Bundeswettbewerb Informatik

35. Bundeswettbewerb Informatik, 2. Runde

Lösungshinweise und Bewertungskriterien

Allgemeines

Es ist immer wieder bewundernswert, wie viele Ideen, wie viel Wissen, Fleiß und Durchhaltevermögen in den Einsendungen zur zweiten Runde eines Bundeswettbewerbs Informatik stecken. Um aber die Allerbesten für die Endrunde zu bestimmen, müssen wir die Arbeiten kritisch begutachten und hohe Anforderungen stellen. Von daher sind Punktabzüge die Regel und Bewertungen über die Erwartungen hinaus die Ausnahme. Lassen Sie sich davon nicht entmutigen! Wie auch immer Ihre Einsendung bewertet wurde: Allein durch die Arbeit an den Aufgaben und den Einsendungen hat jede Teilnehmerin und jeder Teilnehmer einiges gelernt; diesen Effekt sollten Sie nicht unterschätzen.

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

Bewertungsbogen Aus der ersten 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 PMS einsehen. In der ersten Runde ging die Bewertung noch von 5 Punkten aus, von denen bei Mängeln dann abgezogen werden konnte. In der zweiten Runde geht die Bewertung von 20 Punkten aus; dafür gibt es deutlich mehr Bewertungskriterien, bei denen Punkte abgezogen oder auch hinzuaddiert werden konnten.

Terminlage der zweiten Runde Für Abiturienten ist der Terminkonflikt zwischen Abiturvorbereitung und zweiten Runde sicher nicht ideal. Doch leider bleibt uns nur die erste Jahreshälfte für die zweite BwInf-Runde: In der zweiten Jahreshälfte läuft nämlich die zweite Runde des Mathewettbewerbs, dem wir keine Konkurrenz machen wollen. Aber: die Bearbeitungszeit für die zweite BwInf-Runde beträgt etwa vier Monate. Rechtzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, Konflikte mit dem Abitur zu vermeiden.

Dokumentation Es ist sehr gut nachvollziehbar, dass Sie Ihre Energie bevorzugt in die Lösung der Aufgaben, die Entwicklung Ihrer Ideen und die Umsetzung in Software fließen lassen. Doch ohne eine gute Beschreibung der Lösungsideen, eine übersichtliche Dokumentation der wichtigsten Komponenten Ihrer Programme, eine gute Kommentierung der Quellcodes und eine ausreichende Zahl sinnvoller Beispiele (die die verschiedenen bei der Lösung des Problems zu berücksichtigenden Fälle abdecken) ist eine Einsendung wenig wert. Bewerterinnen und Bewerter können die Qualität Ihrer Einsendung nur anhand dieser Informationen vernünftig einschätzen. Mängel können nur selten durch gründliches Testen der eingesandten Programme ausgeglichen werden – wenn diese 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 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.

Bewertungskriterien Bei den im Folgenden beschriebenen Lösungsideen handelt es sich um Vorschläge, nicht um die einzigen Lösungswege, die wir gelten ließen. Wir akzeptieren in der Regel alle Ansätze, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind. Einige Dinge gibt es allerdings, die – unabhängig vom gewählten Lösungsweg – auf jeden Fall diskutiert werden müssen. Zu jeder Aufgabe wird deshalb in einem eigenen Abschnitt erläutert, worauf bei der Bewertung dieser Aufgabe besonders geachtet wurde. Außerdem gibt es aufgabenunabhängig Anforderungen: an theoretische Analyse (Laufzeitüberlegungen, Begründungen), Dokumentation (insbesondere: klare Beschreibung der Lösungsidee, genügend aussagekräftige Beispiele, wesentliche Auszüge aus dem Quellcode enthalten), Quellcode (insbesondere: Strukturierung und Kommentierung, gute Übersicht durch Programm-Dokumentation) und Programm (keine Implementierungsfehler).

Danksagung An der Erstellung der Lösungsideen haben mitgewirkt: Michael Jungmair (Aufgabe 1), Adrian Lison (Aufgabe 2) sowie Niccolò Rigi-Luperti (Aufgabe 3).

Aufgabe 1: Rosinen picken

1.1 Abstrahierung der Aufgabenstellung und Definitionen

Gegeben ist eine Menge von Unternehmen V. Für jedes Unternehmen $v \in V$ ist der aktuelle Wert als rationale Zahl $w(v)$ bekannt. Zusätzlich sind noch Bedingungen der Form "Nimmst Du A, must Du auch B nehmen" gegeben. Diese Bedingungen können wir verallgemeinert als Relation E modellieren: $(a,b) \in E$ beschreibt, dass man mit Unternehmen a auch das Unternehmen b nehmen muss.

Für die Lösung der Aufgabe ist eine *erlaubte* Teilmenge $V' \subseteq V$ gesucht, so dass $\sum_{v' \in V'} w(v')$ möglichst groß wird. Eine Teilmenge V' nennen wir erlaubt, falls für alle $v' \in V'$ gilt:

$$(v', x) \in E \implies x \in V'$$

Leere wertvolle Teilmengen In der Aufgabenstellung ist nicht spezifiziert, ob die wertvolle Teilmenge auch leer sein darf. Wenn die beste nichtleere Teilmenge von Unternehmen immer noch ein Verlustgeschäft ist, so wird in einer kapitalistischen Welt wie unserer jeder Investor (auch Du) dankend ablehnen. Hier gilt sinnvollerweise die Devise: kein Geschäft (entspricht der leeren Teilmenge) ist besser als ein Verlustgeschäft. Natürlich kann man sich auch Szenarien ausdenken, in denen nichtleere Teilmengen verboten sind (z.B. politische Situation). Dies führt jedoch zu einem schwereren Problem und wäre damit eine mögliche Erweiterung der Aufgabenstellung.

Modellierung als Graph

Das Tupel (V,E) kann man auch als gerichteten Graphen G interpretieren. Dabei entspricht V der Menge der Knoten und E der Menge der gerichteten Kanten. $w(v)$ entspricht dem Knotengewicht. Eine erlaubte Teilmenge V' von Knoten (und damit Unternehmen) erkennt man im Graphen daran, dass es keine Kanten von einem Knoten aus V' zu einem Knoten aus $V \setminus V'$ gibt. In einem gerichteten Graphen bezeichnet man eine solche Teilmenge als „Closure".

Unter dem „Closure Problem" versteht man das Optimierungsproblem, in einem gerichteten Graphen das Closure zu finden, bei dem die Summe der Knotengewichte möglichst groß oder möglichst gering ist. Bei der Aufgabenstellung handelt es sich daher um ein Closure-Problem.

1.2 Brute Force

Wie bei vielen (Optimierungs-)Problemen kann man sich zum Vergleich zuerst mit der einfachsten Strategie beschäftigen: Brute-Force. Ein Ansatz könnte wie folgt aussehen:

Algorithmus 1 Brute-Force-Algorithmus

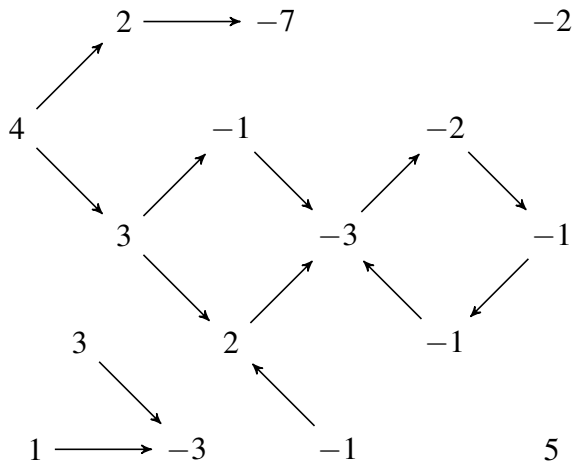
```
Input: G(V,E)
besteSummme ← 0
besteTeilmenge ← ∅
for jede mögliche Teilmenge T von V do
    if T erlaubt then
        berechne Summe = ∑_{t∈T} w(t)
        if Summe > besteSumme then
            besteSumme ← Summe
            besteTeilmenge ← T
        end if
    end if
end for
Output: besteTeilmenge
```

Wie in Algorithmus 1 zu sehen, muss ein Brute-Force-Algorithmus jede mögliche Teilmenge betrachten und diese dann überprüfen. Dafür liefert er garantiert das optimale Ergebnis.

Da es genau $2^{|V|}$ Teilmengen von V gibt, liegt Brute-Force in $\Omega(2^{|V|})$. Praktisch bedeutet dies: Selbst wenn ein Computer nur eine Nanosekunde pro möglicher Teilmenge benötigen würde, würde das Verfahren bei $|V| = 100$ ca. 10 Billionen Jahre benötigen. Dies ist eindeutig zu viel, vor allem, da ein solches Beispiel zur Bearbeitung vorliegt.

1.3 Vereinfachung des Ausgangsproblems

Je kleiner und einfacher die Eingabe ist, desto schneller laufen die darauf angewandten Algorithmen. Dies kann man sich zunutze machen, in dem man mit „günstigen“ Vereinfachungen die ursprüngliche Eingabe in eine äquivalente, kleinere Eingabe transformiert. Bei den nachfolgenden Vereinfachungen gehen wir davon aus, dass die Aufgabenstellung als gerichteter, knotengewichteter Graph vorliegt. Um die einzelnen Vereinfachungen zu zeigen, arbeiten wir beispielhaft mit folgendem Graphen:



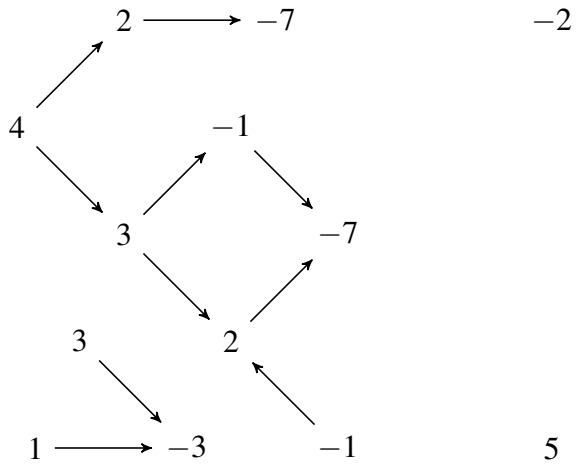
Kondensation

Bei Graphen bietet es sich bei vielen Problemen an, Zyklen zu entfernen. Dies reduziert einerseits die Größe, andererseits ist der resultierende Graph azyklisch und somit sind viele Algorithmen auch anwendbar.

Wenn es von einem Knoten u zu einem Knoten v einen Pfad gibt, also u nicht ohne v genommen werden kann, und es auch einen Pfad von v nach u gibt, so können v und u zu einem Knoten v_u kondensiert werden. Dies ist korrekt, da v und u, da sie ja gegenseitig abhängig sind, entweder beide genommen oder beide nicht genommen werden.

Wir bezeichnen eine Teilmenge T von Knoten in einem gerichteten Graphen als *starke Zusammenhangskomponente*, wenn von jedem Knoten aus T ein Pfad zu jedem anderen Knoten aus T existiert. Für das Finden von starken Zusammenhangskomponenten existieren schnelle ($\mathbf{O}(|V|)$) Algorithmen wie z.B. von Tarjan[1].

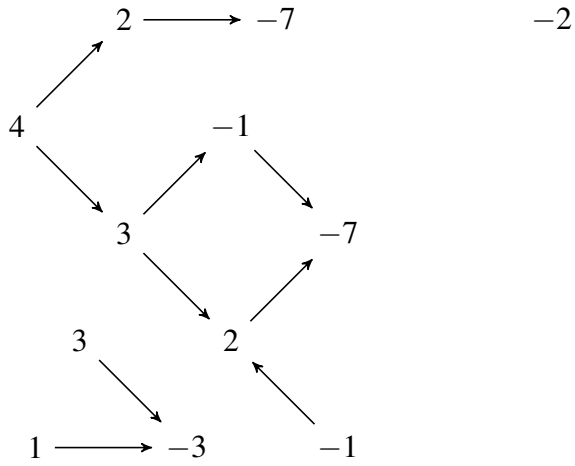
Als kondensiert bezeichnen wir einen Graphen, der um seine Zyklen bereinigt wurde, in dem starke Zusammenhangskomponenten zu einem Gemeinschaftsknoten kondensiert wurden. Dieser hat sinnvollerweise als Knotengewicht die Summe aller Knotengewichte der Knoten, für die er steht. Nach Kondensation entsteht der folgende Graph:



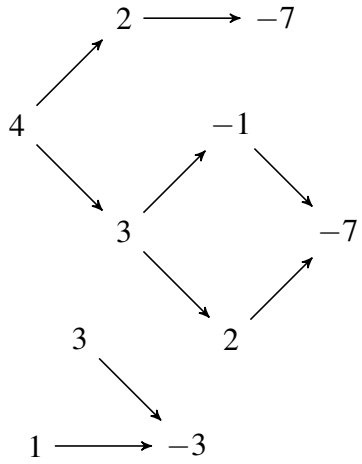
Weitere sinnvolle Vereinfachungen

Nicht negative Knoten ohne ausgehende Kanten ... entsprechen Unternehmen ohne Verlust, die dir ohne Bedingungen geschenkt werden. Es wäre ja blöd, diese nicht zu nehmen! Dementsprechend können alle solchen Knoten aus dem Graphen entfernt und direkt zur Ergebnismenge hinzugefügt werden. Nach diesem Schritt sieht der Graph folgendermaßen aus:

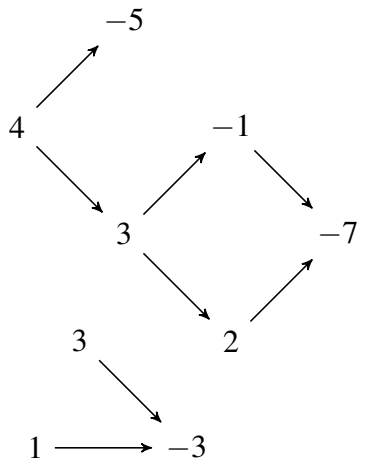
[1] siehe auch `https://de.wikipedia.org/wiki/Algorithmus_von_Tarjan_zur_Bestimmung_starker_Zusammenhangskomponenten`



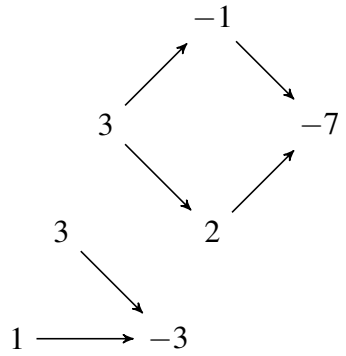
Negative Knoten ohne eingehende Kanten ... können direkt aus dem Graphen entfernt werden, da sie für keinen anderen Knoten erforderlich sind und selbst nie in der wertvollen Teilmenge landen werden:



Hochziehen von negativen Blättern Unter einem Blatt versteht man einen Knoten ohne ausgehende Kanten. Hat ein negatives Blatt nur eine eingehende Kante vom Knoten u, so wird das Blatt zu u hinzugefügt:



Wiederholte Anwendung von Regeln Natürlich sind die Regeln dann solange anzuwenden, bis keine mehr greift:



1.4 Reduktion des Closure-Problems auf Min-Cut

In der Bergbauindustrie ist das Closure-Problem wichtig, um die optimale Grubenform im Tagebau anhand geologischer Untersuchungen zu berechnen. Dabei wird das Erdreich in Blöcke unterschiedlicher Wertigkeit aufgeteilt. Um einen tieferen Block zu gewinnen müssen natürlich alle darüber gelegenen Blöcke abgebaut werden. Das ist ein Closure-Problem; und da der Bergbau nicht das einzige Anwendungsgebiet ist, in dem Closure-Probleme auftreten, existieren eine Vielzahl von Lösungen in der Literatur[2][3][4][5].

[2] http://riot.ieor.berkeley.edu/ dorit/pub/scribe/lec11/Lec11posted.pdf

[3] Hochbaum, D. S. (2001), A new—old algorithm for minimum-cut and maximum-flow in closure graphs. Networks, 37: 171–193. doi:10.1002/net.1012

[4] Picard, J 1976, 'MAXIMAL CLOSURE OF A GRAPH AND APPLICATIONS TO COMBINATORIAL PROBLEMS', Management Science, 22, 11, pp. 1268-1272, Business Source Complete, EBSCOhost, viewed 17 April 2017

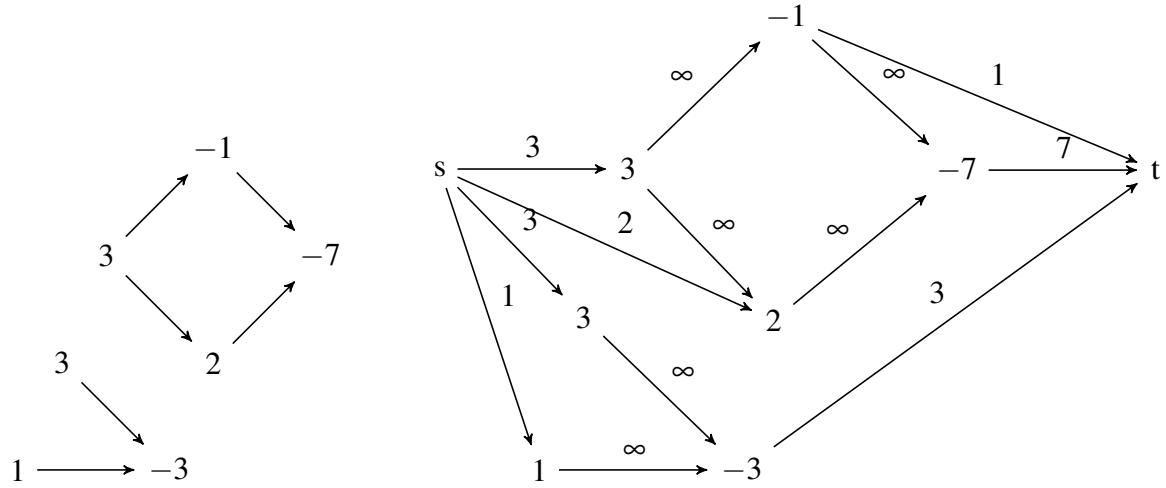
[5] Cook, William J.; Cunningham, William H.; Pulleyblank, William R.; Schrijver, Alexander (2011), "Optimal closure in a digraph", Combinatorial Optimization, Wiley Series in Discrete Mathematics and Optimization, 33, John Wiley & Sons, pp. 49–50, ISBN 9781118031391

Picard konnte 1976 nachweisen, dass man das Closure-Problem lösen kann, indem man es in ein so genanntes Min-Cut Problem überführt. Min-Cut-Probleme lassen sich wiederum gut mit Flussalgorithmen lösen. In diesem Abschnitt stellen wir diesen Lösungsweg schrittweise vor.

