Diplomarbeit

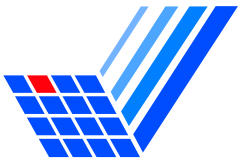
Effiziente Extraktion von Kuratowski-Teilgraphen

Jens M. Schmidt

Gutachter:

Prof. Dr. Petra Mutzel
Dipl.-Ing. Markus Chimani

16. März 2007



Fachbereich Informatik
der Universität Dortmund

Danksagung

An dieser Stelle möchte ich allen danken, die durch ihre Unterstützung zu dieser Diplomarbeit beigetragen haben, insbesondere Prof. Dr. Petra Mutzel, die mir dieses spezielle Diplomarbeitsthema ermöglicht hat, sowie Markus Chimani für die konstruktiven Gespräche und seine ständige Diskussionsbereitschaft. Ein Dank geht auch an Sophia Steireif und meine Eltern, ohne die dieses Studium nicht denkbar wäre.

Dortmund, 16. März 2007 Jens M. Schmidt

Kurzzusammenfassung

Ein Graph ist nach dem Satz von Kuratowski [Kur30] genau dann *planar*, wenn er keine *Kuratowski-Subdivisions* enthält. Eine einzelne davon kann von modernen Planaritätstests bei nicht planaren Eingabegraphen in Linearzeit extrahiert werden. Allerdings existieren Anwendungen, die nicht nur eine, sondern möglichst viele dieser Kuratowski-Subdivisions benötigen.

Dazu gehören Branch-and-Cut-Algorithmen für einige NP-schwere Probleme, wie beispielsweise die Kreuzungsminimierung oder auch verschiedene Varianten des Maximum Planar Subgraph Problems. Die Kuratowski-Subdivisions ermöglichen dort die Berechnung von zusätzlichen Nebenbedingungen einer LP-Relaxierung. Dabei ist es wünschenswert, dass die Kuratowski-Subdivisions paarweise entweder möglichst viele Kanten gemeinsam haben oder weitgehend kantendisjunkt sind.

In dieser Diplomarbeit wird ein Algorithmus entworfen und analysiert, der mehrere Kuratowski-Subdivisions in linearer Zeit $O(n + m + \sum_{K \in \mathrm{S}} |E(K)|)$ extrahieren kann, wobei S die Menge der gefundenen Kuratowski-Subdivisions ist. Dieser Algorithmus stellt eine Erweiterung des aktuellen Planaritätstests von Boyer und Myrvold [BM04] dar und kann zusätzlich so modifiziert werden, dass entweder möglichst ähnliche oder möglichst verschiedene Kuratowski-Subdivisions ausgegeben werden. Die Laufzeit des Algorithmus ist dabei asymptotisch optimal.

Aus diesem Algorithmus wird ein zweiter Ansatz entwickelt, der in der Praxis mehr Kuratowski-Subdivisions extrahieren kann, dafür aber zu einer superlinearen Laufzeit führt. Beide Verfahren werden implementiert und deren Praxistauglichkeit verglichen.

Inhaltsverzeichnis

1 Einführung

1.1 Intention

Ein Graph $G = (V, E)$ ist genau dann *planar*, wenn er ohne Kantenkreuzungen in die Ebene eingebettet werden kann. Ist er nicht planar, existiert nach dem Theorem von Kuratowski [Kur30] eine Unterteilung der Graphen K_5 oder $K_{3,3}$ in G. Solche auch *Kuratowski-Subdivisions* genannten Unterteilungen werden für die Nebenbedingungen von verschiedenen *Branch-and-Cut*-Algorithmen benötigt.

Branch-and-Cut-Algorithmen sind eine Kombination aus Schnittebenenverfahren und Branch-and-Bound. Viele NP-schwere, kombinatorische Optimierungsprobleme wie beispielsweise das *Travelling Salesperson Problem* lassen sich mit Branch-and-Cut derzeit am schnellsten lösen. Dabei wird das behandelte Problem als ganzzahliges, lineares Programm formuliert und dessen Relaxation gelöst. Falls diese Lösung ganzzahlig ist, wurde damit das Optimum gefunden. Andernfalls ist die Lösung fraktional und es muss eine Ungleichung existieren, die die aktuelle Lösung vom Optimum trennt. Ungleichungen dieses Typs beschreiben sogenannte Schnittebenen im Lösungsraum und werden dem linearen Programm als Nebenbedingung hinzugefügt. Falls keine Nebenbedingungen gefunden werden können, verzweigt die Methode mittels Branch-and-Bound zu einem Teilproblem.

Im Falle der *Kreuzungsminimierung* und des *Maximum Planar Subgraph Problems* können die Schnittebenen aus den Kuratowski-Subdivisions des Eingabegraphen berechnet werden. In jeder Kuratowski-Subdivision gilt nämlich, dass sich mindestens zwei enthaltene Kanten überkreuzen müssen, beziehungsweise, dass für jeden planaren Subgraphen mindestens eine Kante gelöscht werden muss. Aus diesen beiden Aussagen lassen sich für beide Probleme neue, benötigte Schnittebenen ableiten.

Allerdings wird dabei vorausgesetzt, dass mehrere Kuratowski-Subdivisions bekannt sind. Derzeitige Planaritätstests können aber nur eine einzelne Kuratowski-Subdivision in Linearzeit extrahieren. Enumerationsverfahren wären zwar in der Lage, sämtliche vorkommende Kuratowski-Subdivisions zu extrahieren, sind aber höchst inpraktikabel, da es im Allgemeinen exponentiell viele Kuratowski-Subdivisions geben kann. Ziel dieser Diplomarbeit ist es deswegen, einen effizienten Algorithmus zu entwickeln und zu analysieren, der möglichst viele Kuratowski-Subdivisions extrahiert.

1.2 Graphentheoretische Grundlagen

Dieses Kapitel enthält eine Einführung in die grundlegenden Begriffe der Graphentheorie. Einige Definitionen sind dabei an [Die05] und [EHK+96] angelehnt.

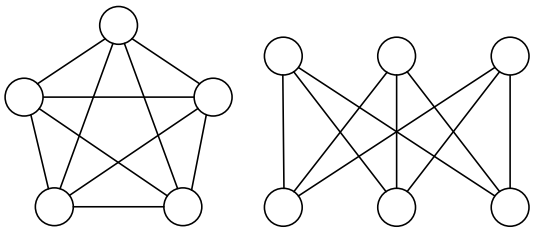
Sei $\binom{V}{2}$ die Menge aller ungeordneten Paare aus der Menge V.

Definition 1.2.1. Ein *ungerichteter Graph* G ist ein geordnetes Paar (V, E), wobei V die endliche Menge der *Knoten* und $E \subseteq \binom{V}{2}$ die Menge der *Kanten* darstellt. Ein *gerichteter Graph* erlaubt geordnete Knotenpaare, so dass dort etwas weniger restriktiv $E \subseteq V \times V$ gefordert wird.

Die Anzahl der Knoten wird dabei als $n := |V|$ und die Kantenanzahl als $m := |E|$ notiert. Die Knoten- und Kantenmengen können in Bezug auf einen Graphen G auch durch die Funktionen $V(G) := \{v_1, \ldots, v_n\}$ und $E(G) := \{e_1, \ldots, e_m\}$ angegeben werden.

Ein Knoten $v \in V$ und eine Kante $e \in E$ heißen *inzident*, falls $v \in e$ gilt. Die Anzahl der zu $v \in V$ inzidenten Kanten wird mit $degree(v)$ bezeichnet. Zwei Knoten $v \in V$ und $w \in V$ heißen *adjazent* oder *benachbart*, falls $\{v, w\} \in E$.

Der Graph $K_n := (V, \binom{V}{2})$ heißt auch *vollständiger* Graph. Ein Graph G ist genau dann *bipartit*, falls $V(G)$ sich in zwei Mengen A und B partitionieren lässt, so dass jede Kante zu Knoten in A und B inzident ist. Falls für alle $a \in A$ und $b \in B$ die Kante $\{a, b\}$ existiert, wird der Graph *vollständig bipartit* genannt und mit $K_{|A|,|B|}$ bezeichnet. Abbildung 1.1 zeigt den vollständigen Graphen K_5 und den vollständigen, bipartiten Graphen $K_{3,3}$.



(a) K_5 (b) $K_{3,3}$

Abbildung 1.1: Der vollständige Graph K_5 und der bipartite Graph $K_{3,3}$.

Die bisher definierten Graphen heißen *einfach*, da jedes Knotenpaar durch maximal eine Kante verbunden ist. Falls Mehrfachkanten zwischen zwei Knoten auftreten oder Kanten $\{v, v\}$ mit demselben Endknoten v existieren, sogenannte *Self-Loops*, muss die Definition auf *Multigraphen* erweitert werden.

Ein *Multigraph* G ist ein geordnetes Paar (V, E) von endlich vielen Knoten V und endlich vielen Kanten E mit einer Abbildung $E \to \binom{V}{2} \cup \bigcup_{v \in V} \{v, v\}$. Diese Abbildung ordnet jeder Kante ihre zwei inzidenten Knoten zu, die nicht notwendigerweise verschieden sein müssen. Ein Multigraph G kann damit Mehrfachkanten und Self-Loops enthalten, insbesondere sind mehrere Self-Loops an demselben Knoten möglich.

Ein Graph $G = (V, E)$ heißt *endlich*, falls $|V|$ und $|E|$ endlich sind. Falls nicht explizit anders erwähnt, wird im Folgenden von endlichen und einfachen Graphen ausgegangen.

Definition 1.2.2. Sei ein Graph $G = (V, E)$ gegeben. Ein Graph $G' = (V', E')$ heißt *Subgraph* bzw. *Teilgraph* von G, geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$.

Falls $G' \neq G$, wird G' auch als *echter* Subgraph von G bezeichnet, in Notation $G' \subset G$. Ein Subgraph G' heißt *induziert*, falls $E' = \{\{v, w\} \in E \mid v, w \in V'\}$. Ein induzierter Subgraph wird auch *Untergraph* genannt und als $G[V']$ notiert.

Definition 1.2.3. Ein *Weg* W ist eine Folge $\{v_0, v_1, \ldots, v_l\}$ von Knoten mit $v_i \in V$ für alle i, so dass für $0 \leq i \leq l-1$ gilt: $\{v_i, v_{i+1}\} \in E$. Die Länge $|W|$ des Weges ist dabei durch die Anzahl der Kanten l gegeben. Ein Weg mit paarweise disjunkten Knoten wird *Pfad* bzw. v_0-v_l-*Pfad* genannt.

Die *inneren* Knoten eines Weges sind die Knoten $v_1, \ldots, v_{l-1}$. Zwei oder mehr Wege heißen *kreuzungsfrei*, wenn ihre inneren Knotenmengen disjunkt sind.

Ein Weg $\{v_0, v_1, \ldots, v_l\}$ heißt *geschlossen*, falls $v_l = v_0$. Ein geschlossener Weg, dessen Knoten $v_0, \ldots, v_{l-1}$ disjunkt sind, heißt auch *Kreis*.

Definition 1.2.4. Ein Graph G wird *azyklisch* genannt, wenn er keine Kreise enthält. Ein azyklischer Graph heißt auch *Wald*. Ein *Baum* ist ein zusammenhängender, azyklischer Graph.

Definition 1.2.5. Ein Graph $G = (V, E)$ heißt für ein $k \in \mathbb{N}$ *k-zusammenhängend*, wenn nach Entfernung beliebiger Knoten V' mit $|V'| < k$ zwischen je zwei verbleibenden Knoten ein Pfad existiert.

Graphe, die 1-zusammenhängend sind, werden auch *zusammenhängend* genannt. Die knotenmaximalen, k-zusammenhängenden Subgraphen von G werden als seine k-fachen *Zusammenhangskomponenten* bzw. *k-Zusammenhangskomponenten* bezeichnet. Die 2-Zusammenhangskomponenten heißen auch *Biconnected Components* bzw. kurz *Bicomps*.

Neu entstandene k-Zusammenhangskomponenten nach dem Löschen einer Menge von Knoten und Kanten werden als *separiert* bezeichnet.

1.3 Eigenschaften planarer Graphen

In diesem Abschnitt werden wesentliche Eigenschaften planarer Graphen zusammengefasst, die im Rahmen der Diplomarbeit von Bedeutung sind.

Ein Graph $G = (V, E)$ ist genau dann *planar*, wenn er ohne Kantenkreuzungen in die Ebene gezeichnet werden kann. Eine *planare Zeichnung* ist dabei eine Abbildung jedes Knotens $v \in V$ auf eine eindeutige Koordinate und jeder Kante $\{a, b\} \in E$ auf eine kontinuierliche, nicht selbstüberschneidende Kurve zwischen den inzidenten Knoten der Ebene.

Die Menge der möglichen planaren Zeichnungen eines Graphen ist unendlich. Allerdings kann von vielen Zeichnungseigenschaften, wie beispielsweise der genauen Darstellung einer Kante als Kurve, abstrahiert werden.

Definition 1.3.1. Zwei planare Zeichnungen Z_1 und Z_2 desselben Graphens G heißen *äquivalent*, in Zeichen $Z_1 \equiv Z_2$, wenn für jeden Knoten die zyklische Reihenfolge der inzidenten Kanten im Uhrzeigersinn identisch ist.

Das folgende Theorem wurde jeweils eigenständig von Wagner 1936 [Wag36] und Fáry 1948 [Fár48] bewiesen.

Theorem 1.3.2 (Wagner / Fáry)**.** *Zu jeder planaren Zeichnung gibt es eine äquivalente gradlinige Zeichnung.*

Somit kann von der speziellen planaren Zeichnung abstrahiert werden und wir können uns für die meisten Eigenschaften planarer Graphen auf die sogenannten *kombinatorischen Einbettungen* zurückziehen.

Definition 1.3.3. Die *kombinatorischen Einbettungen* eines Graphen G sind die Äquivalenzklassen der planaren Zeichnungen von G bezüglich $\equiv$.

Werden alle Knoten und Kanten einer planaren Zeichnung auf der Ebene entfernt, heißen die verbleibenden, zusammenhängenden Gebiete *Faces* oder auch *Flächen*. Die Menge aller Flächen wird mit F bezeichnet. Da der $\mathbb{R}^2$ unendlich ist, wir aber von endlichen Graphen ausgehen, gibt es genau eine unendliche große Fläche, welche als *Außenfläche* bezeichnet wird.

Im Folgenden wird gezeigt, dass die Existenz einer planaren Zeichnung unabhängig von der Wahl der Außenfläche ist.

Definition 1.3.4. Zwei Oberflächen heißen *homöomorph*, wenn es eine bijektive, stetige Funktion f gibt, die die Oberflächen ineinander überführt und dessen Umkehrfunktion f^{-1} ebenfalls stetig ist.

Sei p ein beliebiger Punkt der Kugeloberfläche und $S \setminus p$ die Kugeloberfläche ohne p. Der Punkt p wird im Folgenden auch Nordpol genannt. Dann existiert ein Homöomorphismus zwischen der Ebene und $S \setminus p$. Dabei ist f beispielsweise die stereographische Projektion von Punkten der Ebene auf die Kugeloberfläche (siehe Abbildung 1.2). Dazu wird jeder Punkt $x \in \mathbb{R}^2$ dem Punkt $x' \in S \setminus p$ zugeordnet, der Schnittpunkt von $S \setminus p$ und der Verbindungsgeraden zwischen x und p ist. Bis auf den Nordpol wird damit auf jeden Punkt der Kugeloberfläche eindeutig abgebildet und f ist bijektiv und f^{-1} stetig.

Damit können planare Zeichnungen auf die Kugeloberfläche und umgekehrt projiziert werden. Ein Graph G kann also auch auf Planarität überprüft werden, indem versucht wird, ihn auf die Kugeloberfläche einzubetten. Aus einer solchen Einbettung lässt sich dann durch Anwendung der Umkehrprojektion f^{-1} auf $S \setminus p$ eine planare Zeichnung berechnen. Wird vorher die Kugel auf der Ebene verschoben, ist jeder beliebige Punkt als Nordpol wählbar. Dadurch wird die Fläche der Kugel, die den Nordpol enthält,

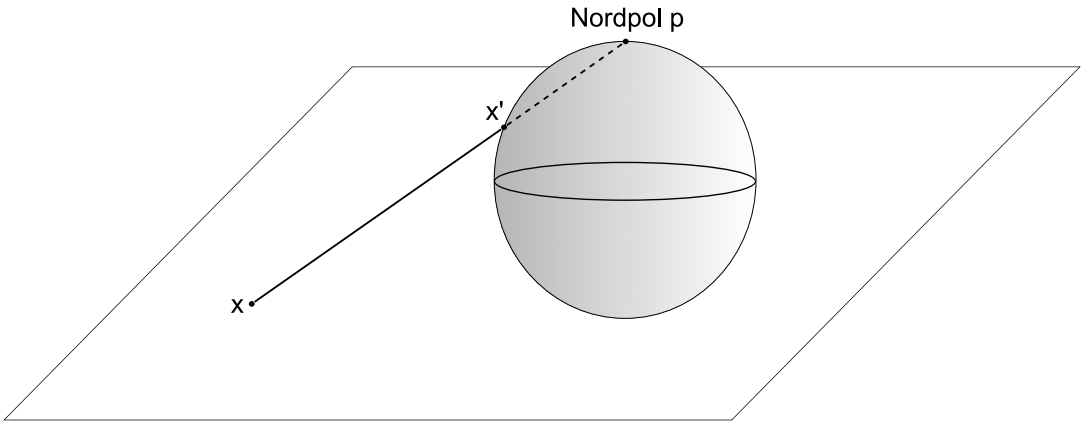


Abbildung 1.2: Stereographische Projektion von der Ebene auf die Kugeloberfläche.

als Außenfläche der planaren Zeichnung gewählt. Da die Reihenfolgen der zu Knoten inzidenten Kanten dabei nicht verändert werden, gilt folgende Aussage.

Proposition 1.3.5. *Zu jeder planaren Zeichnung Z eines Graphen und jeder Fläche i von Z existiert eine planare Zeichnung Z' mit Außenfläche i und $Z \equiv Z'$.*

Somit kann jede beliebige Fläche einer planaren Zeichnung als Außenfläche gewählt werden. Eine kombinatorische Einbettung mit der zusätzlichen Angabe der Außenfläche wird auch *planare Einbettung* genannt.

Konvexe Polyeder sind homöomorph zur Kugeloberfläche. Daher stellen die Skelette konvexer Polyeder eine Teilmenge der planaren Graphen dar. Folgendes Theorem gilt für alle planaren Zeichnungen zusammenhängender Graphen und geht auf die Eulersche Polyederformel zurück, die einen Zusammenhang zwischen den Knoten, Kanten und Faces in konvexen Polyedern beschreibt.

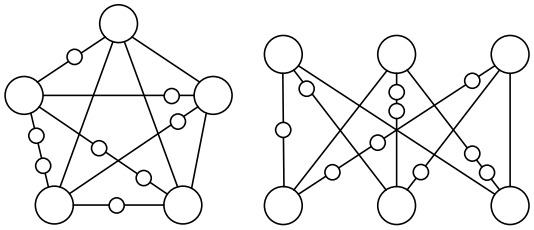
Theorem 1.3.6 (Eulerscher Polyedersatz für planare Graphen)**.** *Sei f die Anzahl der Faces einer planaren Zeichnung eines zusammenhängenden Graphen $G = (V, E)$ mit $n := |V|$ und $m := |E|$. Dann gilt:*

$$n - m + f = 2.$$

Mit dem Polyedersatz lässt sich die Kantenanzahl planarer Graphen linear in der Knotenanzahl beschränken: Für planare Graphen $G = (V, E)$ mit $V(G) \geq 3$ gilt $m \leq 3n - 6$. Falls G keine Kreise der Länge 4 enthält, gilt für $V(G) \geq 4$ sogar $m \leq 2n - 4$. Aus diesen beiden oberen Schranken lässt sich folgern, dass der $K_{3,3}$ und der K_5 nicht planar sein können.

Definition 1.3.7. Sei der Graph $H = (V, E)$ gegeben. Eine *Subdivision* von H (bzw. H-Subdivision) ist ein Graph, der aus H entsteht, indem alle Kanten durch Pfade der Mindestlänge 1 ersetzt werden.

Da der $K_{3,3}$ und der K_5 nicht planar sind, können deren Subdivisions (siehe Abbildung 1.3) nicht planar sein. Somit ist auch ein Graph, der $K_{3,3}$-Subdivisions oder K_5-Subdivisions enthält, nicht planar. Kuratowski bewies 1930 in [Kur30], dass auch die Umkehrung dieser Aussage gilt:



(a) K_5-Subdivision (b) $K_{3,3}$-Subdivision

Abbildung 1.3: Subdivisions des K_5 und des $K_{3,3}$.

Theorem 1.3.8 (Kuratowski)**.** *Ein Graph ist genau dann planar, wenn er weder eine Subdivision des $K_{3,3}$ noch des K_5 enthält.*

Die Graphen $K_{3,3}$ und K_5 werden deswegen auch als die *Kuratowski-Graphen* bezeichnet. Mit dem *Kontrahieren* einer Kante $e = v, w \in E$ eines Graphen $G = (V, E)$ wird das Löschen von e mit anschließendem Verschmelzen von v und w zu einem einzigen Knoten bezeichnet.

Definition 1.3.9. Sei $G = (V, E)$ ein Graph und G' der resultierende Graph nach einer beliebigen Folge von Kantenkontraktionen auf G. Ein Subgraph von G' heißt dann auch *Minor* von G.

Zwischen den Minoren eines Graphen G und den Graphen, deren Subdivisions in G enthalten sind, besteht eine enge Beziehung. Falls G einen $K_{3,3}$-Minor besitzt, ist in G auch eine $K_{3,3}$-Subdivision enthalten. Falls G einen K_5-Minor besitzt, enthält G entweder eine $K_{3,3}$- oder eine K_5-Subdivision. Umgekehrt gilt: Enthält G eine H-Subdivision, ist H auch ein Minor von G.

Wagner fand 1937 die folgende zum Theorem von Kuratowski alternative Formulierung planarer Graphen [Wag37]:

Theorem 1.3.10 (Wagner)**.** *Ein Graph ist genau dann planar, wenn er weder den $K_{3,3}$ noch den K_5 als Minor enthält.*

Das folgende Theorem ermöglicht es, die Planarität von Graphen nur in Abhängigkeit von den enthaltenen Zweizusammenhangskomponenten zu berechnen. Ein Beweis findet sich beispielsweise in [EHK+96].

Theorem 1.3.11. *Ein Graph ist genau dann planar, wenn seine Zweizusammenhangskomponenten planar sind.*

1.4 Planaritätstests

Ziel eines Planaritätstests ist es, einen gegebenen Graphen $G = (V, E)$ automatisiert auf Planarität zu testen und falls G planar ist, eine kombinatorische oder planare Einbettung zu liefern. Andernfalls ist G nicht planar und es soll eine nach dem Theorem von Kuratowski (1.3.8) existierende Subdivision des $K_{3,3}$ oder des K_5 ausgegeben werden.

Aus einer gegebenen kombinatorischen Einbettung lässt sich, beispielsweise mit dem Algorithmus von Read [Rea87], in $O(n)$ eine zugehörige planare Zeichnung berechnen. Zusätzlich kann dabei, im Algorithmus von Read durch die Erweiterung von Kocay und Pantel [KP95], die gewünschte Außenfläche gewählt werden.

Für den Test auf Planarität ist es mit den Ergebnissen des letzten Abschnitts möglich, sich auf zweizusammenhängende Graphen zurückzuziehen, die höchstens $3n - 5$ Kanten haben. Auch Graphen mit höchstens vier Knoten müssen nicht betrachtet werden, da sie wegen der geringen Knotenanzahl keine Kuratowski-Subdivision enthalten können und deswegen planar sind. Planaritätstests für einfache Graphen können ohne weiteres auf Multigraphen erweitert werden, indem vor dem Test alle vorhandenen Mehrfachkanten zu einer einzelnen Kante gebündelt werden. Potentiell vorhandene Self-Loops $\{v, v\} \in E$ werden vorher gelöscht und für eine kombinatorische Einbettung hinterher so eingefügt, dass die beiden inzidenten Kantenabschnitte in der zyklischen Adjazenzliste von v benachbart sind.

1.4.1 Historie

Die beiden Charakterisierungen planarer Graphen von Kuratowski und Wagner bieten leider keine einfache Möglichkeit, Graphen automatisiert auf Planarität zu testen. Eine mögliche Strategie ist, sukzessive durch Hinzunahme von Knoten und Kanten eine planare Zeichnung aufzubauen. Falls der gesamte Graph so eingebettet werden kann, muss dieser planar sein.

Auslander und Parter [AP61] schlugen 1961 einen Algorithmus vor, der einen Kreis C im zweizusammenhängenden Graphen sucht, diesen entfernt und die verbleibenden Teile des Graphen rekursiv weiter zerlegt. Die Einbettung des Graphen wird dann aus den Einbettungen der Teilgraphen unter Hinzunahme des Kreises C berechnet. Allerdings hatte der Algorithmus einen Fehler, der ihn nicht terminieren ließ. Dieser konnte aber 1963 von Goldstein [Gol63] behoben werden. Der so erweiterte Algorithmus wurde 1969 von Shirey [Shi69] in kubischer Laufzeit $O(n^3)$ implementiert.

Daraufhin wurde versucht, die Komplexität zu vermindern. Ein Planaritätstest mit Laufzeit $O(n^2)$ wurde 1964 von Demoucron, Malgrange und Pertuiset [DMP64] entworfen. Dieser gilt bis heute als einer der zugänglichsten Planaritätstests.

Lempel, Even und Cederbaum publizierten 1967 ein alternatives Verfahren, das mit der Einbettung eines Knotens beginnt und dann sukzessive Knoten mit deren inzidenten Kanten hinzufügt [LEC67]. Dieses manchmal auch *vertex addition method* genannte Verfahren hatte ebenfalls eine Laufzeit von $O(n^2)$.

Jahr	Algorithmus	Idee	Laufzeit	Literatur
1961	Auslander und Parter, korrigiert von Goldstein		$O(n^3)$	[AP61] [Gol63] [Shi69]
1964	Demoucron, Malgrange und Pertuiset		$O(n^2)$	[DMP64]
1967	Lempel, Even und Cederbaum		$O(n^2)$	[LEC67]
1974	Hopcroft und Tarjan (erster linearer Test)	APG	$O(n)$	[HT74] [Mut92] [MM96]
1976	Booth und Lueker (benutzt PQ-Trees)	LEC	$O(n)$	[BL76] [ET76] [CNAO85]
$\geq$ 1985	de Fraysseix, de Mendez und Rosenstiehl		$O(n)$	[FMR06, FR85]
1993	Shih und Hsu	LEC	$O(n)$	[SH93, SH99]
2004	Boyer und Myrvold	LEC	$O(n)$	[BM04] [BM99] [BM02] [BFN03]

Tabelle 1.1: Auflistung ausgewählter Planaritätstests. Die Spalte *Idee* gibt dabei den zugrunde liegenden Planaritätstest an. Die Abkürzung *APG* weist auf den Algorithmus von Auslander, Parter und Goldstein hin, *LEC* auf den Algorithmus von Lempel, Even und Cederbaum.

Basierend auf den Ideen von Auslander, Parter und Goldstein gelang es Hopcroft und Tarjan 1974 den ersten Planaritätstest in $O(n)$ zu entwerfen [HT74]. Grundlage ist dabei eine Depth-First-Search-Traversierung (auch DFS-Traversierung genannt, siehe beispielsweise die Beschreibung in [FMR06]). Allerdings gibt der Algorithmus lediglich an, ob der Eingabegraph planar ist, eine planare Einbettung wird im ersten Fall nicht gefunden.

Im Jahr 1976 wurden mehrere Resultate erbracht, die den Lempel-Even-Cederbaum-Algorithmus, kurz LEC-Algorithmus, entscheidend verbesserten. Even und Tarjan zeigten, dass die sogenannte *st*-Nummerierung in linearer Zeit berechnet werden kann [ET76]. Zusätzlich führten Booth und Lueker die Datenstruktur des PQ-Trees ein [BL76], mit der effizient Informationen über mögliche Permutationen von Teilgraphen in einer Einbettung verwaltet werden können. Beide Resultate können in den ursprünglichen LEC-Algorithmus so eingesetzt werden, dass dieser die Laufzeit $O(n)$ erreicht. Allerdings wird auch hier eine entsprechende planare Einbettung nicht in Linearzeit gefunden. Dieses Einbettungsproblem wurde erst 1985 von Chiba, Nishizeki, Abe und Ozawa gelöst [CNAO85].

Mehlhorn und Mutzel [MM96, Mut92] erweiterten und implementierten den Hopcroft-Tarjan-Algorithmus 1996 als Teil der LEDA-Bibliothek. Dabei zeigten sie, dass auch bei diesem eine planare Einbettung in $O(n)$ gefunden werden kann.

Die Planaritätstests waren aber immer noch schwierig zu implementieren. In den letzten 15 Jahren wurde deswegen fortwährend versucht, einfachere Planaritätstests zu entwi-

ckeln. Im Wesentlichen haben sich dabei zwei Ansätze etabliert, die gleichzeitig nach einem Vergleich von Boyer, Cortese, Patrignani und Di Battista in [BCPB03] die momentan schnellsten darstellen:

- Der erste Ansatz geht auf eine alternative Charakterisierung planarer Graphen in Form von Backedge-Konflikten einer DFS-Traversierung zurück, die 1985 von de Fraysseix und Rosenstiehl veröffentlich wurde [FR85]. In mehreren daraufhin folgenden Veröffentlichungen von de Fraysseix, de Mendez und Rosenstiehl wurde aus dieser Charakterisierung ein Planaritätstest in Linearzeit entwickelt [FMR06].
- Der zweite Ansatz basiert auf dem LEC-Algorithmus und vermeidet die komplizierten Anwendungen der PQ-Tree-Datenstruktur. Auch dieser Ansatz wurde, ausgehend von [SH93], in mehreren Veröffentlichungen durch Boyer, Hsu, Myrvold und Shih [SH99, BM99, BM02, BFN03] weiterentwickelt. Boyer und Myrvold beschrieben dann 2004 einen vereinfachten Planaritätstest, der sukzessive Kanten zu einer bestehenden Einbettung hinzufügt [BM04]. Dieser erreicht Laufzeit $O(n)$ und extrahiert im planaren Fall eine planare Einbettung und andernfalls eine Kuratowski-Subdivision.

Eine Auflistung der obigen Auswahl von Planaritätstests findet sich in Tabelle 1.1. Allerdings haben alle genannten Tests gemeinsam, dass bei nicht planaren Eingabegraphen höchstens eine Kuratowski-Subdivision extrahiert wird. Bis zu dem Zeitpunkt der Erstellung dieser Arbeit haben wir trotz intensiver Literaturrecherche keine Kenntnis von einer Methode, die $\omega(1)$ viele Kuratowski-Subdivisions in linearer Zeit extrahieren kann. Da in dieser Arbeit eine solche Methode vorgestellt wird, die auf dem Planaritätstest von Boyer und Myrvold basiert, wird auf diesen kurz eingegangen.

1.5 Der Boyer-Myrvold-Algorithmus

Der Boyer-Myrvold-Algorithmus ist in der Praxis einer der schnellsten Planaritätstests und erreicht Linearzeit. In dem Vergleich von Boyer, Cortese, Patrignani und Di Battista in [BCPB03] hat lediglich der Algorithmus von de Fraysseix, de Mendez und Rosenstiehl eine vergleichbar kleine Konstante in der Laufzeit. Ein Vorteil des Boyer-Myrvold-Algorithmus ist zusätzlich, dass der Eingabegraph anfangs weder in seine Zweizusammenhangskomponenten, noch in seine einfachen Zusammenhangskomponenten zerlegt werden muss. Nützlich ist auch die im Vergleich zu anderen linearen Planaritätstests einfachere Implementierung.

Daher liegt die Idee nahe, den Boyer-Myrvold-Algorithmus als Ausgangspunkt für mögliche Erweiterungen zu nutzen, mit denen effizient mehrere Kuratowski-Subdivisions extrahiert werden können. Im Folgenden wird der Planaritätstest deswegen kurz skizziert. Eine erschöpfende Behandlung auf wenigen Seiten ist trotz aller Versuche, Planaritätstests zu vereinfachen, bisher nicht möglich. Für eine vollständige Beschreibung wird auf den Originalartikel [BM04] verwiesen.

Der Boyer-Myrvold-Algorithmus basiert auf einer DFS-Traversierung. Sei $G = (V, E)$ der

Eingabegraph mit $m \leq 3n - 5$ Kanten, beispielsweise der Graph in Abbildung 1.4(a). Ein Knoten $v \in V$ erhält den Depth-First-Index (DFI-Wert) i, wenn er als i-ter Knoten während der DFS-Traversierung erreicht wird. Die Traversierung spannt einen Wald von *DFS-Tree-Kanten* auf, der aus genauso vielen Bäumen besteht, wie G Zusammenhangskomponenten besitzt (siehe Abbildung 1.4(b)). Die restlichen Kanten heißen *Backedges* und führen von je einem Knoten mit höherem DFI-Wert zu einem Knoten mit kleinerem DFI-Wert.

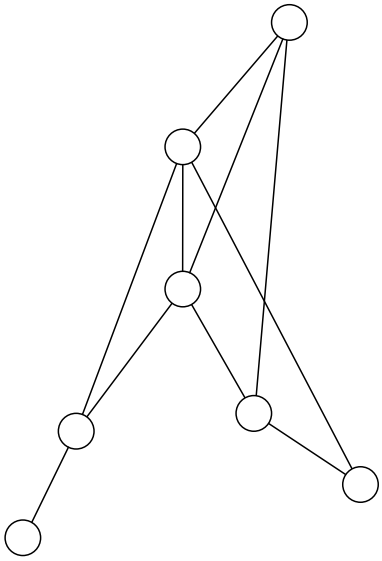
In einem Preprocessing-Schritt werden vorab die *LeastAncestors* und *LowPoints* berechnet. Der *LeastAncestor* eines Knotens v ist der Knoten w mit niedrigstem DFI-Wert, so dass $\{v, w\}$ eine Backedge ist. Der *LowPoint* eines Knotens v ist der DFS-Vorgänger w mit niedrigstem DFI-Wert, der durch einen Pfad $v \to u$ aus ausschließlich absteigenden DFS-Tree-Kanten und einer zusätzlichen Backedge $\{u, w\}$ erreicht werden kann. Falls kein solcher Vorgänger existiert, ist der Lowpoint v selbst.

Jede DFS-Tree-Kante $e = \{v, w\} \in E$ wird jetzt von ihrem Startknoten v getrennt und stattdessen an einen neuen *virtuellen* Knoten v' gehängt (siehe Abbildung 1.4(c)). Die so entstandenen Komponenten $G[e]$ sind Bicomps. Eine solche Bicomp, die lediglich aus einer Kante und ihren inzidenten Knoten besteht, heißt auch *entartet*. Für jeden Knoten v wird eine *SeparatedDFSChild*-Liste angelegt. Diese besteht aus allen DFS-Nachfolgern von v, die in unterschiedlichen Bicomps liegen und enthält deswegen anfangs sämtliche DFS-Nachfolger.

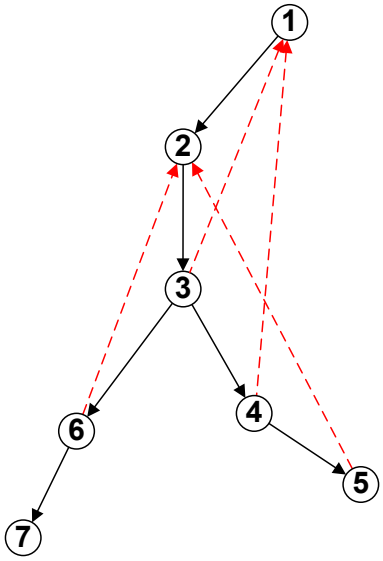
Die Bicomps stellen die eingebetteten Teile des Graphen dar. Das Ziel ist, diese so zu verschmelzen, dass der Originalgraph wiederhergestellt wird und die Kantenreihenfolgen der Knoten eine planare Einbettung erzeugen, falls G planar ist. Dabei werden die anfangs entarteten Bicomps im Verlauf des Algorithmus größer, verlieren aber nie den Zweizusammenhang. Innere Knoten dieser Bicomps sind für folgende Einbettungen nicht interessant, so dass alle relevanten Informationen an den Außenflächen der Bicomps liegen. Um diese traversieren zu können, werden an jedem an die Außenfläche grenzenden Knoten zwei sogenannte *ExternalLinks* zu den jeweiligen Nachbarknoten verwaltet.

Der Algorithmus versucht, nacheinander jeden Knoten in absteigender DFI-Reihenfolge einzubetten. Dabei wird bei dem Knoten mit höchstem DFI-Wert begonnen und jeweils geprüft, ob Backedges an diesem Knoten enden, die dann als *pertinente Backedges* bezeichnet werden. Ist das nicht der Fall, muss auch keine Kante eingebettet werden. Andernfalls wird jede pertinente Backedge so eingebettet, dass sie entweder Bicomps verbindet und somit aus diesen eine neue, größere Bicomp erzeugt, oder eine schon bestehende Bicomp vergrößert.

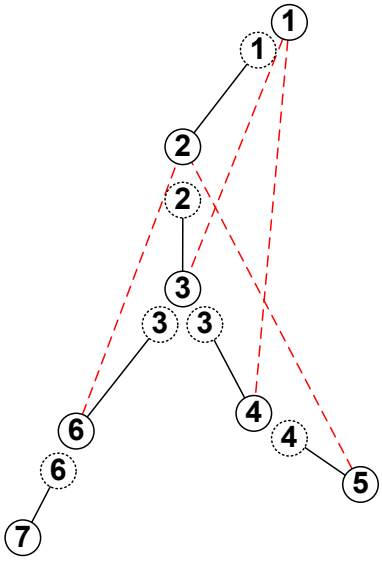
Um die Menge der dabei involvierten Bicomps zu ermitteln, wird der sogenannte *Walkup* ausgeführt. Der Walkup beginnt an dem Startknoten jeder pertinenten Backedge. Sei w einer dieser Startknoten. Ab w existiert ein eindeutiger DFS-Pfad $w \to v$ und damit ein eindeutiger Pfad P von aneinander hängenden Bicomps, der diesen enthält. Der Walkup durchläuft ab w jede Bicomp aus P parallel an der Außenfläche, bis er deren Wurzel und damit die nächste Bicomp oder v findet. Jeder auf dem Weg zur Wurzel besuchte Knoten wird dabei speziell markiert. Falls in einem der folgenden Walkups an v ein



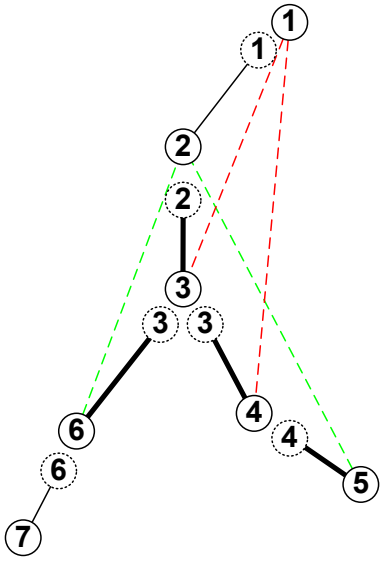
(a) Der Eingabegraph G.



(b) Die DFS-Tree-Kanten (durchgezogen markiert) und Backedges (gestrichelt markiert) nach der DFS-Traversierung.



(c) Aufteilung in entartete Bicomps. Die virtuellen Knoten sind gestrichelt umrandet.



(d) Start der Einbettungsphase des Knotens 2: Kennzeichnung der einzubettenden Bicomps (fett markiert) durch den Walkup auf beiden pertinenten Backedges.

Abbildung 1.4: Einbettungsschritte an einem Beispielgraphen, Teil 1

solcher markierter Knoten gefunden wird, bricht der Walkup dort ab und fährt mit der nächsten pertinenten Backedge fort.

