

Abschlussbericht Studienprojekt

Im Studiengang B.Sc. Data Science

Thema: Links-Rechts-Planaritätstest
Verfasserin: Lisa Tabea Beate Soboth
Bahnhofstraße 10, 35390 Gießen
Gutachter: Prof. Dr. Ralf Köhl
Datum: 28.03.2022

Inhalt

Abkürzungsverzeichnis	III
Abbildungsverzeichnis	IV
Projektumfang	1
Theoretischer Hintergrund	3
Algorithmus	8
Orientierungsphase	8
Testphase	11
Einbettungsphase	17
Implementierung	19
Ausführung des Programms	19
Programmübersicht	19
Orientierungsphase	20
Testphase	22
Einbettungsphase	25
Nutzeroberfläche	27
Visualisierung	34
Literaturrecherche	34
Algorithmus	35
Implementierung	36
Ausblick	41
Fazit	42
Literaturverzeichnis	V

Abkürzungsverzeichnis

DFS	<i>Depth-First Search</i>
LR-Partition	<i>Links-Rechts-Partition</i>
LR-Planaritätskriterium	<i>Links-Rechts-Planaritätskriterium</i>
LR-Planaritätstest	<i>Links-Rechts-Planaritätstest</i>
LR-Sortierung	<i>Links-Rechts-Sortierung</i>

Abbildungsverzeichnis

Abbildung 1: Unterschied planarer Graph / planare Zeichnung (Brandes, 2009, S. 4).....	4
Abbildung 2: DFS Orientation und LR-Partition (Brandes, 2009, S. 4)	5
Abbildung 3: fundamentaler Zyklus; basierend auf: (Brandes, 2009, S. 4)	5
Abbildung 4: LR-Sortierung, eigenes Beispiel	8
Abbildung 5: Beispielgraph für Pseudocode, eigenes Beispiel	8
Abbildung 6: Pseudocode – DFS1: Rückkanten, (Brandes, 2009, S. 17)	9
Abbildung 7: Pseudocode – DFS1: Verschachtelungstiefe und Tiefpunkte der Elternkanten, (Brandes, 2009, S. 17)	10
Abbildung 8: Pseudocode – DFS2: Rückkanten, (Brandes, 2009, S. 20)	12
Abbildung 9: Pseudocode - DFS2: Rückkehrkanten, (Brandes, 2009, S. 20)	12
Abbildung 10: Pseudocode - DFS2: add constraints 1, (Brandes, 2009, S. 22)	13
Abbildung 11: Pseudocode - DFS2: conflicting, (Brandes, 2009, S. 22)	15
Abbildung 12: Pseudocode - DFS2: add constraints 2, (Brandes, 2009, S. 22)	15
Abbildung 13: Pseudocode - DFS2: trim left interval, (Brandes, 2009, S. 24)	16
Abbildung 14: Pseudocode - DFS2: drop entire conflict pairs, (Brandes, 2009, S. 24)	16
Abbildung 15: Pseudocode - embedding phase: preparation, (Brandes, 2009, S. 15)	17
Abbildung 16: Pseudocode - embedding phase: sign, (Brandes, 2009, S. 15)	17
Abbildung 17: Pseudocode - DFS3: tree edge, (Brandes, 2009, S. 27)	18
Abbildung 18: Pseudocode - DFS3: back edges, (Brandes, 2009, S. 27)	18
Abbildung 19: Projektstruktur, eigenes Beispiel	20
Abbildung 20: NetworkX Graph, eigenes Beispiel	20
Abbildung 21: Orientierungsphase Struktur, eigenes Beispiel	21
Abbildung 22: Elternkanten Dictionary, eigenes Beispiel	21
Abbildung 23: Graph als Adjazenzliste, eigenes Beispiel	22
Abbildung 24: Code - DFS1: Kantenorientierung, eigenes Beispiel	22
Abbildung 25: Orientierungsphase Struktur, eigenes Beispiel	23
Abbildung 26: Code - DFS2: Vorbereitung, eigenes Beispiel	23
Abbildung 27: sortierte Adjazenzliste, eigenes Beispiel	23
Abbildung 28: Stackelement, eigenes Beispiel	24
Abbildung 29: Code - DFS2: Rückgabewert planar, eigenes Beispiel	25
Abbildung 30: Einbettungsphase Struktur, eigenes Beispiel	25
Abbildung 31: Code - DFS3: Baumkanten einfügen, eigenes Beispiel	26
Abbildung 32: Code - DFS3: Rückkanten einfügen, eigenes Beispiel	26
Abbildung 33: frame0 Grid-System, eigenes Beispiel	28
Abbildung 34: Code - App: b01/b02, eigenes Beispiel	29
Abbildung 35: Nutzeroberfläche - oberer Auswahlbereich, eigenes Beispiel	29
Abbildung 36: Nutzeroberfläche - mittlerer Arbeitsbereich: Eingabe der Knotenanzahl, eigenes Beispiel	29
Abbildung 37: Code - App: Platzhaltermatrix, eigenes Beispiel	30
Abbildung 38: Code - App: Klasse AdjMatrix, eigenes Beispiel	30

Abbildung 39: Code - App: Diagonalfelder, untere Dreiecksmatrix, eigenes Beispiel.....	31
Abbildung 40: Code - App: obere Dreiecksmatrix (callback-Funktion validieren), eigenes Beispiel	32
Abbildung 41: Nutzeroberfläche - mittlerer Arbeitsbereich: eingegebene Matrix, eigenes Beispiel.....	32
Abbildung 42: Code - App: Matrix im richtigen Format einlesen, eigenes Beispiel.....	33
Abbildung 43: Nutzeroberfläche - mittlerer Arbeitsbereich: eingelesene Matrix, eigenes Beispiel	33
Abbildung 44: Bézierkurve, (Bezier-Kurve, kein Datum)	36
Abbildung 45: Visualisierungsphase Struktur, eigenes Beispiel	37
Abbildung 46: Code - DFSTreeEdges: Koordinaten bestimmen, eigenes Beispiel.....	37
Abbildung 47: Code - BezierCurves: draw_bezier_curves, eigenes Beispiel..	38
Abbildung 48: Hilfs-Bézierpunkt für Rückkanten ohne relevante Blattknoten, eigenes Beispiel	39
Abbildung 49: Code - BezierCurves: Position der Knoten in determine_bezier_points bestimmen, eigenes Beispiel.....	40
Abbildung 50: Code - BezierCurves: Plotten der Bézierkurven, eigenes Beispiel.....	41
Abbildung 51: planare Zeichnung des im Kapitel "Algorithmus" besprochenen Graphen, eigenes Beispiel.....	41
Abbildung 52: nicht-planare Zeichnung eines planaren Graphen, eigenes Beispiel.....	42

Projektumfang

Das hier vorliegende Projekt beschäftigt sich mit dem Links-Rechts-Planaritätstest (LR-Planaritätstest) für Graphen. Dabei wird vor allem auf den Artikel „The Left-Right Planarity Test“ von Ulrik Brandes eingegangen (Brandes, 2009). Brandes erklärt den ursprünglich von de Fraysseix and Rosenstiehl vorgestellten und über Jahrzehnte mit weiteren Autoren weiterentwickelten Test in seinem Artikel anschaulich und stellt außerdem einen optimierten Pseudocode bereit (de Fraysseix & Rosenstiehl, 1982).

In der ersten Phase des Projekts wurde sich theoretisch damit auseinandergesetzt, was bestimmte Begriffe in der Graphentheorie bedeuten. Neben dem generellen Verständnis war hier wichtig, dass man die Begriffe auch voneinander abgrenzen konnte (z.B. Planarität, planare Einbettung, planare Zeichnung). Daraufhin wurde sich noch einmal die Depth-First Search (DFS) vor Augen geführt, da diese in unterschiedlichen Phasen des Tests zum Einsatz kommt. Zum Beispiel wird durch diese der Graph vorbereitet, also strukturiert (Aufteilung in Tiefensuchbaum und Rückkanten). Daraufhin konnte die Links-Rechts-Partition (LR-Partition) und daraus folgend das Links-Rechts-Planaritätskriterium (LR-Planaritätskriterium), sowie die Links-Rechts-Sortierung (LR-Sortierung) genauer betrachtet werden.

Diese Konzepte waren schwierig zu begreifen. Deshalb ist man dazu übergegangen den im Artikel angeführten Pseudocode beispielhaft mit Graphen durchzugehen. Durch absichtlich gewählte Grenzfälle von Graphen konnte man jeden Aspekt des Algorithmus nachvollziehen. In diesem Bericht wird ein Graph durchgegangen, der viele dieser Ausnahmefälle abdeckt. Das Durchgehen des Programms pro Graph dauerte sehr lange und war zusätzlich sehr fehleranfällig, sodass als logischer nächster Schritt die Implementierung folgte.

Als Programmiersprache wurde hier Python gewählt, da schon zuvor mit dieser gearbeitet wurde. Nachdem das Projekt aufgesetzt war,

wurde sich Gedanken darüber gemacht, wie man das Projekt strukturiert. Hier wurde vor allem Wert auf die Verschachtelung der Packages Wert gelegt. Der Algorithmus besteht aus drei Abschnitten, der Orientierungs-, der Test- und der Einbettungs-Phase. Diese sind die äußersten Ebenen. Innerhalb dieser wurde der Codefluss immer weiter aufgeteilt.

Als Datenstruktur für die Graphen wurde die Bibliothek Networkx benutzt. Dessen Knoten und Kanten mussten zuvor immer händisch in den Code eingetragen werden. Dies sollte mit einem User Interface umgangen werden, welches mit dem GUI-Toolkit Tkinter gestaltet wurde. Mit diesem kann ein Nutzer bzw. eine Nutzerin des Programms den gewünschten Graphen in Form einer Adjazenzmatrix eintragen und auf Planarität testen lassen. Eine zweite Möglichkeit ist, dass die Adjazenzmatrix aus einer CSV-Datei importiert wird. Als Ergebnis des Tests erscheint entweder „Your entered graph is planar.“ oder „Your entered graph is not planar.“. Da diese Ausgabe nicht besonders anschaulich ist, wurde sich im Anschluss mit der Visualisierung von planaren Graphen beschäftigt.

Diese Phase des Projekts war sehr zeitintensiv. Als erstes wurde eine Recherchearbeit geleistet, um sich einen Überblick zu verschaffen, welche Visualisierungsverfahren es für planare Graphen gibt. Es wurden viele verschiedene Algorithmen gefunden, aber keine die abbilden konnten, wie der Links-Rechts-Planaritätstest den Graphen umstrukturiert, um die Planarität aufzuzeigen. Nur eine Bachelorarbeit mit dem Titel „Implementation und Animation des Links-Rechts-Planaritätstests“ beschäftigte sich genau mit dem gleichen Problem (Kaiser, 2009). Hier wurde für die Implementierung und Animation die fzc++ Bibliothek LEDA verwendet. Da das hier vorgestellte Projekt in Python implementiert wurde, konnten die Visualisierung nur konzeptionell übernommen werden. Der Tiefensuchbaum wurde nach dem gleichen Prinzip keilartig auf Kreisen angeordnet. Hierfür wurde den Knoten des Networkx Graphen Koordinaten zugewiesen. Die Rückkanten wurden als Bézierkurven gezeichnet. Hierfür wurde die

Python Bibliothek `bezier` benutzt. Diese beiden Komponenten des Graphen mussten dann noch zusammengeführt werden und in einem Ausgabefeld angezeigt werden. Sowohl dem `Networkx` Graph als auch den Beziérkurven konnte ein `Axes` Objekt der `Plotting` Bibliothek `Matplotlib` übergeben werden. Nun musste abschließend dem User Interface eine Option hinzugefügt werden, dass dem Nutzer diese planare Zeichnung des Graphen ausgegeben wird. Mit dem Einbauen dieser Funktion in das Programm wurde das Projekt abgeschlossen. Im Folgenden werden die Ergebnisse der hier kurz angerissenen Phasen des Projekts präsentiert.

Theoretischer Hintergrund

In diesem Abschnitt sollen einige graphentheoretische Grundbegriffe geklärt werden, die für das Verständnis des LR-Planaritätstest grundlegend sind. Außerdem sollen die Begriffe LR-Partition, LR-Planaritätskriterium und LR-Sortierung definiert werden.

Den LR-Planaritätstest kann man für einfache Graphen durchführen. Ein einfacher Graph ist ein ungewichteter, ungerichteter Graph, der keine Schleifen und keine Mehrfachkanten besitzt (Weisstein, o. D.). All diese Eigenschaften haben keine Auswirkungen auf die Planarität, sodass man mit dem bereinigten Graphen arbeitet. Solch ein Graph kann mit einer Adjazenzmatrix eindeutig dargestellt werden. Dabei kennzeichnet eine 1 eine Verbindung und eine 0 keine Verbindung zwischen zwei Knoten.