Beschreibung der Reduktion

Für die Reduktion wird ein neuer Graph G' erzeugt. G' enthält dabei neben allen Knoten aus G auch einen Knoten s (source) und einen Knoten t (target). Das Gewicht einer Kante e in G' heißt *Kapazität* und wird mit $c(e)$ bezeichnet. Alle gerichteten Kanten e, die in G enthalten sind, sind auch in G' enthalten, allerdings mit Kantengewicht $c(e) = \infty$. Zusätzlich werden neue Kanten hinzugefügt:

1. Es wird eine Kante e von s zu einem Knoten v mit Kantengewicht $c(e) = w(v)$ eingefügt, falls $w(v) > 0$.
2. Es wird eine Kante e von einem Knoten v zu t mit Kantengewicht $c(e) = |w(v)|$ eingefügt, falls $w(v) \leq 0$.



In diesem neuen Graphen, kann das Problem nun mittels eines minimalen Schnittes effizient gelöst werden: Ein minimaler Schnitt ist eine Menge von Kanten mit möglichst geringem Kantengewicht, ohne die s und t nicht mehr verbunden sind. Hat man einen minimalen Schnitt gefunden, so ist das Closure mit der größten Knotensumme genau die Menge an Knoten, die anschließend noch von s aus erreichbar sind.

Warum die Reduktion funktioniert

Um nachzuvollziehen, wieso der minimale Schnitt in G' zum Ziel führt, ist es wichtig G' so zu interpretieren:

- Alle Knoten, die (nach einem Schnitt) von s aus erreicht werden können sind in der wertvollen Teilmenge enthalten

- Da t nicht in der wertvollen Teilmenge enthalten sein kann, müssen eventuell Kanten entfernt werden, so dass es keinen Pfad von s nach t gibt. Für das Entfernen von Kanten gibt es dabei 3 Möglichkeiten:

  1. Das Entfernen einer Kante (s,v) entspricht dem Entfernen von v aus der wertvollen Teilmenge. Dies kostet uns natürlich $w(v)$, da $w(v) > 0$. Diese Kosten entsprechen genau dem Kantengewicht.
  2. Das Entfernen einer Kante (u,v) entspricht dem Ignorieren einer Bedingung und ist daher nicht erlaubt. Die Kosten (∞) entsprechen dabei wieder dem Kantengewicht.
  3. Das Entfernen einer Kante (v,t) entspricht dem Behalten eines negativen Knotens in der wertvollen Teilmenge. Dies kostet natürlich $|w(v)|$. Auch hier ist das genau das Kantengewicht.

Da das Entfernen einer Kante immer Kosten bedeutet, wollen wir natürlich genau die Menge von Kanten auswählen, die s und t bei möglichst geringen Kosten trennt. Dies ist eben der minimale Schnitt. Anders formuliert: Entweder wir schließen einen positiven Knoten v aus (Kante von s nach v entfernen) oder wir müssen alle negativen Knoten x auswählen, für die es einen Pfad von v nach x gibt (Kante von x nach t entfernen).

Min-Cut und Maximaler Fluss

Nachdem wir die Lösung unseres Problems auf die Lösung des minimalen Schnittes reduziert haben, müssen wir uns nun Gedanken darüber machen, wie wir diesen ermitteln. Dies gelingt über die Berechnung des maximalen Flusses. Ein maximaler Fluss f in einem Graphen ist eine Funktion $f : E \to \mathbb{N}$ die jeder Kante e einen Fluss $f(e)$ zuweist, wobei natürlich $f(e) \leq c(e)$. Zusätzlich muss für jeden Knoten noch gelten, dass genauso viel in den Knoten hinein fließt, wie hinaus fließt:

$$\forall v \in V \setminus \{s,t\} : \sum_{e \in \delta^-(v)} f(e) = \sum_{e \in \delta^+(v)} f(e)$$

Dabei bezeichnet $\delta^-(v)$ die Menge der eingehenden und $\delta^-(v)$ die Menge der ausgehenden Kanten eines Knotens v. Die Kapazität eines maximalen Flusses $c(f)$ ist der Betrag, der bei f aus s hinaus bzw. in t hinein fließt.

Das Max-Flow-Min-Cut-Theorem besagt: *Ein maximaler Fluss im Netzwerk hat genau den Wert eines minimalen Schnitts.*[6] Daraus lässt sich ableiten, dass für jede Kante e, die Bestandteil eines minimalen Schnittes ist, $f(e) = c(e)$ gilt. Dies lässt sich nutzen, um über den maximalen Fluss und das zugehörige Residualnetzwerk den minimalen Schnitt zu ermitteln. Als Residualgraph oder Residualnetzwerk (auch Restnetzwerk) wird in der Graphentheorie ein Netzwerk bezeichnet, das die restlichen Kantenkapazitäten bezüglich eines Flusses anzeigt.

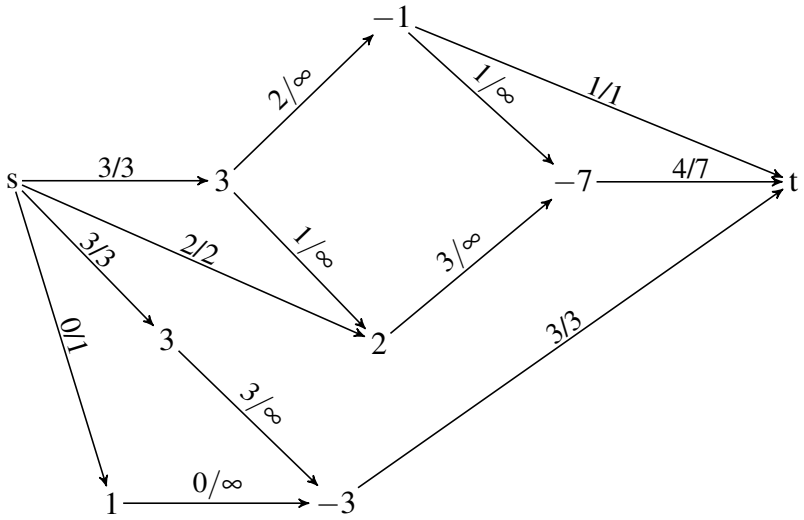
[6] https://de.wikipedia.org/wiki/Max-Flow-Min-Cut-Theorem

Schrittweise Berechnung des minimalen Schnittes

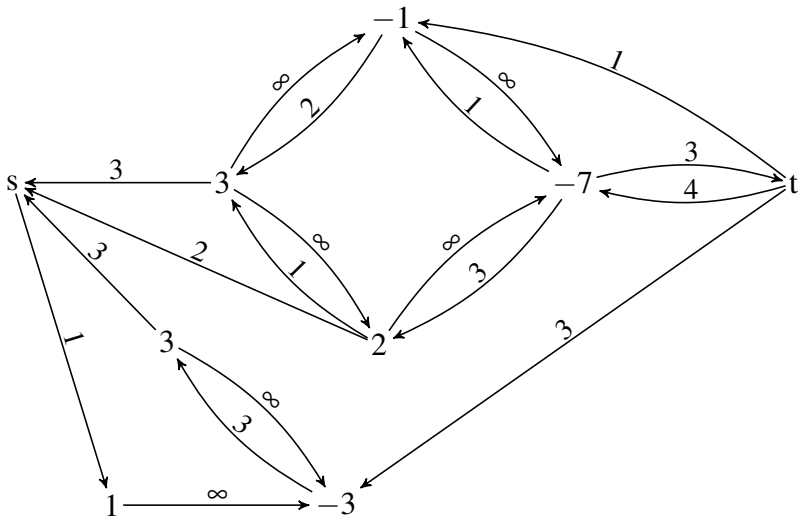
Um nun in G' die wertvolle Teilmenge zu berechnen, muss zunächst der maximale Fluss in G' und der entsprechende Residualgraph ermittelt werden. Die folgenden Algorithmen können zum Finden des maximalen Flusses genutzt werden und berechnen dabei direkt auch einen Residualgraphen:

- Algorithmus von Ford und Fulkerson in $\mathbf{O}(|E| * f)$ (pseudopolynomiell zur Kapazität des maximalen Flusses, damit pseudopolynomiell zum Wert der Unternehmen)
- Algorithmus von Edmonds und Karp in $\mathbf{O}(|V| * |E|^2)$
- Algorithmus von Dinic in $\mathbf{O}(|V|^2 * |E|)$
- Goldberg-Tarjan-Algorithmus in $\mathbf{O}(|V|^2 * |E|)$

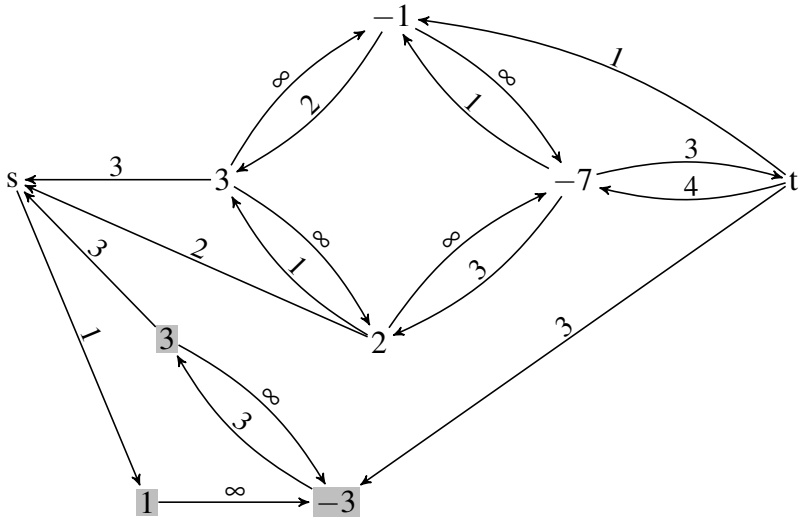
Der Residualgraph könnte aber auch direkt aus dem maximalen Fluss ermittelt werden. Ein maximaler Fluss im Beispiel ist in folgender Grafik dargestellt. Die Kantenbeschriftungen stehen dabei für $f(e)/c(e)$.



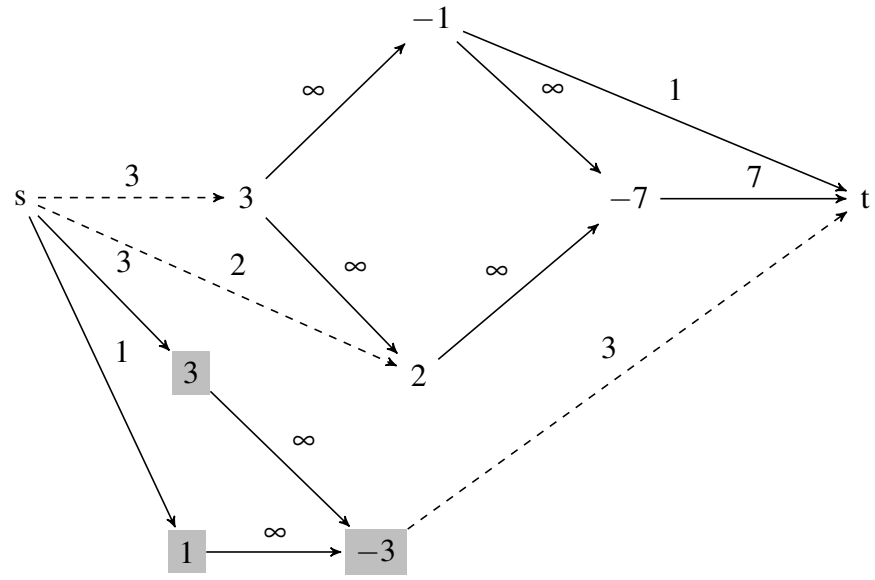
Das zum Fluss gehörende Residualnetzwerk sieht so aus:



Aus dem Residualgraphen kann man den minimalen Schnitt berechnen, in dem von s aus die Menge der Knoten S bestimmt wird, die von s aus erreichbar sind. Alle Knoten, die nicht von s aus erreichbar sind nennen wir T. Der minimale Schnitt MC besteht nun aus allen Kanten $(s',t') \in E, s' \in S, t' \in T$. Da wir zum Lösen der Aufgaben die genauen Kanten des Minimalen Schnittes nicht benötigen, können wir direkt nach ermitteln von S aufhören; dies ist genau die wertvolle Teilmenge aus V. Im Beispiel sind im Residualgraph die folgenden Knoten erreichbar:



Dies ist bereits die wertvolle Teilmenge. Obwohl die Kanten des minimimalen Schnitts für die Aufgabe nicht ermittelt werden müssen zeigt die folgende Graphik jedoch den Zusammenhang ganz gut, wobei gestrichelte Kanten den minimalen Schnitt bilden:



1.5 Formulierung als Integer Programming bzw. Linear Programming Problem

Als Integer Programming Problem (IPP) bezeichnet man ein Optimierungsproblem, in dem Variablen nur ganzzahlige Werte annehmen dürfen. Da die Aufgabe ein Optimierungsproblem darstellt, bietet sich ein die Modellierung als IPP an. Die Aufgabenstellung wird als IPP modelliert indem man für jedes $v \in V$ eine binäre Variable b_v einführt. b_v ist genau dann 1, wenn $v \in V'$, sonst 0. Offensichtlich gibt es einen zu maximierenden Term:

$$\sum_{v \in V} b_v * w(v)$$

Natürlich müssen auch die Nebenbedingungen formuliert werden: Für jedes Paar $(c,d) \in E$ geben wir eine Ungleichung an:

$$b_c \leq b_d$$

Dies entspricht genau der Forderung, dass c nur in der ausgewählten Teilmenge sein darf, falls d auch ausgewählt ist. Um die Variablen auf 0 und 1 einzuschränken muss noch folgende Bedingung erfüllt sein:

$$\forall x : b_x \in \{0,1\}$$

Diese Problemstellung könnten wir bereits mit gängigen Methoden lösen. Jedoch ist das allgemeine, optimale Lösen eines Integer-Programming-Problems laufzeittheoretisch nicht besser als Brute-Force.

Da die den Nebenbedingungen entsprechende Matrix jedoch besondere Eigenschaften erfüllt (total unimodular), kann das Problem auch in folgender Linear-Programming (LP) Variante formuliert werden, ohne auf ganzzahlige Ergebnisse zu verzichten:

$$\begin{aligned} &\textbf{max} \sum_{v \in V} b_v * w(v) \\ &b_c \leq b_d \\ &\forall x : 0 \leq b_x \leq 1 \end{aligned}$$

Im Gegensatz zu IPP können Variablen in einem LP-Problem auch nicht-ganzzahlige Werte annehmen. Hier ist es in polynomieller Zeit möglich, eine optimale Lösung zu finden. Dazu gibt es eine Vielzahl von Algorithmen. Neben dem relativ bekannten Simplex-Verfahren gibt es noch die sogenannten Inneren-Punkte-Verfahren. Während der Simplex-Algorithmus zwar in der Praxis effizient ist, so ist die Laufzeit im worst-case immer noch exponentiell. Im Gegensatz dazu schafft es ein Innere-Punkte-Verfahren in polynomieller Zeit, die Lösung zu finden, ist in der Praxis jedoch etwas langsamer. Da das Implementieren der Verfahren jedoch sehr komplex ist bietet es sich an auf bestehende (und ausreichend geprüfte) Implementierungen wie zum Beispiel `GNU-GLPK` zurückzugreifen.

1.6 Weitere Ansätze

Eine weitere Möglichkeit, die zumindest für die zur Verfügung gestellten Beispiele ausreicht[7] ist das sogenannte Backtracking. Im Gegensatz zu Brute-Force werden nicht alle möglichen Teilmengen zusammengestellt und überprüft, sondern bereits sehr früh möglichst viele unerlaubte Teilmengen ausgeschlossen. Dadurch reduziert sich die praktische Laufzeit (bei geschickter) Implementierung so, dass (bei entsprechender Vereinfachung) gerade noch die vorgegebenen Beispiele bearbeitet werden können. Theoretisch betrachtet liegt die Komplexität natürlich immer noch bei $\mathbf{O}(2^{|V|})$. Daher sind die anderen Verfahren natürlich deutlich zu bevorzugen.