Der vom Walkup traversierte Pfad einer pertinenten Backedge wird auch als deren *Backedgepfad* bezeichnet. Während des Walkups wird an jedem besuchten Startknoten einer Backedge ein *BackedgeFlag* gesetzt, um die direkte Verbindung mit v zu kennzeichnen. Zusätzlich werden für jeden Knoten alle anhängenden, durchlaufenen Bicomps in der sogenannten *PertinentRoots*-Liste festgehalten, um alle nachfolgenden Bicomps später effizient erreichen zu können.

In dem Beispiel in Abbildung 1.4(d) enden an den Knoten $6, \ldots, 3$ keine Backedges. An dem Knoten 2 traversiert der Walkup den Backedgepfad $6 \to 3 \to 2$ der ersten pertinenten Backedge. Der zweite Backedgepfad $5 \to 4 \to 3$ stoppt an dem vorher besuchten Knoten 3. Die dabei besuchten Bicomps sind Teil der weiteren Berechnungschritte, um diese beiden Backedges einzubetten. Dafür müssen die besuchten Knoten zuerst klassifiziert werden. Eine Bicomp mit virtuellem Wurzelknoten v' heißt *Kinderbicomp* eines Knotens v, falls v der auch *real* genannte Knoten ist, von dem v' anfangs abgespalten wurde.

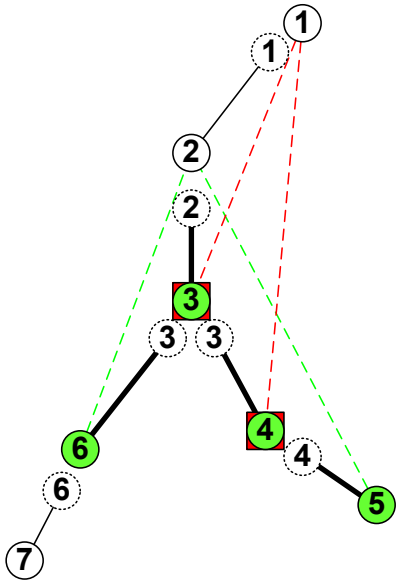
Definition 1.5.1. Sei v der aktuell einzubettende Knoten. Ein Knoten w wird rekursiv als *pertinent* definiert, falls die Backedge $\{w, v\}$ existiert oder ein Knoten einer Kinderbicomp von w pertinent ist.

Definition 1.5.2. Sei v der aktuell einzubettende Knoten. Ein Knoten w wird rekursiv als *extern aktiv* definiert, falls eine Backedge $\{w, u\}$ zu einem Knoten u mit kleinerem DFI-Wert als v existiert oder ein Knoten einer Kinderbicomp von w extern aktiv ist.

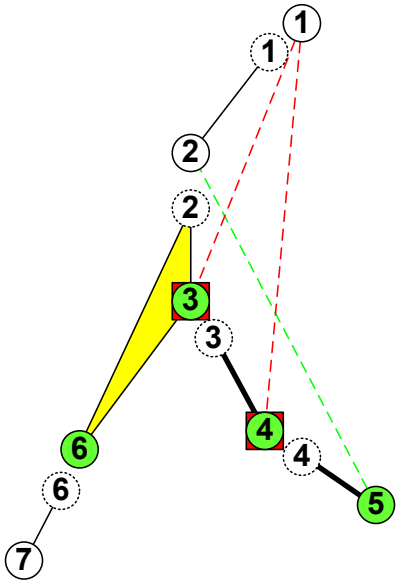
Backedges, die an einem Knoten mit kleinerem DFI-Wert als v enden, werden auch *externe Backedges* genannt. Eine Bicomp heißt *pertinent* bzw. *extern*, wenn sie mindestens einen pertinenten bzw. extern aktiven Knoten enthält. Jeder Knoten kann effizient und dynamisch in konstanter Zeit auf Pertinenz und externe Aktivität getestet werden, indem die Informationen der LeastAncestors, LowPoints, BackedgeFlags und der *PertinentRoots*- und *SeparatedDFSChild*-Listen genutzt werden. Beispielsweise sind in Abbildung 1.5(a) die Knoten $3, \ldots, 6$ pertinent und die Knoten 3 und 4 zusätzlich extern aktiv. Ein Knoten heißt *aktiv*, wenn er pertinent oder extern aktiv ist, andernfalls wird er als *inaktiv* bezeichnet.

Während extern aktive Knoten auf den Außenflächen der Bicomps gehalten werden müssen, da deren externe Backedges sonst Kreuzungen verursachen würden, werden pertinente Knoten, möglicherweise durch Absteigen in nachfolgende Bicomps, eingebettet. Die Schwierigkeit liegt darin, die Reihenfolge dieser Einbettungen und jeweils die Einfügepositionen in den Adjazenzlisten von v und w zu bestimmen. Dieses Problem löst der *Walkdown*, der einmal gegen den Uhrzeigersinn (CCW) und einmal mit dem Uhrzeigersinn (CW) ausgeführt wird.

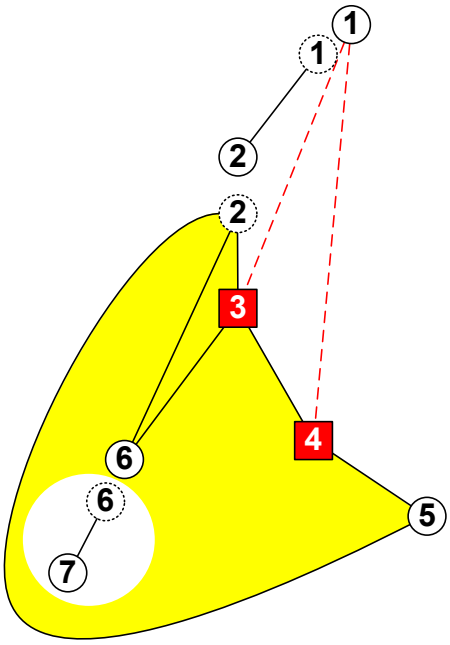
Der Walkdown wird für jede Kinderbicomp von v einzeln ausgeführt und traversiert diese in der jeweils eingeschlagenen Richtung auf der Außenfläche. Sei w der aktuelle Knoten einer solchen Traversierung. Dann werden folgende Regeln für sowohl den CCW-Walkdown als auch für den CW-Walkdown mit absteigender Priorität befolgt:



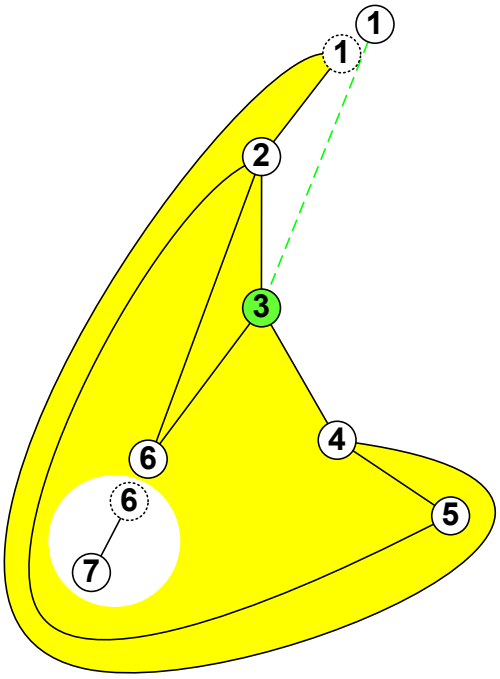
(a) Klassifizierung in pertinente (grün, rund) und extern aktive (rot, quadratisch) Knoten.



(b) Der Walkdown an Knoten 2 nach erfolgter Einbettung der ersten pertinenten Backedge $\{6, 2\}$.



(c) Der Walkdown an Knoten 2 nach Einbettung beider pertinenten Backedges.



(d) Der Walkdown an Knoten 1 nach Einbettung von $\{4, 1\}$.

Abbildung 1.5: Einbettungsschritte an einem Beispielgraphen, Teil 2

- Falls die Backedge $\{w, v\}$ existiert, bette diese so ein, dass sie mit dem bisher besuchten Pfad eine neue Face bildet. Verschmelze dabei alle Bicomps von w bis v.
- Falls eine pertinente Kinderbicomp an w existiert, steige zu dieser ab und traversiere in die Richtung weiter, in der ein erster pertinenter Knoten liegt. Falls dieser wegen zweier externer Knoten nicht erreicht werden kann, breche den Walkdown ab. Existieren mehrere pertinente Kinderbicomps, wird eine nicht externe gewählt, falls vorhanden.
- Falls w extern ist, stoppe den Walkdown, da w nicht mehr pertinent sein kann.
- Falls $w = v$, breche ab. Sonst traversiere die Außenfläche weiter in die aktuelle Richtung.

In Abbildung 1.5(a) beginnt der CCW-Walkdown an dem Knoten $v := 2$ und steigt sofort zu dessem virtuellen Knoten ab. Die Außenfläche der dortigen, entarteten Bicomp wird bis zum Knoten 3 traversiert, an dem die nicht externe Kinderbicomp $G[\{3, 6\}]$ präferiert wird. An Knoten 6 wird eine pertinente Backedge gefunden, die eingebettet wird und die neue Face $2, 3, 6$ bildet. Dabei werden die beiden besuchten Bicomps am Knoten 3 verschmolzen (siehe Abbildung 1.5(b)). Auf der so vergrößerten Bicomp wird weiter in Richtung CCW traversiert, bis am externen Knoten 3 bis zu Knoten 5 abgestiegen wird, an dem die verbleibende pertinente Backedge eingebettet wird (siehe Abbildung 1.5(c)). Der CW-Walkdown findet danach keine pertinenten Knoten mehr und bricht an v ab.

In der nächsten Einbettungsphase an Knoten 1 bettet der Walkdown die Backedge $\{4, 1\}$ ein (siehe Abbildung 1.5(d)), gefolgt von der Backedge $\{3, 1\}$ (siehe Abbildung 1.6(a)). Allerdings existieren danach noch mit $G[\{1\}]$ und $G[\{6, 7\}]$ separierte Komponenten. Diese werden in einem Postprocessingschritt mit den übrigen Komponenten verschmolzen (siehe Abbildung 1.6(b)). Der Beispielgraph G kann also vollständig eingebettet werden und ist damit planar.

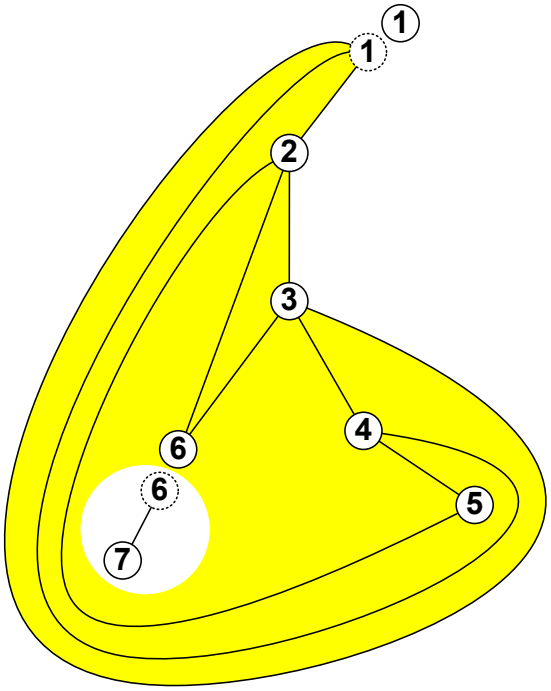
Ein komplexeres Beispiel einer Einbettungsphase zeigen die Abbildungen 1.7(a) und 1.7(b).

Nicht beschrieben werden hier die weiterführenden Konzepte des *Flippens* von Bicomps und der *ShortCircuit*-Kanten. Das Flippen von Bicomps spiegelt diese effizient vor dem Verschmelzen, falls das für die planare Einbettung nötig ist. Die ShortCircuit-Kanten werden während des Walkdowns zwischen dem einzubettenden und externen Knoten künstlich eingefügt, um eine lineare Laufzeit des Walkdowns zu ermöglichen.

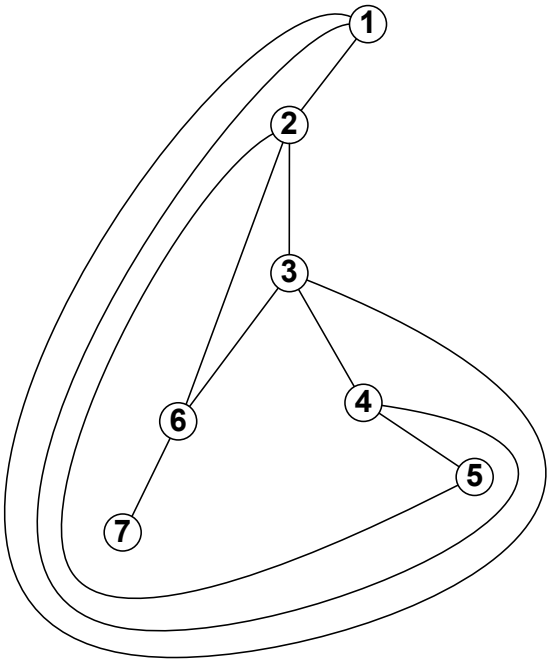
1.5.1 Stoppkonfigurationen

Bisher wurde nicht geklärt, woran nicht planare Graphen erkannt werden können. Das wird durch folgendes Theorem geklärt. Ein Beweis findet sich in [BM04].

Theorem 1.5.3 (Boyer-Myrvold)**.** *Ein Graph ist genau dann planar, wenn der Walkdown in jeder Einbettungsphase sämtliche pertinenten Backedges einbettet.*

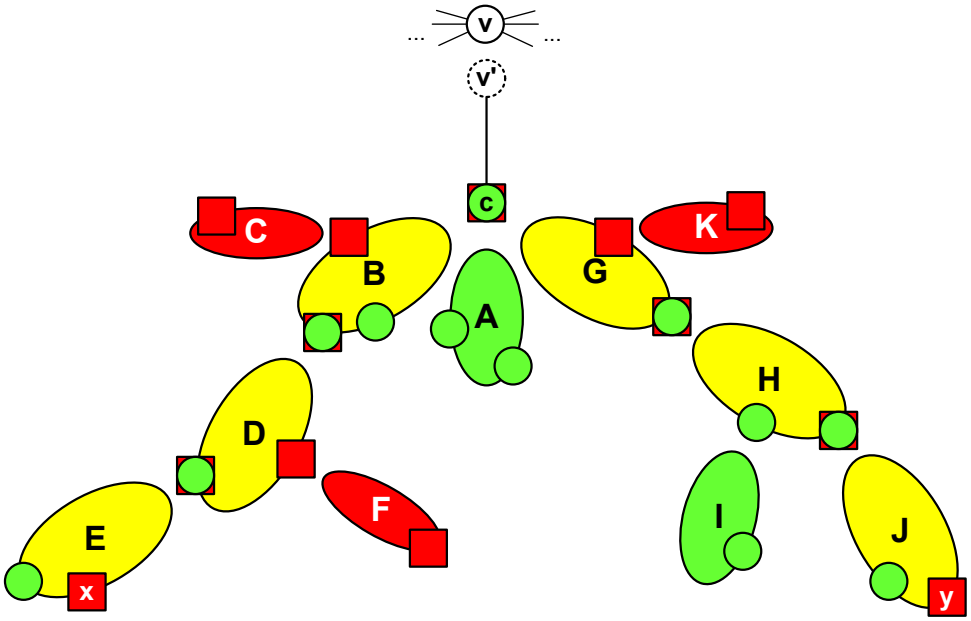


(a) Der vollständig eingebettete Graph.

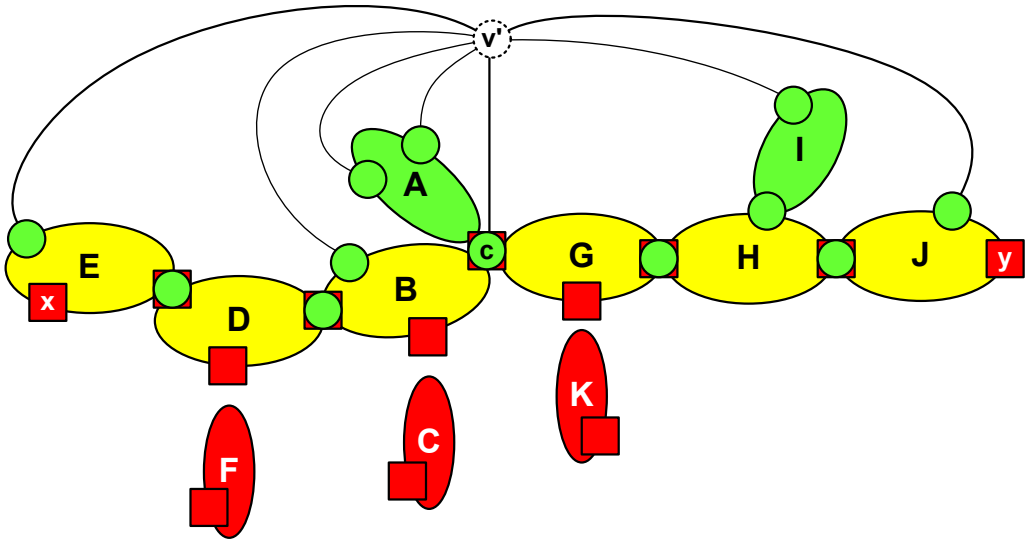


(b) Der Graph G nach Verschmelzen der separierten Komponenten.

Abbildung 1.6: Einbettungsschritte an einem Beispielgraphen, Teil 3



(a) Vor dem Walkdown am Knoten v: Ausschließlich pertinente Bicomps sind grün gezeichnet, ausschließlich externe rot. Alle Bicomps hängen aneinander. Auf die Darstellung der einzelnen virtuellen Knoten wurde aus Übersichtlichkeitsgründen verzichtet.



(b) Der eingebettete Graph nach dem Walkdown. Der CCW-Walkdown stoppt am externen Knoten x, nachdem er die Bicomps A, B, D und E in dieser Reihenfolge an der Außenfläche traversiert und eingebettet hat. Der CW-Walkdown bettet die Bicomps G, H, I und J ein und stoppt dann am Knoten y. Sämtliche Bicomps, bis auf die ausschließlich externen, werden dabei miteinander verschmolzen. Zusätzlich werden alle extern aktiven Knoten an der Außenfläche der neu entstehenden Bicomp platziert.

Abbildung 1.7: Komplexeres Beispiel einer Einbettungsphase.

In einem nicht planaren Graphen existieren also nach einer Einbettungsphase nicht-eingebettete Backedges. Solche Backedges werden auch als *kritisch* bezeichnet. Eine kritische Backedge kann nicht eingebettet werden, da der Walkdown vorher aufgrund zweier externer Knoten stoppt. Diese Knoten können nicht pertinent sein und werden als *Stoppknoten* einer Einbettungsphase, bzw. einzeln als *stopX* und *stopY*, bezeichnet. Eine *Stoppkonfiguration* besteht aus zwei Stoppknoten und einer kritischen Backedge, die durch diese Stoppknoten nicht eingebettet werden konnte. Jeder pertinente, wegen der Stoppkonfiguration nicht erreichte Knoten heißt *kritisch*.

Die Stoppknoten sind Bestandteil einer einzigen Bicomp B. An der Außenfläche von B muss dann ein kritischer Knoten existieren, sonst wäre B nicht pertinent gewesen. Je nach Struktur und Lage von B werden fünf mögliche Kuratowski-Minortypen unterschieden, die eine Stoppkonfiguration verursachen können. Die Minoren der Typen A, B, C und D (siehe Abbildung 1.8) sind dabei jeweils $K_{3,3}$-Minoren, während ein Minor vom Typ E (siehe Abbildung 1.8(e)) ein K_5-Minor ist. Dieser kann wiederum in die fünf Untertypen E_1–E_5 (siehe Abbildung 1.9) aufgeteilt werden, von denen lediglich der letzte auch eine K_5-Subdivision darstellt. Alle anderen Minortypen stellen $K_{3,3}$-Subdivisions dar.

Findet der Boyer-Myrvold-Algorithmus in einem Graphen G eine Stoppkonfiguration, so muss mindestens einer dieser Minortypen in G enthalten sein. Aus diesem wird dann die entsprechende Kuratowski-Subdivision extrahiert und ausgegeben.

1.5.2 Laufzeit

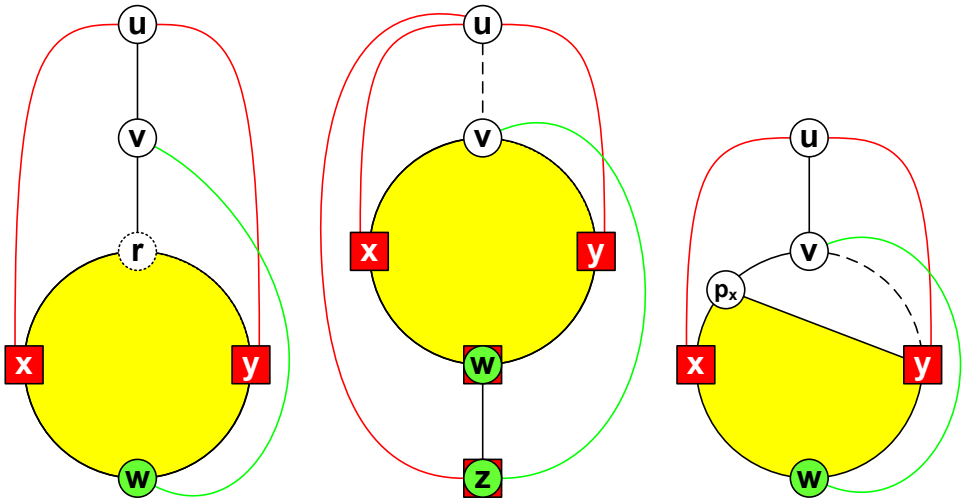
Der originale Boyer-Myrvold-Algorithmus erreicht lineare Laufzeit.

Theorem 1.5.4 (Boyer-Myrvold)**.** *Der Boyer-Myrvold-Algorithmus testet einen Graphen $G = (V, E)$ mit $n = |V|$ und $m = |E|$ in Laufzeit $O(n)$ auf Planarität.*

Beweis. Die Berechnung des DFS-Baums sowie aller LeastAncestors, LowPoints und *SeparatedDFSChild*-Listen im Preprocessingschritt kostet insgesamt lineare Zeit $O(n + m)$. Da nach dem Eulerschen Polyedersatz (1.3.6) nur maximal $m' := 3n - 5$ Kanten auf Planarität getestet werden müssen, ist die Laufzeit hierfür sogar $O(n)$. Alle im Walkup durchlaufenen Backedgepfade, deren Backedges später im Walkdown eingebettet werden, können amortisiert durch die dort entstehenden Faces begrenzt werden. Die Kosten der Walkups sind dabei nicht größer als die Kantensumme dieser entstehenden Faces, da im Walkup durch das parallele Traversieren immer die kürzere Außenfläche der Bicomps gewählt wird. Die Kantensummen aller Faces sind durch $2m'$ und damit durch $O(n)$ beschränkt.

Pertinente Knoten können durch die während des Walkups erstellten BackedgeFlags und *PertinentRoots*-Listen in konstanter Zeit ermittelt werden, extern aktive Knoten durch die LeastAncestors, LowPoints und die *SeparatedDFSChild*-Listen. Falls nötig, können Bicomps im Verlauf des Algorithmus in $O(1)$ geflippt werden.

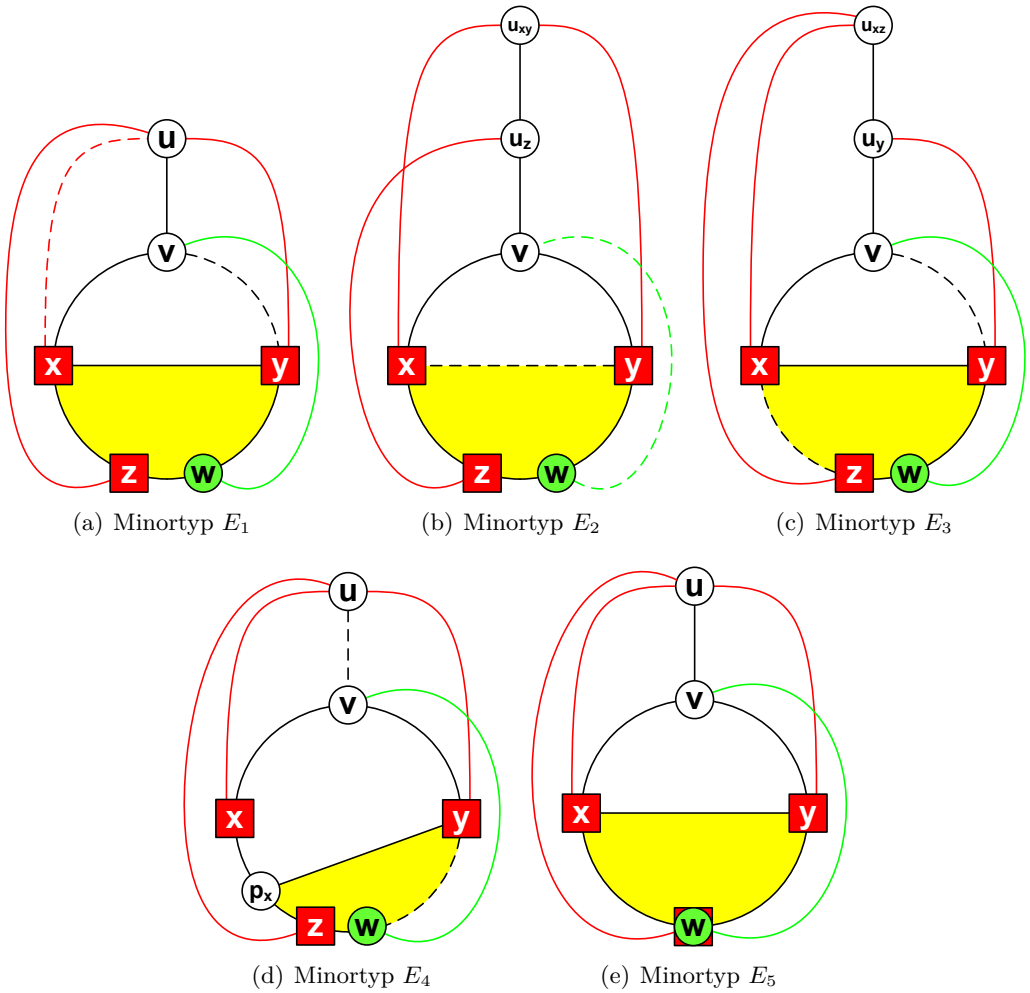
Die Kosten eines Walkdowns können nach der Einbettung einer Backedge analog zu den Kosten des Walkups abgeschätzt werden. Allerdings tritt hier das Problem auf, dass jede



(a) Minortyp A (b) Minortyp B (c) Minortyp C

(d) Minortyp D (e) Minortyp E

Abbildung 1.8: Klassifikation der Minortypen, die in einer Bicomp mit Wurzel v auftreten können. Die gestrichelten Kanten werden für die Extraktion der entsprechenden Kuratowski-Subdivisions nicht benötigt. Minortyp E wird in Abbildung 1.9 in weitere fünf Untertypen unterteilt.



(a) Minortyp E_1 (b) Minortyp E_2 (c) Minortyp E_3

(d) Minortyp E_4 (e) Minortyp E_5

Abbildung 1.9: Die Unterteilung des Minortyps E in seine fünf verschiedenen Untertypen E_1–E_5.

Bicomp einmal im Uhrzeigersinn und einmal gegen den Uhrzeigersinn traversiert wird. Nur jeweils eine Richtung wird hinterher durch die Einbettung einer Backedge abgedeckt. Die Traversierung der anderen Richtung stößt entweder auf einen externen oder pertinenten, in jedem Falle aber aktiven Knoten. Falls in früheren Einbettungsphasen von dem einzubettenden Knoten bis zu jedem Stoppknoten ShortCircuit-Kanten eingefügt worden sind, kann dieser aktive Knoten in $O(1)$ gefunden werden. Damit kann der vorher erfolgte Mehraufwand amortisiert werden.

Die Anzahl der ShortCircuit-Kanten ist mit $O(2n)$ linear begrenzt. Das Einbetten von Backedges ist in konstanter Zeit möglich, das Verschmelzen von zwei Bicomps ebenfalls. Der Test auf nicht eingebettete, pertinente Backedges nach dem Walkup ist für jeden Knoten v in Zeit $degree(v)$ möglich, insgesamt ergibt sich dadurch die Gesamtlaufzeit von $O(n)$. □

2 Ein linearer Algorithmus zur Extraktion von Kuratowski-Subdivisions

Aus einem Graphen $G = (V, E)$ sollen möglichst viele Kuratowski-Subdivisions extrahiert werden. Dafür wird der Planaritätstest von Boyer und Myrvold [BM04] erweitert. Die Anzahl der Kanten m des Graphen darf dabei nicht mehr, wie bei Planaritätstests allgemein üblich, linear in der Knotenanzahl beschränkt werden, da damit die Menge der Kuratowski-Subdivisions im Allgemeinen massiv eingeschränkt wird. Folglich kann m im Worst-Case quadratisch mit n wachsen und muss für die Laufzeit getrennt von n betrachtet werden. Sei S in der gesamten Arbeit die Menge der insgesamt extrahierten Kuratowski-Subdivisions. Dann stellt

$$\Omega(n + m + \sum_{K \in S} |E(K)|) \tag{2.1}$$

eine untere Schranke für die Laufzeit dar, da sowohl Knoten als auch Kanten eingelesen und sämtliche Kanten aller extrahierten Kuratowski-Subdivisions ausgegeben werden müssen. Ziel ist es, einen Algorithmus zu entwerfen, der diese untere Schranke auch erreicht, d. h. zu zeigen, dass diese Schranke auch scharf ist.

Ein erster Ansatz für die Extraktion mehrerer Kuratowski-Teilgraphen ist, den Planaritätstest von Boyer-Myrvold so zu erweitern, dass in jeder auftretenden Stoppkonfiguration nicht nur maximal eine Kuratowski-Subdivision extrahiert wird, sondern möglichst viele. Das kann einerseits durch mehrere vorhandene kritische Knoten zwischen den Stoppknoten ermöglicht werden, andererseits können auch an jedem einzelnen kritischen Knoten mehrere kritische Kanten involviert sein (siehe Abbildung 2.1). Analog dazu kann es im Allgemeinen mehrere externe Backedges an den Stoppknoten geben. Alle auf diese Weise denkbaren Kombinationen von pertinenten und externen Backedges verursachen jeweils unterschiedliche Kuratowski-Subdivisions.

Um weitere Kuratowski-Subdivisions extrahieren zu können, kann zusätzlich die bisherige Klassifikation der verbotenen Minortypen erweitert werden. Der ursprüngliche Planaritätstest extrahiert nur den Minortyp, der in einer Stoppkonfiguration zwangsläufig auftreten muss. Allerdings gibt es weitere, häufig auftretende Untertypen von Minoren, die zu völlig anderen Kuratowski-Subdivisions führen. Zusätzlich können manche der Minortypen gleichzeitig auftreten und so die Anzahl der extrahierten Kuratowski-Subdivisions erhöhen. Allerdings muss darauf geachtet werden, dass alle Tests, die die Menge der neuen Minortypen und -untertypen einer Stoppkonfiguration bestimmen, genauso effizient durchführbar sind wie die Tests auf den bisherigen Minortypen.

Die beiden obigen Erweiterungen beziehen sich auf eine einzelne Stoppkonfiguration und

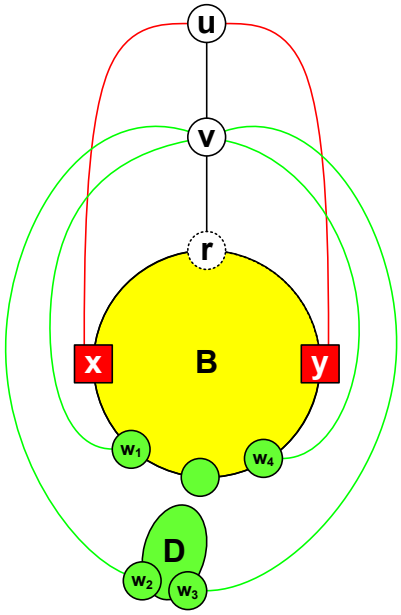


Abbildung 2.1: Der Minor einer einfachen Stoppkonfiguration an einer Bicomp B mit drei kritischen Knoten nach Einbettung des Knotens v. Die quadratischen, roten Knoten deuten auf externe Aktivität hin, während runde, grüne Knoten pertinent sind. Da jede kritische Backedge genau eine Kuratowski-Subdivision erzeugt, können hier insgesamt vier extrahiert werden.

sind deswegen lokal begrenzt. Beide Erweiterungen werden in Kapitel 2.2 ausführlicher betrachtet.

Es ist erfolgversprechender, zusätzlich die Suche nach Kuratowski-Subdivisions auch global zu erweitern. Dafür wird versucht, den Planaritätstest nach einer gefundenen Stoppkonfiguration weiterzuführen. Eine Voraussetzung für die Korrektheit des Planaritätstests ist aber, dass sich alle bisher eingebetteten Kanten innerhalb einer planaren Einbettung befinden. Diese Bedingung ist nach Fund einer Stoppkonfiguration verletzt und muss wiederhergestellt werden. Deswegen werden bei jeder auftretenden Stoppkonfiguration im Walkdown sämtliche kritischen Backedges entfernt, dessen Backedgepfade im DFS-Baum zu Knoten w zwischen den Stoppknoten x und y führen (siehe Abbildung 2.1). Alternativ wäre es auch möglich, alle externen Backedgepfade der beiden Stoppknoten zu entfernen, allerdings würde damit die Klassifizierung weiterer externer Knoten nicht mehr ohne weiteres in konstanter Zeit möglich sein.

Deswegen wird der erste Ansatz verfolgt: Alle kritischen Backedges werden gelöscht. Die aktuell betrachtete Bicomp ist danach planar eingebettet und selbst nicht mehr pertinent. Allerdings ist unklar, wie der Algorithmus fortfährt, denn der originale Planaritätstest würde hier abbrechen. Die betrachtete Bicomp ist nicht mehr zwangsläufig pertinent, also ist es nötig, in effizienter Zeit die nächste Bicomp zu finden, auf der der

Walkdown weitergeführt werden kann. Im weiteren Verlauf der Einbettung wird diese Prozedur jedesmal wiederholt, wenn eine neue Kuratowski-Subdivision gefunden wird.

Durch das Entfernen kritischer Backedges gilt außerdem die ursprüngliche Laufzeitabschätzung nicht mehr, denn diese Backedges erzeugen dann keine neue Face. Um trotzdem Linearzeit zu erreichen, werden neue Datenstrukturen, insbesondere im Walkup, eingeführt. Die Laufzeitabschätzung für diese neue Idee ist ungleich aufwändiger als die des originalen Planaritätstests und wird deswegen im Kapitel 2.3 unterteilt in:

1. Erweiterter Walkup
2. Erweiterter Walkdown
3. Extraktion der Kuratowski-Subdivisions
   a) Extraktion der kritischen Backedges
   b) Extraktion des *HighestFacePath* ohne Flipping
   c) Extraktion der *HighestXYPaths* und deren Lage

Da die meisten Laufzeiten im Folgenden amortisiert abgeschätzt werden müssen, werden die typischen Aussagen über *die Einbettungen aller Knoten* gemacht. Damit ist die Gesamtlaufzeit gemeint, die sich aus den einzelnen Einbettungen der im DFS-Baum absteigend betrachteten Knoten ergibt. Ziel ist es, die untere Schranke (2.1) für jeden der obigen Punkte auch zu erreichen, d. h. die Laufzeit über die Einbettung aller Knoten mit $O(n+m+\sum_{K\in S}|E(K)|)$ abschätzen zu können. Eine bestimmte Anzahl extrahierter Kuratowski-Subdivisions wird dabei nicht garantiert, da diese im Worst-Case auch exponentiell wachsen kann, aber die Anzahl ist in der Praxis hoch. Für die extrahierte Anzahl an Kuratowski-Subdivisions ist diese Laufzeit optimal.

2.1 Strukturresultate

Vorab werden einige Eigenschaften der Struktur eingebetteter Bicomps beschrieben. Einerseits werden dadurch spätere Laufzeitabschätzungen vereinfacht. Andererseits ist es für die Klassifikation der neuen Kuratowski-Minortypen notwendig, die Existenz und Lage aller sogenannten *HighestXYPaths* festzustellen. Das effiziente Testen dieser Eigenschaften wird durch die folgenden Strukturresultate ermöglicht.

Mittlerweile eingebettete Bicomps werden auch *ehemalige* Bicomps genannt. Ein Knoten heißt *Mergeknoten*, falls an ihm zwei ehemalige, aneinander hängende Bicomps verschmolzen wurden.

Lemma 2.1.1. *Sei eine Bicomp B mit Wurzel r gegeben und seien a und b die beiden Backedges an der Außenfläche von B, die an r enden. Dann existiert ein eindeutiger Pfad* Bottom-Chain *aus ehemaligen Bicomps in B, dessen Mergeknoten aller benachbarten Bicomps in der Außenfläche von B enthalten sind und dessen Anfangsbicomp durch a und Endbicomp durch b eingebunden wurde.*

Beweis. Die Bicomp B konnte vom Walkdown in die Ebene eingebettet werden. Alle ehemaligen Bicomps aus B wurden vom Walkup als pertinent markiert und bilden einen

Baum von Bicomps T. Seien C bzw. I die beiden ehemaligen Bicomps, an denen a bzw. b eingebettet worden sind (siehe Abbildung 2.2). Dann müssen C und I Blätter von T sein, da sonst eine Nachfolgerbicomp existieren würde, die entweder vom Walkdown nicht erreicht worden wäre oder nicht pertinent gewesen wäre. In beiden Fällen ergibt sich ein Widerspruch. Sei S der eindeutige Pfad von Bicomps in T, der C und I verbindet. Es wird gezeigt, dass S genau der gesuchte Pfad *Bottom-Chain* ist.

S ist eindeutig und enthält mit C und I bereits die Start- und Endbicomps, die die äußersten Backedges an r eingebettet haben. Es kann keine ehemalige Bicomp geben, die unterhalb von S in B eingebettet worden ist. Andernfalls wäre diese Bicomp oder ein Nachfolger von ihr mit einer Backedge zu r verbunden worden, und es ergäbe sich entweder ein Widerspruch zur Planarität der Einbettung oder zur Annahme, dass die äußersten Backedges an C und I starten. Deswegen stellen die Teile der Außenflächen der Bicomps aus S, die unterhalb der Mergeknoten verlaufen, zusammen mit a und b die gesamte Außenfläche der Bicomp B dar. Damit liegen auch alle Mergeknoten aus S auf der Außenfläche. □

Bemerkung. Es lässt sich noch etwas mehr über die Struktur der Bottom-Chain aussagen. Da diese einem Pfad im Bicompbaum T entspricht, existiert eine eindeutige Bicomp E der Bottom-Chain, die Wurzel des Teilbaums von T ist, der C und I enthält. E ist damit im DFS-Baum Vorgänger aller anderen Bicomps der Bottom-Chain. Somit sind im Walkup vor der Einbettung von B Eigenschaften wie Pertinenz und externe Aktivität der Nachfolgerbicomps auf E rekursiv vererbt worden. Also kann die Bottom-Chain vor der Einbettung in drei zusammenhängende Abschnitte aufgeteilt werden, von denen allerdings bis zu zwei leer sein können: Der erste und letzte Abschnitt enthält pertinente, aber nicht externe Bicomps, während die Bicomps des mittleren Abschnittes pertinent und extern sind. Falls der mittlere Abschnitt nicht leer ist, muss dieser einen externen Knoten aus E enthalten.