Ein einfacher Graph ist planar, wenn er eine planare Zeichnung besitzt. Das bedeutet, er kann in die Ebene gezeichnet werden, ohne dass sich Kanten überschneiden. Die Knoten und Kanten dürfen dafür beliebig im Raum angeordnet werden, Abstände spielen keine Rolle. Die Kanten dürfen außerdem in Kurven gezeichnet werden. Zu einem planaren Graphen kann es viele planare Zeichnungen geben. Eine planare Zeichnung ist also eine Realisation eines planaren Graphen. Eine planare Einbettung eines Graphens gibt an, wie die Knoten und Kanten

zueinanderstehen müssen, damit der Graph planar gezeichnet werden kann. Dies wird zum Beispiel realisiert, indem man die zyklische Reihenfolge der eingehenden Kanten für jeden Knoten angibt (Brandes, 2009, S. 3).

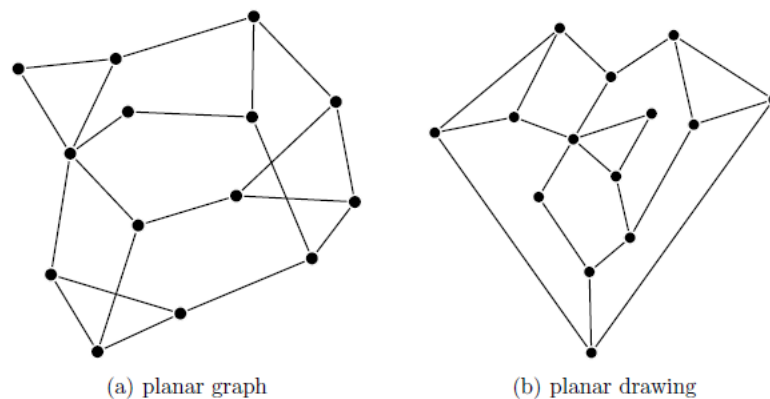


Abbildung 1: Unterschied planarer Graph / planare Zeichnung (Brandes, 2009, S. 4)

In der Abbildung sieht man zweimal den gleichen planaren Graphen: einmal planar gezeichnet (b), einmal nicht planar gezeichnet (a).

Der erste Schritt des LR-Planaritätstests besteht darin eine DFS beim Graphen durchzuführen, um eine DFS-Orientierung desselben zu bekommen. Die DFS funktioniert kurz gesagt so, dass man ausgehend vom Startknoten einem Pfad (Kantenzug) in die Tiefe folgt, bis es keine unbesuchten Knoten mehr gibt. Von dort wird der Pfad so lange zurückgegangen, bis es wieder unbesuchte Knoten gibt, die exploriert werden können. Dies geschieht bis alle Knoten der Zusammenhangskomponente besucht wurden (White & Ray, 2021, S. 199). Die Kanten werden dabei entsprechend des entlanggegangenen Kantenzugs ausgerichtet, sodass ein gerichteter Graph, der Tiefensuchbaum, entsteht. Der Ursprungsgraph kann allerdings noch Kanten besitzen, die einen Knoten mit einem schon zuvor besuchten Knoten verbinden würde. Diese Kanten bezeichnet man Rückkanten. Weil Rückkanten meistens kurvenförmig gezeichnet werden, kann man sie in zwei Gruppen aufteilen. Entweder sie verlaufen mit dem Uhrzeigersinn oder gegen den Uhrzeigersinn. Erstere nennt man

Rechts- und letztere Linkskanten (Brandes, 2009, S. 5). In der folgenden Abbildung sieht man neben der DFS-Orientierung die entstandene LR-Partition. Diese ist in planaren Zeichnungen gut zu erkennen.

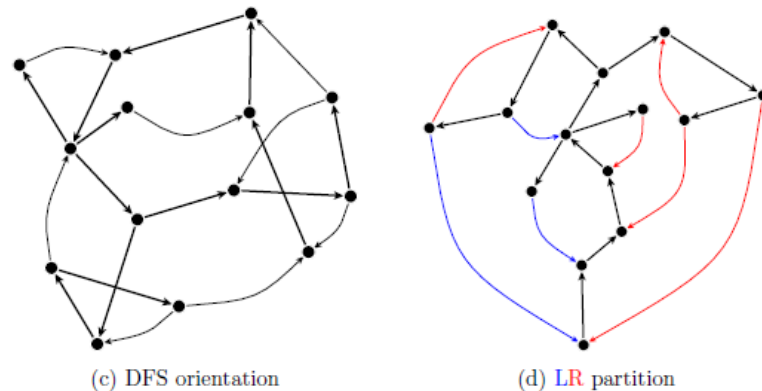


Abbildung 2: DFS Orientation und LR-Partition (Brandes, 2009, S. 4)

Zu jeder Rückkante gehört ein fundamentaler Zyklus. Dieser beginnt beim Endpunkt der Rückkante, läuft den Kantenzug des Tiefensuchbaums bis zum Startpunkt der Rückkante und schließlich die Rückkante entlang. In Abbildung 3 sieht man diesen beispielhaft für die Rückkante $(6 \rightarrow 2)$ gelb eingezeichnet. Ein zur einer Rückkante gehörender Zyklus hat die gleiche Orientierung wie die Rückkante. In diesem Fall also auch gegen den Uhrzeigersinn.

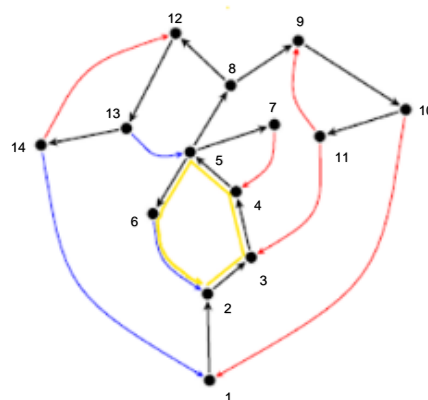


Abbildung 3: fundamentaler Zyklus, basierend auf: (Brandes, 2009, S. 4)

Zwei fundamentale Zyklen heißen überlappend, wenn sie sich eine Kante teilen (Brandes, 2009, S. 5). Eine Gabelung beschreibt den Bereich, an dem sich diese Zyklen aufspalten. Hier in der Abbildung

wäre das zum Beispiel die Kante $(5 \rightarrow 8)$ (allgemein: $(u \rightarrow v)$) zusammen mit den Kanten $(8 \rightarrow 9)$ und $(8 \rightarrow 12)$ (allgemein: e_1 und e_2). Der Knoten 8 ist hier der Verzweigungspunkt. (Brandes, 2009, S. 6).

Zwei überlappende Zyklen sind ineinander verschachtelt, wenn sie in einer planaren Zeichnung zur gleichen Seite orientiert sind. Die Verschachtelungsreihenfolge wird hierbei durch die Tiefpunkte der Zyklen induziert. Mit Tiefpunkt einer Rückkante ist hier der tiefste Knoten des Zyklus gemeint, also der Endpunkt der Rückkante. Der Tiefpunkt einer Kante des Tiefensuchbaums ist der tiefste Knoten von allen Zyklen der Kante, wobei diese tiefer als der Startknoten der Kante sein muss (Brandes, 2009, S. 7). Diese Beobachtungen führen zu den folgenden Definitionen:

- 1) LR-Partition: für jede Gabelung eines Graphen mit DFS-Orientierung müssen
 - a) alle Rückkanten von e_1 , die strikt höher als der Tiefpunkt von e_2 liegen, zur einer Orientierungsklasse (Links oder Rechts) und
 - b) alle Rückkanten von e_2 , die strikt höher als der Tiefpunkt von e_1 liegen, zur der anderen Orientierungsklasse gehören.
- 2) LR-Planaritätskriterium: ein Graph ist genau dann planar, wenn er eine LR-Partition zulässt (Brandes, 2009, S. 9).

Um nun eine planare Einbettung des Graphen angeben zu können, muss man zum einen die LR-Partition auf die Kanten des Tiefensuchbaums ausweiten und zum anderen für jeden Knoten eine Reihenfolge der ausgehenden Kanten angeben. Ersteres erreicht man, indem man für jede Baumkante, alle zugehörigen Rückkehrkanten betrachtet. Das sind alle Rückkanten, bei denen die Baumkante Teil des fundamentalen Zyklus ist. Man wählt nun die Rückkante, die am höchsten endet und ordnet die Baumkante der gleichen Orientierungsklasse zu. Zweiteres ergibt sich, wenn man ausgehende Kanten, die zu rechts orientierten Zyklen gehören, von außen nach

innen durchnummeriert. Das bedeutet heruntergebrochen, dass der Zyklus der tiefer endet, vor dem Zyklus kommen muss, der höher endet. Somit ist garantiert, dass die Zyklen richtig ineinander verschachtelt sind, ohne dass sie sich schneiden. Für linksorientierte Zyklen gilt das verkehrt herum (Brandes, 2009, S. 10). Dies führt zu der folgenden Definition:

3) LR-Sortierung: zu einer gegebenen LR-Partition sind die Kanten für einen Knoten v folgendermaßen im Uhrzeigersinn angeordnet:

1. (u,v) : u ist Elternknoten von v ,
2. $L(e_1^L), e_1^L, R(e_1^L), \dots, L(e_{l^L}^L), e_{l^L}^L, R(e_{l^L}^L)$:
 e_1^L bis $e_{l^L}^L$ sind die von v ausgehenden linksorientierten ineinander verschachtelten Kanten,
3. $L(e_1^R), e_1^R, R(e_1^R), \dots, L(e_{r^R}^R), e_{r^R}^R, R(e_{r^R}^R)$:
 e_1^R bis $e_{r^R}^R$ sind die von v ausgehenden rechtsorientierten ineinander verschachtelten Kanten,

wobei $L(e)$ und $R(e)$ für die links- und rechtseingehenden Rückkanten stehen, dessen Zyklen die Kante e enthalten. Nun gilt für die rechtsorientierten Rückkanten $b_1:(y_1 \rightarrow v)$ und $b_2:(y_2 \rightarrow v)$ der Gabelung $(x \rightarrow y_1), (x \rightarrow y_2)$, dass b_1 nach b_2 in $R(e)$ von v vorkommt, wenn der Zyklus b_2 innerhalb von b_1 liegt. Für linksorientierte Rückkanten gilt das entsprechend andersherum (Brandes, 2009, S. 11).

Dieses abstrakte Konzept lässt sich am besten mit einem Beispiel veranschaulichen. In der nachkommenden Abbildung soll der Knoten 1 v sein. Die LR-Sortierung ist hier: $(0 \rightarrow 1), (1 \rightarrow 2), (5 \rightarrow 1), (4 \rightarrow 1)$, wobei die Rückkante $(4 \rightarrow 1)$ b_1 und die Rückkante $(5 \rightarrow 1)$ b_2 sein soll. Diese Anordnung kann man in der Abbildung nachempfinden, da das die ein- und ausgehenden Kanten um den Knoten 1 sind.

Wie man sieht, kommt b_1 in der Reihenfolge der rechtsorientierten eingehenden Kanten nach b_2 .

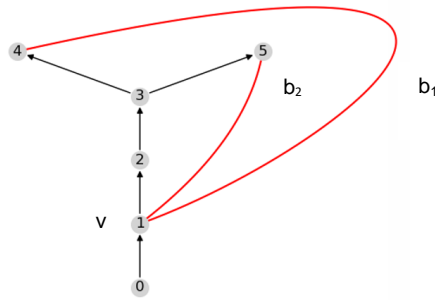


Abbildung 4: LR-Sortierung, eigenes Beispiel

Die Erweiterung der LR-Partition um die LR-Sortierung ergibt immer eine planare Einbettung (Brandes, 2009, S. 11).

Nun hat man alle Begriffe und Definitionen geklärt, um zu verstehen, wie der Algorithmus des LR-Planaritätstest funktioniert.

Algorithmus

In diesem Kapitel wird der in dem Artikel „The Left-Right Planarity Test“ vorgestellte Algorithmus anhand eines Beispiels durchlaufen.

Der ausgesuchte Graph $G = (V, E)$ lautet: $V = \{0, 1, 2, 3, 4, 5\}$,
 $E = \{(0,1), (0,4), (0,5), (1,2), (1,4), (2,3), (2,5), (3,4), (4,5)\}$.

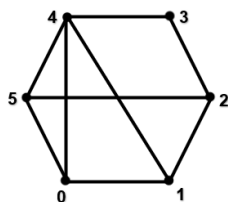


Abbildung 5: Beispielgraph für Pseudocode, eigenes Beispiel

Als erstes wird überprüft, ob $|E| > 3|V| - 6$ gilt. $|E|$ beträgt 6, $|V|$ beträgt 9, also stimmt die Ungleichung $6 > 3 \cdot 9 - 6$ nicht. Wenn sie gelten würde, könnte man schon an diesem Punkt sagen, dass der Graph nicht planar ist.

Orientierungsphase

Als nächstes muss der Graph mithilfe der DFS in einen gerichteten Graphen umgeformt werden. Außerdem will man die Tiefpunkte der

Kanten und die Verschachtelungstiefe bestimmen. Das ist die Orientierungsphase (Brandes, 2009, S. 16).

Zusätzliche Informationen, die man hier braucht, sind die Höhe und die Elternkanten der Knoten. Mit Elternkante ist die Kante gemeint, die vom Elternknoten zum Knoten verläuft. Die Höhe der Knoten wird initial bei allen auf unendlich gesetzt, die Elternkanten auf None (noch nicht belegt). Man merkt sich außerdem für jede Kante zwei Tiefpunkte. Der zweite Tiefpunkt ist nur dafür da, um anzugeben, ob die Kante mehr als einen Rückkehrpunkt hat („chordal“ ist) (Brandes, 2009, S. 11). Hierbei kann es sich nur um Baumkanten handeln.