Gerade bei Optimierungsproblemen wie diesem gibt es aber auch vielfältige Möglichkeiten, auf heuristische Strategien wie Hill Climbing, Simulated Annealing, Ant Colony Optimization etc. zurückzugreifen oder eine komplett eigene Heuristik zu entwickeln. Da aber in diesem Fall hinreichend schnelle optimale Algorithmen existieren, ist das Verwenden von Heuristiken in diesem Fall nicht sinnvoll und soll hier nicht weiter besprochen werden.

1.7 Bewertungskriterien

- Die Problemstellung kann offensichtlich als Graphenproblem modelliert werden. Das sollte sich in der Einsendung zumindest implizit wiederfinden.
- Unabhängig davon welches Verfahren angewendet wurde, darf es nie zu einer Lösung kommen, bei der Nebenbedingungen nicht beachtet wurden.
- Heuristiken sind zwar ein möglicher Ansatz, garantieren aber keine optimale Lösung. Weil polynomielle optimale Verfahren existieren, wird ein optimales Verfahren erwartet. Für die vorgegebenen Beispiele können sogar nach – in der Regel deutlicher – Vereinfachung des Graphen mit Brute-Force-Ansätzen die wertvollsten (also optimalen) Teilmengen gefunden werden. Für Verfahren, die besonders schwache Ergebnisse liefern, kann es deutliche Punktabzüge geben.
- Zu nicht optimalen Ergebnissen kann es auch durch Fehler im Verfahren kommen, z.B. durch unzulässige Vereinfachungen des Graphen.
- Die Laufzeit sollte „in Ordnung“ sein; die vorgegebenen Beispiele sollten alle mit kurzen Laufzeiten bearbeitet werden können. Bei Brute-Force-Ansätzen ohne Vereinfachungen oder kombinatorisch noch aufwändigeren Verfahren ist die Laufzeit nicht mehr in Ordnung.
- Besonders wenn Verfahren aus der Literatur verwendet werden, muss gut begründet werden, dass das Verfahren funktioniert; es soll insbesondere erkennbar werden, dass das Verfahren vollständig verstanden wurde.
- Teilaufgabe 3 fordert ausdrücklich zur Diskussion von Optimalität und Laufzeit auf. Es muss begründet werden, ob und wieso das gewählte Verfahren immer eine optimale Lösung findet. Bei Heuristiken sollte also erkannt worden sein, dass es Probleme gibt;

[7] Dies wurde durch eine eigene Implementierung überprüft.

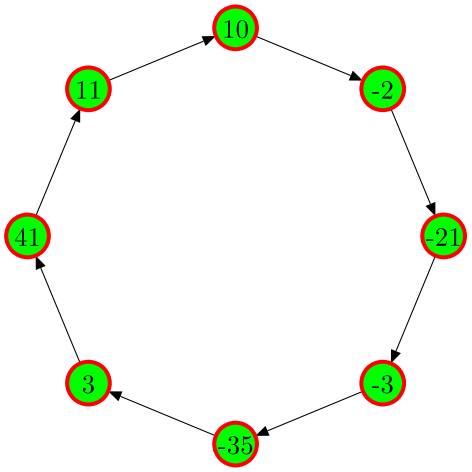
bei Brute-Force-Ansätzen mit Vereinfachungen sollten Beispiele angeben werden, bei denen die Vereinfachungen scheitern und die Laufzeit doch problematisch wird.

- Das Programm muss auf alle angegebenen Beispiele angewendet worden und die Ergebnisse in der Dokumentation enthalten sein. Zusätzliche Beispiele sind nicht gefordert, können aber das Potenzial der Lösung zeigen. Die Ausgabe sollte die wesentlichen Angaben, insbesondere den Wert der gefundenen Teilmenge, übersichtlich darstellen.
- Bei der Implementierung wird ein korrektes Einlesen der Eingabe erwartet sowie eine Ausgabe, die dem angegebenen Format entspricht.
- Das Problem lässt sich nur schwer erweitern. Es ist denkbar, andersartige Nebenbedingungen einzuführen, z.B.: Nimmst du A, darfst du B nicht nehmen.

1.8 Beispiele

Im Folgenden sind die Lösungen zu den vorgegebenen Beispielen grafisch dargestellt. Grün gefärbte Knoten bilden die optimale, also wertvollste Teilmenge, rot umrandete Knoten eine von einer Heuristik gefundene wertvolle Teilmenge. Über (und unter) den Abbildungen sind für beide Ergebnisse die Wertsummen angegeben; außerdem steht rechts eine Tabelle mit den optimalen Wertsummen.

Kreis-8 Heuristik: 4.0, Optimal: 4.0

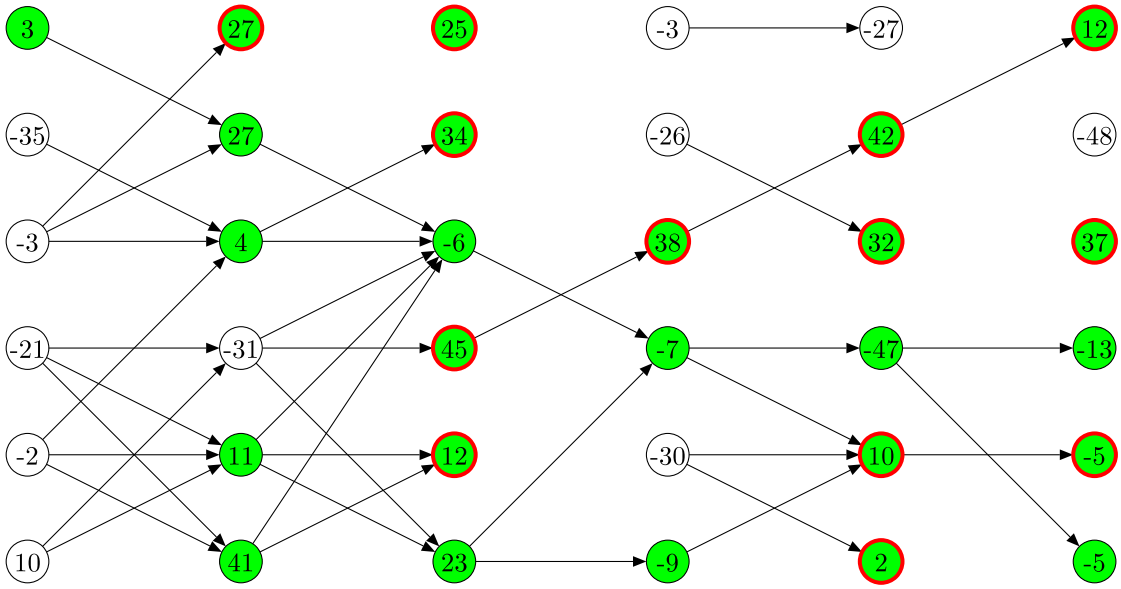


Heuristik: 4.0 Optimal: 4.0

Wertsummen der wertvollsten Teilmengen

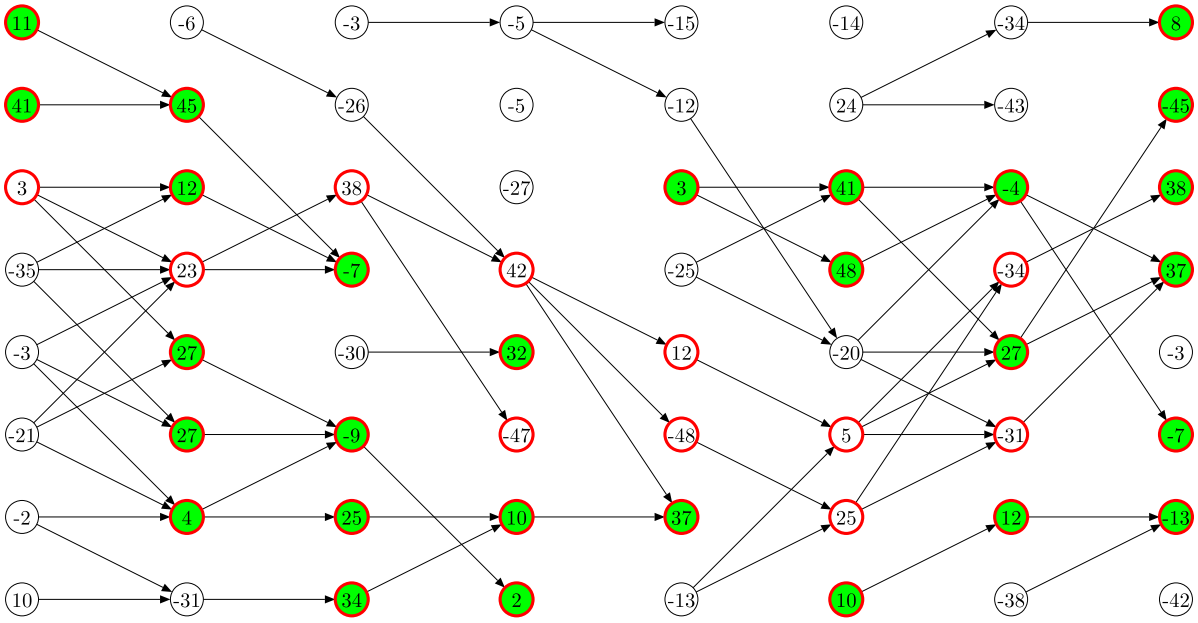
Beispiel	*Wertsumme*
Kreis-8	4
Quadrat-6	333
Quadrat-8	446
Quadrat-13	961
Zufall-7	0
Zufall-40	504
Zufall-100	613

Quadrat-6 (Dieses Beispiel wurde in der Aufgabenstellung abgedruckt.)
Heuristik: 311.0, Optimal: 333.0



