. In: *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation.* ISSAC ’14. Kobe, Japan: ACM, 2014, pp. 296–303. ISBN: 978-1-4503-2501-1.

[9] H. Hummel. *The Edge-Disjoint T-paths problem.* Project in TDT4501. Norwegian University of Science and Technology, Department of Computer Science, Dec. 2019.

[10] R. L. Rivest T. H. Cormen C. E. Leiserson and C. Stein. *Introduction to Algorithms.* 3rd ed. The MIT Press, 2009. ISBN: 978-0-262-53305-8.

[11] J.C.M. Keijsper, R.A. Pendavingh, and L. Stougie. “A linear programming formulation of Mader’s edge-disjoint paths problem”. In: *Journal of Combinatorial Theory, Series B* 95 (2006), pp. 159–163.