Aufgabe 2: Stilvolle Päckchen

Teilnahme-ID: 69082

Bearbeiter dieser Aufgabe: Vincent Xigu Liu

5. März 2024

**Inhaltsverzeichnis**

[1. Lösungsidee 2](#_Toc161237185)

[1.1. Bemerkungen zur Problemmodellierung 2](#_Toc161237186)

[1.2. Ganzzahlige Lineare Optimierung mit Enumeration maximaler Cliquen 3](#_Toc161237187)

[1.2.1. Enumeration maximaler Cliquen und Finden „doppelter Knoten“ 3](#_Toc161237188)

[1.2.2. Formulierung des Ganzzahligen Linearen Programms 3](#_Toc161237189)

[1.2.3 Packen der Päckchen 4](#_Toc161237190)

[2. Analyse des Problems 4](#_Toc161237191)

[2.1. Komplexitätsbetrachtung 4](#_Toc161237192)

[2.2. Laufzeitbetrachtung 4](#_Toc161237193)

[3. Mögliche Erweiterungen 4](#_Toc161237194)

[3.1. Festlegung des Start- und Zielknotens 4](#_Toc161237195)

[4. Implementierung 5](#_Toc161237196)

[5. Beispiele 5](#_Toc161237197)

[6. Quellcode 10](#_Toc161237198)

Das vorliegende Problem, das im Folgenden Päckchen-Pack-Problem (kurz PPP) genannt werden soll, erinnert an eine Variante des Cliquenproblems und hat zum Ziel, zueinander passende Kleidungsstücke zu finden, welche in Boxen gepackt werden. Wir modellieren das Problem als ungerichteten Graphen , wobei jeder Knoten ein Stil repräsentiert und einem Tupel zugeordnet wird, wobei die zugehörige Sorte ist und die Anzahl des jeweiligen Kleidungsstückes einer Sorte repräsentiert. Zwischen zwei Knoten sei jetzt nun genau dann eine Kante, wenn deren Stile zusammenpassen. Da innerhalb einer gepackten Box alle Stile zueinander passen müssen, bilden alle ausgewählten Stile, die in einer Box sind, einen zugehörigen kompletten Teilgraphen. Zum Packen einer Box wählen wir also Stile aus, dessen Knoten in mit einem kompletten Teilgraphen korrespondieren und „entnehmen“ aus den zugehörigen Tupeln von jedem Knoten Kleidungsstücke so, dass in der Box mindestens 1 Kleidungsstück jeder Sorte und höchstens 3 Kleidungsstücke jeder Sorte sind.

Das Ziel ist es nun, durch Packoperationen die gesamte Kleidungsstückanzahl im Graphen, oder formal ausgedrückt,

zu minimieren.

# Lösungsidee

Wir vermuten zunächst, dass das Problem NP-vollständig ist. Ein konkreter Beweis wird weiter unten angegeben. Wir betrachten daher im Folgenden insbesondere Ansätze mit exponentieller Laufzeit, um das Problem zu lösen und bedienen uns Methoden linearer Optimierung, da sich aus einer gegebene Probleminstanz gut ein lineares Programm herstellen lässt.

## Bemerkungen zur Problemmodellierung

Da die Zeitkomplexität gängiger Methoden zum Lösen ganzzahliger linearer Programme mindestens proportional mit der Anzahl von Variablen steigt, minimieren wir zunächst mithilfe einer Hilfsumformung des Problems die benötigten Variablen, indem wir statt der Einzelpäckchen alle Kleidungsstücke in maximalen Cliquen betrachten, die in Boxen gepackt werden sollen. Diese Betrachtung ist möglich, weil folgendes Lemma gilt:

**Lemma 1.1:** Wenn in einem kompletten, ungerichteten Graphen G und der zugehörigen Kleidungsstücktabelle T die größte Summe einer Sorte nicht größer ist als das Dreifache der kleinsten Summe einer Sorte, gibt es ein Programm mit polynomieller Laufzeit, welche alle Kleidungsstücke in Päckchen packen kann.

*Beweis.* Da zunächst G komplett ist, lassen sich beliebige zwei Stile aus dem Graphen sich miteinander kombinieren. Wir betrachten nun die Summen der jeweiligen Sorten und ermitteln die größte Summe und die kleinste Summe . Anschließend erstellen wir Päckchen und verteilen alle Kleidungsstücke, die zu der Sorte von gehören, so auf die Päckchen, dass in jedem Päckchen genau ein Kleidungsstück landet. Nun verteilen wir all die andere Kleidung der anderen Sorten so in die Päckchen, sodass in jedem Päckchen dann von jeder Sorte mindestens ein Kleidungsstück und maximal drei Kleidungsstücke vorhanden sind. Die Sorte der Kleidungsstücke ist dabei egal, da aufgrund der Komplettheit des Graphen garantiert ist, dass jeder Stil zu jedem anderen Stil passt Die untere Schranke ist möglich, weil es keine Sorte gibt, die insgesamt weniger als Kleidungsstücke gibt, und die obere Schranke ist möglich, weil es keine Sorte gibt, die mehr als Kleidungsstücke hat und nach Voraussetzung gilt.

Betrachten wir die Laufzeit eines solchen Programms, ist da Summieren der Kleidungsstücke sowie das Finden der maximalen und minimalen Summe sicher in möglich, wenn n die Anzahl der Sorten und m die Anzahl der Stile ist. Da das Verteilen jedes Kleidungsstückes in die Boxen anschließend auch in Polynomialzeit lösbar ist, ist die Gültigkeit des Lemmas gezeigt

Ein Bild, das Text, Diagramm, Screenshot enthält.

Automatisch generierte BeschreibungDa wegen Lemma 1.1 somit Lösungen, die von den Sortensummen kompletter Graphen abhängen schnell in Päckchen umwandeln kann, wollen wir im Folgenden nur alle maximalen komplette Teilgraphen, auch genannt Cliquen, im Graphen betrachten und mit deren Summen der jeweiligen Sorten arbeiten. Es ist dabei zu beachten, dass es in einem Graphen Knoten gibt, welche Teil mehrerer maximalen Cliquen sind (siehe Abb.1), sodass es unsere Aufgabe es ist, festzulegen, welcher Anteil der Kleidung dieses Knotens jeweils in welche Clique fließt. Die Ausgabe, die wir also brauchen, würde daraus bestehen, wie viele Kleidungsstücke aus jeder maximalen Clique zu Packungen umgewandelt werden sollen und aus welchen Knoten diese stammt, wenn die maximale Clique Knoten mit anderen maximalen Cliquen gemeinsam hat.

