Bundeswettbewerb Informatik

Lösungshinweise

Allgemeines

Es ist immer wieder bewundernswert, wie viel an Ideen, Wissen, Fleiß und Durchhaltevermögen in den Einsendungen zur 2. Runde eines Bundeswettbewerbs Informatik stecken. Um aber die Allerbesten für die Endrunde zu bestimmen, müssen wir die Arbeiten kritisch begutachten und hohe Anforderungen stellen. Deswegen sind Punktabzüge die Regel und Bewertungen mit Pluspunkten über die Erwartungen hinaus die Ausnahme.

Spannende bzw. schwierige Erweiterungen der Aufgabenstellung sind Extrapunkte wert, wenn sie auch praktisch realisiert wurden. Weitere Ideen ohne Implementierung und geringe Verbesserungen der bereits implementierten Lösung einer Aufgabe gelten allerdings nicht als geeignete Erweiterungen. Intensive theoretische Überlegungen wie z. B. ein korrekter Beweis zur Komplexität des Problems werden ebenfalls mit zusätzlichen Punkten belohnt.

Falls Ihre Einsendung nicht herausragend bewertet wurde, lassen Sie sich auf keinen Fall entmutigen! Allein durch die Arbeit an den Aufgaben und ihren Lösungen hat jede Teilnehmerin und jeder Teilnehmer viel gelernt; diesen Effekt sollten Sie nicht unterschätzen. Selbst wenn Sie die Lösung zu nur einer Aufgabe einreichen konnten, so kann die Bewertung Ihrer Einsendung bei der Anfertigung künftiger Lösungen hilfreich für Sie sein.

Bevor Sie sich in die Lösungshinweise vertiefen, lesen Sie bitte kurz die folgenden Anmerkungen zu den Einsendungen und beiliegenden Unterlagen durch.

Bewertungsbogen

Aus der 1. Runde oder auch aus früheren Wettbewerbsteilnahmen kennen Sie den Bewertungsbogen, der angibt, wie Ihre Einsendung die einzelnen Bewertungskriterien erfüllt hat. Auch in dieser Runde können Sie den Bewertungsbogen im Anmeldesystem AMS einsehen. In der 1. Runde ging die Bewertung noch von 5 Punkten aus, von denen bei Mängeln abgezogen werden konnte. In der 2. Runde geht die Bewertung von zwanzig Punkten aus, bei denen Punkte abgezogen und manchmal auch hinzuaddiert werden konnten. In dieser Runde gibt es auch deutlich mehr Bewertungskriterien als in der 1. Runde.

Terminlage

Für Abiturientinnen und Abiturienten ist der Terminkonflikt zwischen Abiturvorbereitung und 2. Runde sicher nicht ideal. Doch leider bleibt dem Bundeswettbewerb Informatik nur die erste Jahreshälfte für diese Runde: In der zweiten Jahreshälfte läuft nämlich die zweite Runde des

Mathematikwettbewerbs, dem wir keine Konkurrenz machen wollen. Aber: die Bearbeitungszeit für die Runde beträgt etwa vier Monate. Frühzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, zeitliche Engpässe am Ende der Bearbeitungszeit gerade mit der wichtigen, ausführlichen Dokumentation der Aufgabenlösungen zu vermeiden. Aufgaben der 2. Runde sind oft deutlich schwerer zu lösen, als sie auf den ersten Blick erscheinen. Erst bei der konkreten Umsetzung einer Lösungsidee stößt man manchmal auf Besonderheiten bzw. noch zu lösende Schwierigkeiten, was dann zusätzlicher Zeit bedarf. Daher ist es sinnvoll, die einzureichenden Aufgaben nicht nacheinander, sondern relativ gleichzeitig zu bearbeiten, um nicht vom zeitlichen Aufwand der jeweiligen Aufgabe kurz vor Ablauf der Bearbeitungszeit unangenehm überrascht zu werden.

Dokumentation

Es ist sehr gut nachvollziehbar, dass Sie Ihre Energie bevorzugt in die Lösung der Aufgaben, die Entwicklung Ihrer Ideen und deren Umsetzung in Software fließen lassen. Doch ohne eine verständliche Beschreibung der Lösungsideen und ihrer jeweiligen Umsetzung, eine übersichtliche Dokumentation der wichtigsten Komponenten Ihrer Programme, eine geeignete Kommentierung der Quellcodes und eine ausreichende Zahl sinnvoller Beispiele (welche die verschiedenen bei der Lösung des Problems zu berücksichtigenden Fälle abdecken) ist eine Einsendung nur wenig wert.

Bewerterinnen und Bewerter können die Qualität Ihrer Aufgabenlösungen nur anhand dieser Informationen vernünftig einschätzen. Mängel in der Dokumentation der Einsendung können nur selten durch Ausprobieren und Testen der Programme ausgeglichen werden – wenn die Programme denn überhaupt ausgeführt werden können: Hier gibt es gelegentlich Probleme, die meist vermieden werden könnten, wenn Lösungsprogramme vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner auf Lauffähigkeit getestet würden. Insgesamt sollte die Erstellung der Dokumentation die Programmierarbeit begleiten oder ihr teilweise sogar vorangehen: Wer nicht in der Lage ist, Idee und Modell präzise zu formulieren, bekommt auch keine saubere Umsetzung hin, in welcher Programmiersprache auch immer.

Einige unterhaltsame Formulierungsperlen sind im Anhang wiedergegeben.

Bewertung

Bei den im Folgenden beschriebenen Lösungsideen handelt es sich nicht um perfekte Musterlösungen, sondern um sinnvolle Lösungsvorschläge. Dies sind also nicht die einzigen Lösungswege, die wir gelten ließen. Wir akzeptieren vielmehr in der Regel alle Ansätze, auch ungewöhnliche, kreative Bearbeitungen, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind. Einige der Fußnoten in den folgenden Lösungsvorschlägen verweisen auf weiterführende Fachliteratur für besonders Interessierte; Lektüre und Verständnis solcher Literatur wurden von den Teilnehmenden natürlich nicht erwartet.

Unabhängig vom gewählten Lösungsweg gibt es aber Dinge, die auf jeden Fall von einer guten Lösung erwartet wurden. Zu jeder Aufgabe wird deshalb in einem eigenen Abschnitt, jeweils am Ende des Lösungsvorschlags erläutert, auf welche Kriterien bei der Bewertung dieser Aufgabe besonders geachtet wurde. Dabei können durchaus Anforderungen formuliert werden, die aus der Aufgabenstellung nicht hervorgingen. Letztlich dienen die Bewertungskriterien dazu,

die allerbesten unter den sehr vielen guten Einsendungen herauszufinden. Außerdem gibt es aufgabenunabhängig einige Anforderungen an die Dokumentation (klare Beschreibung der Lösungsidee, genügend aussagekräftige Beispiele und wesentliche Auszüge aus dem Quellcode) einschließlich einer theoretischen Analyse (geeignete Laufzeitüberlegungen bzw. eine Diskussion der Komplexität des Problems) sowie an den Quellcode der implementierten Software (mit übersichtlicher Programmstruktur und verständlicher Kommentierung) und an das lauffähige Programm (ohne Implementierungsfehler). Wünschenswert sind auch Hinweise auf die Grenzen des angewandten Verfahrens sowie sinnvolle Begründungen z. B. für Heuristiken, vorgenommene Vereinfachungen und Näherungen. Geeignete Abbildungen und eigene zusätzliche Eingaben können die Erläuterungen in der Dokumentation gut unterstützen. Die erhaltenen Ergebnisse für die Beispieleingaben (ggf. mit Angaben zur Rechenzeit) sollten leicht nachvollziehbar dargestellt sein, z. B. durch die Ausgabe von Zwischenschritten oder geeignete Visualisierungen. Eine Untersuchung der Skalierbarkeit des eingesetzten Algorithmus hinsichtlich des Umfangs der Eingabedaten ist oft ebenfalls nützlich.

Danksagung

Alle Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt: Peter Rossmanith (Vorsitz), Hanno Baehr, Jens Gallenbacher, Rainer Gemulla, Torben Hagerup, Christof Hanke, Thomas Kesselheim, Arno Pasternak, Holger Schlingloff, und Melanie Schmidt sowie (als Gäste) Wolfgang Pohl, Hannah Rauterberg und Greta Niemann.

An der Erstellung der im Folgenden skizzierten Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Personen mit: Lina Blume und Maxim Goldshteyn (Aufgabe 1), Tim Pokart und Hans-Martin Bartram (Aufgabe 2) sowie Shuheng Wei (Aufgabe 3). Allen Beteiligten sei für ihre Mitarbeit ganz herzlich gedankt.

Aufgabe 1: Transformers

1.1 Problemdefinition

Graphentheoretische Grundlagen

```
1 4 2
2 o-o....
3 |.|....
4 o.o-o-o
```

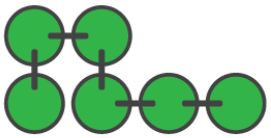


Abbildung 1.1: Eine beispielhafte Figur, die als Eingabe dienen könnte, rechts visualisiert in der typischen Darstellung in grünen Plättchen, die mit grauen Stiften verbunden sind.

Modellierung als Graph Wir abstrahieren über die konkrete Darstellung und betrachten die Eingabe des Problems als Graph $G = (V, E)$, wobei V die Menge der Knoten und $E \subseteq V^2$ die Menge der Kanten bezeichnet. Dabei entsprechen alle grünen Plättchen jeweils einem Knoten und alle grauen Stifte jeweils einer Kante, die immer genau zwei Knoten verbindet.

Koordinatensystem In der Mathematik ist es häufig hilfreich, ein Koordinatensystem einzuführen, um die Objekte benennen zu können, mit denen gearbeitet wird. Wir verwenden hier ein zweidimensionales Koordinatensystem von Punkten $(x \mid y)$, wobei x und y ganze Zahlen sind $(x, y \in \mathbb{Z})$. Zudem entscheiden wir uns dafür, den Knoten oben links mit dem Ursprung $(0 \mid 0)$ zu bezeichnen und folgen der Konvention, dass ein Schritt nach rechts einer Erhöhung von x um 1 und ein Schritt nach unten einer Erhöhung von y um 1 entspricht. [1]

Für Abbildung 1.1 gilt also:

$$V = \{(0 \mid 0), (0 \mid 1), (1 \mid 0), (1 \mid 1), (1 \mid 2), (1 \mid 3)\}$$
$$E = \{((0 \mid 0), (0 \mid 1)), ((1 \mid 1), (1 \mid 2)), ((1 \mid 2), (1 \mid 3)), ((0 \mid 0), (1 \mid 0)), ((0 \mid 1), (1 \mid 1)),$$
$$((0 \mid 1), (0 \mid 0)), ((1 \mid 2), (1 \mid 1)), ((1 \mid 3), (1 \mid 2)), ((1 \mid 0), (0 \mid 0)), ((1 \mid 1), (0 \mid 1))\}$$

Dabei werden die Kanten als Paare dargestellt, die je genau zwei Knoten als Elemente haben. Jede Kante hat dabei eine entsprechende Rückkante, denn der Graph ist ungerichtet, weil die Stäbe keine Richtung kennen.[2]

Translationsinvarianz Sobald ein Koordinatensystem festgelegt wurde, spielt es keine Rolle mehr, wie der Graph verschoben wird, solang alle Knoten und Kanten um einen festen Vektor $\begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}$ verändert werden. Diese Art von Verschiebung wird *Translation* genannt. Dass wir Graphen, die sich bis auf Translation nicht unterscheiden, gleich behandeln, bezeichnen wir als *Translationsinvarianz*.

[1] Im Endeffekt ist es nicht wichtig, wie genau das Koordinatensystem ausgelegt ist oder wo der Ursprung liegt, aber eine Möglichkeit zu haben, die Knoten zu benennen, ist notwendig für ein Verständnis des Problems.

[2] Präziser gesprochen modellieren wir jeden ungerichteten Graphen in diesem Lösungsvorschlag als gerichteten Graphen, wobei jede Kante (u, v) genau eine korrespondierende Rückkante (v, u) besitzt.

Beispielsweise könnte die obige Eingabe um $\begin{pmatrix} 3 \\ -2 \end{pmatrix}$ verschoben werden. Man erhielte die folgenden Mengen:

$$V' = \{(3 \mid -2), (3 \mid -1), (4 \mid -2), (4 \mid -1), (4 \mid 0), (4 \mid 1)\}$$
$$E' = \{((3 \mid -2), (3 \mid -1)), ((4 \mid -1), (4 \mid 0)), ((4 \mid 0), (4 \mid 1)), ((3 \mid -2), (4 \mid -2)),$$
$$((3 \mid -1), (4 \mid -1)), ((3 \mid -1), (3 \mid -2)), ((4 \mid 0), (4 \mid -1)), ((4 \mid 1), (4 \mid 0)),$$
$$((4 \mid -2), (3 \mid -2)), ((4 \mid -1), (3 \mid -1))\}$$

Diese müssen identisch zur bisherigen Eingabe behandelt werden, da in der physisch vorliegenden Figur kein Unterschied vorliegt.

Algorithmisch erfolgt die Renormalisierung durch Ermitteln der minimalen x-und y-Koordinate eines Knotens im Graphen, gefolgt von einer komponentenweisen Verschiebung aller Knoten um ihre Negation. Auf diese Art und Weise entsteht eine kanonische Repräsentation, wobei keine negativen Komponenten auftreten und zugleich die Komponentengröße minimiert wird.

Isomorphie Häufig werden in der Literatur Graphen nur „bis auf Isomorphie“ betrachtet. Das bedeutet, dass Graphen, die durch Umbenennung von Knoten (und entsprechende Anpassung der Kanten) auseinander hervorgehen, nicht unterschieden werden. Eine genauere Definition folgt in Abschnitt 1.2.

Für dieses Problem müssen isomorphe Graphen unterschieden werden, da z.B. die vorangegangenen Eingaben ein unterschiedliches Verhalten aufweisen. Später werden wir weitere Beispiele komplexerer Isomorphiebegriffe aufgreifen und zeigen, dass auch diese das Problem nicht ausreichend modellieren können.

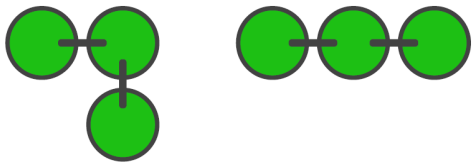


Abbildung 1.2: Zwei isomorphe Graphen $G_l \cong G_r$. Der rechte Graph $G_r = (V_r, E_r)$ entsteht aus dem linken $G_l = (V_l, E_l)$ indem man die Knoten so umbenennt:
$\varphi : V_l \to V_r, \varphi : (0 \mid 0) \mapsto (0 \mid 0),\ (0 \mid 1) \mapsto (0 \mid 1),\ (1 \mid 1) \mapsto (0 \mid 2)$.
Die oberen beiden Knoten bleiben erhalten, und der Knoten $(1 \mid 1)$ unten rechts des Graphen G_l wird in den Knoten $(0 \mid 2)$ ganz rechts umbenannt, da dieser mit den gleichen Knoten verbunden ist.

Eigenschaften des Graphen Wir halten einige Eigenschaften fest, die der so konstruierte Graph haben muss.

Zunächst muss es sich um einen Teilgraphen des Gittergraphen

$$L = (\mathbb{Z}^2, V_L := \{(u,v) \mid (u,v) \in \mathbb{Z}^2,\ \|u - v\|_1 = 1\})$$

handeln, wobei $\mathbb{Z}^2$ alle Punkte zweier ganzzahligen Koordinaten beschreibt, die Subtraktion komponentenweise definiert ist und die Norm $\|u\|_1$ eines Punkts $u = (x \mid y)$ als Summe der

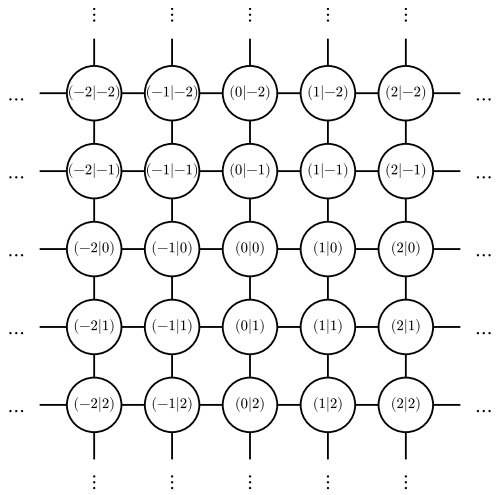


Abbildung 1.3: Der Gittergraph L.

Beträge der Komponenten $|x| + |y|$ definiert ist.[3] Abbildung 1.3 zeigt den unendlichen Graphen L visuell.

Die Eingabe muss ein endlicher Teilgraph von L sein, was bedeutet, dass alle Knoten genau zu einem Punkt gehören und jede Kante nur zwei horizontal oder vertikal adjazente Knoten verbinden muss (Dass die Kanten nur Knoten des Graphen verbinden dürfen, ergibt sich aus $E \subseteq V^2$).

Zusammenhängender Graph Ein *Kantenzug* der Länge $n \in \mathbb{N}_0$ ist eine Folge von Kanten $(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n), (v_n, v_{n+1}) \in E^n$. Wenn alle diese Kanten existieren, spricht man davon, dass der *Pfad* (der Länge $n+1$) $v_1, ..., v_{n+1}$ im Graphen existiert, d.h., v_{n+1} ist von v_1 aus durch wiederholtes Folgen von Kanten *erreichbar*. Ist in einem Graphen jeder Knoten von jedem anderen aus erreichbar, so heißt der Graph *zusammenhängend*. Wir betrachten im Folgenden nur zusammenhängende Graphen, auf die wir unsere Eingaben einschränken – unzusammenhängende Graphen sind keine gültigen Figuren, da sie sonst in zwei unabhängige Teile getrennt werden könnten, die nicht verbunden sind.

Zusammenhangskomponenten In einem beliebigen möglicherweise unzusammenhängenden, aber weiterhin ungerichteten Graphen $G = (V, E)$ betrachten wir nun drei Knoten $u, v, w \in V$. Wir schreiben $u \sim v$, falls v von u aus durch einen eben beschriebenen Pfad erreicht werden kann. Offensichtlich gilt $u \sim u$ (wähle den leeren Kantenzug), $u \sim v \Rightarrow v \sim u$ (denn alle Kanten sind auch Rückkanten) und $u \sim v \wedge v \sim w \Rightarrow u \sim w$, denn wenn v von u aus erreichbar ist und w von v aus, so kann auch w von u aus erreicht werden.

Also teilt die Relation $\sim$ alle Knoten in sogenannte *Zusammenhangskomponenten* auf, sozusagen die größten zusammenhängenden Teilgraphen. Jeder zusammenhängende Graph besteht nur aus einer einzigen Zusammenhangskomponente, die alle Knoten umfasst. Zusammenhangskomponenten sind stets nicht-leer, disjunkt (haben keine gemeinsamen Elemente) und jeder

[3] An dieser Stelle wird die sogenannte *Manhattan-Distanz* oder *1-Norm* verwendet. Man könnte genauso gut aber den auch als *Euklidische Distanz* (bzw. *2-Norm*) auch aus der Schule bekannten geometrischen Abstand $\sqrt{x^2 + y^2}$ im Koordinatensystem verwenden.

Knoten v ist in einer enthalten, die wir $Z_v \subseteq V$ nennen. Diese Begriffe sind wichtig, um die Elementaroperationen sorgfältig zu definieren.

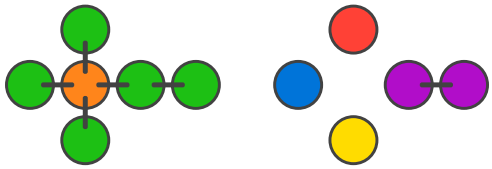


Abbildung 1.4: Falls beim linken Graphen der mittlere Knoten entfernt wird, so müssen 4 weitere Kanten entfernt werden. Dadurch hat der neu entstandene Graph rechts 4 Zusammenhangskomponenten, hier farblich hervorgehoben.

Brücken und Gelenkpunkte Sei $G = (V,E)$ mit $V \neq \emptyset$ ein Graph, und $r \in \mathbb{N}$ die Anzahl seiner Zusammenhangskomponenten. Wir betrachten eine Kante $e = (u,v) \in E$ mit $u,v \in V$. Mithilfe dieser Kante können wir einen neuen Graphen, hier genannt $G - e := G' := (V',E)$, konstruieren, wobei $V' = V$ und $E' = E \setminus \{e\}$ - also entfernen wir die Kante e aus dem Graphen, ohne etwas anderes zu verändern.

Falls die Anzahl von Zusammenhangskomponenten r' des neuen Graphen von r abweicht, so nennen wir e eine *Brücke* von G. In diesem Fall ist $r' = r+1$, da höchstens eine Zusammenhangskomponente in zwei partitioniert werden kann, wenn nur eine einzige Kante wegfällt.

Betrachten wir weiter einen Knoten $v \in V$. Genauso wie wir Kanten von einem Graphen entfernen können, können wir auch Knoten entfernen. Dazu konstruieren wir den Graphen $G - v := G' := (V',E')$, wobei $V' = V \setminus \{v\}$ und $E' = E \setminus \{(v_1,v_2) \in E \mid v_1 = v \vee v_2 = v\}$. Falls die Anzahl von Zusammenhangskomponenten r' des neuen Graphen von r abweicht, so nennen wir v einen *Gelenkpunkt* von G. Anders als bei Brücken, kann sich r' um mehr als 1 von r unterscheiden, da wir mehr als eine Kante vom Graphen entfernen. In unserem Falle kann ein Knoten maximal 4 Nachbarn besitzen (jeweils oben, unten, links und rechts), daher kann sich r' um maximal 3 gegenüber r erhöhen.

Wir interessieren uns im Bezug auf Gelenkpunkte weniger für den allgemeinen Fall als vielmehr einen Spezialfall, den wir *gerichteter Gelenkpunkt* nennen. Ein Gelenkpunkt heißt gerichtet, wenn die Vereinigung der Zusammenhangskomponente der horizontalen Nachbarn disjunkt zur Vereinigung der Zusammenhangskomponente der vertikalen Nachbarn ist. Formal ausgedrückt

$$\left(\bigcup\left\{Z_n \mid n \in E \cap \left\{v + \begin{pmatrix}1\\0\end{pmatrix}, v - \begin{pmatrix}1\\0\end{pmatrix}\right\}\right\}\right) \cap \left(\bigcup\left\{Z_n \mid n \in E \cap \left\{v + \begin{pmatrix}0\\1\end{pmatrix}, v - \begin{pmatrix}0\\1\end{pmatrix}\right\}\right\}\right) = \emptyset$$

Dabei steht der linke Ausdruck für die Zusammenhangskomponenten der horizontalen und der rechte für die der vertikalen Nachbarn. Wir fordern zudem, dass beide diese Mengen nicht-leer sind, um triviale Elementaroperationen ohne betroffenen Knoten zu vermeiden.

Verschiedene Interpretationen der Rotation

In der Aufgabenstellung heißt es, die Plättchen könnten „um eine Stiftachse“ gedreht werden. Dies lässt prinzipiell zwei Interpretationen zu, die im Folgenden als *Knotendrehung* und *Kantendrehung* bezeichnet werden.

Kantendrehung Sei $G = (V, E)$ Instanz. Eine erlaubte Elementaroperation transformiert den Graphen in einen neuen Graphen $G' = (V', E')$. Dies geschieht wie folgt:

1. Wähle eine Brücke $e = (u, v) \in E$, und betrachte $G - e$.
2. Da G zusammenhängend ist ($r = 1$), hat $G - e$ genau zwei Zusammenhangskomponenten, nennen wir sie $Z_1, Z_2 \subseteq V$. Die Kanten, deren Knoten in Z_i für $i \in \{1, 2\}$ liegen, bezeichnen wir als E_i.
3. Wähle eine der beiden Zusammenhangskomponenten Z_i für $i \in \{1, 2\}$. (Die andere ist demzufolge Z_{2-i}).
4. Alle Kanten von L sind entweder horizontal oder vertikal, also ist es auch $e \in E \subseteq E_L$.
5. Erweitere die (geometrische) Strecke $\overline{uv}$, wobei u und v als Punkte in der Ebene interpretiert werden, zu einer Geraden g.
6. Betrachte die Spiegelungsabbildung $s_e : \mathbb{Z}^2 \to \mathbb{Z}^2$: Rechnerisch entspricht dies im Fall einer horizontalen Kante (also $u = (a, y_0), v = (a \pm 1, y_0)$) für jeden Punkt der Abbildung $(x, y) \mapsto (x, 2y_0 - y)$; im Fall einer vertikalen Kante (also $u = (x_0, a), v = (x_0, a \pm 1)$) der Abbildung $s_e : (x, y) \mapsto (2x_0 - x, y)$. Diese Abbildung entspricht geometrisch genau der Spiegelung von Punkten an der Gerade g.
7. Betrachte $Z_i' := \{s_e(z) | z \in Z_i\}$.
8. Ist $Z_i' \cap Z_{2-i} = \emptyset$, so ist die Transformation gültig, da sich keine Knoten überlappen. In diesem Fall betrachten wir auch $E_i' := \{(s_e(a), s_e(b)) | (a, b) \in E_i\}$.
9. $V' := V_i' \cup V_{2-i}$ und $E' := E_i' \cup E_{2-i} \cup \{e\}$

Das heißt wir wählen eine Kante e, z.B. eine horizontale, sodass die links und rechts an e anhängenden Teilfiguren nur durch e zusammenhängen (Analog für eine vertikale Kante oben bzw. unten). Wir wählen einen der beiden Teile und drehen ihn entlang der Achse von e im dreidimensionalen Raum um 180°, was bedeutet, dass dieser Teil hier horizontal gespiegelt wird. Die Operation ist nur dann gültig, wenn sich im Anschluss keine Plättchen überlappen.