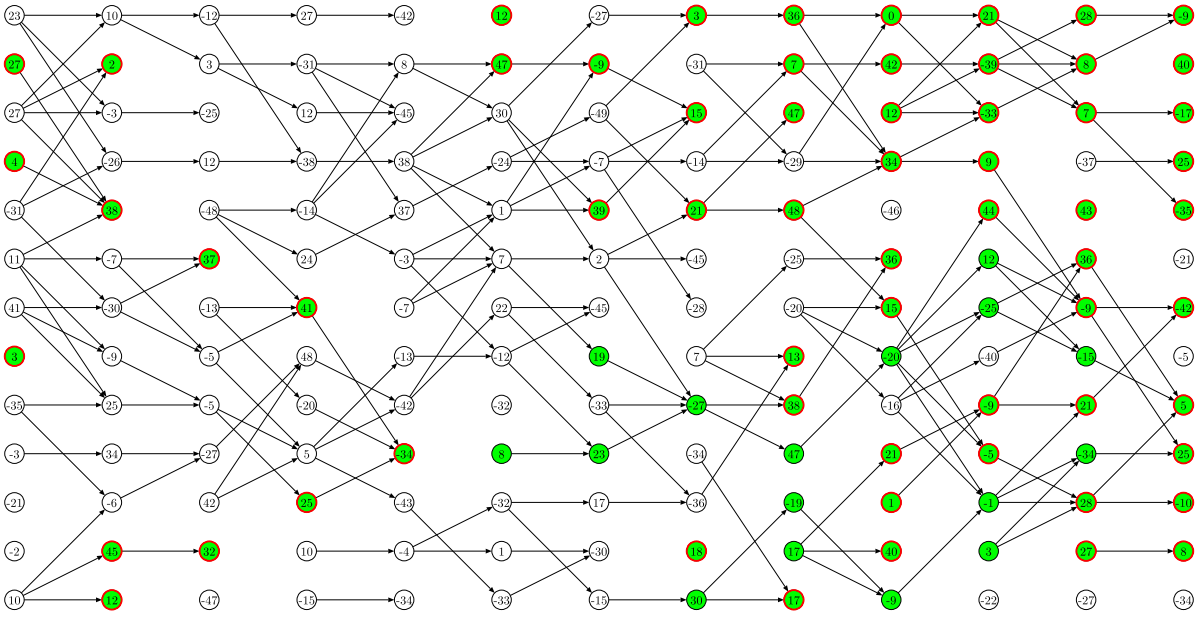
Heuristik: 311.0 Optimal: 333.0

Quadrat-8 Heuristik: 434.0, Optimal: 446.0



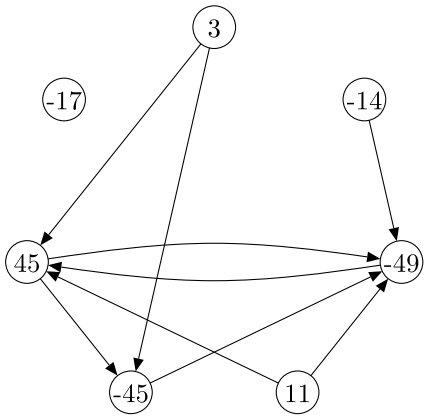
Heuristik: 434.0 Optimal: 446.0

Quadrat-13 Heuristik: 952.0, Optimal: 961.0



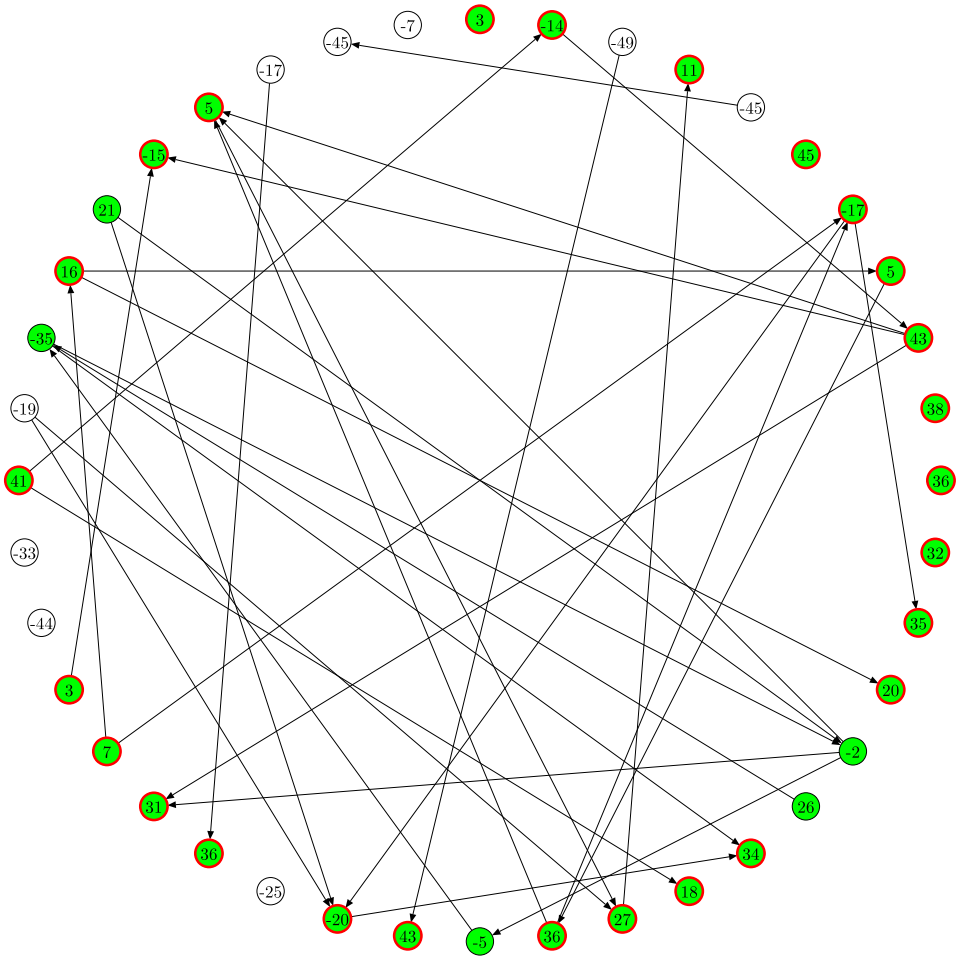
Heuristik: 952.0 Optimal: 961.0

Zufall-7 Heuristik: 0.0, Optimal: 0.0



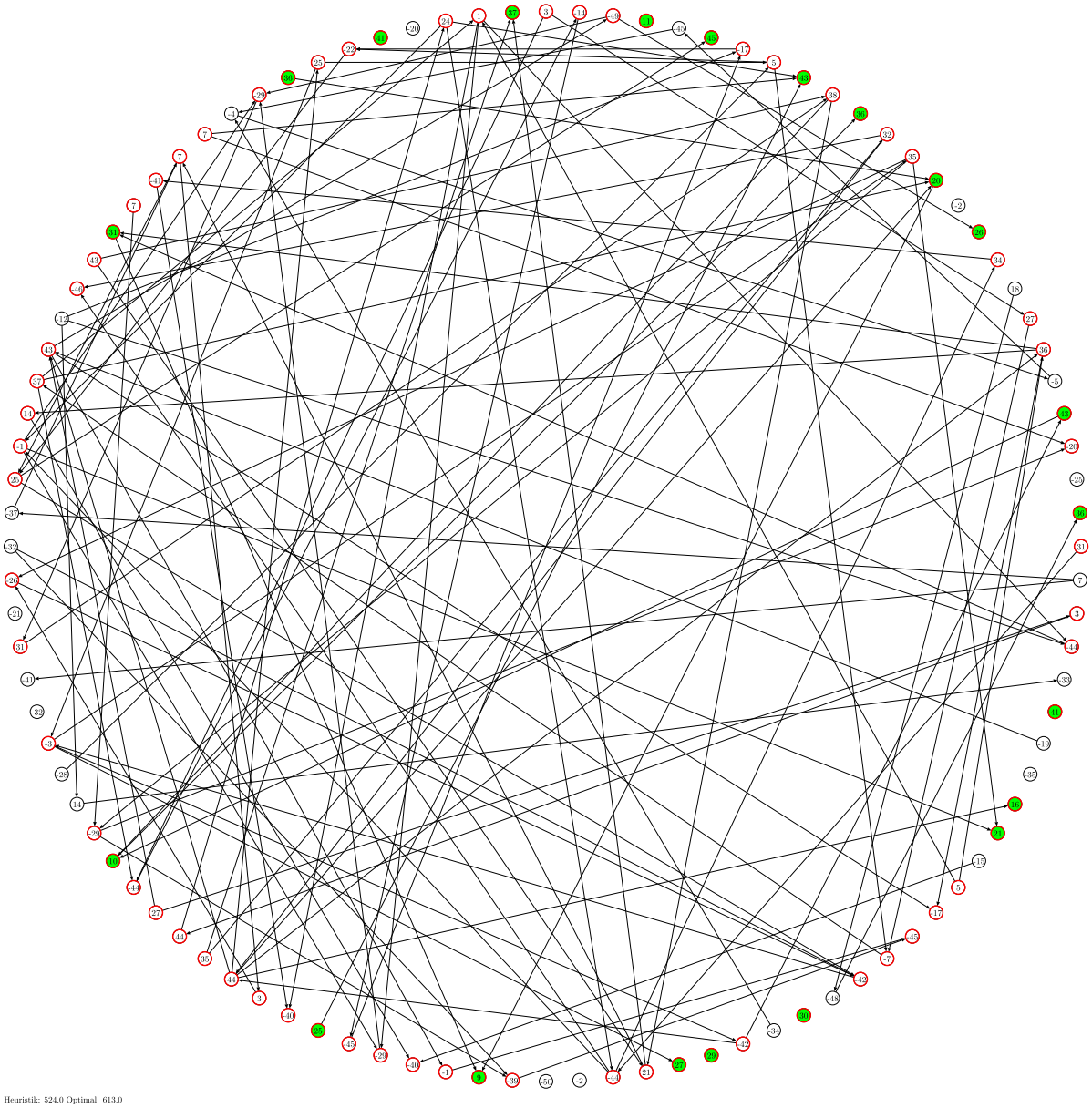
Heuristik: 0.0 Optimal: 0.0

Zufall-40 Heuristik: 499.0, Optimal: 504.0



Heuristik: 499.0 Optimal: 504.0

Zufall-100 Heuristik: 524.0, Optimal: 613.0



Heuristik: 524.0 Optimal: 613.0

Aufgabe 2: Rechtsrum in Rechthausen

2.1 Definition des Linksabbiegens

Unter Abbiegen versteht man im Straßenverkehr im Allgemeinen die Fahrtrichtungsänderung nach links oder rechts an Kreuzungen oder Einmündungen. Für unser Problem ist die Frage zu klären, was "links" beim Abbiegen genau bedeutet und wann beim Übergang von einem Straßenabschnitt in einen anderen eine für die Verkehrssicherheit von Rechthausen relevante Fahrtrichtungsänderung vorliegt.

Was wir vorrangig vermeiden wollen, sind sogenannte Abbiegeunfälle (Unfälle zwischen Abbieger und Gegenverkehr auf derselben Straße). Das hat Auswirkungen auf unsere Definition. Sei d wie in der Aufgabenstellung die Anzahl der Straßenabschnitte, die in eine Kreuzung einmünden.

- Im Fall $d = 1$ (das Ende einer Sackgasse) sollte das Wenden immer erlaubt sein, denn es können gar keine Abbiegeunfälle enstehen, weil es keinen Gegenverkehr gibt.
- Im Fall $d = 2$ (zwei Straßen treffen aufeinander) sollte das Wenden (U-Turn) zwar nicht mehr erlaubt sein, weil dies zu Zusammenstößen führen kann. Das Abbiegen in die andere Straße ist jedoch immer unproblematisch, egal ob man nach links oder rechts fährt, weil das Auto dabei stets auf seiner Fahrspur bleibt.
- Bei Fällen mit $d > 2$ wird es interessant. Auch hier reicht es nicht aus, zu überprüfen, in welche Richtung das Auto fährt. Abb. 1 zeigt eine Situation, in der ein Auto in zwei Straßen einbiegen könnte, die jeweils im Bezug zur aktuellen Straße nach links gerichtet sind, jedoch sollte nur das Abbiegen in Straße Nr. 2 verboten werden. Bei Straße 1 schneidet der Fahrer nämlich nicht die Gegenspur.

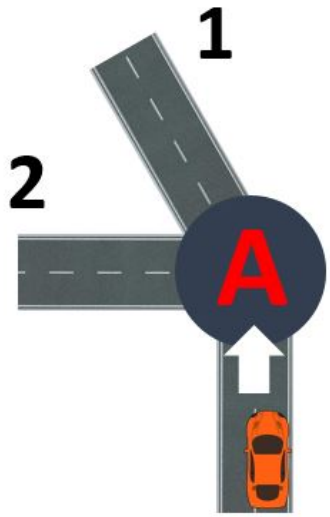
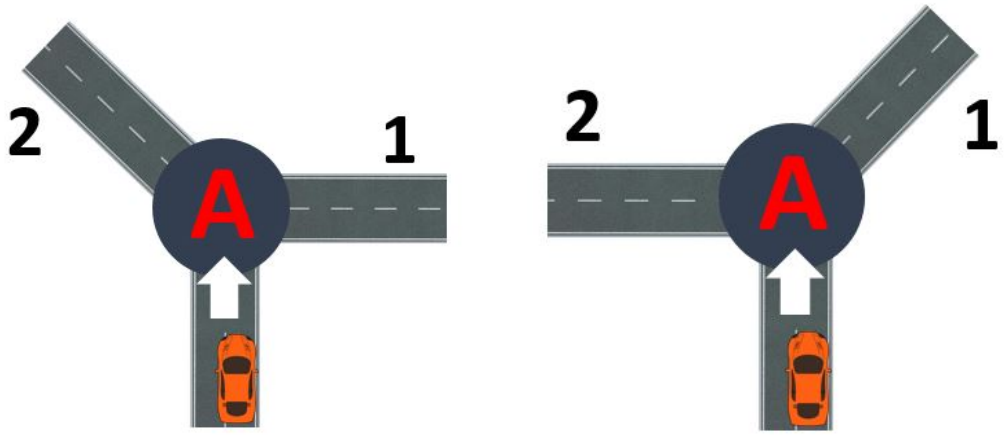


Abbildung 1: Zwei nach links gerichtete Abbiegemöglichkeiten.

Definition des Gegenverkehrs

Um allgemeine Aussagen treffen zu können, benötigen wir eine einheitliche Definition für den Gegenverkehr bei Kreuzungen. Im Idealfall, wenn zwei Straßen sich exakt gegenüberstehen, kommt der Gegenverkehr aus der genau anderen Richtung (180 Grad) wie man selbst. Hiervon ausgehend definieren wir Gegenverkehr als den Verkehr, der aus derjenigen Straße der Kreuzung kommt, die der Straße, aus der man selbst kommt, am stärksten entgegengesetzt ist. Einige Beispiele:

- Für $d = 1$ gibt es keine anderen Straßen, also auch keinen Gegenverkehr.
- Für $d = 2$ gibt es nur eine andere Straße; aus genau dieser kommt der Gegenverkehr.
- Für $d = 3$ gibt es zwei andere Straßen; aus der stärker entgegengesetzten davon kommt der Gegenverkehr. In Abbildung 2a ist das die „linke“ Straße (Nr. 2), in Abbildung 2b ist es die „rechte“ Straße (Nr. 1).
- Für alle größeren d gilt analog, dass der Gegenverkehr aus der am stärksten entgegengesetzten Straße kommt.



(a) Gegenverkehr von links, aus Straße Nr. 2. (b) Gegenverkehr von rechts, aus Straße Nr. 1.

Abbildung 2: Gegenverkehr in verschiedenen Situationen mit $d = 3$

Diese Definition entspricht im Allgemeinen auch der realen Konstruktion von Kreuzungen. Im Normalfall kann in die gegenüberliegende Straße am leichtesten/schnellsten eingefahren werden, oft aufgrund von Vorfahrtsregelungen (abknickende Vorfahrtsstraßen sind eine Ausnahme). Dort ist auch das Risiko am größten, dass ein Linksabbieger mit dem durchfahrenden Gegenverkehr kollidiert. Wir wollen also bei allen Straßen das Abbiegen verbieten, die vom Fahrer aus gesehen “links” von der Gegenverkehrsstraße liegen.

Winkelberechnung

Dazu stellen wir uns eine Kreuzung abstrakt als Kreis vor, mit dem Schnittpunkt der Straßenmitten als Kreismittelpunkt. Jede Straße bekommt einen Winkel zugeordnet, je nachdem an welcher Stelle sie den Kreis schneidet (vgl. Abb. 4). Im Koordinationsystem entspricht die y-Achse einem Winkel von $0°$ (bzw. andersherum $180°$) und die x-Achse einem Winkel von $90°$ (bzw. andersherum $270°$).

Wie stark sich zwei Straßen nun entgegengesetzt sind, lässt sich aus dem Winkel zwischen ihnen ableiten. Es lassen sich zwischen zwei Straßen immer zwei Winkel messen, je nachdem ob man den Kreis im oder gegen den Uhrzeigersinn begeht. Der Winkel bei kompletter Gegenrichtung beträgt, wie bereits erwähnt, eindeutig $180°$. In jedem anderen Fall nimmt man den jeweils kleineren der beiden Winkel. Kommt man beispielsweise von einer $90°$-Straße und will in eine $290°$-Straße fahren, dann gibt es zwei Winkel:

Im Uhrzeigersinn: $290° - 90° = 200°$

Gegen den Uhrzeigersinn: $90° - 290° = -200° \equiv -200° + 360° = 160°$

In diesem Fall beträgt der Winkel zwischen den Straßen also $160°$. Im folgenden wollen wir für diese Art von Winkel den Begriff Differenzwinkel verwenden. Je größer der Differenzwinkel, desto stärker einander entgegengesetzt sind die Straßen.

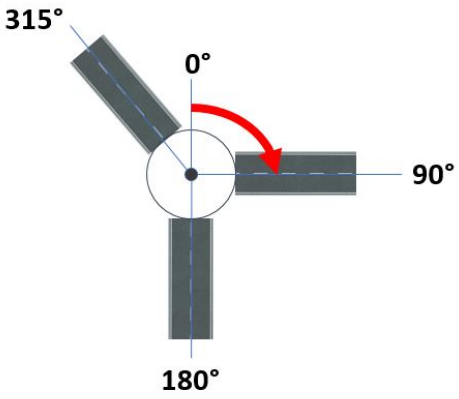


Abbildung 3: Beispiel zur Winkelberechnung von Straßen einer Kreuzung

Methode zum Auffinden von Straßen, die gefährliches Linksabbiegen erfordern

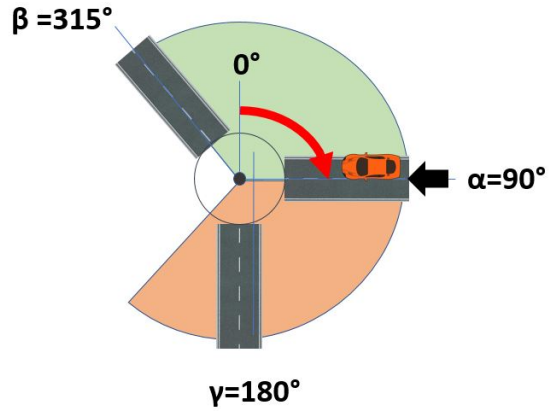
Wir können nun folgende Methode anwenden, um für eine gegebene Kreuzung zu ermitteln, von welcher Straße aus wir in welche Straßen abbiegen dürfen und in welche nicht (also wann es sich um gefährliches Linksabbiegen handelt): Diese Entscheidung ist natürlich immer davon abhängig, aus welcher Straße man gerade kommt. Man bestimmt zuerst die Winkel aller Straßen und dann die Differenzwinkel zwischen der Straße, aus der man kommt, und allen anderen Straßen. Der Gegenverkehr kommt definitionsgemäß dann aus derjenigen Straße, die den größten Differenzwinkel aufweist (maximal $180°$).

Wenn wir die Gegenverkehrsstraße kennen, müssen wir für alle anderen Straßen jetzt nur noch prüfen, ob sie aus Sicht des Fahrers links oder rechts davon liegen. Das ist grafisch für das menschliche Auge sehr leicht, mit Winkelberechnungen wird jedoch eine Fallunterscheidung notwendig. Sei a die Straße, aus der das Auto kommt, b die Gegenverkehrsstraße und c eine zu untersuchende Straße. Die Winkel von a, b und c seien α, β und γ. Dann gilt:

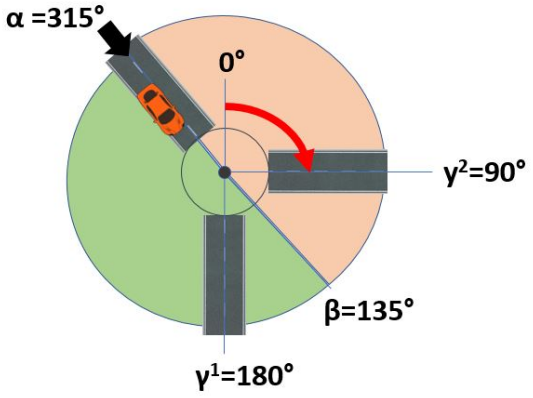
Falls $\beta > \alpha$: Wenn $\beta > \gamma \geq \alpha$, dann liegt c links von b, darf also nicht von a aus befahren werden. Gilt stattdessen $\gamma >= \beta$ oder $\alpha > \gamma$, dann liegt c rechts von b und darf von a aus befahren werden (vgl. Abbildung 4a).

Falls $\alpha > \beta$: Wenn $\alpha > \gamma >= \beta$, dann liegt c rechts von b und darf von a aus befahren werden. Gilt stattdessen $\gamma \geq \alpha$ oder $\beta > \gamma$, dann liegt c links von b und darf nicht von a aus befahren werden (vgl. Abbildung 4b).

Falls $\alpha = \beta$: In diesem Fall ist der Differenzwinkel null, was nur im Fall $d = 1$ auftreten kann. Damit gibt es keinen Gegenverkehr und auch kein Abbiegeverbot. Dieser Sonderfall kann auch einfach in einen der obigen Fälle integriert werden.



(a) Fall $\beta > \alpha$: Abbiegen in den roten Bereich verboten, in den grünen Bereich erlaubt. Im weißen Bereich kann es keine Straßen geben, da sonst die Gegenverkehrsstraße nicht β wäre.



(b) Fall $\alpha > \beta$: Abbiegen in den roten Bereich verboten, in den grünen Bereich erlaubt. Hier liegt der Sonderfall vor, bei dem der Gegenverkehr aus einer virtuellen Richtung ($\beta = 135°$) kommt.

Abbildung 4: Verbotenes Abbiegen

Es gibt einen Sonderfall, der berücksichtigt werden muss. Wenn es nämlich zwei Straßen gibt, die den gleichen Differenzwinkel zur Herkunftsstraße haben, dann kann der Gegenverkehr nicht einer Straße zugeordnet werden (vgl. Abbildungen 4b und 5). In diesem Fall ist es am sinnvollsten, von einem virtuellen Gegenverkehr genau aus der der Herkunftsstraße gegenüberliegenden Richtung (180°) auszugehen, weil diese genau in der Mitte zwischen den beiden realen Straßen liegt. Alles links vom virtuellen Gegenverkehr ist für das Abbiegen verboten, alles rechts davon erlaubt. Diese Methode liefert immer ein eindeutiges Ergebnis, kann in jedem Fall unabhängig von Winkeln und der Anzahl an Straßen (auch d=1, d=2) eingesetzt werden und basiert auf der Problematik von Linksabbiege-Unfällen. Es sind auch noch andere, leicht abgewandelte Definitionen möglich. Wichtig ist hierbei jedoch, dass die genannten Kriterien erfüllt sind und die Definition abstrakt und einheitlich formuliert werden kann, also ohne eine große Zahl an Sonderfällen.

Abbildung 5: Die T-Straße ist ein Beispiel für den Sonderfall. Hier sollte man nur in die rechte Straße abbiegen dürfen.

2.2 Wegfindung

Umwandlung in allgemeines Wegfindungsproblem

Den kürzesten Weg in einem Straßennetz von einer Startkreuzung zu einer Zielkreuzung zu finden, ist ein gut untersuchtes Standardproblem, das graphentheoretisch formuliert werden kann. Wir wollen unser besonderes Wegfindungsproblem mit Abbiege-Restriktion nun als allgemeines Problem formulieren, damit wir es in einem gewöhnlichen Graphen darstellen können. Neben der gesteigerten Einfachheit hat dies den Vorteil, dass sich für das Standardproblem bereits viele kluge Köpfe sehr gute Algorithmen überlegt haben, auf die wir dann zurückgreifen können. (Es reicht schon, neue Verkehrsregeln zu erfinden, da muss man ja nicht gleich auch noch das Rad neu erfinden.)