Abb.1 : Maximale Cliquen eines Graphen. Die Knoten 3 und 4 sind dabei Teil von zwei Cliquen

## Ganzzahlige Lineare Optimierung mit Enumeration maximaler Cliquen

Der folgende Lösungsansatz nutzt Lemma 1.1 und betrachtet die Summen der Kleidungsstücke in den maximalen Cliquen in dem Graphen. Dafür nutzen wir drei Schritte:

* Enumeration aller maximalen Cliquen im Graph G und Finden von „doppelten Knoten“
* Definieren eines Linearen Programmes und Ganzahlige Lineare Optimierung
* Umwandlung der Lösung des linearen Programms zu Päckchen

### Enumeration maximaler Cliquen und Finden „doppelter Knoten“

Um alle maximalen Cliquen in unserem Graphen zu enumerieren, bedienen wir uns hierbei am Bron-Kerbosch-Algorithmus.[[1]](#footnote-1) Dieses stellt einen rekursiven Backtracking-Ansatz dar, mit welchem die maximalen Cliquen in einem ungerichteten Graph gefunden werden können. Anschließend iterieren wir durch die Knoten und ermitteln, ob ein Knoten in mehr als einer maximalen Clique vorkommt. Diese markieren wir dann als „doppelte Knoten“ und müssen besonders berücksichtigt werden.

### Formulierung des Ganzzahligen Linearen Programms

Zunächst ermitteln wir die Summe der Kleidungsstücke in den Cliquen ohne „doppelten Knoten“ und halten diese als fest, wobei eine in 1.2.1 ermittelte maximale Clique ist und s eine Sorte ist. Wir formulieren nun das Problem als ganzzahliges lineares Programm so, dass die Variablen jeweils die Sortensummen der jeweiligen Cliquen beziehungsweise bei „doppelten Knoten“ diese der Anteil jeder Sorte ist, die dem benachbarten Knoten zugeordnet werden soll. Dabei heißt die Anzahl der Sorte s in der Clique ohne doppelte Knoten und die Anzahl der Kleidung der Sorte s im Knoten V G, welche der Clique c zugeordnet wird, . Alle Variablen sind dabei positive ganze Zahlen oder 0. Das lineare Programm sieht dann so aus:

sodass:

für alle und

für alle und alle paarweise verschiedene .

für alle , die „doppelt“ sind und

Mit der Notation ist hierbei gemeint, dass alle Knoten V, die in der Clique c sind, betrachtet werden. In der Zielfunktion werden alle betrachteten Variablen, deren Summe die Anzahl packbarer Kleidungsstücke repräsentiert, dargestellt. Die ersten zwei Einschränkungen, welche aus vielen Ungleichungen bestehen, wobei die Anzahl der doppelten Knoten im Graphen darstellt, schränken die Anzahl möglicher Kleidungsstücke nach der Eingabe in die Stile-Tabelle nach oben ein. Die dritte Einschränkung beschreibt für jede Clique mit , also insgesamt mit Ungleichungen dann die in Lemma 1.1 geforderten Verhältnisse der Summen, damit dieses angewandt werden kann.

**Lemma 1.2.2.1:** Die Gesamtanzahl der Lösungen des linearen Programms übersteigt nicht die Gesamtzahl der verfügbaren Kleidungsstücke von jedem Knoten.

*Beweis.* Folgt sofort aus den durch die Einschränkungen 1 und 2 aufgestellten Ungleichungen, die besagen, dass die Summen der entsprechenden Variablen kleiner gleich die gegebene Anzahl der Kleidungsstücke sein müssen.

**Lemma 1.2.2.2:** Die Summen der Variablen für jede Clique sind so, dass keine Summe mehr als dreimal größer ist als die kleinste Summe.

*Beweis.* Da keine Summe größer als drei Mal so groß sein darf wie die kleinste Summe, ist sie auch kleiner als drei Mal so groß wie jede andere Summe. Seien die Summen der *n* Sorten einer Clique. Wenn jede Summe nun kleiner gleich das Dreifache einer anderen Summe ist, also , ist der Satz erfüllt. Da dies in der dritten Einschränkung für jede Clique gesichert ist, ist die Behauptung gezeigt.

**Satz 1.2.2:** Die optimale Lösung des oben vorgestellten linearen Programms repräsentiert die optimale legitime Verteilung der Kleidungsstücke nach maximalen Cliquen.

*Beweis.* Folgt unmittelbar aus Lemma 1.2.2.1 und Lemma 1.2.2.2.

### 1.2.3 Laufzeitbetrachtung

Wir erkennen, dass die Enumeration aller maximalen Cliquen in durchführbar ist[[2]](#footnote-2), wobei *n* die Anzahl der Stile im Graphen ist. Die Laufzeit des ILP-Optimierungsprozesses ist dabei , wobei R die Anzahl der Terme und N die Anzahl der Variablen sind. Der Packprozess an sich gliedert sich durch die Anzahl der Kleider und die Anzahl der erstellten Boxen. Wir müssen jedes Kleidungsstück betrachten und diese in eine Box packen. Wenn also r die Anzahl der Kleidungsstücke ist, ist die Laufzeit des Programms nicht höher als .

Zusammenfassend ist also die geschätzte Laufzeit des Programms , was einer exponentiellen Laufzeit von bei sehr großen Angaben gleichkommt, weil diese Komponente der Laufzeit am schnellsten wächst.

# Analyse des Problems

## Komplexitätsbetrachtung

Wie schon im oberen Text angedeutet, gehen wir von einer NP-schwere, genauer von einer NP-Vollständigkeit des Problems aus. Im Folgenden soll ein Beweis hierfür erbracht werden, indem wir zeigen, dass das Problem in der Klasse NP liegt und anschließend einen Beweis für die NP-schwere durch eine Polynomialzeitreduktion von 3-CNF-SAT[[3]](#footnote-3) auf dieses Problem durchführen. Dabei sei die Formale Sprache des korrespondierenden Entscheidungsproblems:

PPP = {〈G, T〉 : G = (V, E) ist ein ungerichteter Graph,

r ist die Anzahl der Sorten,

k ist eine Anzahl an Päckchen,

P ist die Menge der Packungen,

T ist die Menge der r-Tupel für jeden Knoten

und alle Elemente aus sind positive ganze Zahlen, und

es existieren mindestens k *r* – Tupel , welche die Kleidungsstücke aus T eindeutig abbilden und für diese die entsprechenden Regeln zum Päckchenpacken erfüllt sind.