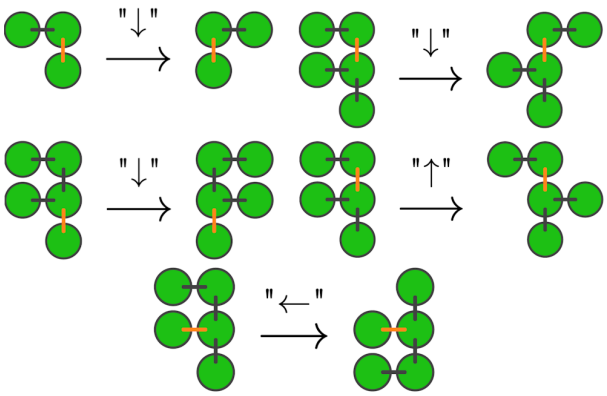


Abbildung 1.5: Beispiele für gültige Kantendrehungen. Dabei wird die Kante e stets orange hervorgehoben. Wir notieren über dem Pfeil, welche Richtung der Kante fixiert wird.

Knotendrehung Die Knotendrehung arbeitet ähnlich zur Kantendrehung, wobei allerdings hier Knoten statt Kanten ausgewählt werden. Der formale Ablauf ist dabei wie folgt, wenn $G = (V, E)$ in $G' = (V', E')$ transformiert werden soll:

1. Wähle einen gerichteten Gelenkpunkt $v \in V$, und wähle eine Art der Spiegelung (horizontal oder vertikal).
2. Es sei $\delta := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ im horizontalen Fall und $\delta := \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ im vertikalen.
3. Für die Spiegelung ist $\{(v, v-\delta), (v, v+\delta)\} \subseteq E$ erforderlich.
4. Betrachte die Spiegelungsabbildung s_e wie zuvor in Abschnitt 1.1 definiert, für $e := (v, v+\delta)$.
5. Betrachte die Zusammenhangskomponente von n in $G - (v, v+\delta) - (v, v-\delta)$.
6. Spiegele sie analog zur Kantendrehung punktweise mittels s_e, wobei alle anderen Zusammenhangskomponenten erhalten und unverändert bleiben.

Intuitiv "greifen"wir hier aus der Figur ein Plättchen (Knoten) statt einer Stiftachse (Kante), halten entweder die linke und rechte oder die obere und untere Seite fest und drehen das Angehangene dreidimensional um 180°, was wieder der Spiegelung an der diesmal aus zwei Kanten bestehenden Achse entspricht.

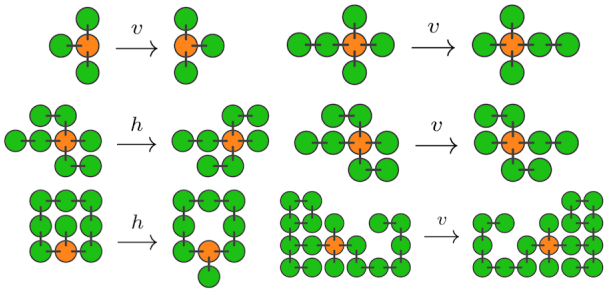


Abbildung 1.6: Beispiele für gültige Knotendrehungen. Dabei wird der Knoten n stets orange hervorgehoben. Wir notieren über dem Pfeil, welche Richtung fixiert wird: die vertikale (v) oder die horizontale (h).

Vollständige Problembeschreibung Als *Elementaroperation* beschreiben wir den Einzelschritt einer Drehung, entweder Kanten- oder Knotendrehung, wie zuvor definiert. Sei $\mathcal{G}$ die Menge aller Probleminstanzen, dann sei $(\to) : \mathcal{G} \times \mathcal{G}$ eine Relation, wobei $G_1 \to G_2 :\Leftrightarrow (G_1, G_2) \in (\to) :\leftrightarrow G_2$ geht durch Elementaroperation aus G_1 hervor. Nun sei $(\sim) : \mathcal{G} \times \mathcal{G}$ die kleinste Äquivalenzrelation mit $(\to) \subseteq (\sim)$. Wir schreiben $G_1 \sim G_2$, um auszudrücken, dass G_1 und G_2 in dieser Hinsicht äquivalent sind.

Wichtig ist an dieser Stelle, dass beide Elementaroperationen reversibel sind. Die Knotendrehung kann durch erneute Spiegelung in gleiche Richtung am selben (evtl. verschobenen) Knoten umgekehrt werden; die Kantendrehung durch entsprechende Wiederholung der Operation an der selben Kante.[4]

Das Entscheidungsproblem, ob zwei Figuren G_A, G_B ineinander überführt werden können, entspricht damit gerade $G_A \overset{?}{\sim} G_B$. Falls $G_A \sim G_B$, wird zudem nach einer Kette $G_A \to G_1 \to G_2 \to \ldots \to G_{k-1} \to G_k \to G_B$ gesucht, wobei $k \in \mathbb{N}_0$ und alle G_j für $j \in \{1, \ldots, k\}$ explizit anzugeben sind. Die Repräsentation der Graphen erfolgt im bekannten textuellen Format.

[4] eine Abbildung ihr eigenes Inverses ist, spricht man auch von einer *Involution*. Dies ist hier bei den Spiegelungsabbildungen der Fall.

1.2 Lösungsansätze

Für einen effizienteren Algorithmus ist es immer gut, unnötige Schritte zu vermeiden und ggf. sogar vorzeitig abzubrechen, daher betrachten wir zuerst einige Methoden um dies zu tun, ehe wir den eigentlichen gesuchten Pfad versuchen zu finden. In Abschnitt 1.2 wird zunächst erläutert, wie man bestimmte Eigenschaften von den Figuren im Vorhinein berechnet, um auf diese hinterher zuzugreifen. In Abschnitt 1.2 werden dann Eigenschaften genant mit denen man sofort feststellen kann, ob zwei Figuren nicht ineinander überführbar sind. In Abschnitt 1.2 wird eine weitere solche Eigenschaft aufgestellt, sowie eine Hilfestellung mit der dann in Abschnitt 1.2 schließlich der gesuchte Pfad bestimmt wird.

Nützliche Subroutinen

Nicht nur aufgrund der mehrfachen Interpretationsmöglichkeiten der Aufgabenstellung ist die Anzahl möglicher und gültiger Lösungen sehr groß. Im Folgenden präsentieren wir einige über die Implementation der Operation selbst nützliche Subroutinen, also Unterprogramme, die als Grundlage für weitere graphentheoretische Überlegungen dienen können. Eine gute Lösung muss dabei die vorgestellten Verfahren nicht oder kann sie stark modifiziert implementieren, viele Verfahren profitieren jedoch von den Algorithmen.

Brückenfindung Für die Kantendrehungsinterpretation ist das Finden von Brücken von hoher Bedeutung, da die Operation definitionsgemäß nur an Brücken ausgeführt werden kann. Eine naive quadratische Strategie unter Verwendung bereits zur Definition der Elementaroperation benötigten Methoden besteht darin, jede Kante temporär zu entfernen und die entstehenden Zusammenhangskomponenten zu zählen. Einen effizienteren Weg, dies in linearer Laufzeit zu tun, ist Tarjans Algorithmus.

Gerichtete Gelenkpunkte Wir zeigen analoge Algorithmen zum Finden gerichteter Gelenkpunkte in Graphen auf, der für die Knotendrehung von großer Bedeutung ist – jede Drehung erfolgt an einem solchen gerichteten Gelenkpunkt. Zum einen einen naiven Algorithmus, der auf der bestehenden Fehlerbehandlung der Implementierung der Elementaroperation selbst beruht. Zum anderen eine Variante des Tarjan-Algorithmus, der vertikale von horizontalen Nachbarn trennt.

Invarianten

Für einen effizienten Algorithmus ist es sinnvoll, keine unnötigen Schritte auszuführen und, wenn möglich, vorzeitig abzubrechen. Zu diesem Zweck untersuchen wir verschiedene Eigenschaften von Figuren, die dies ermöglichen.

Eine *Invariante* ist eine solche Eigenschaft, die bei Ausführen der Operation stets erhalten bleiben muss. Wir betrachten im Folgenden Eigenschaften, die sich nicht verändern können, wenn eine Elementaroperation ausgeführt wird, von einfachen zu komplexeren. Ist eine Invariante verletzt, kann somit sofort widerlegt werden, dass eine Überführung mittels Elementaroperationen möglich ist. Dass eine Invariante erfüllt ist, ist dagegen noch keine Sicherheit dafür, dass die Graphen tatsächlich äquivalent sind, wie die eingängigen Gegenbeispiele zeigen.

Kanten- und Knotenanzahl Offensichtlich bleiben die Anzahl an Kanten $|E|$ und Knoten $|V|$ in jeder Transformation gleich, da Kanten und Knoten nur verschoben, aber nie gelöscht oder hinzugefügt werden.

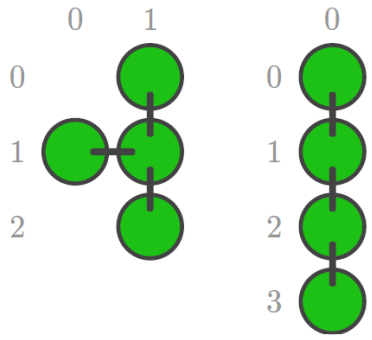


Abbildung 1.7: Beide Graphen haben die gleiche Kanten- und Knotenanzahl, sind aber nicht ineinander überführbar.

Grade Bei Betrachtung der vorherigen Abbildung fällt auf, dass im linken Graph ein Knoten mit drei anderen Knoten verbunden ist, während im rechten Graph alle Knoten höchstens mit zwei anderen Knoten verbunden sind. Die durch eine Kante mit einem Knoten v verbundenen Knoten werden dessen *Nachbarn* genannt, die Anzahl der Nachbarn eines Knoten sein *Grad*, *deg v*.

Zu Beginn der Betrachtung des Problems können wir also eine Tabelle erstellen, die die Anzahl aller Knoten eines gegebenen Grads speichert. Für Abbildung 1.7 ergeben sich die folgenden beiden Tabellen:

Grad	Zugehörige Knoten	Anzahl
1	$\{(1 \mid 0), (0 \mid 1), (1 \mid 2)\}$	3
3	$\{(1 \mid 1)\}$	1

Linker Graph aus Abbildung 1.7.

Grad	Zugehörige Knoten	Anzahl
1	$\{(0 \mid 0), (0 \mid 3)\}$	2
2	$\{(0 \mid 1), (0 \mid 2)\}$	2

Rechter Graph aus Abbildung 1.7.

Da sich der Grad eines Knotens bei Ausführung einer Elementaroperation nicht ändern kann, sind die erste und letzte Spalte dieser Tabelle invariant. Da sie hier verschieden sind, kann keine Ausführung von Elementaroperationen die Graphen ineinander überführen.

Als *Signatur* bezeichnen wir eine kompakte Notation einer bestimmten Invariante. Die Anzahl-Signatur (aus Abbildung 1.7) eines Graphen ist beispielsweise einfach [5] das Tupel $(|V|, |E|)$, während die hier eingeführte Grad-Notation, da wir in Teilgraphen des Gittergraphen arbeiten, eine Gradsignatur der Form (d_1, d_2, d_3, d_4) zulässt, wobei $d_j := |\{v \in V | deg\ v = j\}|$

Einbeziehen der Richtung Ein triviales Beispiel, bei dem die Grad-Signatur jedoch bereits nicht ausreicht, besteht aus den folgenden zwei Graphen:

[5] An dieser Stelle wird es relevant, ob die Kanten als Mengen oder Paare modelliert werden. In dieser Modellierung haben wir uns dafür entschieden, Kanten stets doppelt (Hin- und Rückkante) in der Menge E zu speichern. Zu Zwecken der intuitiven Einsichtigkeit betrachten wir demzufolge das Tupel $(|V|, \frac{|E|}{2})$.

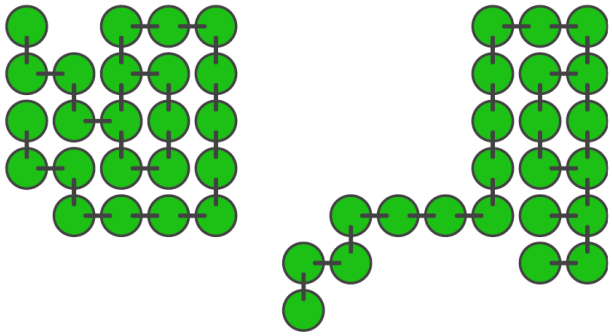
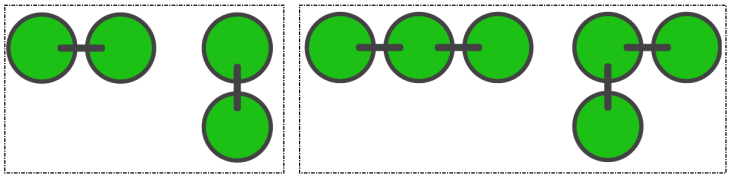


Abbildung 1.8: Es ist nicht unmittelbar ersichtlich, ob die Graphen ineinander überführbar sind. Analyse der Grade verrät jedoch, dass der linke Graph die Grad-Signatur (2, 19, 2, 0) hat, der rechte jedoch (3, 17, 3, 0). Damit können sie nicht ineinander überführt werden.



Beide Figuren können trotz gleicher Anzahl- und Gradsignatur nicht ineinander überführt werden. Die Anzahl-Signatur im linken Beispiel ist $(2,1)$, die Grad-Signatur $(1,0,0,0)$. Die Anzahl-Signatur im rechten Beispiel ist $(3,2)$, die Grad-Signatur $(2,1,0,0)$.

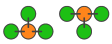
Das Problem liegt offenbar darin, dass die Gerichtetheit des Problems bisher noch unenkodiert blieb. Dies lässt sich ändern, indem wir die vertikalen und horizontalen Kanten separat betrachten. Ein Knoten kann dabei je bis zu zwei horizontale und vertikale Nachbarn haben (also gibt es je Richtung 3 Möglichkeiten). Da wir den Fall des trivialen Graphs vernachlässigen, verbleiben $3 \cdot 3 - 1 = 8$ Optionen, welche die nachfolgende Tabelle illustriert. Die *Freiheitsgrade* beschreiben dabei, wie viele unabhängige Operationen an einem Knoten dieses Typs in der Knotendrehung ausgeführt werden können.

Wir weisen also jedem Knoten einen solchen Typen zu und zählen die Anzahl. Wir erhalten als Richtungs-Signatur diesmal ein 8-Tupel $(r_1, ..., r_8)$, wobei $r_j := |\{v \in V | \text{Typ}(v) = j\}|$, wobei

$$\text{Typ}(v) := \left| \left\{ \left(v, v + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right), \left(v - \begin{pmatrix} 1 \\ 0 \end{pmatrix}, v \right) \right\} \cap E \right| + 3 \left| \left\{ \left(v, v + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right), \left(v - \begin{pmatrix} 0 \\ 1 \end{pmatrix}, v \right) \right\} \cap E \right|$$

Die Grad-Signatur wird von der Richtungs-Signatur subsumiert, da jeder Knotentyp der Richtungs-Signatur einen eindeutigen Grad hat. Die Abbildung $U : \mathbb{Z}^8 \to \mathbb{Z}^4, U : (r_1, ..., r_8) \mapsto (r_1 + r_3, r_2 + r_4 + r_6, r_5 + r_7, r_8)$ rekonstruiert die Grad-Signatur aus der Richtungs-Signatur durch „Vergessen“ der Richtung.

Wenn man diese Signatur für die Graphen in Abbildung 1.8 anwendet, erhält man $(0,3,2,12,0,4,2,0)$ bzw. $(2,3,1,9,0,5,3,0)$. Im Folgenden werden weitere Figuren zusammen mit ihren Signaturen präsentiert.

Beispiele (relevante Knoten orange)	Grad	Freiheitsgrade	Anzahl der h. Nachbarn	Anzahl der v. Nachbarn
	1	1	1	0
	2	0	2	0
	1	1	0	1
	2	2	1	1
	3	1	2	1
	2	0	0	2
	3	1	1	2
	4	0	2	2

Figuren	Anzahl-Signatur	Grad-Signatur	Richtungs-Signatur
	(4,3)	(2,2,0,0)	(2,0,0,2,0,0,0,0)
	(4,3)	(2,2,0,0)	(1,0,1,1,0,1,0,0)
	(6,5)	(2,4,0,0)	(1,1,1,3,0,0,0,0)
	(8,8)	(0,8,0,0)	(0,2,0,4,0,2,0,0)
	(9,12)	(0,4,4,1)	(0,0,0,4,2,0,2,1)

Isomorphie

Naive Isomorphie Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ zwei beliebige Graphen. G_1 heißt *isomorph* zu G_2, wenn eine bijektive (eindeutig umkehrbare) Funktion $\varphi : V_1 \to V_2$ existiert, sodass $V_2 = \varphi(V_1)$ und $E_2 = \{(\varphi(u), \varphi(v)) | (u,v) \in E_1\}$. In diesem Fall heißt die Abbildung *phi Isomorphismus* zwischen G_1 und G_2.

Wir haben bereits in Abschnitt 1.1 gezeigt, dass (jedenfalls naive) Isomorphie nicht hinreichend für Lösbarkeit ist. Allerdings ist sie notwendig. Das zeigen wir analog zum Vorgehen bei Invarianten: jede Elementaroperation kann einen Graphen zur zu einem zu diesem isomorphen transformieren. φ haben wir dabei praktisch bereits in Abschnitt 1.1 explizit konstruiert: die Abbildung schickt alle nicht gespiegelten Knoten auf sich selbst, und alle zu spiegelnden gerade auf ihr Spiegelbild. Da die Anzahl an Knoten und Kanten unverändert bleibt (Überlappung ist unzulässig), ist φ umkehrbar, also bijektiv.

Gerichtete Isomorphie Wie schon in Abschnitt 1.2 ist ein Grundproblem der Isomorphie, dass sie Richtung nicht berücksichtigt. So ist für alle Probleminstanzen $G = (V, E)$ die Bijektion $\varphi_{flip} : \mathbb{Z}^2 \to \mathbb{Z}^2, \varphi_{flip} : (x|y) \to (y|x)$, welche die x- und y-Komponente aller Punkte tauscht, ein Isomorphismus zwischen einem Graphen und seiner Spiegelung entlang der Hauptdiagonale.

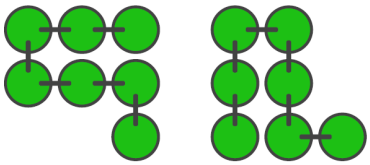


Abbildung 1.9: Der rechte Graph geht aus dem linken durch Spiegelung an der Hauptdiagonalen hervor, aber sie sind nicht überführbar.

Wir betrachten erneut die Definition der Isomorphie durch bijektive Abbildungen. Zwei Graphen G_1, G_2 nennen wir dann *gerichtet isomorph*, wenn ein Isomorphismus $\varphi : V_1 \to V_2$ existiert, der zudem jeden Knoten eines Typs nur auf einen Knoten desselben Typs abbildet, d.h. für alle Typen $j \in 1, ..., 8$ gilt $\varphi(\{v \in V_1 | \text{Typ}(v) = j\}) = \{v \in V_2 | \text{Typ}(v) = j\}$.

Diese neue Form der Isomorphie verfeinert sowohl die naive Isomorphie (sind zwei Graphen nicht isomorph, so können sie auch nicht gerichtet isomorph sein) als auch die Richtungs-Signatur (haben zwei Graphen nicht dieselbe Richtungs-Signatur, so kann es keinen gerichteten Isomorphismus zwischen ihnen geben, da schon die Mengen unterschiedlich groß sind). Das folgende Beispiel zeigt, dass zwei Graphen, die isomorph sind und gleiche Richtungs-Signaturen aufweisen, dennoch nicht notwendigerweise gerichtet isomorph sind:

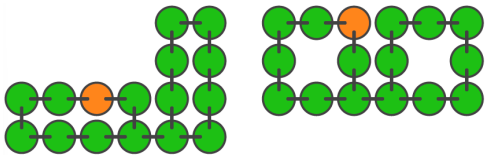


Abbildung 1.10: Die Graphen links und rechts haben beide die Richtungs-Signatur (0,4,0,6,2,4,0,0) und sind isomorph. Das orange eingefärbte Plättchen illustriert, dass der Isomorphismus Knoten auf Knoten anderer Typen abbilden muss. Die Graphen sind jedoch nicht Richtungs-isomorph, wie sich mit etwas mehr Theorie im kommenden Abschnitt zeigen lässt.

Isomorphie-Problem Das Entscheidungsproblem, zu zwei gegebenen beliebigen Graphen ihre Isomorphie oder Nicht-Isomorphie festzustellen, wurde in der Literatur bereits extensiv thematisiert. Nach dem aktuellen Forschungsstand gibt es keinen Algorithmus, der in Polynomialzeit das Isomorphie-Entscheidungsproblem löst. Das Problem liegt offenbar in *NP*, da der Isomorphismus selbst (bzw. eine explizite Darstellung als Lookup-Tabelle) als Zertifikat dienen kann. Es wird vermutet, dass das Problem nicht *NP*-schwer ist.

Ein naiver Ansatz für das allgemeine Isomorphieproblem liegt in dem Ausprobieren aller möglichen Permutationen. Für einen Graph mit $n := |V|$ Knoten skaliert diese Zahl mit der Fakultät von n, geschrieben $n!$. Da diese Funktion für hinreichend große n schneller wächst als jede Exponentialfunktion, ist dieser Ansatz nicht handhabbar.

Glücklicherweise arbeiten wir mit einer sehr speziellen Klasse von Graphen, nämlich Teilgraphen des Gittergraphen. Diese sind wie bereits beschrieben *planar*, können also in der Ebene eingebettet werden. Für diese Klasse von Graphen existieren effiziente Algorithmen, die das Isomorphieproblem in polynomieller Zeit lösen können.

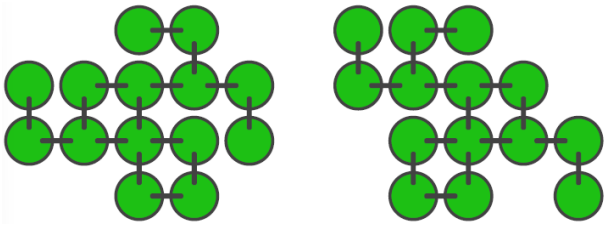
Color Refinement Der Color Refinement-Algorithmus (auch: 1-dimensionaler Weisfeiler-Leman-Algorithmus) arbeitet als starke Erweiterung der zuvor definierten Typen. Die Knoten werden dabei eingefärbt, wobei der Farbbegriff als abstrakte Zuweisung von Identifikatoren zu verstehen ist, die strukturelle Eigenschaften repräsentieren.

Anfänglich wird jedem Knoten als Farbe ein beliebiger trivialer Wert zugewiesen. Für jeden Knoten betrachten wir nun dessen Multimenge der Nachbarn für jede der beiden Richtungen. Wir erstellen in der ersten Iteration einen Hash-Wert, zusammengesetzt aus vorheriger Knotenfarbe, der Multimenge horizontalen Nachbarknotenfarben und der Multimenge vertikalen Knotenfarben.

Dieser Iterationsschritt wird wiederholt, bis Konvergenz eintritt, sich die Verteilungen der Knotenfarben also nicht mehr verändern. Auf diese Art und Weise entstehen sehr aussagekräftige Signaturen, die genau wie die vorherigen als schnelles Ausschlusskriterium genutzt werden können.

Für hochgradig symmetrische Graphen (insb. sogenannte *reguläre* Graphen) kann der Algorithmus fälschlicherweise die Nicht-Richtungsisomorphie nicht erkennen. Da der Rastergraph die mögliche Symmetrie jedoch bereits stark einschränkt, und alle interessanten Problemeingaben hinreichend asymmetrisch sind, können wir davon ausgehen, dass dieser Fall nicht auftritt.

Wir analysieren am folgenden Beispiel die Ausgabe des Algorithmus für dieses Beispiel:



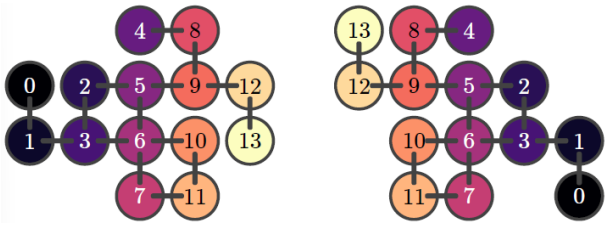
Die Graphen sind gerichtet isomorph, und in diesem Fall auch ineinander überführbar. Wendet man den Algorithmus an, so erhält man bei geeigneter Färbung die folgende Ausgabe (die Signatur-IDs/Knotentypen beginnen hier bei 0):

Sig.-ID	Knoten links	Knoten rechts	Farbe
0	{(0,1)}	{(4,3)}	
1	{(0,2)}	{(4,2)}	
2	{(1,1)}	{(3,1)}	
3	{(1,2)}	{(3,2)}	
4	{(2,0)}	{(2,0)}	
5	{(2,1)}	{(2,1)}	
6	{(2,2)}	{(2,2)}	

Sig.-ID	Knoten links	Knoten rechts	Farbe
7	{(2,3)}	{(2,3)}	
8	{(3,0)}	{(1,0)}	
9	{(3,1)}	{(1,1)}	
10	{(3,2)}	{(1,2)}	
11	{(3,3)}	{(1,3)}	
12	{(4,1)}	{(0,1)}	
13	{(4,2)}	{(0,0)}	

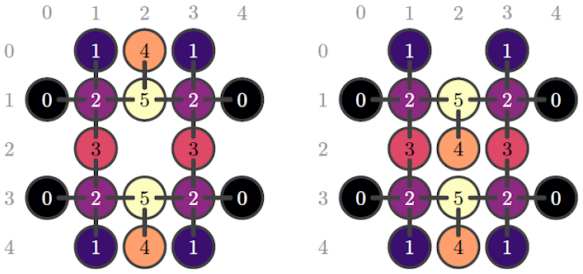
Die Tabelle zeigt, dass es hier 14 vorhandene Signatur-IDs gibt, die die bisherigen Knotensignaturen verallgemeinern. Ebenfalls ist zu erkennen, dass jede Signatur-ID in beiden Fällen genau einem Knoten zugewiesen ist. In diesem Fall ist die darauffolgende Konstruktion des Isomorphismus trivial.