Die Höhe und die Elternkanten von Knoten, welche noch nicht besucht wurden, werden rekursiv bei der DFS gesetzt. Zusätzlich werden für alle Kanten die Tiefpunkte auf die Höhe des Startknotens der Kante gesetzt. Wenn man auf Kanten stößt, die noch nicht ausgerichtet wurden, aber deren Zielknoten schon besucht wurde (deren Höhe nicht mehr unendlich beträgt), handelt es sich um Rückkanten.

```
else /* back edge */  
     $\lfloor lowpt[(v, w)] \leftarrow height[w]$ 
```

Abbildung 6: Pseudocode – DFS1: Rückkanten, (Brandes, 2009, S. 17)

Die erste Rückkante, auf die man bei dem gewählten Graphen stößt, ist die Kante (5→0). Zuvor wurden die Kanten folgendermaßen ausgerichtet: (0→1→2→3→4→5). Für diese Rückkante wird, wie man im obigen Pseudocode sieht, der Tiefpunkt überschrieben. Anstatt der Höhe des Startknotens, wird die Höhe des Zielknotens gewählt. Das ist hier die Höhe des Knotens 0. Für jede Kante (Baum- oder Rückkante) wird die Verschachtelungstiefe gesetzt und überprüft, ob man ihren zweiten Tiefpunkt auch überschreiben muss. Ebenso werden die Tiefpunkte der Elternkante jedes Mal aktualisiert.

```

▼ determine nesting depth
|  $nesting\_depth[(v, w)] \leftarrow 2 \cdot lowpt[(v, w)]$ 
| if  $lowpt2[(v, w)] < height[v]$  then /* chordal */
|   |  $nesting\_depth[(v, w)] \leftarrow nesting\_depth[(v, w)] + 1$ 
|
▼ update lowpoints of parent edge e
| if  $e \neq \perp$  then
|   | if  $lowpt[(v, w)] < lowpt[e]$  then
|     |  $lowpt2[e] \leftarrow \min\{lowpt[e], lowpt2[(v, w)]\}$ 
|     |  $lowpt[e] \leftarrow lowpt[(v, w)]$ 
|   | else if  $lowpt[(v, w)] > lowpt[e]$  then
|     |  $lowpt2[e] \leftarrow \min\{lowpt2[e], lowpt[(v, w)]\}$ 
|   | else
|     |  $lowpt2[e] \leftarrow \min\{lowpt2[e], lowpt2[(v, w)]\}$ 
|

```

Abbildung 7: Pseudocode – DFS1: Verschachtelungstiefe und Tiefpunkte der Elternkanten, (Brandes, 2009, S. 17)

Die Verschachtelungstiefe einer Kante wird auf das Zweifache ihres Tiefpunktes gesetzt, sodass für Rückkanten die höher enden, die Verschachtelungstiefe höher ist als für welche die tiefer enden. Das ergibt Sinn nach dem Prinzip der LR-Sortierung.

Für die betrachtete Kante ($5 \rightarrow 0$) wäre das $2 \cdot 0 = 0$. Die Kante ist nicht chordal. Der erste Tiefpunkt für die Elternkante ($4 \rightarrow 5$) wird aktualisiert. Er wird auf 0 gesetzt, weil auch diese Kante jetzt durch die Rückkante ($5 \rightarrow 0$) zu dem Knoten 0 mit der Höhe 0 zurückkehrt. Nun betrachtet man die zweite Rückkante ($5 \rightarrow 2$). Die Verschachtelungstiefe beträgt $2 \cdot 2 = 4$, auch diese Kante ist nicht chordal. Der zweite Tiefpunkt der Elternkante ($4 \rightarrow 5$) wird auf den Tiefpunkt der Rückkante ($5 \rightarrow 2$) gesetzt. Somit beträgt dieser jetzt 2. Die Rekursion für den Knoten 5 ist nun durchlaufen, sodass man auf die Rekursionsebene der Kante ($4 \rightarrow 5$) zurückkehrt.

Für diese Baumkante wird jetzt auch die Verschachtelungstiefe bestimmt. Diese beträgt $2 \cdot 0 = 0$. Diese Kante ist chordal. Dies wird überprüft, indem man den zweiten Tiefpunkt mit der Höhe des Startknotens vergleicht, welche als Default-Wert zu Beginn gesetzt wurde. Für chordale Kanten wird die Verschachtelungstiefe um eins erhöht. Sie beträgt also jetzt $0 + 1 = 1$. Jetzt werden auch die Tiefpunkte der Elternkante ($3 \rightarrow 4$) aktualisiert. Als ersten werden die Tiefpunkte der

Kante (4→5) übernommen, da die Kante (3→4) auch Teil des fundamentalen Zyklus von (4→5) ist. Als nächstes kommen die Rückkanten (4→0) und (4→1). (4→0) verändert die Tiefpunkte der Elternkante nicht, da die Kante keinen höheren Rückkehrpunkt besitzt. Da (4→1) auf der Höhe 1 endet, wird der zweite Tiefpunkt der Elternkante auf diese Höhe gesetzt. Nun ist auch die Rekursion für den Knoten 4 beendet. Man kehrt zu (3→4) zurück. Nach diesem Schema bestimmt man auch die Verschachtelungstiefen und Tiefpunkte der anderen Kanten.

Testphase

Diese beiden Angaben über den Graphen braucht man für die zweite Phase des LR-Planaritätstest. Das Ziel der zweiten Phase ist eine konsistente LR-Partition zu ermitteln. Konsistent bedeutet, dass alle Rückkanten einer Baumkante, die an ihrem Tiefpunkt enden, derselben Orientierung angehören (Brandes, 2009, S. 10). Die Herausforderung dieser Phase ist, dass sich vor allem mit sogenannten paarweisen Einschränkungen auseinandergesetzt werden muss: entweder sind zwei Rückkanten dazu gezwungen auf derselben (gleiche Bedingung) oder auf unterschiedlichen Seiten (unterschiedliche Bedingung) zu sein (Brandes, 2009, S. 17).

Man erstellt zunächst für jeden Knoten eine Liste von Knoten, zu denen eine gerichtete Kante besteht. Diese Liste nennt man Adjazenzliste.

Diese Knoten sind nach der Verschachtelungstiefe sortiert.

Entsprechend dieser Sortierung wird ein zweites Mal eine DFS beim Graphen durchgeführt. Man braucht außerdem zwei Angaben zu den Kanten, namens side und ref. Für jede Kante wird als ref eine Referenzkante angegeben und als side eine 1 bzw. -1, um festzulegen, ob die Kante zur gleichen Seite wie ihre Referenzkante orientiert ist oder nicht (Brandes, 2009, S. 15). Zusätzlich arbeitet man mit einem Stack, welcher Rückkanten die paarweisen Einschränkungen unterliegen, temporär speichert. Ein Stackelement besteht aus einem linken und einem rechten Intervall, welche wiederum zwei Kanten

beinhalten. Die erste Kante hat den niedrigsten und die zweite Kante den höchsten Tiefpunkt des Intervalls. Zu einem Intervall gehören alle Kanten mit derselben Referenzkante (Brandes, 2009, S. 19).

Außerdem muss man wissen, an welcher Stelle des Stacks, die Rückkanten für die momentan betrachtete Kante liegen (`stack_bottom`). Eine weitere Hilfsvariable speichert die erste Rückkante einer Kante, die zu ihrem Tiefpunkt führt (`lowpt_edge`). Diese braucht man, um sie als Referenz für andere Rückkanten zu verwenden.

Die Adjazenzlisten der Knoten lautet: 0: [1], 1: [2], 2: [3], 3: [4]. 4: [0, 5, 1], 5: [0, 2]. Man geht die DFS rekursiv bis zum Knoten 4 durch. Dort trifft man auf die erste Rückkante (4→0). Da der Stack noch leer ist, wird für den `stack_bottom` None eingetragen.

```
else /* back edge */
  | lowpt_edge[ei] ← ei;  push (∅, [ei, ei]) → S
```

Abbildung 8: Pseudocode – DFS2: Rückkanten, (Brandes, 2009, S. 20)

Wie man in der Abbildung sieht, sind Rückkanten ihre eigene Tiefpunktkanten, also in diesem Fall (4→0). Außerdem legt man die Rückkante als Intervall [(4→0), (4→0)] auf die rechte Seite des Stacks. Da diese Schritte bei jeder Rückkante geschehen, wird das folgend nicht mehr erwähnt.

```
if lowpt[ei] < height[v] then /* ei has return edge */
  | if ei = e1 then
  |   | lowpt_edge[e] ← lowpt_edge[e1]
  | else
  |   | ► add constraints of ei (Algorithm 4)
```

Abbildung 9: Pseudocode - DFS2: Rückkehrkanten, (Brandes, 2009, S. 20)

Da (4→0) zusätzlich die erste Rückkante (e_1) von (3→4) ist, wird diese auch für (3→4) als Tiefpunktkante eingetragen. Die erste Rückkante ist immer die Rückkante, die zum Tiefpunkt der Baumkante führt. Das ergibt sich aus der Verschachtelungstiefe der ersten DFS. Auch dieser Eintrag für die Tiefpunktkanten der Elternkanten wird ab jetzt als bekannt vorausgesetzt.

Nun kommt man zu der nächsten Kante ($4 \rightarrow 5$) (e_2). Da diese eine Baumkante ist, ruft man für den Knoten 5 erneut DFS2 auf. Die erste Kante (e_1) ist die Rückkante ($5 \rightarrow 0$). Diese wird analog wie oben beschrieben behandelt.

Für die nächste Kante ($5 \rightarrow 2$) (e_2) müssen stattdessen Bedingungen festgelegt werden.

```

P ← (∅, ∅)
▼ merge return edges of  $e_i$  into P.R
repeat
  Q ← pop(S)
  if Q.L ≠ ∅ then swap Q.L, Q.R
  if Q.L ≠ ∅ then
    | HALT: not planar
  else
    if lowpt[Q.R.low] > lowpt[e] then /* merge intervals */
      if P.R = ∅ then P.R.high ← Q.R.high
      else ref[P.R.low] ← Q.R.high
      P.R.low ← Q.R.low
    else /* make consistent */
      | ref[Q.R.low] ← lowpt_edge[e]
until top(S) = stack_bottom[ $e_i$ ]

```

Abbildung 10: Pseudocode - DFS2: add constraints 1, (Brandes, 2009, S. 22)

Es wird mit einer Hilfsvariablen P gearbeitet, welche ein Konfliktpaar von überlappenden Intervallen darstellen soll (Brandes, 2009, S. 14). Oberhalb des stack_bottoms liegen auf dem Stack jetzt alle Rückkehrkanten der momentan betrachtenden Kante (e_i). Diese möchte man zu einem Intervall zusammenfassen (Brandes, 2009, S. 21). Man nimmt nacheinander alle Elemente vom Stack bis man den stack_bottom erreicht hat.

Hier liegt nur die Rückkante ($5 \rightarrow 2$) als rechtes Intervall selbst auf dem Stack. Das linke Intervall ist leer. Wenn an dieser Stelle beide Intervalle belegt wären, also durch den Tausch der Intervalle kein leeres linkes Intervall entstehen würde, dann wäre der Graph nicht planar. Der Versuch diese zwei Intervalle zu einem zusammen zu führen, würde eine zuvor bestimmte Bedingung verletzen (Brandes, 2009, S. 21). ($5 \rightarrow 2$) hat einen höheren Tiefpunkt als die Elternkante ($4 \rightarrow 5$), sodass sie dem Konfliktpaar als rechtes Intervall hinzugefügt wird. Sie ist für spätere Bedingungen noch von Interesse und muss weiterhin

berücksichtigt werden. Da die Rückkehrkanten von (e_1) die unterhalb (inklusive) des `stack_bottoms` liegen, nicht höher als die momentan betrachtete Kante enden, gibt es zwischen ihnen keine aufzulösenden Konflikte. Jetzt kann abschließend das erzeugte Konfliktpaar P auf dem Stack gelegt werden.

Man ist nun alle ausgehenden Kanten des Knoten 5 durchgegangen. Nun muss man noch alle Rückkanten vom Stack nehmen, die beim Elternknoten von 5, also beim Knoten 4, enden. Diese braucht man nicht mehr, da sie weder für den Knoten 4 noch für tiefer liegende Knoten als Rückkehrkante zum Tiefpunkt dienen können (Brandes, 2009, S. 24). Solche Rückkanten liegen nicht auf dem Stack, sodass nichts gemacht werden muss. Abschließend muss der Elternkante $(4 \rightarrow 5)$ noch eine Seite zugewiesen werden. Man nimmt als Referenzkante die Rückkante, die am höchsten endet, also $(5 \rightarrow 2)$.