}

**Lemma 2.1.1:** Das Päckchen-Pack-Problem befindet sich in der Komplexitätsklasse NP.

*Beweis*. Die zu zeigende Behauptung ist äquivalent zu einer Annahme, dass ein Polynomialzeitalgorithmus zur Verifikation eines Zertifikats einer entsprechenden Instanz von PPP vorliegt.[[4]](#footnote-4) Es sei dafür das Zertifikat die Menge P, welche alle Packungen zu einem Graphen beinhaltet. Der Algorithmus prüft nun, ob für jedes Kleidungsstück aus den Tupeln höchstens ein Kleidungsstück bzw. Eintrag in existiert und überprüft für jedes *p*, ob diese auch die gegebenen Regeln (minimale und maximale Anzahl der Kleidungsstücke, zusammenpassen der Stile, gekennzeichnet durch die Kanten im Graphen G) erfüllen. Da ein solcher Algorithmus in Polynomialzeit läuft, ist die Behauptung gezeigt.

**Lemma 2.1.2:** Das Päckchen-Pack-Problem ist NP-schwer.

*Beweis.* Um die NP-schwere des PPP zu zeigen, führen wir eine Polynomialzeitreduktion von 3-SAT auf PPP durch und zeigen, dass ein Päckchen derart gepackt werden kann, dass diese eine Lösung für die gegebene Instanz von 3-CNF-SAT darstellt. Dazu sei eine Instanz von 3-CNF-SAT mit s Klauseln und t Variablen gegeben. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass in jeder Klausel nicht eine Variable und ihre Negation gleichzeitig vorzufinden ist, weil diese Klausel ansonsten in jedem Fall wahr ist und wir diese nicht beachten müssen. Nun konstruieren wir für jede der r Variablen genau zwei Knoten in G und entsprechende Tupel in T, welche jeweils zwei Stile repräsentieren und benennen diese nach den Variablen. Für eine Variable konstruieren wir also beispielsweise zwei Stile und . Für jede der s Klausel erstellen wir eine Sorte und legen damit fest, dass die Tupel in T s-Tupel sind. Zudem erstellen wir pro Klausel drei Kleidungsstücke. Die Kleidungsstücke verteilen wir nun den Variablen in den jeweiligen Klauseln entsprechend in den korrespondierenden Stilen von T, wobei die Klausel an sich die Sorte des Kleidungsstücks bestimmt. Ist beispielsweise die Klausel , so addieren wir Kleidungsstücke der Sorte 1 mit den Stilen , und zu T. Zuletzt legen wir fest, dass jeder Stil mit allen anderen Stilen außer der eigenen Komplementärvariable zusammenpasst. Der Stil würde also mit allen Variablen außer zusammenpassen.

Nun zeigen wir, dass eine Instanz von 3-CNF-SAT dann wahr ist, wenn eine derart konstruierte Instanz von PPP mit k = 1 wahr ist, und dann falsch ist, wenn die PPP-Instanz falsch ist. Wenn PPP zur gegebenen Instanz mindestens ein Päckchen packen kann, gilt dann nach den Regeln zu den Päckchen:

1. Mindestens ein Stil und höchstens drei Stile pro Klausel kann gepackt werden, dies korrespondiert damit, dass in der entsprechenden 3-CNF-SAT-Instanz pro Klausel mindestens eine Variable und höchstens drei Variablen Wahr sind.
2. In jeder Packung ist von den paarweise konstruierten Stilen höchstens eins der zwei Stile vorhanden, da diese nicht miteinander kombinierbar sind. Es kann also beispielsweise nur oder in der Packung enthalten sein. Dies korrespondiert damit, dass in der entsprechenden 3-CNF-SAT-Instanz jede Variable nur wahr oder falsch sein kann. Taucht ein Stil und sein Komplementärstil in der Packung garnicht auf, gilt für die 3-CNF-SAT-Instanz, dass diese Variable beliebig wahr oder falsch sein kann.

Da die zwei Bedingungen hinreichend sind für die Erfüllbarkeit der korrespondierenden 3-CNF-SAT-Instanz, ist gezeigt, dass diese dann wahr ist, wenn PPP wahr ist. Eine analoge Begründung sagt zudem aus, dass die 3-CNF-SAT entsprechend dann falsch ist, wenn PPP falsch ist. Damit ist die oben genannte Behauptung gezeigt.

Da der Reduktionsalgorithmus zunächst in Polynomialzeit G (inklusive seiner Kanten) und T erstellt und anschließend in Polynomialzeit anhand der 3-CNF-SAT-Instanz die -Tupel aus T füllt, ist die asymptotische Laufzeit polynomial. Daraus ergibt sich, dass 3-SAT in Polynomialzeit auf PPP reduzierbar ist und somit aufgrund der NP-schwere von 3-CNF-SAT auch NP-schwer ist.

**Satz 2.1:** Das Päckchen-Pack-Problem (PPP) ist NP-vollständig.

*Beweis.* Folgt direkt aus den Lemmata 2.1.1 und 2.1.2.