Projizieren wir die Farben auf die Graphen, so ist die folgende anschauliche Darstellung der Typen zu erkennen:



Extraktion des Isomorphismus Im vorherigen Fall hatte jede Knoten-Signatur genau einen Vertreter. Aufgrund der inkrementellen Natur des Algorithmus ist dieser Fall häufig und eine simple Iteration über alle Typen genügt.

Sig.-ID	Knoten links	Knoten rechts	Farbe
0	{(0,1), (0,3), (4,1), (4,3)}	{(0,1), (0,3), (4,1), (4,3)}	
1	{(1,0), (1,4), (3,0), (3,4)}	{(1,0), (1,4), (3,0), (3,4)}	
2	{(1,1), (1,3), (3,1), (3,3)}	{(1,1), (1,3), (3,1), (3,3)}	
3	{(1,2), (3,2)}	{(1,2), (3,2)}	
4	{(2,0), (2,4)}	{(2,2), (2,4)}	
5	{(2,1), (2,3)}	{(2,1), (2,3)}	



In diesem Fall ist der Graph symmetrisch genug, um nach vollständigem Color Refinement mehrere Knoten gleichen Typs zuzulassen. Es sind algorithmisch aufwändigere Techniken notwendig, um eine Isomorphie zu konstruieren.

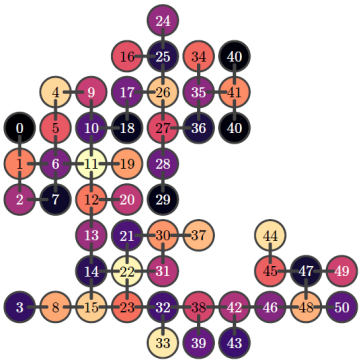
Eine naheliegende Strategie liegt hier in einem Backtracking-Ansatz. Wir konstruieren die Isomorphie schrittweise Knoten für Knoten und treffen in jedem Schritt die notwendigen Konsequenzen: Sobald ein Typ nur noch einen Vertreter besitzt, wird er folgerichtig diesem zugewiesen. Sobald alle eindeutigen Zuweisungen erfolgt sind, wird eine neue Zuweisung geraten und hinzugefügt. Nach jedem Schritt wird überprüft, ob die bisherige Abbildung bezüglich der Kanten der Graphen widersprüchlich ist: beispielsweise dürfte ein Isomorphismus $(0 \mid 1)$ auf $(4 \mid 1)$ schicken und ein anderer $(1 \mid 1)$ auf $(1 \mid 3)$. Die folgende Kombination ist dagegen nicht möglich, da zwischen den Argumenten eine Kante besteht, zwischen den Bildern dagegen nicht: $((0 \mid 1), (1 \mid 1)) \in E$, aber $((4 \mid 1), (1 \mid 3)) \notin E$.

In der praktischen Implementierung ist relevant, in welcher Reihenfolge die Knoten der Abbildung zugewiesen werden. Um den Suchraum lokal möglichst klein zu halten, bietet es sich an, Nachbarn bereits hinzugefügter Knoten eher zu betrachten als solche in fremden Nachbarschaften. Da jeder Knoten höchstens vom Grad 4 ist, verbleiben höchstens so viele Optionen für Zuweisung des Nachbarn, was eine effektive Allokation erlaubt.

Anzahl der Automorphismen Ein *Automorphismus* ist ein Isomorphismus, wobei wir nur einen einzigen Graphen $G = (V, E)$ betrachten und diesen sowohl als G_1 als auch G_2 nach ursprünglicher Definition auffassen. Wir definieren einen *gerichteten Automorphismus* analog zum *gerichteten Isomorphismus*. Jeder Graph lässt den trivialen Automorphismus $\alpha = id_V$ zu; jeder Graph mit hinreichend symmetrischen Substrukturen besitzt auch nicht-triviale Automorphismen. Da Automorphismen zudem unter Verknüpfung abgeschlossen und als Isomorphismen invertierbar sind, bilden sie unter Verknüpfung eine Gruppe.

Wendet man die Problemstellung des Wettbewerbs auf zwei identische Figuren an, so ist die Lösung trivial durch Nichtdurchführen von Elementaroperationen gegeben und damit uninteressant. Wir betrachten Automorphismen daher als Spezialfall von Isomorphismen hier, um den Aspekt der Nichteindeutigkeit und die Anzahl möglicher Abbildungen zu verdeutlichen.[6] Ferner kann jeder nicht-triviale Automorphismus mit einem Isomorphismus verknüpft werden, um einen neuen Isomorphismus zu erhalten [7].

(a) Ein auf mehreren Ebenen selbstsymmetrischer Graph mit genau 32 gerichteten Automorphismen. Jede Farbe hat mindestens zwei Vertreter.



(b) Zufällige Graphen wie dieser sind asymptotisch zwar exponentiell automorph, in der Praxis ist die Basis jedoch klein genug, sodass andere Überlegungen überwiegen. Im konkreten Fall liegen zwei Automorphismen vor: *id* und der Tausch der beiden Knoten mit Typ 40.

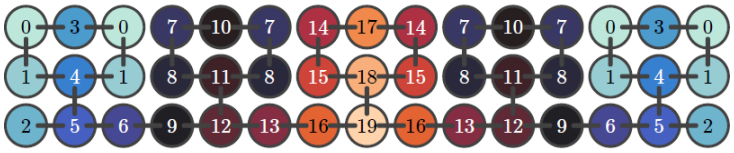


Abbildung 1.12: Erweitert man diesen Graphen horizontal, so zeigt sich, dass die Anzahl der Automorphismen mit der Anzahl an Wiederholungen exponentiell wächst. Genauer: ist $n \in \mathbb{N}$ die Anzahl an horizontal verketteten „Atomen“ (ein Atom hat die Größe 3×3, hier ist $n = 5$), so gibt es genau $2n + 1$ gerichtete Automorphismen.

Da eine Probleminstanz exponentiell viele Automorphismen besitzen kann, ist die Höchstanzahl an Isomorphismen mindestens genauso hoch. Es ist jedoch möglich, die polynomiell große Menge an *Generatoren* mit einem Verfahren wie dem von Schreier-Sims zu bestimmen, sodass jeder Automorphismus als polynomiell große Verknüpfung der Generatoren ausgedrückt werden kann.

Im Folgenden verlassen wir die Betrachtung hochgradig symmetrischer Graphen und nehmen an, dass der Graph asymmetrisch genug ist, um die Automorphismengruppe polynomiell klein relativ zur Eingabegröße zu halten, oder dass nur polynomiell viele Isomorphismen für die

[6] Eine andere interessante Herangehensweise liegt hier in der Dekomposition eines Isomorphismus φ in eine Form $\alpha \circ g \circ \beta$, wobei α, β Automorphismen sind und g ein „kanonischer“ Isomorphismus. Wir befassen uns nicht näher damit.

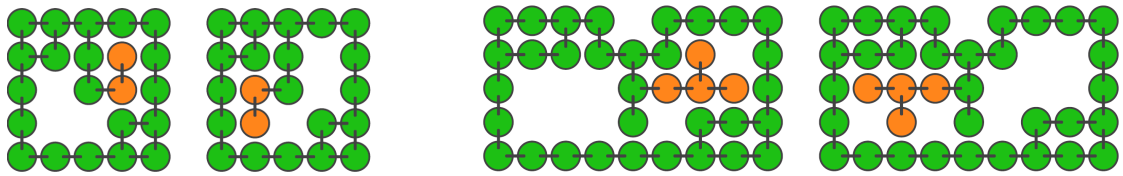
[7] Der triviale Automorphismus kann natürlich auch verknüpft werden, in diesem Fall bleibt der Isomorphismus unverändert.

Korrektheit betrachtet werden müssen. Um den Fokus auf den Kern der Problematik zu legen, klammern wir hochsymmetrische Spezialfälle an dieser Stelle bewusst aus.[8]

Pfadsuche

Auffassen als Suchproblem Wir haben es nun (vermeintlich) geschafft, hinreichend schnell einen Isomorphismus φ zwischen unseren Graphen zu finden (oder zu beweisen, dass es keinen solchen gibt). Insbesondere wissen wir (im typischen asymmetrischen Fall) für jeden Knoten exakt, auf welchen er abgebildet werden muss. Wie schwer kann es noch sein, den tatsächlichen Pfad zu finden?

Mit der komplexitätstheoretischen Frage beschäftigen wir uns im Extraabschnitt 1.4. Zunächst einmal ist es wichtig, zu erwähnen, dass selbst gerichtete Isomorphie, so aussagekräftig sie strukturell bereits ist, noch nicht hinreichend für Lösbarkeit ist. Diese Tatsache sollte intuitiv schon dadurch einleuchten, dass keine der bisherigen Verfahren die Knoten- und Kantendrehungsinterpretationen unterscheiden konnte.



(a) Kantendrehung (b) Knotendrehung

Abbildung 1.13: Zwei gerichtete isomorphe Graphen, die mit der jeweiligen Drehung nicht ineinander überführbar sind. Aufgrund der äußeren Struktur können die orange eingefärbten Knoten nicht durch Bijektion überführt werden.

Beide Beispiele haben gemein, dass zwei simultane Schritte nötig wären, um die Transformation durchzuführen. Isomorphismen stellen stets eine globale Sicht auf das graphentheoretische Problem dar und erlauben hier nicht, die lokalen Nebenbedingungen der Unterteilbarkeit in Elementaroperationen zu respektieren.

In diesem Fall kann das Problem als Suchproblem in einem weiteren Graphen aufgefasst werden. An Stellen wie diesen zeigt sich wieder einmal die faszinierende Allgemeinheit abstrakter Graphentheorie... Der Graph sei $G_S = (V_S, E_S)$, wobei V_S die Menge aller durch Graphen repräsentierten Figuren ist[9] und $(G_1, G_2) \in E_S$ genau dann, wenn G_2 durch Elementaroperation aus G_1 hervorgeht.

Größe des Suchraums Um die Laufzeit des Suchverfahrens abschätzen zu können, ist es notwendig, die Größe des Suchraums zu ermitteln. Die Mehrheit der bisherigen Beispiele wurde bewusst einfach konzipiert, mit möglichst wenig Freiheitsgraden und allgemeinem Spielraum, um die Konzepte möglichst verständlich zu präsentieren.

[8] Tiefere komplexitätstheoretische Argumente können zeigen, dass das Bestimmen von Generatoren dieser Klasse von Graphen P-reduzierbar für das entsprechende Graphisomorphieproblem ist. Mit Planarität folgt ein effizienter Algorithmus für beide Probleme. Wir vermuten, dass mithilfe der Generatoren auch die Untergruppe der überführbaren Automorphismen ermittelt werden kann.

[9] Diese kann als entweder als unendliche Menge aufgefasst werden, oder man beschränkt sich auf die endliche Menge aller Graphen einer bestimmten Signatur. In der Praxis macht diese Unterscheidung keinen großen Unterschied, wenn man auf den Versuch, die Elemente zu enumerieren, verzichtet.

Zunächst ist festzuhalten, dass der Suchraum stets endlich ist und durch einen Ausdruck der Form $2^{p(n)}$ für ein Polynom p nach oben beschränkt werden kann. Um dies zu erkennen, stellen wir fest, dass die Breite und Höhe im Koordinatensystem der durch Elementartransformationen veränderten Graphen die Anzahl der Kanten nie übersteigen kann, da der Graph zusammenhängend bleibt. Es gibt im $n \times n$-Quadrat allerdings (asymptotisch) nur $2n^2$ potenzielle Kanten. Da sich ein zusammenhängender Graph auch vollständig über seine Kanten beschreiben lässt, ist eine krude Schranke für die obere Anzahl von Figuren im Suchraum durch 2^{4n^2} gegeben.

Bereits sehr simple Instanzen führen dabei zu exponentiell großen Suchräumen. Man betrachte für $m \in \mathbb{N}$ die Instanz $G_m := (V_m, E_m)$ mit

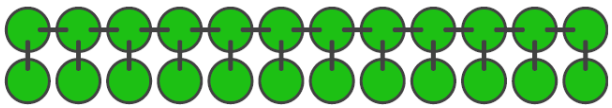
$$V_m := \{(x,y) \mid x \in \{0,\ldots,m-1\}, y \in \{0,1\}\}$$

und

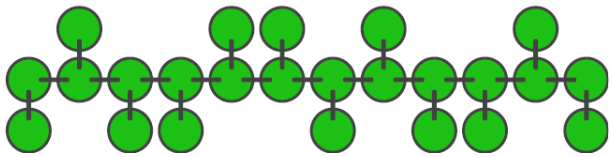
$$E_m := \{(v_1,v_2) \mid (v_1,v_2) \in V^2,\ v_1 = (x_1 \mid y_1),\ v_2 = (x_2 \mid y_2),\ y_1 = y_2 = 0 \vee y_1 \neq y_2\} \cap E_L,$$

wobei E_L die Kanten des Gittergraphen L bezeichnet.

Betrachten wir exemplarisch G_{12}:



Es ist zu erkennen, dass jedes einzelne Blatt (Knoten mit einer ausgehenden Kante) durch Elementaroperationen an $y = 0$ einzeln gespiegelt werden kann. Interpretiert man die Position als je ein Bit, so kann jede Binärzahl zwischen 0 und $4095 = 2^{12} - 1$ dargestellt werden, als illustratives Beispiel hier einmal die $1234 = 010011010010_2$:



Offenbar hat G_m bei exakt $2m$ Knoten einen Zustandsraum der Größe 2^m, wenn alle Verschiebungen möglich sind (Um ein äquivalentes Beispiel für die Knotendrehung zu konstruieren, füge man am linken und rechten Ende je noch ein weiteres Plättchen an.).

Implizite Graphenrepräsentation Während mathematische Modellierungen von Graphen ohne größere Probleme auch Graphen mit unendlichen Kanten- und Knotengraphen darstellen können – man denke zurück an den Gittergraphen L – ist eine explizite Repräsentation wie mithilfe der ‘Graph‘-Klasse in realen Rechnern nicht möglich, da der Speicherplatz solcher Geräte bedauerlicherweise endlich ist.

Dennoch ist es möglich und sinnvoll, unendliche Graphen auf Computern zu speichern, indem implizite Darstellungsmethoden gewählt werden, die bestimmte typische Operationen anbieten, ohne den Speicherplatz mit einer naiven Repräsentation zu füllen. Eine Darstellungsweise eines Graphen $G = (V,E)$, wobei $V \subseteq U$ mit einer bekannten repräsentierbaren Menge U (z.B.

$\{0,1\}^*$, die Menge aller endlichen Bitstrings [10]) umfasst zwei Funktionen $\mathbf{1}_V : U \to \{0,1\}$ und $\mathbf{1}_E U^2 \to \{0,1\}$, die *Indikatorfunktionen* von V und E. Dies entspricht einer mathematischen Konzeption entscheidbarer Mengen.

In der Praxis ist es häufig sinnvoller, für zusammenhängende Graphen $G = (V,E)$ eine Darstellung (v_0, Γ) zu verwenden, wobei $v_0 \in V$ als *Startknoten* und $\Gamma : V \to V^*$ als *Nachbarfunktion* bezeichnet wird, die jedem Knoten ihre Nachbarn zuweisen. Dies ist insbesondere dann attraktiv, wenn Inklusion in V komplexer zu definieren ist als die Nachbarschaftsbeziehungen im Graphen untereinander.

Dies ist hier zweifelsfrei der Fall, wir betrachten als Startknoten einen der beiden als Eingaben erhaltenen Graphen und als Nachbarfunktion alle durch eine Elementaroperation aus diesem hervorgehenden weiteren Graphen. Auf diese Art und Weise können Suchalgorithmen, die nur von diesen Funktionen Gebrauch machen, einfach für potenziell unendliche oder wie hier immerhin exponentiell große Graphen verwendet werden.

Breiten- und Tiefensuche Die klassische Herangehensweise für ein derartiges Suchproblem in einem ungewichteten Graphen ist die *Breitensuche*. Hierbei wird als Datenstruktur eine Schlange verwendet, die Paare (Knoten, Pfad) speichern. Sie wird einelementig mit dem Paar (v_0, ε) initialisiert, wobei ε den leeren Pfad repräsentiert. Zugleich verwalten wir eine Menge besuchter Knoten; aufgrund des großen Speicheraufwands, die vollständigen Graphen zu speichern, ist es sinnvoll, hier nur Hashes (z.B. 128 Bit) zu speichern.[11]

Während die Schlange nicht leer ist, entfernen wir das zuerst eingefügte Element (v_c, p) (FIFO-Prinzip). Ist es der Zielknoten, terminieren wir und geben den Pfad p zurück. Andernfalls markieren wir seinen Hash als besucht und fügen die unbesuchten Nachbarn samt um v_c konkatenierten Pfad ans Ende der Schlange ein.

Vorteile der Breitensuche liegen darin, dass gewährleistet werden kann, dass stets der kürzeste Pfad gefunden wird, da alle Knoten nach Pfadlänge aufsteigend geordnet besucht werden. Da die Aufgabe nicht fordert, einen möglichst kurzen Pfad zu finden, ist dieser Vorteil vernachlässigbar. Problematisch ist der hohe Speicheraufwand der schnell wachsenden Schlange und die allgemein langsame Exploration des Suchraums.

In der *Tiefensuche* wird statt der Schlange ein Stapel (Kellerspeicher) verwendet, der dem LIFO-Prinzip folgt, ansonsten ist das Verfahren analog. Für viele Eingaben ist dieser Algorithmus sinnvoller, da mehr spekulative Änderungen durchgeführt werden, ehe ein (zur Isomorphiesuche analoges) Backtracking stattfindet. Negativ ist, dass die Pfade sehr viel länger werden, was auch Einfluss auf den Speicherverbrauch der involvierten Datenstrukturen nimmt. Auch eine heuristische Kombination der beiden Verfahren ist denkbar.

Heuristik: Greedy Matching Wir verwenden den Isomorphismus als Grundlage für eine einfache Heuristik: wir zählen die Anzahl an Knoten, die sich gemäß der Isomorphie bereits an der richtigen Position befinden, und führen gierig den Zug aus, der die Anzahl maximiert. Weiterhin speichern wir bereits besuchte Positionen, und backtracken im Falle einer Sackgasse. In der Praxis ermöglicht die Heuristik das schnellere Lösen vieler kleinerer Beispiele.

[10] Für eine Menge Σ bezeichnet $\Sigma^* := \bigcup_{n\in\mathbb{N}0} \Sigma^n = \cup\, \Sigma^0 = \{() = \varepsilon, Sigma^1 = \Sigma, \Sigma^2, \Sigma^3, \cdots\}$ die Kleenesche Hülle von Σ, also die Menge aller endlich langen Tupel bzw. Strings mit Elementen aus Σ. Häufig wird Σ als *Alphabet* und ein Element $w \in \Sigma^*$ als *Wort* bezeichnet.

[11] Auch der Einsatz probabilistischer Datenstrukturen wie z.B. von Bloom-Filtern kann an dieser Stelle angebracht sein, wenn die Abwägungen zwischen Effizienz und Korrektheit sinnvoll getroffen werden.

Diese simple Abgleich-Heuristik kann selbst ohne einen konkreten Isomorphismus nur auf Basis einer Typ-Logik funktionieren, indem die Anzahl der dem korrekten Typ zugewiesenen Knoten optimiert wird. Ist ein Isomorphismus gegeben, so sind auch weitere Kriterien, wie die Minimierung der Distanz zwischen einer Knoten-Startposition und seiner Zielposition, denkbar. In diesem Fall muss jedoch abgewogen werden, ob dies problembezogen formuliert werden kann, da jede Drehung potenziell Knoten beliebig weit innerhalb des Graphen verschieben kann.

Für einen Einsatz in Algorithmen wie A^* ist die Heuristik dagegen nur eingeschränkt einsetzbar, da sie die benötigten Eigenschaften der Zulässigkeit und Konsistenz verletzt, wie Beispiele zeigen.

Heuristik: Kanonische Form / „Aufklappen“ Einige Beispieleingaben sind bewusst so konzipiert, dass eine Herangehensweise mit dem vorherigen Verfahren nicht genügt. In diesem Fall kann es sinnvoll sein, den Graphen „aufzuklappen“, indem die Höhe und Breite einer ausgewählten lokalen Struktur zunächst gierig maximiert wird, um anschließend eine Änderung vorzunehmen, die die vorhergehenden Schritte voraussetzte. Es gibt verschiedene Varianten einer Implementierung dieser Heuristik im Code, die unterschiedlich gut mit diversen Graphen umgehen.

Heuristik: Segmentierung In vielen vorkommenden Graphen sind Substrukturen zu erkennen, die sich gegenseitig nicht beeinflussen können und daher separat betrachtet werden können. Im Idealfall wird so der multiplikative Zustandsraum auf einen additiven reduziert. Auch hier verzichten wir auf eine detailliertere mathematische Beschreibung oder Implementierungsnotizen zugunsten eines Überblicks über die den Autor:innen einsichtigen Heuristiken.

Heuristik: Gauss Jordan Elimination Nachdem ein gerichteter Isomorphismus gefunden wurde ist es möglich daraus eine „naive“ Folge von Elementaroperationen zu extrahieren, die die eine Figur in die andere überführt. Die Folge ist „naiv“ in dem Sinne, dass nicht berücksichtigt wird, ob zwei Knoten möglicherweise nach einer Operation überlappen und damit ist es oft keine gültige Folge. Dennoch gibt dieses Verfahren eine grobe Richtlinie wie die Knoten bzw. Kanten gedreht werden müssen.

Das Verfahren besteht darin, das Problem als lineares Gleichungssystem darzustellen. Der gerichtete Isomorphismus gibt vor, welche Orientierung die Knoten annehmen müssen, daher ist es möglich für jeden Knoten einzutragen, in wie fern sich die Orientierungen der anderen Knoten ändern, falls eine horizontale Drehung durchgeführt wird. Falls die Orientierung gedreht wird, so wird dies als 1 kodiert, falls nicht, so wird dies als 0 kodiert. Dadurch entsteht eine $n \times n$-Matrix über dem Körper $\mathbb{F}_2$ [12], da nur 0 und 1 verwendet werden, wir nennen diese Matrix A. Zusätzlich wird ein Vektor $\vec{b} \in \mathbb{F}_2^n$ erzeugt, wessen i-ter Eintrag angibt, ob der Knoten i horizontal gedreht werden werden muss. Indem man nun das Gleichungssystem $A\vec{x} = \vec{b}$ löst mit $\vec{x} \in \mathbb{F}_2^n$ erhält man einen Vektor, welcher beschreibt, welche Knoten gedreht werden müssen. Ist der i-te Eintrag von $\vec{x}$ gleich 1, so muss der i-te Knoten gedreht werden, andernfalls nicht. Da das doppelte drehen äquivalent dazu ist, dass der Knoten nicht gedreht wird, ist es korrekt, dass bei der Lösung des Gleichungssystems $1 + 1 = 0$ ergibt.

[12] Hiermit ist der eindeutige Körper, welcher ausschließlich aus dem neutralen Element der Addition und dem neutralen Element der Multiplikation besteht gemeint, sprich es existieren nur 0 und 1. Eine andere Schreibweise für den gemeinten Körper ist oft $GF(2)$

Indem man dieses Verfahren einmal für die horizontale Richtung und einmal für die vertikale Richtung durchführt, erhält man die Menge der zu drehenden Knoten und die subsequente Ausführung der Elementaroperationen überführt die eine Figur in die andere. Dabei ist jedoch zu beachten, dass die Lösung des Gleichungssystems nicht beachtet, ob sich Knoten bei der Ausführung der Elementaroperationen überlappen. Somit ist dies nur eine Heuristik.

Das Gleichungssystem selbst wird mit der Gauss Jordan Elimination, auch bekannt als „ Gauss Verfahren“, gelöst. Die Besonderheit hierbei liegt jedoch darin, dass das Skalieren von Zeilen vollständig entfällt (da man nur mit 1 skalieren könnte) und das Addieren von Zeilen dem bitweise XOR der beiden Zeilen entspräche.

1.3 Beispiele

Eingabe	Lösbar mit		Invariante mit der abgebrochen wird
	Knoten-drehung	Kanten-drehung	
`transform0y.txt`	ja	ja	-
`transform0n.txt`	nein	nein	-
`transform01.txt`	nein	nein	Grad-Signatur
`transform02.txt`	ja	ja	-
`transform03.txt`	nein	nein	keine
`transform04.txt`	nein	nein	naive Isomorphie
`transform05.txt`	nein	nein	keine
`transform06.txt`	ja	ja	-
`transform07.txt`	ja	ja	-
`transform08.txt`	ja	ja	-
`transform09.txt`	ja	ja	-
`transform10.txt`	ja	ja	-
`transform11.txt`	ja	ja	-
`transform12.txt`	ja	ja	-
`transform13.txt`	ja	nein	-

1.4 Exkurs: NP-Schwere

An dieser Stelle präsentieren wir einen Beweis der *NP*-Schwere des Entscheidungsproblems, also der Fragestellung, ob für zwei Figuren G_1, G_2 die Aussage $G_1 \sim G_2$ gilt. Unter der in der Komplexitätstheorie üblichen Annahme, dass $Pc \neq NP$, kann es also keinen effizienten (Polynomialzeit-) Algorithmus geben, welcher die Fragestellung allgemein beantworten kann[13]. Es sei explizit gesagt, dass eine solche (so ausführliche) Betrachtung nicht erwartet wird, aber durchaus mit Pluspunkten belohnt wird.

An dieser Stelle beweisen wir NP-Schwere für die Interpretation der Knotendrehung. Für die Kantendrehung kann der Beweis mit leichten Änderungen der Struktur (vgl. die entsprechenden Beispieldateien) komplett analog geführt werden.

[13] Es ist dabei *Pc* die Klasse der in Polynomialzeit lösbaren Entscheidungsprobleme; *NP* ist charakterisiert als die Klasse der in Polynomialzeit verifizierbaren Entscheidungsprobleme. Näher wollen wir auf diese Begriffe hier nicht eingehen und verweisen auf die entsprechende Literatur.

Der Beweis basiert auf einer Konstruktion, die im Paper „Classic Nintendo Games are (Computationally) Hard“ ausführlich vorgestellt wird. Weitere Informationen zum Verfahren sind in den Kapiteln 2 und 3 des Papers zu finden. Auch auf YouTube finden sich Videos, die die Struktur des Beweis anhand vom Beispiel „Super Mario Bros“ darstellen[14].