Definition 2.1.2. Sei eine Stoppkonfiguration in Bicomp B mit Wurzel r gegeben und seien *stopX* und *stopY* deren Stoppknoten mit einem dazwischen liegenden, kritischen Knoten w. Ein $XYPath(w)$ ist ein Pfad in B, der bis auf den Startknoten p_x und den Endknoten p_y aus innerhalb B's liegender Knoten besteht und folgende zusätzliche Eigenschaften erfüllt:

- p_x ist auf der Außenfläche von B im Pfad $r \rightarrow stopX \rightarrow w$ enthalten.
- p_y ist auf der Außenfläche von B im Pfad $r \rightarrow stopY \rightarrow w$ enthalten.
- $p_x \neq v$, $p_x \neq w$, $p_y \neq v$ und $p_y \neq w$.

In vielen der nicht-planaren Minortypen wäre ein vorhandener $XYPath(w)$ Bestandteil einer Kuratowski-Subdivision, weswegen dieser ein wichtiges Kriterium für die Extraktion ist. Da im Allgemeinen aber in einer Bicomp exponentiell viele *XYPaths* existieren können, wäre die Extraktion aller dieser Pfade zu aufwändig. Stattdessen beschränken wir uns auf einen eindeutigen Pfad, den sogenannten $HighestXYPath(w)$.

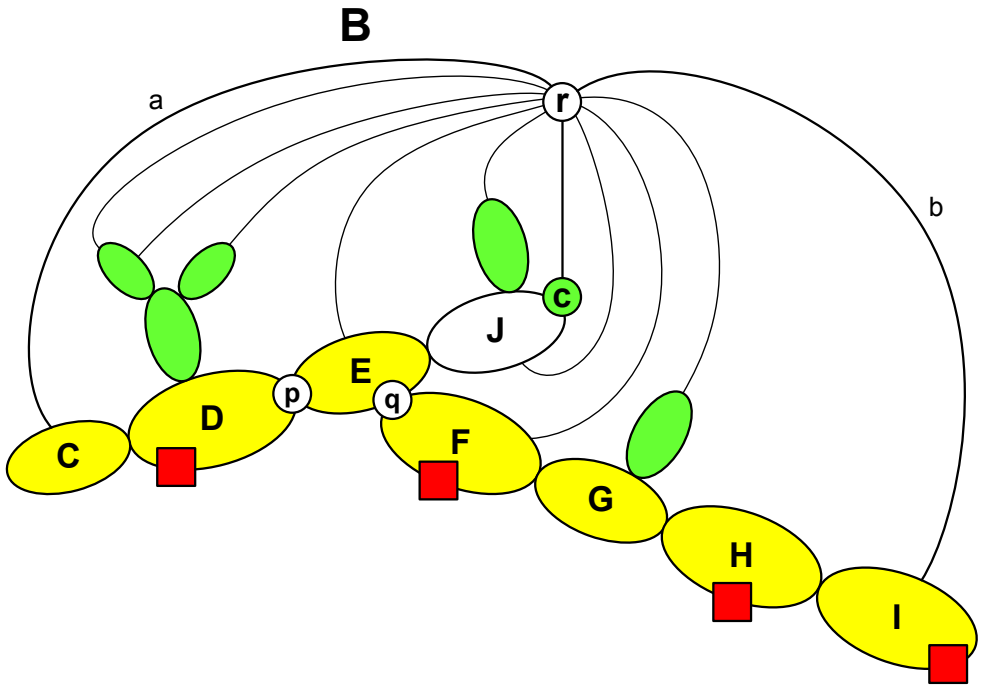


Abbildung 2.2: Eine vollständige Bicomp B, bestehend aus ihren ehemaligen, jetzt eingebetteten Bicomps. Die Kanten a und b stellen die beiden auf der Außenfläche von B liegenden Backedges dar, die an den Blättern C und I des ehemaligen Bicompbaums eingebettet wurden. Die gelben Bicomps C–I bilden damit die Bottom-Chain. Sämtliche Mergeknoten benachbarter Bicomps wie beispielsweise p und q befinden sich dadurch auf der Außenfläche von B.

Definition 2.1.3. Sei eine Stoppkonfiguration in Bicomp B mit Wurzel r gegeben und sei f die innere Face von B, die durch Löschen aller Kanten an r, außer den beiden Kanten der Außenfläche, und anschließendem Löschen der dadurch separierten Zweizusammenhangskomponenten entsteht. Der *HighestXYPath*(w) ist derjenige *XYPath*(w), der Bestandteil des begrenzenden Weges von f ist.

Das Löschen der inneren Kanten an r und der dadurch separierten Zweizusammenhangskomponenten vermindert die Anzahl der *XYPaths*(w) nicht, da diese homöomorph zu einer inneren Kante an r sind und der Knoten r nicht in einem *XYPath* enthalten sein kann. Damit existiert der *HighestXYPath*(w) genau dann, wenn ein *XYPath*(w) existiert. In Abbildung 2.3 wurde die Beispielbicomp B aus 2.2 durch einen kritischen Knoten zwischen den Stoppknoten erweitert und alle für die Berechnung des *HighestXYPath*(w) nötigen Löschungen durchgeführt.

Es wird gezeigt, dass der *HighestXYPath*(w) tatsächlich der oberste *XYPath*(w) in B ist und zusätzlich durch die ehemaligen Bicomps charakterisiert werden kann:

Theorem 2.1.4. *Sei eine Bicomp B mit Wurzel r gegeben und sei w ein kritischer Knoten einer Stoppkonfiguration in B, den die ehemalige Bicomp D enthält. Falls der* HighestXYPath(w) *existiert, ist dieser der oberste* XYPath(w) *und bildet zudem genau den vom Walkdown eingebetteten Teil der Außenfläche von D.*

Beweis. Für die Berechnung des *HighestXYPath*(w) werden Kanten und ehemalige Bicomps temporär gelöscht, um die Face f zu erhalten. Es muss vorab gezeigt werden, dass dadurch die Menge der *XYPaths*(w) nicht verändert wird: Jede der gelöschten und zu r inzidenten Kanten zusammen mit den dadurch separierten Zweizusammenhangskomponenten ist homöomorph zu einer Kante von einem inneren Knoten in B zu r. In keinem der *XYPaths*(w) kann aber eine solche gelöschte Komponente enthalten sein, da sie ausschließlich zu r führt. Insbesondere bleibt damit der *HighestXYPath*(w) unberührt.

Übrig bleibt die Face f, die an den *HighestXYPath*(w) grenzt. Da f inzident zu r ist, muss der *HighestXYPath*(w) den obersten *XYPath*(w) darstellen. Dann gilt für die Start- und Endknoten p_x und p_y des *HighestXYPath*(w), dass sie unter allen Endknoten der *XYPaths*(w) den kürzesten Abstand zu r auf der Außenfläche von B aufweisen. Andernfalls würde mindestens eine Kantenüberkreuzung entstehen, da ganz B bis auf die kritischen Backedges planar eingebettet wurde.

Die Endknoten p_x und p_y sind damit eindeutig bestimmt. Die genaue Position dieser beiden Knoten wird jetzt benutzt, um eine Charakterisierung durch die ehemaligen Bicomps zu erhalten: Betrachtet wird die Menge der ehemaligen Bicomps, die vor dem Walkdown vom Walkup markiert wurden. Zur Berechnung der Face f wurden bereits die ehemaligen Bicomps temporär aus der Menge entfernt, deren Außenflächen im Walkdown vollständig innerhalb von B eingebettet werden konnten, da genau diese separiert wurden. Die verbleibenden Bicomps stellen damit genau die Bottom-Chain dar. Nach Lemma 2.1.1 liegt jeder Mergeknoten zwischen zwei Bicomps der Bottom-Chain auf der Außenfläche von B. Daher stellen auch die unteren Außenflächen zwischen den Mergeknoten der Bottom-Chain einen Teil der Außenfläche von B dar. Die ehemalige Bicomp

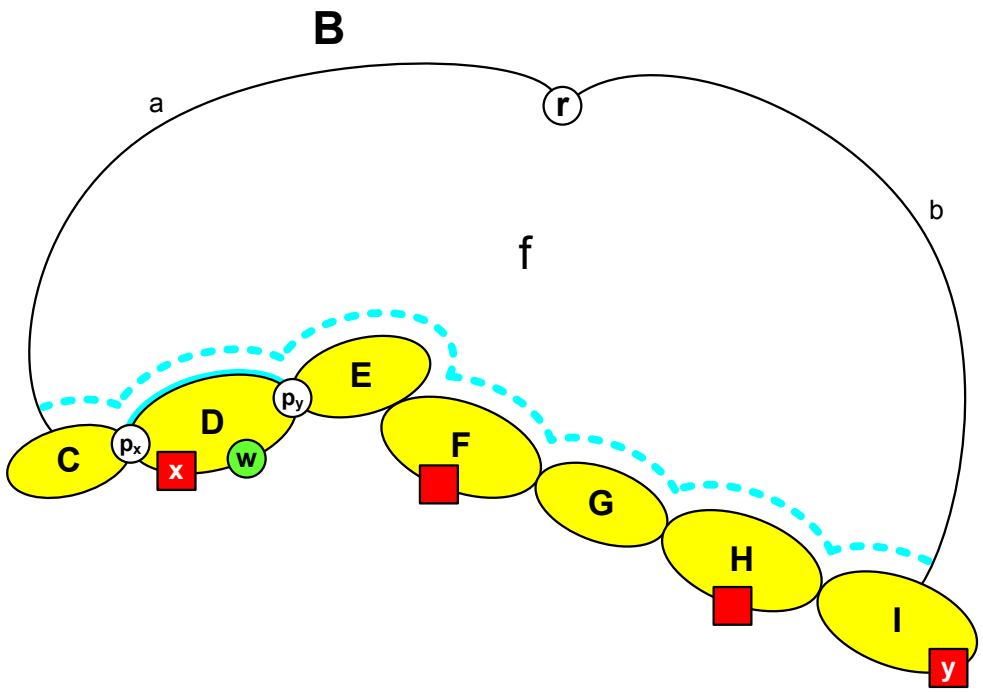


Abbildung 2.3: Die Bicomp B während der Bestimmung des *HighestXYPath*(w). Der Walkdown wurde an den Knoten x und y gestoppt, weswegen in B mit dem kritischen Knoten w eine Stoppkonfiguration vorliegt. Wird die Außenfläche von B entgegen dem Uhrzeigersinn betrachtet, beginnen alle *XYPaths*(w) auf der Außenfläche der Bicomp D nach oder am Knoten p_x, durchlaufen dann die Bicomp D und enden spätestens am Knoten p_y. Dabei wird in keinem Pfad der Knoten w traversiert. Der hier durchgezogen türkis markierte, eindeutige *HighestXYPath*(w) ist der *XYPath*(w) von p_x nach p_y, der an die Face f grenzt. Dieser ist Bestandteil des *HighestFacePath*(B), welcher gestrichelt markiert ist.

D enthält den nicht eingebetteten Knoten w und ist deswegen in der Bottom-Chain enthalten.

Da die Bottom-Chain einen Pfad darstellt, muss jeder Weg zwischen zwei in ihr enthaltenen Bicomps über die dazwischen liegenden Mergeknoten führen, solange dieser Weg nicht über die Wurzel r verläuft. Daher kann jeder *HighestXYPath* nur an einem Mergeknoten der Bottom-Chain oder an einem der direkten Nachbarknoten von r der Außenfläche beginnen und enden.

Liegen p_x und p_y jeweils an einem Mergeknoten, so müssen sich diese in derselben ehemaligen Bicomp befinden, also genau den beiden Mergeknoten der ehemaligen Bicomp D entsprechen (siehe Abbildung 2.3). Andernfalls würde ein Weg zwischen zwei Bicomps der Bottom-Chain existieren, der nicht alle dazwischen liegenden Mergeknoten enthält und es ließe sich ein Widerspruch zur Pfadstruktur der Bottom-Chain oder zu einer der definierenden Eigenschaften eines *HighestXYPaths* herleiten. Für den Fall, dass p_x (bzw. p_y) nicht an einem Mergeknoten liegt, sondern an einem Nachbarknoten von r existiert, ist D die erste (bzw. letzte) Bicomp der Bottom-Chain und der *HighestXYPath*(w) beginnt (bzw. endet) an dem zuletzt eingebetteten Knoten aus D und endet (bzw. beginnt) an dem verbleibendem Mergeknoten von D.

Im Allgemeinen lässt sich bei Betrachtung der Außenfläche von B gegen den Uhrzeigersinn festhalten, dass p_x der letzte Mergeknoten der Bottom-Chain vor w ist, sofern dieser existiert. Falls nicht, ist p_x der erste Knoten ungleich r. Analog dazu ist p_y der erste Mergeknoten der Bottom-Chain nach w, oder, falls nicht existent, der letzte Knoten ungleich r. In allen Fällen liegen p_x und p_y, und damit auch der *HighestXYPath*(w) selbst, an derselben ehemaligen Bicomp und der *HighestXYPath*(w) stellt genau den Teil der Außenfläche von D dar, der vom Walkdown eingebettet wurde. □

Korollar 2.1.5. Jede ehemalige Bicomp, die in B eingebettet werden konnte, ist entweder in der Bottom-Chain enthalten oder stellt eine vom Walkdown vollständig eingebettete, innerhalb von B liegende Bicomp dar.

Es können an jeder Stoppkonfiguration mehrere *HighestXYPaths*(w) für verschiedene w auftreten. Jeder dieser Pfade ist aber nach Definition 2.1.3 von der Face f begrenzt und kann effizient aus dieser extrahiert werden. Deswegen ist es möglich, sich im Wesentlichen auf die Berechnung des sogenannten *HighestFacePaths* zurückzuziehen.

Definition 2.1.6. Sei die nicht entartete Bicomp B gegeben und sei f wieder die innere Face von B, die durch Löschen aller Kanten an r (außer den beiden Kanten der Außenfläche) und anschließendem Löschen der dadurch separierten Zweizusammenhangskomponenten entsteht. Dann ist der *HighestFacePath*(B) der Teil des begrenzenden Weges der Face f ohne die beiden äußeren Kanten an r.

Der *HighestFacePath* stellt damit eine Möglichkeit dar, sämtliche *HighestXYPaths* einer Bicomp zu berechnen:

Korollar 2.1.7. Die *HighestXYPaths* aus B überlappen sich nicht und können mit Hilfe des *HighestFacePath*(B) ermittelt werden, da alle *HighestXYPaths* darin vollständig enthalten sind.

Korollar 2.1.8. Jeder Start- und Endknoten p_x und p_y des *HighestXYPath*(w) ist jeweils entweder ein Endknoten des umfassenden *HighestFacePath* oder stellt einen Mergeknoten zwischen zwei benachbarten Bicomps der Bottom-Chain dar.

2.2 Lokale Erweiterungen

Alle lokalen Erweiterungen beziehen sich auf die Stoppkonfiguration einer einzelnen Bicomp, an der möglichst viele Kuratowski-Subdivisions extrahiert werden sollen. Die Erweiterungen dieses Kapitels können somit an jeder Stoppkonfiguration angewendet werden, die mit Hilfe der globalen Erweiterungen gefunden wird. Dadurch wird die Gesamtausbeute an Kuratowski-Subdivisions erhöht.

2.2.1 Zusätzliche Backedgepfade

Jede Stoppkonfiguration erzeugt nach Boyer und Myrvold mindestens eine Kuratowski-Subdivision [BM04], allerdings sind in der Praxis üblicherweise weitaus mehr vorhanden. Der Grund dafür ist, dass zwischen den Stoppknoten mehrere kritische Knoten mit jeweils mehreren kritischen Backedges existieren können. Mit jeder dieser kritischen Backedges kann dann jeweils mindestens eine Kuratowski-Subdivision erzeugt werden.

Es gibt jedoch noch weitere Möglichkeiten, die Anzahl der extrahierten Kuratowski-Subdivisions zu erhöhen. Im Allgemeinen starten die Backedges nicht direkt an den kritischen Knoten der Bicomp, sondern sind durch einen Pfad in den Nachfolgerbicomps mit diesen verbunden. Der aus den jeweiligen DFS-Vorgängerknoten bestehende Backedgepfad vom Startknoten der Backedge bis zu einem kritischen Knoten ist mit Ausnahme eines einzigen Minortyps Bestandteil der erzeugten Kuratowski-Subdivision. Sind die Nachfolgerbicomps nicht entartet, existieren durch den zweifachen Zusammenhang an jeder solchen Bicomp zwischen den Knoten eines jeden Knotenpaares mindestens zwei kreuzungsfreie Pfade, beispielsweise die beiden Pfade auf der Außenfläche. Die Außenflächenpfade aller Nachfolgerbicomps können so kombiniert werden, dass neue Kuratowski-Subdivisions entstehen (siehe Abbildung 2.4). Mit dieser Erzeugung zusätzlicher Backedgepfade ist es im besten Fall sogar möglich, exponentiell viele verschiedene Kuratowski-Subdivisions für eine einzelne kritische Backedge zu finden.

Dabei ist es allgemein wünschenswert, Kuratowski-Subdivisions mit möglichst kleiner Kantenanzahl zu extrahieren. Während der Erzeugung der kritischen Backedgepfade werden insbesondere alle kürzesten Backedgepfade gefunden, die auf den Außenflächen der Nachfolgerbicomps liegen. Diese erzeugen die Kuratowski-Subdivisions mit der kleinsten Kantenanzahl. Falls es erforderlich ist, die Länge der extrahierten Backedgepfade in Abhängigkeit zu der Länge eines kürzesten Backedgepfades zu beschränken, beispielsweise

durch einen konstanten Faktor, so kann dies durch die Auswahl der einzelnen Pfade an den Außenflächen der Bicomps geschehen.

Jede traversierte Außenflächenkante ist Bestandteil mindestens einer weiteren extrahierten Kuratowski-Subdivision, die hinterher einen Teil der Ausgabe darstellt. Der Mehraufwand an solchen Kanten wird also durch eine größere Menge extrahierter Kuratowski-Subdivisions kompensiert. Die Berechnung aller kritischen Backedges einer Stoppkonfiguration ist dabei in $O(m)$ möglich, indem alle Backedges des einzubettenden Knotens v durchlaufen werden. Bei Anwendung der globalen Erweiterungen werden aber im Allgemeinen mehrere verschiedene Stoppkonfigurationen gefunden, womit die Anzahl extrahierter Kuratowski-Subdivisions drastisch erhöht werden kann.

Es ist möglich, dass dabei $\Omega(m)$ verschiedene Stoppkonfigurationen gefunden werden. In dem Fall würde dann allerdings durch die Berechnung jeweils involvierter, kritischer Backedges die quadratische Laufzeit $\Omega(m^2)$ auftreten. Mit Hilfe der globalen Erweiterung wird aber gezeigt werden, dass sich diese kritischen Backedges einschließlich der erzeugten kritischen Knoten sogar in amortisierter Zeit

$$O(1 + \frac{n}{m} + \frac{1}{m} \sum_{K \in S} |E(K)|)$$

berechnen lassen. Dadurch kann die Linearität der Laufzeit gerettet werden.

Neben der Verwendung mehrerer kritischer Backedges in einer Stoppkonfiguration können auch mehrere externe Backedges an den beiden Stoppknoten hilfreich sein. Bei k vorhandenen externen Backedges an einem Stoppknoten würde dabei die Anzahl bisheriger extrahierter Kuratowski-Subdivisions um den Faktor k vergrößert werden. Auch die externen Backedges können an Nachfolgerbicomps der Stoppknoten starten, weswegen zusätzlich eine Kombination der Außenflächenpfade dieser Nachfolgerbicomps analog zu den kritischen Backedgepfaden möglich ist. Im Gegensatz zu den kritischen Backedges ist es hier aber nicht ohne weiteres möglich, sämtliche externen Backedges eines Stoppknotens zu berechnen, ohne Effizienzeinbußen zu bewirken. Daher ist die Erzeugung von zusätzlichen externen Backedgepfaden in der Praxis nur eingeschränkt nutzbar.

2.2.2 Klassifikation neuer Minortypen

Bisher wurden in der Stoppkonfiguration alle Kombinationen von Backedgepfaden der kritischen Backedge und zweier externer Backedges betrachtet. An jeder solchen Kombination existiert eine Kuratowski-Subdivision. Allerdings wird diese je nach Lage der Backedgepfade aus verschiedenen Pfaden der Bicomp gebildet. Boyer und Myrvold [BM04] klassifizierten dafür die fünf Minortypen A–E (siehe Abbildung 1.8) und die fünf Untertypen E_1–$E5$ (siehe Abbildung 1.9).

Definition 2.2.1. In der Einbettungsphase des Knotens v heißt eine Bicomp, deren Wurzel virtueller Knoten von v ist, *Forebear-Bicomp*.

Alle anderen Bicomps heißen *NonForebear-Bicomps*. In der Klassifikation von Boyer und Myrvold ist der Minortyp A der einzige, der aus Stoppkonfigurationen an NonForebear-

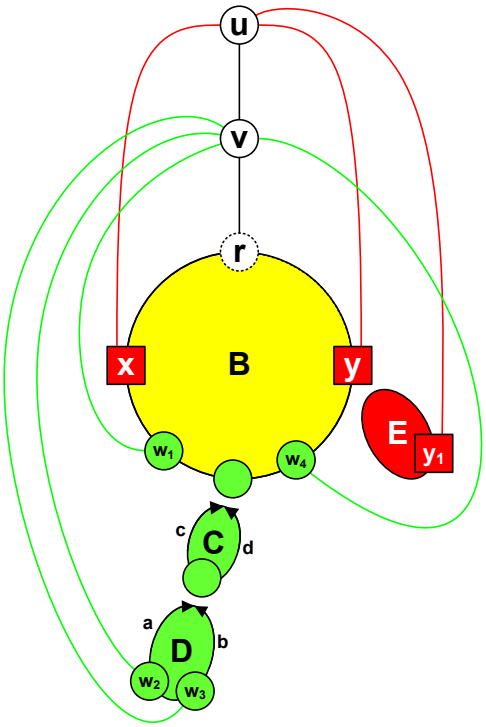


Abbildung 2.4: Der Minor einer komplexeren Stoppkonfiguration an der Bicomp B während der Einbettungsphase des Knotens v. An B hängen die pertinente Bicomp C und die externe Bicomp E. Die Kanten $\{u, v\}$, $\{v, r\}$ und die Kanten aus B stellen üblicherweise Pfade des Graphen dar, die in diesem Minor aber zu einer Kante kontrahiert wurden. Jeder kritische Knoten wird von einem Backedgepfad mindestens einer kritische Backedge erreicht. Somit entstehen alleine durch die Verwendung der unterschiedlichen kritischen Backedges vier verschiedene Kuratowski-Subdivisions. Zusätzlich können die Außenflächenkanten a–d der Nachfolgerbicomps C und D benutzt werden, um weitere sechs Kuratowski-Subdivisions zu erzeugen. Die zehn bisher verwendeten Backedgepfade können schließlich durch Kombination mit der externen Backedge an y_1 verdoppelt werden, so dass insgesamt 20 Kuratowski-Subdivisions aus dieser einzelnen Stoppkonfiguration extrahiert werden können.

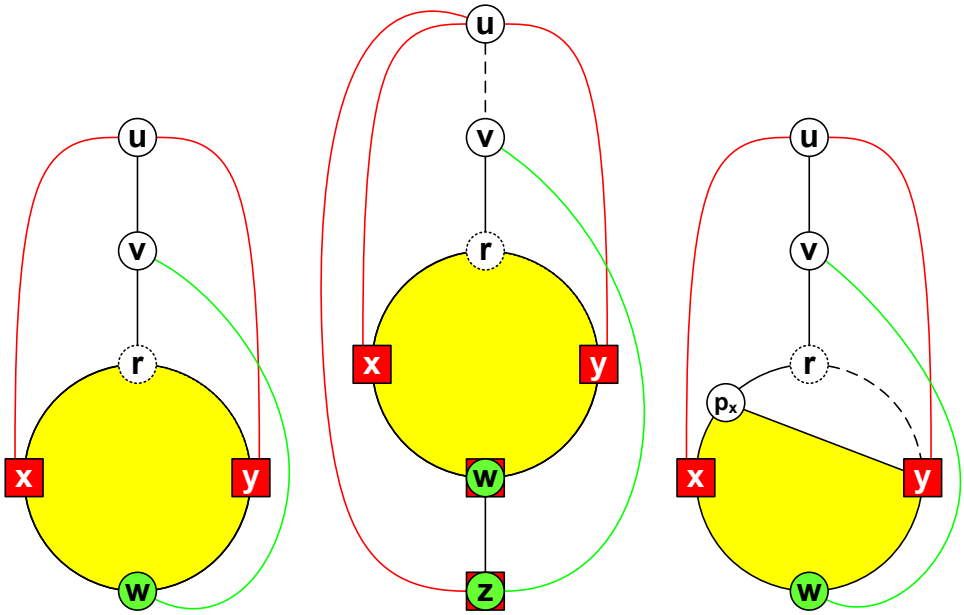
Bicomps extrahiert wird. Alle anderen Minortypen beziehen sich auf Stoppkonfigurationen an einer Forebear-Bicomp.

Es ist jedoch möglich, in einer NonForebear-Bicomp noch weitere Minortypen zu finden. Diese kommen im Gegensatz zum Minortyp A nicht zwangsläufig vor, gewähren aber die Chance auf zusätzliche Kuratowski-Subdivisions, da an ihnen andere Pfade der Bicomp verwendet werden. Die Abbildung 2.5 stellt die vier neuen Minortypen AB–AE für NonForebear-Bicomps dar. In Abbildung 2.6 wird der Minortyp AE nochmals in vier neue Untertypen AE_1–AE_4 aufgeteilt.

Jeder dieser Minortypen enthält einen $K_{3,3}$. Dazu müssen lediglich die jeweils gestrichelt dargestellten Kanten entfernt werden. Sei eine Stoppkonfiguration in einer NonForebear-Bicomp mit pertinenter Backedge p und den Stoppknoten $stopX$ und $stopY$ gegeben. Sei weiterhin w der kritische Knoten der Bicomp, an dem der Backedgepfad von p endet und p_x und p_y die beiden Endknoten des $HighestXYPath(w)$, falls dieser existiert. Anders als in [BM04] wird hier eine Definition angestrebt, bei der alle Minortypen einschließlich ihrer Symmetrien disjunkt sind. Die neuen Minortypen an der Backedge p lassen sich wie folgt klassifizieren:

- *Minortyp AB* entsteht, wenn an w eine extern aktive Kinderbicomp existiert, die vom Backedgepfad von p traversiert wird. In dem Fall wird der Pfad $u \rightarrow v$ für die Kuratowski-Subdivision nicht benötigt. Da ein Minor betrachtet wird, stellt u nicht unbedingt die Wurzel des DFS-Baums dar.

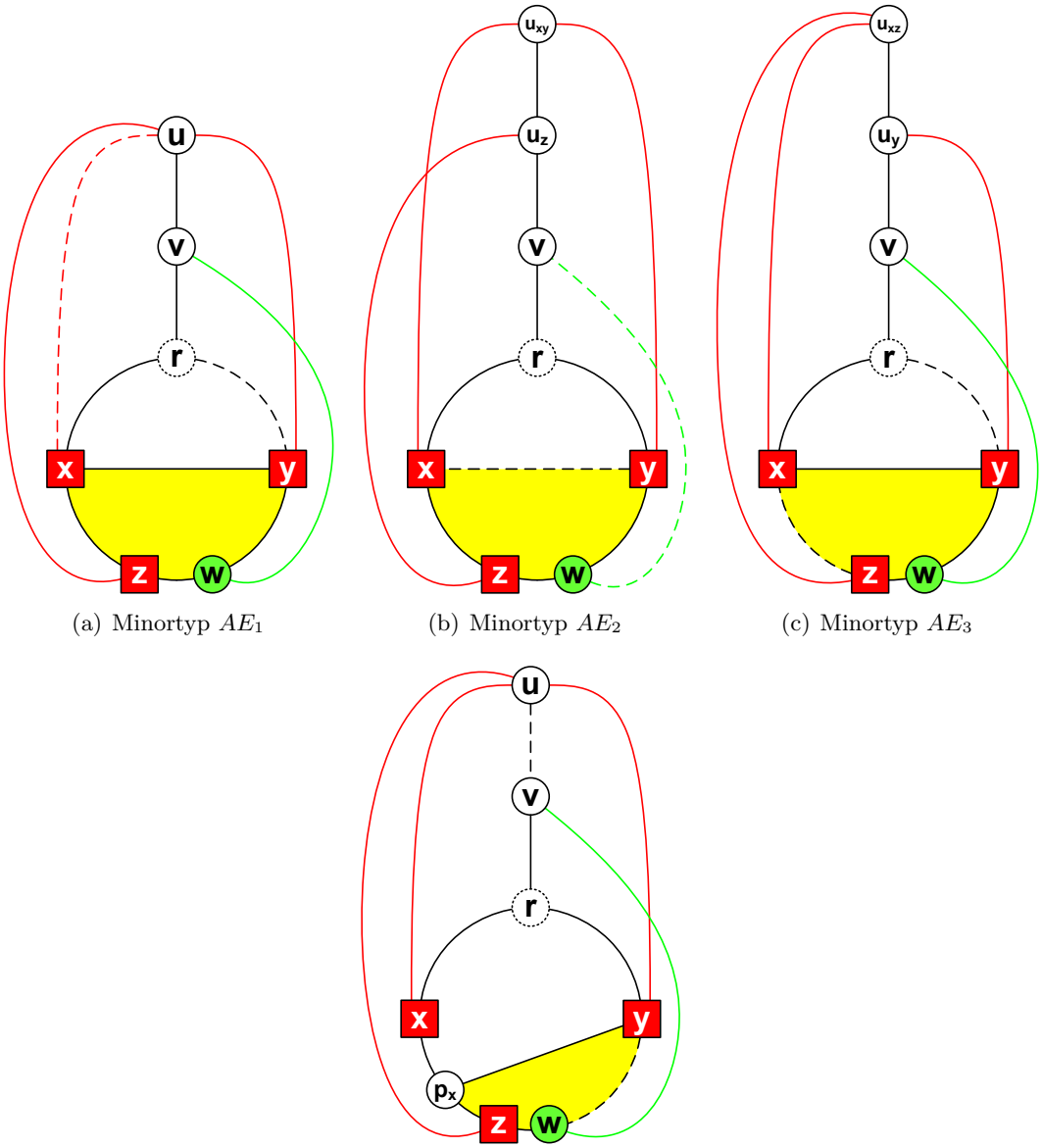
  Für alle weiteren Minortypen muss der $HighestXYPath(w)$ existieren:
- *Minortyp AC* existiert, wenn sich p_x (bzw. p_y) auf einem inneren Knoten des Pfades $r \rightarrow stopX$ (bzw. $r \rightarrow stopY$) befindet. Für die Kuratowski-Subdivision wird dabei der Pfad $r \rightarrow stopY$ (bzw. $r \rightarrow stopX$) nicht benötigt. Der Minortyp AC wird auch dann gebildet, wenn diese beiden, zueinander symmetrischen Bedingungen erfüllt sind.
- *Minortyp AD* entsteht durch die Existenz eines weiteren Pfades $r \rightarrow z$ innerhalb der Bicomp, wobei z ein innerer Knoten des $HighestXYPath(w)$ ist. Die beiden Endknoten p_x und p_y können dabei auch verschieden von den Stoppknoten sein. Für die Kuratowski-Subdivision werden die beiden Pfade der Außenfläche der NonForebear-Bicomp nicht benötigt, die an r starten und an dem jeweils Näheren der beiden Knoten p_x und $stopX$ (bzw. p_y und $stopY$) enden.
- *Minortyp AE* existiert, falls ein externer Knoten z auf der Außenfläche unterhalb des $HighestXYPath(w)$ existiert und einer der beiden Knoten p_x und p_y mindestens so weit von r entfernt ist wie der zugehörige Stoppknoten. Sei q der Backedgepfad der externen Backedge, die z extern aktiv macht. Falls $z = w$ ist, müssen zusätzlich p und q auf allen Knoten mit einem größeren DFI-Wert als w knotendisjunkt sein. Diese Definition ist etwas spezieller gehalten als die vergleichbare für den Minortyp E in [BM04] beschriebene. Dadurch kann bei Existenz des Minortyps AE durch folgende Bedingung gewährleistet werden, dass auch mindestens einer der Untertypen AE_1–AE_4 auftritt:



(a) Minortyp A (b) Minortyp AB (c) Minortyp AC

(d) Minortyp AD (e) Minortyp AE

Abbildung 2.5: Klassifikation der Minortypen, die in einer NonForebear-Bicomp mit Wurzel r auftreten können. Die jeweils gestrichelten Kanten werden für die Extraktion der entsprechenden Kuratowski-Subdivisions nicht benötigt. Minortyp AE wird in Abbildung 2.6 in weitere vier Untertypen unterteilt. Alle Minortypen enthalten eine $K_{3,3}$-Subdivision.



(a) Minortyp AE_1 (b) Minortyp AE_2 (c) Minortyp AE_3

(d) Minortyp AE_4

Abbildung 2.6: Die Unterteilung des Minortyps AE in seine vier verschiedenen Untertypen.

Falls p_x (bzw. p_y) ein innerer Knoten des Pfades $r \to stopX$ (bzw. $r \to stopY$) ist, muss mindestens eine der folgenden Gleichungen verletzt sein:

$$\begin{aligned} p_y &= stopY \text{ (bzw. } p_x = stopX) \\ u_x &= u_y = u_w \\ z &= w \end{aligned}$$

Im Folgenden werden die einzelnen Untertypen von AE definiert:

- *Minortyp* AE_1 existiert, wenn $z \neq w$ ist. Dadurch können zwei symmetrische Fälle entstehen, je nachdem, ob z näher an dem Knoten $stopX$ oder $stopY$ liegt. Im ersten Fall werden die Pfade $stopX \to u$ und $r \to stopY$ für die Kuratowski-Extraktion nicht benötigt, im zweiten Fall die Pfade $stopY \to u$ und $r \to stopX$ nicht.
- *Minortyp* AE_2 existiert, wenn die externe Backedge, die z verursacht, an einem Knoten u_z mit größerem DFI-Wert endet als die beiden externen Backedges von $stopX$ und $stopY$. Dabei wird für eine Kuratowski-Extraktion sowohl der *HighestXYPath* als auch der Pfad $w \to v$ nicht benötigt. Folglich ist dieser Minortyp unabhängig von der gewählten pertinenten Backedge. Damit existieren genauso viele AE_2-Minoren in der Stoppkonfiguration, wie Backedges an externen Knoten z vorhanden sind. Die z-Knoten können dabei unter einem beliebigen *HighestXYPath* der Stoppkonfiguration liegen.

  Bemerkung. Eine kritische Backedge kann also einerseits eine Stoppkonfiguration verursachen, muss aber andererseits nicht notwendigerweise Bestandteil der extrahierten Kuratowski-Subdivision sein. Bei den Minortypen E_2 und AE_2 wird das dadurch ermöglicht, dass auf die Einbettungsphase des Knotens u_z vorausgegriffen wird. In dieser wird die externe Backedge an z pertinent sein, während $stopX$ und $stopY$ durch die niedrigeren DFI-Werte noch immer extern sein werden. Damit tritt dann zwangsläufig ein Minor vom Typ A auf. Dieser wird durch die Minortypen E_2 bzw. AE_2 bereits im Voraus erkannt und extrahiert.
- *Minortyp* AE_3 existiert, wenn die externe Backedge an $stopY$ (bzw. an $stopX$) an einem Knoten u_y (bzw. u_x) endet, der einen höheren DFI-Wert als u_x und u_z (bzw. u_y und u_z) aufweist. Sei hier o. B. d. A. u_y der Knoten mit dem höchsten DFI-Wert, der andere Fall ist dazu symmetrisch. Sei a derjenige der beiden Knoten p_x und $stopX$, der am weitesten von r entfernt ist. Dann wird für eine Kuratowski-Extraktion sowohl der Pfad $r \to stopY$, als auch der Pfad unterhalb des $HighestXYPath(w)$ von a bis zu dem nächsten der beiden Knoten z und w nicht benötigt.
- *Minortyp* AE_4 existiert, wenn p_x (bzw. p_y) weiter von r entfernt ist als $stopX$ (bzw. $stopY$). Sei $a \in \{p_y, stopY\}$ der Knoten (bzw. aus $\{p_x, stopX\}$), der am weitesten von r entfernt ist. Für die Extraktion der Kuratowski-Subdivision wird der Pfad $u \to v$ und der Pfad unter dem $HighestXYPath(w)$ von a zu dem nächsten der beiden Knoten w und z nicht benötigt.

In einer NonForebear-Bicomp können also insgesamt sieben weitere Minortypen und ihre jeweiligen symmetrischen Fälle vorkommen. Falls die Klassifikation dieser Minortypen ebenso effizient möglich ist wie die der bisherigen Minortypen, beläuft sich der Mehraufwand nur auf einen konstanten Faktor, da nur endlich viele Minortypen hinzugefügt worden sind. Für die Klassifizierung der neuen Minortypen werden im Wesentlichen folgende Informationen benötigt:

1. Die Knoten v, r, $stopX$ und $stopY$.
2. Alle kritischen Backedges p mit Backedgepfaden und den dadurch erzeugten, kritischen Knoten w.
3. Jeweils mindestens eine externe Backedge pro Stoppknoten und jeweils dessen Endknoten u_x und u_y. Außerdem werden die Backedgepfade der externen Backedges benötigt.
4. Der $HighestXYPath(w)$, dessen Endknoten p_x und p_y und die Information, ob ein innerer Knoten des $HighestXYPath(w)$ existiert, der über einen Pfad mit r verbunden ist (siehe Abbildung 2.5(d)).
5. Die Lage von p_x und p_y, falls der $HighestXYPath(w)$ existiert. Wichtig sind dabei die Informationen, ob p_x und p_y oberhalb oder unterhalb des jeweiligen Stoppknotens liegen oder identisch zu ihm sind.
6. Die Existenz und Lage externer Knoten z unterhalb des $HighestXYPath(w)$ (siehe beispielsweise Abbildung 2.6(c)). Wichtig sind dabei die Informationen, ob $z = w$ ist oder auf einem inneren Knoten des Pfades $stopX \rightarrow w$ oder des Pfades $w \rightarrow stopY$ liegt.

Alle anderen Informationen lassen sich beispielsweise durch Knotenvergleiche und Überprüfungen auf Pertinenz, externe Aktivität und LowPoints in konstanter Zeit errechnen. Zum Beispiel wird für den Test auf Minortyp AB die Bicomp betrachtet, die die letzte Kante des Pfades p enthält. Aus dieser letzten Kante lässt sich die inzidente Wurzel dieser Bicomp in $O(1)$ berechnen. Weiterhin kann auch der eindeutige Knoten c mit kleinstem DFI-Wert dieser Bicomp in konstanter Zeit errechnet werden. Genau dann, wenn der LowPoint(c) kleiner ist als der DFI-Wert von v, existiert ein externer Knoten in der Bicomp und damit auch ein Minor vom Typ AB. Pfade, die nicht für jeden Minortyp extrahiert werden müssen, wie beispielsweise $u \rightarrow v$, können bei Bedarf in linearer Zeit zur ihrer Länge mit Hilfe des DFS-Baums extrahiert werden.

Stehen alle aufgezählten Informationen zur Verfügung, kann der Test auf einen beliebigen Minortyp damit in konstanter Zeit durchgeführt werden. Dabei sind die Informationen identisch mit denen, die für die bisherigen Minortypen benötigt wurden, so dass asymptotisch kein Mehraufwand entsteht. Im Folgenden wird skizziert, wie die aufgezählten Informationen zusammengetragen werden:

zu 1) Sämtliche benötigten Knoten liefert der Walkdown in konstanter Zeit, da er v einbettet, r traversiert und jeweils an den Stoppknoten $stopX$ und $stopY$ hält.

zu 2,4,5) Die Berechnung dieser Informationen ist komplex und benötigt aus Effizienzgründen die Datenstrukturen der globalen Erweiterung. In dieser wird aber gezeigt

werden, dass sie sich amortisiert in linearer Gesamtzeit berechnen lassen.

zu 3) Hier werden die externen Backedges der beiden Stoppknoten, deren Endknoten u_x und u_y und die involvierten Backedgepfade benötigt. Anhand eines Beispiels wird hier gezeigt, wie diese Informationen extrahiert werden können:

Wir extrahieren den externen Backedgepfad an dem Knoten *stopX*, deren Backedge und den Knoten u_x. Die Vorgehensweise ist unterschiedlich zu der Extraktion kritischer Backedgepfade, da im Gegensatz dazu die verursachende Backedge nicht bekannt ist. Ausgehend von *stopX* wird der gesamte Backedgepfad bis hin zu u_x traversiert (siehe Abbildung 2.7).