# Mögliche Erweiterungen

## Festlegung des Start- und Zielknotens

„Der Pilot hat bemerkt, dass er oftmals einen langen Umweg machen muss, um zum Startknoten des Pfades zu gelangen. Daher möchte er nun den Start- und Endknoten festlegen können.“ Um unseren Algorithmus der linearen Optimierung auf dieses Problem anzupassen, legen wir einfach in weiteren Beschränkungen fest, dass die Hilfsvariable, assoziiert mit dem gewählten Startknoten immer die Zahl 1 und die Hilfsvariable, assoziiert mit dem gewählten Endknoten, immer n als Wert besitzt. Mit dieser Methode können wir auch festlegen, welchen Knoten wir beispielsweise als zweites besuchen wollen. Im Fall unseres Greedy Algorithmus mit iterativer Suche, liegt die Lösung auch nahe. Wir versuchen einfach nicht mehr von jedem Knoten aus, mithilfe des

1. [https://www.math.leidenuniv.nl/~hwl/PUBLICATIONS/1983i/art.pdf](http://www.math.leidenuniv.nl/~hwl/PUBLICATIONS/1983i/art.pdf)
2. https://en.wikipedia.org/wiki/Integer\_programming

# Implementierung

Die Lösungsidee implementieren wir in Python mit der Pulp-Bibliothek, welche einen ILP-solver mit Optimierungen bereitstellt, um das lineare Programm zu lösen. Für die Darstellung des Graphen und der entsprechenden Tupeln nutzen wir dabei ein Dictionary, die für jeden Knoten mit Nummer als Schlüssel die Liste [[liste\_an\_benachbarten\_knoten],[tupeln\_der\_knoten]] enthält. Die Programme sind nach ihren Aufgaben benannt und enthalten am Anfang eine Kurzbeschreibung ihrer Funktionalität.(TODO?)

# Beispiele

Die folgende Tabelle fasst die Ergebnisse der verschiedenen Ansätze zusammen. In den Spalten ist die kürzeste gefundene Tour eingetragen. Wir gehen dabei in allen approximativen Ansätzen von einer ungefähren Laufzeit von 300s aus. In dem ILP Verfahren ist die Laufzeit angegeben. Das gleiche gilt für die untere Schranke. In dem ILP Verfahren wurde in einigen Beispielen das Verhalten des Solvers auf aggressiv gesetzt. Dies ist allerdings gleichzusetzen mit der Verwendung eines schnelleren Solvers und bestimmt daher nicht die Qualität der Ergebnisse. Der benötigte Arbeitsspeicher ist in allen Fällen gering und übersteigt nie wenige Gigabyte. Eine Verwendung der unteren Schranke, setzt eine Installation des GLPK Solvers vorraus.

Die betrachteten Ansätze sind die folgenden:

ILP unser optimaler Lösungsweg

G1 unser Greedy Algorithmus mit iterativer Suche

G2 unsere erste Kombination in der gültige Pfade optimiert werden, die einen neuen kürzesten gültigen Pfad formen

G3 unsere zweite Kombination, in der alle gültigen Pfade optimiert werden US unsere untere Schranke

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Eingabedatei | ILP | G1 | G2 | G3 | US |
| wenigerkrumm1 | 847.434 in 91s | 847.434 | 847.434 | 847.434 | 847.4342000 in 20.8s |
| wenigerkrumm2 | 2183.66 in 141s | 2183.66 | 2183.66 | 2183.66 | 2153.285600 in 4.2s |
| wenigerkrumm3 | 1848.05 in 548s | 1878.27 | 1878.27 | 1848.53 | 1815.649473 in 134.8s |
| wenigerkrumm4 | 1205.07 in 2s | 1205.07 | 1205.07 | 1205.07 | 1205.068140 in 0.1s |
| wenigerkrumm5 | 3257.92 in 53s | 3335.79 | 3317.75 | 3257.92 | 2729.794309 in 4.0s |
| wenigerkrumm6 | 3457.99 in 2871s | 3821.9 | 3757.81 | 3553.59 | 2992.783828 in 16.3s |
| wenigerkrumm7 | 4150.64 in 12173s | 4665.42 | 4660.58 | 4279.77 | 3663.550349 in 48.3s |

Die Programme liefen dabei auf einem Kern eines Intel i7-5960X.

Im Folgenden werden die kürzesten Ergebnispfade aller Dateien aufgeführt. Dabei bedeutet

-> , dass wir uns von einem Ort zum nächsten bewegen. Alle angegebenen Beispiele sind die Lösung des Solvers.

Wenigerkrumm1

(405|15) -> (400|0) -> (390|0) -> (380|0) -> (370|0) -> (360|0) -> (350|0) -> (340|0) -> (330|0) ->

(320|0) -> (310|0) -> (300|0) -> (290|0) -> (280|0) -> (270|0) -> (260|0) -> (250|0) -> (240|0) ->

(230|0) -> (220|0) -> (210|0) -> (200|0) -> (190|0) -> (180|0) -> (170|0) -> (160|0) -> (150|0) ->

(140|0) -> (130|0) -> (120|0) -> (110|0) -> (100|0) -> (90|0) -> (80|0) -> (70|0) -> (60|0) -> (50|0) ->

(40|0) -> (30|0) -> (20|0) -> (10|0) -> (0|0) -> (-5|15) -> (0|30) -> (10|30) -> (20|30) -> (30|30) ->

(40|30) -> (50|30) -> (60|30) -> (70|30) -> (80|30) -> (90|30) -> (100|30) -> (110|30) -> (120|30) -> (130|30) -> (140|30) -> (150|30) -> (160|30) -> (170|30) -> (180|30) -> (190|30) -> (200|30) -> (210| 30) -> (220|30) -> (230|30) -> (240|30) -> (250|30) -> (260|30) -> (270|30) -> (280|30) -> (290|30) -

> (300|30) -> (310|30) -> (320|30) -> (330|30) -> (340|30) -> (350|30) -> (360|30) -> (370|30) -> (380|30) -> (390|30) -> (400|30)

Laenge: 847.434

Wenigerkrumm2

(173.205|-100) -> (190.211|-61.8034) -> (198.904|-20.9057) -> (198.904|20.9057) -> (190.211| 61.8034) -> (173.205|100) -> (148.629|133.826) -> (117.557|161.803) -> (81.3473|182.709) -> (41.5823|195.63) -> (0|200) -> (-41.5823|195.63) -> (-81.3473|182.709) -> (-117.557|161.803) -> (- 148.629|133.826) -> (-173.205|100) -> (-190.211|61.8034) -> (-198.904|20.9057) -> (-198.904|-

20.9057) -> (-190.211|-61.8034) -> (-173.205|-100) -> (-148.629|-133.826) -> (-117.557|-161.803) -

> (-81.3473|-182.709) -> (-41.5823|-195.63) -> (0|-200) -> (41.5823|-195.63) -> (81.3473|-182.709)