Anmerkung zur den Grafiken in diesem Kapitel: Zur besseren Übersichtlichkeit haben wir anstelle der knotenzentrierten Darstellung eine kantenzentrierte gewählt. Wir heben in Orange weiterhin die Kante hervor, an der die Spiegelung durchgeführt wird. Mehrschrittige Übergänge sind von links nach rechts, dann von oben nach unten in Pfeilrichtung zu lesen.

Grundlagen Wir stellen eine Reduktion von 3-SAT auf das Problem vor. 3-SAT ist dabei das Entscheidungsproblem, zu einer gegebenen aussagenlogischen Formel in konjunktiver Normalform mit maximal 3 Literalen pro Klausel zu bestimmen, ob diese *erfüllbar* ist. Ob also eine Belegung der Variablen mit booleschen Werten existiert, welche die Formel wahr werden lässt.

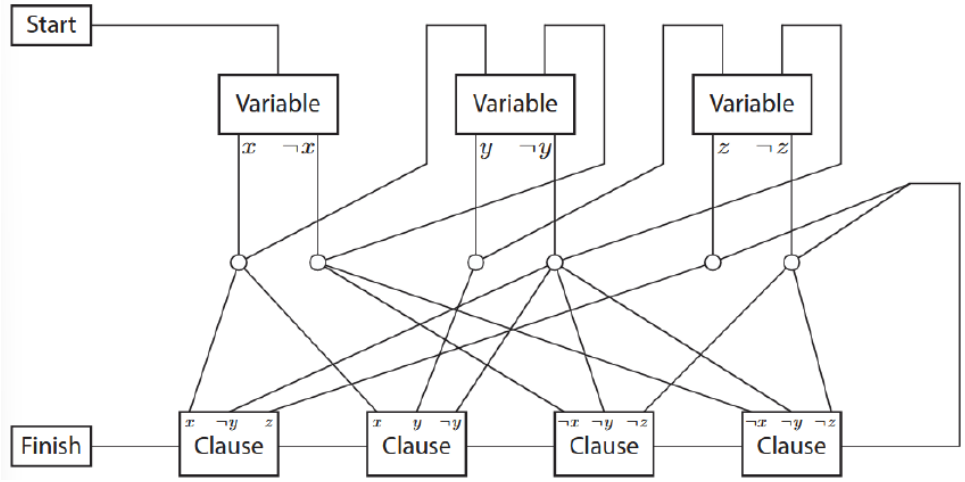
Die grundsätzliche Idee besteht darin, mittels bestimmter Teil-Figuren ein Signal zu simulieren, welches durch die gesamte Figur „geleitet“ wird. An bestimmten Stellen muss sich das Signal „entscheiden“, welchen Pfad es befolgen soll. Auf diese Art und Weise wird eine Entscheidung für eine positive bzw. negative Belegung einer Variable simuliert. Die Variablen sind verbunden mit den Klauseln, in denen sie jeweils vorkommen; falls eine Klausel erfüllt wird, so öffnet sie den Weg für eine spätere Durchleitung des Signals.

Nachdem alle Variablen entschieden wurden, werden alle Klauseln durchgegangen. Nur dann, wenn alle Klauseln erfüllt sind, kann das Signal das Ende erreichen. Um diese Konstruktion zu realisieren werden folgende Teil-Figuren (im folgenden *Gadgets* genannt) benötigt:

1. *Wire-Gadgets* sind dafür zuständig, ein Signal zum nächsten Gadget zu übermitteln. Sie stellen das Grundgerüst für die gesamte Konstruktion dar.
2. *Variable-Gadgets* sind dafür zuständig, eine binäre Entscheidung zu erzwingen. Sie repräsentieren damit die Auswahl einer positiven bzw. negativen Variable.
3. *Clause-Gadgets* sind mittels Wire-Gadgets zu den zugehörigen Variable-Gadgets verbunden. Falls von dem Variable-Gadget ein Signal kommt, so ist das Gadget erfüllt und lässt das Signal an einer anderen Stelle durch, ansonsten nicht.
4. *Fan-Out-Gadgets* teilen ein Signal in zwei Signale auf.
5. *Crossover-Gadgets* erlauben zwei Signalen sich zu kreuzen, ohne dass die Signale von ihren zugehörigen Strecken abweichen.
6. Das *Start-Gadget* kreiert das initiale Signal.
7. Das *Finish-Gadget* empfängt das initiale Signal und speichert ab, ob das Signal empfangen wurde.

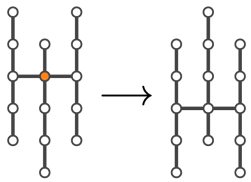
Der gesamte Ablauf kann wie folgt für die Formel $(x \vee \neg y \vee z) \wedge (x \vee y \vee \neg y) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (\neg x \wedge \neg y \wedge \neg z)$ visualisiert werden:

[14] https://www.youtube.com/watchv=unLPk4H1hto, https://youtu.be/oS8m9fSk-Wk?si=TVDR6tpVjNlaoTX5

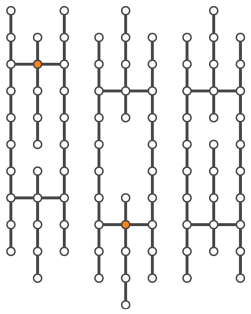


Generelles Gerüst für die Konstruktion, Bild aus diesem Paper [15] entnommen.

Wire- und Start-Gadget Die Grundlage für die gesamte Konstruktion bildet die Übergabe von Signalen. Wir behandeln diese dazu zuerst.

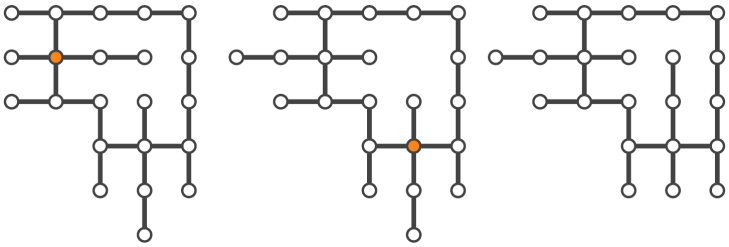


Das Signal wird als das Abhandensein eines Knotens modelliert. Nur falls ein Knoten abhanden ist, ist es möglich, eine gewisse Elementaroperation bei den einzelnen Gadgets auszuführen. Im Falle vom Wire-Gadget lässt sich der mittlere Knoten nur dann drehen, falls oben keine Knoten vorhanden sind. Damit bewegt sich effektiv die Position vom Signal, da nun unten keine Knoten anliegen und das nächste Gadget das Signal empfangen kann. Das folgende Beispiel zeigt, wie mehrere Wire-Gadgets vertikal aneinandergereiht werden können:

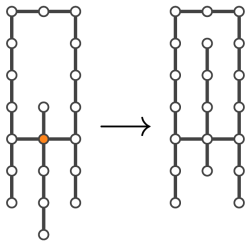


Weiter lässt sich ein Signal wie folgt über Ecken leiten:

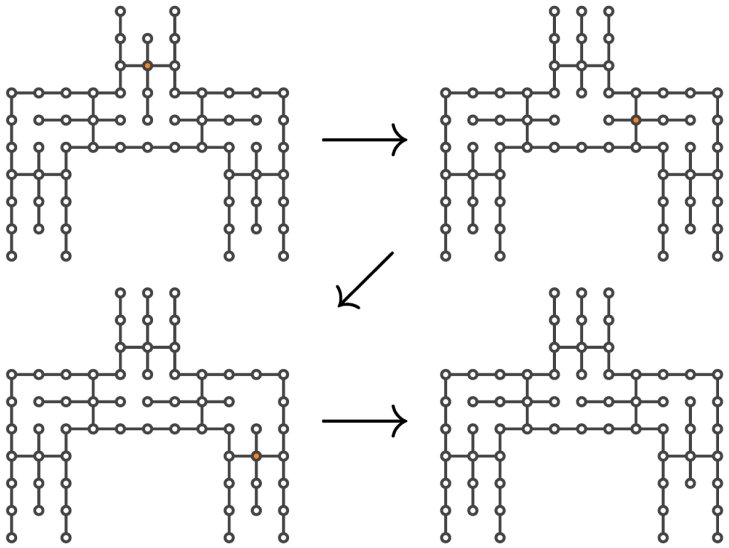
[15] G. Aloupis, E. D. Demaine, A. Guo, und G. Viglietta, „Classic Nintendo games are (computationally) hard“, *Theoretical Computer Science*, Bd. 586, S. 135–160, 2015, doi: https://doi.org/10.1016/j.tcs.2015.02.037.



Indem die Gadgets gedreht werden, kann man Signale somit über das gesamte Koordinatensystem senden. Das Start-Gadget muss ein Signal initiieren, daher muss es lediglich ein Wire-Gadget sein, welches, ohne Einfluss anderer Gadgets, in der Anfangsposition gedreht werden kann:

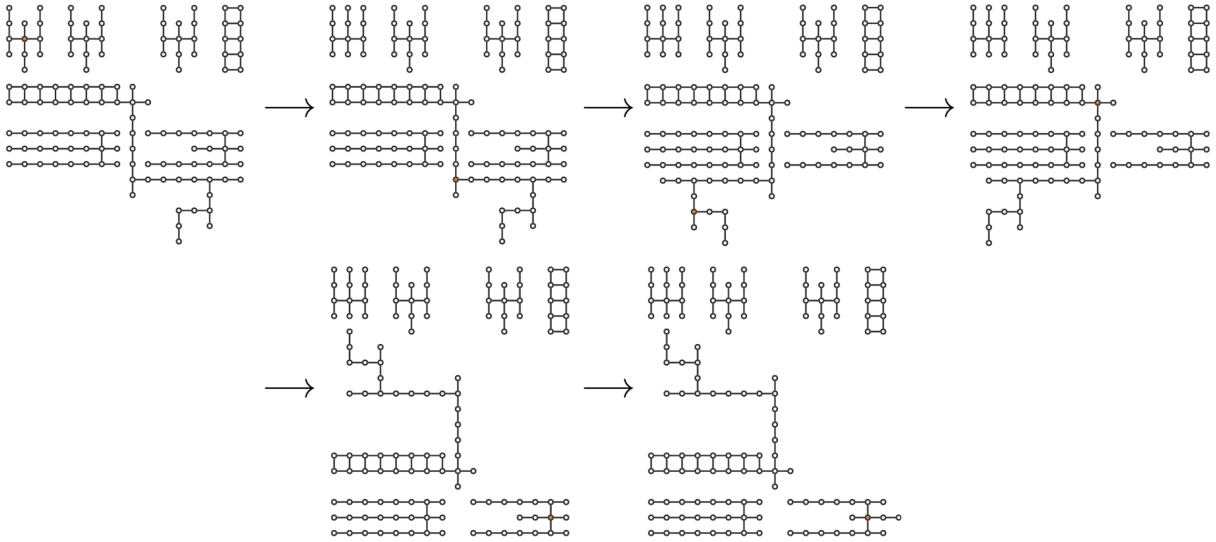


Variable-Gadget Jedes Variable-Gadget entspricht einer Variablen x_i und erzwingt eine binäre Entscheidung - Es kommt von oben ein Signal rein und es kommt entweder links oder rechts vom Gadget raus. Falls sich für den linken Ausgang entschieden wird, so wird ein Signal mittels Wire-Gadgets und Fan-Out-Gadgets an alle Clause-Gadgets gesendet, welche x_i beinhalten. Falls sich für den rechten Ausgang entschieden wird, so wird ein Signal an alle Clause-Gadgets, welche $\neg x_i$ beinhalten gesendet. In diesem Beispiel wurde sich für $\neg x_i$ entschieden:

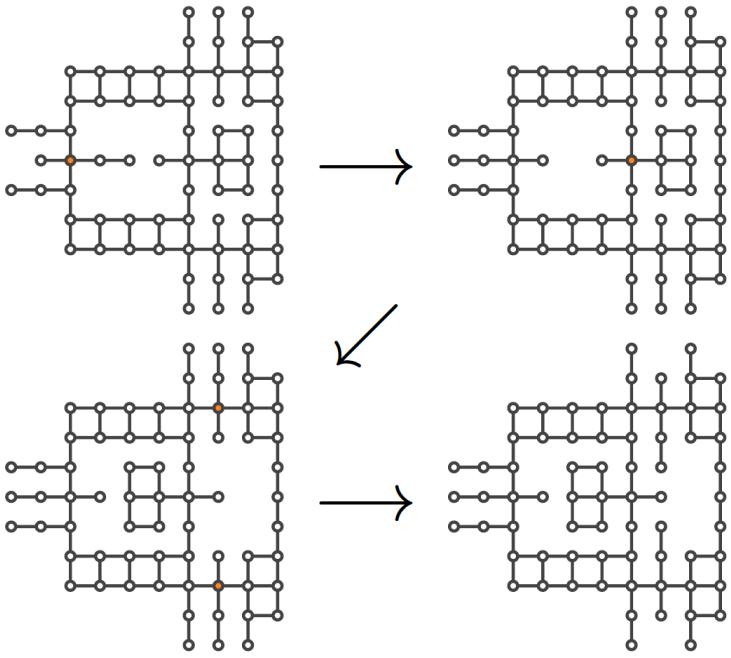


Clause-Gadget Das Clause-Gadget besteht aus drei Bestandteilen. Zum einen befinden sich oben drei Wire-Gadgets, die zu den drei Literalen der Klauseln gehören. Falls ein Signal von oben erreicht wird, so wird Platz gemacht für den mittleren Teil. Der mittlere Teil blockiert

den unteren Teil und kann nur aus dem Weg gehen, falls mindestens eines der Wire-Gadgets vom oberen Teil ein Signal erhält. Der mittlere Teil kann so eingestellt werden, dass ein Signal von jeweils den drei Wire-Gadgets oben erkannt werden kann. Nachdem Platz für den unteren Teil gemacht wurde, kann ein Signal durch das untere Teil passieren. Ein Signal muss durch jeden unteren Teil aller Clause-Gadgets passieren, damit es zum Finish-Gadget kommt; somit wird geprüft ob alle Klauseln erfüllbar sind. Das folgende Beispiel zeigt das Verhalten vom Clause-Gadget falls es durch das erste Literal erfüllt wird:



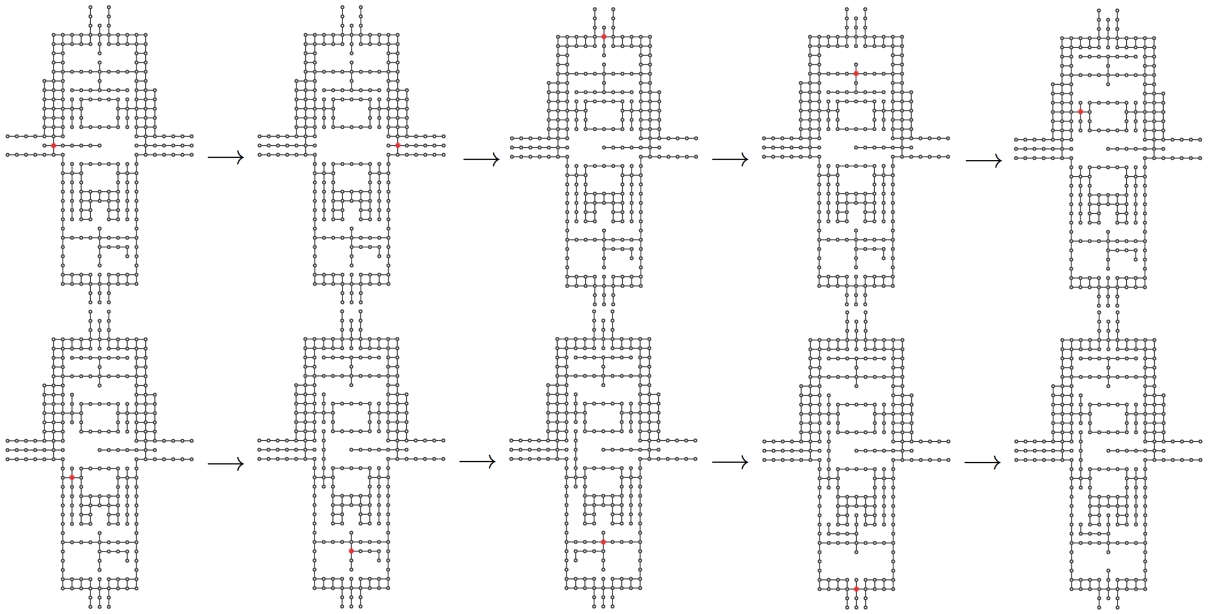
Fan-Out-Gadget Das Fan-Out-Gadget wird verwendet, um das Signal ausgehend von einem Variable-Gadget zu vermehrfachen, sodass ein Signal bei jedem zugehörigen Clause-Gadget ankommt. Dabei kommt von links ein Signal welches verdoppelt werden soll und jeweils oben und unten kommen zwei neue Signale heraus. Das Gadget ist so entworfen, dass das Signal nur in die andere Richtung kann, falls sowohl oben als auch unten jeweils ein Signal vorliegt - In dem Sinne agiert das Gadget wie ein Fan-In, falls man es rückwärts durchgeht.



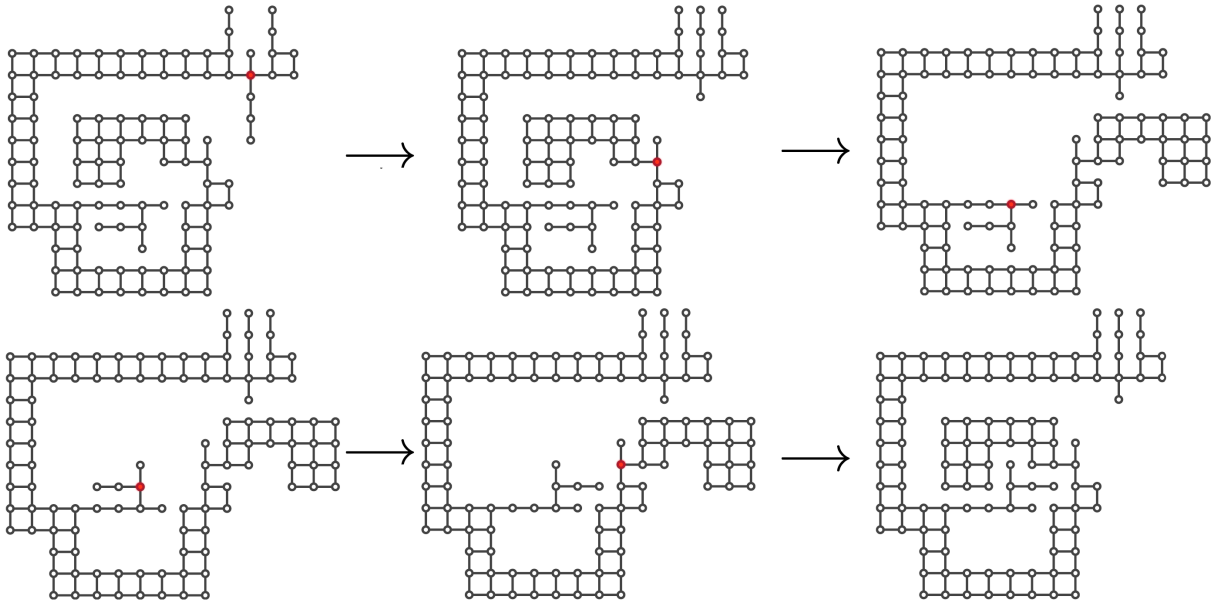
Crossover-Gadget Das hier präsentierte Crossover-Gadget ermöglicht es, zwei Wire-Gadgets auf eine der drei folgenden Arten zu überkreuzen:

1. **Signal von links nach rechts** - Hier ist das eine Ende der Wire-Gadgets besonders lang und das andere besonders kurz, sodass falls sie gedreht werden, sie die Wire-Gadgets von oben nach unten nicht behindern. Ansonsten lässt sich die Übergabe eines Signals von links nach rechts als eine gewöhnliche Kette zweier Wire-Gadgets betrachten.
2. **Signal von oben nach unten** - Hier wird das Signal zunächst mittels einem internen Fan-Out in zwei Signale aufgeteilt. Das linke Wire-Gadget nach dem Fan-Out wird vom Wire-Gadget, welches zur Übergabe von Signalen von links nach rechts zuständig ist, blockiert, daher kann nur das Signal rechts vom Fan-Out passieren. Dieses wird dann wieder mit dem linken Wire-Gadget zusammengeführt und wird als Signal nach unten übergeben.
3. ***zuerst* ein Signal von links nach rechts und dann von oben nach unten** - Zuerst passiert ein Signal von links nach rechts wie in 1. Danach wird das Signal von oben nach unten wie in 2. aufgespalten, dieses Mal wird jedoch die rechte Seite blockiert, daher kann das Signal nur auf der linken Seite passieren. Der Rest läuft wie in 2. ab.

Dieses Design ist nach [1] hinreichend, um alle möglichen Fälle für die Überkreuzung abzudecken, da keine Signale spontan entstehen können oder ein Fall einer Signalübergabe existiert, welcher von den 3 oben genannten abweicht. Fall 3 (als kompliziertester) ist hier illustriert:



Finish-Gadget Das Finish-Gadget „speichert“ sich ab, das ein Signal erreicht wurde, indem eine Folge von Elementaroperationen ausgeführt wird, die nur möglich sind, falls das Gadget nicht von einem Wire-Gadget blockiert wird. Nachdem diese Operationen durchgeführt sind, kann das Signal wieder zurückgesendet werden, da der neue Zustand bereits erreicht wurde und das Wire-Gadget dadurch nicht behindert wird. Da alle Elementaroprationen reversibel sind kann wieder der Anfangszustand der Konstruktion erreicht werden, wobei das Finish-Gadget im neuen Zustand geblieben ist. Das Entscheidungsproblem, ob die gesamte Figur überführt werden kann in die selbe Figur mit dem Finish-Gadget in der anderen Position ist durch diese Konstruktion somit NP-Schwer.



1.5 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Problem adäquat modelliert*: Es gibt zwei zentrale Bereiche der Modellierung: die Darstellung der Figuren und die Drehungen bzw. Drehoperationen. Die Modellierung muss folgende Anforderungen gut unterstützen:

- Einzelne Plättchen oder auch ganze Teile einer Figur dürfen nicht überlappen; der 2D-Charakter der Figuren muss gewährleistet werden.
- Zwei Figuren müssen auf Gleichheit geprüft werden können.
- Die Anwendung einer Drehoperation auf eine Figur muss möglich sein.

Dabei darf eine von der Beschreibung im FAQ abweichende Definition der Drehoperation verwendet werden. Wichtig ist, dass die Modellierung die Drehungen so umsetzt wie in der Dokumentation beschrieben.

(2) *Verfahren nicht unnötig ineffizient*: Brücken bzw. Gelenkpunkte oder andere entscheidende Elemente dürfen nur ein Mal berechnet werden. Suchstrategien müssen zielgerichtet sein.

(3) *Laufzeit des Verfahrens in Ordnung*: Der Zustandsraum aller Figuren kann von oben durch eine exponentielle Funktion abgeschätzt werden. Ein Verfahren, das theoretisch langsamer ist, aber eine akzeptable praktische Laufzeit hat, ist in Ordnung.
Das Programm sollte in der Lage sein, die Beispieleingaben `01` bis `07`, `10` und `11` zu lösen. Ist das der Fall, werden keine Punkte abgezogen. Wenn dies nicht der Fall ist, kann es dafür unterschiedliche Gründe geben. Ein sehr einfaches Verfahren, wie zum Beispiel eine reine Breiten- oder Tiefensuche ohne Pruning oder Verbesserungen reicht nicht aus. Es werden also weitergehende Überlegungen erwartet, etwa zu Isomorphien, Zusammenhangskomponenten und ähnlichen Eigenschaften, mit denen (während der Suche) Möglichkeiten zur Anwendung von Drehungen oder (im Vorhinein) die Überführbarkeit insgesamt ausgeschlossen werden können. Wenn solche Überlegungen nicht existieren bzw. nicht umgesetzt wurden, werden 4 Punkte abgezogen. Gibt es nur sehr einfache Überlegungen, werden bis zu 3 Punkte abgezogen. Wurden aus anderen Gründen die Beispiele nicht gelöst, können bis zu 2 Punkte abgezogen werden.

(4) *Speicherbedarf in Ordnung*: Es muss Ansätze geben, um den schlimmstenfalls exponentiellen Speicherbedarf einzudämmen. Bei besonders effektiven Ansätzen können Pluspunkte vergeben werden.

(5) *Verfahren mit korrekten Ergebnissen*: Das Verfahren muss die Frage nach der Überführbarkeit korrekt entscheiden können. Die Definition der Drehoperation ist dabei maßgeblich für die Korrektheit der Entscheidung. Eine Optimierung, etwa bezüglich der Anzahl der Drehungen für die Überführung, ist nicht gefordert. Für die Anwendung dieses Kriteriums werden nur die Beispiele `01` bis `07`, `10` und `11` berücksichtigt. Werden für diese Beispiele keine korrekten Ergebnisse geliefert, werden bis zu 4 Punkte abgezogen:

(6) *Besonders effektives Verfahren*: Wenn weitergehende Ansätze das Verfahren so weit verbessern, dass alle Beispiele korrekt bearbeitet werden können, insbesondere die Beispiele `08`, `09`, `12` und `13`, können Pluspunkte vergeben werden.

2. Theoretische Analyse

(1) *Verfahren / Qualität insgesamt gut begründet*: Die Interpretation der Drehoperation muss erläutert werden. Es muss erkannt werden, dass eine reine Breiten- oder Tiefensuche nicht ausreicht, um die Aufgabe erfolgreich zu bearbeiten. Falls Maßnahmen zur Einschränkung der Suche angewandt wurden, muss deren Korrektheit begründet sein.

(3) *Vergleichende Analysen*: Wenn die Ergebnisse mehrerer Verfahren kommentarlos angegeben werden, können 2 Punkte abgezogen werden. Für eine Einsendung, die mehrere Verfahren vergleicht und Begründungen für deren Qualität liefert, können Pluspunkte vergeben werden.