Eine potentielle, direkt an *stopX* startende Backedge kann über den LeastAncestor in $O(1)$ gefunden werden. Falls diese nicht vorhanden ist, muss eine externe Backedge existieren, dessen Startknoten in einer Nachfolgerbicomp von *stopX* liegt. Daher kann die *SeparatedDFSChild*(*stopX*)-Liste nicht leer sein und es wird in die Bicomp abgestiegen, auf die der erste Eintrag dieser Liste verweist. Da die *SeparatedDFSChild*-Listen aufsteigend nach den LowPoint-Werten sortiert sind, ist dadurch sichergestellt, dass in eine externe Bicomp abgestiegen wurde. Der LowPoint-Wert l des gewählten Eintrags entspricht dabei genau dem DFI-Wert des späteren Knotens u_x. Damit ist der Endknoten der Backedge gefunden. Im Folgenden wird versucht, die nächste Backedge zu finden, die an u_x endet.

Der Startknoten der gesuchte Backedge mit Endknoten u_x kann entweder in der aktuellen Bicomp oder in einer deren Nachfolgerbicomps liegen. Deswegen wird auf der Außenfläche der Bicomp ein paralleler Suchlauf in beide Richtungen gestartet. Dieser terminiert an einem Knoten t, dessen LeastAncestor entweder u_x ist oder an dem eine *SeparatedDFSChild*(t)-Liste existiert, deren erster Eintrag den Lowpoint l aufweist. Im ersten Fall ist damit die externe Backedge gefunden, andernfalls wird der erste Eintrag der *SeparatedDFSChild*(t)-Liste gewählt und das Verfahren auf den entsprechenden Nachfolgerbicomps iteriert, bis u_x gefunden wird. Die letzte Kante des so traversierten Pfades stellt die gesuchte Backedge dar.

Bemerkung. Da wir im Allgemeinen Kuratowski-Subdivisions mit möglichst geringer Kantenanzahl suchen, stellt sich die Frage, warum gerade die Backedge extrahiert wird, die an dem Knoten mit niedrigstem DFI-Wert endet. Dadurch werden häufig unnötig lange externe Backedgepfade extrahiert. Allerdings hat diese Methode in der Praxis einen Vorteil: Dadurch, dass der Backedgepfad relativ lang ist, werden während der Extraktion zusätzlich die meisten kürzeren Backedgepfade ohne Mehraufwand gefunden. Diese können benutzt werden, um eine größere Anzahl von Kuratowski-Subdivisions zu extrahieren. Unter diesen befinden sich dann auch die gewünschten Kuratowski-Subdivisions mit kleiner Kantenanzahl.

zu 6) Es wird gezeigt, wie die Existenz und Lage mindestens eines z-Knotens unterhalb des *HighestXYPath*(w) effizient berechnet werden kann. Dieser Knoten z ist dabei für die Untertypen von E und AE ein wichtiges Unterscheidungsmerkmal. Falls die Stoppkonfiguration an einer NonForebear-Bicomp auftritt, enthält der zwangsläufig auftretende Minor vom Typ A den gesamten Außenflächenpfad unter dem

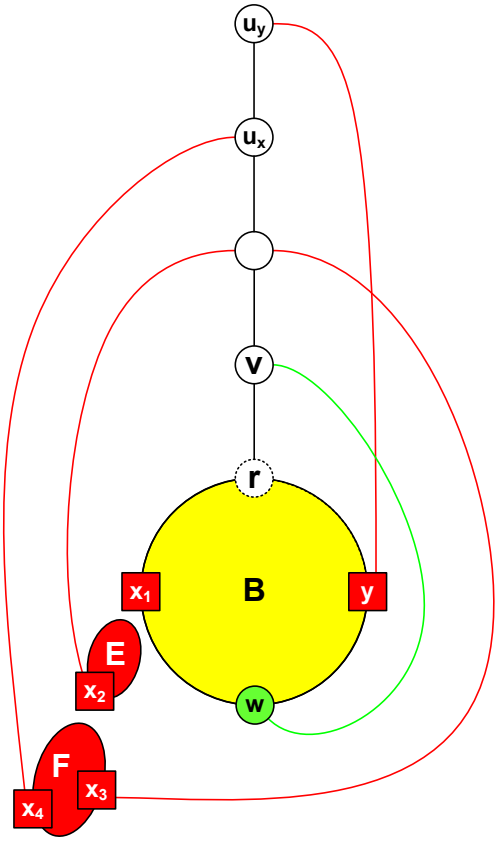


Abbildung 2.7: Beispiel der Berechnung eines externen Backedgepfades an dem Stoppknoten $stopX = x_1$. An x_1 existiert keine direkte Backedge, weswegen in die nachfolgende Bicomp E abgestiegen wird. Dabei wird der LowPoint l des gewählten Eintrages der $SeparatedDFSChild(stopX)$-Liste festgehalten. Dieser entspricht dem DFI-Wert des gesuchten Endknotens u_x einer noch zu suchenden externen Backedge. Auf den Außenkanten der Bicomp E wird ein paralleler Suchlauf gestartet, bis der externe Knoten x_2 gefunden wird. Auch an diesem Knoten existiert keine direkte Backedge zu u_x, aber der LowPoint von x_2 ist l. Daher wird die an x_2 hängende Bicomp F traversiert, wobei zuerst der Knoten x_3 erreicht wird. Hier ist eine direkte Backedge vorhanden, allerdings nicht zum Knoten u_x. Aufgrund des LowPoints l muss aber ein weiterer Knoten existieren, dessen Backedge an u_x endet. Bei Fortführung des Suchlaufs auf den Außenkanten wird der Knoten x_4 mit dieser Eigenschaft gefunden.

HighestXYPath(w). In diesem Fall kann jeder Knoten dieses Pfades auf externe Aktivität geprüft werden und es werden alle möglichen z-Knoten gefunden.

Andernfalls existiert die Stoppkonfiguration in einer Forebear-Bicomp und einer der Minortypen B–E (siehe Abbildung 1.8) muss vorhanden sein. In dem Fall, dass mindestens einer der Minortypen B–D auftritt, wird ebenfalls der gesamte Außenflächenpfad unterhalb des *HighestXYPath*(w) extrahiert und jeder z-Knoten kann wiederum effizient berechnet werden. Andernfalls treten ausschließlich Untertypen von Minortyp E auf (siehe Abbildung 1.9). Das Traversieren aller Knoten unterhalb des *HighestXYPath*(w) würde jetzt in einer superlinearen Gesamtlaufzeit resultieren, da dieser Pfad im Allgemeinen nicht komplett extrahiert werden muss.

Falls wie in Minortyp E_5 $w = z$ gilt, ist der z-Knoten gefunden. Daher bleiben lediglich die Untertypen E_1–E_4 übrig, in denen jeweils $w \neq z$ gelten muss. An diesen gelten jeweils folgende drei Eigenschaften für den Außenflächenpfad unter dem *HighestXYPath*(w):

a) Es muss ein Knoten z existieren, da andernfalls eine Stoppkonfiguration ohne Kuratowski-Subdivision aufgetreten wäre.

b) Es werden entweder der vollständige Pfad $w \rightarrow stopX$ oder der vollständige Pfad $w \rightarrow stopY$ oder beide extrahiert.

c) Der Pfad $w \rightarrow z$ wird ebenfalls extrahiert.

Dadurch wird folgender Algorithmus zur Bestimmung mindestens eines z-Knotens ermöglicht: Es werden ausgehend von w zwei parallele Traversierungen der Außenfläche in entgegengesetzte Richtungen gestartet. Jede Traversierung bricht spätestens an p_x bzw. p_y ab. Während der Traversierung wird an jedem Knoten in $O(1)$ überprüft, ob er extern ist. Wird an einer Traversierung ein solcher externer Knoten gefunden, werden beide Traversierungen abgebrochen und der entsprechende Minor mit dem Knoten z extrahiert. Die asymptotische Laufzeit entspricht dabei der Länge des kürzeren der traversierten Pfade. Nach Extraktion des Minors ist aufgrund von Eigenschaft b) entweder der Pfad $w \rightarrow stopX$ oder $w \rightarrow stopY$ extrahiert worden. Dieser kann zusätzlich nach weiteren Vorkommen externer Knoten durchsucht werden, um möglichst viele Kuratowski-Subdivisions zu erhalten.

Insgesamt können die neuen Minortypen also genauso effizient berechnet werden wie die bisherigen Minortypen.

2.2.3 Einsatz der neuen Minortypen

Der ursprüngliche Planaritätstest extrahiert nur genau einen, zwangsläufig auftretenden Minortyp aus der Menge der Typen $\{A, B, C, D, E_1, \ldots, E_5\}$. Falls die Stoppkonfiguration in einer NonForebear-Bicomp auftritt, existieren aber zusätzlich zu A gleichzeitig oft mehrere der neuen Minortypen AB,AC,AD und AE_1–AE_4 an denselben Backedgepfaden. Da gezeigt wurde, dass diese Minortypen effizient extrahiert werden können, erhöht

das die Ausbeute an Kuratowski-Subdivisions.

Zudem ist es für jede Kombination aus kritischen und externen Backedgepfaden möglich, dass auch manche der bisherigen Minortypen $\{B, C, D, E_1, \ldots, E_5\}$ gleichzeitig auftreten können. Dabei muss im Falle eines Untertyps von E beachtet werden, dass die Kombination auch den Backedgepfad des gewählten z-Knotens enthält. Die potentiell vorhandenen z-Knoten in gleichzeitig auftretenden Minortypen müssen folglich denselben Knoten darstellen. Jede solche Kombination von Backedgepfaden kann nun direkt auf alle möglichen Minortypen getestet werden, anstatt den Test nur solange fortzuführen, bis die erste Kuratowski-Subdivision gefunden wird. Da wir nur eine endliche Menge von Minortypen klassifiziert haben, bleibt auch hierbei die asymptotische Laufzeit bestehen. Die Tabelle 2.1 zeigt, welche Minortypen gemeinsam auftreten können. Falls ein Minor während des Algorithmus gefunden wurde, müssen nur noch diejenigen Minortypen geprüft werden, die dann nicht durch die Tabelle ausgeschlossen werden.

Um das Zusammenspiel der einzelnen lokalen Erweiterungen zu verdeutlichen, wird eine allgemeine Vorgehensweise für die Extraktion möglichst vieler lokaler Kuratowski-Subdivisions in Pseudocode dargestellt (siehe Algorithmus 1). Dabei wird die Erzeugung zusätzlicher Backedgepfade, die Einbeziehung neuer Minortypen und das Extrahieren mehrfacher Minortypen an derselben Backedge berücksichtigt. Auch die Reihenfolge, in der die für die Minorklassifizierung benötigten Informationen zusammengetragen werden, ist dabei wichtig. Nach erfolgreicher Extraktion müssen die kritischen Backedges gelöscht werden, damit der bisher eingebettete Subgraph planar bleibt und der Planaritätstest fortfahren kann.

2.2.3.1 Unterschiedliche Kuratowski-Subdivisions

Es ist wünschenswert, dass die in der lokalen Erweiterung extrahierten Kuratowski-Subdivisions nicht identisch sind. Folgendes Resultat bereitet die äquivalente Aussage für Kuratowski-Subdivisions der globalen Erweiterung vor.

Lemma 2.2.2. *Die mit der lokalen Erweiterung extrahierten Kuratowski-Subdivisions einer Stoppkonfiguration sind paarweise mindestens in einer Kante verschieden.*

Beweis. Von verschiedenen Minortypen werden immer auch verschiedene Kuratowski-Subdivisions erzeugt, da sich jede berechnete Kuratowski-Subdivision zu genau einem Minortyp zuordnen lässt. Deswegen können durch unterschiedliche Minortypen keine identischen Kuratowski-Subdivisions entstehen.

Andererseits könnten Kuratowski-Subdivisions identisch sein, wenn sie von demselben Minortyp sind. Dann müssen diese sich aber wegen der Vorgehensweise der Extraktion in den benutzten kritischen und externen Backedgepfaden unterscheiden. Externe Backedgepfade werden nur dann berechnet, wenn sie auch verwendet werden und verursachen damit keine Probleme. Identische Kuratowski-Subdivisions sind also nur möglich, falls der kritische Backedgpfad für die Extraktion nicht benötigt wird. Das ist bei Minoren des Typs E_2 und AE_2 der Fall, bei denen der kritische Backedgepfad nicht im extrahierten

	A	AB	AC	AD	AE_1	AE_2	AE_3	AE_4	B	C	D	E_1	E_2	E_3	E_4	E_5
A	×	×	×	×	×	×	×	×								
AB	×	×	×	×	×	×	×	×								
AC	×	×	×	×	×	×	×	×								
AD	×	×	×	×	×	×	×	×								
AE_1	×	×	×	×	×	×	×	×								
AE_2	×	×	×	×	×	×		×								
AE_3	×	×	×	×	×		×	×								
AE_4	×	×	×	×	×	×	×	×								
B									×	×	×	×	×	×	×	×
C									×	×	×	×*	×*	×*	×*	
D									×	×	×	×	×	×	×	×
E_1									×	×*	×	×	×	×	×	
E_2									×	×*	×	×	×		×	
E_3									×	×*	×	×		×	×	
E_4									×	×*	×	×	×	×	×	
E_5									×		×					×

Tabelle 2.1: Gegenüberstellung aller Minortypen bei identisch gewählten Backedgepfaden in einer Stoppkonfiguration. Im Einzelnen inbegriffen sind dabei die Backedgepfade an den kritischen Knoten, den beiden Stoppknoten und im Falle der Minortypen AE oder E auch die Backedgepfade an den z-Knoten. Ein Kreuz zwischen zwei Minortypen bedeutet, dass beide Typen in einer solchen Stoppkonfiguration nebeneinander existieren können, andernfalls schließen sich diese aus. Die mit einem Stern markierten Einträge weisen darauf hin, dass die beiden Minortypen in der ursprünglichen Definition von Boyer und Myrvold nicht disjunkt sind und deswegen ein Vergleich keinen Sinn macht. Beispielsweise können dort Minoren existieren, die gleichzeitig dem Typ C und einem Symmetriefall eines E-Untertypen entsprechen. Eine disjunkte Definition ist aber analog zu der Definition der neuen AE-Untertypen möglich. Bei Anwendung dieser können die Minortypen an einem mit einem Stern markierten Eintrag gleichzeitig vorkommen.

Algorithmus 1 Lokale Extraktion einer Stoppkonfiguration

1: **für alle** kritischen Backedges p
2: Berechne den Backedgepfad s von p
3: **für alle** erweiterten Backedgepfade s' von s
4: Berechne für beide Stoppknoten möglichst viele externe Backedges
5: **für alle** externen Backedges
6: Berechne den zugehörigen externen Backedgepfad
7: **end für**
8: **für alle** Paare (t, u) von Backedgepfaden an jeweils *stopX* und *stopY*
9: Berechne die Informationen 1–6 (vgl. Seite 36)
10: Berechne alle durch s', t und u verursachten Minortypen
11: **für alle** auftretenden Minortypen $\in \{A, AB, AC, AD, B, C, D\}$
12: Extrahiere die Kuratowski-Subdivision aus dem Minor
13: **end für**
14: **für alle** berechneten z-Knoten in Punkt 6 (vgl. Seite 36)
15: **für alle** auftretenden Minortypen $\in \{AE_2, E_2\}$
16: **falls** s' = erster Backedgepfad unter dem *HighestXYPath* {
17: Extrahiere die Kuratowski-Subdivision aus dem Minor
18: **end falls**
19: **end für**
20: **für alle** auftr. Minortypen $\in \{AE_1, AE_3, AE_4, E_1, E_3, E_4, E_5\}$
21: Extrahiere die Kuratowski-Subdivision aus dem Minor
22: **end für**
23: **end für**
24: **end für**
25: **end für**
26: Lösche die Backedge p
27: **end für**

$K_{3,3}$ enthalten ist (siehe beispielsweise Abbildung 2.6(b)). Deswegen könnten durch diese Minortypen identische Kuratowki-Subdivisions für verschiedene kritische Backedges a und b extrahiert werden.

Befinden sich a und b allerdings unter verschiedenen *HighestXYPaths*, dann müssen auch die benötigten z-Knoten in E_2 und AE_2 unterschiedlich sein und auch die Kuratowski-Subdivisions sind verschieden. Identische Kuratowski-Subdivisions könnten also höchstens dann auftreten, wenn a und b unter demselben *HighestXYPath* existieren. Das lässt sich vermeiden, indem für die Minortypen E_2 und AE_2 jeweils nur ein einziger kritischer Backedgepfad pro *HighestXYPath* betrachtet wird. □

2.2.3.2 ShortCircuit-Kanten

Während der Extraktion einzelner Pfade einer $K_{3,3}$- oder K_5-Subdivision ist es möglich, dass ShortCircuit-Kanten gefunden und benutzt werden. Diese sind jedoch nicht Bestandteil des Originalgraphen und müssen deswegen durch die überbrückten Originalpfade ersetzt werden.

Um die überbrückten Originalpfade zu berechnen, können für alle Knoten zusätzlich die ExternalLinks gespeichert werden, die jeweils ohne Einbettungen von ShortCircuit-Kanten gültig wären. Diese Verweise werden *RealLinks* genannt. Mit diesen ist es möglich, zusätzlich auf den tatsächlichen unterliegenden Außenflächenpfad einer ShortCircuit-Kante zuzugreifen und somit auch die vollständigen Kuratowski-Subdivisions zu extrahieren.

Effizienzeinbußen entstehen durch diese Form von Extraktion asymptotisch nicht, da jeder Pfad einer Kuratowski-Subdivision in voller Länge extrahiert werden muss und durch die RealLinks maximal zwei, also konstant viele, zusätzliche Verweise pro Knoten gespeichert werden. Bei jeder Einbettung einer Kante, die keine ShortCircuit-Kante ist, muss allerdings beachtet werden, dass sowohl die ExternalLinks als auch die RealLinks am entsprechenden Knoten aktualisiert werden müssen.

2.3 Globale Erweiterungen

Um auch mehrere Stoppkonfigurationen zu ermöglichen, kann der Planaritätstest global so erweitert werden, dass auch mehrere Stoppkonfigurationen auftreten können. Dafür muss sowohl der Walkup als auch der Walkdown angepasst werden. In jeder auftretenden Stoppkonfiguration können dann effizient Kuratowski-Subdivisions extrahiert werden. Nach der Extraktion müssen alle kritischen Backedges gelöscht werden, um weiterhin auf einer planaren Einbettung arbeiten zu können.

2.3.1 Erweiterter Walkup

Im erweiterten Walkup werden mehrere zusätzliche Datenstrukturen eingesetzt. Sei v der aktuell einzubettende Knoten. Für jede an v endende Backedge werden auf dem Pfad vom Anfangsknoten der Backedge bis zu einem bestimmten DFS-Vorgänger alle besuchten Knoten auf einem Stack gespeichert, anstatt direkt markiert zu werden. Der DFS-Vorgänger ist entweder ein schon besuchter Knoten oder der eindeutige virtuelle Knoten v' des realen Knotens v, der Wurzel des Teilbaums ist, der den Anfangsknoten der Backedge enthält. Wird die Wurzel der momentan traversierten Bicomp erreicht, werden alle Knoten im Stack als besucht markiert und mit einem Link *LinkToRoot* auf den virtuellen Wurzelknoten versehen. So kann von jedem Knoten in $O(1)$ die aktuelle Bicompwurzel berechnet werden. Weiterhin wird der Startknoten jeder pertinenten Backedge mit einem BackedgeFlag markiert, der auf die verbundene Backedge zeigt.

Alle traversierten Knoten der Backedgepfade werden mit v markiert. Die *PertinentRoots*-Liste wird immer dann aktualisiert, wenn ein Knoten einer Elternbicomp erreicht wird. Zusätzlich wird nun aber auch parallel eine Liste *PertinentNodesInWalkup* aller pertinenten Knoten einer Bicomp an dessen Wurzel verwaltet. Diese enthält nur die pertinenten Knoten der Bicomp zur Zeit des Walkups, wird also während der Einbettung von v nicht dynamisch aktualisiert. Die Verwaltung ist leicht mit konstantem Mehraufwand möglich, da alle Knoten mit nicht-leerer *PertinentRoots*-Liste schon besucht worden sind und deswegen über *LinkToRoot* mit der aktuellen Bicompwurzel verlinkt sind. Für nicht besuchte Knoten werden die beiden Listen erst nach Erreichen eines besuchten Knotens oder der Bicompwurzel aktualisiert.

Die Liste PertinentNodesInWalkup an der Wurzel dient später beim Extrahieren einer Kuratowski-Subdivision dazu, involvierte kritische Backedges effizient zu berechnen. Die Summe aller Wurzellisteneinträge eines Walkups ist durch die Anzahl der pertinenten Knoten gegeben. Jeder dieser pertinenten Knoten wurde jedoch vom Walkup traversiert, weswegen die Laufzeit für die Erstellung dieser Listen durch die Laufzeit des Walkups begrenzt werden kann. Im Allgemeinen werden nicht alle Markierungen, Wurzellinks und Listen in den unmittelbar folgenden Walkdowns gelöscht. Stattdessen wird bei folgenden Walkups jeder besuchte Knoten auf die Markierung mit dem aktuell einzubettenden Knoten überprüft und im gegenteiligen Fall alle Datenstrukturen des Walkups an dem Knoten gelöscht.

Auf jeden Walkup können mehr als zwei Walkdowns folgen, im Worst-Case $O(m)$ viele, da es bis zu $O(m)$ virtuelle Knoten v' an v geben kann und für jeden dieser virtuellen Knoten Walkdowns gestartet werden. Für die effiziente Berechnung der kritischen Backedges an Stoppkonfigurationen der Bicomp mit Wurzel v' ist es deswegen nötig, die Menge aller an v endenden Backedges zu partitionieren. Jede Backedge wird dabei der Äquivalenzklasse eines virtuellen Knotens v' von v zugeordnet. v' ist dabei die Wurzel derjenigen Bicomp, die im DFS-Baum Vorgänger der Bicomp ist, die den Backedgestartknoten enthält. Die Partition wird während des Walkups erstellt.

Dazu wird ein Array *HighestVirtualNode* verwaltet, dass zu jeder Backedge den zugehörigen virtuellen Knoten angibt. Wird bei einer Backedge im Walkup der Backedgepfad

bis zu dem eindeutigen virtuellen Knoten v' von v traversiert, kann v' sofort eingetragen werden. Andernfalls muss der Backedgepfad vorzeitig an einem schon besuchten Knoten abgebrochen worden sein. Hier ist vorher der Wert *VisitedWithBackedge* hinterlegt worden, womit auf die frühere besuchende Backedge und damit auf den virtuellen Knoten in $O(1)$ geschlossen werden kann.

Der Wert VisitedWithBackedge muss hierfür an jedem besuchten Knoten im Walkup hinterlegt werden, was aber nur konstanten Mehraufwand verursacht. Nach dem Walkup werden die Informationen des *HighestVirtualNode*-Arrays für jede pertinente Backedge ausgelesen und in eine Liste namens *BackedgesOnVirtualNode* an dem entsprechenden virtuellen Knoten eingetragen. Auch diese Liste dient hinterher der effizienten Berechnung involvierter Backedges beim Extrahieren einer Kuratowski-Subdivision.

2.3.1.1 Laufzeit

Jeder besuchte Knoten kann in $O(1)$ behandelt werden. Zudem wird jeder Knoten bis auf Überlappungen der Backedgepfade höchstens einmal besucht. Die Anzahl der sich in den Pfaden überlappenden Knoten ist durch die Anzahl der Backedges begrenzt. Die Laufzeit des Walkups ist also weiterhin bis auf konstante Faktoren äquivalent zu der Summe der Längen aller traversierten Backedgepfade. Wird eine Backedge eingebettet, ist es möglich, die Traversierungskosten zum Teil durch die neu entstandene Face abzudecken. Insgesamt ist die Summe aller Facekanten durch $2m$ linear begrenzt. Allerdings kann nun nicht mehr erwartet werden, dass dadurch alle Traversierungskosten abgedeckt sind, denn es werden bei jeder Kuratowski-Extraktion kritische Backedges gelöscht, welche dann auch nicht mehr eingebettet werden können. Diese Backedges verursachen Traversierungskosten, die anders abgedeckt werden müssen.

Um festzuhalten, welche Kosten wodurch abgedeckt werden, wird die Buchhaltermethode mit drei Konten eingesetzt: Das *Face-Konto* deckt die Traversierungskosten durch neu entstehende Faces ab, das *ShortCircuit-Konto* durch eingefügte ShortCircuit-Kanten und das *Kuratowski-Konto* durch extrahierte Kuratowski-Subdivisions. Wenn jedes Konto über alle Knoteneinbettungen hinweg insgesamt lineare Zeit braucht, ist auch der Walkup in Linearzeit möglich.

Die Walkups einer einzelnen Knoteneinbettung verbrauchen im Worst-Case allerdings zuviel, nämlich Zeit $\Omega(m)$. Dieser Mehraufwand wird hinterher aber kompensiert, weswegen hier die Gesamtlaufzeit über alle Knoten betrachtet wird: Mit dem obigen Ansatz können alle eingebundenen Kanten durch das Face-Konto durch Kosten $O(n+m)$ beschränkt werden, da diese nicht nochmals auf der Face-Innenseite traversiert werden. Die ShortCircuit-Kanten können im Walkup zwar durchlaufen und dazu genutzt werden, einen vorigen, im Walkdown verursachten Mehraufwand zu kompensieren, aber es entstehen keine neuen Kosten. Alle verbleibenden Kosten werden auf das Kuratowski-Konto gebucht, für das dann noch maximal Kosten von $O(n+m+\sum_{K\in S}|E(K)|)$ entstehen dürfen. Dazu werden vorab noch zwei Lemmata benötigt:

Lemma 2.3.1. *Sei v der einzubettende Knoten. Die Walkup-Traversierungskosten auf*

Backedgepfaden gelöschter Backedges auf den inneren Faces aller im Verlauf des Algorithmus vorhandenen Forebear-Bicomps sind durch $O(m)$ beschränkt.

Beweis. Betrachtet wird jeder Knoten v direkt nach seiner Einbettungsphase. Zu diesem Zeitpunkt wurden bereits alle Backedges an v eingebettet, außer denen, die Kuratowski-Subdivisions verursacht haben. Jede Einbettung einer Backedge hat dann genau eine innere, an v grenzende Face erzeugt. Umgekehrt ist auch jede innere, an v grenzende Face der Forebear-Bicomp durch eine Backedge an v eingebettet worden. Die Angrenzung der Faces an v ist dabei zentral, denn diese Aussage gilt nicht für die inneren Faces der vorherigen, mittlerweile eingebetteten Bicomps. Alle im Walkup markierten Backedgepfade sind paarweise kantendisjunkt, weswegen sie zusammengenommen in den inneren, an v grenzenden Faces vollständig enthalten sind. Eine einmal eingebettete, innere Face wird im weiteren Verlauf weder aufgebrochen, noch erneut traversiert. Folglich sind die Gesamtkosten über alle Knoteneinbettungen durch die Anzahl der Facekanten, also durch $2m = O(m)$ begrenzt. □

Lemma 2.3.2. *Seien $c(v)$ die Walkup-Traversierungskosten auf Backedgepfaden gelöschter Backedges auf der Außenfläche der Forebear-Bicomp bei Einbettung des Knotens v. Sei $C = \sum_{v \in V} c(v)$ die Summe dieser Kosten über alle Knoteneinbettungen. Dann gilt:*

$$C = O(n + m + \sum_{K \in S} |E(K)|).$$

Beweis. Sei v der aktuell einzubettende Knoten. Sei weiterhin z der erste vom Walkup erreichte Knoten, der in der Forebear-Bicomp enthalten ist und D die ehemalige Bicomp innerhalb der Forebear-Bicomp, die z enthält. Die vom Walkup traversierte Außenfläche der Forebear-Bicomp wird in den Teil, der in D enthalten ist und den potentiell vorhandenen, restlichen Teil getrennt (siehe Abbildung 2.8). Sei a der letzte Knoten in D auf der Außenfläche der Forebear-Bicomp, den der Walkup ausgehend von z traversiert hat. Der Pfad $z \rightarrow a$ stellt dabei im Allgemeinen nicht den gesamten, vom Walkup traversierten Pfad dar, da hier durch Lemma 2.3.1 potentielle Kanten innerhalb der Forebear-Bicomp nicht betrachtet werden müssen. Sei b der von der Kuratowski-Extraktion zuletzt besuchte Knoten auf der Außenfläche der Forebear-Bicomp in D. Dabei ist $a = b$ nicht ausgeschlossen.

Da nur die Kosten auf den Außenkanten der Bicomp D betrachtet werden, existieren jetzt zwei Fälle: Einerseits kann der Pfad $z \rightarrow a$ im extrahierten Pfad $z \rightarrow b$ enthalten sein, dann sind die Walkup-Kosten durch diese Überlappung abgedeckt, oder die beiden Pfade verlaufen in entgegengesetzte Richtungen. In letzterem Fall werden die Kosten der Kanten in $z \rightarrow a$ nicht durch die Kuratowski-Extraktion kompensiert, da keine Kante von dieser überdeckt wird. Allerdings muss dann der alternative Walkuppfad von z aus in Richtung b mindestens so lang wie der Pfad $z \rightarrow a$ sein, da er vom Walkup letztendlich nicht gewählt wurde. Damit können alle Walkup-Traversierungskosten einerseits durch die Extraktion des Pfades $z \rightarrow b$ und andererseits durch potentiell vorhandene

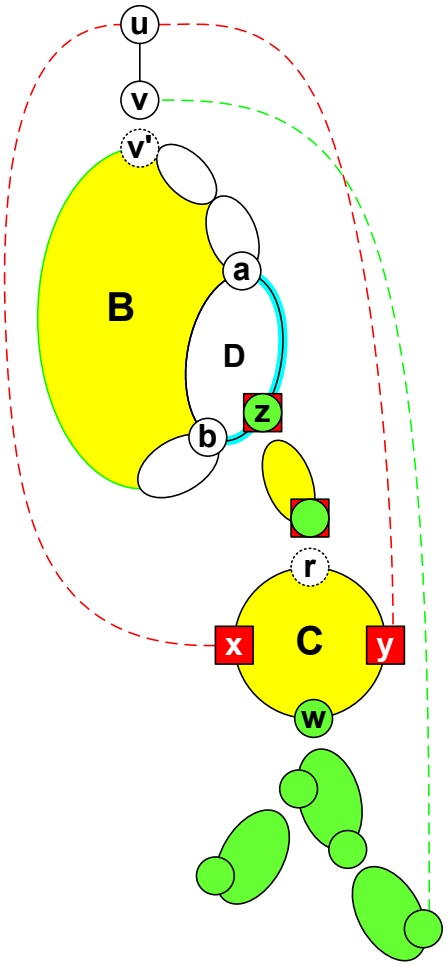


Abbildung 2.8: Beispiel des extrahierten Minortyps A mit der Forebear-Bicomp B und der NonForebear-Bicomp C, in der eine Stoppkonfiguration gefunden wurde. Der Walkup hat die weiß gefüllten, ehemaligen Bicomps traversiert, welche mittlerweile in die Forebear-Bicomp eingebettet worden sind. Der Pfad $z \rightarrow a$ wurde vom Walkup durchlaufen, während die Kuratowski-Extraktion den Pfad $z \rightarrow b$ gewählt hat.

Innenkanten der Forebear-Bicomp kompensiert werden. Über alle Knoteneinbettungen belaufen sich die Extraktionskosten auf $O(n + \sum_{K \in S} |E(K)|)$, während die Kosten auf den Innenkanten nach Lemma 2.3.1 durch $O(m)$ beschränkt sind. Es ergeben sich also maximal Gesamtkosten von $O(n + m + \sum_{K \in S} |E(K)|)$.

Es verbleiben die Außenflächen der Vorgängerbicomps von D, auf denen Kosten des Walkups aufgrund der gelöschten Backedge entstanden sind. Von diesen Vorgängern müssen nur diejenigen betrachtet werden, deren Kosten durch die zugehörige Kuratowski-Extraktion nicht aufgefangen wurden. Im Worst-Case kann das jedoch auf alle Vorgängerbicomps zutreffen. Die Forebear-Bicomp kann nicht entartet sein, da sie andernfalls keine Einbettung und damit auch keine Vorgängerbicomps haben könnte. Auf jeder Vorgängerbicomp hat der Walkup einen Pfad gewählt, der bei einem eindeutigen, zuerst erreichten Knoten beginnt. Zu diesem Zeitpunkt war der Pfad in der entgegengesetzten Richtung mindestens so lang wie der gewählte. Da die Forebear-Bicomp zweizusammenhängend und in diesem Fall nicht entartet ist, muss dieser alternative Pfad auf einer inneren Face der Forebear-Bicomp liegen, die an v grenzt. Damit können die verbleibenden Kosten durch die Anzahl der innere Facekanten und damit nach Lemma 2.3.1 durch $O(m)$ abgeschätzt werden. Zusammen ergibt sich daraus bei Betrachtung aller Knoteneinbettungen im Algorithmus die Behauptung. □

Mit den obigen Abschätzungen der Walkupkosten an der Forebear-Bicomp kann jetzt die folgende Aussage über das Kuratowski-Konto bewiesen werden:

Theorem 2.3.3. *Die Kosten des Kuratowski-Kontos für den Walkup können durch*

$$O(n + m + \sum_{K \in S} |E(K)|)$$

abgeschätzt werden.

Beweis. Problematisch sind alle Traversierungskosten, die weder durch das Face-Konto noch durch das ShortCircuit-Konto abgedeckt werden. Betrachtet wird die Menge aller Backedgepfade im Walkup, die später keine Einbettungen, aber Kuratowsi-Subdivisions erzeugen. Für jeden dieser Backedgepfade werden zwei Fälle unterschieden, je nachdem, ob sich die Stoppkonfiguration innerhalb einer Forebear- oder NonForebear-Bicomp befindet. In beiden Fällen wird o. B. d. A. der Worst-Case für die Pfadlängen angenommen. Da Backedgepfade, die vor dem virtuellen Knoten $v' := \mathit{HighestVirtualNode}(w)$ enden, nur einen Teil der Kosten verursachen, folgt die Aussage dort analog.

- NonForebear-Bicomp:
  Mit der betrachteten Backedge können die Minortypen AB, AC, AD und $AE1$–$AE4$ gefunden werden. In jedem Fall induziert sie jedoch den Minortyp A (siehe Abbildung 2.5(a)), welcher auch während des Walkdowns extrahiert wird. Es wird vorerst angenommen, dass dieser effizient extrahiert werden kann und dabei für die einzelnen Kuratowski-Pfade bei jeder traversierten Bicomp mittels Parallelisierung der kürzeste Pfad auf der Außenfläche gewählt wird. Dieser Ansatz, die

Kuratowski-Pfade zu berechnen, ist analog zur Methode im Walkup, so dass die ausgewählten Pfade während der Extraktion in den noch nicht eingebetteten Bicomps mindestens so viele Kosten wie dort verursachen. Sie können auch mehr Kosten verursachen, da der Walkup von vorherigen Backedgepfaden und deren besuchten Knoten profitieren kann. Dadurch können die Kosten für die nicht eingebetteten Bicomps bezahlt werden, die eingebetteten Bicomps müssen allerdings anders abgeschätzt werden.

Bemerkung. Natürlich wäre es auch möglich, alle vom Walkup traversierten Pfade zu speichern und dann genau diese Pfade für die Extraktion der Kuratowski-Subdivision zu benutzen, egal ob die zugehörigen Bicomps eingebettet oder nicht eingebettet sind. In diesem Fall würden nur die Kosten auftreten, die im Walkup verbraucht worden sind. Allerdings werden immer Kuratowski-Subdivisions mit möglichst geringer Kantenanzahl bevorzugt. Falls bereits Bicomps in die Forebear-Bicomp eingebettet worden sind, wäre dadurch die beschriebene Neuberechnung der einzelnen Kuratowski-Pfade erfolgversprechender, weil die erzeugten Kuratowski-Subdivisions häufig kleiner sind. Zusätzlich könnten hier auch die jeweils kleinsten oder auch sämtliche Kuratowski-Subdivisions beider Varianten berechnet werden, ohne die lineare Laufzeit aufgeben zu müssen.

Der Backedgepfad wird mit den Kanten des extrahierten Minors A verglichen. Dazu wird er in vier Abschnitte unterteilt (siehe Abbildung 2.8):

1. vom Startknoten der Backedge bis zu w (möglicherweise leer).
2. von w bis zur Wurzel r der NonForebear-Bicomp.
3. von r bis zum ersten Knoten z der eindeutigen Forebear-Bicomp. (möglicherweise leer)
4. von z bis zu v'.

Der erste Pfadabschnitt ist nicht teurer als die entsprechenden Kanten, die im Minor A extrahiert werden. Der zweite Pfadabschnitt ist eine Teilmenge der Außenfläche, die beim Minor A an dieser Bicomp vollständig extrahiert wird. Die Bicomps des dritten Teilpfades, sofern dieser nicht leer ist, können nicht eingebunden worden sein, andernfalls wären diese in die Forebear-Bicomp integriert worden. Dadurch wird auch hier ein Pfad extrahiert, der mindestens so lang wie der im Walkup traversierte ist. Die Walkupkosten der ersten drei Pfadabschnitte werden also durch die spätere Extraktion amortisiert werden.

Der letzte Pfadabschnitt betrachtet die Traversierung auf der Forebear-Bicomp. Die Bicomp mit Wurzel r ist extern, da sie zwei Stoppknoten enthält. Zum Zeitpunkt des Walkups sind damit rekursiv auch alle Vorgängerbicomps extern. Folglich ist auch der Knoten z extern und liegt auf der Außenfläche der Forebear-Bicomp. Ausgehend von z existiert eine eindeutige, potentiell leere Folge von ehemaligen Vorgängerbicomps bis hin zum Knoten v'. Der Backedgepfad verteilt sich nun auf die Innenseiten und Außenseiten dieser ehemaligen Bicomps. Dabei müssen

die Vorgängerbicomps nicht zwangsläufig eine Außenfläche haben. Im Allgemeinen können sie im Verlauf des Walkdown von anderen Einbettungen überdeckt worden sein, so dass sie sich vollständig im Inneren der Forebear-Bicomp befinden.

Nach Lemma 2.3.1 können sämtliche Kosten im Innern aller Forebear-Bicomps über alle Knoteneinbettungen durch $O(m)$ beschränkt werden. Die verbleibenden Kosten, die an den Außenseiten der Forebear-Bicomp auftreten, können mit Lemma 2.3.2 durch $O(n + m + \sum_{K \in S} |E(K)|)$ begrenzt werden, woraus die Behauptung folgt.

- Forebear-Bicomp:
  Da die betrachtete Stoppkonfiguration in der Forebear-Bicomp enthalten ist, kann der betrachtete Backedgepfad des Walkups in folgende zwei Abschnitte gegliedert werden:

  1. vom Startknoten der Backedge bis zu w (möglicherweise leer)
  2. von w bis zur Wurzel v' der Forebear-Bicomp

  Die im ersten Abschnitt traversierten Bicomps können nicht eingebettet worden sein, da dieser am kritischen Knoten w zwischen den beiden Stoppknoten endet. Alle auf diesen Bicomps während der Kuratowski-Extraktion extrahierten Pfade werden durch paralleles Traversieren der Außenflächen in beide Richtungen gewonnen. Dadurch werden auf diesen Bicomps mindestens so viele Kanten traversiert, wie vom Walkup Kosten erzeugt wurden. Die Kosten dieses Abschnittes können also vollständig über den extrahierten kritischen Backedgepfad des Knotens w abgedeckt werden.