-> (117.557|-161.803) -> (148.629|-133.826) -> (129.904|-75) -> (142.658|-46.3525) -> (149.178|- 15.6793) -> (149.178|15.6793) -> (142.658|46.3525) -> (129.904|75) -> (111.472|100.37) -> (88.1678|121.353) -> (61.0105|137.032) -> (31.1868|146.722) -> (0|150) -> (-31.1868|146.722) -> (-61.0105|137.032) -> (-88.1678|121.353) -> (-111.472|100.37) -> (-129.904|75) -> (-142.658|

46.3525) -> (-149.178|15.6793) -> (-149.178|-15.6793) -> (-142.658|-46.3525) -> (-129.904|-75) ->

(-111.472|-100.37) -> (-88.1678|-121.353) -> (-61.0105|-137.032) -> (-31.1868|-146.722) -> (0|-

150) -> (31.1868|-146.722) -> (61.0105|-137.032) -> (88.1678|-121.353) -> (111.472|-100.37)

Laenge: 2183.66

Wenigerkrumm3

(20.4382|91.6377) -> (20.4382|108.362) -> (23.9155|124.721) -> (30.718|140) -> (40.5484|153.53)

-> (59.4516|153.53) -> (69.282|140) -> (76.0845|124.721) -> (79.5618|108.362) -> (79.5618| 91.6377) -> (67.4611|73.0836) -> (52.9772|64.7214) -> (47.0228|64.7214) -> (40.5484|53.5304) -> (40.5484|46.4696) -> (47.0228|35.2786) -> (52.9772|35.2786) -> (67.4611|26.9164) -> (79.5618| 8.36228) -> (79.5618|-8.36228) -> (76.0845|-24.7214) -> (69.282|-40) -> (59.4516|-53.5304) ->

(40.5484|-53.5304) -> (30.718|-40) -> (23.9155|-24.7214) -> (20.4382|-8.36228) -> (20.4382| 8.36228) -> (16.6329|21.7482) -> (23.9155|24.7214) -> (32.5389|26.9164) -> (30.718|40) -> (30.718|60) -> (32.5389|73.0836) -> (23.9155|75.2786) -> (16.6329|78.2518) -> (0|80) -> (- 16.6329|78.2518) -> (-32.5389|73.0836) -> (-47.0228|64.7214) -> (-59.4516|53.5304) -> (-69.282|

60) -> (-76.0845|75.2786) -> (-79.5618|91.6377) -> (-79.5618|108.362) -> (-76.0845|124.721) -> (-

69.282|140) -> (-59.4516|153.53) -> (-47.0228|164.721) -> (-32.5389|173.084) -> (-16.6329|

178.252) -> (0|180) -> (16.6329|178.252) -> (32.5389|173.084) -> (47.0228|164.721) -> (52.9772|

164.721) -> (67.4611|173.084) -> (83.3671|178.252) -> (100|180) -> (116.633|178.252) -> (132.539|173.084) -> (147.023|164.721) -> (159.452|153.53) -> (169.282|140) -> (176.085| 124.721) -> (179.562|108.362) -> (179.562|91.6377) -> (176.085|75.2786) -> (169.282|60) -> (159.452|53.5304) -> (147.023|64.7214) -> (132.539|73.0836) -> (116.633|78.2518) -> (100|80) -> (83.3671|78.2518) -> (76.0845|75.2786) -> (69.282|60) -> (59.4516|53.5304) -> (59.4516|46.4696)

-> (69.282|40) -> (76.0845|24.7214) -> (83.3671|21.7482) -> (100|20) -> (116.633|21.7482) -> (132.539|26.9164) -> (147.023|35.2786) -> (159.452|46.4696) -> (169.282|40) -> (176.085| 24.7214) -> (179.562|8.36228) -> (179.562|-8.36228) -> (176.085|-24.7214) -> (169.282|-40) -> (159.452|-53.5304) -> (147.023|-64.7214) -> (132.539|-73.0836) -> (116.633|-78.2518) -> (100|-80)

-> (83.3671|-78.2518) -> (67.4611|-73.0836) -> (52.9772|-64.7214) -> (47.0228|-64.7214) ->

(32.5389|-73.0836) -> (16.6329|-78.2518) -> (0|-80) -> (-16.6329|-78.2518) -> (-32.5389|-73.0836)

-> (-47.0228|-64.7214) -> (-59.4516|-53.5304) -> (-69.282|-40) -> (-76.0845|-24.7214) -> (-

79.5618|-8.36228) -> (-79.5618|8.36228) -> (-76.0845|24.7214) -> (-69.282|40) -> (-59.4516|

46.4696) -> (-47.0228|35.2786) -> (-32.5389|26.9164) -> (-16.6329|21.7482) -> (0|20)

Laenge: 1848.05

Wenigerkrumm4

(-129.104|-155.042) -> (-82.8641|-104.174) -> (-98.7604|-81.7706) -> (-137.318|-20.1469) -> (-

191.717|-28.3605) -> (-221.15|-32.8625) -> (-239.848|8.6714) -> (-239.414|40.4271) -> (-240.369|

57.4261) -> (-219.149|103.685) -> (-154.088|115.023) -> (-119.026|168.454) -> (-107.989|185.174)

-> (20.2122|156.013) -> (33.3797|100.161) -> (28.9137|58.6999) -> (-20.9712|-5.63711) -> (- 16.7231|-12.6895) -> (51.0081|5.7696) -> (101.499|33.4842) -> (139.447|0.233238) -> (153.13|-

20.3609) -> (144.833|-43.4763) -> (94.7899|-67.0877) -> (42.1378|-60.3199)

Laenge: 1205.07

Wenigerkrumm5

(267.846|127.627) -> (244.229|119.193) -> (171.596|135.521) -> (116.703|132.022) -> (63.5416| 55.1402) -> (62.3667|50.7139) -> (45.1237|31.7402) -> (25.0982|35.2055) -> (-8.93692|13.5439) -> (-70.1835|73.7383) -> (-86.4576|105.836) -> (-82.1735|119.466) -> (-68.4462|137.179) -> (27.7063| 169.284) -> (36.5998|147.885) -> (47.0405|141.207) -> (51.4177|146.418) -> (38.6547|188.609) -> (-30.9914|186.807) -> (-44.6699|170.088) -> (-49.4474|173.211) -> (-104.782|158.212) -> (- 177.686|158.266) -> (-223.04|171.558) -> (-258.869|166.669) -> (-280.008|11.6578) -> (-286.024|-