(4) *NP-Schwere gezeigt*: Falls die NP-Schwere des Problems erkannt und korrekt begründet wird, etwa durch eine (informelle) Reduktion von einem bekannten NP-schweren Problem, können bis zu 2 Pluspunkte vergeben werden. .

3. Dokumentation

(3) *Vorgegebene Beispiele dokumentiert*: Es müssen alle Beispieleingaben (`0n` bis `13`) dokumentiert sein, ansonsten können Punkte abgezogen werden.
Für alle Beispiele sollen die 1. und 2. Zeile der Ausgabe abgedruckt sein. Außerdem müssen für `0n`, `0y`, `01`, `02`, `03`, `05` und `10` die kompletten Ausgaben in der Dokumentation enthalten sein. Für die restlichen Beispiele müssen diese der Einsendung beiliegen.

(5) *Ergebnisse nachvollziehbar dargestellt*: Es muss erkennbar sein, ob ein Beispiel als lösbar oder unlösbar klassifiziert wird. Außerdem muss die Anzahl der benötigten Drehoperationen angegeben werden. Sollte das Beispiel lösbar sein, müssen die benötigten Zwischenschritte angegeben sein.

Aufgabe 2: Gießroboter

In dieser Aufgabe sollen Gießroboter in einer gegebenen Gartenanlage verteilt werden. Ihre Aufgabe ist es, die dortigen Bäume zu bewässern, wobei jeder der baugleichen Roboter—begrenzt durch seine Akkukapazität—nur eine Maximallänge fahren kann. Ziel ist es, möglichst wenige der Roboter für die vollständige Bewässerung der Anlage zu verwenden.

Jeder der N_B Bäume hat eine Identifikationsnummer (Id) i und Koordinaten $b_i = (x_i, y_i)$; die Maximallänge der Roboterpfade sei $L_{\max}$. Nun soll eine Zuordnung der Bäume auf Routen der Roboter vorgenommen werden. Für jeden der insgesamt N_R verwendeten Roboter j soll dabei ein Startpunkt $s_j = (x_j, y_j)$ für die Ladestation sowie eine Route $R_j = (b_{j,1}, b_{j,2}, \ldots, b_{j,n_j})$ mit Ids der n_j Bäume zugewiesenen. Die Tour eines Roboters beginnt bei seiner Ladestation, führt dann entlang der Route und endet wieder an einer Ladestation, also: $s_j \to b_{j,1} \to \ldots \to b_{j,n_j} \to s_j$. Die Gesamtlänge L_j des Pfades ergibt sich aus der Summe der Teilabschnitte, sodass

$$L_j = d(s_j, b_{j,1}) + \left[\sum_{k=1}^{n_k-1} d(b_{j,k}, b_{j,k+1})\right] + d(b_{j,n_j}, s_j).$$

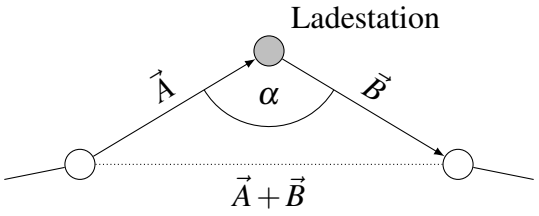
Als Abstandsfunktion verwenden wir den euklidischen Abstand

$$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Eine Route ist gültig, wenn $L_j \leq L_{\max}$. Insgesamt soll jeder Baum mindestens einmal bewässert werden. Die Bäume selbst stellen keine Hindernisse dar und durch sie kann einfach hindurchgefahren werden.

2.1 Grundlegende Lösungsidee

Zunächst bietet es sich an, die Wahl des Startpunktes zu diskutieren. Die Ladestation sollte immer an einem der Bäume entlang der Route eines Roboters (oder auf der Verbindungslinie zwischen zweien) platziert werden, da alle anderen Positionen nur zusätzliche Fahrstrecken erzeugen. Dieser intuitiv zugängliche Fakt kann auch mathematisch untermauert werden. Dafür betrachten wir die unten skizzierte Situation, bei der von der Ladestation der Weg entlang des Vektors $\vec{B}$ zum ersten Baum einer Route und vom letzten Baum der Vektor $\vec{A}$ zurück zur Ladestation gefahren werden muss.



Die Gesamtlänge des Pfades ergibt sich nun aus einem fixierten Teil durch die Wege zwischen den Bäumen und dem Abschnitt zur Ladestation. Sei nun $L_D = |\vec{A} + \vec{B}|$ die Länge des direkten Wegs und $L_L = |\vec{A}| + |\vec{B}|$ die Länge mit Umweg über die Ladestation. Es lässt sich zeigen, dass der direkte Weg immer kürzer als der über die Ladestation ist also $L_D \geq L_L$. Dies ist entspricht der Dreiecksungleichung: die Summe zweier Seitenlänge eines Dreiecks ist stets länger als die

Länge der verbleibenden. Um den Satz zu beweisen, sei α der Winkel zwischen $\vec{A}$ und $\vec{B}$. Der Beweis folgt dann direkt durch die folgende Überlegung:[16]

$$\begin{aligned} L_D^2 = |\vec{A}+\vec{B}|^2 &= (\vec{A}+\vec{B})\cdot(\vec{A}+\vec{B}) = |\vec{A}|^2+|\vec{B}|^2+2\vec{A}\cdot\vec{B} = (|\vec{A}|+|\vec{B}|)^2-2|\vec{A}||\vec{B}|+2\vec{A}\cdot\vec{B} \\ &= L_L^2 - 2|\vec{A}||\vec{B}|(1-\cos\alpha) \leq L_L^2. \end{aligned}$$

Die untere Schranke wird erreicht, wenn $\cos\alpha = 1$, die Ladestation also auf der Verbindungslinie zwischen dem ersten und letzten Baum einer Route liegt. In dieser Lösung wird deswegen die Ladestation direkt am ersten Baum einer Route platziert.

Um für eine Gartenanlage eine möglichst sparsame Routenbelegung zu finden, konstruieren wir zunächst mit einem Greedy Algorithmus eine (gute) gültige Startlösung, wie in Abschnitt 2.2 erklärt. Kernelement dafür ist ein Ansatz zur Lösung des Problems des Handlungsreisenden, der in Abschnitt 2.3 diskutiert wird. In Abschnitt 2.4 wird schlussendlich ein Verfahren diskutiert, womit die erhaltene Lösung durch einen Clustering-Ansatz verfeinert wird.

2.2 Greedy Aufbau der Roboterrouten

Ausgehend von einem Startknoten erweitern wir den Cluster sukzessive um die Knoten, bei denen wir annehmen, dass sie eine kürzestmögliche Route ergeben. Dafür wird einerseits eine Liste mit noch nicht bewässerten Bäumen im Garten benötigt, die wir G' nennen. Die Länge der Liste ist $|G'|$, also die Anzahl noch nicht bewässerter Bäume.

Auswahl des Startknotens. Um den Startknoten auszuwählen, bieten sich Heuristiken an, die vom Rand des Gartens (oder zumindest seines noch nicht bewässerten Teils) nach innen arbeiten. Damit kann verhindert werden, dass sich am Ende der Prozedur vereinzelte Bäume am Rand befinden. Diese müssten dann—der Erwartung nach—mit ineffizienten Roboterrouten aufgefangen werden. Durch das Vorgehen von Außen nach Innen kann dies verhindert werden.

Ausgehend von den noch verbleibenden Routen in G' berechnen wir den Mittelpunkt μ', dessen Koordinaten sich wie folgt ergeben:

$$\mu'_x = \frac{1}{|G'|}\sum_{i\in G'} x_i, \qquad \mu'_y = \frac{1}{|G'|}\sum_{i\in G'} y_i.$$

Er kann entsprechend in $\mathcal{O}(N_B)$ berechnet werden und die Auswahl des Baumes mit dem größten Abstand zum Mittelpunkt erfolgt ebenfalls in $\mathcal{O}(N_B)$.

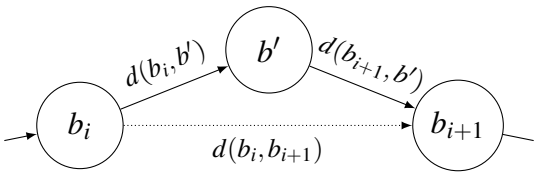
Zwar ist der Mittelpunkt des Gartens eine gute Wahl als Zielpunkt, für große Beispieldateien bietet es sich allerdings zusätzlich an, auch Punkte abseits der Mitte zu versuchen. Dafür implementieren für die finalen Ergebnisse ebenfalls die Wahl eines zufälligen Baums als angestrebten Mittelpunkt. Dadurch können verschiedene Variationen von Routenplänen erzeugt werden, die möglicherweise insgesamt weniger Roboter benötigen.

Aufbau der Route. Als nächsten Schritt wird die Route ausgehend vom Startpunkt so lange erweitert, bis sie die Maximallänge $L_{\max}$ überschreitet. Dafür bietet es sich an zwei Größen

[16] Für das gewohnte Skalarprodukt gilt $|A|^2 = \vec{A}\cdot\vec{A}$ und $\vec{A}\cdot\vec{B} = |A||B|\cos\alpha$.

miteinander auszugleichen: einerseits die Strecke, um die eine Route durch das Hinzufügen eines Baums länger wird. Andererseits sollte die Route sich nicht zu sehr in die Mitte des Gartens „fressen“, sondern auch leicht unvorteilhafte Bäume, welche dafür näher am Rand liegen, berücksichtigen.

Fügen wir den Baum b' entlang einer Route zwischen den Bäumen b_i und b_{i+1} ein, kann der nötige Umweg Δ_i durch das folgende Dreieck berechnet werden:



Damit ergibt sich

$$\Delta_{b',i} = d(b_i, b') + d(b', b_{i+1}) - d(b_i, b_{i+1}). \tag{2.1}$$

Der nötige Umweg um den Baum in einer Route einzufügen, ist entsprechend $\Delta_{b'} = \min_i \Delta_{b',i}$, was in $\mathcal{O}(N_B)$ berechnet werden kann. Um zusätzlich das anfangs erwähnte „Einfressen“ zu verhindern, berechnen wir für jeden Knoten den Abstand zur Mitte

$$D_i = |\mu' - b_i|.$$

Aus beiden Größen berechnen wir den Score $s_i = \Delta_i - \eta D_i$, wobei der Gewichtungsparameter $\eta > 0$ frei wählbar ist. Als nächsten Kandidaten für die Route fügen wir den Baum hinzu, der den niedrigsten Score hat; also einen möglichst kleinen Umweg Δ erzeugt und einen möglichst großen Abstand D zur Mitte hat. Empirisch wurde $\eta = 0.2$ als guter Wert gefunden, bei dem möglichst gute Lösung erreicht werden. Dieser Wert wird entsprechend in der gesamten Lösung verwendet. Nach $N_a = 5$ hinzugefügten Bäumen führen wir eine Iteration des in Abschnitt 2.3 eingeführten TSP Algorithmus aus, um die Route weiter zu verbessern.

Effizientere Datenstruktur. Für die Suche nach Kandidaten und den in Abschnitt 2.4 eingeführten Verbesserungsalgorithmus bietet es sich an, die Bäume zusätzlich in einer anderen Datenstruktur zu initialisieren, die wir PROXIMITYGRAPH nennen. Dort speichern wir für jeden Baum i alle Nachbarbäume, die höchstens $L_{\max}$ von ihm entfernt sind. Dafür benötigen wir einmalig Laufzeit und Speicher $\mathcal{O}(N_b^2)$.

2.3 Finden einer kurzen Rundreise

Um die Roboterrouten optimal festlegen zu können, muss für eine gegebene Menge von Bäumen (eine Teilmenge des gesamten Gartens) eine kürzest-mögliche Route zwischen diesen gefunden werden. Damit kann aus einer unsortierten Zuordnung von Bäumen zu einem Roboter entschieden werden, ob eine Route unterhalb der Maximallänge $L_{\max}$ existiert. Die naiv zusammengestellten Routen aus Abschnitt 2.2 können damit verbessert werden.

Für das Problem des Handelsreisenden (TSP) existieren viele Lösungsansätze. Im Rahmen dieser Aufgabe gilt es, viele, eher kleine Probleme zu lösen. Dazu nutzen wir bei kleinen Problemgrößen den dynamischen Held-Karp Algorithmus, welcher exakt die kürzeste Route berechnet.

Größere Probleme werden genähert mit einem Greedy Ansatz gelöst, der eine vorhandene Route bestmöglich erweitert und einige Abschnittsvertauschungen ausprobiert. Verglichen werden diese beiden Lösungsansätze mit einer Formulierung als Problem der ganzzahligen linearen Optimierung (ILP – Integer Linear Programming), womit für alle hier relevanten Routen die exakt kürzeste Reihenfolge gefunden werden kann.

Held-Karp Algorithmus

Die Idee des Held-Karp Algorithmus ist es, alle möglichen Routen durch die n Bäume $B = \{b_1, \ldots, b_n\}$ durch geschicktes dynamischen Buchführen zu betrachten. Dafür wird ein „Startbaum“ b_1 festgelegt. Ausgehend von diesem wird für alle Teilmengen $S \subseteq B \setminus b_1$ und $b \in S$ eine Funktion $g(S,b)$ definiert, welche die Länge des kürzesten Wegs von b_1 nach b über alle Bäume in S angibt. Als Beispiel: ist $S = b_2, b_3, b_4$ und $b = b_4$, so gibt $g(S,b)$ die Länge des kürzesten Wegs von b_1 nach b_4 über b_2 und b_3 an; dies können nur $b_1 \to b_2 \to b_3 \to b_4$ oder $b_1 \to b_3 \to b_2 \to b_4$ sein. Die Funktion g kann für die trivialen Strecken vom Startpunkt zu allen anderen Bäumen initialisiert werden, da für alle $i > 1$ gilt, dass $g(\{b_i\}, b_i) = d(b_1, b_i)$.

Nun kann g dynamisch berechnet werden. Für die kürzeste Route von b_1 nach b über S werden dafür alle Vorgänger $b_k \in S \setminus b$ von b betrachtet. Die kürzeste Route kann dann aus den kürzesten Routen zu allen Vorgängern berechnet werden, in dem der Vorgänger mit der kürzesten Gesamtstrecke ausgewählt wird, sodass

$$g(S,b) = \min_{b_k \in S} g(S \setminus b, b_k) + d(b_k, b).$$

Ausgehend von der trivialen Strecken mit einem Baum können die mit zwei, drei,… Bäumen berechnet werden. Die Länge L des kürzesten Rundwegs ist dann

$$L = \min_{b_k \neq b_1} g(\{b_2, \ldots, b_n\}, b_k) + d(b_k, b_1).$$

Zusätzlich zur Länge kann der kürzeste Pfad dadurch rekonstruiert werden, dass man für jeden Eintrag in $g(S,b)$ den Vorgänger von b mitführt. In Algorithmus 1 ist ein Pseudocode für den Held-Karp Algorithmus präsentiert. Aus den drei die Laufzeit dominierenden Schleifen ergibt sich für den Held-Karp Algorithmus das asymptotische Verhalten $\mathcal{O}(n^2 2^n)$. Die Funktion g hat am Ende $\mathcal{O}(n2^n)$ Einträge, was dem hauptsächlichen Speicheraufwand entspricht.

Implementierungsdetails. Um die Funktion g effizient verwenden zu können, bedarf es einer effizienten Datenstruktur für ihre Speicherung. Die Mengen S können dabei in 64-Bit Ganzzahlen gespeichert werden, bei denen das Bit an der Stelle i gesetzt ist, wenn $b_{i-1} \in S$. Für jeden Baum b kann $g(S,b)$ dann in einem Array der Länge 2^{n-2} gespeichert werden. Gleiches gilt für den Vorgänger. Um alle Mengen S mit einer bestimmten Länge $s = |S|$ zu iterieren, kann ein Startbitstring aus $2^s - 1$ initialisiert werden und dann jede nächste Kombination durch Bithacks generiert werden.[17]

[17] Die Bitstrings werden in lexikographischer Ordnung effizient generiert. Dabei hat die Ordnung an sich keine weitere Bedeutung; relevant ist, dass alle Bitstrings mit genau s gesetzten Bits generiert werden. Ein Beispiel in C ist hier zu finden: `https://graphics.stanford.edu/~seander/bithacks.html#NextBitPermutation`.

Algorithmus 1 Held-Karp Algorithmus zum Bestimmen der kürzesten Rundreise durch B

function HELDKARP(Bäume $B = \{b_1, \dots b_n\}$)
  for $k = 2, \dots, n$ **do**
    Initialisiere $g(\{b_k\}, b_k) = d(b_1, b_k)$
  end for
  for $s = 2, \dots, n-1$ **do**
    for all $S \subseteq \{b_2, \dots b_n\}$ **mit Länge** $|S| = s$ **do** ▷ $\mathcal{O}(2^n)$
      for $b \in S$ **do** ▷ $\mathcal{O}(n)$
        $g(S, b) = \min_{b_k \in S, b_k \neq b} g(S \setminus b_k) + d(b_k, b)$ ▷ $\mathcal{O}(n)$
      end for
    end for
  end for
  return $\min_{b_k \neq b_1} g(\{b_2, \dots, b_n\}, b_k) + d(b_k, b_1)$
end function

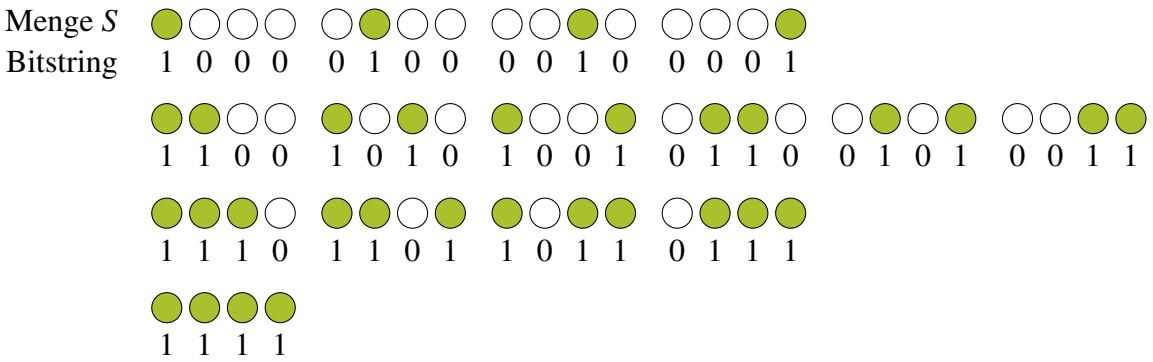


Abbildung 2.1: Beispielhafte Darstellung aller 15 nicht-leeren Teilmengen S aus vier Bäumen $B = \{b_2, \dots, b_5\}$ mit der entsprechenden Kodierung als Bitstring. Grün markierte Kreise sind Bäume, die in S enthalten sinds. Die Bitstrings in jeder Zeile sind zusätzlich lexikographisch sortiert, was der Iterationsreihenfolge in Algorithmus 1 entspricht.

Greedy TSP (Nächster-Baum Erweiterung)

Für größere Routen[18] bietet sich der folgende Greedy Algorithmus aus: Ausgehend von der Menge an zu ordnenden Bäumen $B = \{b_i\}$, wird zunächst ein Baum an die Spitze der Route R gestellt. Danach wird von B wiederholt der Baum entfernt, der sich entsprechend Gleichung (2.1) am besten in die bereits konstruierte Route einbauen lässt. Am Ende ist B leer und eine Näherung für die Route gefunden.

Paarweises Kantentauschen. Durch das naive Hinzufügen des nächstbesten Baums zur Route werden schnell suboptimale Routen konstruiert, da die Routen selber danach nicht mehr verändert werden können. Um dies abzuschwächen, kann zusätzlich ein Kantentauschen (bekannt als Lin–Kernighan Heuristik) implementiert werden. Wir beschränken uns dabei auf die in Abbildung 2.2 dargestellten Möglichkeiten, für die jeweils zwei oder drei Kantenpaaren getauscht werden. Werden nur Vertauschungen zweier Kanten betrachtet, nennen wir die Heuristik 2-OPT, werden zusätzlich Vertauschungen von drei Kanten bei denen zwei benachbart

[18] Praktisch definieren wir große Routen als solche mit mehr als 25 Bäumen.

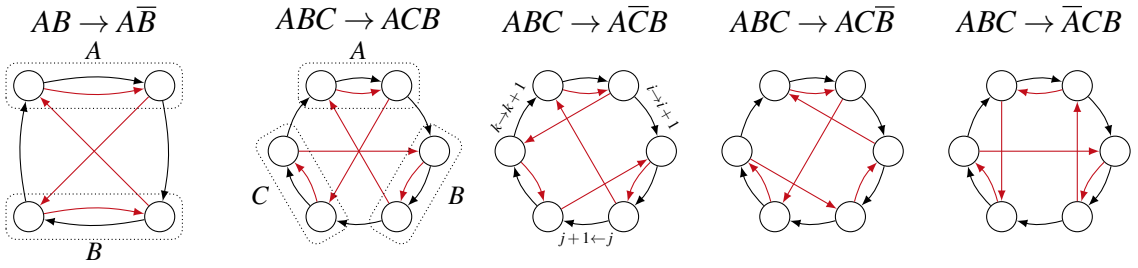


Abbildung 2.2: Darstellung der fünf Vertauschungen für den Greedy TSP Algorithmus, bei dem zwei beziehungsweise drei Kanten berücksichtigten werden. Die schwarzen Pfeile markieren den Weg vor der Vertauschung, die roten den danach. Aus dem Pfad mit den Segmenten AB oder ABC (bei drei getauschten Kanten) werden die Pfade $A\overline{B}$, ACB, $A\overline{C}B$, $AC\overline{B}$ und $\overline{A}CB$, wobei $\overline{X}$ bedeutet, dass X nach der Vertauschung in entgegengesetzter Richtung durchlaufen wird. Im dritten Diagramm sind die Indices i, j und k aus Gleichung (2.2) eingetragen.

sind betrachtet 2.5-OPT und bei der Berücksichtigung aller Vertauschungen mit drei Kanten 3-OPT.

Implementierungsdetails. Damit die Vertauschungen schnell implementiert werden können, werden die Routen als doppelt verkettete Liste implementiert. In dieser werden für jeden Baum der Route sein Vorgänger beziehungsweise Nachfolger gespeichert. Die Route kann dann durchlaufen werden, in dem für einen beliebigen Startbaum ein fiktiver Vorgänger gewählt wird und an dann sukzessive der verbleibende Nachfolger bestimmt wird. Dadurch können Vertauschungen, bei denen die Durchlaufreihenfolge invertiert wird, in $\mathcal{O}(1)$ implementiert werden.

Es gibt jeweils $\mathcal{O}(n^2)$ beziehungsweise $\mathcal{O}(n^3)$ Kantenpaare beim Tauschen von zwei beziehungsweise drei Kanten. Allerdings müssen nach dem Hinzufügen eines neuen Baums nur Vertauschungen mit den jeweils zwei neu entstandenen Kanten überprüft werden, wodurch sich die Laufzeit auf $\mathcal{O}(n)$ und $\mathcal{O}(n^2)$ reduziert. Um die Vertauschungen zu bestimmen, werden zunächst solche mit zwei Kanten und anschließend die mit drei Kanten durchprobiert. Die günstigste wird jeweils gewählt. Seien die Wege $b_i \to b_{i+1}$ und $b_j \to b_{j+1}$ die Kanten, die getauscht werden sollen. Dann ist der Kantentausch vorteilhaft, wenn

$$d(b_i,b_j)+d(b_{j+1},b_{i+1}) < d(b_i,b_{i+1})+d(b_i,b_{j+1}).$$

Für drei Kanten mit den Vertauschungen von $b_i \to b_{i+1}$, $b_j \to b_{j+1}$ und $b_k \to b_{k+1}$ berechnen wir die Längen $L_0 = d(b_i,b_{i+1})+d(b_j,b_{j+1})+d(b_k,b_{k+1})$ und

$$\begin{aligned}
L_{ACB} &= d(b_i,b_{j+1})+d(b_k,b_{i+1})+d(b_j,b_{k+1}),\\
L_{A\overline{C}B} &= d(b_i,b_k)+d(b_{j+1},b_{i+1})+d(b_j,b_{k+1}),\\
L_{AC\overline{B}} &= d(b_i,b_{j+1})+d(b_k,b_j)+d(b_{i+1},b_{k+1}),\\
L_{\overline{A}CB} &= d(b_{k+1},b_{j+1})+d(b_k,b_{i+1})+d(b_j,b_i).
\end{aligned} \tag{2.2}$$

Ist eine dieser vier Längen kleiner als L_0 wird der jeweils günstigste Tausch ausgeführt.

Auswahl des Startbaums. Die gefundene Route hängt vom Startbaum ab. Um eine möglichst optimale Route zu finden, können beispielsweise mehrere zufällige Bäume oder jeder auf

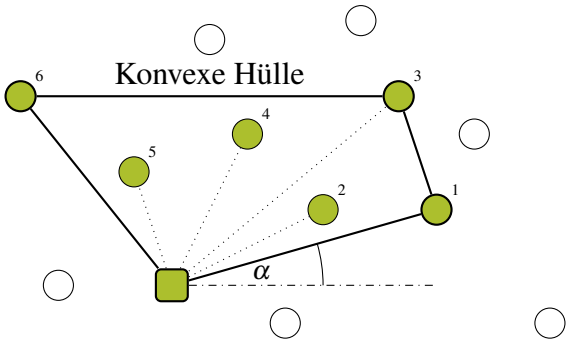


Abbildung 2.3: Illustration der konvexen Hülle einer Route, der die grün markierten Bäume zugeordnet sind. Die sie aufspannenden Punkte dienen als Startpunkte für den Greedy-Algorithmus zum Finden einer kurzen Rundreise (TSP). Um die konvexe Hülle zu konstruieren, wird ein Punkt am Rand (hier der unterste, markiert als Quadrat) ausgewählt und ausgehend von diesem werden alle anderen nach ihrem Winkel α zum Startpunkt sortiert (hier von rechts nach links). Die Sortierung ist beispielhaft durch die kleinen Zahlen symbolisiert.

der konvexen Hülle der Route als Start gewählt werden. Hinter der Verwendung der konvexen Hülle steckt die Idee, dass Punkte am Rand einen natürlichen Startpunkt für den Aufbau der Route bilden und die konvexe Hülle eine über den ganzen Rand verteilte Auswahl dieser darstellt. In Abbildung 2.3 ist eine solche konvexe Hülle illustriert.