Jetzt kehrt man zu der Rekursionsebene von $(4 \rightarrow 5)$ (e_2) zurück. Auch für diese Kante müssen Bedingungen ausgewertet werden. Als erstes nimmt man wieder $(\emptyset, [(5 \rightarrow 2), (5 \rightarrow 2)])$ vom Stack. Der Tiefpunkt von $(5 \rightarrow 2)$ ist ebenfalls höher als der von $(3 \rightarrow 4)$. Also wird diese Kante wieder dem Konfliktpaar als rechtes Intervall hinzugefügt. Das nächste Element ist $(\emptyset, [(5 \rightarrow 0), (5 \rightarrow 0)])$. Da hier der Tiefpunkt nicht höher ist, wird für die Kante $(5 \rightarrow 0)$ als Referenzkante die Tiefpunktkante der Elternkante eingetragen. Somit stellt man sicher, dass man eine konsistente LR-Partition erzeugt. Jetzt ist man am `stack_bottom` angelangt. Da es mit den Rückkehrkanten von e_1 zu keinen Konflikten kommt, wird abschließend das Konfliktpaar P auf den Stack gelegt.

Der Knoten 4 hat noch eine dritte ausgehende Kante $(4 \rightarrow 1)$ (e_3). Wie zuvor schon wird $(\emptyset, [(4 \rightarrow 1), (4 \rightarrow 1)])$ als rechtes Intervall des Konfliktpaares P eingetragen. Nun ist man schon am `stack_bottom` angekommen. Allerdings kommt es hier mit Rückkehrkanten der vorherigen Kanten e_1 und e_2 zu Konflikten.

```
where
boolean conflicting(interval  $I$ , edge  $b$ )
└ return ( $I \neq \emptyset$  and  $lowpt[I.high] > lowpt[b]$ )
```

Die Funktion `conflicting` wird mit den unterhalb (inklusive) des `stack_bottoms` liegenden Intervallen aufgerufen. Wenn der Tiefpunkt der höchsten Kante des Intervalls höher ist als der Tiefpunkt der momentan betrachteten Kante, stehen die Kanten im Konflikt zueinander, da sie sich schneiden würden, wären sie zur selben Seite orientiert.

```

▼ merge conflicting return edges of  $e_1, \dots, e_{i-1}$  into  $P.L$ 
  while  $\text{conflicting}(\text{top}(S).L, e_i)$  or  $\text{conflicting}(\text{top}(S).R, e_i)$  do
     $Q \leftarrow \text{pop}(S)$ 
    if  $\text{conflicting}(Q.R, e_i)$  then swap  $Q.L, Q.R$ 
    if  $\text{conflicting}(Q.R, e_i)$  then
      | HALT: not planar
    else /* merge interval below  $\text{lowpt}(e_i)$  into  $P.R$  */
      |  $\text{ref}[P.R.\text{low}] \leftarrow Q.R.\text{high}$ 
      | if  $Q.R.\text{low} \neq \perp$  then  $P.R.\text{low} \leftarrow Q.R.\text{low}$ 
      if  $P.L = \emptyset$  then  $P.L.\text{high} \leftarrow Q.L.\text{high}$ 
      else  $\text{ref}[P.L.\text{low}] \leftarrow Q.L.\text{high}$ 
       $P.L.\text{low} \leftarrow Q.L.\text{low}$ 

```

Oben auf dem Stack liegt an dieser Stelle $(\emptyset, [(5 \rightarrow 2), (5 \rightarrow 2)])$. Der Tiefpunkt von $(5 \rightarrow 2)$ ist höher als der von $(4 \rightarrow 1)$, sodass man das Element vom Stack nimmt, einer Variablen Q zuweist und das Rechte mit dem linken Intervall tauscht. Wenn das nun rechte Intervall auch mit der Kante kollidieren würde, wäre der Graph nicht planar (Brandes, 2009, S. 24). In diesem Beispiel ist das rechte Intervall leer, sodass die Intervalle nicht kollidieren können. Wenn dieses Intervall nicht leer wäre, müsste man dieses in die rechte Seite des Konfliktpaares P integrieren, also die Orientierung und die Intervallgrenzen anpassen. Jetzt wird die linke Seite des Konfliktpaares P mit der linken Seite von Q überschrieben: $([(5 \rightarrow 2), (5 \rightarrow 2)], [(4 \rightarrow 1), (4 \rightarrow 1)])$. Jetzt liegt noch $(\emptyset, [(4 \rightarrow 0), (4 \rightarrow 0)])$ auf dem Stack. Weil der Tiefpunkt von $4 \rightarrow 0$ nicht höher liegt als der Tiefpunkt von $(4 \rightarrow 1)$, geht man nicht erneut in die Schleife. Im letzten Schritt wird P wieder auf den Stack gelegt.

Man ist nun alle ausgehenden Kanten des Knotens 4 durchgegangen, sodass jetzt theoretisch alle Rückkanten vom Stack genommen werden müssten, die beim Elternknoten 3 enden. Allerdings gibt es keine. Der

Elternkante (3→4) wird schließlich als Referenz die Kante (5→2) zugewiesen.

Jetzt kehrt man zu den Rekursionsebenen der Knoten 3, 2 und 1 zurück. Bei dem Elternknoten 2 des Knotens 3 muss die Rückkante (5→2) entfernt werden. Man nimmt das Konfliktpaar $P([(5→2), (5→2)], [(4→1), (4→1)])$ vom Stack. Jetzt weist man der höchsten Kante des linken Intervalls so lange dessen eigene Referenzkanten zu, bis es keinen Referenzeintrag mehr gibt und der Eintrag None wird. Man beginnt hier mit der Kante (5→2).

```

 $\dot{P} \leftarrow \text{pop}(S)$ 
▼ trim left interval
  while  $P.L.high \neq \perp$  and  $\text{target}(P.L.high) = u$  do
     $P.L.high \leftarrow \text{ref}[P.L.high]$ 
  if  $P.L.high = \perp$  and  $P.L.low \neq \perp$  then /* just emptied */
     $\text{ref}[P.L.low] \leftarrow P.R.low$ 
     $\text{side}[P.L.low] \leftarrow -1$ 
     $P.L.low \leftarrow \perp$ 

```

Abbildung 13: Pseudocode - DFS2: trim left interval, (Brandes, 2009, S. 24)

Da die tiefste Kante des linken Intervalls aber noch belegt ist, wird für diese als Referenzkante die tiefste Kante des rechten Intervalls angegeben und als side -1 eingetragen. Somit muss sich (5→2) auf der entgegengesetzten Seite von (4→1) befinden. Schließlich wird für die tiefste Kante auch None eingetragen und das Konfliktpaar $P(\emptyset, [(4→1), (4→1)])$ wieder auf den Stack gelegt.

Die verbleibenden Rückkanten (4→1) und (4→0) werden für die Elternknoten 1 des Knotens 2 und 0 des Knotens 1 komplett vom Stack genommen, wie man in nachfolgender Abbildung sieht.

```

▼ drop entire conflict pairs
  while  $S \neq \emptyset$  and  $\text{lowest}(\text{top}(S)) = \text{height}[u]$  do
     $P \leftarrow \text{pop}(S)$ 
    if  $P.L.low \neq \perp$  then  $\text{side}[P.L.low] \leftarrow -1$ 

```

Abbildung 14: Pseudocode - DFS2: drop entire conflict pairs, (Brandes, 2009, S. 24)

Als Referenzkante für die Elternkante (2→3) wird die höchste Kante des obersten Elements des Stacks, also (4→1) eingetragen. Für (1→2)

wird dementsprechend ($4 \rightarrow 0$) eingetragen. Für ($0 \rightarrow 1$) wird keine Kante angegeben, da es sich um die Wurzel handelt und für diese die Orientierung nicht von Belang ist. (Brandes, 2009, S. 25)

Nachdem die Testphase abgeschlossen wurde, hat man entweder eine konsistente LR-Partition gefunden oder es hat sich herausgestellt, dass der Graph nicht planar ist. Für unser Beispiel gilt ersteres, sodass man nun eine planare Einbettung bestimmen kann.

Einbettungsphase

Im ersten Schritt wird die Verschachtelungstiefe aktualisiert, damit die zur linken Seite orientierten Kanten in den Adjazenzlisten vor den rechts orientierten Kanten kommen. Man ruft für jede Kante die Funktion `sign` auf, die für linksorientierte Kante -1 und für rechtsorientierte Kanten 1 zurückgibt.

```
for  $e \in E$  do  $nesting\_depth[e] = \text{sign}(e) \cdot nesting\_depth[e]$ 
sort adjacency lists according to non-decreasing  $nesting\_depth$ 
```

Abbildung 15: Pseudocode - embedding phase: preparation, (Brandes, 2009, S. 15)

Dies wird jetzt beispielhaft für die Kante ($5 \rightarrow 2$) gezeigt. Die Referenzkante von ($5 \rightarrow 2$) ist ($4 \rightarrow 1$), beträgt also nicht None. Um die Seite von ($5 \rightarrow 2$) zu bestimmen, muss man die Seite von ($4 \rightarrow 1$) kennen. Also ruft man für diese Kante auch `sign` auf. Die Referenz von ($4 \rightarrow 1$) selbst ist None, sodass direkt ihre side, also 1, zurückgegeben wird. Die Seite von ($5 \rightarrow 2$) ist also $-1 * 1 = -1$ und ihre Referenzkante wird auf None gesetzt. Jetzt wird -1 zurückgegeben. Diese -1 wird mit der Verschachtelungstiefe von ($5 \rightarrow 2$) multipliziert: $-1 * 4 = -4$.

```
integer sign(edge  $e$ )
    if  $ref[e] \neq \perp$  then
         $side[e] \leftarrow side[e] \cdot \text{sign}(ref[e])$ 
         $ref[e] \leftarrow \perp$ 
    return  $side[e]$ 
```

Abbildung 16: Pseudocode - embedding phase: sign, (Brandes, 2009, S. 15)

Neben der (5→2) werden auch die Verschachtelungstiefen der Kanten (4→5) und (3→4) mit -1 multipliziert. All diese Kante sind nach links orientiert, da ihre höchste Rückkehrkante (5→2) ist.

Nachdem man für alle Kanten die Verschachtelungstiefe aufgestellt hat, werden die Adjazenzlisten der Knoten neu sortiert. Es ergibt sich: 0: [1], 1: [2], 2: [3], 3: [4]. 4: [5, 0, 1], 5: [2, 0].

Im nächsten Schritt muss man die noch fehlenden eintreffenden Rückkanten in die Adjazenzlisten eintragen. Es wird zum dritten Mal eine DFS durchlaufen. Allen Knoten wird der Elternknoten als erster Knoten in der Adjazenzliste eingetragen. Außerdem werden ausgehende Baumkanten als leftRef und rightRef für den momentan betrachteten Knoten eingetragen. Diese Hilfsvariablen braucht man, um links- und rechtsorientierte Kanten an der richtigen Position einzutragen.

```

if  $e_i = \text{parent\_edge}[w]$  then /* tree edge */
    make  $e_i$  first edge in adjacency list of  $w$ 
     $\text{leftRef}[v] \leftarrow e_i$ ;  $\text{rightRef}[v] \leftarrow e_i$ 
    DFS3( $w$ )

```

Abbildung 17: Pseudocode - DFS3: tree edge, (Brandes, 2009, S. 27)

Rechtsorientierte Rückkanten werden immer direkt rechts neben dem Eintrag rightRef des Zielknotens eingetragen. Linksorientierte Rückkanten werden immer direkt links vor dem Eintrag leftRef des Zielknotens eingetragen. Für diese muss außerdem leftRef immer auf die gerade eingefügte Kante gesetzt werden, da die linksorientierten Kanten von innen nach außen hinzugefügt werden.

```

else /* back edge */
    if  $\text{side}[e_i] = 1$  then
        place  $e_i$  directly behind  $\text{rightRef}[w]$  in adjacency list of  $w$ 
    else
        place  $e_i$  directly before  $\text{leftRef}[w]$  in adjacency list of  $w$ 
         $\text{leftRef}[w] \leftarrow e_i$ 

```

Abbildung 18: Pseudocode - DFS3: back edges, (Brandes, 2009, S. 27)

Die durch diesen Part entstandenen Adjazenzlisten sind: 0: [1, 4, 5], 1: [0, 2, 4], 2: [1, 5, 3], 3: [2, 4]. 4: [3, 5, 0, 1], 5: [4, 2, 0]. Für jeden Knoten

kann man jetzt die ein- und ausgehenden Kanten eines Knoten ablesen, die im Uhrzeigersinn um ihn herum angeordnet sind. Das ist die planare Einbettung des Graphen.

Im folgenden Kapitel soll dieser Algorithmus implementiert werden.

Implementierung

Ausführung des Programms

Um die nachfolgenden Ausführungen nachvollziehen zu können, ist es hilfreich, das fertige Programm ausführen zu können. Das Python-Projekt wurde in der Entwicklungsumgebung PyCharm angelegt. Der Python Interpreter arbeitet mit der Python-Version 3.9. Außerdem müssen einige externe Python-Packages in die Python-Umgebung installiert werden: matplotlib, networkx, pillow und bezier. Da die Installation des Packages bezier häufiger Probleme bereitet, verweise ich auf dessen [Dokumentation](#). Falls die Installation mit dem Befehl „pip install bezier“ nicht funktioniert, kann man auf [PyPi](#) für das jeweilige Betriebssystem eine Built Distribution herunterladen und manuell installieren. Wenn alle Packages erfolgreich installiert sind, kann das Programm gestartet werden, indem man die Datei app.py ausführt.