Um unser Problem zu standardisieren, müssen wir den Graphen unseres Straßennetzes so erweitern, dass die Linksabbiege-Restriktion schon hinreichend durch die vorhandenen oder nicht vorhanden Kanten umgesetzt wird. Mit anderen Worten: *Wo man nicht abbiegen darf, darf im standardisierten Graphen auch keine Kante sein.*

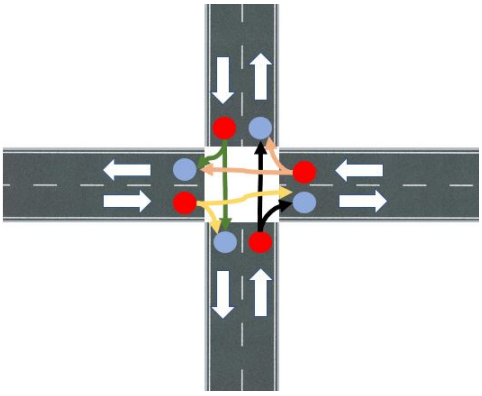
Nun ist unsere Abbiege-Definition natürlich davon abhängig, aus welcher Richtung man gerade kommt. Deshalb können wir nicht einfach Kanten aus dem vorliegenden Straßennetz entfernen – das würde das Netz nämlich ungültig machen, weil auch Abbiegevorgänge, die eigentlich möglich sind, verhindert würden.

Detailhafte Darstellung des Straßennetzes

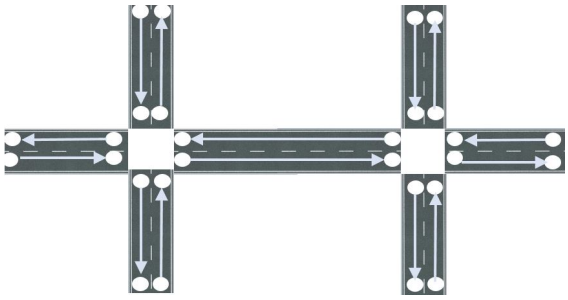
Wir wenden stattdessen einen Trick an: wir blähen das Netz auf, indem wir in die Kreuzungen hineinzoomen: Es gibt jetzt nicht mehr nur *einen* Knoten pro Kreuzung, sondern pro Kreuzung für jede einmündende Straße zwei Knoten (rechte und linke Seite, aus Sicht eines in Richtung der Kreuzung fahrenden Autos). Eine Kreuzung mit $d = 4$ hat somit z.B. 8 Knoten (vgl. Abb. 6a). Wenn ich mich gerade auf einem Knoten der rechten Straßenseite einer Straße befinde, dann bin ich kurz davor, in die zugehörige Kreuzung hineinzufahren. Wenn ich mich stattdessen auf einem Knoten der linken Seite befinde, dann komme ich gerade aus der Kreuzung heraus.

Zwischen diesen Knoten können wir nun passende Kanten zeichnen. Eine Kante steht für einen möglichen Teilpfad im Straßennetz.

- Zwischen Knoten, die sich auf der gleichen Seite von zwei verschiedenen Straßen derselben Kreuzung befinden, darf es grundsätzlich keine Kanten geben. Denn: Wer auf der rechten Seite in eine Kreuzung hineinfährt, darf nicht in die (von der anderen Richtung aus gesehen!) rechte Seite der Gegenstraße einbiegen – sonst wäre er auf der falschen Straßenseite! Gleiches gilt für die linke Seite. Wir möchten schließlich keine Geisterfahrer in Rechthausen haben.
- Von jedem rechtsseitigen (eingehenden) Knoten aus bestimmen wir mit unserer Methode aus Abschnitt 1 alle Straßen, in die man von dort Abbiegen darf und zeichnen Kanten zu den linksseitigen (ausgehenden) Knoten dieser Straßen.
- Schließlich verbinden wir die Kreuzungen miteinander, indem wir alle Knoten einer Straße zwischen zwei Kreuzungen miteinander verbinden.



(a) Eine Kreuzung mit $d = 4$. Rote Punkte stehen für inward-Knoten, blaue Punkte für outward-Knoten. Die Pfeile repräsentieren Kanten, die für gültige Abbiegevorgänge stehen.



(b) Verbindungen zwischen zwei Kreuzungen. Die Pfeile stehen für Kanten. Die Verbindungen innerhalb der Kreuzungen wurden zur Übersichtlichtkeit ausgelassen.

Abbildung 6: Details des Straßennetz-Graphen

Die Fahrt von D über E nach H im Beispiel der Aufgabenstellung läuft nun so ab:

1. Wir befinden uns irgendwo an der Kreuzung D und starten am aus D *hinausführenden* Knoten der Straße zwischen D und E.
2. Wir fahren von dem aus D hinausführenden Knoten die Straße DE entlang bis zum in E *hineinführenden* Knoten.
3. Wir fahren von dem in E hineinführenden Knoten durch die Kreuzung zum aus E *hinausführenden* Knoten der Straße EH.
4. Wir fahren von dem aus E hinausführenden Knoten der Straße EH zu dem in H *hineinführenden* Knoten.
5. Wir sind am Ziel.

Damit wir wie in Schritt 1 *irgendwo an der Kreuzung D* starten können, fügen wir "in der Mitte" jeder Kreuzung noch einen künstlichen Knoten ein, den wir mit allen ausgehenden Knoten der Straßen von D verbinden. Ansonsten müssten wir nämlich schon beim Start festlegen, von welcher Straße aus wir beginnen und wenn wir eine ungünstige erwischen, erstmal herumrangieren. Die künstlichen Knoten in der Kreuzungsmitte könnte man also als *Einstiegspunkte* in den Graphen betrachten.

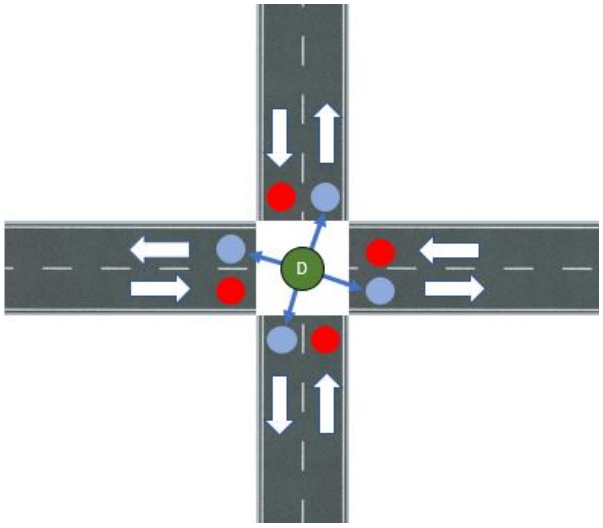


Abbildung 7: Kreuzung mit Einstiegspunkt

Kantengewichte Da wir das Durchfahren von Kreuzungen bei der Distanzberechnung nicht berücksichtigen, erhalten alle Kanten zwischen den Knoten einer Kreuzung ein Gewicht von Null. Die Kanten, die die wirklichen Straßen darstellen erhalten als Gewicht das passende Distanzmaß: Wenn wir nur die *Anzahl* an befahrenen Straßen eines Pfades zählen, bekommen die Kanten ein Gewicht von Eins. Ansonsten verwenden wir als Gewicht die euklidische Distanz zwischen den beiden Kreuzungen a und b, welche die Straße verbindet:

$$d(a,b) = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$$

Beurteilung der Knoten- und Kantenanzahl

Wie sehr haben wir den Graphen nun aufgebläht? Diese Frage ist für spätere Laufzeitbetrachtungen relevant.

- Für jede Kreuzung benötigen wir den künstlichen Mittelknoten plus jeweils zwei Knoten pro angrenzende Straße. Da eine Straße immer an zwei Knoten angrenzt, brauchen wir 4 Knoten pro Straße. Insgesamt hat unser neuer Graph also Anzahl Kreuzungen $+\,4 \cdot$ Anzahl Straßen Knoten. Nehmen wir den unrealistischen Fall an, dass jede Kreuzung direkt mit jeder anderen Kreuzung verbunden ist (vollständiger Graph), dann gibt es für n Kreuzungen $\frac{n(n-1)}{2}$ Straßen und somit maximal $n + 4 * \frac{n(n-1)}{2} = n * (2n-1)$ Knoten.

- Auf jede Straße kommen zwei Kanten (Hin- und Rückweg). Wie viele Kanten es pro Kreuzung gibt, hängt davon ab, wie viele Abbiegepfade durch die Restriktion verboten wurden und wie viele Straßen pro Kreuzung aufeinander treffen. Eine allgemeine Aussage lässt sich deshalb nicht treffen. Wir können aber eine Obergrenze festlegen: Nehmen wir wieder den unrealistischen Fall mit $\frac{n(n-1)}{2}$ Straßen an. Jede Kreuzung hat eine maximale Anzahl an Straßen von $n-1$. Lassen wir jetzt noch das Linksabbiegeverbot außer Acht, dann gibt es pro Kreuzung $(n-1)^2$ Kanten, weil man von jeder Straße in jede andere und in die Straße selbst (U-Turn) fahren kann. Insgesamt haben wir in einem Netz mit n Kreuzungen und $\frac{n(n-1)}{2}$ Straßen also maximal $\frac{n(n-1)}{2} * 2 + n * (n-1)^2$ Kanten, was einem Faktor von 2n entspricht.

Insgesamt haben wir im Extremfall bei einem vollständigen Straßennetz mit n Kreuzungen $(2n-1)$ mal so viele Knoten wie Kreuzungen und $2n$ mal so viele Kanten wie Straßen.

Implementierung

Unseren Graphen können wir mithilfe von Matrizen (Adjazenz- oder Inzidenzmatrix) oder mithilfe von Zeigerstrukturen, bei der für jeden Knoten alle seine Nachbarn vermerkt werden (Adjazenzliste), implementieren. Letzteres verbraucht insbesondere weniger Platz, da der Straßennetzgraph relative wenige Kanten hat (also *licht* ist).

2.3 Anwendung von Wegfindungsalgorithmen

Nun können wir unser Problem nach Umwandlung des Straßennetzes in einen normalen Graphen als sogenanntes *single-pair shortest-path*-Problem formulieren: *Finde in einem gegebenen Graphen G ohne negative Kantengewichte einen möglichst kurzen Pfad zwischen zwei Knoten s und t.*

Algorithmus von Dijkstra

Ein sehr bekannter Algorithmus zur Lösung dieser Aufgabe ist der **Dijkstra-Algorithmus**, benannt nach seinem Erfinder Edsger W. Dijkstra. Dabei werden solange sukzessive vom Startknoten ausgehend die kürzesten Wegstrecken zu allen anderen erreichbaren Knoten berechnet, bis ein Pfad zum Zielknoten gefunden wird, der dann automatisch der kürzeste ist.

Für jeden Knoten wird der zu ihm bisher kürzeste bekannte Weg vom Startknoten aus gespeichert, in dem der jeweilige Vorgänger und die Länge des Weges (Abstand) notiert werden. Anfangs sind noch keine Wege bekannt, bis auf den Weg vom Startknoten zu sich selbst, der den Abstand null hat. Kein Knoten wurde bisher besucht. Nun beginnen wir mit dem Startknoten. Für jeden Knoten, den man vom Startknoten aus direkt erreichen kann, tragen wir den zugehörigen Abstand und als Vorgänger den Startknoten ein; danach ist der Startknoten abgeschlossen. Als nächstes besuchen wir unter den verbleibenden unbesuchten Knoten denjenigen mit dem geringstem Abstand (nennen wir ihn u). Für jeden seiner Nachbarn v untersuchen wir, ob es im Vergleich mit dem aktuellen Abstand von v eine Abkürzung wäre, vom Startknoten

Algorithmus 2 Dijkstra-Algorithmus

```
 1: procedure DIJKSTRA(GRAPH, STARTKNOTEN, ZIELKNOTEN)
 2:
 3:     abstand[ ], vorgaenger[ ]
 4:
 5:     Für jeden Knoten v in GRAPH:
 6:         abstand[v] ← ∞
 7:         vorgaenger[v] ← null
 8:
 9:     abstand[STARTKNOTEN] ← 0
10:
11:     ▷ Liste aller Knoten, die noch nicht besucht wurden:
12:     ListeQ ← alle Knoten von GRAPH
13:
14:     ▷ Algorithmus
15:     Solange Q nicht leer:
16:         u ← Knoten in Q mit geringstem Wert in abstand[ ]
17:         entferne u aus Q
18:         Wenn u = ZIELKNOTEN: break
19:         Für jeden Nachbarn v von u:
20:             Wenn v ∈ Q:
21:     ▷ Nach kürzerem Weg suchen (vom STARTKNOTEN über u nach v)
22:     ▷ unter Verwendung des Kantengewichts (u,v):
23:                 alternativWeg ← abstand[u] + abstand_zwischen(u,v)
24:                 Wenn alternativWeg < abstand[v]:        ▷ kürzeren Weg gefunden
25:                     abstand[v] ← alternativWeg
26:                     vorgaenger[v] ← u
27: end procedure
```

nach u zu gehen und dann von u nach v. Haben wir damit tatsächlich eine Abkürzung gefunden, dann aktualisieren wir den Abstand und legen u als Vorgänger von v fest. Damit ist u abgeschlossen, und wir machen mit dem nächsten noch unbesuchten Knoten weiter.

Dadurch, dass immer der Knoten mit dem geringstem Abstand als nächstes besucht wird (diese Strategie nennt man *greedy*), sind die Pfade der bereits besuchten Knoten zu jedem Zeitpunkt schon optimal. Das funktioniert, weil die kürzesten Teilstrecken zwischen Knoten in einem Pfad zusammen auch die kürzeste Strecke auf diesem Pfad bilden. Wenn wir irgendwann den Zielknoten besucht haben, dann können wir sicher sein, schon einen kürzesten Pfad zu ihm gefunden zu haben, und können aufhören. Um nun den kürzesten Pfad zu erhalten, müssen wir lediglich vom Zielknoten aus über die gespeicherten Vorgänger rückwärts iterieren, bis wir beim Startknoten angekommen sind.

Algorithmus 2 beschreibt das Vorgehen in Pseudocode. Bei der einfachen Implementierungsform des Algorithmus (Verwendung von Listen oder Arrays) ergibt sich eine Zeitkomplexität von $\mathbf{O}(|V|^2)$, wobei V die Anzahl an Knoten meint. Unser Graph hat, wie untersucht, bei n Kreuzungen maximal $n(2n-1)$ Knoten, deshalb benötigen wir $\mathbf{O}((n(2n-1))^2) = \mathbf{O}(n^4)$ Zeit. Aufgrund der polynomiellen Laufzeit dieser Methode könnte man das gefundene Vorgehen als

ausreichend effizient bezeichnen. Bei der Implementierung des Algorithmus von Dijkstra ist durch die Verwendung einer Prioritätswarteschlange eine Optimierung möglich.

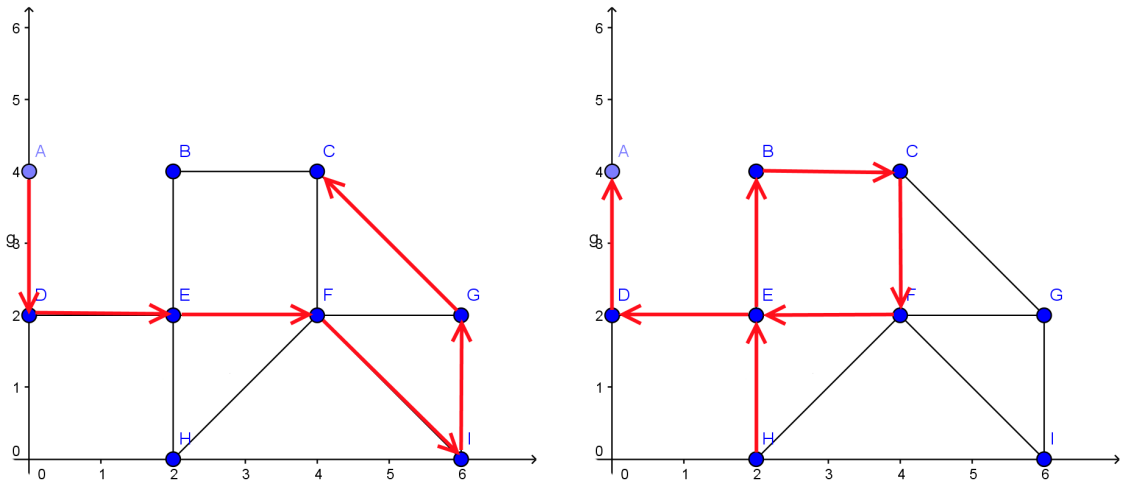
Weitere Algorithmen

Eine effiziente Erweiterung des Dijkstra-Algorithmus ist der A*-Algorithmus. Hier besucht man immer diejenigen Knoten zuerst, die *wahrscheinlich* schnell zum Ziel führen. Um den vielversprechendsten Knoten zu ermitteln, wird eine Heuristik benutzt, die angibt, wie lang der Pfad vom Start zum Ziel unter Verwendung des betrachteten Knotens im *günstigsten* Fall ist. Der Knoten mit dem kleinsten Wert wird als nächster untersucht. In unserem Fall ist die Luftlinie (euklidische Distanz zum Zielknoten) eine geeignete Schätzung, denn die tatsächliche Strecke ist nie kürzer als die direkte Verbindung. Mit dem A*-Algorithmus lässt sich keine geringere Worst-Case-Laufzeit als beim Dijkstra-Algorithmus erreichen, aber in vielen Fällen ist A* schneller als Dijkstra.