55.9552) -> (-284.548|-60.155) -> (-278.41|-111.914) -> (-281.679|-187.718) -> (-260.478|-

196.956) -> (-251.657|-195.19) -> (-214.362|-193.265) -> (-116.832|-191.381) -> (-41.263|-

144.118) -> (-57.2662|-115.738) -> (-58.6842|-76.9889) -> (-31.5486|-55.2239) -> (-74.6394|

25.5429) -> (-81.3844|32.3683) -> (-95.6218|77.4685) -> (-136.787|79.5017) -> (-235.099|47.8103)

-> (-247.341|-160.278) -> (-234.711|-162.775) -> (-202.218|-178.736) -> (-147.023|-166.22) -> (26.4511|-192.813) -> (30.3668|-167.573) -> (90.5846|-164.218) -> (142.766|-118.682) -> (162.493|-84.574) -> (141.513|2.82114) -> (92.6399|22.216) -> (106.033|69.7549) -> (209.545| 94.2671) -> (239.64|79.4911) -> (253.535|38.015) -> (283.99|-101.866) -> (263.237|-144.293)

Laenge: 3257.92

Wenigerkrumm6

(-187.485|-177.031) -> (-191.216|-162.689) -> (-154.226|-135.522) -> (-107.197|-77.7926) -> (-

90.1602|-25.2008) -> (-72.5653|-24.2818) -> (-51.3438|-57.6548) -> (-4.59066|-40.0672) -> (19.7653|-99.2364) -> (64.9434|-119.784) -> (92.2558|-93.5141) -> (155.405|-56.4379) -> (221.029|-139.435) -> (216.826|-152.024) -> (157.589|-144.201) -> (143.114|-135.867) -> (134.693|-102.153) -> (150.526|-88.2301) -> (187.669|-122.656) -> (210.186|-127.404) -> (238.583|-133.144) -> (245.021|-167.449) -> (242.81|-182.054) -> (202.347|-189.07) -> (152.103|-

193.381) -> (46.6743|-193.09) -> (21.1766|-165.422) -> (-19.31|-131.811) -> (-31.7454|-69.208) -> (-18.5074|-22.9053) -> (14.0056|-14.0153) -> (40.3276|19.216) -> (58.7166|32.8359) -> (58.0197| 49.9379) -> (56.7165|66.9596) -> (20.2183|88.0312) -> (21.0674|122.165) -> (49.0919|150.679) -> (96.7817|141.371) -> (120.906|131.799) -> (132.794|135.681) -> (151.432|121.427) -> (175.678| 98.9293) -> (228.929|82.625) -> (231.945|82.9611) -> (246.622|101.706) -> (255.135|115.595) -> (277.822|104.263) -> (289.299|56.0513) -> (245.415|44.7941) -> (172.837|59.1842) -> (155.342|- 20.2528) -> (138.137|-31.3488) -> (112.833|-38.0576) -> (76.6471|-7.2897) -> (81.7404|10.2763) -

> (121.393|56.6941) -> (100.007|76.5793) -> (85.0438|108.946) -> (73.6898|110.224) -> (64.559| 82.5676) -> (83.0059|64.6468) -> (102.909|60.1079) -> (121.661|85.2677) -> (154.871|140.328) -> (102.223|174.202) -> (-55.0915|198.827) -> (-89.4538|162.237) -> (-100.569|140.809) -> (-175.118| 77.8426) -> (-194.987|101.364) -> (-167.995|138.195) -> (-97.3917|124.121) -> (-102.7|95.6321) ->

(-139.742|57.9367) -> (-126.57|-30.6452) -> (-144.888|-73.4954) -> (-189.988|-98.0439) -> (- 293.833|-165.44) -> (-288.744|-173.35)

Laenge: 3457.99

Wenigerkrumm7

(59.8272|-170.714) -> (92.298|-146.169) -> (217.599|-189.258) -> (235.827|-143.839) -> (275.793|- 129.415) -> (278.105|-93.7718) -> (276.277|-49.4487) -> (256.476|-46.5914) -> (224.599|-34.7985)

-> (205.718|-24.9765) -> (178.198|37.032) -> (158.742|62.6188) -> (162.923|117.466) -> (208.593| 136.618) -> (216.691|156.314) -> (217.219|164.295) -> (130.855|195.695) -> (95.9472|183.278) -> (100.044|161.295) -> (120.375|115.89) -> (126.8|112.727) -> (148.108|123.558) -> (169.99|154.26)

-> (170.514|161.17) -> (221.808|186.241) -> (271.301|142.524) -> (296.912|25.8116) -> (239.617|- 21.9442) -> (189.387|-4.46523) -> (158.552|-19.2544) -> (172.389|-53.1333) -> (118.99|-80.2036) -

> (88.8189|-42.8345) -> (55.5509|-45.09) -> (47.0114|-30.8872) -> (-0.200936|-21.9277) -> (- 9.58087|-17.5166) -> (-18.3161|27.7559) -> (-4.43492|33.1649) -> (3.15211|27.1039) -> (55.4348|

62.7292) -> (57.5554|64.4173) -> (56.3898|71.6185) -> (54.5519|71.5671) -> (40.8971|78.1523) -> (16.5732|104.021) -> (8.64366|135.907) -> (-84.6269|148.216) -> (-114.146|190.615) -> (-153.13| 187.817) -> (-192.681|174.523) -> (-201.485|155.275) -> (-200.771|147.741) -> (-157.423|126.8) -> (-134.985|132.945) -> (-120.387|170.589) -> (-13.0303|174.698) -> (34.079|187.731) -> (54.7665| 154.054) -> (53.4365|125.683) -> (74.8875|80.5865) -> (129.024|29.7017) -> (141.433|-6.0231) -> (135.781|-13.0534) -> (126.904|-80.7333) -> (106.599|-107.434) -> (68.9109|-82.1233) -> (34.9591|-106.842) -> (-17.3566|-125.254) -> (-47.2666|-66.984) -> (-46.4031|-13.7558) -> (- 48.3544|9.09141) -> (-42.7047|37.6795) -> (-27.912|48.3267) -> (-56.9145|92.5012) -> (-126.569|