  Allerdings wird dieser Backedgepfad nicht immer gebraucht: In dem Minortyp E_2 ist der kritische Backedgepfad und die kritische Backedge selbst nicht Bestandteil des unterliegenden $K_{3,3}$ und wird deswegen auch nicht extrahiert. Dadurch können die Walkupkosten auf dem gesamten kritischen Backedgepfad nicht abgedeckt werden. Im Worst-Case könnte das bei vielen vorhandenen E_2 Minoren in dem Graphen zu einer quadratischen Laufzeit führen. Allerdings ist dafür notwendige Voraussetzung, dass E_2 immer einziger auftretender Minortyp ist, da jeder andere Typ den kritischen Backedgepfad benötigt.

  Das sogenannte *E_2-Konto* wird erstellt, das alle Walkup-Traversierungskosten verwaltet, die wegen eines solchen E_2-Minors nicht kompensiert werden können. Dann lässt sich zeigen, dass dessen Kosten linear in der Kantenanzahl beschränkt sind: Die kritischen Backedgepfade, die auf das E_2-Konto gezahlt werden, können nach Voraussetzung keine anderen Minortypen erzeugen. Deswegen kann kein Knoten des Backedgepfades extern aktiv sein, da sonst entweder ein Minor des Typs B oder des Typs E existieren würde. Alle Nachfolgerbicomps von w können deswegen nicht extern aktiv sein. Diese Bicomps konnten zudem bisher wegen der Stoppkonfiguration nicht eingebettet werden. Da aber keine Bicomp extern aktiv ist, werden diese Bicomps auch in späteren Einbettungsphasen nicht eingebettet. Damit ist jede Kante des E_2-Kontos Bestandteil von genau einem kritischen Backedgepfad

und das E_2-Konto kann, wie behauptet, durch $O(m)$ beschränkt werden.

Wir betrachten jetzt den zweiten Abschnitt des Backedgespfades in der Forebear-Bicomp. Sei G die ehemalige, jetzt eingebettete Bicomp, die w enthält. G hat eine eindeutige, möglicherweise leere Folge von ehemaligen Vorgängerbicomps im DFS-Baum, die am virtuellen Knoten v' endet. Alle Kosten des Backedgepfades auf diesen Vorgängerbicomps können analog zu den Lemmata 2.3.1 und 2.3.2 (zweiter Absatz) durch den Walkdown begrenzt werden. Die gleiche Argumentation folgt für etwaige Kosten auf der Innenseite der Bicomp G.

Es verbleibt die Analyse für die Außenfläche der eingebettete Bicomp G: Während der Kuratowski-Extraktion wird mindestens ein Minor der hier möglichen Typen B, C, D (siehe Abbildung 1.8) oder E_1–E_5 (siehe Abbildung 1.9) extrahiert. Im Unterschied zu der Analyse der NonForebear-Bicomp wird hier aber nicht zwangsläufig immer ein bestimmter Minor extrahiert, der die Außenfläche vollständig überdeckt. Die im Walkup entstandenen Kosten müssen also mit allen möglichen Minortypen verglichen werden:

Befinden sich die Minortypen B, E_2 oder E_5 $(= K_5)$ unter den extrahierten Minoren, wird in mindestens einem die Außenfläche vollständig extrahiert. Dadurch sind die Kosten des Walkups auf der Außenfläche von G abgedeckt. Befindet sich andererseits einer der Typen C, D, E_1, E_3 oder E_4 unter den extrahierten Minoren, existiert in mindestens einem Minor der $\mathit{HighestXYPath}(w)$, der auch extrahiert wird. Nach Theorem 2.1.4 entspricht dieser genau dem Teil der Außenfläche von G innerhalb der Forebear-Bicomp. Die beiden Endknoten p_x und p_y des $\mathit{HighestXYPath}(w)$ entsprechen deswegen den beiden Endknoten der Außenfläche von G.

Einem Teil der betrachteten Minortypen, nämlich C, D und E_1, ist gemeinsam, dass sie die Außenfläche der Forebear-Bicomp von w bis p_x und von w bis p_y extrahieren. Da diese Endknoten sind, wird dadurch mit dem $\mathit{HighestXYPath}(w)$ die gesamte Außenfläche von G extrahiert, womit alle verbleibenden Kosten abgedeckt sind. Es verbleiben die Minortypen E_3 und E_4. In diesen wird lediglich einer der beiden Pfade $w \to p_x$ oder der Pfad $w \to p_y$ extrahiert. Die Kosten des Walkups auf der Außenfläche in G entsprechen aber maximal der Länge des kürzesten beider Pfade. Insofern werden die Walkupkosten unabhängig von der Wahl des extrahierten Pfades kompensiert.

□

Korollar 2.3.4. Der erweiterte Walkup benötigt für alle Knoteneinbettungen insgesamt Laufzeit $O(n + m + \sum_{K \in S} |E(K)|)$.

2.3.2 Erweiterter Walkdown

Der erweiterte Walkdown unterscheidet sich vom originalen Walkdown dadurch, dass er bei einer Stoppkonfiguration nicht abbricht. Deswegen muss eine effiziente Methode gefunden werden, die den Walkdown an einem Knoten fortführt, an dem die Chance auf Einbettung aller verbleibenden Backedges besteht. Es werden die drei Fälle unterschieden, dass der Walkdown keine Stoppkonfiguration, eine Stoppkonfiguration ohne kritische Backedges oder eine kritische Stoppkonfiguration findet.

1. Wird keine Stoppkonfiguration gefunden, die zur Extraktion einer Kuratowski-Subdivision führt, ist der Walkdown identisch zum Originalalgorithmus von Boyer und Myrvold. Die entstehenden Traversierungskosten werden dabei zum großen Teil auf das Face-Konto verrechnet. Da jede neu gebildete Face aus Kanten besteht, die der Walkdown traversiert hat, kann der Betrag des Face-Kontos linear begrenzt werden. Im Allgemeinen sind dort aber nicht alle vom Walkdown traversierten Kanten enthalten, da dieser an jeder erreichten Bicompwurzel parallel beide Pfade der Außenfläche entlangläuft, bis in jeder Richtung ein aktiver Knoten gefunden worden ist. Wird eine Backedge eingebettet, können mit der entstehenden Face nur die Kosten einer der beiden vorigen Traversierungsrichtungen abgedeckt werden. Die Kosten der anderen Traversierungsrichtung werden auf das ShortCircuit-Konto verrechnet. Dieses muss später linear begrenzt werden.
2. Wird eine nicht-kritische Stoppkonfiguration gefunden, hält der Walkdown in einer Bicomp A. Da in A keine kritischen Backedges existieren, müssen alle Backedges, die in dem aktuellen Walkdown eingebettet werden sollen, bereits eingebettet worden sein und der Walkdown ist abgeschlossen.
3. Andernfalls hält der Walkdown an einer kritischen Stoppkonfiguration. In dieser wird für jede kritische Backedge mindestens eine Kuratowski-Subdivision extrahiert. Um die bisherige Einbettung weiterhin planar zu halten, werden danach alle kritischen Backedges der Stoppkonfiguration gelöscht.

   Zusätzlich werden während der Extraktion sowohl alle *PertinentRoots*-Listen an den kritischen Knoten als auch an den Wurzelknoten der Bicomps der entsprechenden Backedgepfade gelöscht. Damit sind sämtliche Knoten zwischen den Stoppknoten nicht mehr kritisch. An A können weiterhin keine anderen pertinenten Knoten existieren, da der Walkdown sonst nicht an A geendet hätte. Deswegen ist auch A selbst nicht mehr pertinent. Sei r die virtuelle Wurzel von A. Dann wird, um die Konsistenz aufrecht zu erhalten, zusätzlich der Eintrag aus der *PertinentRoots*-Liste an dem realen Knoten von r gelöscht, der auf A verweist.

   Falls in dem aktuellen Walkdown des virtuellen Knotens v' keine weitere pertinente Backedge e mit $\mathit{HighestVirtualNode}(e) = v'$ existiert, ist der Walkdown beendet. Insbesondere ist er dann beendet, wenn A eine Forebear-Bicomp ist, da dann nach der Kuratowski-Extraktion keine pertinente Backedges mehr existieren können. Andernfalls ist A eine NonForebear-Bicomp und es muss versucht werden, die verbleibenden Backedges einzubetten. Dazu wird der Zustand hergestellt, den der Walkdown hätte, wenn die Bicomp A vom Walkup nicht als pertinent markiert

worden wäre.

Für den Fall 3 klärt folgender Abschnitt, wie der Walkdown nach Extraktion einer Kuratowski-Subdivision korrekt fortgeführt werden kann.

2.3.2.1 Korrektheit

Sei T der durch die DFS-Traversierung eindeutig bestimmte, an der Forebear-Bicomp gewurzelte Bicompbaum, dessen Bicomps pertinent sind und vom Walkdown erreicht werden (siehe Abbildung 2.9). Im Allgemeinen müssen nicht alle pertinenten Bicomps vom Walkdown erreicht werden, da sie in der Nachfolge eines kritischen Knotens liegen können, vor dem aufgrund von Stoppknoten angehalten wird. Ein Walkdown heißt genau dann *korrekt*, wenn er ausschließlich die Backedges nicht einbettet, die bei ihrer Einbettung Kuratowski-Subdivisions verursachen würden.

Theorem 2.3.5. *Sei $T \neq \emptyset$ und A eine Bicomp, an der eine Kuratowski-Extraktion gestartet wurde, aufgrund derer A nicht mehr pertinent ist. Sei $N(A)$ die erste pertinente Bicomp in T auf dem eindeutigen Bicomppfad von A bis zur Wurzel von T. Wenn der erweiterte Walkdown auf der Bicomp $N(A)$ an deren zuletzt besuchtem Knoten in derselben Richtung fortfährt, ist er korrekt.*

Beweis. Vorerst wird angenommen, dass während des Walkdowns keine Bicomp aus T eingebettet werden kann. Die erste kritische Stoppkonfiguration des Walkdowns muss dann an einem Blatt A von T gefunden werden. In T wird der Pfad von der Wurzel bis zur Bicomp A betrachtet: Da nach Annahme auf diesem Pfad keine Bicomp eingebettet wurde, sind sämtliche Startknoten pertinenter Backedges unerreichbar gewesen. Ausgehend davon lässt sich jede Bicomp als *aktiv* oder *passiv* klassifizieren:

- *Aktive* Bicomps enthalten den Startknoten mindestens einer pertinenten Backedge. Da davon jedoch keine eingebettet werden konnte, muss in jeder aktiven Bicomp eine Stoppkonfiguration existieren.
- *Passive* Bicomps enthalten keine Startknoten von pertinenten Backedges und werden deswegen nur traversiert, um nachfolgende Bicomps mit solchen Startknoten zu erreichen.

Die Aktivität von Bicomps ist ein anderen Konzept als die Aktivität von Knoten. So können beispielsweise passive Bicomps im Gegensatz zu passiven Knoten noch immer pertinent sein.

Nach Extraktion der involvierten Kuratowski-Subdivisions ist A passiv und nicht mehr pertinent (siehe Abbildung 2.9). Damit sind auch alle passiven Vorgängerbicomps in T nicht mehr pertinent, sofern an diesen kein aktiver Nachfolger existiert. Da der Walkdown nur pertinente Bicomps betrachten muss, wird solange zur jeweiligen Elternbicomp in T aufgestiegen, bis die erste pertinente Bicomp $N(A)$ erreicht worden ist. Diese Bicomp wurde vom Walkdown ehemals wegen der Pertinenz von Bicomp A verlassen. Die dabei involvierten pertinenten Backedges und Pertinenzinformationen sind aber gelöscht worden, da A nicht eingebettet werden konnte.

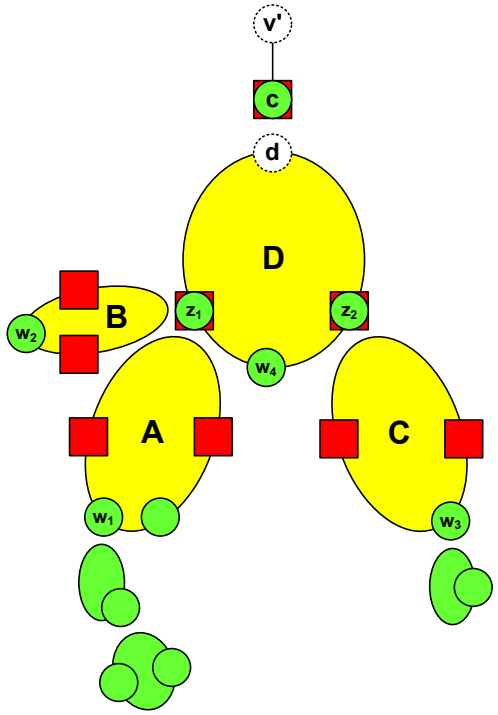


Abbildung 2.9: Beispiel eines CCW-Walkdowns, der wegen einer kritischen Stoppkonfiguration an Bicomp A stoppt. Der Bicompbaum T besteht hier aus der Forebear-Bicomp und den Bicomps A, B, C und D. Nach Extraktion der Kuratowski-Subdivisions an A und Löschen der involvierten kritischen Backedges ist A nicht mehr pertinent. Der Walkdown sucht im Bicompstack nach dem letzten besuchten Knoten der Vorgängerbicomp $N(A) = D$, findet z_1 und versucht deswegen Bicomp B einzubetten. Aber auch hier existiert eine Stoppkonfiguration. Nach Extraktion der Kuratowski-Subdivisions ist der Knoten z_1 nicht mehr pertinent und der Walkdown müsste stoppen. Da D aber noch immer pertinent ist, werden die beiden vorher erzeugten ShortCircuit-Kanten $d \rightarrow z_1$ und $d \rightarrow z_2$ in konstanter Zeit traversiert. Gefunden wird der pertinente Knoten z_2, woraufhin in Bicomp C abgestiegen wird und wieder alle Kuratowski-Subdivisions extrahiert werden. Beim Überprüfen der beiden ShortCircuit-Kanten in Bicomp D werden danach keine pertinenten Knoten mehr gefunden. Dadurch liegt auch hier eine Stoppkonfiguration vor, und die durch den Knoten w_4 verursachten Kuratowski-Subdivisions werden extrahiert.

$N(A)$ ist nun entweder selbst aktiv oder besitzt eine aktive Nachfolgerbicomp. In beiden Fällen kann es vorkommen, dass der Knoten z_1 in $N(A)$, an dem ursprünglich zu A abgestiegen wurde, weiterhin wegen aktiver Nachfolgerbicomps pertinent ist. Dann werden alle diese Nachfolgerbicomps analog zu Bicomp A behandelt. Sei B die letzte dieser Nachfolgerbicomps. Falls $N(B) \neq N(A)$, dann wiederholt sich der gesamte Vorgang auf Bicomp $N(B)$.

Andernfalls ist $N(B) = N(A)$ und z_1 ist nicht mehr pertinent, aber extern aktiv, da beispielsweise in Bicomp A Stoppknoten existieren. Der originale Walkdown würde hier an Knoten z_1 stoppen. Allerdings kann es sein, dass der Pfad auf der Außenfläche der Bicomp $N(A)$, der von der Wurzel der Bicomp in die entgegengesetzte Richtung zu z_1 verläuft, an einem pertinenten Knoten z_2 endet. Dann hätte ein Walkdown, an dem Knoten z_1 nicht pertinent gewesen wäre, während des Abstiegs zur Bicomp $N(A)$ den Pfad in Richtung z_2 statt z_1 gewählt. Also muss der Walkdown auf dem Knoten z_2 fortgeführt werden, falls dieser pertinent ist. Dies ist der einzige Knoten auf den ShortCircuit-Kanten, der noch pertinent sein kann, denn in beiden Richtungen der Außenfläche müssen externe Knoten erreicht worden sein und sämtliche vom Walkdown besuchte Knoten auf dem Pfad von der Wurzel von $N(A)$ nach z_1 können nicht mehr pertinent sein.

Wenn der Walkdown alle Nachfolgerbicomps an z_1 und z_2 traversiert hat, können die Endknoten nicht mehr pertinent sein. Allerdings stellen sie eine Stoppkonfiguration dar, weswegen überprüft werden muss, ob sich zwischen ihnen kritische Knoten befinden, da dann wiederum Kuratowski-Subdivisions extrahiert werden müssen. Ist dies erfolgt, wird die Methode iteriert und die nächste pertinente Bicomp $N(N(A))$ gesucht.

Die Korrektheit des Boyer-Myrvold-Algorithmus besagt, dass erstens alle eingebetteten Backedges keine Kuratowski-Subdivisions erzeugt haben und zweitens die pertinente Backedge, die als erste nicht eingebettet worden konnte, eine Kuratowski-Subdivision erzeugt. Die zweite Aussage wird durch den erweiterten Walkdown auf alle pertinenten Backedges der Einbettungsphase an v verallgemeinert, die nicht eingebettet werden können. Im erweiteren Walkdown werden alle Nachfolgerbicomps von z_1 nach Extraktion der zugehörigen Kuratowski-Subdivisions nicht mehr betrachtet. An den restlichen Bicomps aus T werden keine Änderungen vorgenommen.

Zusätzlich wird im Nachhinein genau die Richtung in der Bicomp $N(A)$ gewählt, den der originale Walkdown auf einem Graphen ohne die Nachfolgerbicomps von z_1 gewählt hätte. Somit wird der Walkdown immer an den Knoten weitergeführt, die ohne die Kuratowski-Backedges traversiert würden. Der erweiterte Walkdown stoppt also nicht nach der ersten nicht einbettbaren Backedge, sondern traversiert einen Restgraphen, auf dem im Folgenden so eingebettet wird, als würde diese Backedge nicht existieren.

Es verbleibt der Fall, dass während des erweiterten Walkdowns eine Bicomp eingebettet wird. Sei F diese Bicomp. F muss in T enthalten gewesen sein, da F pertinent war. Damit werden alle Bicomps auf dem Pfad in T von F bis zur Forebear-Bicomp verschmolzen. Der Walkdown fährt dann auf der vergrößerten Forebear-Bicomp am entsprechenden Knoten fort. Falls im Folgenden alle Backedges eingebettet werden können, erzeugen diese keine Kuratowski-Subdivisions und die Aussage folgt sofort. Andernfalls existiert

im aktualisierten Bicompbaum T wieder eine Bicomp mit einer kritischen Stoppkonfiguration, die vom Walkdown als Erste erreicht wird und obiges Argument kann wiederholt werden. Damit ist der erweiterte Walkdown insgesamt korrekt. □

2.3.2.2 ShortCircuit-Kanten

Genutzt werden ShortCircuit-Kanten im Walkup und Walkdown, erzeugt werden sie aber ausschließlich im Walkdown, weswegen nur dort Kosten entstehen können. Im Originalalgorithmus werden ShortCircuit-Kanten dann eingefügt, nachdem ein Pfad von dem einzubettenden Knoten v zu einem Stoppknoten traversiert wurde. Die Kosten dieser Traversierung sind im Allgemeinen nicht durch entstehende Faces abgedeckt und werden deswegen auf das ShortCircuit-Konto verrechnet. Würde die ShortCircuit-Kante nicht erzeugt werden, wäre eine wiederholte Traversierung des Pfades so kostspielig, dass insgesamt keine lineare Laufzeit mehr garantiert werden könnte. ShortCircuit-Kanten ermöglichen also wiederholte Traversierungen durch einen einmaligen Mehraufwand. Das ShortCircuit-Konto kompensiert diesen Mehraufwand im Nachhinein zugunsten einer amortisiert linearen Laufzeit.

Durch die Möglichkeit, den Planaritätstest nach gefundenen Kuratowski-Subdivisions weiterzuführen, reicht dieses Konzept allerdings nicht mehr aus. An einer Bicompwurzel existieren maximal zwei ShortCircuit-Kanten, jeweils höchstens eine in jede Richtung auf der Außenfläche. Wenn der Walkdown zu einer Wurzel einer Kinderbicomp B absteigt, müssen die aktiven Knoten von B beider Richtungen gefunden werden. Während dies im Originalalgorithmus durch die ShortCircuit-Kanten in $O(1)$ realisierbar ist, kann es hier vorkommen, dass ShortCircuit-Kanten infolge von Kantenlöschungen auch an inaktiven Knoten enden. In dem Fall wird dieser Algorithmus so erweitert, dass er die ShortCircuit-Kante von der Bicompwurzel r bis zu dem Knoten a verlängert, welcher der nächste aktive Knoten der gewählten Richtung ist.

Falls die Kosten der Traversierung des Pfades nicht andersweitig abgedeckt werden, entstehen für das ShortCircuit-Konto die Kosten $|r \to a|$, was proportional zur Länge des überbrückten Pfades ist. Für die Verlängerung der ShortCircuit-Kante bei bekanntem a entstehen dabei nur konstante Kosten pro Wurzelknoten. Diese verschlechtern die asymptotische Laufzeit des Algorithmus nicht. Im Folgenden wird gezeigt, dass sich die Kosten von ShortCircuit-Kanten im Allgemeinen amortisieren. Die Korrektheit des Walkups und Walkdowns bei Verwendung der ShortCircuit-Kanten ist dabei dadurch sichergestellt, dass sämtliche Knoten innerhalb des abgekürzten Pfades inaktiv sind.

Theorem 2.3.6. *Die Kosten des ShortCircuit-Kontos sind durch $O(n+m)$ beschränkt.*

Beweis. Die ShortCircuit-Kanten lassen sich zwei disjunkten Mengen zuordnen. Die Menge A enthält alle ShortCircuit-Kanten, die nach der Erzeugung von einem Walkup oder Walkdown traversiert werden. Die Menge B besteht aus den verbleibenden ShortCircuit-Kanten, die nach deren Erzeugung nicht traversiert werden. Beide Mengen werden getrennt abgeschätzt:

- ShortCircuit-Kanten der Menge A:
  Für jede ShortCircuit-Kante gilt, dass der gesamte überbrückte Pfad bei einer späteren Traversierung in $O(1)$ abgekürzt wird. Bis auf diesen konstanten Aufwand wird das ShortCircuit-Konto dadurch um genau den Betrag reduziert, den der Mehraufwand vorher verursacht hatte. Wird dieselbe ShortCircuit-Kante mehrfach traversiert, verringern sich die Kosten sogar weiter, so dass der konstante Aufwand für das Traversieren der ShortCircuit-Kante selbst kompensiert wird. Damit wächst das ShortCircuit-Konto durch jede Kante aus A um maximal $O(1)$. Es bleibt zu zeigen, dass die Größe von A linear beschränkt ist: In jeder Bicomp existieren nie mehr als zwei ShortCircuit-Kanten. Da maximal n Bicomps existieren können, ist die Anzahl der ShortCircuit-Kanten in A durch $2n$ begrenzt. Folglich ist $|A| = O(n)$.
- ShortCircuit-Kanten der Menge B:
  ShortCircuit-Kanten aus B können sich nicht gegenseitig enthalten, da sonst mindestens eine dieser Kanten traversiert werden müsste. Sie können sich ferner nicht gegenseitig überlappen, da sie sonst Kreuzungen verursachen würden, was ein Widerspruch zur planaren Einbettung wäre. Da sich die ShortCircuit-Kanten aus B also weder gegenseitig enthalten noch überlappen, überbrücken sie disjunkte Mengen von Kanten. Daher ist die Anzahl involvierter Kanten linear begrenzt und diese verursachen höchstens die Kosten $O(m)$ auf dem ShortCircuit-Konto.

Zusammen folgt daraus die obere Schranke $O(n + m)$ für die Kosten des ShortCircuit-Kontos. □

2.3.2.3 Laufzeit

Die Kosten des Face-Kontos im Walkdown und die Kosten des ShortCircuit-Kontos sind linear begrenzt. Übrig bleiben die Traversierungskosten an den Bicomps, die nicht eingebettet werden konnten und die Kosten für die Suche nach der nächsten, pertinenten Bicomp, auf der der Walkdown fortgeführt werden muss.

Lemma 2.3.7. *Die Traversierungskosten des erweiterten Walkdowns an nicht einbettbaren Bicomps sind insgesamt durch $O(n + m + \sum_{K \in S} |E(K)|)$ begrenzt.*

Beweis. Sei R die Menge aller vom Walkup als pertinent markierten Bicomps, die im Laufe des Walkdowns nicht eingebettet werden konnten. Sei T wieder der durch die DFS-Traversierung eindeutig bestimmte, an der Forebear-Bicomp gewurzelte Bicompbaum, dessen Bicomps vor dem Walkdown pertinent waren und vom Walkdown erreicht wurden. Dann wurde jede Bicomp in $R \cap V(T)$ in beide Richtungen der Außenfläche bis zum ersten aktiven Knoten traversiert. Da die Bicomp nicht eingebettet wurde, müssen beide gefundenen Endknoten extern aktiv sein. Ist die Bicomp ein Blatt von T, gilt zusätzlich, dass die Endknoten nicht pertinent sein können, andernfalls würde eine weitere Nachfolgerbicomp vom Walkdown erreicht werden. In jeder Bicomp der Menge $R \cap V(T)$ ist

somit in beide Richtungen der Außenfläche eine ShortCircuit-Kante eingefügt worden, weswegen die so entstandenen Kosten nach Theorem 2.3.6 linear begrenzt sind.

Neben den Traversierungskosten entstehen für diese Bicomps einzig Kosten für die Entscheidung, ob an den beiden erreichten Endknoten in eine Kinderbicomp abgestiegen werden kann. Diese benötigt jeweils nur konstante Zeit pro Bicomp. Allerdings kann die Anzahl der Bicomps $|R \cap V(T)|$ durch mehrfach besuchte NonForebear-Bicomps superlinear sein. Dann können die entstehenden Kosten aber durch die extrahierten Kuratowski-Subdivisions kompensiert werden, da $|R \cap V(T)|$ einen Teil der Kosten des Kuratowski-Kontos im Walkup darstellt. Diese Kosten sind nach Theorem 2.3.3 durch $O(n + m + \sum_{K \in S} |E(K)|)$ begrenzt. □

Bemerkung. Falls die entstehenden Kosten nicht durch die extrahierten Kuratowski-Subdivisions abgedeckt werden können, entsteht genau dann eine superlineare Laufzeit, wenn $|R \cap V(T)| = \omega(1)$ gilt. Exemplarisch wird angenommen, das $\frac{n}{4}$ NonForebear-Bicomps in $R \cap V(T)$ vorhanden sind. Da jede Bicomp nur konstante Zeit benötigt, entstehen in dem Fall die Kosten $\frac{n}{4} * O(1)$ in einer einzelnen Einbettungsphase. Allerdings könnten dieselben NonForebear-Bicomps auch in weiteren Einbettungsphasen nicht eingebettet werden, wenn in diesen immer wieder externe Knoten zu kritischen Knoten werden. Die Kosten $O(\frac{n}{4})$ können also in insgesamt $\Omega(n)$ vielen Einbettungphasen auftreten, was ohne extrahierte Kuratowski-Subdivisions in einer superlinearen Laufzeit resultieren würde.

Es bleibt noch zu zeigen, dass auch das Finden der nächsten, pertinenten Bicomps über alle Knoteneinbettungen lineare Zeit benötigt. Dazu werden die während des erweiterten Walkups gesammelten Informationen genutzt.

Lemma 2.3.8. *Für das Finden der nächsten, pertinenten Bicomps, auf denen der erweiterte Walkdown fortgeführt wird, wird insgesamt Laufzeit $O(n + m + \sum_{K \in S} |E(K)|)$ benötigt.*

Beweis. Bei jedem Abstieg des Walkdowns in eine Bicomp wird ein eindeutiger Identifikationsknoten der Elternbicomp in einem Traversierungsstack gespeichert. Zusätzlich wird der Knoten, an dem abgestiegen wurde und die aktuelle Richtung des Walkdowns an diesem Knoten gespeichert. Somit können diese Informationen während der Suche nach der nächsten Vorgängerbicomp in $O(1)$ gewonnen werden. Wird eine Bicomp auf diese Art vom Stack genommen, wird sie während der gesamten Einbettung des aktuellen Knotens nicht wieder betrachtet. Damit ist die Anzahl der berechneten Vorgängerbicomps durch $|R \cap V(T)|$ begrenzt und damit wiederum nach Theorem 2.3.3 über alle Knoteneinbettungen linear beschränkt.

Für jede auf diese Weise betrachtete Vorgängerbicomp D muss effizient entschieden werden, ob diese pertinent ist, um $N(A)$ zu finden. Sei A die Nachfolgerbicomp von D, an der eine Kuratowski-Extraktion gestartet wurde und die aufgrund dieser nicht mehr pertinent ist. Dann sind die Pertinenzinformationen an D im Allgemeinen nicht mehr konsistent, da D nur aufgrund der in A gelöschten Backedges pertinent gewesen sein

könnte. Ein Ausweg wäre es, den Walkup neu zu starten, um konsistente Pertinenzinformationen zu erhalten, doch dies würde im Allgemeinen keine lineare Laufzeit erlauben.

Die Vorgängerbicomp D ist aber genau dann pertinent, wenn sie entweder selbst aktiv ist oder aktive Nachfolgerbicomps an ihr existieren. Beide Eigenschaften können in konstanter Zeit überprüft werden. Dafür wird die Knotenmenge der Außenfläche von D betrachtet. Seien z_1 und z_2 die Endknoten der beiden ShortCircuit-Kanten an dieser Außenfläche, die existieren müssen, da D nicht eingebettet worden ist und vorher vom Walkdown traversiert wurde. Dabei müssen beide Endknoten extern aktiv sein, allerdings ist $z_1 = z_2$ nicht ausgeschlossen. Die Außenfläche wird nun in die Menge der Knoten, die im Pfad von z_1 über die Wurzel r der Bicomp bis hin zu z_2 enthalten sind und den vom Walkdown nicht erreichten Knoten partitioniert. Beide betrachtete Knotenmengen erlauben eine effiziente Überprüfung auf die Aktivität von D und deren Nachfolgerbicomps.

- Knoten des Pfades $z_1 \rightarrow r \rightarrow z_2$ der Außenfläche:
  Durch LinkToRoot kann r in $O(1)$ gewonnen werden. Beide Endknoten z_1 und z_2 können durch Traversieren der ShortCircuit-Kanten von r ebenfalls in konstanter Zeit berechnet werden. Alle Knoten auf dem restlichen betrachteten Teil der Außenfläche sind nicht aktiv und damit auch nicht pertinent, da r nicht aktiv sein kann und alle anderen Knoten von den ShortCircuit-Kanten überbrückt wurden. Übrig bleiben die Knoten z_1 und z_2, von denen keiner Startknoten einer pertinenten Backedge sein kann, da D sonst vorzeitig eingebettet worden wäre. Damit existiert kein Startknoten einer pertinenten Backedge im gesamten betrachteten Pfad und der Test auf Aktivität von D entfällt.

  Der Test auf aktive Nachfolgerbicomps an den verbleibenden Knoten z_1 und z_2 kann in konstanter Zeit durchgeführt werden, indem überprüft wird, ob einer von ihnen pertinent ist. Da die Anzahl der Tests linear begrenzt ist, ist auch der Gesamtaufwand linear begrenzt.

- Vom Walkdown nicht erreichte Knoten der Außenfläche:
  Der Test auf diesen Knoten braucht nur ausgeführt zu werden, wenn an z_1 und z_2 keine aktiven Nachfolgebicomps existieren, da D sonst zwangsläufig pertinent ist. Damit sind z_1 und z_2 extern aktiv und nicht pertinent. Da damit eine Stoppkonfiguration existiert, endet der Walkdown an den beiden Stoppknoten und traversiert keine Knoten hinter diesen. Würde der Test alle dahinter liegenden Knoten explizit auf Pertinenz prüfen, würde eine quadratische Laufzeit unvermeidbar sein, da dieser Pfad $O(m)$ Kanten enthalten und im Verlauf des Algorithmus $O(m)$ Mal getest werden kann.

  Einen Ausweg stellt die Benutzung der PertinentNodesInWalkup-Liste dar. Diese wurde zwar nach dem Walkup nicht mehr aktualisiert, aber da D nicht eingebettet wurde, können keine pertinente Knoten in D hinzugekommen sein. Damit enthält die Liste eine Obermenge aller pertinenten Knoten in D und es können lediglich Einträge der Liste ungültig geworden sein. Allerdings sind z_1 und z_2 die einzigen Knoten, deren Pertinenz verändert worden sein kann. Somit reicht es aus, die

PertinentNodesInWalkup-Liste solange zu durchlaufen, bis ein Knoten w mit $w \neq z_1$ und $w \neq z_2$ gefunden worden ist, um die Pertinenz der Bicomp nachzuweisen. Wird kein Knoten w gefunden, kann D nicht pertinent sein.

In beiden Fällen werden maximal die ersten drei Einträge der Liste durchsucht, was einem konstanten Aufwand entspricht. Zudem wird die PertinentNodesInWalkup-Liste nur einmal benutzt, nämlich genau dann, wenn z_1 und z_2 keine aktiven Nachfolgerbicomps besitzen und die Bicomp D auf Pertinenz geprüft werden soll. Bei diesem Test wird D entweder als nicht pertinent eingestuft oder eine Kuratowski-Extraktion gestartet. In beiden Fällen existieren nach dem Test keine pertinenten Knoten mehr in D, weswegen die Bicomp und damit auch ihre Wurzelliste PertinentNodesInWalkup in der aktuellen Einbettungsphase nicht mehr betrachtet wird. Dieser Test ist also auch in konstanter Zeit möglich, insgesamt ergibt sich durch die lineare Anzahl der Tests der Gesamtaufwand von $O(n + m + \sum_{K \in S} |E(K)|)$.

Bemerkung. Die PertinentNodesInWalkup-Listen müssen zusätzlich erzeugt und gelöscht werden, was einen größeren Aufwand als die Überprüfung auf Pertinenz erfordert. Allerdings kann die Summe der Elemente aller PertinentNodesInWalkup-Listen nicht beliebig groß werden. Da jeder pertinente Knoten in genau einer PertinentNodesInWalkup-Liste enthalten ist, kann die Summe der Elemente durch die Anzahl der im aktuellen Walkdown involvierten Backedges plus der Anzahl der vom Walkup als pertinent markierten Bicomps abgeschätzt werden. Damit kann der Aufwand nach Abschnitt 2.3.1 ebenfalls durch die Laufzeit des Walkups und damit durch $O(n + m + \sum_{K \in S} |E(K)|)$ begrenzt werden.

□

Theorem 2.3.9. *Der erweiterte Walkdown ist nach Theorem 2.3.5 korrekt und benötigt nach den Lemmata 2.3.7 und 2.3.8 für alle Knoteneinbettungen die Zeit* $O(n + m + \sum_{K \in S} |E(K)|)$.

2.3.3 Extraktion der Kuratowski-Subdivisions

Wurde vom Walkdown eine kritische Stoppkonfiguration gefunden, verursacht jede Backedge mindestens eine Kuratowski-Subdivision, wenn ihr Backedgepfad einen kritischen Knoten traversiert. Damit der erweiterte Walkdown ordnungsgemäß weitergeführt werden kann, muss jede dieser Backedges während der Extraktion gelöscht werden. Dazu werden in einem ersten Schritt alle involvierten Backedges der Stoppkonfiguration effizient bestimmt. Dies wird in Abschnitt 2.3.3.1 beschrieben.

Für jede einzelne Backedge ist es wünschenswert, möglichst viele der Minorentypen zu ermitteln, die von dieser erzeugt werden können. Hierfür wird vorab der *HighestFacePath* benötigt, aus dem hinterher der *HighestXYPath* für jeden kritischen Knoten errechnet wird. Für die Extraktion des *HighestFacePath* kann im Gegensatz zum Originalalgorithmus nicht mehr angenommen werden, dass alle Knoten richtig orientiert sind, da

Bicomps geflippt worden sein können. Deswegen muss eine neue Methode gefunden werden, die auch bei unbekannten Knotenorientierungen die inneren Faces traversieren und daraus den *HighestFacePath* extrahieren kann. Diese Methode wird in Abschnitt 2.3.3.2 vorgestellt.

Im darauf folgenden Abschnitt 2.3.3.3 werden dann auf dieser Basis alle benötigten *HighestXYPaths* extrahiert. Zusätzlich wird die Lage der Endknoten in Bezug zu den Stoppknoten bestimmt, um die auftretenden Minortypen effizient klassifizieren zu können. Mit den berechneten Informationen können dann die Methoden der lokalen Erweiterung benutzt werden, um möglichst viele Kuratowski-Subdivisions aus der betrachteten Stoppkonfiguration zu extrahieren. Für alle Abschnitte sei B die aktuell betrachtete Bicomp mit Wurzel r, an der eine Stoppkonfiguration vorliegt. Weiterhin seien $stopX$ und $stopY$ die beiden Stoppknoten und w ein kritischer Knoten zwischen diesen.

2.3.3.1 Extraktion der kritischen Backedges

Im erweiterten Walkdown wurde bereits festgestellt, dass kein Knoten auf dem Pfad $stopX \rightarrow r \rightarrow stopY$ der Außenfläche von B pertinent sein kann. Also sind alle pertinenten Knoten von B kritisch und verursachen damit mindestens eine Kuratowski-Subdivision. Eine kritische Backedge muss im Allgemeinen nicht direkt an einem kritischen Knoten in B starten. Der Startknoten kann auch in einer Nochfolgerbicomp eines kritischen Knotens enthalten sein. Für die Extraktion der kritischen Backedges werden je nach Bicomptyp von B zwei Fälle unterschieden:

- B ist eine NonForebear-Bicomp:
  Um die Liste der kritischen Backedges zu berechnen, kann die im Walkup erzeugte PertinentNodesInWalkup-Liste an der Wurzel von B benutzt werden. Alle Einträge dieser Liste, bis auf die in Lemma 2.3.8 beschriebenen Ausnahmeknoten $stopX$ und $stopY$, sind pertinent. Mit Hilfe des BackedgeFlags kann jeder Knoten in konstanter Zeit daraufhin überprüft werden, ob er ein Startknoten einer pertinenten Backedge ist. Alle auf diese Weise gefundenen Backedges sind kritisch und werden in einer Liste gespeichert.

  Es verbleiben die Backedges, deren Startknoten in Nachfolgerbicomps kritischer Knoten liegen. Dazu wird jeder Knoten der PertinentNodesInWalkup-Liste bis auf $stopX$ und $stopY$ auf eine nicht-leere *PertinentRoots*-Liste überprüft. Jeder Eintrag der *PertinentRoots*-Liste verweist auf einen eindeutigen Knoten einer anhängenden Bicomp, von dem in $O(1)$ auf dessen Wurzelknoten geschlossen werden kann. An diesem Wurzelknoten existiert mit der PertinentNodesInWalkup-Liste wiederum eine Liste aller pertinenten Knoten, die analog zu B behandelt werden kann. Die Idee ist jetzt, diese Traversierung rekursiv bis zu allen Bicomps, die eine kritische Backedge enthalten, durchzuführen.

  Die Verknüpfung der Wurzelknoten pertinenter Bicomps mit der *PertinentRoots*-Liste bestimmt einen eindeutigen, an B gewurzelten Baum T von Bicomps (siehe Abbildung 2.10). Um alle kritischen Backedges zu erhalten, wird T rekursiv traver-

siert: An jeder Bicomp wird dabei, beginnend mit B, die PertinentNodesInWalkup-Liste deren Wurzel durchgegangen. Für jeden Knoten dieser Liste wird dann zunächst ein etwaige, direkt startende Backedges extrahiert und daraufhin zu allen in den *PertinentRoots*-Listen verlinkten Kinderbicomps abgestiegen.

Dies entspricht einer Preorder-Traversierung von T. Die Menge der pertinenten Knoten in den PertinentNodesInWalkup-Listen sind dabei bis auf *stopX* und *stopY* gültig, da alle Nachfolgerbicomps von B nicht erreicht wurden und somit auch nicht modifiziert worden sind. Weiterhin kann keine PertinentNodesInWalkup-Liste leer sein, da die entsprechende Bicomp sonst nicht pertinent und damit auch nicht in T wäre. Dadurch ist sichergestellt, dass in jeder auf diese Art gefundenen Bicomp oder in deren Nachfolgerbicomps mindestens ein Startknoten einer pertinenten Backedge existiert, die extrahiert werden muss.