Zum Konstruieren einer konvexen Hülle um die Bäume $B = \{b_1, \dots b_n\}$ in $\mathcal{O}(n \log n)$ kann der Graham-Scan verwendet werden. Dafür muss zunächst ein Baum b_* am Rand gefunden werden, beispielsweise kann b_* der unterste sein. Nun werden alle anderen Bäume b_j entsprechend des Winkels sortiert, den die Strecke $b_j b_*$ mit einer horizontalen durch b_* einnimmt. Der Winkel ist in Abbildung 2.3 konstruiert. Für die Implementierung muss der Winkel nun nicht explizit berechnet werden: eine Funktion die unterscheidet, ob der Winkel einer Strecke $b_i b_*$ größer, gleich oder kleiner der von $b_j b_*$ ist genügt. Bildet man das Kreuzprodukt der Vektoren $\vec{A} = b_i b_*$ und $\vec{B} = b_j b_*$, hat dieses eine positive z-Komponente wenn $\vec{A}$ „rechts“ von $\vec{B}$, andernfalls ist sie negativ (oder null, wenn die Vektoren kolinear sind). Es reicht also, die Größe

$$Z(b_i, b_*, b_j) = (\vec{A} \times \vec{B})_z = (y_* - y_i)(x_j - x_*) - (x_* - x_i)(y_j - y_*)$$

zu berechnen. Damit kann durch $Z > 0$ oder $Z < 0$ entschieden werden, ob b_i vor oder hinter b_j sortiert werden muss. Wenn Z null ist, wird der Baum, welcher näher an b_* ist, vor dem anderen einsortiert. Der Graham Scan geht dann ausgehend vom Startknoten entlang der sortierten Liste und entfernt alle Bäume, die links von ihren Vorgängern und Nachfolgern sind.

ILP-Formulierung

Um ebenfalls die exakt kürzeste Route verwenden zu können, kann TSP als ganzzahliges lineares Optimierungsproblem (ILP) formuliert werden. Grundsätzlich bleibt die NP-Schwere dadurch erhalten, allerdings existieren für ILP spezialisierte Optimierer, die alle für uns relevanten Problemgrößen schnell und exakt lösen. Konkret verwenden wir Gurobi. Die ILP Formulierung wird ähnlich wie die beiden vorherigen Verfahren nur auf die vom in Abschnitt 2.2 eingeführten Algorithmus bestimmten Teilmengen angewandt und bestimmt für diese die kürzeste Route.

Um das Problem des Handelsreisenden als ILP zu formulieren, werden die binären Variablen

$y_{ij} \in \{0,1\}$ eingeführt. Dabei gibt $y_{ij} = 1$ an, dass die Kante von Knoten i nach j Teil des kürzesten Pfades ist; andernfalls ist $y_{ij} = 0$. Die Optimierungsgröße ist dann die Gesamtlänge: sei dafür d_{ij}, der Abstand von i und j, dann besteht die Aufgabe darin,

$$\sum_{ij} d_{ij} y_{ij}$$

zu optimieren. Nicht jede Belegung der y_{ij} entspricht einer gültigen Rundreise. An jedem Knoten muss es eine eingehende und eine ausgehende Kante geben, also muss für alle i gelten, dass $y_{ii} = 0$ und dass

$$\sum_{j} y_{ij} = 2.$$

Außerdem muss die Rundreise zusammenhängend sein und darf nicht aus einzelnen Routen bestehen. Um dies sicherzustellen, verwenden wir die Dantzig–Fulkerson–Johnson Formulierung. Bei dieser wird ausgeschlossen, dass eine abgeschlossene Route für die Knoten $Q \subsetneq B$ existiert. Dafür wird zusätzlich für alle möglichen Q mit mindestens zwei Elementen die Bedingung

$$\sum_{i \in Q} \sum_{\substack{j \neq i \\ j \in Q}} y_{ij} \leq |Q| - 1$$

eingeführt. Anschaulich zählt die Summe die vollständig in Q liegenden Kanten. Würde diese der Anzahl der Elemente in Q entsprechend, wäre die Route in Q vom Rest isoliert und die gesamte Route wäre keine Rundreise. Da es exponentiell viele solche Teilmengen Q gibt, wird die Bedingung in der Praxis nur eingefügt, wenn eine vermeintliche Lösung y_{ij} mit solchen getrennten Routenteilen gefunden wurde. Für genau diese Konstellation wird dann die Randbedingung eingefügt und das Programm aktualisiert.[19] Zusammengefasst gilt es das folgende ILP zu lösen:

$$\begin{aligned}
&\min\left(\sum_{ij} d_{ij} y_{ij}\right): \\
&\qquad \sum_{j} y_{ij} = 2, && \forall i \in B; \\
&\qquad y_{ii} = 0, && \forall i \in B; \\
&\qquad \sum_{i \in Q} \sum_{\substack{j \neq i \\ j \in Q}} y_{ij} \geq |Q| - 1, && \forall Q \subsetneq B, |Q| \geq 2.
\end{aligned}$$

2.4 Verfeinerung fertiger Zuordnungen

Naturgemäß produziert der Greedy Routenbau nur eine Näherungslösung, für die noch kleine Optimierungen möglich sind. Hierfür eignet sich eine Variation des Hartigan-Wong Algorithmus, mit dem üblicherweise Clustering von Daten vorgenommen wird. Abhängig von der verwendeten Kostenfunktion kann hier entweder eine Balance der Routen erzielt werden, sodass diese möglichst gleichlang sind, oder eine ungültige Lösung kann hin zu einer gültigen optimiert werden. Zum Verbessern der Routen werden diese zunächst balanciert. Dann werden geeignete Routen gelöscht und die ihnen zugeordneten Bäume auf andere Routen umverteilt.

[19] Siehe Tutorial von Daniel Schermer zum Traveling Salesperson Problem für JuMP.jl.

Dadurch entstehen zu lange und damit ungültige Routen, die—wenn möglich—zu gültigen ausgeglichen werden. Ist dies nicht möglich, wird die ursprüngliche Route wieder eingefügt und das Entfernen einer anderen versucht.

Hartigan-Wong Algorithmus

Der Hartigan-Wong Algorithmus ist ursprünglich als Lokale Suche zum Finden eines k-Means Clusterings formuliert, kann aber angepasst werden, um die Beschränkungen der Roboterrouten zu berücksichtigen. Hier wird zunächst die ursprüngliche Idee erklärt, im nächsten Abschnitt dann die Anpassung an das Gießroboter Problem. Beim k-Means Clustering sollen für eine Menge an Punkten $\vec{x}_1, \ldots, \vec{x}_n$ insgesamt k Cluster $S_1, \ldots, S_k$ um die Mittelwerte $\vec{\mu}_1, \ldots, \vec{\mu}_k$ gebildet werden. Die Punkte sollen dabei „möglichst nah“ an den Mittelwerten sein, so dass die Varianz

$$\text{Var} = \sum_{i=1}^{k} \underbrace{\sum_{\vec{x} \in S_i} |\vec{x} - \vec{\mu}_i|^2}_{\Phi(S_i)}$$

minimiert ist. Beim Hartigan-Wong Algorithmus wird dann die Kostenfunktion $\Phi(S_i)$ aus den Gesamtkosten eines Clusters definiert. Nun wird für jeden Knoten $x \in S_n$ berechnet, ob es sich lohnt, diesen stattdessen der Menge S_m zuzuordnen. Solange es einen Schritt gibt, mit dem die Varianz abnimmt, wird der lohnenswerteste ausgeführt.

Im Fall der Gießroboter kann der Hartigan-Wong Algorithmus ähnlich verwendet werden, wobei zusätzlich noch die Beschränkung durch die Maximallänge berücksichtigt werden muss. Für eine Route R wird dafür eine Kostenfunktion $\Phi(R)$ definiert, welche die Abweichung vom Wunschzustand misst. Der Wunschzustand ist entweder, dass die Routen möglichst gleichlang sind oder dass die Routen gültig sind; Details zur Kostenfunktion folgen im nächsten Abschnitt. Ausgehend von der ursprünglichen Zuordnung in die N_R Routen $\{R_j \mid j \in 1, \ldots, N_R\}$ berechnen wir für jedes Paar von Clustern $n, m \in 1, \ldots, N_R$ und jedes $x \in R_n$ den Gewinn, wenn x in R_m verschoben wird als

$$G(x \in R_n \to R_m) = \Phi(R_n) + \Phi(R_m) - \Phi(R_n \setminus x) - \Phi(R_m \cup x). \tag{2.3}$$

Für den Eintrag mit dem größten Gewinn wird der entsprechende Schritt ausgeführt. Wenn kein Gewinn mehr erzielt werden kann, also alle $G \leq 0$ sind, ist der Durchlauf beendet. Die Funktion G zu berechnen bedarf $\mathcal{O}(N_B N_R)$ Operationen. Allerdings kann nach dem ersten Aufstellen auf ein vollständiges Update verzichtet werden und stattdessen werden nach dem Verschieben eines Baums nur dessen erreichbare Nachbarn neu berechnet. Diese sind im PROXIMITYGRAPH gespeichert. Außerdem genügt es, das Tauschen in die veränderten Routen zu überprüfen. In Algorithmus 2 ist das Verfahren als Pseudocode skizziert. Als Abbruchbedingung führen wir neben der Konvergenz des Algorithmus noch ein Iterationslimit von 2000 ein, was in der Praxis bei keiner der Beispieldateien erreicht wird. Da die meisten Routen fast optimal gefüllt sind, können fast nur die Bäume der nahegelegenen Routen verschoben werden, wodurch ungefähr $\mathcal{O}(N_B/N_R)$ Verschiebungen zu erwarten sind.

Algorithmus 2 Hartigan-Wong Algorithmus zum Optimieren bestehender Routen

```
function HartiganWong(Routen ℜ = {R_1, ... R_{N_R}}, Kostenfunktion Φ)
    for all R_i ∈ ℜ do
        Entferne R_i aus ℜ und verteile die Bäume auf benachbarten Routen
        Initialisiere G aus Gleichung (2.3)
        while ein Eintrag in G > 0 and Iterationen < Limit do
            Finde optimale Verschiebung x ∈ R_n → R_m in G und führe sie aus
            Aktualisiere die betroffenen Einträge in G
        end while
        Verbleiben ungültige Routen, stelle den Ausgangszustand wieder her
    end for
end function
```

Kostenfunktionen

Abhängig von der Wahl der Kostenfunktion können unterschiedliche Änderungen an den Routen vorgenommen werden. Als Teil dieser Lösung soll mit Φ_{bal} eine balancierte Verteilung der Routenlängen erreicht werden, sodass alle möglichst gleich lang sind. Mit Φ_{val} soll aus einer ungültigen eine gültige Lösung produziert werden. Beide benötigen die Länge $L(R)$ einer Route R und sind wie folgt definiert:

$$\Phi_{\text{bal}}(R) = \begin{cases} L(R)^2, & \text{wenn } L(R) < L_{\max}, \\ \infty, & \text{sonst,} \end{cases}$$
$$\Phi_{\text{val}}(R) = \max(0, L(R) - L_{\max}).$$

Für $\Phi_{\text{bal}}(R)$ werden also alle gültigen Routen so kurz wie möglich gewählt und ungültige Routen durch das unendliche Gewicht ausgeschlossen. Bei $\Phi_{\text{val}}(R)$ werden dagegen alle gültigen Routen direkt mit dem Gewicht null bewertet, ohne auf deren konkrete Länge einzugehen.

2.5 Laufzeit

In Tabelle 1 sind die im Haupttext hergeleiteten Laufzeiten der einzelnen Routinen zusammengefasst.

Greedy Clustering (Abschnitt 2.2)

N_B Bäume im gesamten Garten.

Auswahl des Startknotens	$\mathcal{O}(N_B)$
Aufbau der Route	$\mathcal{O}(N_B)$

Problem des Handelsreisenden (Abschnitt 2.3)

n Bäume in der aktuellen Route.

Held-Karp	$\mathcal{O}(n^2 2^n)$
Greedy TSP	$\mathcal{O}(n^2)$ (Routenaufbau ausgehend von der konvexen Hülle)
	$\mathcal{O}(n^3)$ (Kantentauschen)

Hartigan-Wong Verfeinerung (Abschnitt 2.4)

N_R Routen und N_B Bäume im gesamten Garten.

Entfernte Routen	$\mathcal{O}(N_R)$
Berechnung von G	$\mathcal{O}(N_B N_R)$
Iterationen	$\mathcal{O}(N_B/N_R)$, maximal 2000

Tabelle 1: Zusammenfassung der Laufzeit der einzelnen Algorithmuskomponenten.

2.6 Beispiele

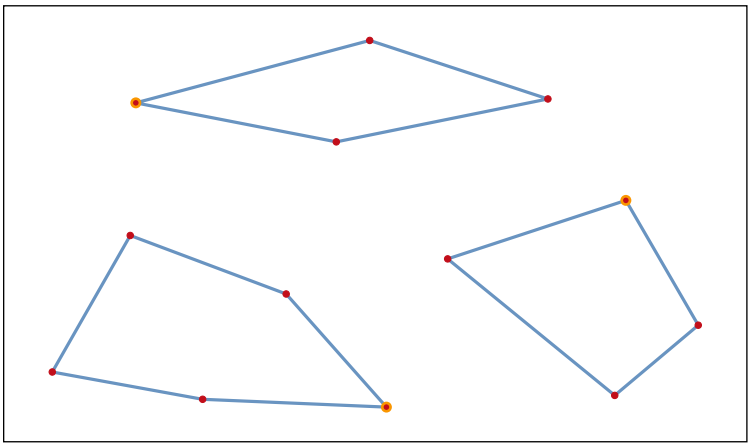
Eingabedatei	**Roboteranzahl**					
	GREEDY	+ OPT-2	+ OPT-2.5	+ OPT-3	+ HELDKARP	ILP
`roboter01.txt`	3	3	3	3	3	3
`roboter02.txt`	8	8	8	8	8	8
`roboter03.txt`	56	56	56	56	56	56
`roboter04.txt`	37	37	37	37	37	37
`roboter05.txt`	4	3	3	3	3	3
`roboter06.txt`	6	6	6	6	6	6
`roboter07.txt`	168	163	162	159	159	157
`roboter08.txt`	536	529	529	526	526	526
`roboter09.txt`	45	45	45	45	45	45
`roboter10.txt`	432	428	431	426	426	426
`roboter11.txt`	1741	1721	1723	1711	1710	1710

Tabelle 2: Übersicht der Lösungen für verschiedene Varianten der TSP Implementierung. Bei $\eta = 0.2$ sind für den Greedy Ansatz die jeweiligen Verbesserungen durch die stärker werdenden Heuristiken beziehungsweise durch das Hinzufügen des exakten Ansatzes für kleine Probleme zu sehen. Die ILP Formulierung löst das TSP Problem exakt und misst die Güte der Zuweisungen von Bäumen zu Routen. Es findet jeweils keine nachträgliche Optimierung statt.

In den folgenden Darstellungen sind Bäume rote Punkte, die Ladestation orange Kreise und die Routen blau.

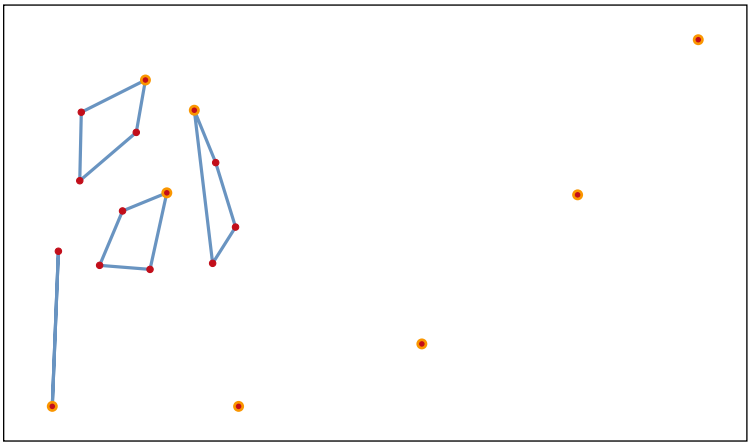
roboter01.txt

```
3
175
139.40949748667006
156 728
9 6 7 8
164.54621220182653
113 675
10 3 4 13 11
157.20876847687683
68 753
1 2 12 5
```



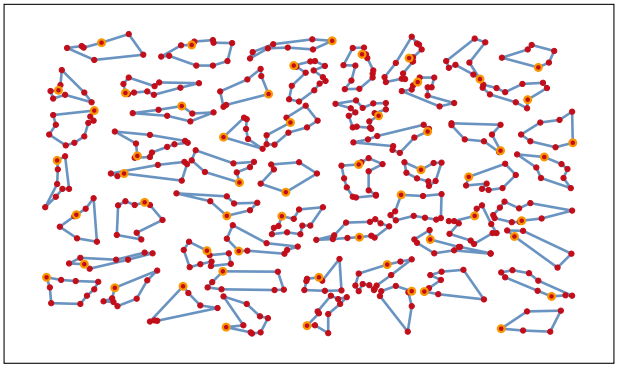
roboter02.txt

```
8
175
0.0
472 789
18
0.0
393 712
17
0.0
291 638
16
154.20765220960988
49 607
14 13
0.0
171 607
15
133.871974723677
124 713
6 3 11 10
164.4417634732132
142 754
12 7 8 9
149.7445950825309
110 769
5 2 4 1
```



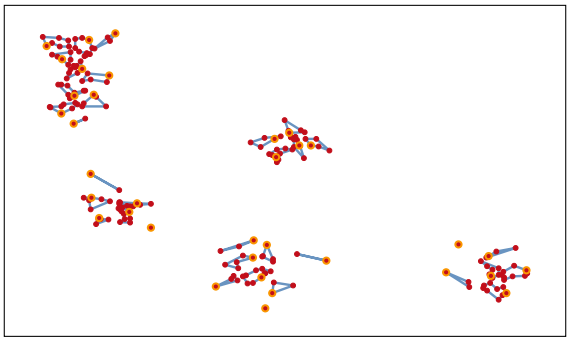
roboter03.txt

```
55
300
...
```



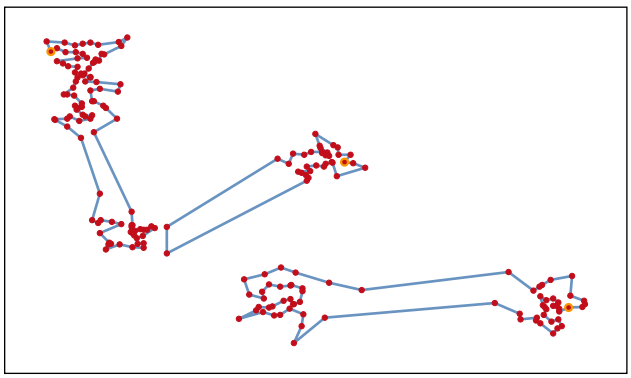
roboter04.txt

```
35
100
...
```



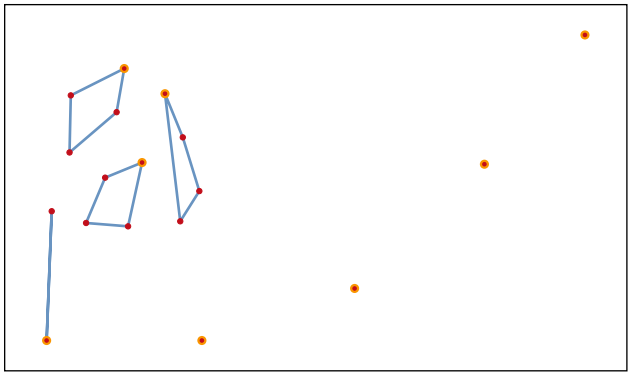
roboter05.txt

```
3
1400
1381.8789109293532
763 244
42 35 37 64 45 48 57 66 52 61 44 47 60 65 54 56 63 43 50 59 36 67
   62 41 134 129 103 111 107 132 117 130 121 127 105 124 114 131
   108 113 122 112 125 102 110 106 126 118 116 128 104 123 133 119
   115 120 109 46 51 55 68 39 58 53 38 40 49
1347.7422871166086
160 698
15 14 34 31 22 26 1 21 32 7 8 3 10 28 18 13 23 2 30 5 179 175 188
   194 4 195 198 200 192 172 186 168 185 197 77 83 86 73 76 87 84
   98 92 94 97 95 74 78 90 81 71 99 82 70 93 96 88 69 72 101 85 91
   80 75 79 183 193 181 180 170 182 191 176 184 190 196 199 178 171
    169 187 177 174 189 173 16 11 29 12 19 27 24 9 33 6 25 17 20
801.1252529807066
502 502
139 137 163 160 148 157 140 145 161 142 155 136 147 146 162 150 138
    152 100 89 159 158 156 167 141 135 144 153 143 164 154 166 165
   151 149
```



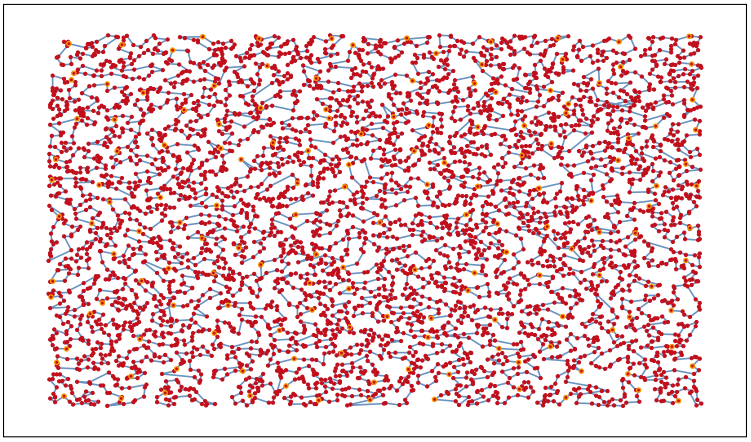
roboter06.txt

```
6
700
487.7912390513498
755 211
56 54 65 60 47 44 61 52 66 57 48 45 64 37 35 42 49 40 38 53 58 39
   68 55 51 46 41 62 67 36 59 50 43 63
685.5210260063409
408 260
128 104 116 118 123 133 119 115 126 106 110 102 120 109 125 112 122
    113 108 131 107 111 134 103 129 132 117 114 124 130 121 127 105
392.10375718837923
502 502
139 137 163 160 148 161 142 155 136 147 145 140 157 146 162 150 152
    138 141 167 156 159 158 135 144 153 143 164 154 166 165 151 149
439.2822871387616
255 351
82 70 93 96 88 69 72 80 91 75 79 77 85 101 86 83 73 98 76 87 84 90
   78 92 94 97 95 74 100 89 99 71 81
606.8818289286261
206 714
26 22 31 34 14 15 20 17 25 6 33 9 24 27 19 12 29 11 16 173 189 174
   177 187 169 171 178 199 182 180 181 170 172 192 200 198 194 188
   175 179 5 30 2 23 13 18 28 10 3 8 7 32 21 1
256.77331803680545
237 579
185 197 183 193 191 196 190 176 184 168 186 195 4
```



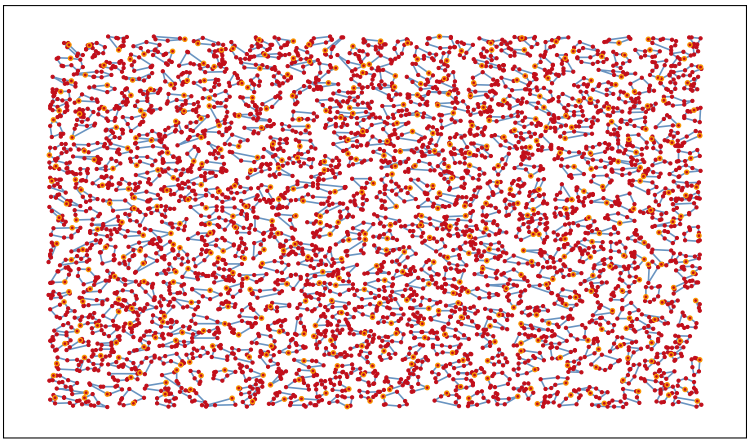
roboter07.txt

```
155
300
...
```



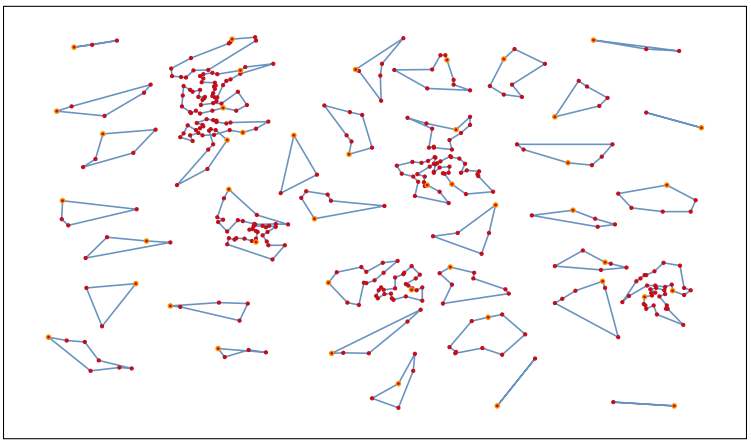
roboter08.txt

```
522
100
...
```



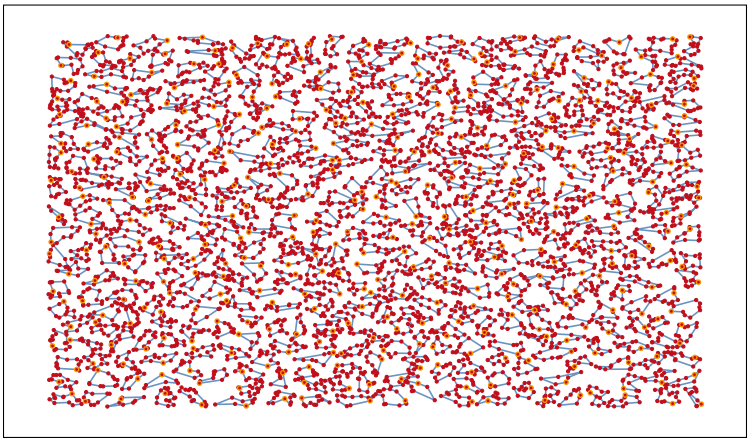
roboter09.txt

```
44
300
...
```



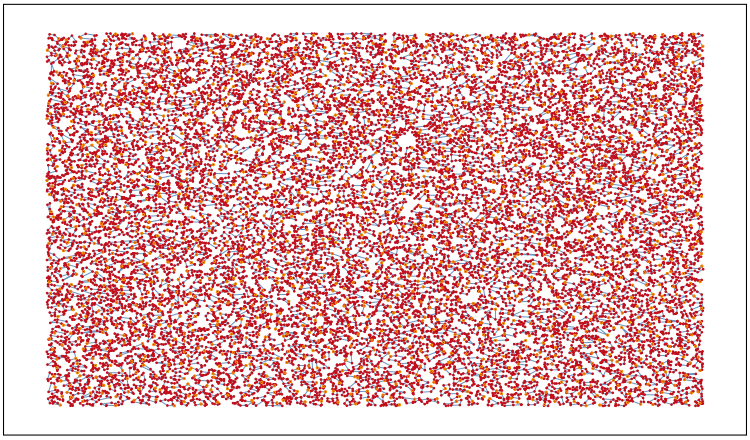
roboter10.txt

```
422
300
...
```



roboter11.txt

```
1699
300
...
```



2.7 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Problem adäquat modelliert*: Die Modellierung muss folgende Anforderungen gut unterstützen:

- Es können beliebige Routen zwischen allen Bäumen erstellt werden.
- Bäume können mehrmals bewässert werden.
- Die Routen können durch die Bäume verlaufen.