Außerdem könnte man den Bellman-Ford-Algorithmus verwenden, der in mehreren Phasen über alle Kanten des Graphen interiert und nach kürzesten Verbindungen zwischen dem Startknoten und anderen Knoten des Graphen sucht. Auch dieser Algorithmus ist optimal, benötigt jedoch im Allgemeinen mehr Zeit als Dijkstra oder A*.

Ergebnisse

Da wir einen Graphen konstruiert haben, der das Straßennetz samt Linksabbiegeverbot korrekt repräsentiert, liefert uns ein Algorithmus mit dem kürzesten Pfad in unserem Graphen auch die kürzeste erlaubte Route im Straßennetz. Um die Route wieder einfacher darstellen zu können, fassen wir die Knoten, die beim Durchfahren von Kreuzungen besucht werden, zu einer Station zusammen. So wird aus D-E-E-H dann D-E-H. Die Distanz des Pfades im Graphen entspricht auch der Weglänge im Straßennetz, weil wir diese ja bei den Kantengewichten berücksichtigt haben.



(a) Route von A nach C (b) Route von H nach A

Abbildung 8: Routen ohne Linksabbiegen im Beispiel der Aufgabenstellung

Im Straßennetz aus der Aufgabenstellung ist ein kürzester Weg ohne Linksabbiegen von A nach C die Route A-D-E-F-I-G-C (mit einer Länge l von zirka 13,66) und ein kürzester Weg von H nach A die Route H-E-B-C-F-E-D-A ($l = 14,0$); vgl. Abbildung 8. In beiden Fällen zeigt sich durch das Linksabbiegeverbot ein deutlicher Umweg; auf der Strecke von H nach A wird sogar eine Schleife gefahren.

2.4 Erreichbarkeit im Straßennetz

Nun möchten wir wissen, ob jede Kreuzung von jeder anderen Kreuzung aus ohne Linksabbiegen erreicht werden kann.

Single-Source und All-Pair Shortest Path

Im oben genannten Verfahren endet der Dijkstra-Algorithmus, wenn der Zielknoten besucht wurde. In der allgemeineren Version wird jedoch weitergesucht, bis *alle* Knoten besucht wurden. Anschließend sind die kürzesten Pfade vom Startknoten zu *allen* Knoten des Graphen bekannt. Diese Variante heißt *single-source shortest path* und ist für uns sehr hilfreich. Da wir den Abstand von Knoten zum Startknoten anfänglich mit unendlich initialisieren und den Vorgänger auf null setzen, wird nach Ablauf des Algorithmus die Distanz derjenigen Knoten unendlich betragen, die vom Startknoten aus unerreichbar sind. Somit können wir Dijkstra benutzen, um herauszufinden, ob von einer Kreuzung aus alle anderen Kreuzungen erreicht werden können, oder nicht.

Wir führen nun für jede Kreuzung unseren Algorithmus in der abgewandelten Form durch und erhalten die kürzesten Wege von jeder Kreuzung zu jeder anderen Kreuzung. Diese Konstellation nennt man *all-pair shortest path* (APSP). Wenn wir schon die kürzesten Wege zwischen allen Kreuzungen kennen, dann können wir auch mit Leichtigkeit sagen, ob jede Kreuzung von jeder anderen Kreuzung aus erreichbar ist – nämlich genau dann, wenn es keine Pfade mit der Länge unendlich gibt.

Alternatives Vorgehen

Anstatt den Algorithmus von Dijkstra für jede Kreuzung anzuwenden, können auch andere Graphenalgorithmen, die das APSP-Poblem lösen, benutzt werden. Als Beispiel sei hier der Algorithmus von Floyd und Warshall genannt, der meistens mit einer Adjazenzmatrix verwendet wird und dynamische Programmierung benutzt. Der Algorithmus findet sukzessive die kürzesten Teilpfade, die über Knoten mit Index kleiner als k (k wächst schrittweise) führen und errechnet dann alle kürzesten Wege über Knoten mit Index höchstens k. Dies wird solange wiederholt, bis alle Knoten berücksichtigt wurden.

Ergebnisse

Im Straßennetz aus der Aufgabenstellung ist jede Kreuzung von jeder anderen Kreuzung aus erreichbar. Es lässt sich auch argumentieren, dass, wenn ein Straßennetz zusammenhängend ist und jede Kreuzung von jeder Kreuzung aus (ohne Verbot) erreichbar ist, ein Linksabbiegeverbot zwar die Strecke im Allgemeinen beliebig verlängern kann, aber trotzdem immer noch jede Kreuzung von jeder Kreuzung aus erreichbar sein wird. Das Verbot verhindert also keine Verbindungen (falls Wenden in Sackgassen möglich ist).

Der Grund dafür ist, dass man auch ohne Linksabbiegen immer in einen Straßenabschnitt hinein und dann wieder aus der anderen Richtung hinaus fahren kann. Entweder erreicht man im Straßenabschnitt nämlich irgendwann eine Sackgasse, in der man ja wenden darf, oder man gelangt wieder zu einer Straße zurück, die man schon befahren hat, und schließt so einen Kreis. Und ein Kreis kann immer in mindestens eine Richtung befahren werden. Wenn wir jetzt, um von A nach B zu gelangen, in eine Straße abbiegen möchten, es aber wegen des Verbots nicht dürfen, so können wir in die Gegenverkehrsstraße einfahren und (nach unbestimmter Zahl an Schritten) aus der anderen Richtung wieder zur Kreuzung zurückkommen. Von der Gegenverkehrsstraße aus gesehen ist aber die vorher links gewesene Straße nun rechts, und wir dürfen in sie einbiegen. Diese Argumentation lässt sich induktiv von allen Teilabschnitten auf das gesamte Netz übertragen.

2.5 Finden der größten Wegverschlechterung durch das Linksabbiegeverbot

Wenn wir nun noch herausfinden wollen, welche Strecke zwischen zwei Kreuzungen in Rechthausen durch das Verbot am stärksten betroffen ist (um beispielsweise sagen zu können, welcher brave Bürger am meisten leiden wird, weil er nun deutlich länger von seinem Haus zu seinem Arbeitsort fahren muss), dann müssen wir die kürzesten Pfade im Netz vor und nach Verbot vergleichen.

Dazu konstruieren wir zwei Graphen. Beim ersten Graphen wenden wir wie gehabt das Linksabbiegeverbot nach unserer Methode aus Abschnitt 2.1 an, beim zweiten Graphen lassen wir es jedoch außer Acht. Anschließend lösen wir mithilfe der oben genannten Erweiterung des Dijkstra-Algorithmus oder mit dem Algorithmus von Floyd und Warshall das APSP-Problem für beide Graphen. Dann kennen wir die kürzesten Pfade zwischen allen Kreuzungen mit und ohne Verbot. Nun müssen wir noch für jedes Paar von Kreuzungen die gefundenen Weglängen vergleichen und ermitteln, um welchen Faktor sich die Route verlängert hat. Das Paar, bei dem sich der Pfad um den maximalen Faktor verlängert hat, ist das gesuchte. Um den Sachverhalt sowohl bzgl. Anzahl an Straßen als auch Länge des Weges zu beleuchten, führen wir die genannten Schritte einmal mit Kantengewichten vom Wert 1 und einmal mit Kantengewichten, die der euklidischen Distanz entsprechen, durch.

Ergebnisse

Beispiel aus der Aufgabenstellung Im Straßennetz aus der Aufgabenstellung verlängert sich die Route **H-D** durch das Linksabbiegeverbot am meisten. Ohne Restriktion ist der

Pfad H-E-D ($l = 4,0$) möglich, mit Restriktion jedoch nur H-E-B-C-F-E-D ($l = 12,0$). Das ist ein Faktor von 3,0.

Beispiele aus der EI-Community

Hier einige Beispiele, die in der EI-Community geteilt wurden.

Straßennetz von Leon Windheuser

Beispiel 1: 5 Kreuzungen und 4 Straßen. Die Route von E nach C verlängert sich durch das Linksabbiegeverbot am meisten (Faktor 3,0).

Alle Paare von Kreuzungen können (ohne Linksabbiegen) erreicht werden. Route von E nach C ohne Restriktion: E-D-C ($l = 2,0$) und mit Restriktion: E-D-B-A-B-D-C ($l = 6,0$).

Straßennetze von Fabian Michel

Beispiel 1: 26 Kreuzungen und 35 Straßen. Die Route von A nach P verlängert sich durch das Linksabbiegeverbot am meisten (Faktor 6,561208747436233).
Alle Paare von Kreuzungen können (ohne Linksabbiegen) erreicht werden. Route von A nach P ohne Restriktion: A-B-P ($l = 4,0$) und mit Restriktion: A-B-C-J-Y-X-J-C-B-P ($l = 26,24483498974493$).

Beispiel 2: 72 Kreuzungen und 104 Straßen. Die Route von R nach L verlängert sich durch das Linksabbiegeverbot am meisten (Faktor 9.384776310850237).
Alle Paare von Kreuzungen können (ohne Linksabbiegen) erreicht werden.

Beispiel 3: 200 Kreuzungen und 304 Straßen. Die Route von fc nach bh verlängert sich durch das Linksabbiegeverbot am meisten (um den Faktor 9.064495102245981).
Alle Paare von Kreuzungen können (ohne Linksabbiegen) erreicht werden. Route von fc nach bh ohne Restriktion: fc-bi-bh ($l = 4,0$) und mit Restriktion: fc-fb-ai-ah-ar-fi-ab-bd-fh-as-at-bh ($l = 36,257980408983926$)

2.6 Bewertungskriterien

- Es wurde eine „sinnvolle“ allgemeine Definition des Linksabbiegens aufgestellt, die die Anforderungen aus der Aufgabe erfüllt. Die Definition sollte …

  … allgemein, also auf möglichst beliebige Kreuzungen anwendbar sein. Separate „Definitionen“ für verschiedene d-Werte sind nicht sinnvoll.

  … zu eindeutigen Entscheidungen bzgl. des Linksabbiegens führen.

  … die Beispielrouten aus der Aufgabe wie vorgegeben entscheiden.

  … sich auf das Phänomen erhöhter Unfallgefahr beim Linksabbiegen beziehen.

  Eine Definition, die gleich mehrere dieser Bedingungen nicht erfüllt, kann auch als insgesamt ungeeignet bewertet sein.

- Es werden anschauliche Beispiele zur Anwendung der Definition gegeben, auch für besondere Fälle.
- Die Definition wurde in eine Methode umgesetzt, die für jede Situation (aktuelle Straße und Zielstraße) effektiv und korrekt entscheidet, ob das Abbiegen zulässig ist.
- Bei der Bearbeitung der Teile 3 bis 5 liegt, ähnlich wie bei Aufgabe 1, eine Modellierung mit Graphen nahe. Eine entsprechende, geeignete Abstraktion muss gefunden worden sein. Das Linksabbiegeverbot kann bei der Wegsuche durch Modifikation des Graphen oder des Verfahrens berücksichtigt werden. Alle korrekten Vorgehensweisen, die mit beiden Weglängenmaßen funktionieren, sind hier in Ordnung.
- Auch spezifische Aspekte der Teilaufgaben sollten geeignet und korrekt bearbeitet sein. Bei Teilaufgabe 4 genügt auch eine Tiefensuche (pro Knoten) zur Prüfung der Erreichbarkeit.
- Offiziell wurde nur ein Beispiel vorgegeben; dieses muss bearbeitet worden sein, einschließlich der Beantwortung der in Teil 3 explizit gestellten Fragen. Da das Beispiel nicht alle Fälle abdeckt, werden weitere Beispiele erwartet.
- Die Ergebnisse der Straßennetzanalyse bzw. der Algorithmen werden auf sinnvolle Art und Weise dargestellt.
- Das vorgegebene Eingabeformat muss korrekt verarbeitet werden.
- Diese Aufgabe bietet einige Möglichkeiten für weitergehende Analysen. Beispiele: *An wie vielen Stellen wird nach dem Verbot überhaupt ein gefährliches Linksabbiegen vermieden?*, *An welcher Stelle könnten zusätzliche Straßen sinnvoll sein, um Wegverlängerungen zu vermeiden?*, *Wie verändert sich die Frequentierung einzelner Straßen durch das Verbot, wenn man annimmt, dass jede Kreuzung gleich oft von jeder anderen aus besucht wird?*, etc.
- Verschiedene Erweiterungen sind denkbar (evtl. auf Grundlage der o.g. Analysen), z.B. das Einfügen von Straßenabschnitten zur Vermeidung von Wegverlängerungen, die Berücksichtigung von Einbahnstraßen etc.

Aufgabe 3: Kreis-Code

Diese Aufgabe ermöglicht unzählige kreative Herangehensweisen. Zum Beispiel eignen sich die abrupten Übergänge von schwarz auf weiß gut für eine Kantenerkennung. Die großen schwarzen Flächen (und der innere weiße Ring) dagegen legen nahe, zunächst einzelne Objekte im Bild zu identifizieren. Es würde den Umfang dieses Dokumentes sprengen, mehrere solcher Ansätze im Detail anzusprechen. Deshalb werden wir uns hier nur auf eine einzelne exemplarische Lösung beschränken. Teilaufgabe 3 entfällt als eigenständiger Abschnitt, da gleich der allgemeine Fall mit Störeffekten behandelt wird.

3.1 Teilaufgabe 1: Mittelpunkte bestimmen

Lösungsidee

Es wird millionenfach nach dem Muster Dunkel-Hell-Dunkel-Hell gesucht, in allen möglichen Größenordnungen, Rotationen und Positionen. Die Punkte, an denen das Muster besonders oft gefunden wurde, werden zu Mittelpunkten deklariert.

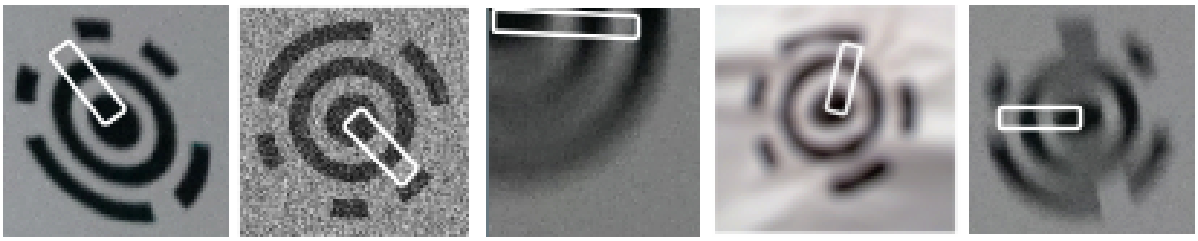


Abbildung 9: Das Suchmuster in Kreiscodes, die verschiedenen Störeffekten ausgesetzt sind.

Umsetzung

Eine Schablone zum Suchen Um das Nachdenken über den Suchvorgang zu vereinfachen, führen wir das Konzept der Schablone ein. Das ist ein Objekt, das man – beliebig rotiert und skaliert – auf einen Ausschnitt des Bildes legen kann, um dann durch einen Test zu erfahren, ob die überdeckten Pixel (hinreichend gut) dem gesuchten Muster entsprechen.

Abbildung 10: Das Muster nach dem gesucht wird, in einer Schablone.

Der Schablonentest Da sehr viele Bildausschnitte auf ihre Ähnlichkeit mit dem Muster hin getestet werden müssen, ist es nützlich, den Test minimalistisch zu halten; zum Beispiel, indem im Schablonenbereich nur 4 Punkte ausgemessen werden. Das ist das Minimum an Punkten, um das gesuchte Muster gerade noch nachzuahmen. Alternativ könnte man auch

etwas mehr Messpunkte benutzten und anders platzieren, um sich stärker auf die Kanten statt auf die Flächen zu konzentrieren.

Abbildung 11: Vier Messpunkte nähern das gesuchte Muster an.

Den Grad der Ähnlichkeit der gemessenen Punkte mit dem Muster könnte durch eine Prozentangabe beziffert werden. Es reicht aber auch, nur eine Ja/Nein-Antwort zurückzugeben. Ein möglicher Test wäre dann: Ist der Kontrast zwischen den „dunklen“ und den „hellen“ Messpunkten groß genug? Für diesen Test hat sich ein Mindestkontrast von 30 als gute Schwelle erwiesen. Das Helligkeitsspektrum erstreckt sich dabei von 0 (schwarz) bis 255 (weiß).

Einzelne Punkte prüfen Hat man den Schablonentest definiert, bietet es sich an, jeweils für einen Punkt durch Rotation der Schablone überprüfen zu lassen, mit welcher Wahrscheinlichkeit er ein Mittelpunkt ist. Einen Punkt, den wir überprüfen, nennen wir ab nun Ankerpunkt (weil die Schablone an ihm „verankert“ ist).