Jeder Knoten jeder Bicomp kann in konstanter Zeit daraufhin überprüft werden, ob er pertinent, identisch zu *stopX* bzw. *stopY* oder mit einer pertinenten Backedge verbunden ist. Dadurch kann jedes Kind eines Knotens in T in $O(1)$ gefunden werden. Die Größe von $V(T)$ ist durch die Kosten des Walkups begrenzt, da nur pertinente Bicomps in T enthalten sind und jede dieser Bicomps vom Walkup traversiert wurde. Daraus folgt eine Gesamtlaufzeit von $O(n + m + \sum_{K \in S} |E(K)|)$ für die Menge aller gefundenen Stoppkonfigurationen.

- B ist eine Forebear-Bicomp:
  Da der Walkdown im Allgemeinen mehrere andere Bicomps in B eingebettet haben kann, sind die Pertinenzinformationen der PertinentNodesInWalkup-Liste nicht mehr gültig und die Liste selbst nicht mehr vollständig. Allerdings wurden alle pertinenten Backedges während des Walkups einem eindeutigen Wurzelknoten v' zugeordnet, nämlich der Wurzel der späteren Forebear-Bicomp, an der der verlängerte Backedgepfad der Backedge geendet hat. Da im aktuellen Walkdown nur eine Forebear-Bicomp existiert, ist v' eindeutig bestimmt und es können alle pertinenten Backedges des aktuellen Walkdowns über die Liste $BackedgesOnVirtualNode(v')$ bestimmt werden. An einer Forebear-Bicomp wird höchstens einmal pro Walkdowns eine kritische Stoppkonfiguration gefunden. Da die *BackedgesOnVirtualNode*-Listen zusätzlich disjunkt sind, summieren sich die Gesamtkosten für das Auslesen der Listen zu höchstens m auf.

  Es bleibt zu zeigen, dass jede dieser m Backedges in konstanter Zeit darauf getestet werden kann, ob sie kritisch ist. Für jede Backedge einer Liste können drei Fälle zutreffen:

  1. Die Backedge wurde aufgrund einer Kuratowski-Extraktion gelöscht.
  2. Die Backedge wurde eingebettet.
  3. Die Backedge wurde weder gelöscht noch eingebettet.

  Mit der Markierung jeder eingebettete Backedge können die beiden ersten Eigenschaften, und damit auch die dritte, in jeweils $O(1)$ getestet werden. Da die verursachende Stoppkonfiguration an der Forebear-Bicomp existiert, muss der CCW-

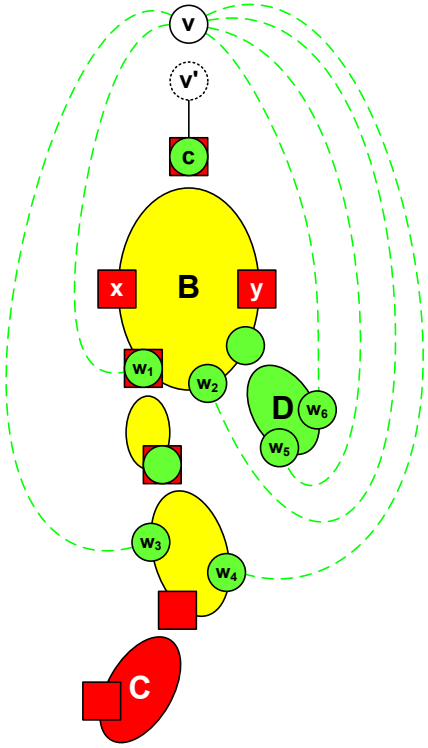


Abbildung 2.10: Beispiel der Extraktion kritischer Backedges. Alle an w_1 bis w_6 startenden, pertinenten Backedges sind aufgrund der Stoppknoten x und y kritisch. Es wird eine Preorder-Traversierung auf dem Bicompbaum T gestartet, der hier alle Bicomps außer der Forebear-Bicomp und der externen Bicomp C enthält. Mit Hilfe der BackedgeFlags werden die an den Knoten w_1 und w_2 direkt in B startenden Backedges extrahiert. Dann wird zur ersten Nachfolgerbicomp an w_1 abgestiegen, an der lediglich eine weitere Bicomp hängt. An dieser werden die Backedges an w_3 und w_4 gefunden und extrahiert. Da an einem Blatt von T angelangt worden ist, können keine weiteren pertinenten Nachfolger mehr existieren. Die Preorder-Traversierung durchläuft T deswegen bis zur Bicomp D und extrahiert die verbleibenden Backedges an w_5 und w_6.

Walkdown vorher an einem Stoppknoten *stopX* und der CW-Walkdown an einem Stoppknoten *stopY* abgebrochen worden sein. Deswegen sind genau die Backedges, die der Walkdown in beide Richtungen nicht erreichen und damit nicht einbetten konnte, entweder gelöscht oder erzeugen kritische Knoten zwischen den Stoppknoten. Folglich stellen die Backedges, die die dritte Eigenschaft aufweisen, genau die Menge der gesuchten kritischen Backedges dar. Insgesamt erfordert die Extraktion die Laufzeit $O(n+m)$, da zusätzlich an jeder der $O(n)$ Forebear-Bicomps überprüft werden muss, ob kritische Backedges vorhanden sind.

Insgesamt wurde damit folgende Aussage gezeigt:

Proposition 2.3.10. *Die Laufzeit für die Extraktion der kritischen Backedges aller Stoppkonfigurationen ist durch* $O(n + m + \sum_{K \in S} |E(K)|)$ *beschränkt.*

Da maximal m Backedges existieren können und in jeder Stoppkonfiguration mindestens eine kritische Backedge gelöscht wird, gilt weiterhin:

Korollar 2.3.11. Die Gesamtanzahl der im Algorithmus auftretenden Stoppkonfigurationen ist maximal m. In jeder dieser Stoppkonfigurationen benötigt die Extraktion der kritischen Backedges eine amortisierte Laufzeit von

$$O(1 + \frac{n}{m} + \frac{1}{m} \sum_{K \in S} |E(K)|).$$

Damit können alle kritische Backedges, die in Punkt 2 auf Seite 36 benötigt werden, effizient extrahiert werden. Auch die dort benötigten Backedgepfade und kritischen Knoten können, ähnlich zum Walkup, durch eine Traversierung auf den Bicomp-Außenflächen effizient berechnet werden.

2.3.3.2 Extraktion des *HighestFacePath* ohne Flipping

Sind alle kritischen Backedges einer Stoppkonfiguration gefunden, muss für jede untersucht werden, welche Typen von Kuratowski-Subdivisions durch sie erzeugt werden. Eine wichtige zu testende Eigenschaft im Originalalgorithmus war dabei die Existenz und Lage von XYPaths(w) in B. Es war möglich, sich auf die Berechnung des *HighestXYPath*(w) zu beschränken, der Bestandteil der inneren Face f von B ist, die durch Löschen aller Kanten an r bis auf die Kanten der Außenfläche entsteht.

Allerdings werden im Originalalgorithmus Kuratowski-Subdivisions nur an einem einzelnen kritischen Knoten der Stoppkonfiguration extrahiert. Da der *HighestXYPath*(w) abhängig vom gewählten kritischen Knoten ist, musste dort nur ein einziger *HighestXYPath* extrahiert werden. Die Aufgabe hier ist aber, Kuratowski-Subdivisions für jeden existierenden kritischen Knoten zu extrahieren, so dass mehrere *HighestXYPaths* berechnet werden müssen. Nach Korollar 2.1.7 sind alle *HighestXYPaths* einer Bicomp B im *HighestFacePath*(B) enthalten. Die Extraktion des *HighestFacePath*(B) gestaltet sich aber aufgrund von möglicherweise vorhandenen, geflippten Bicomps schwierig.

Angenommen, während des Entstehungsprozesses von B wären keine Bicomps geflippt worden: Dann hätte jeder Knoten in B die ursprüngliche, während der Einbettung in B verwendete Orientierung. In diesem Fall kann der *HighestFacePath*(B) extrahiert werden, indem vorab die Face f' durch das Löschen der inneren Kanten an r gebildet wird. Der begrenzende Weg von f' wird nun einmal durchlaufen. An jedem dabei mehrfach besuchten Knoten müssen anhängende Bicomps durch das Löschen der Kanten an r separiert worden sein. Diese Bicomps können nicht Bestandteil von f sein. Deswegen werden alle besuchten Knoten in einem Stack zwischengespeichert. Bei wiederholtem Erreichen eines Knotens wird der zwischen den beiden Besuchen traversierte vom Stack gelöscht. Der letztendlich im Stack enthaltene Kreis stellt somit f dar, woraus der *HighestFacePath*(B) durch Löschen der verbleibenden Kanten an r direkt errechnet werden kann.

Falls andererseits Bicomps in B geflippt worden sind, kann f' nicht auf diese Weise bestimmt werden. Die Traversierung einer Face benötigt konsistente Adjazenzlisten und genau diese sind hier nicht gegeben, da während des Flippens von Bicomps die Orientierungen der involvierten Knoten nicht aktualisiert wurden. Die Adjazenzlisten solcher Knoten sind damit invertiert. Eine erste Möglichkeit, dies zu verhindern, wäre, an jedem Knoten die aktuelle Orientierung festzuhalten, anstatt lediglich an der Wurzel des geflippten Teilbaums. Dann wäre eine Flip-Operation aber nicht mehr in konstanter Zeit möglich und es würde eine quadratische Gesamtlaufzeit entstehen. Eine weitere Möglichkeit wäre ein Preprocessingschritt vor der Extraktion des *HighestFacePath*(B), der alle Knotenorientierungen geflippter Bicomps in B aktualisiert, allerdings ist auch dieser nicht in effizienter Zeit möglich. Der Grund dafür ist, dass alle als geflippt markierten Wurzelknoten des in B liegenden DFS-Baums traversiert werden müssten. Dabei kann B in $\Omega(n)$ vielen Preprocessingschritten vorkommen und B gleichzeitig $\Omega(n)$ viele dieser Knoten enthalten, was ebenfalls in einer quadratischen Gesamtlaufzeit resultieren würde.

An jedem neu erreichten Knoten einer möglichen Traversierung von f' muss wegen der unbekannten Knotenorientierung eine Auswahl zwischen den beiden Nachbarn der Eingangskante in der zyklischen Adjazenzliste des Knotens getroffen werden, um den gewünschten Nachfolgerknoten zu erhalten. Diese Auswahlmöglichkeiten sind hinreichend, da die Reihenfolge der Adjazenzlisteneinträge bis auf Inversion nicht verändert wird und deswegen für jeden Knoten nur die beiden Nachbareinträge in Frage kommen. Als Entscheidungshilfe für die Auswahl kann die Eigenschaft des Walkdowns ausgenutzt werden, dass die beiden ExternalLinks eines Knotens nicht gelöscht werden, wenn dieser durch weitere Einbettungen von der Außenfläche verschwindet. Folglich befinden sich an jedem inneren Knoten in B noch immer die ExternalLinks zu den beiden Kanten, die zuletzt die Außenfläche dargestellt haben.

Die Idee ist jetzt, die Traversierung von f' anhand dieser restlichen Link-Informationen vorzunehmen, anstatt dafür die Adjazenzlisten zu benutzen. Das hat den Vorteil, dass die Traversierung unabhängig von den Knotenorientierungen ist. Allerdings sind die ExternalLinks während der Einbettung von B an vielen Knoten überschrieben worden und damit hochgradig unvollständig. Um die Traversierung starten zu können, werden daher weitere Strukturinformationen benötigt. An jedem Knoten innerhalb von B existierten

vor der Einbettung die beiden Außenflächen-Links *OldLinks* der ehemaligen Bicomps. Durch eine Kombination der alten und neuen Links an jedem Knoten kann die Auswahl zwischen den beiden möglichen, benachbarten Adjazenzlisteneinträgen getroffen werden, und damit f' unabhängig von den Knotenorientierungen traversiert werden.

Allerdings muss dafür eine spezielle Traversierung von f' gewählt werden, dessen Struktur im folgenden beschrieben wird: Ziel ist es, die Traversierung von f' so in zwei Teile aufzuspalten, dass ein Teil den besuchten Knoten des CCW-Walkdowns und der andere Teil den besuchten Knoten des CW-Walkdowns entspricht. Die Bicomp B kann nicht entartet sein und besteht deswegen aus den vom Walkup markierten, ehemaligen Bicomps, die vom Walkdown auch eingebettet werden konnten. Nach Löschen der inneren Kanten aus B, die inzident zur Wurzel r sind, lassen sich diese nach Korollar 2.1.5 in die Bicomps der Bottom-Chain und in die Menge der vollständig eingebetteten, durch die Löschung separierten Bicomps, aufteilen.

Sei c der Knoten aus B mit dem niedrigsten DFI-Wert und seien a und b die beiden zu r inzidenten Kanten auf der Außenfläche. Dann ist c Wurzel des DFS-Teilbaums, den die Menge der ehemaligen, jetzt eingebetteten Bicomps enthält. Dadurch können von c aus alle ehemaligen Bicomps aus B erreicht werden. Analog zum Walkdown wird die Traversierung deswegen an c gestartet (siehe Abbildung 2.11). Zusätzlich wird überprüft, ob die eindeutige Kante $\{r, c\}$ der Kante a oder b entspricht. Ist das der Fall, ist c Endknoten auf der Bottom-Chain und es existiert von c aus nur eine mögliche Richtung auf dieser, die an f' grenzt. Diese Richtung kann in konstanter Zeit berechnet werden, indem der Nachbar der Eingangskante in der Adjazenzliste gewählt wird, der keinen Link zur Außenfläche darstellt. Andernfalls existieren von c aus zwei an f' grenzende Richtungen und es wird nacheinander für jede der beiden Nachbarkanten von $v \rightarrow c$ eine Traversierung gestartet. In diesem Fall kann hinterher aus beiden Traversierungen der *HighestFacePath* berechnet werden.

Ausgehend von c wird irgendwann ein erster Knoten z der Bottom-Chain erreicht. Dabei ist z in den Traversierungen beider Richtungen derselbe Knoten, da von c aus alle ehemaligen Bicomps erreicht werden können und der Bicomppfad von c nach z eindeutig ist. Ausgehend von z wird nun der an f' grenzende Teil der Bottom-Chain in der jeweils eingeschlagenden Richtung traversiert, bis entweder an der Kante a oder b gestoppt wird. Der *HighestFacePath*(B) kann dann aus den beiden traversierten Pfaden berechnet werden, indem wieder jede separierte Zweizusammenhangskomponente durch die Erkennung mehrfach besuchter Knoten unter Verwendung eines Stacks vermieden wird. Insbesondere werden dadurch die beiden unterschiedlichen Teilpfade von c nach z nicht in den *HighestFacePath*(B) aufgenommen.

Es bleibt zu zeigen, wie zu jedem Knoten dieser speziellen Traversierungen der korrekte Nachfolgerknoten der Face f' gefunden werden kann. Das Problem konnte auf die Wahl des richtigen Nachbareintrags zur Eingangskante in der Adjazenzliste zurückgeführt werden. Während des Walkdowns werden bei jedem Verschmelzen zweier Bicomps die ExternalLinks des dabei entfernten virtuellen Knotens in dem Array OldLinks an dem realen Knoten gespeichert. Falls OldLinks schon Links enthält, werden diese überschrieben. Somit zeigen die OldLinks eines Knotens in B immer auf die Außenfläche der

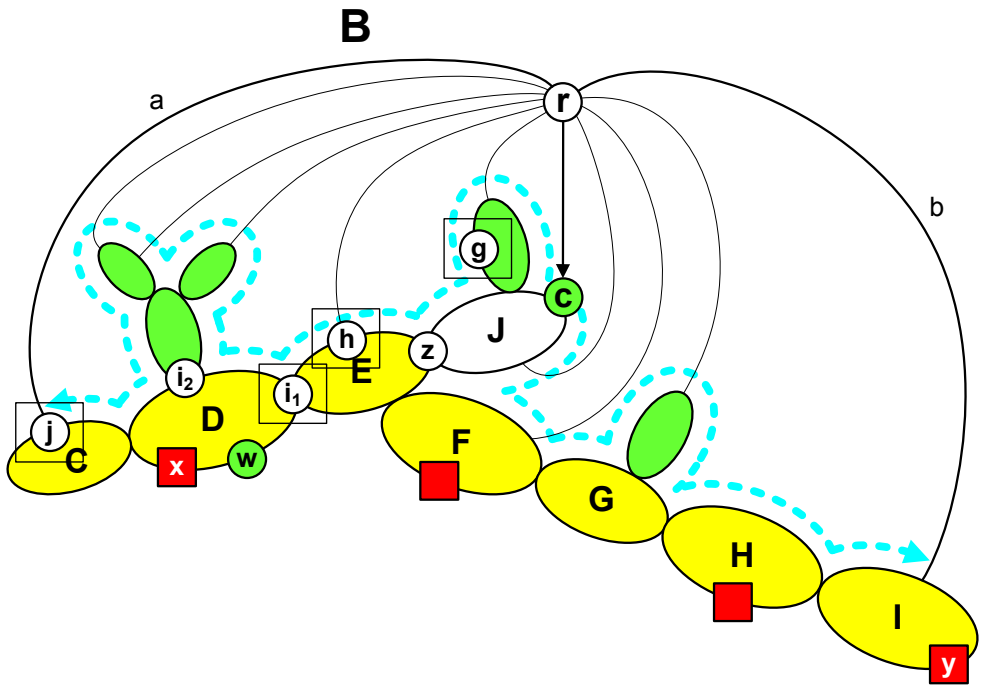


Abbildung 2.11: Extraktion des *HighestFacePath* einer Bicomp B, an der die Bottom-Chain durch alle gelben Bicomps dargestellt ist. An dem eindeutigen Knoten c mit niedrigstem DFI in G wird mit Hilfe der ExternalLinks festgestellt, dass c vollständig innerhalb B eingebettet worden ist und deswegen zwei Richtungen auf der Bottom-Chain zu traversieren sind. Diese beiden Traversierungspfade sind durch die beiden dick gestrichelten, am Knoten c startenden Pfeile gekennzeichnet. Beide erreichen denselben ersten Knoten z auf der Bottom-Chain und enden dann jeweils an den Kanten a und b. Alle Bicomps, die nicht in der Bottom-Chain enthalten sind, werden in einem zweiten Schritt von den beiden Traversierungspfaden separiert. Während der Traversierungen wird an jedem Knoten die Nachfolgerkante durch die Informationen der ExternalLinks und der OldLinks bestimmt. Unterschiedliche Knotentypen erfordern dabei unterschiedliche Berechnungen. Die Knoten g–j stellen dabei jeweils Beispiele für die Knotentypen 1–4 dar und werden in Abbildung 2.12 detailliert dargestellt.

ehemaligen Bicomp, die an diesem Knoten zuletzt vom Walkdown eingebettet wurde.

Da die beiden möglichen Traversierungen symmetrisch sind, wird o. B. d. A. die an der Kante a endende Traversierung betrachtet. Jeder dabei besuchte Knoten e kann in vier Typen aufgeteilt werden, die jeweils folgende Eigenschaften aufweisen:

- Typ 1: An e wurde weder eine Backedge, noch eine Kinderbicomp eingebettet.
- Typ 2: An e wurde zuletzt eine Backedge eingebettet, die jetzt innerhalb von B liegt.
- Typ 3: e wurde zuletzt mit einer Kinderbicomp verschmolzen.
- Typ 4: e ist Startknoten der äußeren Backedge a.

Die Knotentypen sind disjunkt und deren Klassifikation vollständig, da der Walkdown höchstens Bicomps und Backedges einbetten kann. Für jeden Knotentyp wird jetzt durch die Kombination der ExternalLinks und OldLinks in konstanter Zeit der richtige Nachbareintrag in der Adjazenzliste von e gewählt:

Falls e vom Typ 1 ist, wurde vom Walkdown an e weder eine Backedge, noch eine Kinderbicomp eingebettet. Die ExternalLinks an e können deswegen nicht überschrieben worden sein. Daher verweisen die ExternalLinks genau auf die beiden, zu e inzidenten Kanten an der Außenfläche der eingebetteten, ehemaligen Bicomp. Es reicht also aus, den ExternalLink zu traversieren, der nicht der Kante entspricht, über die e erreicht worden ist (siehe Abbildung 2.12(a)).

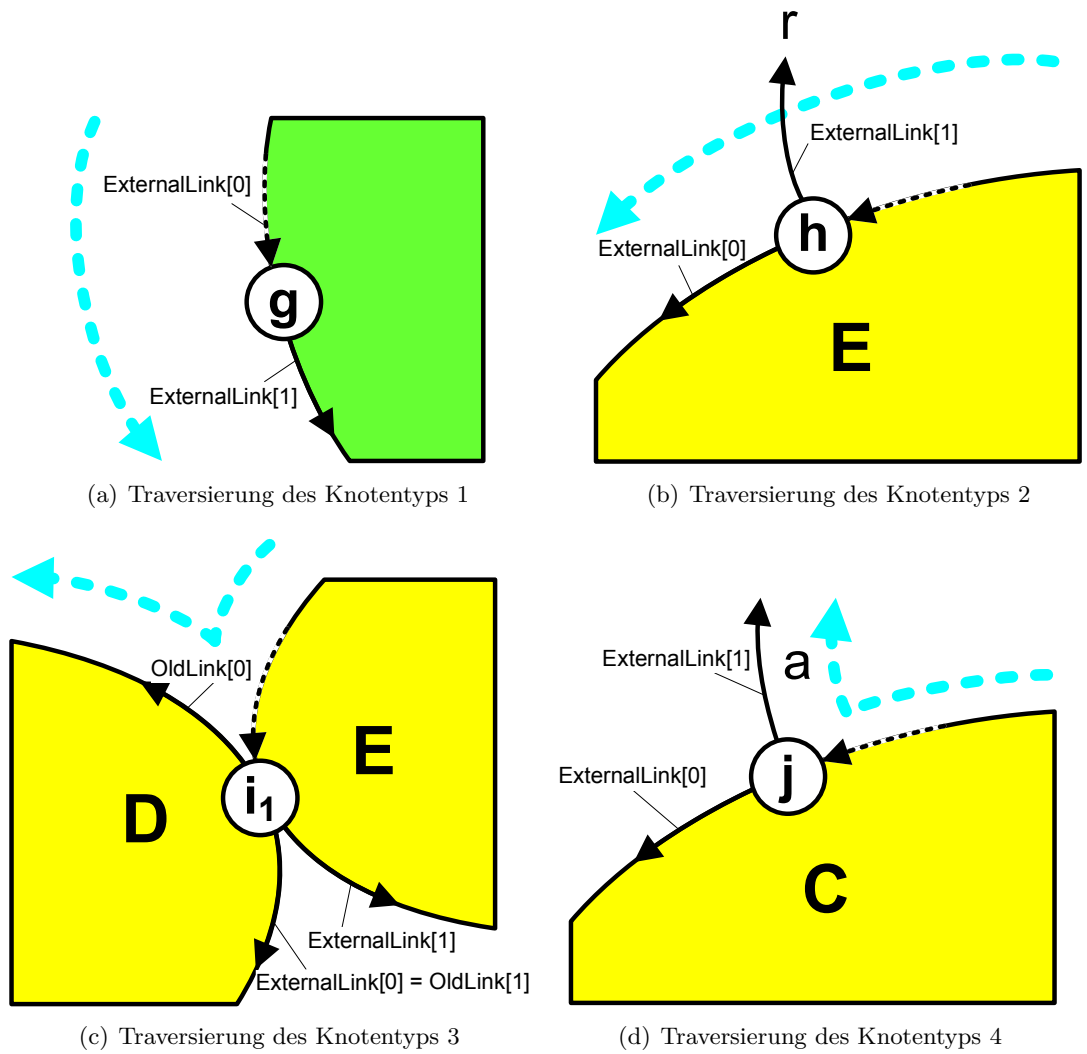
Ist e vom Typ 2, wurde vom Walkdown zuletzt eine pertinente Backedge eingebettet. Dann zeigen die ExternalLinks an e einerseits auf die eingebettete Backedge und andererseits auf die nächste Kante in f'. Also muss hier immer der ExternalLink traversiert werden, der auf die Kante zeigt, die nicht inzident zu r ist (siehe Abbildung 2.12(b)).

Falls e vom Typ 3 ist, wurde vom Walkdown zuletzt eine Kinderbicomp eingebettet, weswegen an e zwei OldLinks gespeichert worden sind. Diese verweisen auf die ehemaligen Außenflächenkanten der letzten an e eingebetteten Kinderbicomp. Beim Verschmelzen zweier Bicomps bleibt immer jeweils eine Seite beider Bicomps auf der Außenfläche. Die aktuelleren Link-Informationen der ExternalLinks verweisen genau auf die Kanten dieser Seiten.

Deswegen muss eine der beiden OldLinks mit dem ExternalLink an der Kinderbicomp übereinstimmen. Der verbleibende OldLink muss sich an einer inneren Face von B befinden und ist Bestandteil der Kinderbicomp, weswegen er genau dem gesuchten Nachbareintrag der Adjazenzliste entspricht und daher traversiert wird (siehe Abbildung 2.12(c)). Für den Fall, dass an e mehrere Kinderbicomps eingebettet worden sind, wird nur die zuletzt eingebettete traversiert und alle anderen bleiben unberührt. Da die davor eingebetteten Kinderbicomps aber vom Walkdown vollständig innerhalb B eingebettet worden sein müssen, stellen sie separable Zweizusammenhangskomponenten dar, die nicht Bestandteil des $\mathit{HighestFacePath}(B)$ sein können.

Ist e vom Typ 4, wird die Kante, die zu r inzident ist, gewählt. Diese ist eindeutig und entspricht der Kante a (siehe Abbildung 2.12(d)). Damit ist die Traversierung beendet.

Bemerkung. Beachtet werden muss, dass die Bicomps im Allgemeinen entartet sein kön-



(a) Traversierung des Knotentyps 1

(b) Traversierung des Knotentyps 2

(c) Traversierung des Knotentyps 3

(d) Traversierung des Knotentyps 4

Abbildung 2.12: Die vier verschiedenen Knotentypen während der Traversierung der Bicomp B aus Abbildung 2.11 im Detail. Alle vorhandenen Link-Informationen der ExternalLinks und OldLinks sind eingezeichnet. Der jeweils zu dem Knoten inzidente, dünn gestrichelte Pfeil kennzeichnet die Kante, über die der Knoten erreicht worden ist. Die jeweils dick gestrichelte Kante neben den Bicomps zeigt in die Richtung, in die letztendlich traversiert werden muss.

nen. Dann fallen beide OldLinks zu einer Kante zusammen und können deswegen nicht mehr unterschieden werden. Allerdings stellt das für die korrekte Traversierung von f' kein Problem dar, denn unabhängig von dem gewählten OldLink in den einzelnen Knotentypen wird dann derselbe Endknoten über dieselbe verlinkte Kante erreicht.

Laufzeit

Theorem 2.3.12. *Die im Verlauf des Algorithmus benötigten* HighestFacePaths *können in einer Gesamtlaufzeit von $O(n + m)$ extrahiert und verwaltet werden.*

Beweis. Für die Extraktion eines *HighestFacePaths* müssen Zusatzinformationen im Array OldLinks gespeichert werden. Diese Zusatzinformationen werden während des Walkdowns gespeichert und bewirken nur einen konstanten Mehraufwand pro Knoten, weswegen die asymptotische Laufzeit des Walkdowns erhalten bleibt. Falls während der Extraktion der Typ eines traversierten Knotens bekannt ist, kann dessen Anschlusskante mit einer konstanten Anzahl von Vergleichen der ExternalLinks und OldLinks berechnet werden, da diese jeweils genau zwei Einträge enthalten. Allerdings muss der Knotentyp dazu erst selbst bestimmt werden.

Die Knotentypen 2 und 4 können klassifiziert werden, indem jeder der beiden ExternalLinks auf eine zu r inzidente Kante getestet wird. Entspricht diese Kante dabei a oder b, wurde der Knotentyp 4 gefunden, andernfalls der Knotentyp 2. Falls keine zu r inzidente Kante existiert, werden beide OldLinks, sofern diese existieren, daraufhin überprüft, ob sie mit einem der ExternalLinks identisch sind. Ist das der Fall, liegt an dem betrachteten Knoten der Typ 3 vor. Andernfalls muss der Knoten vom Typ 1 sein. Für die Bestimmung der Knotentypen werden somit insgesamt höchstens $2 + 2 + 4 = 8$ Vergleiche ausgeführt, was einem konstanten Aufwand pro Knoten entspricht.

Entscheidend für die Laufzeit der Extraktionen ist damit die Länge der extrahierten Pfade. Allerdings ist es möglich, dass $\Omega(nm)$ viele Kanten in *HighestFacePaths* extrahiert werden. Das ist beispielsweise der Fall, wenn eine NonForebear-Bicomp, die einen *HighestFacePath* der Länge $\Omega(m)$ enthält, während $\Omega(n)$ verschiedener Knoteneinbettungen neue Kuratowski-Subdivisions erzeugt und dabei nie in eine andere Bicomp eingebettet wird. Die wiederholte Berechnung dieses *HighestFacePaths* würde damit die lineare Gesamtlaufzeit sprengen. Die Anzahl der Kanten in unterschiedlichen *HighestFacePaths* ist jedoch weitaus geringer. Aber auch bei Betrachtung ausschließlich verschiedener *HighestFacePaths* können mehr als m Kanten involviert sein, da manche Kanten Bestandteil mehrerer *HighestFacePaths* sein können. Daher muss die Gesamtanzahl extrahierter Kanten detaillierter analysiert werden:

Es genügt zu zeigen, dass die Gesamtlänge aller insgesamt benötigten *HighestFacePaths* durch $2m$, also linear, begrenzt ist. Sei B eine Bicomp, deren *HighestFacePath* extrahiert wurde. Sei zusätzlich A die erste Bicomp im späteren Verlauf des Algorithmus, die B enthält und an der ein weiterer *HighestFacePath* extrahiert werden muss. $A = B$ ist dabei nicht ausgeschlossen. Es werden zwei Fälle unterschieden:

1. Fall $A \neq B$:

Hier muss die Bicomp B vom Walkdown in A eingebettet worden sein, da sie sonst nicht Bestandteil von A wäre. O. B. d. A. sei im *HighestFacePath*(A) eine Teilmenge der Kanten von B enthalten, andernfalls ist die Gesamtlänge der *HighestFacePaths* leicht durch m abschätzbar. Nach Korollar 2.1.5 ist B damit eine Bicomp der Bottom-Chain von A. Der *HighestFacePath*(A) traversiert damit höchstens die Kanten an der Außenfläche von B. Bis auf drei Ausnahmen befinden sich alle Kanten des *HighestFacePath*(B) innerhalb von B und können deswegen nicht in weiteren *HighestFacePaths* vorkommen. Die Ausnahmen stellen einerseits die beiden Kanten a und b auf der Außenfläche von B, und andererseits die Kanten entarteter, ehemaliger Bicomps von B dar. Die entarteten Bicomps enthalten dabei jeweils dieselbe Kante gleichzeitig auf der Außen- und Innenfläche von B.

Ziel ist es, zu beweisen, dass die Kanten dieser drei Ausnahmen nur in endlich vielen weiteren extrahierten *HighestFacePaths* vorkommen können. In A treten diese Kanten in zwei unterschiedlichen Mengen auf: In der Menge X, dessen Kanten auf dem *HighestFacePath*(A) liegen und in der Menge Y der verbleibenden Kanten auf der Außenfläche von A. Die Bicomp B kann keine entartete Bicomp von A sein, da zu einem vorigen Zeitpunkt eine Stoppkonfiguration an ihr aufgetreten ist. Damit befindet sich jede Kante aus X vollständig innerhalb von A und kann nicht wieder auf einen *HighestFacePath* zukünftiger Bicomps gelangen.

Andererseits gilt diese Aussage für die Kanten aus Y nicht, denn diese könnten durch eine Einbettung in eine andere Bicomp in zukünftigen *HighestFacePaths* enthalten sein. Bis zu dieser Einbettung bleibt aber jede Kante aus A entweder an der Außenfläche oder ist innerhalb von Bicomps eingebettet worden, an denen kein *HighestFacePath* extrahiert wird. Also ist diese Kante bis dahin auch nicht in einem *HighestFacePath* enthalten. Nach der Extraktion lässt sich dieselbe Argumentation wie auf die Menge X anwenden. Es folgt, dass jede Kante eines gefundenen *HighestFacePaths* in maximal einem weiteren *HighestFacePath* enthalten sein kann. Da es maximal m Kanten gibt, ist damit die Gesamtlänge aller Kanten in *HighestFacePaths* maximal $2m$.

2. Fall $A = B$:
   In diesem Fall wurde B seit der letzten Extraktion nicht erweitert und muss deswegen eine NonForebear-Bicomp darstellen. Der enthaltene *HighestFacePath*(B) hat sich damit nicht verändert. Um denselben *HighestFacePath* nicht wiederholt extrahieren zu müssen, wird jeder Knoten eines einmal gefundenen *HighestFacePaths* markiert und zusätzlich jeweils dessen Nachfolgerkante im Pfad abgespeichert. Es kann dann für jede Bicomp in konstanter Zeit überprüft werden, ob an dieser bereits ein *HighestFacePath* extrahiert wurde, indem der Wurzelknoten auf eine solche Markierung getestet wird. Wird der *HighestFacePath* einer Bicomp ungültig, indem die Bicomp in eine andere eingebettet wird, existiert auch der ehemalige, virtuelle Wurzelknoten nicht mehr. Die Markierung bedarf also keiner weiteren Verwaltung, falls *HighestFacePaths* ungültig werden. An A wird also in konstanter Zeit der ehemals in B extrahierte *HighestFacePath* gefunden.

Insgesamt ist die Länge aller *HighestFacePaths* also durch $2m$ beschränkt und jeder Knoten kann in konstanter Zeit behandelt werden. Da die Markierungen aller Knoten zuätzlich einen Speicherplatz von $O(n)$ benötigen, ergibt sich die Gesamtlaufzeit von $O(n+m)$. □

2.3.3.3 Extraktion der *HighestXYPaths* und deren Lage

Nach Korollar 2.1.7 und 2.1.8 können alle benötigten *HighestXYPaths* aus dem zugehörigen *HighestFacePath* einer Bicomp extrahiert werden. Dieser liegt dabei als doppelt verkettete Liste von Knoten und Kanten vom linken Endknoten bis zum rechten Endknoten vor. Jeder Mergeknoten der Bottom-Chain sei mit dem entsprechenden Knoten auf dem bereits extrahierten *HighestFacePath* verlinkt. Ist das nicht der Fall, kann diese Verlinkung leicht durch ein Postprocessing auf dem *HighestFacePath* hergestellt werden.

Zu einem gegebenen kritischen Knoten w steht vorerst nicht fest, ob an diesem überhaupt ein *HighestXYPath*(w) existiert. Deswegen muss vor der Extraktion dessen Existenz sichergestellt werden. Dafür wird jeder Knoten eines extrahierten *HighestFacePath*(B) einer Bicomp B vorab als besucht markiert. Als Marker dient dabei der eindeutige Knoten c mit niedrigstem DFI-Wert aus B, damit später keine *HighestFacePaths* benutzt werden, die aufgrund einer Einbettung ungültig geworden sind. Falls eine solche Markierung an w existiert und diese gültig ist, dann liegt w auf dem *HighestFacePath*(B) und es kann kein *HighestXYPath*(w) existieren. Andernfalls wurde w nicht vom *HighestFacePath*(B) erreicht und der *HighestXYPath*(w) muss existieren.

In letzterem Fall muss dieser berechnet werden: Sind dessen Endknoten p_x und p_y bekannt, können die zwischen p_x und p_y befindlichen Knoten auf dem doppelt verlinkten *HighestFacePath*(B) traversiert werden. Damit erhalten wir den gewünschten *HighestXYPath*(w). Im Allgemeinen wird aber nur einer der beiden Endknoten bekannt sein. Dann muss die Richtung bekannt sein, in der der *HighestFacePath*(B) ab diesem Endknoten traversiert werden soll. Diese Aufgabe wird dadurch erschwert, dass die Knotenorientierungen durch Flippen von Bicomps nach wie vor nicht konsistent sein müssen. Insgesondere ist unbekannt, ob beispielsweise eine an w startende CCW-Traversierung wirklich gegen den Uhrzeigersinn verläuft. Ziel ist es insgesamt, zu einem gegebenen *HighestFacePath* und einem gegebenen kritischen Knoten w einen der beiden Endknoten des *HighestXYPath*(w) und die zu traversierende Richtung im *HighestFacePath* ab diesem Knoten zu berechnen.

Dazu wird vorab die Extraktion des *HighestFacePath* so erweitert, dass alle Mergeknoten fortlaufend nummeriert werden. Die Extraktion wurde im letzten Abschnitt in zwei Teile, die CCW-Extraktion und die CW-Extraktion, aufgespalten. Beide Teiltraversierungen starten dabei an dem Knoten c, dem der Wert 0 zugewiesen wird. Die CCW-Extraktion wird jetzt dahingehend erweitert, dass jeder erreichte Mergeknoten mit einem um 1 höheren Wert markiert wird (siehe Abbildung 2.13). Zusätzlich wird der erreichte Endknoten des *HighestFacePath* auf diese Weise markiert, falls dieser nicht bereits ein Mergeknoten ist. Im Gegensatz dazu werden die Mergeknoten der CW-Extraktion immer mit einem

um 1 niedrigeren Wert, startend bei 0, markiert.

Beachtet werden muss, dass dabei nicht nur die Mergeknoten zwischen zwei Bicomps der Bottom-Chain markiert werden, sondern auch Mergeknoten zu inneren Bicomps. Dadurch kann es vorkommen, dass erreichte Mergeknoten bereits markiert worden sind. Diese Knoten müssen dann innere Mergeknoten sein und werden deswegen speziell gekennzeichnet. Nach erfolgter Extraktion des *HighestFacePath* wird dann die Markierung an allen gekennzeichneten Knoten gelöscht, da sie für den *HighestXYPath*(w) nicht interessant sind. Jeder Mergeknoten der CCW-Extraktion ist positiv und jeder Mergeknoten der CW-Extraktion negativ.

Steht die Nummerierung aller Mergeknoten fest, kann der *HighestXYPath*(w) berechnet werden. Sei D die ehemalige, jetzt eingebettete Bicomp, die den kritischen Knoten w enthält. Wir starten jetzt ab w parallele Traversierungen in beide Richtungen auf der Außenfläche von B. Wenn ein erster nummerierter Knoten i gefunden worden ist, werden beide Traversierungen beendet. Die Voraussetzung für die Terminierung dieser Vorgehensweise ist aber, dass an keinem Knoten des besuchten Pfades eine ShortCircuit-Kante von r aus endet. Wäre das der Fall, könnte die Außenfläche von D mittels einer solchen ShortCircuit-Kante verlassen werden. Dadurch wird der Knoten r erreicht und die Traversierung würde möglicherweise endlos die Außenfläche von B umrunden.

Um die Traversierung auf der Außenfläche von D zu halten, wird deshalb jede besuchte Kante daraufhin überprüft, ob sie eine ShortCircuit-Kante zu r darstellt. Sei o der momentan besuchte Knoten und sei p die nächste zu besuchende Kante der Außenfläche (siehe Abbildung 2.14). Falls p eine solche ShortCircuit-Kante ist, traversieren wir diejenige Nachbarkante von p aus der Adjazenzliste, über die wir nicht nach o gekommen sind. Unabhängig von der Knotenorientierung an o wird damit immer die nächste Kante auf der Außenfläche von D gewählt, da an o weder eine pertinente Backedge noch eine Kinderbicomp eingebettet worden sein kann. Die ExternalLinks der so erreichten Knoten sind bis auf die ShortCircuit-Kanten noch immer aktuell, so dass die Außenfläche von D weiter traversiert werden kann.