Programmübersicht

Hier sieht man das aufgesetzte Python-Projekt. Neben der Orientierungs-, Test- und Einbettungsphase sieht man hier eine letzte Phase „Visualisierung“. Diese Phase wird in einem späteren Kapitel behandelt.

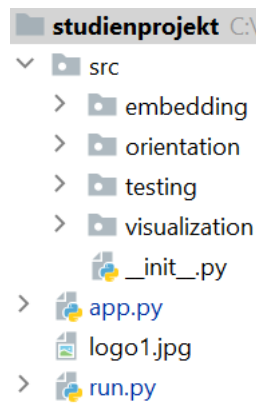


Abbildung 19: Projektstruktur, eigenes Beispiel

In der Python Datei run.py werden die einzelnen Funktionen des Algorithmus, welche für die Phasen des Algorithmus stehen, aufgerufen. Außerdem werden die meisten Hilfsvariablen und Datenstrukturen hier angelegt und initialisiert. In der Datei app.py befindet sich der Code zur Erzeugung der Nutzeroberfläche. Diesem Thema wird auch ein eigenes Kapitel gewidmet.

Der Graph wird hier mit der Bibliothek NetworkX dargestellt. Solch ein NetworkX Graph ist praktisch, da seine Eigenschaften abgefragt werden können. Zum Beispiel sieht man in der ersten Zeile der unteren Abbildung den im vorherigen Kapitel besprochenen Graphen als NetworkX Graphen. In der Zweiten befinden sich seine Kanten, in der Dritten seine Knoten.

```
Graph with 6 nodes and 9 edges
[(0, 1), (0, 4), (0, 5), (1, 2), (1, 4), (2, 3), (2, 5), (3, 4), (4, 5)]
[0, 1, 2, 3, 4, 5]
```

Abbildung 20: NetworkX Graph, eigenes Beispiel

Der Graph wird erzeugt mithilfe der Funktion `from_numpy_array`. Der Funktion muss dazu ein zweidimensionales Numpy-Array übergeben werden, welches die Adjazenzmatrix des Graphen beinhaltet. Diese Matrix wird vom Nutzer eingelesen.

Orientierungsphase

In diesem Abschnitt wird sich mit implementierungsspezifischen Aspekten der Orientierungsphase beschäftigt.

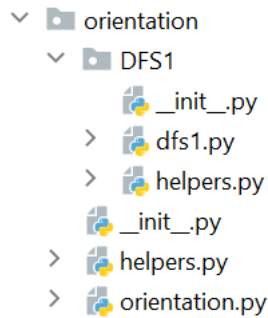


Abbildung 21: Orientierungsphase Struktur, eigenes Beispiel

Hier sieht man die Struktur des Packages orientation. In der Datei orientation.py wird die Funktion dfs1, welche in der gleichnamigen Datei dfs1.py zu finden ist, initial aufgerufen. In den Dateien helpers.py befinden sich Hilfsfunktionen für die jeweilige Ebene.

Für die DFS1 braucht man die Angaben height, parent_edge, low_pt, low_pt_2 und nesting_depth. Da man all diese Werte für alle Knoten bzw. Kanten benötigt, wurden Dictionaries zuvor in run.py angelegt. Height und parent_edge müssen schon zu Beginn per default belegt sein.

```
parent_edge = dict.fromkeys(graph.nodes, (math.nan, math.nan))
```

Abbildung 22: Elternkanten Dictionary, eigenes Beispiel

Hier sieht man beispielhaft wie allen Knoten des Graphen als parent_edge ein Tupel von zwei NaN Werten des Moduls Math zugewiesen wird. Ein Tupel von zwei Werten steht im gesamten Programm für eine Kante. Das Dictionary height wird entsprechend initialisiert, nur dass für jeden Knoten des Graphen ein math.inf Wert eingetragen wird. Die Dictionaries low_pt und low_pt_2 haben als Schlüssel die Kanten des Graphen und als Werte die Tiefpunkte. Die Schlüssel von nesting_depth sind ebenso die Kanten, die Werte sind die Verschachtelungstiefen.

Außerdem benötigt man für jeden Knoten eine Liste von Kanten, in denen der Knoten involviert ist. Da im Pseudocode eine Kante nach der anderen orientiert wird und dieses Verhalten nicht mit den NetworkX Graphen realisierbar ist, wird für jeden Knoten ein „u“ für unorientiert notiert, welches mit einem „o“ für orientiert überschrieben werden kann.

Diese Adjazenzlisten werden in der Hilfsfunktion `generate_adj_lists` generiert und werden in einem Dictionary gespeichert.

```
{0: {1: 'u', 4: 'u', 5: 'u'}, 1: {0: 'u', 2: 'u', 4: 'u'}, 2: {1: 'u', 3: 'u', 5: 'u'},  
3: {2: 'u', 4: 'u'}, 4: {0: 'u', 1: 'u', 3: 'u', 5: 'u'}, 5: {0: 'u', 2: 'u', 4: 'u'}}
```

Abbildung 23: Graph als Adjazenzliste, eigenes Beispiel

Jetzt hat man alle Vorbereitungen für die DFS1 abgeschlossen, sodass diese gestartet werden kann. In der nachfolgenden Abbildung sieht man wie das Durchlaufen der noch nicht orientierten Kanten eines Knotens `v` umgesetzt wurde. Wenn die Kante schon orientiert ist, überspringt man diese. Noch nicht orientierte Kanten werden sowohl für den Start- als auch für den Endknoten als orientiert eingetragen.

```
for w, ori in adj_lists[v].items():  
    if ori == 'o':  
        continue  
    if ori == 'u':  
        adj_lists[v][w] = 'o'  
        adj_lists[w][v] = 'o'
```

Abbildung 24: Code - DFS1: Kantenorientierung, eigenes Beispiel

Alle andere Abschnitte von DFS1 wurden dem Pseudocode ziemlich genau nachempfunden, weshalb sie hier nicht mehr aufgeführt werden.

Testphase

Das Package testing ist genau wie das Package orientation aufgebaut. Allerdings gibt es noch die zusätzlichen Dateien `add_constraints.py` und `trim_back_edges.py`, die man für die DFS2 benötigt.

```
▼ testing  
  ▼ DFS2  
    __init__.py  
    > add_constraints.py  
    > dfs2.py  
    > helpers.py  
    > trim_back_edges.py  
    __init__.py  
    > helpers.py  
    > stack.py  
    > testing.py
```

Abbildung 25: Orientierungsphase Struktur, eigenes Beispiel

In der Datei `stack.py` wird die Datenstruktur für den Stack der Konfliktpaare definiert. Ein Stack-Objekt besteht aus einer Liste von Elementen und einer Angabe darüber, wie viele Elemente es insgesamt gibt. Außerdem besitzt eine Instanz der Klasse die Funktionen `push`, `pop`, `top` und `is_empty`.

Auch hier werden in der Datei `testing.py` Vorbereitungen für die DFS2 getätigt, sodass anschließend die Funktion `dfs2` der Datei `dfs2.py` aufgerufen werden kann. Zu diesen Vorbereitungen gehört, dass die Adjazenzlisten nach der in der ersten Phase bestimmten Verschachtelungstiefe sortiert werden. Dies geschieht in der Hilfsfunktion `sort_adj_lists`, welche für jeden Knoten `v` des Graphen die Funktion `get_sorted_edges_for_v` aufruft.

```
def get_sorted_edges_for_v(nesting_depth, n):
    v_dict = {}
    for v, w in nesting_depth:
        if n == v:
            v_dict[(v, w)] = nesting_depth[(v, w)]
    v_dict_sorted = {k: v for k, v in sorted(v_dict.items(),
                                           key=lambda item: item[1])}
    return list(v_dict_sorted.keys())
```

Abbildung 26: Code - DFS2: Vorbereitung, eigenes Beispiel

Man geht mit einer Schleife die Schlüssel des Dictionary `nesting_depth` durch. Wenn der Startpunkt der Kante mit dem momentan betrachtenden Knoten übereinstimmt, dann wird die Kante dem Dictionary des Knotens hinzugefügt und als Wert die Verschachtelungstiefe eingetragen. Nach dieser können die Schlüssel des Dictionary dann sortiert und als Liste zurückgegeben werden. Somit erhält man die Adjazenzliste eines Knotens. Zusammengesetzt sieht das Dictionary `sorted_adj_lists` so aus:

```
{0: [(0, 1)], 1: [(1, 2)], 2: [(2, 3)], 3: [(3, 4)],
 4: [(4, 0), (4, 5), (4, 1)], 5: [(5, 0), (5, 2)]}
```

Abbildung 27: sortierte Adjazenzliste, eigenes Beispiel

Weitere Vorbereitungen betreffen das Initialisieren der Dictionaries `stack_bottom` und `low_pt_edge`. Im Gegensatz zu denen von `ref` und `side`, die in `run.py` vorbelegt werden mit den Werten 1 bzw. (`math.nan`, `math.nan`), werden diese nicht in einer späteren Phase gebraucht. Man legt sie also erst hier an. In der DFS2 werden nach und nach Schlüssel-Wert-Paare eingetragen. Beide Dictionaries haben als Schlüssel die Kanten des Graphen. Für `low_pt_edge` werden als Werte die Tiefpunktkanten eingetragen. `Stack_bottom` hat als Werte Stackelemente.

Ein solches Element besteht aus einer Liste von zwei Listen, welche für das linke, sowie das rechte Intervall stehen. Zuerst war geplant gewesen, die Intervalle mit Tupeln zu realisieren. Dies war nicht möglich, da Tupel immutable sind. Das bedeutet, dass die Elemente eines Tupels nicht mehr geändert werden können, nachdem dem Tupel einmal initial Werte zugewiesen wurden (Programiz, o. D.). Die Intervallgrenzen müssen aber aktualisiert werden können, sodass man stattdessen Listen gewählt hat, welche mutable sind. Die Intervallgrenzen selbst sind Kanten, welche durch Tupel dargestellt werden. In der Abbildung sieht man beispielhaft ein Element.

```
[[ (nan, nan), (nan, nan) ], [ (4, 0), (4, 0) ]]
```

Abbildung 28: Stackelement, eigenes Beispiel

Nachdem die Vorbereitungen abgeschlossen wurden, wird die Funktion `dfs2` aufgerufen. Diese Funktion sowie die in ihr verwendeten Funktionen `add_constraints` und `trim_back_edges` entsprechen im Grunde dem Pseudocode, sodass an dieser Stelle nicht weiter auf sie eingegangen wird. Allerdings wird in dieser Phase des Algorithmus bestimmt, ob der Graph planar ist, sodass die Funktion `dfs2` einen Rückgabewert benötigt. Das ist die boolesche Variable `planar`. Wenn der Graph nicht planar ist, muss die Testphase abgebrochen werden. Dazu muss die Variable `planar` direkt ausgewertet werden. Falls sie `False` beträgt, wird auch die momentane Rekursionsebene von `dfs2` abgebrochen und `False` zurückgegeben. Das heißt an jedem Punkt, an

dem die Funktion rekursiv aufgerufen wird, muss dieser Wert abgefangen und ausgewertet werden.

```
if e_i == parent_edge[e_i[1]]:
    planar = dfs2(e_i[1], sorted_adj_list, parent_edge, stack_bottom,
                  stack, low_pt_edge, low_pt, low_pt_2, height, ref, side)
    if not planar:
        return False
```

Abbildung 29: Code - DFS2: Rückgabewert planar, eigenes Beispiel

Wenn der Graph planar ist, kann im Anschluss eine planare Einbettung des Graphen bestimmt werden.

Einbettungsphase

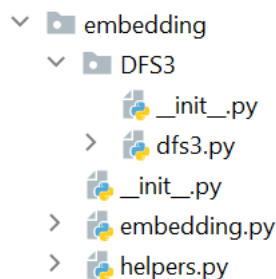


Abbildung 30: Einbettungsphase Struktur, eigenes Beispiel

Das Package embedding entspricht vom Aufbau her den anderen beiden zuvor vorgestellten Packages.