106.965) -> (-147.363|59.6082) -> (-171.355|25.4631) -> (-184.093|5.73728) -> (-226.788|2.65886)

-> (-248.169|80.1322) -> (-284.129|107.253) -> (-287.058|113.6) -> (-268.739|143.276) -> (- 240.249|179.335) -> (-222.492|169.033) -> (-215.114|120.741) -> (-189.062|104.286) -> (-185.649|

90.1445) -> (-207.665|81.4104) -> (-211.429|27.7704) -> (-217.282|-43.3166) -> (-244.96|-111.047)

-> (-189.135|-139.079) -> (-155.652|-138.152) -> (-133.731|-113.306) -> (-152.13|-93.8443) -> (- 172.378|-88.2982) -> (-202.829|-101.7) -> (-181.209|-192.623)

Laenge: 4150.64

#### Die hier ausgegebenen Lösungen befinden sich auch in einem Ordner in der Abgabe. Darin finden sich auch weitere Beispiele, die die Funktionalität der Erweiterungen beweisen.

# Quellcode

In dem tatsächlichen Quellcode wird vor jeder Funktion nochmal ausführlich auf deren Funktionalitäten und Parameter eingegangen. Dies hätte allerdings komplett den Rahmen von 10 Seiten Quellcode gesprengt und wird daher weggelassen. Das gleiche gilt für den Quellcode der Erweiterungen, der in dem C++ Header zu finden ist.

#linear programming ansatz.

import pulp as pl

import numpy as np

file = open("paeckchen5.txt", "r")

content = file.readlines()

file.close()

#print(content)

s,r = [eval(i) for i in content[0].split()]

passend = []

stopp\_zeile = 0

for i in content:

    if i == "\n":

        stopp\_zeile = content.index(i)

for i in range(1, stopp\_zeile):

    passend.append([eval(j) for j in content[i].split()])

print("s, r:", s,r)

##wir erstellen ein dictionary.

stile\_graph = {}

#datensruktur: name\_des\_knotens: [benachbarte\_knoten][anzahl der kleidungsstücke]

for i in range(r):

    stile\_graph[i+1] =[[],[0 for \_ in range(s)]]

# einfügen der benachbarte knoten

for a, b in passend:

    stile\_graph[a][0].append(b)

    stile\_graph[b][0].append(a)#knoten sind immer gegenseitig

for i in content[stopp\_zeile+1:]:

    sorte\_i, stil\_i, anzahl\_i = [eval(j) for j in i.split()]

    #print(i, content\_i)

    stile\_graph[stil\_i][1][sorte\_i-1] = anzahl\_i

    #tabelle[content\_i[0]-1][content\_i[1]-1] = content\_i[2]

print(stile\_graph)

def bron\_kerbosch(graph):

    def bron\_kerbosch\_recursive(R, P, X):

        if not P and not X:

            cliques.append(R)

            return

        for v in list(P):

            bron\_kerbosch\_recursive(R.union({v}), P.intersection(graph[v][0]), X.intersection(graph[v][0]))

            P.remove(v)

            X.add(v)

    cliques = []

    knoten = set(graph.keys())

    bron\_kerbosch\_recursive(set(), knoten, set())

    return cliques

def finde\_einfache\_knoten(cliquen):

    knoten = []

    doppelte\_knoten = []

    for clique in cliquen:

        for k in clique:

            if k not in knoten:

                knoten.append(k)

            elif k not in doppelte\_knoten:

                doppelte\_knoten.append(k)

            else:

                pass

    return list(set(knoten).difference(set(doppelte\_knoten))), doppelte\_knoten

def summe\_der\_sorten(clique, doppelt = True):#checkt ob eine Kombination funktionieren kann oder nicht. Eine Kombination funktioneirt dann NICHT, wenn max(Summe der knoten, die einem Clique angehören) 3 mal größer ist als min (summer aller knoten der cliquen für jeden knoten.)

    summe = [0 for \_ in range(s)]

    if doppelt:

        for k in clique:#k steht für knoten

            summe = [sum(x) for x in zip(stile\_graph[k][1], summe)]##hmm gucken ob das optimale geschwindigkeit ist?

        return summe

    else:

        for k in clique:

            if k in einfache\_knoten:

                summe = [sum(x) for x in zip(stile\_graph[k][1], summe)]##hmm gucken ob das optimale geschwindigkeit ist?

        return summe

cliquen = bron\_kerbosch(stile\_graph)

print("Cliquen: ", cliquen)

einfache\_knoten, doppelte\_knoten = finde\_einfache\_knoten(cliquen)

print("Einfache Knoten in Cliquen:", einfache\_knoten)

summen\_mit\_doppelknoten = []

summen\_ohne\_doppelknoten = []#brauchen wir glaub ich nicht

for clique in cliquen:

    summen\_mit\_doppelknoten.append(summe\_der\_sorten(clique, doppelt = True))

    summen\_ohne\_doppelknoten.append(summe\_der\_sorten(clique, doppelt = False))

print("summen mit und ohne Doppelknoten:")

print(summen\_mit\_doppelknoten)

print(summen\_ohne\_doppelknoten)

#erstelle Problem

problem = pl.LpProblem("StilvollePackungen", pl.LpMaximize)

#erstelle Entscheidungsvariablen

cliquenvariablen = []

for i in range(len(cliquen)):

    stile = []

    for j in range(s):

        stile.append(pl.LpVariable(f"clique{i}\_stil\_{j}", lowBound = 0, cat = 'Integer'))

    cliquenvariablen.append(stile)

print("Cliquenvariablen: ", cliquenvariablen)

doppelknotenvariablen = []

for i, clique in enumerate(cliquen):

    c = []#vorläufige liste aller variablen der Doppelknoten der Clique

    for count, k in enumerate(doppelte\_knoten):

        if k in clique:

            stile = []

            for j in range(s):

                stile.append(pl.LpVariable(f"doppelknoten\_clique{i}\_knoten{k}\_stil\_{j}", lowBound = 0, cat = 'Integer'))

            c.append(stile)

        else:

            c.append(None)

    doppelknotenvariablen.append(c)

print("Doppelknotenvariablen: ", doppelknotenvariablen)