Da w auf der Außenfläche von B liegt, muss i entweder ein Mergeknoten auf der Bottom-Chain oder einer der beiden Endknoten des *HighestFacePath*(B) sein. Zudem liegt i noch auf der Außenfläche von D, da zum Verlassen von D eine Überschreitung der Mergeknoten notwendig wäre. Da der *HighestXYPath*(w) existiert und dieser nach Theorem 2.1.4 dem eingebetteten Teil der Bicomp D entspricht, muss damit i einer der Endknoten p_x oder p_y sein. Es bleibt herauszufinden, welchen dieser beiden Knoten i darstellt und in welche Richtung der *HighestXYPath*(w) auf dem *HighestFacePath*(B) ab i verläuft.

Wir betrachten dazu die noch immer gültigen Informationen der OldLinks. Diese können an jedem Mergeknoten auf der Außenfläche von D existieren. Allerdings verweisen lediglich die OldLinks an der ehemaligen, virtuellen Wurzel auf die Kanten von D selbst. In allen anderen Mergeknoten wurden Kinderbicomps eingebettet, weswegen die dortigen OldLinks auf die Außenkanten einer anderen Bicomp verweisen. Da die ehemalige Wurzel während des Einbettungsprozesses zu einem Endknoten des *HighestXYPath*(w) verschmolzen wurde, existieren damit entweder an p_x oder an p_y OldLinks, die auf Kan-

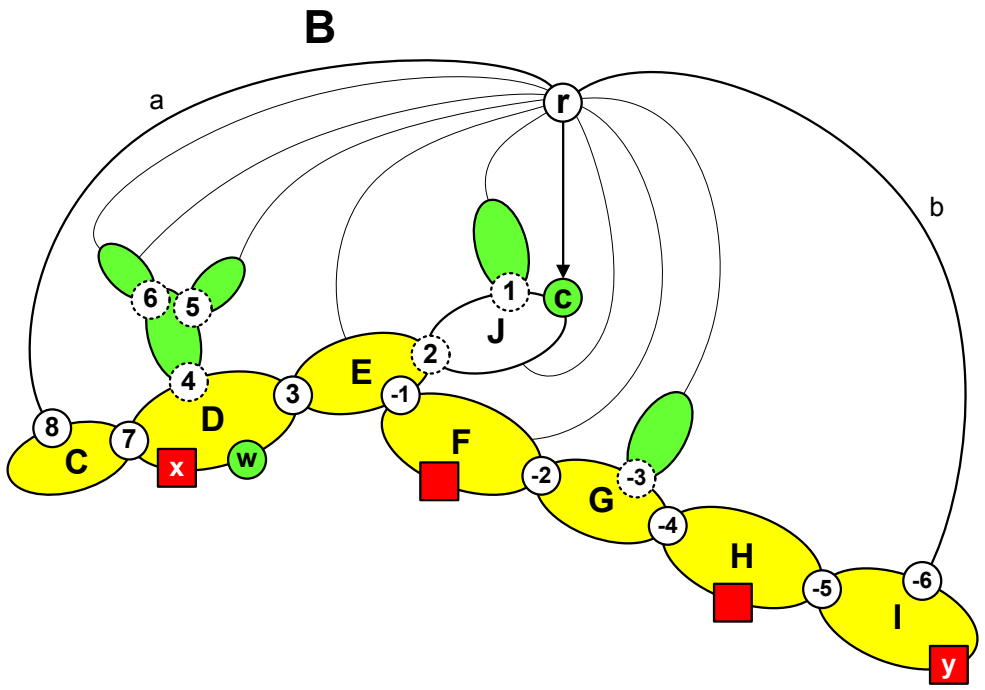


Abbildung 2.13: Die Bicomp B nach erfolgter Markierung der Mergeknoten. Die Markierung beginnt an Knoten c, wobei jeweils einmal nach links die Knoten 1 bis 8 und nach rechts die Knoten -1 bis -6 markiert werden. Die gestrichelt umrandeten Knoten sind innere Mergeknoten, deren Markierungen im Nachhinein gelöscht werden, da sie nicht auf der Außenfläche von B liegen. Um den ersten Endknoten des $HighestXYPath(w)$ zu erhalten, wird die Außenfläche von D ab w parallel traversiert, ohne die möglicherweise inkonsistenten Knotenorientierungen zu berücksichtigen. Sei 7 der zuerst erreichte Knoten. An 7 müssen OldLinks existieren, die hier allerdings wegen der Einbettung der Bicomp E nicht auf Kanten aus D verweisen. Daher kann 7 nicht die Wurzel der ehemaligen Bicomp D gewesen sein. Zudem ist 7 positiv, daher muss der andere Endknoten des $HighestXYPath(w)$ rechts vom Knoten 7 liegen. Der vorab berechnete $HighestFacePath(B)$ wird deswegen in diese Richtung traversiert, bis ein Knoten mit kleinerer Markierung gefunden wurde. An diesem Knoten 3 wird gestoppt und der bis dahin traversierte Pfad als $HighestXYPath(w)$ gespeichert. Es verbleibt, die Lage der Endknoten 7 und 3 in Bezug zu den Stoppknoten zu berechnen. Da auf dem Pfad $w \rightarrow x \rightarrow 7$ der Stoppknoten x gefunden wurde und keine ShortCircuit-Kante bis zu einem Knoten des Pfades $w \leftrightarrow 3$ existiert, muss 7 bei einer CCW-Traversierung von B vor dem Stoppknoten x und 3 vor dem Stoppknoten y liegen.

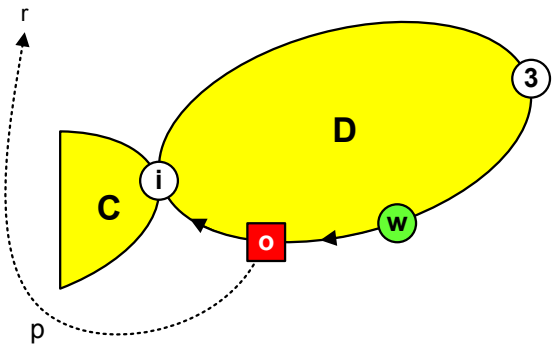


Abbildung 2.14: Vermeidung der ShortCircuit-Kante p während der Traversierung des Pfades $w \to o \to i$.

ten von D zeigen.

Das gibt einen Hinweis darauf, in welcher Richtung c liegt. Sei u der Mergeknoten der Bottom-Chain mit niedrigstem DFI-Wert. Ist die Markierung an i positiv, dann muss i links von dem Knoten u liegen. Existieren weiter die beschriebenen OldLinks an i, muss i vorher Wurzel von D gewesen sein und deswegen näher an u als der andere Endknoten des *HighestXYPath*(w) liegen. Dann ist $i = p_y$ und p_x liegt links davon auf dem *HighestFacePath*. Der *HighestXYPath*(w) wird berechnet, indem ab p_y in der *HighestFacePath*-Liste solange nach links traversiert wird, bis ein Knoten mit höherer Markierung gefunden wird. Da die Markierung aller inneren Mergeknoten gelöscht wurde, entspricht der gefundene Knoten genau p_x und die Extraktion ist beendet.

Falls i positiv markiert ist und an i keine OldLinks existieren, die auf Kanten aus D verweisen, ist $i = p_x$. Dann liegt sowohl p_y als auch u rechts von i auf dem *HighestFacePath* und der *HighestXYPath*(w) kann analog gefunden werden. Der Endknoten p_y wird dabei an dem ersten Knoten gefunden, der mit einem niedrigeren Wert als p_x markiert ist. Die Fälle, in denen i negativ ist, sind dazu symmetrisch. Auch dort kann immer entschieden werden, welcher Endknoten identisch mit i ist und in welche Richtung der *HighestXYPath*(w) verläuft.

Somit kann der *HighestXYPath*(w) mit seinen Endknoten p_x und p_y vollständig extrahiert werden. Dadurch stehen jetzt die Informationen für die Klassifikation der Minortypen an w zur Verfügung, die für Punkt 4 auf Seite 36 benötigt werden. Die Zusatzinformation für Minortyp D und AD, ob auf dem *HighestXYPath*(w) ein Knoten existiert, der über einen Pfad mit der Wurzel verbunden ist, kann dabei während der Extraktion des *HighestFacePath* in Linearzeit berechnet werden. Solche Knoten sind entweder direkt durch eine Backedge mit r verbunden oder stellen innere Mergeknoten dar.

Bemerkung. In der Praxis ist es sinnvoll, während der Traversierung jeden erreichten Knoten auf externe Aktivität zu überprüfen, bis entweder p_x, p_y oder ein Stoppknoten gefunden worden ist. Werden solche, von w verschiedene Knoten gefunden, stellen sie z-Knoten der Minortypen AE und E dar. Dann existiert zwangsläufig einer der beiden Minortypen und der Aufwand für eine explizite Berechnung der z-Knoten an diesem

Minortyp wird gespart.

Lage des *HighestXYPaths* Es bleibt die Lage der Endknoten p_x und p_y in Bezug auf die Stoppknoten zu ermitteln. In Punkt 5 auf Seite 36 wird dafür die Information benötigt, ob p_x bei einer CCW-Traversierung der Außenfläche von B vor oder nach dem Stoppknoten $stopX$ auftritt oder identisch zu ihm ist. Analog dazu wird die Information über die Lage von p_y in Bezug auf $stopY$ gebraucht.

Wenn sowohl p_x als auch p_y mit dem jeweiligen Stoppknoten identisch sind, ist deren Lage bekannt. Andernfalls muss die relative Position der Stoppknoten gefunden werden. Sei o. B. d. A. $i = p_x$, da der alternative Fall $i = p_y$ dazu symmetrisch ist. Dann wurde während der Traversierung der ehemaligen Bicomp D der gesamte Weg zu i durchlaufen. Am Ende des Walkdowns an B wurden wegen den zwei vorhandenen Stoppknoten ShortCircuit-Kanten zu diesen eingefügt. Ein traversierter Knoten des Pfade $w \rightarrow p_x$, an dem zum ersten Mal eine ShortCircuit-Kante zu r gefunden wurde, muss deswegen identisch mit einem Stoppknoten sein. Wegen $i = p_x$ kann das nur der Stoppknoten $stopX$ sein. Falls also eine ShortCircuit-Kante auf dem Pfad $w \rightarrow p_x$ gefunden wurde, befindet sich p_x zwangsläufig vor $stopX$ auf der Außenfläche von B. Andernfalls liegt p_x bei einer CCW-Traversierung von B hinter $stopX$.

Da die Pfade $w \rightarrow p_x$ und $w \rightarrow p_y$ parallel traversiert werden, besteht die Möglichkeit, dass auch der Stoppknoten $stopY$ gefunden wird. In dem Fall liegt $stopY$ bei einer CCW-Traversierung von B vor p_y und die Lage beider Endknoten ist bekannt. Falls allerdings keine ShortCircuit-Kante auf dem Pfad $w \rightarrow p_y$ gefunden wird, läßt sich daraus nicht die gegenteilige Aussage schließen. Der Pfad $w \rightarrow p_y$ ist nämlich durch die parallele Traversierung mindestens so lang wie der Pfad $w \rightarrow p_x$ und kann somit Knoten enthalten, die überhaupt nicht besucht werden. Die Menge dieser Knoten ist im Allgemeinen zu groß, um effizienten besucht werden zu können, deswegen muss anderweitig festgestellt werden, ob $stopY$ im Pfad $w \rightarrow p_y$ enthalten ist.

Dazu wird der Walkdown zu allen Zeitpunkten betrachtet, an dem in der Bicomp B eine Stoppkonfiguration gefunden wurde. Aufgrund der Stoppknoten müssen dann jedesmal an r zwei ShortCircuit-Kanten bis zu jedem der dort aufgetretenen Stoppknoten eingebettet worden sein. Vor dem Erzeugen dieser beiden ShortCircuit-Kanten kann der Walkdown jeweils ohne asymptotischen Mehraufwand alle Knoten an der Außenfläche mit c markieren, die überbrückt werden. Die eingebetteten zwei ShortCircuit-Kanten der zuletzt gefundenen Stoppkonfiguration enden dabei zwangsläufig an den aktuellen Stoppknoten $stopX$ und $stopY$ und überdecken alle ShortCircuit-Kanten vorheriger Stoppkonfigurationen an B. Ist jetzt $stopY$ im traversierten Pfad $w \rightarrow p_y$ enthalten, sind damit alle besuchten Knoten ab $stopY$ mit c markiert. Unter der Annahme, dass $p_y \neq stopY$ ist, muss deswegen lediglich p_y auf die Markierung c überprüft werden. Wurde diese Markierung gefunden, liegt p_y in einer CCW-Traversierung von B hinter $stopY$, andernfalls vor $stopY$.

Laufzeit

Theorem 2.3.13. *Sämtliche im Verlauf des Algorithmus benötigten* HighestXYPaths *und deren Lage können in linearer Laufzeit zu ihrer Länge extrahiert werden.*

Beweis. Die *HighestFacePaths* können nach Theorem 2.3.12 effizient extrahiert werden und liegen als doppelt verkettete Liste vor. Alle Mergeknoten können zusätzlich während der Berechnung des *HighestFacePath* in konstanter Zeit auf die entsprechenden Knoten der Liste verlinkt werden. Zudem kann durch die vorherige Markierung des *HighestFacePath* in konstanter Zeit entschieden werden, ob an w überhaupt ein *HighestXYPath*(w) existiert.

Ist der *HighestXYPath*(w) vorhanden, wird der gesamte *HighestFacePath* nummeriert. Für jeden *HighestFacePath* muss diese Nummerierung nur ein einziges Mal durchgeführt werden, da dieser sich nicht ändert, solange er gültig ist. Die auf dem Pfad zu nummerierenden Mergeknoten können in konstanter Zeit erkannt werden, indem die OldLinks analog zur Extraktion des *HighestFacePath* mit den ExternalLinks verglichen werden. Alle inneren Mergeknoten können gelöscht werden, indem sie als wiederholt besuchte Knoten erkannt werden. Diese Methode ist analog zur Extraktion des *HighestFacePath* während des Entfernens der Teilpfade separierter Bicomps und kann mit dieser verbunden werden. Folglich benötigt die Nummerierung der ausschließlich an der Außenfläche liegenden Mergeknoten nicht mehr Zeit, als die Extraktion des *HighestFacePath* selbst.

Ab w wird die parallele Traversierung an der Außenfläche der Bicomp D gestartet. Jeder Knoten muss dabei auf externe Aktivität und auf inzidente ShortCircuit-Kanten zu r geprüft werden. Beides benötigt nur konstante Zeit, wobei die nächste zu wählende Kante nach einer ShortCircuit-Kante eindeutig bestimmt werden kann. Da parallel in beide Richtungen traversiert wird, ist die Laufzeit durch die Länge des kürzeren Pfades bis zu den Endknoten p_x und p_y beschränkt. Die Kosten dafür werden auf das Kuratowski-Konto verbucht. In jedem Minortyp, der einen vorhandenen *HighestXYPath* voraussetzt, muss aber mindestens einer der beiden Pfade $w \to p_x$ und $w \to p_y$ für die Kuratowski-Subdivision extrahiert werden. Dadurch werden sämtliche Kosten durch Kuratowski-Extraktionen kompensiert. Würde die Traversierung einfach beide Pfade $w \to p_x$ und $w \to p_y$ einzeln durchlaufen, könnte ein quadratischer Mehraufwand nicht vermieden werden, da in manchen Minortypen nur einer der beiden Pfade $w \to p_x$ und $w \to p_y$ extrahiert werden muss.

Wurde p_x bzw. p_y erreicht, kann die Richtung, in der der gesuchte *HighestXYPath*(w) auf dem *HighestFacePath* liegt, durch die OldLinks und Knotennummerierungen in konstanter Zeit berechnet werden. In dieser Richtung kann der *HighestXYPath*(w) dann in linearer Zeit extrahiert werden. Die Lage der Endknoten ist zudem mit den vorab gemachten Walkdown-Markierungen in konstanter Zeit berechenbar. □

Damit liegen alle Informationen vor, die in den lokalen Erweiterungen benötigt werden und sämtliche Kuratowski-Subdivisions können aus der betrachteten Stoppkonfiguration extrahiert werden.

2.4 Gesamtlaufzeit

Die globalen Erweiterungen können mit den lokalen kombiniert werden. Falls im Verlauf des Algorithmus eine Stoppkonfiguration auftritt, werden durch die globalen Erweiterungen alle für die Extraktion der Kuratowski-Subdivisions benötigten Informationen berechnet. Mit Hilfe dieser Informationen werden lokal so viele Kuratowski-Subdivisions wie möglich aus der Stoppkonfiguration extrahiert und dabei sämtliche kritischen Merkmale gelöscht. Dadurch wird sichergestellt, dass die Einbettung trotz der gefundenen Kuratowsi-Subdivisions planar bleibt. Um den Einbettungsprozess weiterführen zu können, ermittelt der erweiterte Walkdown dann die Bicomp, die als nächstes eingebettet werden muss.

Die Korrektheit folgt daraus, dass alle auftretenden Kuratowski-Subdivisions so durch Löschen der kritischen Backedges verändert werden, dass der verbleibende Graph planar eingebettet werden kann. Falls Kuratowski-Subdivisions extrahiert werden konnten, ist der ursprüngliche Graph G nicht planar, andernfalls ist er planar.

Theorem 2.4.1. *Der Algorithmus zur Extraktion von Kuratowski-Subdivisions benötigt die lineare Laufzeit*

$$O(n + m + \sum_{K \in S} |E(K)|).$$

Beweis. Alle auftretenden Kosten wurden auf insgesamt vier Konten, das Face-Konto, das Kuratowski-Konto, das ShortCircuit-Konto und das E_2-Konto verteilt. Die Kosten, die auf diesen Konten während des erweiterten Walkups, des erweiterten Walkdowns und der Kuratowski-Extraktion anfallen, wurden getrennt analysiert. Die Tabelle 2.2 zeigt die Laufzeitabschätzungen für jeden dieser Berechnungsschritte.

Jeder aufgeführte Berechnungsschritt wird dabei über alle Knoteneinbettungen betrachtet. Die Gesamtlaufzeit ist daher die Summe aller Einzellaufzeiten, welche sich zu $O(n + m + \sum_{K \in S} |E(K)|)$ ergibt. Damit wurde die untere Schranke 2.1 tatsächlich erreicht und die Laufzeit des Algorithmus ist optimal. □

Berechnungsschritt	Erw.	Gesamtlaufzeit	Referenz	Konten
Erweiterter Walkup	global	$O(n+m+\sum_{K\in S}\lvert E(K)\rvert)$	Korollar 2.3.4	F,K,E_2
Erweiterter Walkdown	global	$O(n+m+\sum_{K\in S}\lvert E(K)\rvert)$	Theorem 2.3.9	F,K,S
– ShortCircuit-Kanten	global	$O(n+m)$	Theorem 2.3.6	S
Zusätzliche Backedgepfade	lokal	$O(\sum_{K\in S}\lvert E(K)\rvert)$	Abschnitt 2.2.1	K
Klassifikation der Minortypen	lokal	$O(n+m+\sum_{K\in S}\lvert E(K)\rvert)$	Abschnitt 2.2.2	K
Extraktion...				
– ...von v,r,$stopX$,$stopY$	global	$O(m)$	S. 36 Punkt 1	K
– ...kritischer Backedges	global	$O(n+m+\sum_{K\in S}\lvert E(K)\rvert)$	Proposition 2.3.10 und S. 36 Punkt 2	K
– ...externer Backedgepfade	lokal	$O(\sum_{K\in S}\lvert E(K)\rvert)$	S. 36 Punkt 3	K
– ...der *HighestFacePaths*	global	$O(n+m)$	Theorem 2.3.12	K
– ...und Lage der *HighestXYPaths*	global	$O(\sum_{K\in S}\lvert E(K)\rvert)$	Theorem 2.3.13 und S. 36 Punkt 4+5	K
– ...externer z-Knoten für die Minortypen E/AE	lokal	$O(\sum_{K\in S}\lvert E(K)\rvert)$	S. 36 Punkt 6	K
Extraktion aller Minortypen	lokal	$O(\sum_{K\in S}\lvert E(K)\rvert)$	Abschnitt 2.2.2 und [BM04]	K

Tabelle 2.2: Übersicht über die Laufzeiten der einzelnen Berechnungsschritte des Algorithmus. Die erste Spalte gibt an, ob der jeweilige Berechnungsschritt Teil der lokalen oder der globalen Erweiterung ist. Die darauf folgende Gesamtlaufzeit wird über alle Knoteneinbettungen gemessen. Das K ist dabei Element der Menge aller extrahierten Kuratowski-Subdivisions. Referenzen auf die zugehörigen Laufzeitbeweise werden in der nächsten Spalte gegeben. In der letzten Spalte werden die Konten aufgeführt, auf die die Kosten des Berechnungsschrittes verteilt werden.

3 Eigenschaften und Erweiterungen des Algorithmus

In den folgenden Abschnitten werden ausgewählte Eigenschaften des Algorithmus analysiert und mögliche Erweiterungen diskutiert. Dazu gehört die Abgrenzung der ausgegebenen Einbettung zu einem *Maximal Planar Subgraph* in Abschnitt 3.1 und eine Modifizierung des Algorithmus in Abschnitt 3.2, die ermöglicht, eindeutige Kuratowski-Subdivisions zu extrahieren.

In Abschnitt 3.3 wird untersucht, wie möglichst verschiedene bzw. möglichst ähnliche Kuratowski-Subdivisions extrahiert werden können. Ein möglicher Weg, die Anzahl der extrahierten Kuratowski-Subdivisions zu erhöhen, stellt eine Randomisierung des zugrunde liegenden DFS-Baums dar (siehe Abschnitt 3.4). Dasselbe Ziel verfolgt die sogenannte *Bundle*-Variante in Abschnitt 3.5, die den Algorithmus grundlegender erweitert, dafür allerdings eine superlineare Laufzeit aufweist.

3.1 Abgrenzung zum *Maximal Planar Subgraph*

Der Algorithmus gibt zu jedem Eingabegraphen G eine planare Einbettung eines Subgraphen $H \subseteq G$ aus. Falls G planar ist, gilt $H = G$. Andernfalls ist G nicht planar und $H \subset G$ stellt einen echten Subgraphen dar.

Der Subgraph H muss von dem Begriff des *Maximal Planar Subgraphs* abgegrenzt werden. Dieser ist als planarer Subgraph definiert, der durch jede weitere hinzugefügte Kante nicht planar wird. In G können dagegen Kanten $e \notin E(H)$ existieren, die zu H hinzugefügt werden können, ohne dessen Planarität zu zerstören (siehe Abbildung 3.1).

Der Grund dafür ist, dass eine kritische Backedge a aufgrund zweier externer Backedges b_1 und b_2 eine Stoppkonfiguration bilden und deswegen gelöscht werden kann. Seien b_1 und b_2 die einzigen externen Backedges dieser Stoppkonfiguration. Falls jetzt in einer späteren Einbettungsphase beispielsweise die Backedge b_1 kritisch wird, muss auch sie gelöscht werden. Dann hätte aber a nicht gelöscht werden müssen und die Kante a kann demnach zur Einbettung hinzugefügt werden, ohne deren Planarität zu verletzten. Der Algorithmus findet also im Allgemeinen keinen Maximal Planar Subgraph.

Allerdings stellt dessen Ausgabe eine Heuristik für einen solchen Subgraphen dar und kann in einem Branch-and-Bound- oder Branch-and-Cut-Szenario für die Entwicklung unterer Schranken nützlich sein. Insbesondere ist diese Heuristik in Kombination mit den extrahierten Kuratowski-Subdivisions für das *Maximum Planar Subgraph Problem* interessant, das einen kardinalitätsmaximalen Maximal Planar Subgraph berechnet.

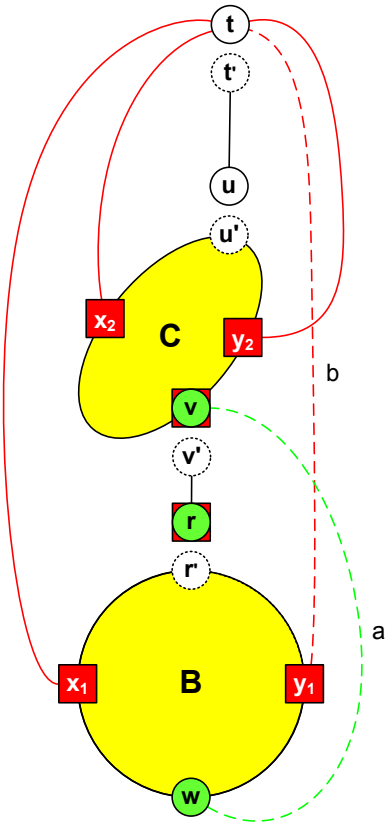


Abbildung 3.1: Beispiel eines Graphen, bei dem der Algorithmus keinen Maximal Planar Subgraph als planare Einbettung ausgibt. Die Kante a ist in der Einbettungsphase des Knotens v kritisch und wird aufgrund der in der Bicomp B gefundenen Stoppkonfiguration gelöscht. In der darauf folgenden Einbettungphase des Knotens u entsteht die Bicomp C. Der Knoten v wird während der Einbettung des Knotens t durch die Backedge b kritisch und verursacht eine zweite Stoppkonfiguration, weswegen b gelöscht wird. Damit hätte die Backedge a nicht gelöscht werden müssen.

3.2 Eindeutigkeit extrahierter Kuratowski-Subdivisions

In den lokalen Erweiterungen wurde bereits gezeigt, dass alle extrahierten Kuratowski-Subdivisions aus einer Stoppkonfiguration unterschiedlich sind (Lemma 2.2.2). Das Lemma kann auf Kuratowski-Subdivisions aus beliebigen Stoppkonfigurationen verallgemeinert werden:

Unterschiedliche Stoppkonfigurationen enthalten verschiedene kritische Backedges, da diese nach jeder Extraktion einer Stoppkonfiguration gelöscht werden. Wenn Minortypen betrachtet werden, die die jeweilige kritische Backedge enthalten, sind die zugehörigen Kuratowski-Subdivisions damit verschieden.

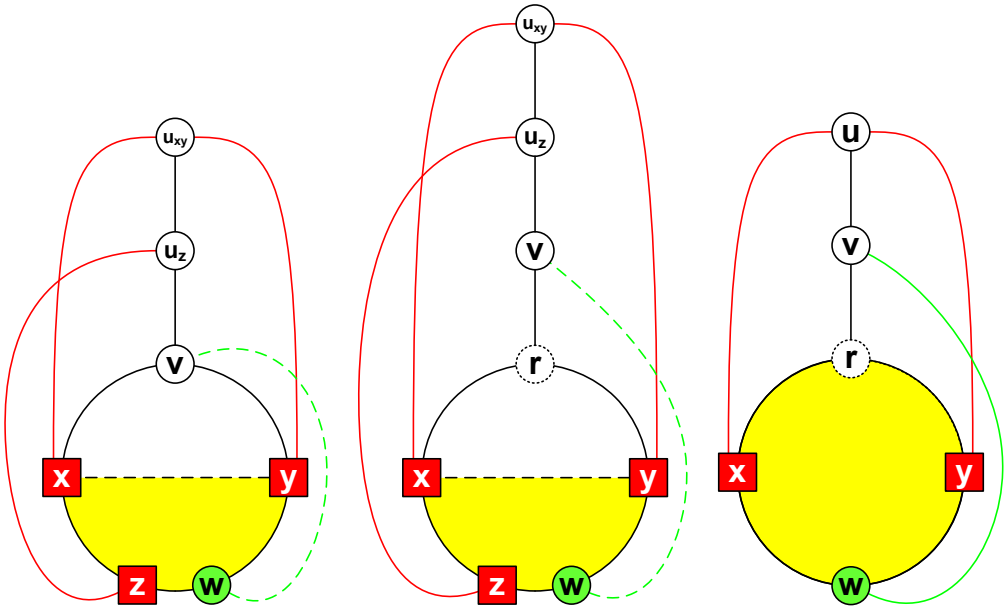
Ausnahmen bilden lediglich die beiden Minortypen E_2 und AE_2 (siehe Abbildung 3.2), an denen die jeweilige kritische Backedge nicht im unterliegenden $K_{3,3}$ benötigt wird. Angenommen, ein Minor vom Typ E_2 oder AE_2 wird gefunden. Alle kritischen Backedges der Stoppkonfiguration werden dann gelöscht, allerdings bleiben die drei externen Backedgepfade an *stopX*, *stopY* und an dem externen Knoten z unter dem *HighestXYPath* bestehen. Sei u_z der Endknoten der Backedge, welche die externe Aktivität des Knotens z verursacht. Analog dazu seien u_x und u_y die beiden Endknoten der gewählten, externen Backedges, die die Stoppknoten extern machen.

Nach Definition der Minortypen E_2 und AE_2 besitzt u_x den höchsten DFI-Wert der drei Endknoten. In den folgenden Einbettungsphasen wird damit zuerst der Knoten u_x eingebettet. Zu diesem Zeitpunkt ist der ehemals externe Backedgepfad an z pertinent, da die zugehörige Backedge an u_z endet. Da die beiden ehemaligen Stoppknoten weiterhin extern sind, ist er sogar kritisch und verursacht eine Stoppkonfiguration. Falls die z enthaltende Bicomp B nicht vorher durch beispielsweise einen zweiten Backedgepfad an einem der Stoppknoten eingebettet wurde, tritt damit der Minortyp A auf. Hier ist es möglich, dass exakt dieselben Kanten extrahiert werden, die schon Bestandteil der Kuratowski-Subdivision des ursprünglichen E_2- bzw. AE_2-Typs waren.

Damit werden Kuratowski-Subdivisions aus E_2 bzw. AE_2-Minortypen unter Umständen zweimal extrahiert. Weitere Extraktionen derselben Kuratowski-Subdivision können danach nicht mehr auftreten, da spätestens nach der Extraktion des Minortyps A dessen kritische Backedge gelöscht wird und diese Kante Bestandteil der Kuratowski-Subdivision ist.

Zusätzlich ist es möglich, dass Kuratowski-Subdivisions des Minortyps AE_2 bis zu $O(n)$ Mal identisch extrahiert werden. Das ist beispielsweise dann der Fall, wenn die Bicomp B in folgenden Einbettungsphasen nicht eingebettet werden kann, aber in jeder Phase ein ehemals externer Backedgepfad kritisch wird. Dann kann es vorkommen, dass in jeder einzelnen Phase der Minortyp AE_2 mit identischer Kantenmenge extrahiert wird.

Beide Arten von Extraktionen identischer Kuratowski-Subdivisions lassen sich aber vermeiden. Es ist möglich, für die Minortypen E_2 und AE_2 verschiedene Markierungen zu benutzen, mit denen nach jeder Extraktion die eindeutige externe Backedge markiert wird, die die externe Aktivität von z verursacht. Bei jeder folgenden Extraktion eines Minortyps AE_2 kann dann die an u_z endende, externe Backedge auf eine solche Mar-



(a) Minortyp E_2 (b) Minortyp AE_2 (c) Minortyp A

Abbildung 3.2: Die drei in die Extraktion von identischen Kuratowski-Subdivisions verwickelten Minortypen.

kierung in $O(1)$ überprüft werden. Falls der Test positiv ausfällt, wird die Extraktion des Minortyps abgebrochen. Analog dazu wird bei einem auftretenden Minortyp A die eindeutige, kritische Backedge auf eine solche Markierung untersucht.

Somit wird einerseits vermieden, dass mehrfach dieselben Kuratowski-Subdivisions an AE_2-Typen extrahiert werden. Andererseits können auch in Minortypen A nicht dieselben Kuratowski-Subdivisions wie in vorigen Minortypen E_2 oder AE_2 extrahiert werden, da dann der kritische Backedgepfad identisch mit dem externen sein müsste. Da an jeder Kante nur konstant viele Informationen abgefragt werden müssen, wird durch diese Tests asymptotisch kein Mehraufwand verursacht. Auch die Eigenschaft des Algorithmus, dass bei einem nicht planaren Graphen immer mindestens eine Kuratowski-Subdivision gefunden wird, bleibt erhalten.

Damit gilt sowohl für unterschiedliche als auch für identische Stoppkonfigurationen, dass extrahierte Kuratowski-Subdivisions in mindestens einer Kante unterschiedlich sind. Damit wurde folgende Aussage gezeigt.

Proposition 3.2.1. *Der Algorithmus kann so modifiziert werden, dass für jedes Paar* $\{K_1, K_2\} \in \binom{S}{2}$ *die Kuratowski-Subdivisions* K_1 *und* K_2 *in mindestens einer Kante verschieden sind.*

3.3 Verschiedene und ähnliche Kuratowski-Subdivisions

In einem Branch-and-Cut-Szenario, das die extrahierten Kuratowski-Subdivisions als Nebenbedingungen nutzt, ist es wünschenswert, dass diese entweder möglichst viele oder möglichst wenige gemeinsame Kanten enthalten. Das lässt sich auf unterschiedlichen Wegen erreichen.

Mit der jeweiligen Einbettungsphase der Kuratowski-Subdivisions existiert beispielsweise ein Merkmal, nach dem diese gruppiert oder auch unterschieden werden können. Kuratowski-Subdivisions, die in derselben Einbettungphase extrahiert werden, teilen sich tendenziell mehr Kanten als Kuratowski-Subdivisions aus unterschiedlichen Einbettungsphasen und umgekehrt. Ein zweites, spezielleres Unterscheidungsmerkmal stellt eine identische Stoppkonfiguration dar, in der sich extrahierte Kuratowski-Subdivisions jeden Typs einen großen Teil der Bicomp-Außenfläche teilen. Der Minortyp selbst erzwingt oder vermeidet dabei zusätzlich eine strukturelle Abhängigkeit.

Durch eine alternative Wahl der kritischen und externen Backedgepfade können, wie in Abschnitt 2.2.1 beschrieben, weitere Kuratowski-Subdivisions extrahiert werden. Dieses Konzept ermöglicht beinahe identische Kuratowski-Subdivisions, die sich lediglich in einem einzigen Pfad, nämlich dem gewählten Backedgepfad, unterscheiden.

Sämtliche Unterscheidungsmerkmale lassen sich während des Algorithmus in konstanter Zeit berechnen. Damit kann die Menge der Kuratowski-Subdivisions effizient in möglichst unterschiedliche oder möglichst ähnliche Gruppen partitioniert werden.

3.4 Randomisierung

Um die Anzahl der extrahierten Kuratowski-Subdivisions weiter zu erhöhen, kann Randomisierung verwendet werden. Hierbei werden vorab in der Datenstruktur des Graphen alle Knoten- und Kantenreihenfolgen pseudo-gleichverteilt permutiert. Mit einer darauf folgenden DFS-Traversierung wird dann der dem Algorithmus zugrunde liegende DFS-Baum erstellt.

Durch Anwendung des Algorithmus auf den so randomisierten DFS-Baum können neue Kuratowski-Subdivisions gefunden werden, die aufgrund von Kantenlöschungen sonst nicht aufgetreten wären. Dieser Randomisierungsprozess kann so oft iteriert werden, bis eine gewünschte Anzahl von Kuratowski-Subdivisions erreicht wurde. Allerdings muss dabei beachtet werden, dass die Kuratowski-Subdivisions verschiedener Iterationen dann nicht mehr notwendigerweise verschieden sind.

3.5 *Bundle*-Variante

Bisher wurden für die Extraktion der Kuratowski-Subdivisions ausschließlich Backedgepfade benutzt, die Teile der Außenfläche von Bicomps durchlaufen. Diese Bicomps sind aber oft nicht entartet und enthalten deswegen häufig innere, zur Außenfläche alternative

Backedgepfade. Die Idee der *Bundle*-Variante des Algorithmus ist, diese inneren Pfade für zusätzliche Kuratowski-Subdivisions zu verwenden.

Ein *Bundle* ist dabei die Menge von nicht notwendigerweise kreuzungsfreien s-t-Pfaden einer Bicomp B, wobei s deren Wurzel und t der Eintrittsknoten des Backedgepfades in B ist. Wird während der Extraktion einer Kuratowski-Subdivision ein solches Bundle gefunden, können daraus beliebige s-t-Pfade als Teil von neuen Backedgepfaden gewählt werden. Neben der Erhöhung der Anzahl der extrahierten Kuratowski-Subdivisions werden dadurch zusätzlich sehr ähnliche Kuratowski-Subdivisions erzeugt.

Allerdings kann die Menge der enthaltenen s-t-Pfade eines Bundles exponentiell groß sein, da in einer Bicomp während des Algorithmus beinahe beliebige planare Strukturen auftreten können. Abbildung 3.3 zeigt eine gitterähnliche Innenstruktur einer Bicomp B, in der exponentiell viele s-t-Pfade existieren. Um die verschiedenen s-t-Pfade zu berechnen, kann *Backtracking* angewendet werden. Bei s startende Teilpfade werden dabei rekursiv mit allen möglichen, vorhandenen Kanten fortgeführt, bis der Endknoten t schließlich erreicht ist und das Verfahren zum letzten Knoten zurückkehrt.

Mit Backtracking können alle verschiedenen s-t-Pfade berechnet werden, allerdings tritt dabei folgendes Laufzeitproblem auf (siehe Abbildung 3.3): Angenommen, der Teilpfad $s \to x$ wird besucht und die nächste zu wählende Kante sei e. Falls dann e und der Endknoten t durch Löschen des Teilpfades in verschiedenen Zusammenhangskomponenten liegen, kann frühestens nach Rückkehr des Backtracking-Verfahrens zu x ein weiterer s-t-Pfad gefunden werden. Zwischen zwei gefundenen Pfaden werden also im Worst-Case exponentiell viele ungültige Pfade besucht, die nicht an t enden.

Ein solcher exponentieller Mehraufwand lässt sich aber vermeiden, indem dynamische Verfahren zur Verwaltung der Zusammenhangskomponenten benutzt werden. In diesem Szenario werden auf einem Graphen G' üblicherweise folgende Operationen unterstützt:

- *insert*(u, v): Fügt die Kante $\{u, v\}$ in G' ein.
- *delete*(u, v): Entfernt die vorhandene Kante $\{u, v\}$ aus G'.
- *connected*(u, v): Testet, ob die Knoten u und v in derselben Zusammenhangskomponente von G' liegen.

In dynamischen Graphproblemen kann der Graph durch eine Folge dieser Operationen verändert werden. Dabei sind die jeweils nächsten Operationen nicht bekannt. Das Problem, diese Operationen möglichst effizient umsetzen zu können, kann auf die Verwaltung eines *aufspannenden Waldes* zurückgeführt werden. Eppstein, Italiano, Tamassia, Tarjan, Westbrook und Yung bewiesen 1992 für planare Graphen, dass ein solcher aufspannender Wald in $O(\log n)$ pro Operation verwaltet werden kann [EIT+92]. Holm, de Lichtenberg und Thorup verallgemeinerten dieses Resultat 2001 auf allgemeine Graphen mit einer polylogarithmischen Laufzeit [HLT01]. Die logarithmische Laufzeit auf planaren Graphen ist nach einer unteren Schranke von Pătraşcu und Demaine [PD04] optimal. Sie gilt sogar, falls lediglich der Zusammenhang des Gesamtgraphen erfragt wird.