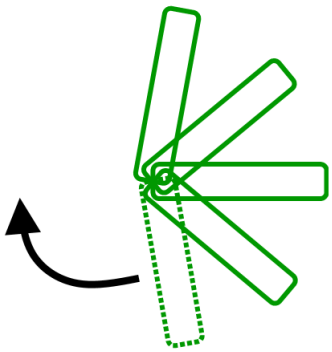


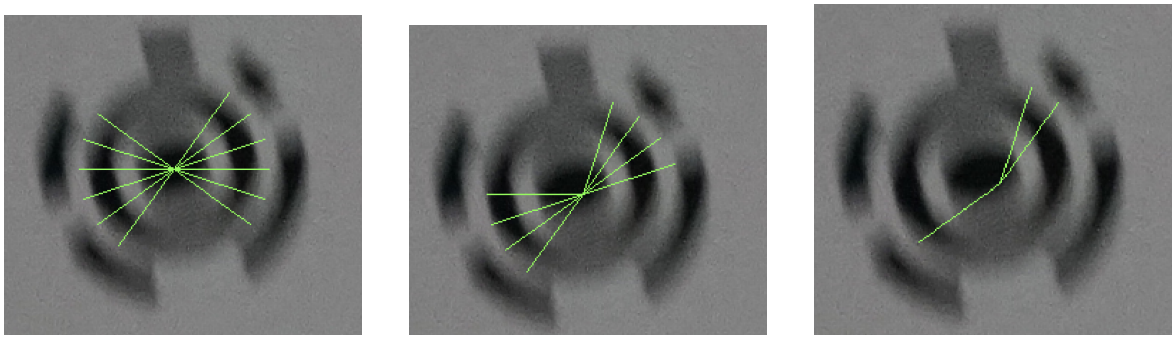
Abbildung 12: Die Trefferquote eines Ankerpunktes wird ermittelt.

Hat man das Testergebnis als Ja/Nein-Rückgabe definiert, ist folgendes Maß naheliegend:

$$\text{Trefferquote} = \frac{\text{Anzahl erfolgreiche Tests}}{\text{Anzahl durchgeführte Tests}}$$

Je höher die Trefferquote, desto wahrscheinlicher handelt es sich beim untersuchten Ankerpunkt um einen Mittelpunkt. Die Trefferquoten werden später die Daten sein, anhand derer Mittelpunkte bestimmt werden.

Länge der Schablone ändern Bei dem Schablonen-Ansatz muss die Länge der Schablone verändert werden, um verschiedene Kreiscodegrößen zu erkennen. Damit alle Größenordnungen identisch behandelt werden (Skaleninvarianz) ist es empfehlenswert, die Längenänderung exponentiell zu gestalten, sodass z.B. jede neue Schablone 15% länger ist als die



(a) Trefferquote $= 12/20 = 60\%$ (b) Trefferquote $= 8/20 = 40\%$ (c) Trefferquote $= 3/20 = 15\%$

Abbildung 13: Es wurden jeweils 20 Tests durchgeführt. Die grünen Linien markieren jeweils die erfolgreichen Tests.

voherige. Folgendes Vorgehen ist möglich: Es wird eine kleinste Länge (*min*) und ein Skalierungsfaktor (k) festgelegt. Mit diesen beiden Werten können dann mit steigendem Exponenten n beliebig viele, immer größer werdende Schablonenlängen $L(n)$ erzeugt werden:

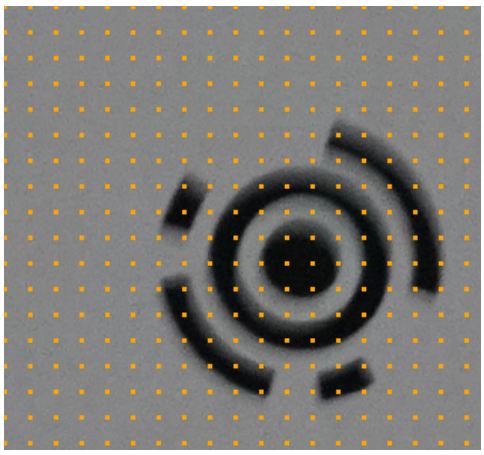
$$L(n) = min \cdot k^n$$

Als kleinste Länge ist 12 Pixel ausreichend klein, als Skalierungsfaktor ist 1,15 ein guter Kompromiss. Als größte Länge kann z.B. der kleinere Wert aus Bildbreite und Bildhöhe gewählt werden.

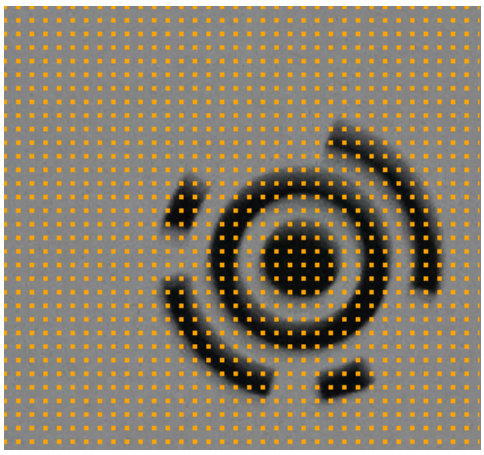
Ankerpunkte auswählen Es ist zu rechenaufwändig, jedes einzelne Pixel als Ankerpunkt zu betrachten. Eine erste filternde Auswahl könnte deshalb zum Beispiel durch ein Suchraster getroffen werden. Dadurch werden je nach Schablonenlänge 30% bis 99% aller Pixel übersprungen. Damit die Suchgenauigkeit nicht leidet, muss jedoch der Abstand der Ankerpunkte untereinander weiterhin klein genug sein, dass keine Kreiscodes übersprungen werden. Ein guter Kompromiss für den Abstand ist beispielsweise ein Zehntel der aktuellen Schablonenlänge (vgl. Abbildung 14).

Nach Trefferquoten filtern Nachdem die Suche beendet ist, existiert ein großer Datenhaufen, in unserem Fall Trefferquoten. Nicht alle dieser Daten sind nützlich, manche stören sogar, weil sie auf Mittelpunkte an den falschen Orten hinweisen. Diese störenden Ankerpunkte können z.B. durch einen Filter entfernt werden, der alle Ankerpunkte aussortiert, deren Trefferquote zu niedrig ist. Zu streng darf dieser Filter jedoch auch nicht sein, sonst werden Kreiscodes übersehen (vgl. Abbildung 15).

Mittelpunkte auswählen Es könnten einzelne Ankerpunkte als Mittelpunkte deklariert werden, die lokal die höchste Trefferquote haben. Meistens ist es aber robuster, die Informationenen mehrerer Ankerpunkte zu akkumulieren. Eine mögliche Vorgehensweise: Es werden gleichfarbige (d.h.: mit der gleichen Schablonenlänge getestete) Ankerpunkte zu Clustern zusammengefasst. Bei diesen Clustern lässt sich dann ein Schwerpunkt berechnen, wobei Ankerpunkte mit höherer Trefferquote stärker gewichtet werden könnten. Falls mehrere Cluster eng

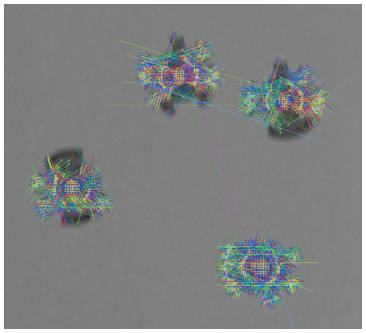


(a) 1/5 Schablonenlänge: Zu großer Abstand zwischen den Ankerpunkten. Der Mittelpunkt wird stark verfehlt.

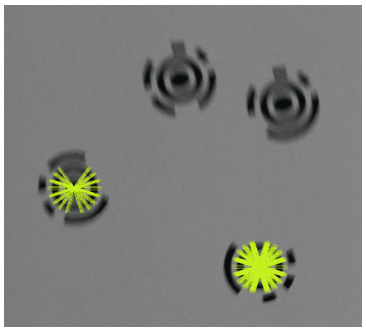


(b) 1/10 Schablonenlänge: Ausreichend kleiner Abstand zwischen den Ankerpunkten. Der Mittelpunkt wird mindestens einmal sehr gut getroffen.

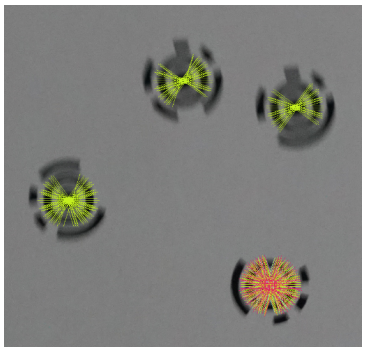
Abbildung 14: Verschiedene Abstände zwischen den Ankerpunkten



(a) Filter: Trefferquote $\geq$ 10%. Das ist nicht streng genug, es gibt zu viele störende, aus Versehen entstandene Treffer.



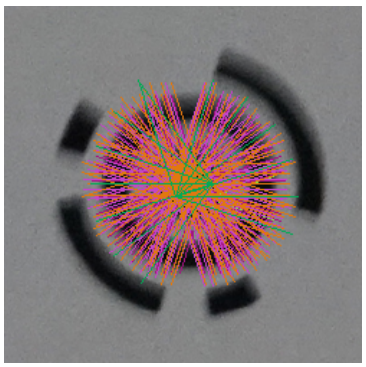
(b) Filter: Trefferquote $\geq$ 70%. Das ist zu streng, die beiden unscharfen Kreiscodes werden nicht mehr erkannt.



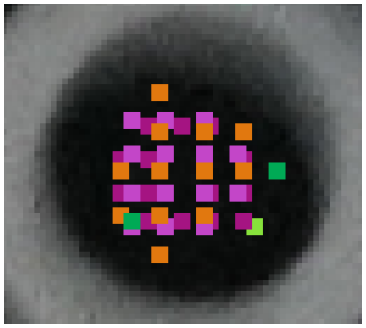
(c) Filter: Trefferquote $\geq$ 35%. Ein guter Kompromiss

Abbildung 15: Filterung nach Trefferquoten

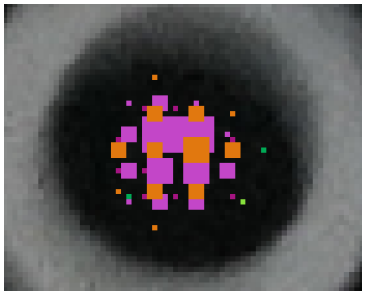
beieinander liegen, kann z.B. der stärkste unter ihnen ausgewählt und dessen Schwerpunkt als Mittelpunkt deklariert werden. Ein mögliches Maß für die Stärke eine Clusters ist die Summe der Trefferquoten der zugehörigen Ankerpunkte. Wenn bis zu dieser Stelle alles wie geplant ablief, existiert nun für jeden Kreiscode genau ein Cluster.



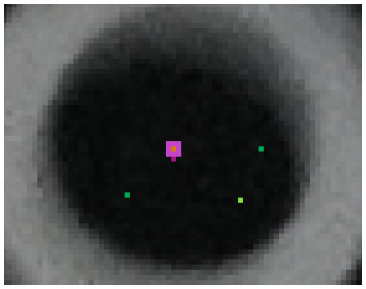
(a) Ankerpunkte (und Treffer) ab einer Trefferquote von 35%. Längen sind durch Farbe unterscheidbar.



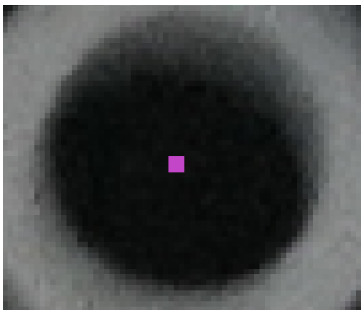
(b) Die Ankerpunkte in Nahaufname. Bei den orangenen z.B. ist das Suchraster noch gut zu erkennen.



(c) Die Ankerpunkte skaliert nach ihrer Trefferquote. Wie gehofft: In der Mitte die höchsten Werte.



(d) Es wurden Cluster gebildet, die auf ihrem Schwerpunkt eingezeichnet sind. Die beiden dunkelgrünen Ankerpunkte existieren weiterhin einzeln, da sie zu weit voneinander entfernt liegen, um ein gemeinsames Cluster zu bilden.



(e) Das pinke Cluster hatte die größte Summe an Trefferquoten. Deshalb hat es Schritt für Schritt alle benachbarten Cluster gelöscht. Jetzt, wo es alleine steht, wird seine Position als Mittelpunkt deklariert.

Abbildung 16: Auswahl des Mittelpunkts durch Clustering

3.2 Teilaufgabe 2: Bedeutungen decodieren

Lösungsidee

Mit Wissen der Cluster-Schablonenlänge (Radius) und durch zusätzliche Ellipsennäherungen wird die Form des äußersten Rings ermittelt. Der ausgelesene äußerste Ring wird dann in 16 Blöcke aufgeteilt, die abschließend in eine dunkle und eine helle Gruppe aufgeteilt werden.

Umsetzung

Barcode erzeugen Um die Segmente unabhängig vom eigentlichen Kreiscode zu analysieren, ist es praktisch, den äußersten Ring in eine Art Barcode umzuwandeln. Solch ein Barcode lässt sich einfacher weiterverarbeiten und abstrahiert von irrelevanten Informationen wie Größe oder Position des Kreiscodes.

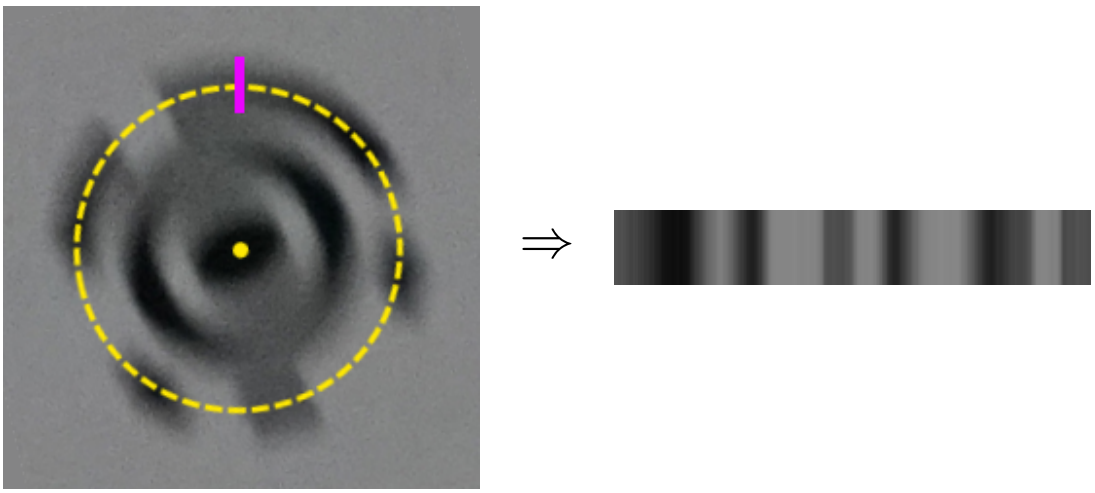


Abbildung 17: Von Kreiscode zu Barcode. Dazu werden, bei 12 Uhr beginnend, einzelne Pixel entlang des äußersten Rings reihum ausgelesen. Zur besseren Ansicht wurde der Barcode in der Höhe gestreckt.

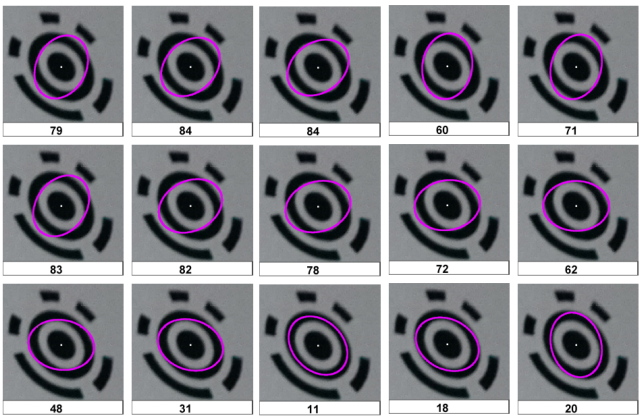
Im einfachsten Fall ist der Kreiscode ein Kreis geblieben. Dann lässt sich der Radius des äußersten Rings sofort berechnen aus der Schablonenlänge des zugehörigen Clusters. In konstanten Winkelabständen können dann Helligkeitswerte entlang des Rings gemessen werden (vgl. Abbildung 17).

Verzerrungen beachten Im Allgemeinen darf nicht davon ausgegangen werden, dass die Kreiscodes im Bild perfekt kreisförmig erscheinen. Stattdessen sind sie oft etwas verzerrt, z.B. weil sie angewinkelt fotografiert wurden oder weil das Papier an ihrer Stelle zerknittert ist. Zumindest die Verzerrung durch Anwinkeln sollte behandelt werden, da sie in den Beispielbildern häufiger auftaucht. Eine gute Näherung ist, die verzerrten Kreiscodes als Ellipsen aufzufassen. Kennt man dann je Kreiscode dessen genaue Ellipsen-Parameter, dann lassen sich die Verzerrungseffekte aufheben.

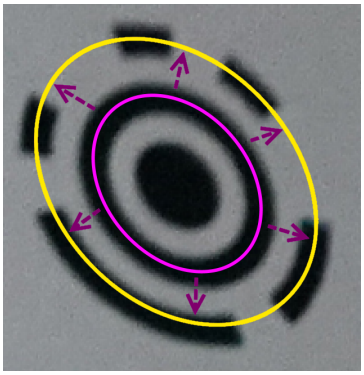
Als Referenz am Kreiscode kann z.B. der mittlere schwarze Ring verwendet werden, der möglichst exakt mit einer Ellipse nachgeahmt werden soll. Es können unterschiedlich geformte und gedrehte Ellipsen ausprobiert und die am genauesten passende ausgewählt werden. Als Maß der Annäherung eignet sich z.B. die durchschnittliche Helligkeit der bedeckten Pixel. Je kleiner dieser Wert, desto dunkler sind die Pixel, auf denen die aktuelle Ellipse liegt. Und je dunkler die Pixel, desto wahrscheinlicher befindet sich die Ellipse im Zentrum des mittleren schwarzen Rings.