In der Funktion embed der Datei embedding.py werden zwei neue Dictionaries left_ref und right_ref angelegt. Ihre Schlüssel sind die Knoten des Graphen, ihre Werte sind die Kanten, neben denen in der Adjazenzliste die nächste Rückkante eingetragen werden soll. Diese werden nur in dieser Phase gebraucht und müssen dementsprechend nicht in run.py angelegt werden. Als nächstes wird die Verschachtelungstiefe mit der Hilfsfunktion sign aktualisiert, woraufhin die Adjazenzlisten in sorted_adj_lists neu bestimmt werden müssen. Dies geschieht genau wie in der Testphase mit der Hilfsfunktion sort_adj_lists. Man legt jetzt eine Kopie namens final_adj_lists von sorted_adj_lists an. Das wird das Dictionary sein, welches die finalen Adjazenzlisten enthalten wird, also die planare Einbettung darstellt. Dazu müssen den sortierten Adjazenzlisten noch die eintreffenden

Elternkanten hinzugefügt werden. Eine Kopie wird hier erstellt, da man die unveränderten Adjazenzlisten noch braucht, um über sie iterieren zu können. Die unterschiedliche Verwendung der beiden Dictionaries sieht man veranschaulicht in der untenstehenden Abbildung. Dieser Codeausschnitt ist aus der Funktion dfs3 der Datei dfs3.py, welche initial aus embed heraus aufgerufen wurde.

```
for e_i in sorted_adj_lists[v]:
    w = e_i[1]
    if e_i == parent_edge[w]:
        final_adj_lists[w].insert(0, e_i)
```

Abbildung 31: Code - DFS3: Baumkanten einfügen, eigenes Beispiel

Anschließend möchte man noch die Rückkanten eintragen. Auch an dieser Stelle wird eine Kopie für jede Adjazenzliste der final_adj_lists benötigt. Hier muss man mit den Indizes der Listen arbeiten, damit man weiß an welcher Stelle die Rückkante eingefügt werden soll. Allerdings kann man nicht über die gleiche Liste iterieren, bei der sich die Indizes durch das Erweitern der Liste verändern.

```
tmp_list = copy.deepcopy(final_adj_lists[w])
if side[e_i] == 1:
    for idx, s in enumerate(tmp_list):
        if s == right_ref[w]:
            final_adj_lists[w].insert(idx+1, e_i)
```

Abbildung 32: Code - DFS3: Rückkanten einfügen, eigenes Beispiel

Wenn die Funktion abgeschlossen ist, kann auch in der Funktion embed auf die fertig gestellte final_adj_lists zugegriffen werden, da Python nach der Evaluierungsstrategie „call by object reference“ (auch „call by assignment“ genannt) agiert (Is Python call by reference or call by value, 2021). Das bedeutet, dass Veränderungen, die an einem Objekt innerhalb einer Funktion getätigt werden (dafür muss das Objekt mutabel sein), in der aufrufenden Funktion sichtbar sind. Die final_adj_lists wird von der Funktion embed zurückgegeben.

Mit dem Abschluss der Einbettungsphase hat man den LR-Planaritätstest vollständig implementiert. Um diesen auch Nutzern

zugänglich zu machen, wurde anschließend eine Nutzeroberfläche erstellt.

Nutzeroberfläche

Die Nutzeroberfläche wurde mit dem GUI-Toolkit Tkinter gestaltet. Dieses wurde gewählt, da es in der Python-Standardbibliothek integriert ist und nur eine relativ einfache Oberfläche erstellt werden sollte. Den nachstehenden Anforderungen sollte diese gerecht werden:

1. selbsterklärend, sodass Erklärungstexte nicht zwingend benötigt werden.
2. konsistente Abläufe, sodass zwischen der Eingabe und dem Einlesen von Graphen gewechselt werden kann, ohne dass das Programm neu gestartet werden muss.
3. fehlerresistent, sodass auf den Versuch falsche Eingaben zu machen, nicht oder entsprechend reagiert wird.

Anhand dieser Überlegungen wurde die Nutzeroberfläche entwickelt. Diese sollte eine Startseite haben, auf der man zwischen der Eingabe einer Matrix und dem Einlesen einer Matrix switchen kann.

Wenn man eine Matrix einlesen will, dann sollte als erstes angegeben werden, wie viele Knoten (n) der Graph hat. Mit dieser Information sollte dann eine leere Adjazenzmatrix ($n \times n$) erstellt werden können, die vom Nutzer ausgefüllt werden muss. Dann sollte die Möglichkeit gegeben sein, den Graphen auf Planarität testen zu können.

Das Einlesen einer Matrix im CSV-Format sollte ähnlich verlaufen. Wenn eine Datei im richtigen Format ausgewählt wurde, dann sollte diese angezeigt werden. Auch hier sollte man überprüfen können, ob der Graph planar ist.

Man möchte die Oberfläche also in verschiedene Bereiche aufteilen:

- a) oberer Auswahlbereich: Wechsel zwischen Eingabe und Einlesen

- b) mittlerer Arbeitsbereich: Eingabe bzw. Einlesen der Matrix und Testdurchführung
- c) unterer Ausgabebereich: Mitteilung über Planarität

Um dies zu verwirklichen, wurde das Layout-Managementsystem grid von Tkinter gewählt. Das basiert auf dem Konzept, dass die Komponenten eines Rahmens in Zeilen und Spalten arrangiert werden (Tkinter Grid, o. D.). Den Komponenten muss beim Erstellen der zugehörige Referenzrahmen übergeben werden. Eine Komponente kann selbst wieder ein Rahmen sein. Erst wenn man die Komponente mit grid aufruft, erscheint diese innerhalb ihres Rahmens. Mit der grid_remove Funktion kann man die Komponente wieder entfernen. Das ist vor allem dann notwendig, wenn derselbe Vorgang neu gestartet oder zu einem anderen übergegangen werden soll.

Die Anwendung befindet sich wie oben erwähnt in der Datei app.py. Als erstes wird ein äußerster Rahmen namens root, also Wurzel, erstellt. Dieser Rahmen wird auf die Bildschirmgröße gesetzt, sodass die App den gesamten Bildschirm des Nutzers einnimmt.

Im Anschluss möchte man den oberen Auswahlbereich realisieren. Es wird ein frame0 innerhalb des Rahmens root erstellt, der das Logo der App und zwei Auswahl-Button „Enter Matrix“ und „Read in Matrix“ enthält. Der frame0 ist folgendermaßen aufgebaut, wobei die erste Angabe für die Zeile und die Zweite für die Spalte steht. Für die obere Zelle wurde außerdem die columnspan auf 2 gesetzt.

(0,0)	
(1,0)	(1,1)

Abbildung 33: frame0 Grid-System, eigenes Beispiel

Dieses grid-System sieht man beispielhaft angewendet auf die beiden Buttons b01 und b02. Die 0 im Namen steht für den frame0, die 1 bzw.

2 für den ersten und zweiten Button. Nach gleichem Schema werden auch alle anderen Komponenten der App benannt.

```
b01 = tk.Button(frame0, text='Enter matrix', bg='#D7D7D7',
                command=lambda: show_enter_matrix())
b02 = tk.Button(frame0, text='Read in matrix', bg='#D7D7D7',
                command=lambda: show_read_in_matrix())
b01.grid(row=1, column=0)
b02.grid(row=1, column=1)
```

Abbildung 34: Code - App: b01/b02, eigenes Beispiel

Umgesetzt sieht der obere Auswahlbereich schließlich so aus:



Abbildung 35: Nutzeroberfläche - oberer Auswahlbereich, eigenes Beispiel

Wie man im oberen Codeausschnitt sieht, triggern die Buttons verschiedene Funktionen. Wenn der Nutzer den Button „Enter Matrix“ drückt, wird die Funktion `show_enter_matrix` aufgerufen. Diese bewirkt, dass der mittlere Arbeitsbereich für das Eingeben einer Matrix angezeigt wird. Das ist der `frame1`, welcher in `frame11` und `frame12` aufgeteilt ist. In `frame11` soll die Eingabe der Knotenanzahl von statten gehen.



Abbildung 36: Nutzeroberfläche - mittlerer Arbeitsbereich: Eingabe der Knotenanzahl, eigenes Beispiel

Mit dem Drücken des Buttons „Create Matrix“ wird die Funktion `create_adj_matrix` aufgerufen. Für eine Knotenanzahl zwischen 0 und 12 soll eine Matrix erstellt werden können. Dieser Bereich wurde zum einen gewählt, da bis zu dieser Größe die Matrizen auf der Nutzeroberfläche gut platziert werden können. Zum anderen funktioniert

die später eingeführte Visualisierung für kleinere Graphen besser und schnittfreier. Für andere Eingaben außerhalb dieses Bereichs passiert nichts.

In der Funktion wird als erstes eine neue leere Matrix erstellt. Dafür muss die in einem vorherigen Durchgang erstellte Matrix entfernt werden. Das muss man tun, da ansonsten die Matrizen übereinandergelegt werden würden und Größere hinter Kleineren hervorscheinen würden. Da beim erstmaligen Erstellen einer Matrix keine Vorherige vorhanden wäre, wird initial eine 0x0-Matrix namens matrix als Platzhalter erstellt. Diese ist allerdings nicht zu sehen.

```
frame12 = tk.Frame(frame1)
matrix = AdjMatrix(frame12, 0)
frame12.grid(row=1, column=0)
```

Abbildung 37: Code - App: Platzhaltermatrix, eigenes Beispiel

Die Matrix selbst ist eine Instanz der Klasse AdjMatrix, welche die Klassenattribute entry, rows und columns besitzt. Rows und columns stehen für die Form der Matrix, also nxn. Entry ist ein Dictionary, welches Tkinter Eingabefelder enthält.

```
class AdjMatrix(tk.Frame):
    def __init__(self, parent, n):
        tk.Frame.__init__(self, parent)
        self.entry = {}
        self.rows = n
        self.columns = n
```

Abbildung 38: Code - App: Klasse AdjMatrix, eigenes Beispiel

Den Eingabefeldern wird ein Name zugewiesen. Dieser beschreibt die Position des Feldes innerhalb der Matrix. Über diesen Index aus Zeile und Spalte kann man die Eingabefelder später ansprechen. Außerdem unterteilt man die Felder in drei Kategorien: Felder der Diagonalen, Felder, die rechts oberhalb der Diagonalen liegen (obere Dreiecksmatrix) und Felder die links unterhalb der Diagonalen liegen (untere Dreiecksmatrix). Diese Typen von Feldern haben unterschiedliche Einstellungen. Die Diagonalfelder sowie die Felder der unteren Dreiecksmatrix sind deaktiviert. Die Diagonalfelder sind außerdem festgelegt auf den Wert 0, da keine Schleifen erlaubt sind.

Die Felder der unteren Dreiecksmatrix füllen sich erst dann, wenn der Nutzer die Felder der oberen Dreiecksmatrix eingibt. Damit garantiert man, dass die Matrix symmetrisch ist und beugt Eingabefehlern vor.

```
for row in range(self.rows):
    for column in range(self.columns):
        index = (row, column)
        if row == column:
            e = tk.Entry(self, name=str(row) + ',' + str(column), justify='center', width=8)
            e.insert(0, '0')
            e.config(state='disabled')
        elif row > column:
            e = tk.Entry(self, name=str(row) + ',' + str(column), justify='center', width=8)
            e.config(state='disabled')
```

Abbildung 39: Code - App: Diagonalfelder, untere Dreiecksmatrix, eigenes Beispiel

Für das Eingeben der Felder der oberen Dreiecksmatrix braucht man eine callback-Funktion, welche überprüft, ob die eingegebenen Werte valide sind. Die Werte 0 und 1 sind erlaubt, sodass in ihrem Fall auch das zugehörige Feld der unteren Dreiecksmatrix auf den gleichen Wert gesetzt werden kann. Wenn ein ungültiger Wert oder kein Wert eingegeben wird, wird der Wert intern sowie der Wert des dazugehörigen Feldes auf 0 gesetzt. Um das zugehörige Feld anzusprechen, benötigt man den Index des Ausgangsfeldes. Dessen Reihe und Spalte wird einfach vertauscht. Die callback-Funktion braucht also als Übergabewerte die Eingabe des Feldes und dessen Index.

Außerdem muss die callback-Funktion registriert werden, sodass Tkinter sie identifizieren kann. Dann muss dem Eingabefeld gesagt werden, wann die callback_Funktion aufgerufen werden soll. Hier wurde als validate-Wert „key“ gewählt, also mit Veränderung der Eingabe. Zusätzlich muss angegeben werden, welche Informationen über das Eingabefeld mitgegeben werden müssen. Hier braucht man einmal den Input %P und den Namen des Feldes %W, aus welchem man den Index herauslesen kann. Diese werden dem Entry als validatecommand übergeben (Shipman, 2013).

```

else:
    reg = self.register(self.callback)
    e = tk.Entry(self, name=str(row) + ',' + str(column), justify='center', validate="key",
                 validatecommand=(reg, '%P', '%W'), width=8)

```

Abbildung 40: Code - App: obere Dreiecksmatrix (callback-Funktion validieren), eigenes Beispiel

Die fertiggestellte Matrix matrix sowie der Button b121 mit dem Label „Planarity Test“ werden dem frame 12 hinzugefügt, welcher im Anschluss auf der Nutzeroberfläche angezeigt wird. Das geschieht, indem man diesen wiederum dem frame1 hinzufügt. Nun kann der Nutzer die Adjazenzmatrix des ihn interessierenden Graphen eingeben. Das sieht man in der nachfolgenden Abbildung.

The image shows two side-by-side screenshots of a GUI. Both have a 'Number of nodes' input field with the value '4' and a 'Create Matrix' button. Below this is a 4x4 matrix. In the left screenshot, the matrix has zeros on the diagonal and zeros elsewhere. In the right screenshot, the matrix has ones on the diagonal and ones at (0,1), (0,2), (1,0), (1,3), (2,0), and (3,1). Below the matrix is a 'Planarity Test' button.