Das Resultat von Eppstein et al. kann dazu verwendet werden, den exponentiellen Mehraufwand des Backtrackings zwischen zwei gefundenen Pfaden auf einen logarithmischen

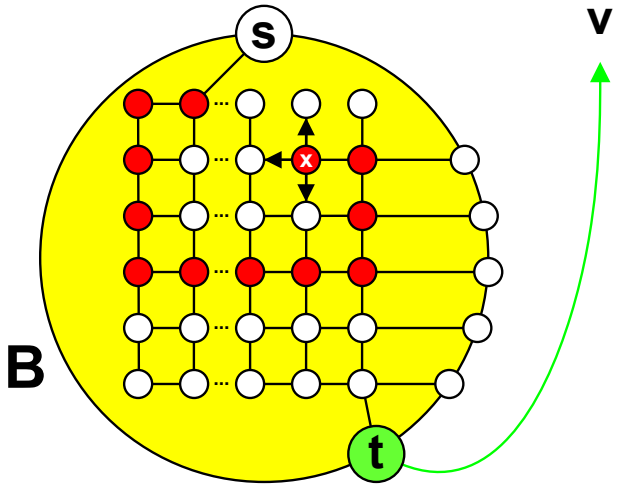


Abbildung 3.3: Eine Bicomp B mit Wurzel s und einem kritischen Backedgepfad, der B zwischen s und t durchquert. Die Bundle-Variante extrahiert alle s-t Pfade innerhalb von B mittels Backtracking. Sei dabei bereits der rot markierte Teilpfad $s \rightarrow x$ gefunden. Unabhängig von der nächsten gewählten Kante an x kann dieser Teilpfad nicht zu einem s-t-Pfad erweitert werden. Der Grund ist, dass x und t in verschiedenen Zusammenhangskomponenten des Graphen liegen, wenn bereits besuchte Knoten und Kanten gelöscht werden. Mittels Backtracking würden hier exponentiell viele weitere Pfaderweiterungen von $s \rightarrow x$ berechnet werden, die letztendlich alle nicht an t enden können und somit ungültig sind. Diese Situation kann durch eine dynamische Verwaltung des Zusammenhangs mit logarithmischem Mehraufwand pro gewählter Kante vermieden werden.

Mehraufwand zu reduzieren. Dazu wird jede Kante $e = \{u, v\}$ nach ihrer Traversierung temporär gelöscht. Durch *connected*(v, t) kann dann in logarithmischer Zeit getestet werden, ob von v aus noch ein Pfad zu t existiert. Wenn das der Fall ist, fährt das Backtracking fort und findet irgendwann diesen Pfad. Andernfalls befinden sich v und t in verschiedenen Zusammenhangskomponenten und die Kante e wird im aktuellen Backtrackingschritt übergangen. Insgesamt ensteht pro gefundenem s-t-Pfad einer Bicomp B ein Mehraufwand von $O(\log |E(B)|)$. Da die Kantenanzahl jeder Bicomp nach dem Theorem von Euler (siehe 1.3.6) durch $3n-6$ beschränkt ist, ergibt sich folgende superlineare Gesamtlaufzeit:

Proposition 3.5.1. *Die Gesamtlaufzeit der Bundle-Variante kann durch*

$$O(n + m + \log n \sum_{K \in S} |E(K)|)$$

abgeschätzt werden.

3.5.1 Anwendungsgebiete von Bundles

Die durch Bundles erzeugten Mengen von Kuratowski-Subdivisions stellen eine mögliche Zusatzinformation für die Anwendungsgebiete des Algorithmus dar.

So ist es bei dem Branch-and-Cut-Ansatz zur Lösung des Maximum Planar Subgraph Problems sinnvoll, dass bevorzugt solche Kanten gelöscht werden, die nicht Teil eines Bundles sind. Dazu kann jede Nebenbedingung einer Kuratowski-Subdivision so formuliert werden, dass entweder eine Nicht-Bundlekante gelöscht werden muss oder für jeden s-t-Pfad des Bundles mindestens eine Kante gelöscht werden muss. Hintergrund ist dabei, dass das Entfernen einer Kante auf einem s-t-Pfad nur dann die Kuratowski-Subdivision entfernt, wenn auch alle anderen s-t-Pfade des Bundles getrennt werden.

Ein weiterer Anwendungsfall ergibt sich bei dem Branch-and-Cut-Ansatz zur Kreuzungsminimierung. Auch hier können bevorzugt solche Kantenkreuzungen gewählt werden, die keine Bundlekanten enthalten. Andernfalls würden neben der Bundlekante zusätzlich Kanten aus allen anderen s-t-Pfaden des Bundles gekreuzt werden. Das führt bei größeren Bundles zu einer großen Anzahl von Kreuzungen und damit tendenziell eher zu einer suboptimalen Lösung als das Löschen von Nicht-Bundlekanten.

4 Experimentelle Analyse

Der erweiterte Algorithmus und dessen Bundle-Variante wurden als Teil des *Open Graph Drawing Frameworks* (OGDF, C++-basiert) implementiert. Dabei kam fortwährend das Konzept des Algorithm-Engineerings zum Einsatz. So bestand der gesamte Entwicklungsprozess aus einem wiederkehrenden Kreislauf von Entwurf, Analyse, Implementierung und experimenteller Bewertung.

Getestet wurde auf einem Intel P4 mit 3GHz und 1GB RAM mit dem GNU-Compiler gcc-3.3.3 (-O1). Ein Teil der Tests wurde auf der sogenannten *Rome-Library*, einer von Di Battista et al. [BGL+97] vorgeschlagenen Testmenge von insgesamt 11528 Graphen, durchgeführt. Diese enthält 8249 nicht planare Graphen im Knotenintervall $[11; 100]$. Die übrigen Testgraphen wurden durch einen Graphgenerator der OGDF zufällig erzeugt. Diese Zufallsgraphen besitzen weder Mehrfachkanten, noch Self-Loops und enthalten bei n Knoten $2n$ Kanten.

Aufgrund der Komplexität der Algorithmen sind während der Implementierung einige Berechnungsschritte vereinfacht worden:

So werden beispielsweise Stoppkonfigurationen an einer Bicomp B erkannt, indem die gesamte Außenfläche nach kritischen Knoten durchsucht wird. Für die Identifizierung der kritischen Backedges der Stoppkonfiguration wird vorerst der $MaxPoint(z)$ definiert. Dieser gibt den höchsten DFI-Wert eines Knotens im DFS-Teilbaum mit Wurzel z zurück. Angenommen, wir möchten für die Stoppkonfiguration in Bicomp B alle kritischen Backedges berechnen. Sei v der einzubettende Knoten und sei c der eindeutige DFS-Nachfolger der Wurzel von B. Dann kann für jede bisher nicht eingebettete Backedge $e = \{x, v\}$ in $O(1)$ festgestellt werden, ob der DFI-Wert von x im Intervall $[DFI(c); MaxPoint(c)]$ enthalten ist. Genau wenn das der Fall ist, stellt e eine kritische Backedge dar.

Weitere Vereinfachungen werden bei der Extraktion des *HighestFacePaths* sowie der *HighestXYPaths* einer Bicomp B gemacht. Hier werden vorab alle entsprechend markierten Bicomps geflippt, so dass die gesuchten Pfade durch eine in [BM04] beschriebene Traversierung der inneren Face an der Wurzel von B erhalten werden kann.

Allen beschriebenen Vereinfachungen ist gemeinsam, dass deren Verwendung zu einer theoretisch superlinearen Laufzeit führt. Wie sich zeigt, sind die Implementationen in der Praxis dennoch sehr schnell. Als Referenz werden in Abbildung 4.1 die Laufzeiten der OGDF-Implementationen der originalen Planaritätstests von Boyer-Myrvold und Booth-Lueker auf den Graphen der Rome-Library gezeigt. Diese beiden Tests werden mit dem hier vorgestellten, erweiterten Algorithmus in der Variante ohne (siehe Abbildung 4.1) und mit Bundles (siehe Abbildung 4.2) verglichen. Alle Testergebnisse mussten dabei

wegen der kurzen Laufzeiten über 100 Iterationen gemittelt werden und beziehen sich auf jeweils einen einzelnen Graphen.

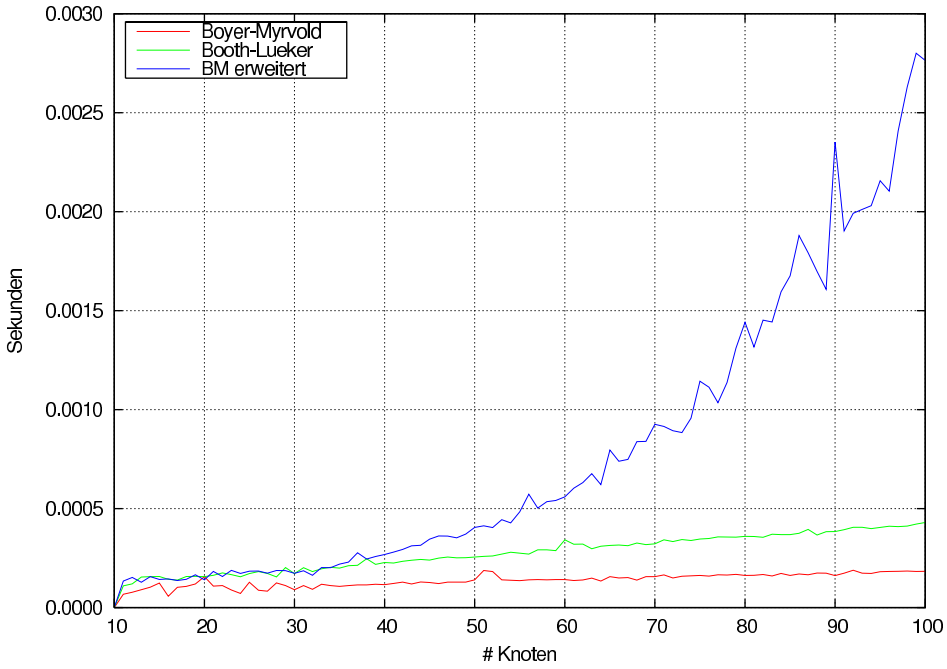


Abbildung 4.1: Rome-Library: Die Laufzeiten der Planaritätstests von Boyer-Myrvold und Booth-Lueker im Vergleich mit der Laufzeit des hier vorgestellten Algorithmus (BM erweitert). Jeder Graph wurde in einer Zeit von unter 3 ms abgearbeitet.

Der Mehraufwand gegenüber den Planaritätstests ist mit der wachsenden Anzahl extrahierter Kuratowski-Subdivisions zu erklären. Deren Kanten sind Bestandteil der Laufzeit und müssen deswegen berücksichtigt werden, um die Linearität des erweiterten Algorithmus beurteilen zu können. Die Abbildung 4.3 zeigt, dass die theoretisch lineare Laufzeit von $O(n+m+\sum_{K\in S}|E(K)|)$ im Wesentlichen auch in der Praxis erreicht wird. Wichtig ist dabei, dass die Genauigkeit der Testdaten bei höheren x-Achsenwerten abnimmt, da nur an wenigen Graphen der Rome-Library mehr als beispielsweise 30000 Kuratowski-Kanten extrahiert wurden.

Die Bundle-Variante kann in der Rome-Library bis zu 3,5 Millionen Kuratowski-Kanten pro Graph extrahieren und benötigt für jeden Graphen höchstens 0,45 Sekunden (siehe Abbildung 4.4). Die maximale Anzahl der extrahierten Kuratowski-Subdivisions ist dabei mit 3500 wesentlich höher als die der Variante ohne Bundles mit 250 (siehe Abbildung 4.5). Allerdings ist die Variante ohne Bundles mit maximal 0,13 Sekunden pro Graph schneller.

Eine prozentuale Auflistung der extrahierten Minortypen der Variante ohne Bundles zeigt Abbildung 4.6. Der Minortyp A ist bei kleinen Graphen sehr stark vertreten und

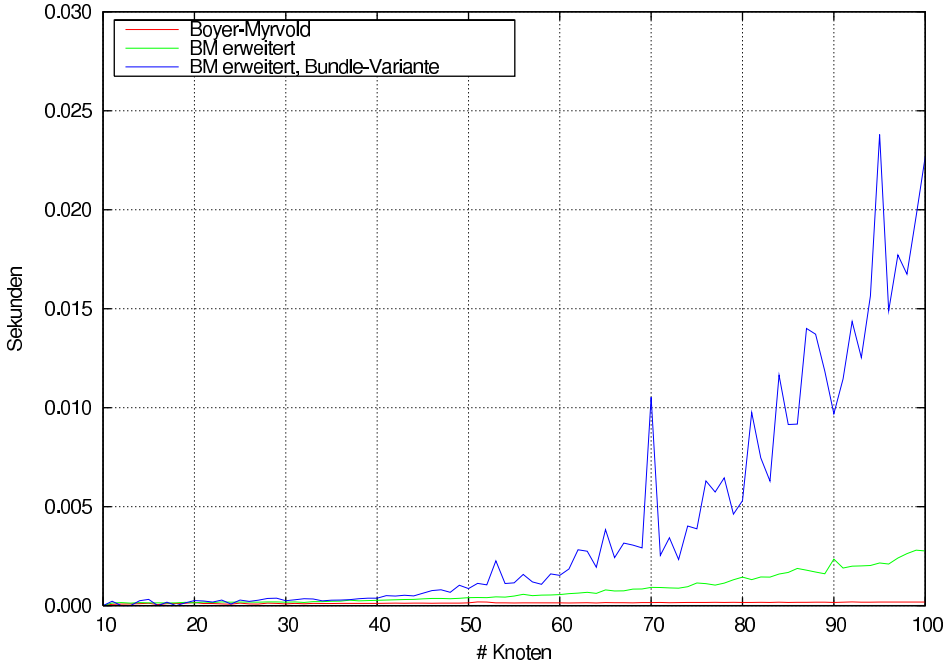


Abbildung 4.2: Rome-Library: Die Laufzeiten des Boyer-Myrvold-Planaritätstests und des erweiterten Algorithmus mit und ohne Bundles.

verschwindet dann mit steigender Knotenanzahl. Aus Minoren des neu klassifizierten Typs AE, der aus den Untertypen AE_1–AE_4 besteht, können in größeren Graphen die meisten Kuratowski-Subdivisions extrahiert werden. Die Minortypen B, AB, C, AC, D, AD wurden aufgrund des geringen Vorkommens zusammengefasst.

Im Folgenden werden Zufallsgraphen mit 10–500 Knoten betrachtet, deren Kantenanzahl jeweils doppelt so hoch wie die Knotenanzahl ist. Die Verteilung der Minortypen wird dabei aufgrund der höheren Knotenanzahl von dem Minortyp AE dominiert (siehe Abbildung 4.7).

Der erweiterte Algorithmus wurde so implementiert, dass angegeben werden kann, nach wie vielen gefundenen Kuratowski-Subdivisions bzw. Stoppkonfigurationen abgebrochen wird. Dadurch wird auch eine Anwendung auf sehr großen Graphen ermöglicht. Um den Unterschied deutlich zu machen, sind für die Zufallsgraphen drei Testläufe des Algorithmus ohne Bundles ausgeführt worden: In dem ersten wurde die Anzahl der extrahierten Kuratowski-Subdivisions nicht begrenzt, während in dem zweiten ein Limit von 10000 festgesetzt wurde. Der dritte Testlauf extrahierte lediglich die Kuratowski-Subdivisions aus der ersten 30 gefundenen Stoppkonfigurationen. Die resultierenden Laufzeiten und Anzahlen der extrahierten Kuratowski-Subdivisions finden sich in den Abbildungen 4.8, 4.9 und 4.10.

Die Testdaten unterlagen dabei einer hohen Varianz und wurden deswegen aus 10 Zufallsgraphen für jede Knotenanzahl gemittelt. Bei der Beschränkung auf 10000 Kuratowski-

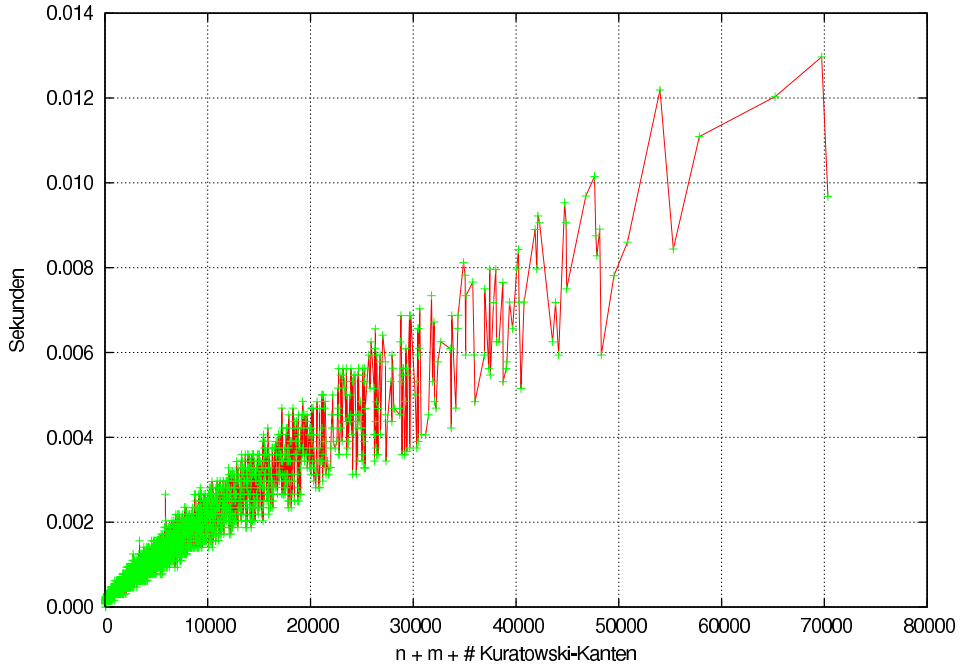


Abbildung 4.3: Rome-Library: Die Laufzeit des erweiterten Algorithmus im Vergleich zu den extrahierten Kuratowski-Kanten.

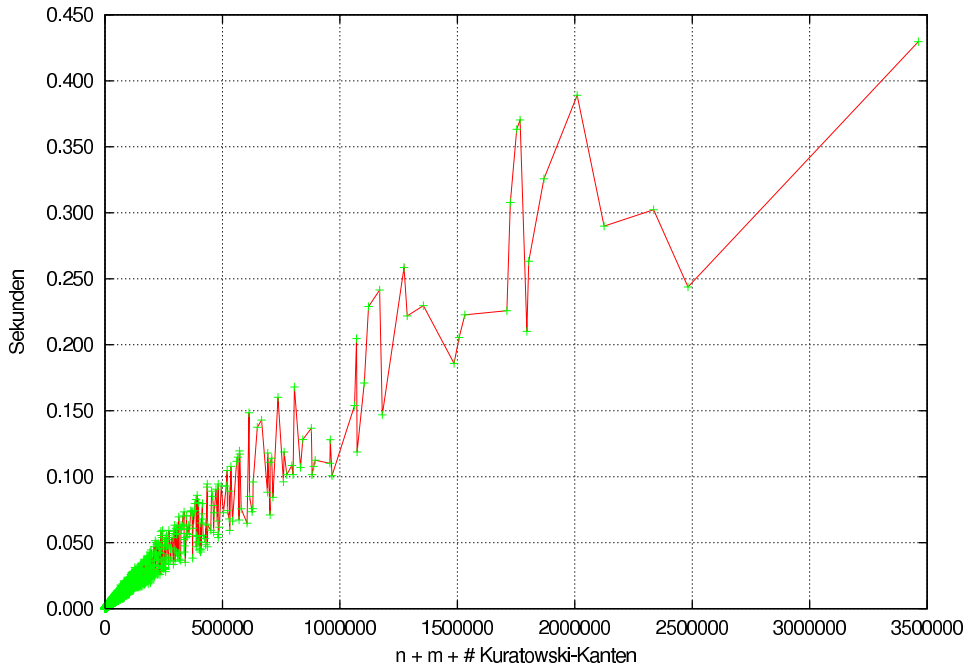


Abbildung 4.4: Rome-Library: Die Laufzeit der Bundle-Variante im Vergleich zu den extrahierten Kuratowski-Kanten.

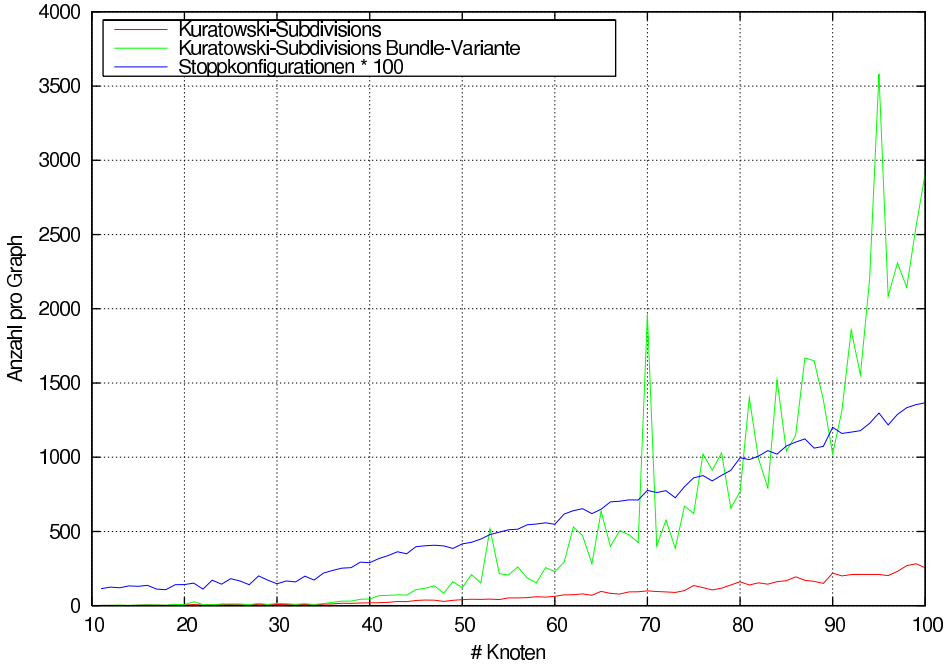


Abbildung 4.5: Rome-Library: Die Anzahl der extrahierten Kuratowski-Subdivisions mit und ohne Bundles. Zusätzlich ist die Anzahl der gefunden Stoppkonfigurationen (*100) angegeben, welche bei beiden Varianten identisch ist.

Subdivisions ist es somit nicht überraschend, dass die gemittelte Anzahl unterhalb von 10000 liegt, auch wenn die unbeschränkte Variante diese Grenze überschreitet.

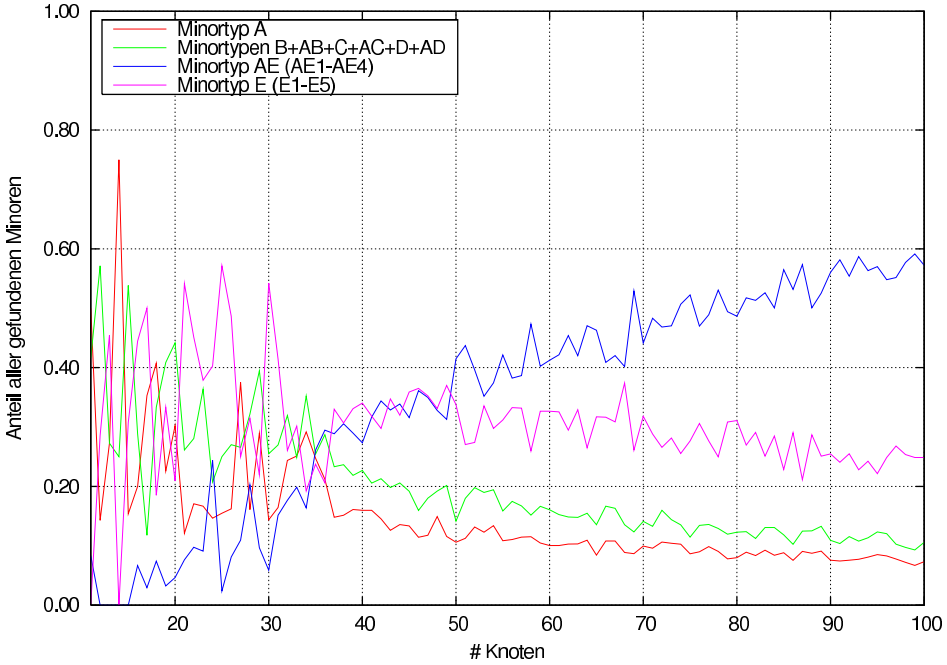


Abbildung 4.6: Rome-Library: Der Anteil der verschiedenen Minortypen an der Gesamtanzahl gefundener Minoren.

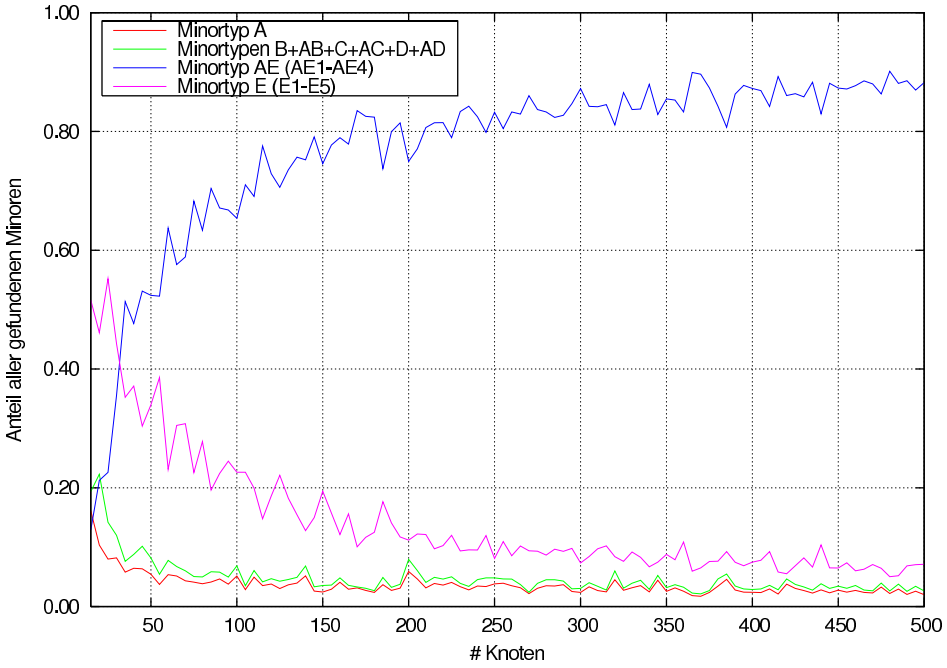


Abbildung 4.7: Zufallsgraphen: Der Anteil der verschiedenen Minortypen an der Gesamtanzahl gefundener Minoren.

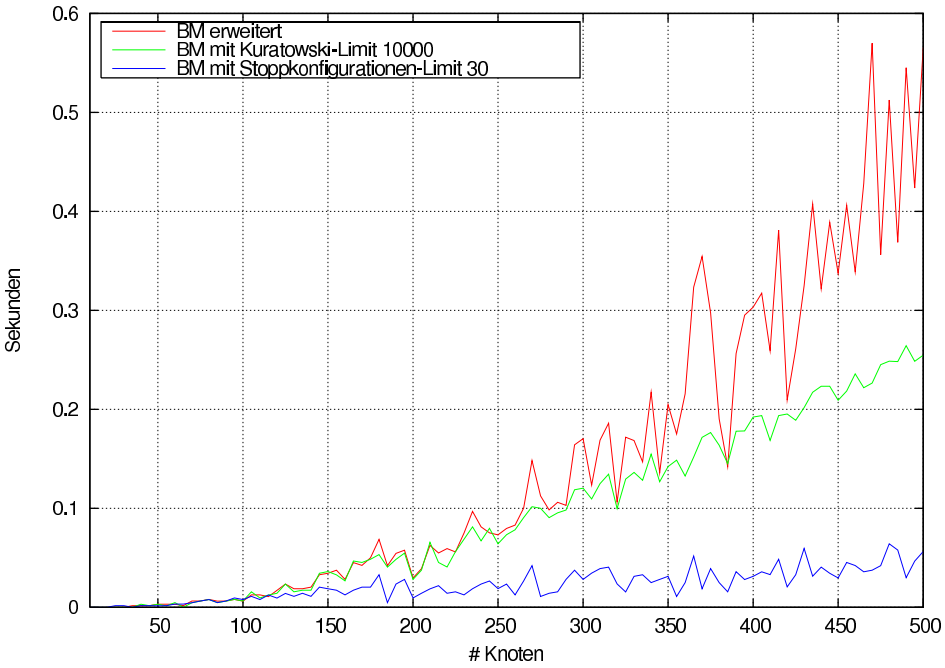


Abbildung 4.8: Zufallsgraphen: Die Laufzeiten der Testläufe des erweiterten Algorithmus mit und ohne Beschränkung der Anzahl der Kuratowski-Subdivisions bzw. Stoppkonfigurationen.

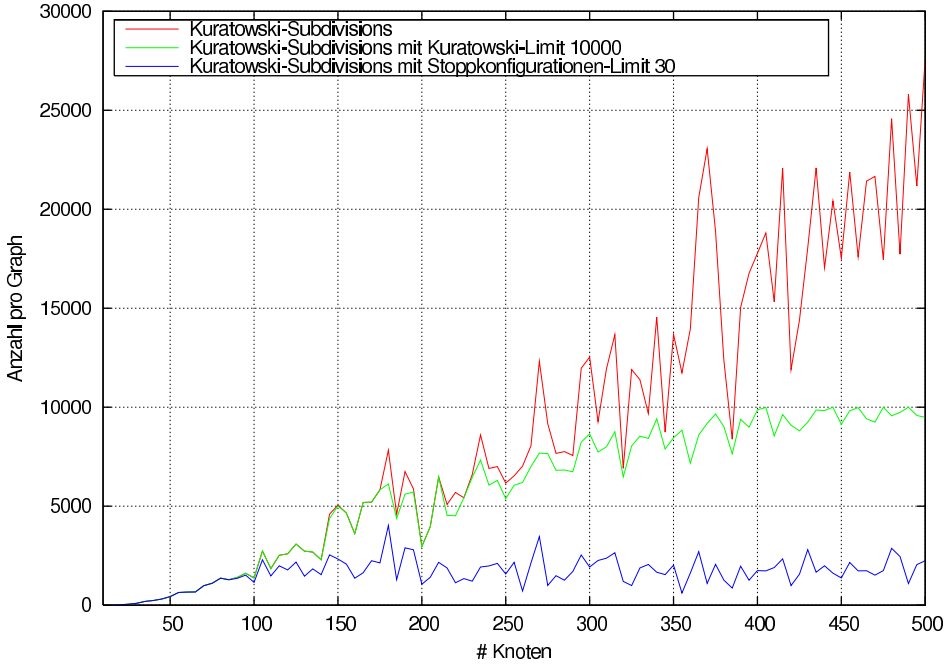


Abbildung 4.9: Zufallsgraphen: Die Anzahl der extrahierten Kuratowski-Subdivisions mit und ohne Beschränkung der Extraktionen.

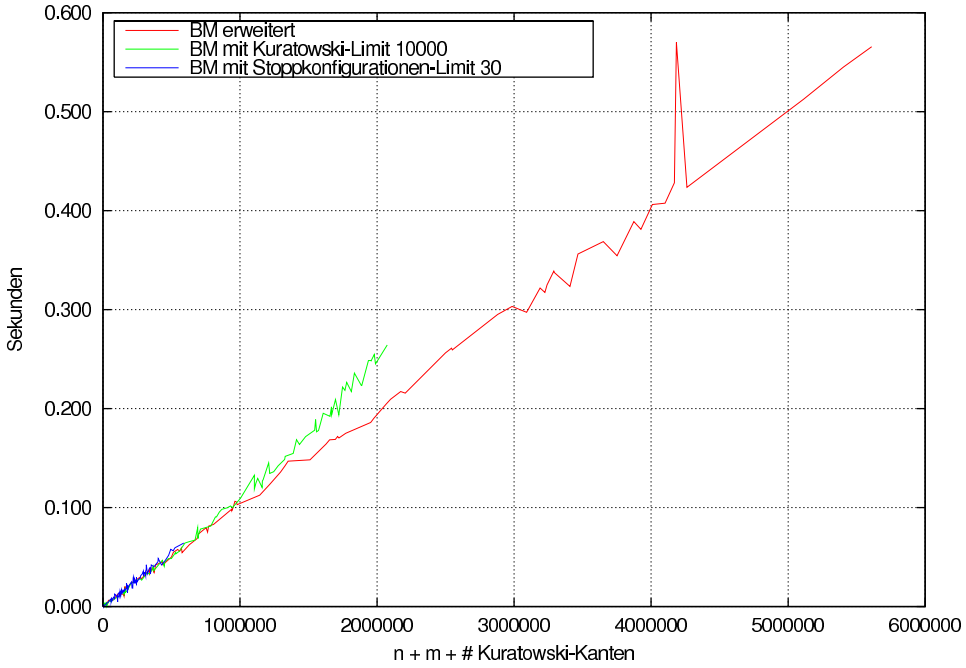


Abbildung 4.10: Zufallsgraphen: Vergleich der extrahierten Kuratowski-Subdivisions mit der jeweiligen Laufzeit aus den drei Testläufen.

5 Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde der erste Algorithmus vorgestellt, der mehr als konstant viele, unterschiedliche Kuratowski-Subdivisions in Linearzeit extrahieren kann. Dazu wurde der Boyer-Myrvold-Planaritätstest [BM04] sowohl lokal als auch global erweitert. Zudem wurde die sogenannte Bundle-Variante des Algorithmus beschrieben, welche zwar theoretisch keine Linearzeit mehr aufweist, in der Praxis aber weit mehr Kuratowski-Subdivisions extrahieren kann.

Der Algorithmus ist bei Eingabegraphen bis zu einer Größe von 500 Knoten und 1000 Kanten mit einer Laufzeit von unter einer Sekunde sehr schnell. Dadurch können die Schnittebenen von Branch-and-Cut-Algorithmen bei verschiedenen NP-schweren Graphenproblemen wesentlich effizienter gefunden werden. Die beiden Varianten des Algorithmus lassen sich auch auf größere Graphen durch eine Beschränkung der Kuratowski-Anzahl anwenden.

Allerdings wird bei nicht planaren Graphen vom Algorithmus keine untere Schranke (bis auf ≥ 1) für die Anzahl extrahierter Kuratowski-Subdivisions garantiert. Ein weiterer Kritikpunkt ist die hohe Komplexität des Algorithmus, die es notwendig macht, bei der Implementierung Vereinfachungen anzunehmen.

Es wäre interessant zu wissen, ob der Algorithmus so erweitert werden kann, dass er eine größere Mindestanzahl an extrahierten Kuratowski-Subdivisions garantiert. Offen bleibt auch die Frage, ob eine einfache Modifizierung des Algorithmus existiert, mit der aus der zurückgegebenen Einbettung ein Maximal Planar Subgraph erstellt werden kann. Für die Branch-and-Cut-Anwendungsgebiete wäre es zudem hilfreich, in einer experimentellen Analyse zu untersuchen, in wie weit diese von einer Kombination mit dem vorgestellten Algorithmus profitieren können.

Literaturverzeichnis

[AP61] L. Auslander and S. V. Parter. On imbedding graphs in the plane. *J. Math. and Mech.*, 10(3):517–523, 1961.

[BCPB03] J. M. Boyer, P. F. Cortese, M. Patrignani, and G. Di Battista. Stop minding your P's and Q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. In G. Liotta, editor, *Proceedings of the 11th International Conference on Graph Drawing*, volume 2912 of *Lecture Notes in Computer Science*, pages 25–36. Springer-Verlag, September 2003.

[BFN03] J. M. Boyer, C. G. Fern, and A. Noma. Correcting and implementing the PC-tree planarity algorithm, 2003.

[BGL+97] G. Di Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An experimental comparison of four graph drawing algorithms. *Comput. Geom. Theory Appl.*, 7(5-6):303–325, 1997.

[BL76] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using *PQ*-Tree algorithms. *J. Comp. Sys. Sci.*, 13:335–379, 1976.

[BM99] J. M. Boyer and W. Myrvold. Stop minding your P's and Q's: A simplified O(n) planar embedding algorithm. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 140–146, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[BM02] J. M. Boyer and W. Myrvold. Simplified $O(n)$ planarity algorithms, 2002.

[BM04] J. M. Boyer and W. J. Myrvold. On the cutting edge: Simplified $O(n)$ planarity by edge addition. *Journal of Graph Algorithms an Applications*, 8(3):241–273, 2004.

[CNAO85] N. Chiba, T. Nishizeki, S. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using *PQ*-Trees. *J. Comput. Syst. Sci.*, 30(1):54–76, 1985.

[Die05] R. Diestel. *Graph Theory.* Springer, third edition, 2005.

[DMP64] G. Demoucron, Y. Malgrange, and R. Pertuiset. Graphes planaires: Reconnaissance et construction de representations planaires topologiques. *Rev. Francaise Recherche Operationelle*, 8:33–47, 1964.

[EHK+96] T. Emden-Weinert, S. Hougardy, B. Kreuter, H. J. Prömel, and A. Steger. Einführung in Graphen und Algorithmen. Skriptum, 1996.

[EIT+92] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. R. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic planar

graph. *J. Algorithms*, 13(1):33–54, March 1992. Special issue for 1st SODA.

[ET76] S. Even and R. E. Tarjan. Computing an *st*-Numbering. *Theor. Comput. Sci.*, 2(3):339–344, 1976.

[Fár48] I. Fáry. On straight line representation of planar graphs. *Acta Univ. Szeged. Sect. Sci. Math.*, 11:229–233, 1948.

[FMR06] H. de Fraysseix, P. O. de Mendez, and P. Rosenstiehl. Trémaux Trees and planarity. *Int. J. Found. Comput. Sci.*, 17(5):1017–1030, 2006.

[FR85] H. de Fraysseix and P. Rosenstiehl. A characterization of planar graphs by Trémaux orders. *Combinatorica*, 5(2):127–135, 1985.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-Completeness*. W. H. Freeman, first edition, 1979.

[Gol63] A. J. Goldstein. An efficient and constructive algorithm for testing whether a graph can be embedded in a plane. Technical report, Technical Report Contract No. NONR 1858-(21), Department of Mathematics, Princeton University, 1963.

[Har69] F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, first edition, 1969.

[HLT01] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[HT74] J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.

[KP95] W. Kocay and C. Pantel. An algorithm for constructing a planar layout of a graph with a regular polygon as outer face. *Utilitas Mathematica*, 48:161–178, 1995.

[Kur30] K. Kuratowski. Sur le problème des corbes gauches en topologie. *Fundamenta Mathematicæ*, 15:271–283, 1930.

[LEC67] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Theory of Graphs: International Symposium*, pages 215–232, New York, 1967. Gordon and Breach.

[MM96] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16(2):233–242, 1996.

[Mut92] P. Mutzel. A fast $0(n)$ embedding algorithm, based on the Hopcroft-Tarjan planary test. Technical Report 107, Zentrum für Angewandte Informatik Köln, Lehrstuhl Jünger, 1992.

[OGD] OGDF - Open Graph Drawing Framework. University of Dortmund, Chair of Algorithm Engineering and Systems Analysis. Website under Construction.

[PD04] M. Pătraşcu and E. D. Demaine. Lower bounds for dynamic connectivity. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 546–553, New York, NY, USA, 2004. ACM Press.

[Rea87] R. C. Read. A new method for drawing a graph given the cyclic order of the

edges at each vertex. *Congressus Numerantium*, 56:31–44, 1987.

[RS84] N. Robertson and P. D. Seymour. Generalizing Kuratowski's theorem. *Congr. Numer.*, 45:129–138, 1984.

[SH93] W. K. Shih and W. L. Hsu. A simple test for planar graphs. *Proceedings of the International Workshop on Discrete Mathematics and Algorithms*, pages 110–122, 1993.

[SH99] W. K. Shih and W. L. Hsu. A new planarity test. *Theoretical Computer Science*, 223:179–191, 1999.

[Shi69] R. W. Shirey. *Implementation and analysis of efficient graph planarity testing alogithms.* PhD thesis, University of Wisconsin, 1969.

[Tör03] A. M. Törsel. An implementation of the Boyer-Myrvold algorithm for embedding planar graphs. University of Applied Sciences Stralsund, Germany. Diploma Thesis., 2003.

[Wag36] K. Wagner. Bemerkungen zum Vierfarbenproblem. *Jahresber. Deutsch. Math.-Verein.*, 46:26–32, 1936.

[Wag37] K. Wagner. Über eine Eigenschaft der ebenen Komplexe. *Mathematische Annalen*, 114(1):570–590, December 1937.

[Wil84] S. G. Williamson. Depth-first search and Kuratowski Subgraphs. *J. ACM*, 31(4):681–693, 1984.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel und Quellen verfasst habe. Alle Zitate habe ich dabei kenntlich gemacht.

Dortmund, 16. März 2007

Jens M. Schmidt