(2) *Verfahren nicht unnötig ineffizient*: Das Verfahren sollte eine Route $R = (b_1, \ldots, b_n)$ mit genau dieser Reihenfolge der Bäume nur ein Mal betrachten. Für reine Brute-Force-Ansätze, die zum Beispiel nach und nach Bäume zu Routen hinzufügen und jeweils alle Anzahlen an Robotern durchprobieren, werden 4 Punkte abgezogen.

(3) *Laufzeit des Verfahrens in Ordnung*: Sei N_B die Anzahl der Bäume. Eine erste Routeneinteilung (vor der Anwendung von Verbesserungsheuristiken) muss in $\mathcal{O}(N_B^3)$ gefunden werden. Verfahren mit iterativer Verbesserung, die mit längerer Laufzeit immer bessere Routen produzieren, müssen ein sinnvolles Abbruchkriterium haben. Es müssen alle Beispiele geschafft werden.

(4) *Speicherbedarf in Ordnung*: Die Routen sollten mit linearem Aufwand in der Anzahl der Bäume, also $\mathcal{O}(N_B)$, gespeichert werden können.

(5) *Verfahren mit korrekten Ergebnissen*: Das Verfahren muss Routen so konstruieren, dass die Roboter insgesamt alle Bäume bewässern und die Maximallänge der Routen nicht verletzt wird.

(6) *Verfahren mit guten Ergebnissen*: Es wird immer mit der geringsten Punktzahl bewertet, in deren Spalte (oder in denen darunter) insgesamt mindestens 2 Beispielergebnisse liegen. Bonuspunkte gibt es nur, wenn die Beispiele `roboter07.txt` bis `roboter11.txt` gelöst wurden. Werden die Ergebnisse nur mit Hilfe eines ILP-Solvers erreicht, werden keine Pluspunkte vergeben.

Beispiel (max. Distanz)	$+2$	$+1$	0	-2	-4
`roboter01.txt` (175)			$[3,3]$		$[4,\ldots)$
`roboter02.txt` (175)			$[8,8]$		$[9,\ldots)$
`roboter03.txt` (300)	$[53,55]$		$[56,56]$	$[57,61]$	$[62,\ldots)$
`roboter04.txt` (100)	$[35,36]$		$[37,37]$	$[38,39]$	$[40,\ldots)$
`roboter05.txt` (1400)	$[3,3]$		$[4,4]$		$[5,\ldots)$
`roboter06.txt` (700)			$[6,6]$	$[7,7]$	$[8,\ldots)$
`roboter07.txt` (300)	$[155,158]$	$[159,163]$	$[164,185]$	$[186,190]$	$[191,\ldots)$
`roboter08.txt` (100)	$[498,525]$	$[526,529]$	$[530,560]$	$[561,600]$	$[601,\ldots)$
`roboter09.txt` (300)	$[41,44]$	$[45,45]$	$[46,46]$	$[47,48]$	$[49,\ldots)$
`roboter10.txt` (300)	$[402,425]$	$[426,428]$	$[429,450]$	$[451,490]$	$[491,\ldots)$
`roboter11.txt` (300)	$[1625,1709]$	$[1710,1723]$	$[1724,1800]$	$[1801,1940]$	$[1941,\ldots)$

2. Theoretische Analyse

(1) *Verfahren / Qualität insgesamt gut begründet*: Es muss begründet sein, welche Ansätze zu guten Routenbelegungen führen sollen. Die Qualität von verwendeten Heuristiken

muss diskutiert werden. Die Platzierung der Ladestationen muss diskutiert werden.

(2) *Gute Überlegungen zur Laufzeit des Verfahrens*: Für alle statischen, deterministischen Algorithmen müssen (nicht zwingend formale) Angaben zur Laufzeit vorliegen. Entspricht der Algorithmus einem Standardverfahren (z.B. Breitensuche, Dijkstra, A*), so kann seine Laufzeit ohne Herleitung angegeben werden. Andernfalls müssen Begründungen zur Laufzeit des Algorithmus vorhanden sein. Für alle Algorithmen, die auf iterativer Verbesserung basieren (wie etwa Clustering), müssen Konvergenzkriterien oder andere Abbruchbedingungen aufgeführt werden.

(3) *Vergleichende Analysen*: Wenn die Ergebnisse mehrerer Verfahren kommentarlos angegeben werden, können 2 Punkte abgezogen werden. Für eine Einsendung, die mehrere Verfahren vergleicht und Begründungen für deren Qualität liefert, können Pluspunkte vergeben werden.

(4) *NP-Schwere gezeigt*: Es ist relativ leicht zu sehen, dass die Aufgabe NP-Schwer ist. Falls die NP-Schwere der Aufgabe erkannt und besonders schön und korrekt begründet wird, etwa durch eine formelle Reduktion von einem bekannten NP-schweren Problem, kann bis zu 1 Pluspunkt vergeben werden.

3. Dokumentation

(3) *Vorgegebene Beispiele dokumentiert*: Es müssen alle Beispieleingaben (`01` bis `11`) dokumentiert sein, ansonsten können Punkte abgezogen werden.
Für alle Beispiele sollen die 1. und 2. Zeile der Ausgabe abgedruckt sein. Außerdem müssen für die Beispiele `01`, `02`, `05` und `06` die kompletten Ausgaben in der Dokumentation enthalten sein. Für die restlichen Beispieldateien müssen diese der Einsendung beiliegen.

(5) *Ergebnisse nachvollziehbar dargestellt*: Es müssen jeweils die benötigte Anzahl Roboter, die Längen ihrer Routen und die Positionen der Ladestationen dokumentiert sein.

Aufgabe 3: Lieferkette

3.1 Lösungsidee

Beschreibung der Aufgabe

Für eine bessere Erläuterung der Lösungsidee werden zunächst einige Begriffe eingeführt.

Produktion Für eine *Produktion* müssen an einem Objekt m verschiedene *Arbeitsschritte* in der richtigen Reihenfolge durchgeführt werden. Dafür gibt es unterschiedliche *Werke*, die je einen bestimmten Arbeitsschritt durchführen können. Diese liegen auf einer Straßenkarte, was mit einem ungerichteten, gewichteten Graphen modelliert werden kann. Die Knoten stellen die Werke dar und die Kanten die Straßen, wobei die Gewichte den Fahrtdauern entsprechen. Zusätzlich gibt es zwei besondere Knoten S und T, die keine Werke sind. Die Produktion soll in S beginnen und in T enden. Ein Werk wird *produzierend* genannt, wenn ein Arbeitsschritt dort durchgeführt wird. Während der Produktion kann es zu bis zu einem *Streik* in einem Werk kommen. Der Versuch, einen Arbeitsschritt durchzuführen, schlägt fehl und man muss ein anderes Werk nehmen. Das erfährt man jedoch erst, wenn man am Werk ankommt, um den Arbeitsschritt tatsächlich durchzuführen.

Pläne Betrachtet werden *Pläne* für die Produktion. Ein solcher besteht aus einer normalen Streckenplanung und einer alternativen Streckenplanung für jedes produzierende Werk. Eine *normale Streckenplanung* ist ein Weg durch den Graphen, der bei S startet und bei T endet. Außerdem muss für jeden Arbeitsschritt genau ein Werk auf dem Weg ausgewählt werden, um diesen Schritt durchzuführen. Die ausgewählten Werke müssen den jeweiligen Arbeitsschritt durchführen können. Zusätzlich muss die Auswahl der Werke so getroffen werden, dass die Arbeitsschritte in der richtigen Reihenfolge durchgeführt werden. Eine *alternative Streckenplanung* ist für den Fall eines Streiks vorgesehen. Sie fängt, statt bei S, beim bestreikten Werk an. Es werden nur noch Werke für die Arbeitsschritte ab dem fehlgeschlagenen ausgewählt. Zusätzlich muss für den fehlgeschlagenen Schritt ein *alternatives* Werk ausgewählt werden, das nicht das bestreikte Werk ist.

Dauer Die *Dauer einer Streckenplanung* ist die Länge des Weges, also die Summe aller Kantengewichte, von S bis T. Bei alternativen Streckenplanungen wird bis zum Streik die normale Streckenplanung vorangesetzt. Die *Dauer eines Plans bzw. einer Produktion (nach einem Plan)* ist die maximale Dauer einer der Streckenplanungen (ob normal oder alternativ). Die *minimale Dauer der Produktion* ist die kleinste Dauer, für die es einen gültigen Plan gibt. Manchmal wird bei den letzten beiden auch von *im schlimmsten Fall* gesprochen. Damit wird zusätzlich betont, dass in der Dauer bis zu ein Streik berücksichtigt wird.

Die Aufgabe Es wird eine Zeitfrist für die Gesamtproduktion vorgegeben. Die Aufgabe fragt, ob es einen Plan mit einer Dauer im schlimmsten Fall gibt, die kleiner gleich diese Zeit ist. Wenn ja, soll ein solcher Plan berechnet werden. Dies ist auch äquivalent dazu, ob die minimale Dauer der Produktion kleiner gleich die Zeitfrist ist. Im Folgenden werden wir ein Verfahren betrachten, das die minimale Dauer der Produktion bestimmt und einen Plan dieser Dauer liefert.

S und T als Werke Als Vereinfachungen werden S und T nun als Werke betrachtet. In S wird ein neuer 0-ter Arbeitsschritt durchgführt, in T ein neuer $(m+1)$-ter Arbeitsschritt. Wir nehmen für beide an, dass sie nicht streiken können. Es gibt also keine alternativen Streckenplanungen. Dies steht im Einklang mit der Aufgabe. Denn wir starten bei S, wo jetzt zusätzlich der erste 0-te Schritt ausgeführt werden muss. Und wir enden immer in T, wo der letzte $(m+1)$-te Schritt ausgeführt wird.

Zur Lösung des Problems werden zuerst die kürzesten Wege zwischen allen Werken berechnet. Anschließend wird mit Dynamischer Programmierung die minimale Dauer der Produktion berechnet. Erst ohne Streiks zu berücksichtigen, dann mithilfe der Ergebnisse auch unter Berücksichtigung von einem Streik. Zum Schluss wird ein Plan rekonstruiert. Da das kürzeste Wege Problem ein bekanntes Problem ist, wird es erst als gegeben angenommen und nur am Schluss kurz beschrieben.

Dynamischer Programmierung

Das Problem kann mit Dynamischer Programmierung (DP) gelöst werden. DP ist ein Prinzip, nach dem Algorithmen entworfen werden. Dabei wird das Problem rekursiv auf kleinere Probleme zurückgeführt. Die Lösungen der kleineren Problemen werden für eine effiziente Berechnung zwischengespeichert.

Statt der ganzen Produktion von S bis T betrachten wir nur noch die restliche Produktion ab einem Arbeitsschritt in Werk u. Es wird also in u gestartet und versucht, dort den zugehörigen Arbeitsschritt durchzuführen. Alle vorherigen Arbeitsschritte werden vernachlässigt, bzw. als bereits durchgeführt angenommen. Das Problem wird somit auf den restlichen Teil der Produktion beschränkt (und wir legen einen neuen Startpunkt und ersten Arbeitsschritt fest). Die ursprüngliche Frage bleibt aber gleich.

Auf zwei Probleme wenden wir diese Form der Verkleinerung an. Zum einen das ursprüngliche Problem ohne Streik: Was ist die minimale Dauer der (restlichen) Produktion ohne Streiks. Zum anderen das ursprüngliche Problem mit Streiks: Was ist die minimale Dauer der (restlichen) Produktion im schlimmsten Fall. Hierfür definieren wir zuerst Folgendes:

1. S_i: Menge aller Werke, in denen Arbeitsschritt i durchführbar ist.
2. $\text{dist}[u][v]$: minimale Fahrtdauer zwischen den Werken u und v, also Länge des kürzesten Weges zwischen u und v.
3. $\text{DP}_{\text{ohne\_streik}}[u]$: minimale Dauer der restlichen Produktion, startend mit einem Arbeitsschritt in u. Es wird davon ausgegangen, dass es keine Streiks gibt in diesem Teil der Produktion.
4. $\text{DP}_{\text{mit\_streik}}[u]$: minimale Dauer der restlichen Produktion im schlimmsten Fall, startend mit einem Arbeitsschritt in u. Es kann bis zu ein Streik in diesem Teil der Produktion auftreten.

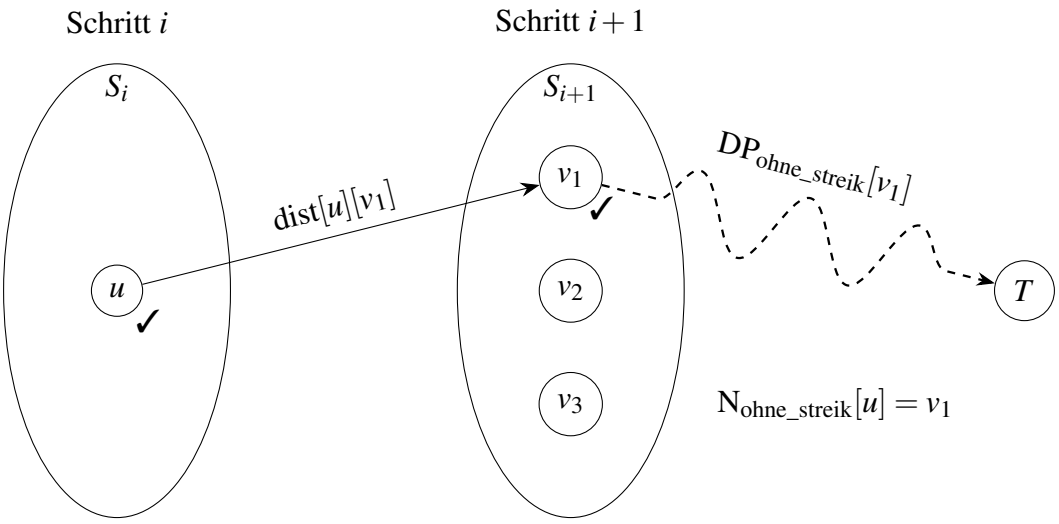
Die Punkte 3. und 4. beschreiben die zwei Arten von kleineren Problemen. Insbesondere ist mit der Betrachtung von S als Werk $\text{DP}_{\text{mit\_streik}}[S]$ nach Definition das gesuchte Ergebnis, nämlich die minimale Dauer der Produktion im schlimmsten Fall. $\text{DP}_{\text{ohne\_streik}}$ wird später für die Berechnung von $\text{DP}_{\text{mit\_streik}}$ verwendet. Daher schauen wir uns zuerst den Fall ohne Streiks an.

Lösung von $\mathrm{DP}_{\text{ohne\_streik}}$

Wir vernachlässigen erst Streiks. Sei u ein beliebiges Werk und i der zugehörige Arbeitsschritt, also $u \in S_i$. Nachfolgend wird gezeigt, wie $\mathrm{DP}_{\text{ohne\_streik}}[u]$ berechnet werden kann, wenn $\mathrm{DP}_{\text{ohne\_streik}}[v]$ für alle $v \in S_{i+1}$ des nächsten Arbeitsschritts bekannt sind. Man kann damit $\mathrm{DP}_{\text{ohne\_streik}}$ von hinten berechnen. Zuerst für Werke des letzten Arbeitsschrittes, des vorletzten, und so weiter. Für den letzten $(m+1)$-ten Schritt ist $\mathrm{DP}_{\text{ohne\_streik}}[T] = 0$ bekannt, denn im $(m+1)$-ten Schritt in T ist man schon fertig.[20]

Wir suchen $\mathrm{DP}_{\text{ohne\_streik}}[u]$ und betrachten hier nur die Produktion ab einem Arbeitsschritt in u. Nach einem Arbeitsschritt in u sind alle Werke in S_{i+1} mögliche Werke für den nächsten Arbeitsschritt. Betrachte den Fall, dass $v \in S_{i+1}$ den nächsten Arbeitsschritt ausführt. In diesem Fall besteht die Dauer der Produktion aus der Fahrtdauer von u zu v und der Dauer der restlichen Produktion ab v. Die beiden Werte sind unabhängig voneinander. Deswegen können wir die Produktionsdauer minimieren, indem wir beide Werte einzeln minimieren. Die minimale Weglänge ist $\mathrm{dist}[u][v]$. Die minimale Dauer der restlichen Produktion ab v ist $\mathrm{DP}_{\text{ohne\_streik}}[v]$ und hängt nicht vom vorherigen Teil der Produktion ab. Hier sehen wir die rekursive Zurückführung auf ein kleineres Problem. Zusammengefasst ist in diesem Fall die minimale Dauer $\mathrm{dist}[u][v] + \mathrm{DP}_{\text{ohne\_streik}}[v]$. Den gesuchten Wert von $\mathrm{DP}_{\text{ohne\_streik}}[u]$ erhält man nun als das Minimum unter allen $v \in S_{i+1}$:

$$\mathrm{DP}_{\text{ohne\_streik}}[u] = \min_{v \in S_{i+1}} \left\{ \mathrm{dist}[u][v] + \mathrm{DP}_{\text{ohne\_streik}}[v] \right\}$$



Berechnung von $\mathrm{DP}_{\text{ohne\_streik}}[u]$. Schritt i wird in u erfolgreich ausgeführt. Hier wird für den nächste Schritt v_1 angefahren. $\mathrm{DP}_{\text{ohne\_streik}}[v_1]$ gibt an, wie lange die restlichen Schritte bis T ab dort dauern. Wenn v_1 als nächstes angefahrene Werk die Dauer minimiert, haben wir $\mathrm{N}_{\text{ohne\_streik}}[u] = v_1$.

Lösung von $\mathrm{DP}_{\text{mit\_streik}}$

Nun nehmen wir bis zu einen Streik hinzu. Auch hier berechnen wir die Werte rückwärts. Dies funktioniert ähnlich wie bei $\mathrm{DP}_{\text{ohne\_streik}}$ und wird aus den Werten des nächsten Arbeitsschritts

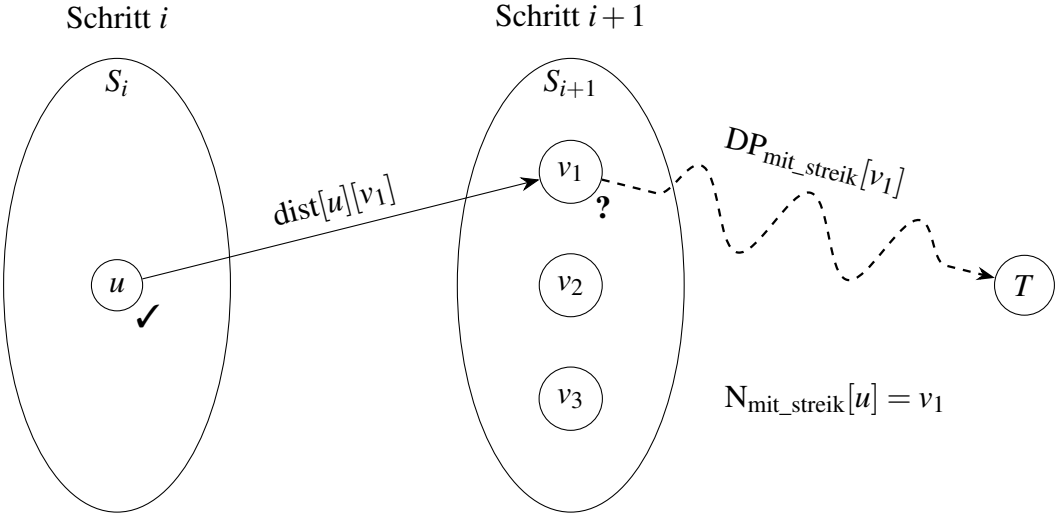
[20] Dieser iterative Ansatz wird auch als Bottom-Up bezeichnet. Es geht auch rekursiv, was Top-Down heißt.

berechnet, verwenden jedoch auch die vorher berechneten Werte von $\mathrm{DP}_{\text{ohne\_streik}}$. Analog haben wir auch hier $\mathrm{DP}_{\text{mit\_streik}}[T] = 0$. Die minimale Dauer der ganzen Produktion im schlimmsten Fall wird am Ende in $\mathrm{DP}_{\text{mit\_streik}}[S]$ berechnet. Bei der Berechnung in S und dem 0-ten Schritt gibt es den Spezialfall, dass kein Streik möglich ist. Daher entfällt der gleich vorgestellte zweite Fall.

Sei u wieder ein beliebiges Werk und i der zugehörige Arbeitsschritt, also $u \in S_i$. Für $\mathrm{DP}_{\text{mit\_streik}}[u]$ geht es auch nur um die Fertigung ab u. Hier gibt es jedoch zwei Fälle: Es gibt *keinen* Streik beim Arbeitsschritt in u oder es gibt einen. Für beide müssen wir getrennt eine Streckenplanung finden, die jeweils möglichst schnell ist. Da wir die Dauer im schlimmsten Fall betrachten, ist die Lösung von $\mathrm{DP}_{\text{mit\_streik}}[u]$ schließlich das Maximum der beiden Dauern.

Betrachte zuerst den Fall, in dem es keinen Streik beim Arbeitsschritt in u gibt. Die Produktion gelingt und wir suchen das Werk für den nächsten Schritt. Hier gehen wir ähnlich wie bei $\mathrm{DP}_{\text{ohne\_streik}}$ vor. Der Unterschied ist nur, dass gestreikt werden könnte ab dem nächsten Schritt. Für jedes Werk $v \in S_{i+1}$ wäre die minimale Dauer der Produktion somit $\mathrm{dist}[u][v] + \mathrm{DP}_{\text{mit\_streik}}[v]$. Die minimale Produktionsdauer im ersten Fall ist dann:

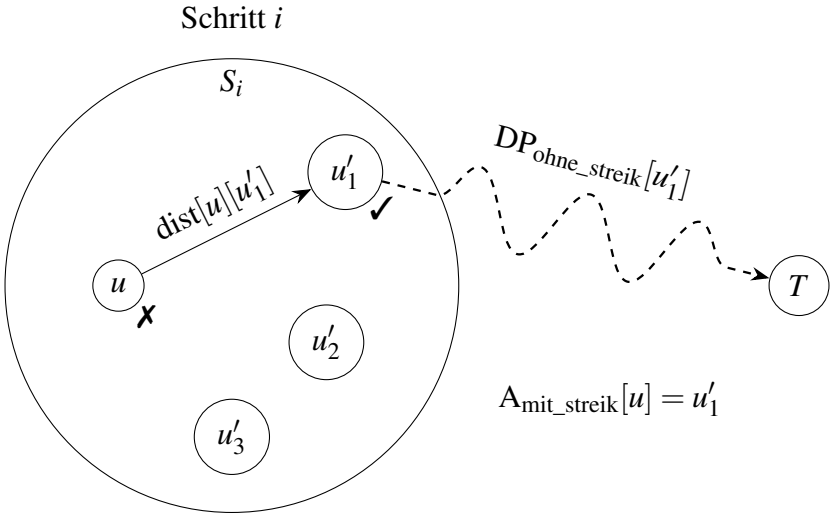
$$x_1 = \min_{v \in S_{i+1}} \Big\{ \mathrm{dist}[u][v] + \mathrm{DP}_{\text{mit\_streik}}[v] \Big\}$$



Berechnung von $\mathrm{DP}_{\text{mit\_streik}}[u]$ im ersten Fall, dass die Produktion in u gelingt. Als nächstes wird hier $\mathrm{N}_{\text{mit\_streik}}[u] = v_1$ angefahren. Ab dort kann es zu Streiks kommen und es ist ungewiss, ob die nächsten Schritte gelingen. Daher beschreibt $\mathrm{DP}_{\text{mit\_streik}}[v_1]$ die restliche Produktion.