0			
	0		
		0	
			0

0	1	1	0
1	0	0	1
1	0	0	0
0	1	0	0

Abbildung 41: Nutzeroberfläche - mittlerer Arbeitsbereich: eingegebene Matrix, eigenes Beispiel

Wenn der Button „Planarity Test“ getätigt wird, wird die Funktion `convert_adj_matrix_to_np_array` aufgerufen. Diese holt sich die aktuelle Version der Matrix mithilfe deren `get`-Methode. Diese gibt die Matrix in Form einer zweidimensionalen Liste der Einträge zurück. Anschließend wird diese Liste in ein Numpy-Array überführt, sodass sie der Funktion `planar_test` übergeben werden kann. Nachdem dieses Array in einen Networkx Graphen umgeformt wurde, kann die Funktion `run(graph)` ausgeführt werden, welche im Kapitel „Implementierung“ ausführlich besprochen wurde. Der Rückgabewert ist die boolesche Variable `planar`. Falls diese `True` ist, wird eine Message-Box mit dem Text „Your entered graph is planar.“ erstellt. Falls diese `False` ist, lautet der Text „Your entered graph is not planar.“ Diese Message-Box gehört zum unteren Ausgabebereich. Dieser wird durch den frame2 präsentiert.

Jetzt soll noch der alternative mittlere Arbeitsbereich für das Einlesen einer Matrix besprochen werden. Dieser wird im Code mit `frame1a` bezeichnet. Der Buchstabe „a“ steht für alternative.

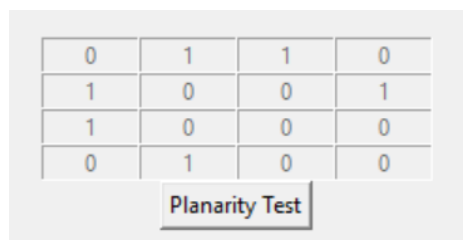
Wenn der Nutzer im oberen Auswahlbereich auf den Button „Read in Matrix“ drückt, triggert er die Funktion `show_read_in_matrix`. Hier wird mit der Tkinter-Funktion `askopenfile` eine vom Nutzer ausgewählte CSV-Datei aus dessen Dateiverzeichnis eingelesen. Dazu erscheint dem Nutzer ein Auswahlfeld.

Wenn erfolgreich eine CSV-Datei ausgewählt wurde, wird aus der Matrix der Datei ein Numpy-Array gemacht. Das ist nur dann möglich, wenn die Zahlen der Matrix mit Kommas voneinander getrennt wurden. Im nächsten Schritt wird das Numpy-Array mithilfe der Funktion `check_correct_matrix_format` auf Richtigkeit geprüft. Es muss quadratisch sowie symmetrisch sein, nur aus Einsen und Nullen bestehen und auf der Diagonalen nur Nullen stehen haben.

```
file = askopenfile(parent=frame0, mode='rb', title='Choose a file',  
                  filetype=[("Csv File", "*.csv")])  
if file:  
    adj_matrix_array = np.genfromtxt(file, delimiter=',')  
    if check_correct_matrix_format(adj_matrix_array):  
        global matrixa
```

Abbildung 42: Code - App: Matrix im richtigen Format einlesen, eigenes Beispiel

Das Array muss also die gleichen Voraussetzungen erfüllen, wie die Matrix, die im ersten Fall vom Nutzer eingegeben wird. Erst dann wird eine Matrix namens `matrixa` der Klasse `AdjMatrix` erstellt. Ihre Entry-Felder werden mit der `set`-Methode gesetzt. Diese `matrixa` wird jetzt zusammen mit dem Button `b1a1`, der ebenso wie der Button `b121` für das Ausführen des Planaritätstest steht, dem Nutzer angezeigt.



0	1	1	0
1	0	0	1
1	0	0	0
0	1	0	0

Planarity Test

Abbildung 43: Nutzeroberfläche - mittlerer Arbeitsbereich: eingelesene Matrix, eigenes Beispiel

Auch hier erscheint derselbe `frame2` mit dem Ausgabefeld.

Somit wurden alle Abschnitte der App, deren Funktionsweise und Implementierung vorgestellt. Im nächsten Kapitel soll der App noch eine

Möglichkeit zur Visualisierung von planaren Graphen hinzugefügt werden.

Visualisierung

Bei der Visualisierung wurde sich auf planare Graphen beschränkt, da für diese schon eine planare Einbettung vorliegt. Hierbei möchte man nicht nur zeigen, dass der Graph planar ist, sondern auch wie der LR-Planaritätstest den Graphen in einen Tiefensuchbaum und links-bzw. rechtsorientierte Rückkanten aufteilt.

Literaturrecherche

Bei der durchgeführten Literaturrecherche wurden einige geläufige Methoden zur Zeichnung von planaren Graphen gefunden. Die wohl bekannteste, befasst sich mit dem Zeichnen von geradlinigen Kanten und nennt sich „Straight line drawing“ (Nishizeki & Rahman, 2004, S. 45). Eine Abwandlung dieser Methode ist die des „Rectangular Drawing“, wobei die Kanten nur horizontal oder vertikal verlaufen dürfen, sodass jedes eingeschlossene Gebiet die Form eines Rechtecks annimmt (Nishizeki & Rahman, 2004, S. 129). Eine weitere Variante ist die des „Orthogonal Drawing“. Hier dürfen die Kanten zusätzlich zum geradlinigen Verlauf im rechten Winkel abknicken (Duncan & Goodrich, 2013, S. 234). Wenn ein Knoten zu mehr als vier Kanten gehört, funktioniert diese Methode nicht mehr. Stattdessen kann man mit der Methode „Polyline Drawing“ arbeiten. Eine Kante läuft dabei einen Linienzug ab, wobei die Kante nicht mehr rechtwinklig abknicken muss. Hier wird mit Winkeln gearbeitet.

Außerdem gibt es einige Ansätze, die sich damit beschäftigen, Linienzüge, um zuvor gesetzte hinderliche Knoten herum zu zeichnen und diese schließlich durch eine Spline-Kurve zu glätten und lokal anzupassen (Kobourov, Nöllenburg, & Monique, 2013, S. 41). Ein Spline ist eine Funktion, die sich aus Polynomstücken zusammensetzt. (Walz, 2017). Diese Methode wurde erweitert, indem man den Knoten eines Graphen mit einer Umgebung versehen hat, welche proportional

zu dessen Grad ist. Auf dem Rand der Umgebung werden in gleichmäßigen Abständen Ports für die Kanten angelegt. Durch die Einführung der Ports, erreicht man eine konstante Kantenkomplexität und ein nahezu optimale Winkelauflösung (Kobourov, Nöllenburg, & Monique, 2013, S. 42). Ersteres beschreibt die maximale Anzahl von Bögen und Geraden pro Kante. (Kobourov, Nöllenburg, & Monique, 2013, S. 42). Letzteres den kleinsten Winkel zwischen zwei Kanten, die zu demselben Knoten führen (Hoffmann, van Kreveld, Kusters, & Rote, 2014).

Neben diesen allgemeingültigen Verfahren, wurde auch eine Bachelorarbeit mit dem Titel „Implementation und Animation des Links-Rechts-Planaritätstests“ (Kaiser, 2009) gefunden, in welcher der Visualisierung des LR-Planaritätstests ein Kapitel gewidmet wurde. In diesem wurde konzeptuell und mit Codeausschnitte erläutert, wie die planare Zeichnung erstellt wurde. Inspiriert von den Ansätzen der Literaturrecherche und nach Vorlage der Bachelorarbeit, wird im Folgenden zusammengefasst, wie die Visualisierung in diesem Projekt umgesetzt wurde.

Algorithmus

Der Tiefensuchbaum soll von unten nach oben aufgebaut werden. Das heißt, dass man beginnend beim Startknoten, die ausgehenden Kanten nach oben zeichnet, sodass die Kindknoten höher positioniert sind. Für deren Kindknoten soll wiederum dasselbe gelten. Dies gelingt, indem man jedem Knoten einen Keil zuweist, welcher den Bereich für dessen Unterbaum angibt. Der Startknoten bekommt einen Keil von 180° , welcher auf die Kindsknoten gleichmäßig aufgeteilt wird. Da die Keile alle gleich groß sind, braucht man noch eine Angabe an welcher Gradzahl der Keil beginnt, also den Winkel. Durch diese Angaben kann die Position der Knoten bestimmt werden (Kaiser, 2009, S. 47).

Die Rückkanten sollen mithilfe von Bézierkurven gezeichnet werden. Bézierkurven sind Kurven, deren Verlauf durch Punkte definiert werden. Am einfachsten lässt sich das anhand eines Beispiels erkennen.

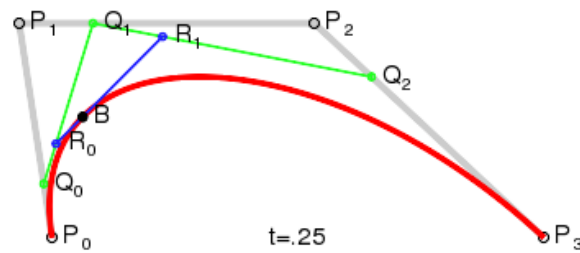


Abbildung 44: Bézierkurve, (Bezier-Kurve, kein Datum)

Für jede Rückkante muss man wissen, um welche Blattknoten sie herumgezeichnet werden muss. Die Bézierpunkte für diese Knoten müssen sich oberhalb von diesen befinden, also in Verlängerung ihrer Elternkante. Außerdem möchte man zwischen Rückkanten, die bei einem Blatt starten und Rückkanten, die bei einem Baumknoten starten unterscheiden. Das ist sinnvoll, da Letztere tiefer starten als die Blattknoten, um die sie herumverlaufen soll. Also möchte man noch einen weiteren Bézierpunkt hinzufügen, welcher zwischen dem Startpunkt der Rückkante und ihrem ersten relevanten Blattknoten liegt. Zusätzlich wird dieser noch orthogonal um einen bestimmten Wert nach außen verschoben. Dies kann man mit einem Normalenvektor umsetzen. Damit erreicht man, dass die Bézierkurve, um den höher liegenden ersten Blattknoten „herumkommt“. Nach dem gleichen Prinzip behandelt man Rückkanten, die um keine Blattknoten herum verlaufen müssen. Dies macht man, da es passieren kann, dass die Rückkante ohne eine Verschiebung nach außen auf den Baumkanten liegt.

Implementierung

In dem Python-Projekt wird ein neues Package visualization hinzugefügt. Man startet in der Funktion visualize der Python-Datei visualization.py.

```

v visualization
  > BezierCurves
  > DFSTreeEdges
    __init__.py
  > visualization.py
    __init__.py

```

Abbildung 45: Visualisierungsphase Struktur, eigenes Beispiel

Als erstes wird ein neuer Networkx Graph erstellt. Diesmal handelt sich um einen DiGraph, also um einen gerichteten Graphen. Dieser besitzt zu Beginn weder Knoten noch Kanten. Zusätzlich erstellt man ein Axes-Objekt der Bibliothek matplotlib. Das Axes-Objekt steht für das Koordinatensystem, in welchem man den Graphen schlussendlich plotten will. Das Erstellen des Tiefensuchbaums wird in dem Unterpaket DFSTreeEdges behandelt. In der Datei `get_dfs_tree_edges.py` wird in der gleichnamigen Funktion ein Dictionary `sorted_tree_edges` erstellt, welches für jeden Knoten die ausgehenden Kanten des Tiefensuchbaums in einer Liste speichert. Diese werden aus `final_adj_lists` mithilfe der Elternkanten `parent_edge` in der richtigen Reihenfolge herausgefiltert. Nun werden in der Funktion `set_angle_and_wedge` der gleichnamigen Datei die Keile und Winkel der Knoten rekursiv gesetzt. Diese Funktion wurde einem Codeausschnitt der Bachelorarbeit nachempfunden (Kaiser, 2009, S. 48). Anschließend kann man die Position der Knoten bestimmen und diese auch dem Networkx Graphen hinzufügen. Dem Startknoten werden dafür die Koordinaten (0,0) zugewiesen. Mit diesem Knoten als `node` wird die rekursiv-definierte `set_coordinates` Funktion erstmalig aufgerufen. Diese führt erneut eine DFS durch.

```
def set_coordinates(graph, sorted_tree_edges, node, coord, wedge, angle):
    for edge in sorted_tree_edges[node]:
        w = edge[1]
        p = coord[node]
        new_x = p[0] + math.cos(math.radians(angle[w] + (wedge[w] / 2)))
        new_y = p[1] + math.sin(math.radians(angle[w] + (wedge[w] / 2)))
        coord[w] = (new_x, new_y)
        graph.add_node(w, pos=coord[w], label=w)
        graph.add_edge(node, w)
        set_coordinates(graph, sorted_tree_edges, w, coord, wedge, angle)
```

Abbildung 46: Code - DFSTreeEdges: Koordinaten bestimmen, eigenes Beispiel

Die Positionen der Kindsknoten werden gesetzt, indem man zu den x- bzw. y-Koordinaten des Elternknotens den Kosinus bzw. Sinus des Winkels addiert. Dies kann man sich mit dem Einheitskreis gut veranschaulichen. Auf den Startwinkel wird nur die Hälfte des Keils

addiert, damit der Knoten sich genau in der Mitte des Keils befindet. Jetzt kann der Knoten mit den berechneten Koordinaten dem Graphen hinzugefügt werden, genauso wie die Kante vom Eltern- zum Kindknoten. Auch zu dieser Implementierung gab es ein Codebeispiel (Kaiser, 2009, S. 48).