#maximierungsgleichung hinzufügen:

gleichung = None

for clique in doppelknotenvariablen:

    for knoten in clique:

        if knoten:

            for var in knoten:

                gleichung += var

for clique in cliquenvariablen:

    for var in clique:

        gleichung += var

problem += gleichung

#ungleichungen hinzufügen

for count, clique in enumerate(cliquen):

    #die summe darf die summe ohne doppelte knoten nicht übersteigen.

    for sorte in range(s):

        problem += cliquenvariablen[count][sorte] <= summen\_ohne\_doppelknoten[count][sorte]

for count, k in enumerate(doppelte\_knoten):

    #die summe der doppelten knoten muss immer zueinander passen.

    for sorte in range(s):

        linke\_seite = None

        for clique, cliquenknoten in enumerate(doppelknotenvariablen):#fügt alle doppelknoten innerhalb einer Clique

            if cliquenknoten[count]:

                linke\_seite += cliquenknoten[count][sorte]

        #print(linke\_seite)

        problem += linke\_seite <= stile\_graph[k][1][sorte]

for count, clique in enumerate(cliquen):#die summe der gepackten variablen dürfen nicht größer als die dreifache Summe der korrespondierenden Summen der anderen Variablen sein

    for stila in range(s):

        for stilb in range(s):

            if stila == stilb:

                continue#wir vermeiden selbstschleifen, dort ist die zu zeigende Ungleichugn trivial

            linke\_seite = None

            linke\_seite += cliquenvariablen[count][stila]

            linke\_seite -= 3\*cliquenvariablen[count][stilb]

            for k in clique:

                if k in doppelte\_knoten and doppelknotenvariablen[count][doppelte\_knoten.index(k)]:

                    linke\_seite += doppelknotenvariablen[count][doppelte\_knoten.index(k)][stila]

                    linke\_seite -= 3\*doppelknotenvariablen[count][doppelte\_knoten.index(k)][stilb]

            problem += linke\_seite <= 0

#löse das Problem

problem.solve()

summe\_der\_kleidung = 0

print("Status:", pl.LpStatus[problem.status])

print("Doppelknotenvariablen:")

for clique in doppelknotenvariablen:

    for knoten in clique:

        if knoten:

            for var in knoten:

                print(f"{var.name}: {var.value()}")

                summe\_der\_kleidung += var.value()

print("cliquenvariablen:")

for clique in cliquenvariablen:

    for var in clique:

        print(f"{var.name}: {var.value()}")

        summe\_der\_kleidung += var.value()

print("gepackte Kleidungsstücke:     ", summe\_der\_kleidung)

summe\_total = 0

for i in stile\_graph:

    for j in stile\_graph[i][1]:

        summe\_total += j

print("tatsächliche Kleidungsstücke: ", summe\_total)

print("Übrige Kleidungsstücke: ", summe\_total-summe\_der\_kleidung)

for clique in cliquen:

    print("Clique: ", clique)

boxenverteilung = []

for count in range(len(cliquen)):

    boxenverteilung.append([int(cliquenvariablen[count][i].value()+sum([doppelknotenvariablen[count][j][i].value() for j in range(len(doppelte\_knoten)) if doppelknotenvariablen[count][j]])) for i in range(s)])

    print(boxenverteilung[-1])

#tatsächliche Illustration der Boxen:

packungen = []

# Disclaimer: die Boxen werden nicht exakt nach den lösungen gepackt, weil einzelknoten kleidungen bevorzugt werden.

# das ergebnis bleibt aber gleich, es blebt nur häufiger was in doppelknoten was übrig.

for count, clique in enumerate(cliquen):

    packung\_clique = []

    boxen\_anzahl = min(boxenverteilung[count])

    min\_index = boxenverteilung[count].index(boxen\_anzahl)

    for i in range(boxen\_anzahl):

        packung\_clique.append([[] for \_ in range(s)])#erstellt dummy boxen die alle gepackt werden müssen

    #nimmt aus den ganzen doppelten knoten die zugeteilten Kleider und packt diese in die Boxen

    for sorte in range(s):

        box\_counter = 0

        zu\_packende\_kleidung = 0

        for k in range(len(doppelte\_knoten)):#für alle doppelten knoten

            if doppelknotenvariablen[count][k]:

                zu\_packende\_kleidung += doppelknotenvariablen[count][k][sorte].value()

        for k, knoten in enumerate(doppelte\_knoten):#für alle doppelten knoten

            if zu\_packende\_kleidung == 0:

                break

            if doppelknotenvariablen[count][k]:

                for i in range(int(doppelknotenvariablen[count][k][sorte].value())):#für jede zu packende kleidung:

                    zu\_packende\_kleidung -= 1

                    stile\_graph[knoten][1][sorte] -= 1

                    packung\_clique[box\_counter%boxen\_anzahl][sorte].append([k, sorte])

                    box\_counter += 1

    #füllt in jede box 1 im minimum index (jetzt für auf jede sorte erweitern und

    # wir müssten fertig sein)

    for sorte in range(s):

        box\_counter = 0

        for k in list(set(clique).difference(set(doppelte\_knoten))):

            if boxenverteilung[count][sorte] == 0:#die sorte ist fertig!

                break

            while stile\_graph[k][1][sorte] > 0:#die sorte ist fertig!

                if boxenverteilung[count][sorte] == 0:

                    break

                stile\_graph[k][1][sorte] -= 1

                packung\_clique[box\_counter%boxen\_anzahl][sorte].append([k, sorte])#packt das kleidungsstück in eine richtige box

                boxenverteilung[count][sorte] -= 1#nimmt eins weg von den zu packenden kleidungsstücken

                box\_counter += 1

    packungen.append(packung\_clique)

    print("Packung clique: ")

    for sublist in packung\_clique:

        print(\*sublist)

    print(stile\_graph)

summe\_naher = 0

for i in stile\_graph:

    for j in stile\_graph[i][1]:

        summe\_naher += j

print(summe\_naher)

1. https://en.wikipedia.org/wiki/Bron%E2%80%93Kerbosch\_algorithm [↑](#footnote-ref-1)
2. https://en.wikipedia.org/wiki/Bron%E2%80%93Kerbosch\_algorithm [↑](#footnote-ref-2)
3. https://de.wikipedia.org/wiki/3-SAT [↑](#footnote-ref-3)
4. „Introduction to algorithms“, second edition, S.981 [↑](#footnote-ref-4)