Betrachte als nächstes den zweiten Fall, dass es einen Streik beim Arbeitsschritt in u gibt. Der Arbeitsschritt schlägt fehl und es wird ein alternatives Werk gesucht. Dabei können wir ausnutzen, dass es keinen Streik mehr geben kann. Denn wir haben angenommen, dass höchstens ein Werk streikt. Sei $u' \in S_i$ ein alternatives Werk für den Arbeitsschritt i. Die Dauer der Produktion unterteilen wir wieder in Fahrtdauer und Dauer der restlichen Produktion ab u'. Für letztere suchen wir die minimale Dauer der Produktion ab einem Schritt in u', wobei es keine Streiks mehr gibt. Dies haben wir mit $\mathrm{DP}_{\text{ohne\_streik}}[u']$ schon berechnet. Daher haben wir für die Alternative u' die minimale Produktionsdauer $\mathrm{dist}[u][u'] + \mathrm{DP}_{\text{ohne\_streik}}[u']$. Über alle möglichen Alternativen ergibt sich für den zweiten Fall:

$$x_2 = \min_{u' \in S_i} \Big\{ \mathrm{dist}[u][u'] + \mathrm{DP}_{\text{ohne\_streik}}[u'] \Big\}$$



Berechnung von $\text{DP}_{\text{mit\_streik}}[u]$ im zweiten Fall, dass die Produktion in u fehlschlägt. Dann wird die Alternative $\text{A}_{\text{mit\_streik}}[u] = u'_1$ angefahren. Ab dort gibt es keine Streiks mehr, daher $\text{DP}_{\text{ohne\_streik}}[u'_1]$.

Wie bereits gesagt müssen wir jetzt nur noch das Maximum aus x_1 und x_2 nehmen:

$$\text{DP}_{\text{mit\_streik}}[u] = \max\left\{ \min_{v \in S_{i+1}} \left\{ \text{dist}[u][v] + \text{DP}_{\text{mit\_streik}}[v] \right\}, \min_{u' \in S_i} \left\{ \text{dist}[u][u'] + \text{DP}_{\text{ohne\_streik}}[u'] \right\} \right\}$$

Rekonstruktion der Planung

Wir können nun die minimale Dauer berechnen. Gesucht ist jedoch auch ein expliziter Plan. Den können wir nach der Berechnung ebenfalls konstruieren, wenn wir einige zusätzliche Informationen speichern. Fürs Erste reicht es aus, zu wissen, wo Arbeitsschritte ausgeführt werden sollen. Für die Fahrt zwischen den produzierenden Werken reicht es, einen beliebigen kürzesten Weg einzufügen. Wie man einen solche berechnen kann, wird später erläutert.

Die wesentliche Idee ist die folgende: Bei jeder minimalen Produktionsdauer können wir speichern, wo der nächste Arbeitsschritt stattfinden muss, damit diese minimale Produktionsdauer erreicht werden kann. Denn bei unseren DP-Verfahren probieren wir alle möglichen Werke für den nächsten Arbeitsschritt aus. Für jedes berechnen wir die minimale Produktionsdauer, *wenn es als nächstes den Arbeitsschritt ausführt*. Die kleinste minimale Produktionsdauer wird genommen. Wir können also einfach speichern, zu welchem nächsten Werk diese kleinste Dauer gehört.

Konkret müssen wir für jedes Werk in drei Fällen das nächste produzierende Werk speichern.

- Zum Wert von $\text{DP}_{\text{ohne\_streik}}[u]$ speichern wir, mit welchem nächsten (produzierenden) Werk $\text{N}_{\text{ohne\_streik}}[u]$ diese möglich ist.
- Bei $\text{DP}_{\text{mit\_streik}}[u]$ müssen wir unterscheiden, ob es am aktuellen Werk zu einem Streik kommt. Die Produktionsdauer für diese beiden Fälle haben wir in x_1 und x_2 getrennt berechnet. Daher speichern wir für diese beiden Fälle getrennt das nächste produzierende Werk.
  - Wenn kein Streik passiert, führt $\text{N}_{\text{mit\_streik}}[u]$ den nächsten Arbeitsschritt durch.
  - Sonst ist $\text{A}_{\text{mit\_streik}}[u]$ die Alternative für den gleichen Arbeitsschritt.

Die Streckenplanungen werden Stück für Stück aufgebaut. Betrachten wir zunächst die normale Streckenplanung, wenn kein Streik auftritt. Das Werk für den 0-ten Schritt ist $u_0 = S$. Beachte, dass wir nicht bei $\mathrm{DP}_{\mathrm{ohne\_streik}}[S]$ anfangen, auch wenn es hier um eine Route ohne Streiks geht. Denn wir wollen bei Streiks ausweichen können und das haben wir nur in $\mathrm{DP}_{\mathrm{mit\_streik}}[S]$ berücksichtigt. Das Werk für den ersten Arbeitsschritt ist $u_1 = \mathrm{N}_{\mathrm{mit\_streik}}[S]$. Für den zweiten Arbeitsschritt ist es $u_2 = \mathrm{N}_{\mathrm{mit\_streik}}[u_1]$. Das wiederholen wir analog weiter, bis wir für alle Arbeitsschritte ein Werk gefunden haben. Für die alternativen Streckenplanungen ab dem i-ten Arbeitsschritt wird als alternatives Werk $\mathrm{A}_{\mathrm{mit\_streik}}[u_i]$ genommen. Danach baut man die restliche Streckenplanung analog mit $\mathrm{N}_{\mathrm{ohne\_streik}}$ auf, denn es kann keine Streiks mehr geben.

Zum Schluss werden zwischen den produzierenden Werken kürzeste Wege eingefügt.

Berechnung der kürzesten Wege

Die Kantengewichte sollten alle nicht-negativ sein. Deswegen kann für die Berechnung der kürzesten Wege zwischen den Werken der Algorithmus von Dijkstra benutzt werden. Dies müssen wir für alle Werke als Startknoten machen. Für die Berechnung der Streckenplanungen muss der Weg rekonstruiert werden.

Dijkstras Algorithmus berechnet ausgehend von einem Startknoten den kürzesten Weg zu allen Knoten. Dabei wird eine Menge von Knoten verwaltet, deren Distanz bereits bekannt ist. Anfangs ist das nur der Startknoten mit Distanz 0. Schritt für Schritt wird diese Menge um jeweils einen Knoten erweitert. Für jeden Knoten außerhalb der Menge wird eine (vorläufige) minimale Distanz berechnet, wenn man zu diesem eine Kante von einem der Knoten mit bekannten Distanzen nimmt. Unter diesen Knoten hat die mit der kürzesten vorläufigen Distanz bereits die kürzeste Distanz. Würde der Weg über einen anderen Knoten noch ohne bekannte Distanz führen, dann wäre der Weg länger. Es gibt nämlich keine negativen Gewichte.

Die Rekonstruktion eines Weges ist möglich, wenn man für jeden Knoten den Vorgänger im kürzesten Weg speichert. Vom Zielknoten aus baut man dann den Weg „rückwärts“ auf, bis man beim Startknoten angekommen ist.

Laufzeit

Sei V die Anzahl von Werken und E die Anzahl der Straßen. Für die Ausführungen von Dijkstra haben wir eine Laufzeit von $\mathcal{O}(V(V+E)\log V)$ mit einer Vorrangwarteschlange. Bei unseren DP Berechnungen wird für jedes Werk, grob abgeschätzt, durch jedes andere Werk iteriert. Dabei wird nur auf vorberechnete Werte zugegriffen, weshalb mit Arrays hier eine Laufzeit von $\mathcal{O}\left(V^2\right)$ vorliegt. Insgesamt dominiert hier die Berechnung der kürzesten Wege. Die Rekonstruktion ist deutlich langsamer und benötigt $\mathcal{O}(mV^2)$. Jede Strecke dauert $\mathcal{O}(V^2)$, da man möglicherweise für jeden Arbeitsschritt hin und her fährt. Und es werden m Strecken ausgegeben.

Optimierung der Laufzeit mit Dijkstra

Das Vorberechnen aller Distanzen mit Dijkstra dauert relativ lang. Dies kann man optimieren, sodass man nur asymptotisch m Mal Dijkstra benötigt. Ein möglicher Ansatz sieht wie folgt aus. Die Distanzen werden nicht mehr vorberechnet. Stattdessen werden die DP-Werte mit Dijkstra

berechnet. Für jeden Arbeitsschritt reicht es dabei aus, drei mal Dijkstra für alle Werke dieses Arbeitsschrittes durchzuführen. Dafür werden nur die DP-Werte des vorherigen Arbeitsschrittes benötigt.

Zunächst betrachten wir $\mathrm{DP}_{\text{ohne\_streik}}$. Es werden mehrere Startknoten gesetzt. Jedes Werk v des vorherigen Schritts bekommt als Distanz $\mathrm{DP}_{\text{ohne\_streik}}[v]$ und wird in die Warteschlange eingefügt. Führt man Dijkstra aus, dann sind die Distanzen die kürzesten Distanzen zu einem dieser Startknoten $v + \mathrm{DP}_{\text{ohne\_streik}}[v]$. Das ist genau unser gesuchter Wert.

Bei $\mathrm{DP}_{\text{mit\_streik}}$ kann dies in ähnlicher Weise benutzt werden. Die Berechnung setzt sich hier wieder aus zwei Teilen, oben als x_1 und x_2 benannt, zusammen. Die x_1 Werte (es wird in dem Schritt nicht gestreikt) lassen sich ganz analog zu $\mathrm{DP}_{\text{ohne\_streik}}$ berechnen. Die Startknoten sind wieder die Werke vom vorherigen Schritt, allerdings mit Distanz $\mathrm{DP}_{\text{mit\_streik}}[v]$.

Die x_2 Werte (es wird in dem Schritt gestreikt) erfordern eine Anpassung. Startknoten sind die Werke des gleichen Schritts, mit Distanz $\mathrm{DP}_{\text{ohne\_streik}}[u]$. Wenn man den normalen Dijkstra verwendet, könnte am Ende die kürzeste Distanz vom gleichen Knoten ausgehen. Wir suchen aber ein alternatives Werk, das also nicht das gleiche ist. Daher benötigen wir auch die kleinste Distanz von einem anderen Startknoten. Folgende Anpassung ist nötig: Für jeden Knoten verwalten wir eine zweitkleinste Distanz, die von einem anderen Startknoten ausgehen muss. Außerdem wird zu beiden Distanzen der zugehörige Startknoten gespeichert. Jeder Knoten wird für beide Distanzen in die Warteschlange eingefügt. So wird eine korrekte Berechnung nach gleichem Prinzip wie beim regulären Dijkstra sichergestellt. Der gesuchte x_2 Wert ist die kleinste Distanz, falls der Startknoten ein anderer ist. Sonst die zweitkleinste.

3.2 Beispiele

Nachfolgend sind vollständige Ausgaben für `lieferung00.txt` bis `lieferung03.txt` und `lieferung09.txt` zu finden. Für die restlichen sind es nur die ersten zwei Zeilen. Das Ausgabeformate ist das vorgegebene. Bei allen Beispielen haben die Pläne minimale Dauer der Produktion. Im Falle von „unmöglich“ wird zusätzlich in der zweiten Zeile auch die minimale Dauer der Produktion angegeben, was nicht erwartet war.

lieferung00.txt

```
MOEGLICH
15

13
S [1 1] 2 [4 2] T
10
1 [2 1] 3 [5 2] T
15
4 T [5 2] T
```

lieferung01.txt

```
MOEGLICH
40
```

```
32
S [1 1] [4 2] [8 3] T
40
1 S [2 1] S 1 [4 2] [7 3] T
37
4 [5 2] [8 3] T
36
8 [6 3] T
```

lieferung02.txt

```
MOEGLICH
188

127
S [1 1] S [2 2] [4 3] 3 5 [6 4] 5 [7 5] 13 2 S 1 T
138
1 6 [5 1] [3 2] [4 3] 2 13 7 [10 4] [7 5] 13 2 S 1 T
157
2 13 7 5 [3 2] [4 3] 2 13 7 [10 4] [7 5] 13 2 S 1 T
188
4 3 5 6 12 [11 3] 9 8 [10 4] [7 5] 13 2 S 1 T
143
6 [14 4] 6 5 [7 5] 13 2 S 1 T
127
7 [13 5] 2 S 1 T
```

lieferung03.txt

```
UNMOEGLICH
299
```

lieferung04.txt

```
MOEGLICH
264
...
```

lieferung05.txt

```
MOEGLICH
930
...
```

lieferung06.txt

```
UNMOEGLICH
1541
```

lieferung07.txt

```
MOEGLICH
675
...
```

lieferung08.txt

```
MOEGLICH
1333
...
```

lieferung09.txt

```
MOEGLICH
11

9
S [3 1] [5 2] T
10
3 [2 1] [4 2] T
11
5 T [4 2] T
```

lieferung10.txt

```
MOEGLICH
1462
...
```

3.3 Bewertungskriterien

Die aufgabenspezifischen Bewertungskriterien werden hier erläutert.

1. Lösungsweg

(1) *Problem adäquat modelliert*: Die Modellierung muss folgende Anforderungen gut unterstützen:

- Straßen können in beide Richtungen befahren werden.
- S und T, also Start- und Endpunkt der Streckenplanung, sind erkennbar.
- Werke mit und ohne Arbeitsschritte sind unterscheidbar.
- Die zeitliche Abfolge der Arbeitsschritte ist dargestellt.

(2) *Verfahren nicht unnötig ineffizient*: Es darf nur eine normale Streckenplanung bestimmt werden. Es dürfen nicht ab jedem ausfallenden Werk alle Wege neu berechnet werden.

(3) *Laufzeit des Verfahrens in Ordnung*: Die Laufzeit sollte nicht schlechter als bei Anwendung des Dijkstra-Algorithmus sein. Bei der Anwendung des Algorithmus kann man sich auf die Anzahl der Produktionsschritte m beschränken; dann ergibt sich eine Gesamtlaufzeit von $\mathcal{O}(m(V+E)\log V)$. Ohne die Beschränkung auf m werden 2 Punkte abgezogen. Wird der Floyd-Warshall-Algorithmus verwendet, wird zusätzlich 1 Punkt abgezogen. Die Rekonstruktion der Wegen darf maximal eine Laufzeit von $\mathcal{O}(m^2V)$ haben.

(4) *Speicherbedarf in Ordnung*: Der Speicherbedarf sollte, abgesehen von dem quadratischen Speicherbedarf für Distanzvorberechnung, maximal in $\mathcal{O}(V)$ liegen. Das ist insbesondere mit DP zu erreichen. Für die Rekonstruktion der besten Routen darf zusätzlicher Speicher verwendet werden.

(5) *Verfahren entscheidet korrekt*: Das Verfahren muss für eine Eingabe bestimmen können, ob die Zeitfrist eingehalten werden kann (dann ist die Eingabe *lösbar*) oder nicht. Die Lösbarkeit muss für alle Beispiele richtig bestimmt werden. Wird ein Beispiel falsch gelöst, werden mindestens 2 Punkte abgezogen. Bei mehreren falsch gelösten Beispielen bis zu 4 Punkte. Wenn nachgewiesen werden kann, dass ein Verfahren nicht garantiert korrekte Ergebnisse liefert, werden 4 Punkte abgezogen.

Beispiel	Lösbar	Max. Routendauer	Beispiel	Lösbar	Max. Routendauer
`lieferung00.txt`	ja	15	`lieferung06.txt`	nein	1541
`lieferung01.txt`	ja	40	`lieferung07.txt`	ja	675
`lieferung02.txt`	ja	188	`lieferung08.txt`	ja	1333
`lieferung03.txt`	nein	299	`lieferung09.txt`	ja	11
`lieferung04.txt`	ja	264	`lieferung10.txt`	ja	1462
`lieferung05.txt`	ja	930			

(6) *Verfahren erzeugt gültige Streckenplanungen*: Kann die Zeitfrist eingehalten werden, muss eine gültige Streckenplanung erzeugt werden. Diese enthält, neben den produzierenden Werken, auch alle Werke, an denen der LKW vorbeifährt.

2. Theoretische Analyse

(1) *Verfahren / Qualität insgesamt gut begründet*: Es ist nicht schwierig, ein funktionierendes Verfahren für dieses Problem zu finden. Umso besser muss das Verfahren daher begründet sein. Die Korrektheit des Verfahrens muss begründet werden; insbesondere DP darf nicht ohne Begründung verwendet werden.

3. Dokumentation

(3) *Vorgegebene Beispiele dokumentiert*: Es müssen alle Beispieleingaben (`00` bis `10`) dokumentiert sein, ansonsten können Punkte abgezogen werden.
Für alle Beispiele sollen die 1. und 2. Zeile der Ausgabe abgedruckt sein. Außerdem müssen für Beispiele `00` bis `03` die vollständigen Ausgaben in der Dokumentation enthalten sein. Für die restlichen Beispiele müssen diese der Einsendung beiliegen.

(5) *Ergebnisse nachvollziehbar dargestellt*: Es muss erkennbar sein, ob eine Eingabe als lösbar oder unlösbar klassifiziert wird. Außerdem müssen die alternativen Routen je streikendem Werk in der Ausgabe enthalten sein. Auf den Routen müssen alle Werke auf dem Weg aufgeführt sein, auch wenn dort kein Arbeitsschritt stattfindet. Die Werke, in denen ein Arbeitsschritt ausgeführt wird, müssen erkennbar sein. Die Dauer der Routen muss angegeben sein.

Aus den Einsendungen: Perlen der Informatik

Hier findet ihr die Perlen der Informatik aus den diesjährigen Einsendungen zur 2. Runde des 44. Bundeswettbewerbs Informatik. Farbige Schrift kennzeichnet einen Kommentar.

Aufgabe 1: Transformers

Dies war wahrscheinlich der Hauptmoment, wo ich mich machtlos gegenüber dieser Struktur gefühlt habe, aber wenn die Struktur erfüllbar ist, dann liegt es nun nur noch daran eine Heuristik zu finden, die es noch schneller macht. Wenn sie nicht erfüllbar ist, dann führt das Beispiel 2:0.

Ich hatte nicht genug Zeit, um alle Beispiele laufen zu lassen, aber sie sollten alle in angemessener Zeit lösbar sein.

Soweit man findet, dass diese umxgedreht und umygedreht den erwarteten Werten entspricht [...].

Mit dem passenden Prompt ließ sich dann eine ausführliche Übersicht über die Problematik generieren - sowohl in Form einer Infografik, als auch als Podcast.

Sind die Grafen gleich, verändern wir diese in den nächsten Übergangsgrafen.

```
if Speicherauslastung > 80% then
 Abbruch der Breitensuche
 return NULL
end
```

```
# Prüft auf abweichung => nicht Isomorph falls abweichend
_check_config_multisets(configs_g1, configs_g2, stage="refinement")
"Pasted image 20260402235701.png" konnte nicht gefunden werden.
# Erstellt an hand der Konfigurationen neue Labels
```

[...] ich weiß, es zerfetzt den Speicher und ist arg ineffizient.

Aufgabe 2: Gießroboter

[...] erhalten die Bäume maximal die n-fache Menge ihres benötigten Wassergehaltes, wobei n die Anzahl an verwendeten Robotern darstellt. Dies kann bei vielen Bäumen zwar zu Problemen führen, da es allerdings auch Bäume gibt, die einen großen Wasserüberschuss vertragen, zum Beispiel weil sie im Wasser stehen (Beispiel: Echt Sumpfzypresse), gehe ich im Folgenden davon aus, dass es sich in dieser Aufgabe um solche Bäume handelt.

Nun ist es endlich Zeit für die Zeitaufwandsanalyse.

Die ganze Aufgabe lässt sich sogar unter einem einzigen Namen beschreiben, nämlich das „fixed charge multi-depot Multiple Euclidean Traveling Salesman Problem“

Ich verstehe ehrlich gesagt nicht, warum ich die ersten 2 Zeilen dokumentieren soll, wenn die 2. nur aus der Eingabe kommt, aber bitte:

Rechnung: $1^2 + 1^2 = 1^2$ Geht nicht, da

$> 2 = 1^2$ |

> ca 1.6!=1

Natürlich sind die Bäume in den vorgegebenen Beispielen nicht zwangsweise durchschnittlich gleich dicht verteilt. Trotzdem ist es für zeitgenössische Botanische Gärten nicht unüblich, dass die intentionalen, kuratorischen Eingriffe durch die ästhetische Willkür der Gartenarchitekten als pseudozufällige Störungen zu modellieren sind, und letztlich statistisch nicht mehr von einem rein zufälligen, Poissonschen Rauschen zu unterscheiden ist.

```
While True:
 points ++
```

Die Funktion `greedy_approach()` ist darauf ausgelegt mit Clustern zu arbeiten, weshalb ich für die Anforderungen mit einem Cluster, das im Grunde genommen kein richtiges Cluster ist, ein Cluster erstellt habe.

Aufgabe 3: Lieferkette

Vorweg soll gesagt sein, dass ich leidenschaftlicher Euro-Truck-Simulator-2-Spieler bin. In diesem Spiel gibt es meines Wissens nach keine einzige Straße, die nur in eine Richtung befahren werden kann (Autobahnauffahrten ausgenommen).

Das bestreikte Werk liegt im besten Fall gar nicht auf unserer optimierten Hauptroute.

$$(\sum_{h=1}^{m}(s_{c_{d_{1,o,1},h}c_{d_{1,o,1},h+1}}) + u_{c_{d_{1,o,1},m+1}}), \dots$$

`besteRoute=MÄ`

`Straassennetz (Dijkstra, gecacht)`

Auf einem langsamen Laptop war es zu langsam und ich konnte kein Ergebnis abwarten, auf einem schnelleren waren Arbeitsspeicher und Festplatte während der Ausführung voll und meine Eltern wollten dafür keinen Gaming-PC eines anderen leihen.

Anstatt anzunehmen, dass C++ 15 oder 20 mal schneller ist als Python, habe ich den gesamten Code per Hand in C++ neugeschrieben.

`FinalCode_A3_dial_v4_final_ABGABEVERSION.cpp`

Dies wird für die Dijkstras sehr praktisch, da somit die Logik für die anderen Arbeitsschritte auch auf diese übertragen werden kann, ohne dass der Code wegen undefined behaviour eine Pizza bestellt.

```
# TODO REMOVE !!!!! TODO
ans = solve_problem(fileNum, True)
```

$R[n]$ definiert die $i$te Fabrik entlang der Strecke.

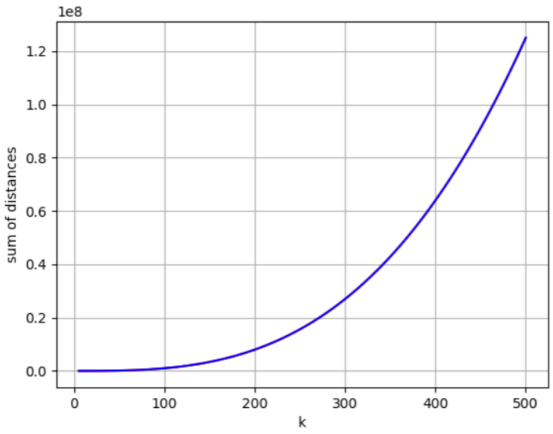
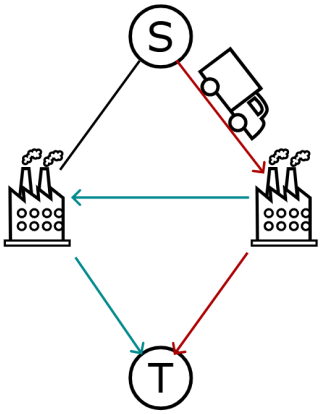


Abbildung 2: Blau: Summe der Weglängen, Rot (ist fast dieselbe Kurve): k^3



Als ich jedoch während der Doomscrolling-Phase zufällig auf „Minecraft: Grass vs. Mycelium“ Videos gestoßen bin ist mir aufgefallen, dass zwei Anfangsknoten in einem zusammenhängenden Graphen sich irgendwann in der Mitte treffen müssen, wodurch eine Berechnung doch noch möglich ist.

Allgemeines

Das ausführbare Programm ist Windows.

Diesmal habe ich jedoch noch mehr selbst erledigt, da die KI etwas überfordert war und mich sogar ein paar Mal in falsche Richtungen gelenkt hat. Ich denke, das war ein guter Weckruf, mehr selbst nachzudenken.

eine Fußnote: potentially citation needed

Der Quellcode ist in Python 14 geschrieben.

Bei dieser Aufgabe wurde keine künstliche Intelligenz genutzt, stattdessen wurde sie vollständig von menschlicher Dummheit bearbeitet.

Außerde hatte ich am Anfang IntelliJs lokale Inline Code-Completion aktiv, bis mich die konstanten falschen Vorschläge zu sehr genervt haben und ich sie deaktiviert habe.

Teile des Programmcodes, insbesondere das Grundgerüst der Eingabe, wurden aus Lösungen zu vergangenen Aufgaben entnommen, wofür ich die Copy-Paste-Funktion meines Linux-Desktops benutzt habe.

Um Gedanken (unter Anderem auf Bahnfahrten, während fantastischer Unterrichtsstunden im Bildungsbunker uvm.) festzuhalten, wurde tatsächlich zu Stift und Papier gegriffen.

Die Umsetzung dieses Projekts fand in Java mit maven statt. Grund dafür ist, dass es, anders als Python, tatsächlich auch läuft, wenn man es mal auf einem anderen Rechner testen will, relativ schnell ist und dann doch etwas einfacher und lesbarer als C ist. Außerdem braucht mein Projekt nicht 3 Jahre zum Aufsetzen, nur um dann zu failen (looking at you Gradle).

Interessante Wortneuschöpfungen:

- Algorythmen
- Tilenahme-ID
- Hertzstücke
- NP-Mitgliedschaft
- ein Indizie *i*

Ich möchte dieses Gelegenheit nutzen, um mich bei verschiedenen Leuten zu bedanken. Allen voran möchte ich mich bei Justus, Peter und Bob bedanken, deren Stimmen mich durch viele harte Nächte des Knobelns und Programmierens gebracht haben. Auch bedanken möchte ich mich bei dem ungesehenen Helden jedes BwInfs: Dem Computer. Ohne den wäre ich wahrscheinlich häufiger draußen aber ohne ihn wäre unsere Welt so viel düsterer. Wochenlang rechnet er, ohne jemals Dank zu fordern. Nicht bedanken möchte ich mich bei Claude Code: nicht nur, hat es mir regelmäßig trotz klarer Anweisungen den ganzen Code zerschossen, sondern wurden mir auch trotz des wirklich verschwindenden Nutzens 20 € von Anthropic in Rechnung gestellt.