Nun hat man alle Vorbereitungen für die Visualisierung des Tiefensuchbaums abgeschlossen. Die Rückkanten werden im Unterpackage BezierCurves umgesetzt. Dazu ruft man in visualize die Funktion `draw_bezier_curves` aus der gleichnamigen Datei auf.

```
def draw_bezier_curves(graph, ax, final_adj_list,
                      sorted_tree_edges, parent_edge, height, side):

    position = nx.get_node_attributes(graph, 'pos')
    nodes = nx.draw_networkx_nodes(graph, position, node_size=200,
                                   node_color='lightgray', ax=ax)
    labels = nx.draw_networkx_labels(graph, position,
                                     labels={n: n for n in graph}, ax=ax)
    edges = nx.draw_networkx_edges(graph, position, ax=ax)
```

Abbildung 47: Code - BezierCurves: `draw_bezier_curves`, eigenes Beispiel

In dieser werden im ersten Schritt die Knoten, deren Label und die Kanten gezeichnet. Networkx hat eigene Funktionen dafür. Wichtig ist hierbei, dass man das Axes-Objekt mit übergibt.

Anschließend geht man die Rückkanten durch. Das sind solche Kanten, die in den Adjazenzlisten von `final_adj_lists` vorkommen, aber nicht in den `sorted_tree_edges`. Für jede Rückkante werden in der Funktion `find_relevant_nodes` die Blattknoten, welche umlaufen werden sollen, bestimmt. Für rechtsorientierte Rückkanten müssen dazu alle Baumkanten, die in der Adjazenzliste des Startknoten nach der Kante aufgelistet sind, von links nach rechts durchlaufen werden. Für linksorientierte Rückkanten müssen die Baumkanten, die vor der Rückkante liegen, von rechts nach links durchlaufen werden. Die Zielknoten der Baumkanten besitzen Unterbäume, welche rekursiv bis zu den Blattknoten durchlaufen werden müssen. Dies geschieht in der Hilfsfunktion `add_subtree_leafes`. Ein Blattknoten wird dadurch identifiziert, dass dieser in dem Dictionary `sorted_tree_edges` eine leere

Liste als Wert besitzt.

Sobald man die `relevant_nodes` bestimmt hat, müssen die Bézierpunkte berechnet werden. Dies geschieht in der Funktion `determine_bezier_points`. Wenn die Liste `relevant_nodes` leer ist, wird der oben erläuterte Hilfs-Bézierpunkt in der Funktion `bezier_point_by_normal_vector` erstellt. Für solche Rückkanten verläuft der Normalenvektor für rechtsorientierte Rückkanten (`side = 1`) vom Mittelpunkt aus in positiver x-Richtung und für linksorientierte Rückkanten (`side = -1`) in negativer x-Richtung. Dies sieht man in der folgenden Abbildung für die rechtsorientierte Rückkante ($4 \rightarrow 5$) und für die linksorientierte Rückkante ($4 \rightarrow 1$).

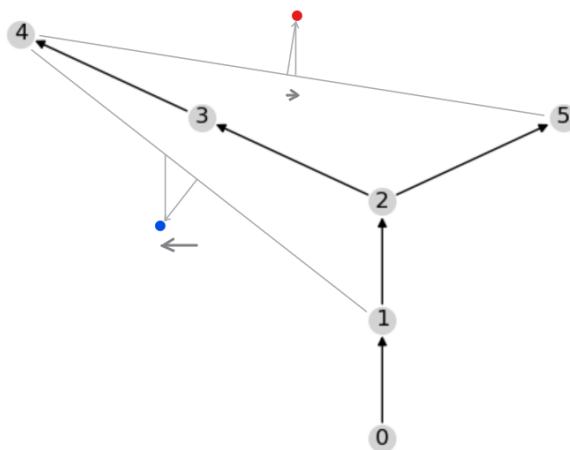


Abbildung 48: Hilfs-Bézierpunkt für Rückkanten ohne relevante Blattknoten, eigenes Beispiel

Ebenfalls aufgerufen wird diese Funktion, wenn der Startpunkt der Rückkante kein Blattknoten ist. Hierbei wird der übergebene Wert `side` mit -1 multipliziert, da der Normalenvektor für rechts- und linksorientierte Rückkanten genau in die entgegengesetzte Richtung verlaufen muss. Man stelle sich beispielhaft in der obigen Abbildung vor, dass die rechtsorientierte Rückkante ($3 \rightarrow 5$) um den Knoten 4 herum gezeichnet werden soll. Der Normalenvektor auf halber Strecke der Knoten 3 und 4 würde genau wie der Normalenvektor der linksorientierten Rückkante ($4 \rightarrow 1$) verlaufen.

Für alle Rückkanten werden nun die restlichen Bézierpunkte oberhalb der `relevant_nodes` gesetzt. Hierfür benötigt man die Positionen der Blattknoten und deren Elternknoten. Diese können aus dem Networkx Graphen ausgelesen werden. `Position` ist ein Dictionary, welches für jeden Knoten die Koordinaten als Tupel gespeichert hat.

```
position = nx.get_node_attributes(graph, 'pos')
```

Abbildung 49: Code - BezierCurves: Position der Knoten in `determine_bezier_points` bestimmen, eigenes Beispiel

Die Länge der Strecke von einem Blattknoten zu seinem zugehörigen Bézierpunkt soll ein Vielfaches der Länge der Strecke von Elternknoten zu Blattknoten betragen. Das Konzept und die Implementierung wurde aus der Bachelorarbeit übernommen (Kaiser, 2009, S. 51) und wird hier nicht näher erläutert. Erwähnenswert ist an dieser Stelle, dass für die verschiedenen Kategorien von Rückkanten die Parameter `hdc` und `counter` unterschiedlich belegt wurden. `Hdc` gewichtet den Höhenunterschied des Start- und Endknoten der Rückkanten und `counter` zählt die Anzahl der relevanten Blattknoten. Die Werte wurden im Nachhinein durch Ausprobieren festgelegt, um den Verlauf der Rückkanten zu optimieren.

Abschließend wurde ein weiterer Hilfs-Bézierpunkt mithilfe des Normalenvektors zwischen dem letzten relevanten Blattknoten und dem Endpunkt der Rückkante hinzugefügt.

Die Koordinaten der Bézierpunkte wurden einer Liste `bezier_coords` hinzugefügt. Diese enthält zwei Listen, wobei die Erste die x-Koordinaten und die Zweite die y-Koordinaten speichert. Dieses Format wird von dem verwendeten Package `bezier` benötigt. Die Liste wird am Ende der Funktion `determine_bezier_points` zurückgegeben und in der Funktion `draw_bezier_curves` verwendet, um die Bézierkurven zu zeichnen.

```

nodes = np.asfortranarray(bezier_coords)
curve = bezier.Curve(nodes, degree=len(bezier_coords[0]) - 1)
if side[back_edge] == 1:
    curve.plot(num_pts=1000, ax=ax, color='red')
elif side[back_edge] == -1:
    curve.plot(num_pts=1000, ax=ax, color='blue')

```

Abbildung 50: Code - BezierCurves: Plotten der Bézierkurven, eigenes Beispiel

Nun hat man den Code für die Visualisierung des Graphen fertig gestellt. Dieser ist aber noch nicht in die App eingebaut. Dazu wird für planare Graphen in der Funktion `planar_test` in der Datei `app.py` neben dem Ausgabefeld ein Button `b21` namens „Visualize Graph“ eingebaut, welcher die Funktion `visualize_the_graph` aufruft. Diese Funktion macht nichts anderes als die Funktion `visualize` in dem Package `visualization` aufzurufen.

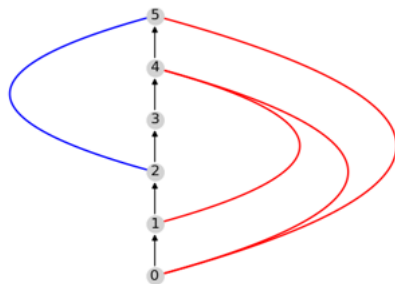


Abbildung 51: planare Zeichnung des im Kapitel "Algorithmus" besprochenen Graphen, eigenes Beispiel

Nun wurden alle Funktionen des Programms vorgestellt. Abschließend soll ein Ausblick gegeben werden, wie das Programm erweitert bzw. verbessert werden kann.

Ausblick

Das Programm kann bis dato nur die Visualisierung von planaren Graphen realisieren. Dabei funktioniert die Visualisierung der Rückkanten, die nicht bei Blattknoten starten, mit steigender Anzahl von Rückkanten immer schlechter. Das bedeutet, dass sich die Rückkanten vermehrt schneiden, da sie nicht relativ zu anderen Rückkanten gezeichnet werden.

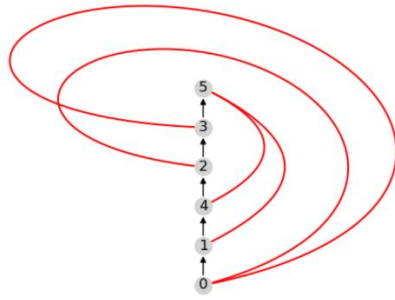


Abbildung 52: nicht-planare Zeichnung eines planaren Graphen, eigenes Beispiel

Die Rückkanten verlaufen dennoch in die richtige Richtung (links/ rechts) als auch um alle Knoten, um die sie herum verlaufen sollen. An dieser Stelle könnte man versuchen noch weitere Kriterien einzubauen, damit die Anzahl an Schnitten minimiert wird.

Außerdem wäre eine Erweiterung um die Visualisierung von nicht planaren Graphen erstrebenswert. Man könnte entweder anzeigen lassen, welche Kanten dafür verantwortlich sind, dass der Graph nicht planar ist. Oder der Graph wird auf einen Kuratowski-Graph ($K_{3,3}$ und K_5) zurückgeführt, um zu beweisen, dass der Graph nicht planar sein kann.

Ein anderer Ansatzpunkt wäre die Weitergestaltung der Nutzeroberfläche. Diese wurde bis jetzt sehr schlicht gehalten. Der Nutzer sollte sich zurechtfinden und alle eingebauten Features des Programms verwenden können. Schön wäre es hier, wenn die einzelnen Abschnitte der Oberfläche mit Erklärungstexten versehen werden würden. Dadurch würde sich der Zugang zum Programm vereinfachen. Ebenfalls könnte man das Design mehr der Moderne und deren ästhetischen Ansprüchen anpassen.

Fazit

Zusammenfassend kann man sagen, dass alle zu Beginn gesetzten Ziele umgesetzt werden konnten: Das Verfahren des LR-Planaritätstest konnte nachvollzogen, sowie implementiert und durch die Visualisierung des Graphen erweitert werden. Somit ist das Projekt in sich geschlossen und kann an diesem Punkt abgeschlossen werden.

Literaturverzeichnis

- Bezier-Kurve*. (kein Datum). Von Academic: <https://de-academic.com/dic.nsf/dewiki/167126> abgerufen
- Brandes, U. (2009). *The Left-Right Planarity Test*.
- de Fraysseix, H., & Rosenstiehl, P. (1982). A depth-first characterization of planarity. *Annals of Discrete Mathematics* 13, S. 75-80.
- Duncan, C. A., & Goodrich, M. T. (2013). Planar Orthogonal and Polyline Drawing Algorithms. In R. Tamassia, *Handbook of Graph Drawing and Visualization* (S. 223-246). CRC Press.
- Hoffmann, M., van Kreveld, M., Kusters, V., & Rote, G. (2014). Quality Ratios of Measures for Graph Drawing Styles. *Proceedings of the 26th Canadian Conference on Computational Geometry*, (S. 1). Halifax.
- Is Python call by reference or call by value*. (19. April 2021). Von GeeksforGeeks: <https://www.geeksforgeeks.org/is-python-call-by-reference-or-call-by-value/> abgerufen
- Kaiser, D. (2009). *Implementation und Animation des Links-Rechts-Planaritätstests*. Konstanz.
- Kobourov, S., Nöllenburg, M., & Monique, T. (2013). A Brief History of Curves in Graph Drawing. *Dagstuhl Seminar 13151 - Drawing Graphs and Maps with Curves*, (S. 40). Dagstuhl.
- Nishizeki, T., & Rahman, M. (2004). *Planar Graph Drawing*. Singapore: World Scientific Publishing Co. Pre. Ltd.
- Programiz*. (o. D.). Von Python Tuple: <https://www.programiz.com/python-programming/tuple> abgerufen
- Shipman, J. W. (31. 12 2013). *Tkinter 8.5 reference: a GUI for Python*. Von About New Mexico Tech: <https://anzelg.github.io/rin2/book2/2405/docs/tkinter/entry-validation.html> abgerufen
- Tkinter Grid*. (o. D.). Von Python Tutorial: <https://www.pythontutorial.net/tkinter/tkinter-grid/> abgerufen
- Walz, P. D. (2017). *Splinefunktionen*. Von Spektrum - Lexikon der Mathematik : <https://www.spektrum.de/lexikon/mathematik/splinefunktionen/9907> abgerufen
- Weisstein, E. W. (o. D.). *Simple Graph*. From MathWorld--A Wolfram Web Resource. Von <https://mathworld.wolfram.com/SimpleGraph.html> abgerufen
- White, R. T., & Ray, A. T. (2021). *Practical Discrete Mathematics*. Birmingham: Packt Publishing Ltd.