Anhand der Schablonenlänge des Clusters kann der Radius r des mittleren schwarzen Rings bestimmt werden. Mögliche Parameterschranken der zu testenden Ellipsen:
Kleine Halbachse b: $0,8r - 1,2r$
Große Halbachse: $1b - 2b$

Ist die am besten passende Ellipse gefunden, kann diese hochskaliert werden. Dadurch erhält man den Verlauf des äußersten Rings.



(a) 15 untersuchte eigene Ellipsen (von insgesamt 648). Die Zahl gibt jeweils die durchschnittliche Helligkeit der bedeckten Pixel an. Unten in der Mitte gibt es eine sehr gute Näherung.



(b) Die gefundene Ellipse wird mit dem Wert 5/3 vergrößert. Die neue Ellipse (gelb) beschreibt dann den Verlauf des äußersten Rings.

Abbildung 18: Bestimmung der am besten passenden Ellipse

Verzerrte Segmentgrößen beachten Wurde eine Ellipse mit Verzerrung entdeckt, folgt: Die einzelnen Segmente sind auch verzerrt und nehmen unterschiedliche Winkelbreiten ein (statt ursprünglich jeweils $360°/16 = 22{,}5°$). Beim Verfahren der Barcodeerzeugung wäre es nun ungeschickt, weiterhin konstante Winkelabstände zum Auslesen zu benutzten. Stattdessen bietet es sich an, bei weit entfernten Segmenten in engeren Abständen zu messen und bei nahen Segmenten in weiteren Abständen. Dadurch wird der Verzerrungseffekt aufgehoben (s. Abbildung 19).

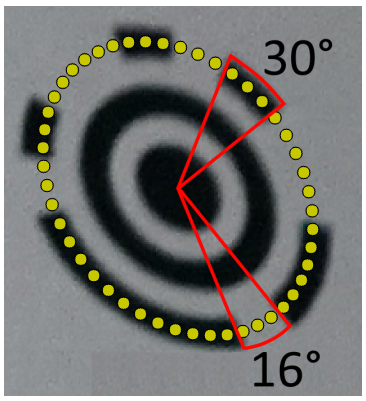


Abbildung 19: Im Umfeld des 16°-Segments wird enger ausgelesen und im Umfeld des 30°-Segments breiter, um die Verzerrung aufzuheben. Hier insgesamt 48 Messwerte, durchschnittlich 3 pro Segment.

Barcode in Blöcke einteilen Ein Barcode enthält keine explizite Information darüber, wo die einzelnen Segmente anfangen bzw. aufhören. Er ist stattdessen nur eine lückenlose Aneinanderreihung von Messwerten (s. Abbildung 20). Bei anderen Verfahren wird man in

ähnlicher Weise auf das Problem stoßen, zwischen verwaschenen Übergängen einzelne Segmente auszumachen.

Abbildung 20: Ein Barcode, bestehend aus 128 Messwerten (im Durchschnitt 8 pro Segment); gelb eingefärbt, um die Ansicht in den nächsten Abbildungen zu erleichtern.

Hier ist es naheliegend, den Barcode in 16 Blöcke zu einzuteilen (die dann jeweils für ein Segment stehen). Bei einer guten Einteilung sollte jeder Block eine möglichst homogene Helligkeit besitzen, also entweder hauptsächlich dunkel oder hauptsächlich hell sein. Ein mögliches Maß: Pro Block wird die Varianz der Helligkeiten berechnet. Je kleiner die Varianz, desto homogener der Block. Aus diesen 16 Blockvarianzen wird dann der Durchschnitt gebildet. Je kleiner dieser Durchschnitt, desto besser die Einteilung (vgl. Abbildung 21). Hier wird vorausgesetzt, dass im Barcode alle Segmente die selbe Breite haben, also aus der selben Anzahl an Messwerten bestehen. Das ist der Fall, wenn der Barcode vorher entzerrt wurde (vgl. Abbildung 19).

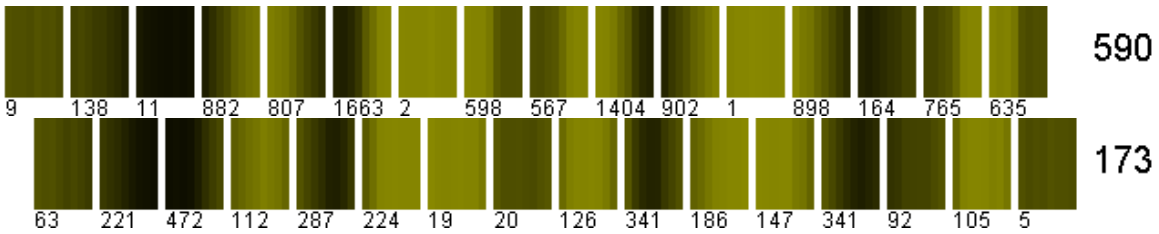


Abbildung 21: Zwei Einteilungen (von insgesamt 8 möglichen). Die untere teilt den Barcode besser auf als die obere, weil ihre Blöcke homogener sind. In diesem Fall ist die untere Einteilung sogar die beste und die obere die schlechteste

Bits zuweisen Abschließend kann jedem Block dessen „Bitwert“ zugewiesen werden: 0 oder 1. Ein Ansatz ist, die Blöcke untereinander in eine helle Gruppe und eine dunkle Gruppe aufzuteilen. Schließlich sollte erwartet werden können, dass jeder Block entweder ziemlich hell oder ziemlich dunkel ist. Ein mögliches Verfahren: Bei jedem Block wird zunächst der Durchschnitt seiner Messwerte gebildet. Nach diesem Durchschnitt werden die Blöcke sortiert und dann die größte paarweise Veränderung ermittelt, quasi der größte Helligkeitssprung. Bei diesem Sprung werden die beiden Gruppen aufgeteilt; vgl. Abbildung 22.

Anmerkung: Die beiden Kreiscodes „00“ (komplett weiß) und „Falsch“ (komplett schwarz) sind mit diesem Verfahren nicht kompatibel und müssen vorher abgefangen werden. Dies geht z.B., indem geprüft wird, ob der hellste Block und der dunkelste Block hinreichend nahe beieinander liegen. Als gut erwiesen hat sich hierfür eine Schwelle von 70.

In der Tabelle nachschlagen Ist die Bitfolge bestimmt, ist das Nachschlagen in der Tabelle kein großer Aufwand mehr. Es gibt 134 Einträge, jeder besteht aus 16 Bits, die auf 16

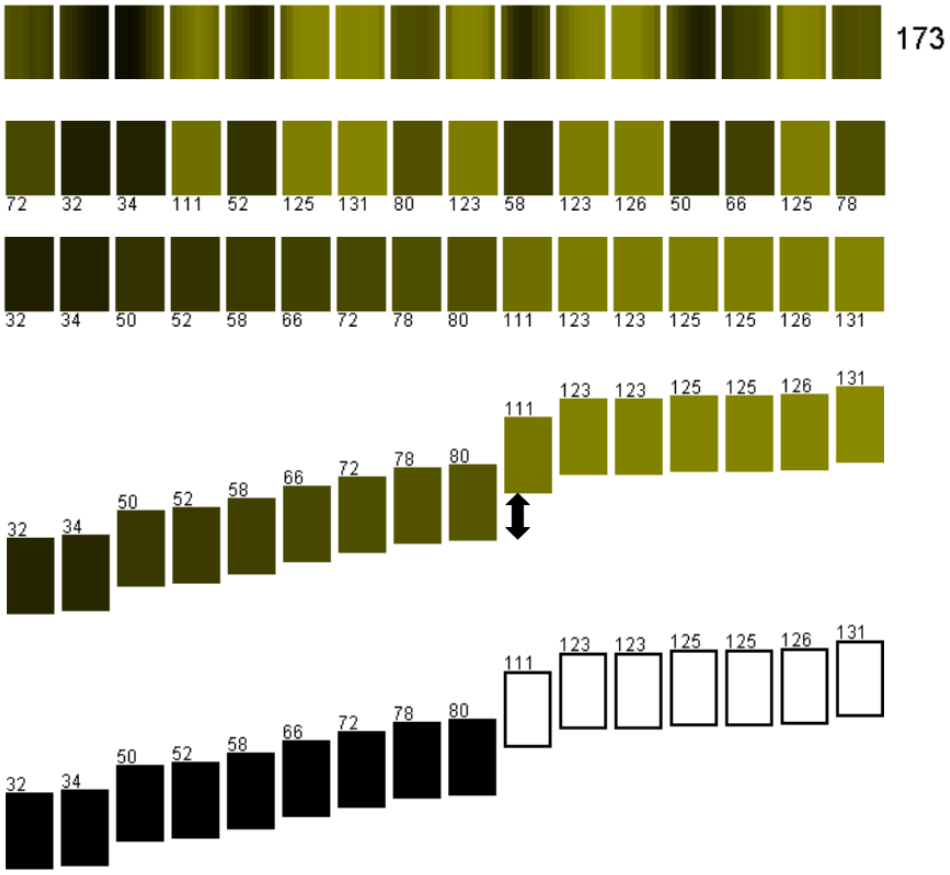


Abbildung 22: Vom Barcode zu einzelnen Bits. Die Werte pro Block stehen für deren Durchschnittshelligkeit. Der Helligkeitssprung von 80 auf 111 ist gut erkennbar, aber nicht so stark ausgeprägt wie in der Theorie erhofft.

verschiedene Arten rotiert sein können. Macht maximal $134 \cdot 16 \cdot 16 = 34,304$ nötige Vergleiche (plus ein bisschen Overhead). Das schafft ein Computer heutzutage im Bruchteil einer Sekunde.

Dieser Ansatz skaliert jedoch sehr schlecht. Wenn man den Anspruch hat, auch auf größere Tabellensysteme vorbereitet zu sein (z.B. 64 Segmente, 50.000 Einträge, ...), dann kann über Beschleunigungen nachgedacht werden. Eine erste wäre beispielsweise, die bitweisen Vergleiche nur auf diejenigen Einträge zu beschränken, welche dieselbe Anzahl an Einsen haben wie die gefundene Folge. Das allein reduziert den Rechenaufwand um 80-99%, verglichen mit dem naiven Ansatz.

Abgeschnittene Kreiscodes am Rand Auch bei einem unvollständigen Kreiscode können bestimmte Bedeutungen schon ausgeschlossen werden. Deshalb bietet es sich an, in diesen Fällen wenigstens eine reduzierte Liste aller noch möglichen Bedeutungen anzugeben. Das sind dann zwar oft noch sehr viele, aber immer noch besser als gar keine Auswahl.

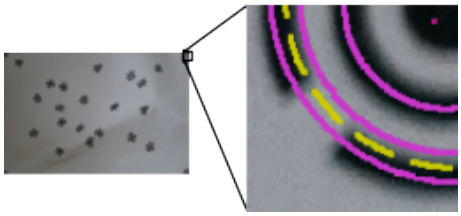


Abbildung 23: Gefundene Bitfolge: 11–.—-.—-.-110. Daraus ergeben sich folgende mögliche Bedeutungen: 02 0e 17 1d 1e ’) * . 5 = > ? @ A B C E I M N Q R T U Ö ‘ c e g k l m n o r s u z ö ü ß ll ul lr Wahr

3.3 Analyse

Laufzeitanalyse – Berechnung

Üblicherweise gibt man die Laufzeitkomplexität eines Algorithmus in der Landau- bzw. „Groß O“-Notation an. Das geht vor allem dann gut, wenn möglichst wenig Spielraum herrscht beim Eingabe- und Ausgabeformat. Bei dieser Aufgabe aber ist die Eingabe höchst chaotisch. Es gibt keinen vernünftigen Weg, mit nur ein paar Variablen eine Konstellation von über einer Million Pixel zu beschreiben.

Eine andere Art von Analyse ist jedoch möglich: Vorherzusagen, in welcher Größenordnung sich die Rechenzeit (in Sekunden) befindet. Dazu müssen zwei Sachen bekannt sein: Die Anzahl der elementaren Rechenoperationen, die das Programm ausführen wird, und die Taktfrequenz des Prozessors, die angibt, wie viele Operationen dieser pro Sekunde verarbeiten kann. Als elementare Rechenoperationen zählen wir hier Addition, Array lesen, Array schreiben und ähnliche Aufgaben, die sehr viel weniger Aufwand bedeuten als z.B. Fließkommaarithmetik. Bei unserem (unoptimierten) Ansatz verursachen die Schablonentests den größten Rechenaufwand. An ihnen wollen wir deshalb exemplarisch die Größenordnung der benötigten Laufzeit analysieren.

Als erstes wird ermittelt, wie viele Tests es je einzelnem Raster gibt. Da Raster zweidimensional aufgespannt werden, fließt der Rasterabstand $d(n)$ quadriert in die Rechnung ein. Er ist bei uns definiert als Schablonenlänge $L(n)$ durch Rasterauflösung a.

$$\text{Tests im Raster} = \frac{bh}{d(n)^2} \cdot t = \frac{bh}{(L(n)/a)^2} \cdot t = bh \left(\frac{a}{min \cdot k^n}\right)^2 \cdot t$$

Als zweites wird ermittelt, wie viele verschiedene Längen erzeugt werden; pro Länge gibt es schließlich ein neues Raster zu testen. Die Anzahl kann am größten benötigten Exponenten abgelesen werden, den wir N nennen.

$$max = min \cdot k^N \Leftrightarrow N = \frac{ln(max/min)}{ln(k)}$$

$$N = \frac{ln(2561/12)}{ln(1,15)} \approx 38,3742 \approx 38$$

Tabelle 1: Parameter, die man selbst festlegen kann bzw. muss, sind fett gedruckt. Die restlichen Parameter sind durch äußere Umstände vorbestimmt. Anmerkung zu p: Je Schablonenmesspunkt werden in unserer Implementierung 20-30 Operationen ausgeführt (also insges. ca. 100 Operationen pro Test). Für einen genauen Wert müsste man „Operation“ präziser definieren. Für eine solche Analyse reicht es aber, wenn die Größenordnung stimmt.

Parameter		Wert für Beispielrechnung
b	Bildbreite	3895 (Dimension Beispielfotos)
h	Bildhöhe	2561 (Dimension Beispielfotos)
min	Minimallänge	12
max	Maximallänge	2561 (Kleinerer Wert aus Breite, Höhe)
k	Skalierfaktor	1,15
t	Tests pro Ankerpunkt	20
a	Rasterauflösung	10 (Zehntel der Schablonenlänge)
p	Operationen pro Test	100 (geschätzt)
f	Taktfrequenz des Prozessors	2,4 GHz (in Systemsteuerung nachgeschaut)
n	aktueller Exponent	-

Durch Aufsummieren über alle Raster erhält man die Gesamtanzahl der durchzuführenden Tests:

$$\begin{aligned} \text{Anzahl Tests} &= \sum_{n=0}^{N} \text{Tests im Raster}(n) \\ &= \sum_{n=0}^{38} 3895 \cdot 2561 \cdot \left(\frac{10}{12 \cdot 1{,}15^n}\right)^2 \cdot 20 \approx 5{,}68 \cdot 10^8 \end{aligned}$$

Damit ergibt sich die Laufzeit folgendermaßen:

$$\text{Laufzeit} = \frac{p \cdot \text{Anzahl Tests}}{f} = \frac{100 \cdot 5{,}68 \cdot 10^8}{2{,}4 \cdot 10^9} = 23{,}6$$

Die Analyse sagt also eine Laufzeit von 23,6 Sekunden voraus. Das erlaubt folgende Aussage: Sogar wenn man sich beim Parameter p (Operationen pro Test) um eine ganze Größenordnung verschätzt haben sollte, terminiert das Programm in menschlicher Zeit (ca. 4 Minuten). Die Formel für Anzahl Tests eignet sich nun auch, um die Auswirkungen der einzelnen Parameter auf die Laufzeit zu betrachten. Die Dimension des Bildes z.B. wirkt sich fast nur linear auf die Laufzeit aus. N steigt nur logarithmisch mit der Dimension des Bildes, und wenn trotzdem neue Raster entstehen, befinden sich in ihnen exponentiell weniger Ankerpunkte als in den kleineren Rastern. Somit ist die Laufzeitkomplexität der Mittelpunktsfindung mit $\mathbf{O}(n)$ zu charakterisieren, wobei $n =$ Anzahl Pixel.

Falls mehrere echte Laufzeiten gemessen und kommentiert wurden, ist eine solche Analyse bei dieser Aufgabe nicht zwingend nötig. Aber sie ist eine schöne Übung und erlaubt es, die Ausführbarkeit des Algorithmus zu bestätigen, unabhängig von der Ausführumgebung. Und falls sich das Programm z.B. von den Bewertern nicht starten lässt, hilft sie, die Bewerter trotzdem von der behaupteten Laufzeit zu überzeugen.

Laufzeitanalyse – Messung

bitmap.gif	noise50.jpg	camB.jpg
19,8	20,2	19,5
20,3	19,5	19,9

Die oben angegebenen Messwerte (in Sekunden) bestätigen die vorhergesagte Laufzeit. Dass in der Analyse die einzige Information über das Bild dessen Pixelanzahl n war, wird durch das Messergebnis auch widergespiegelt: Alle drei (gleich großen) Bilder brauchen gleich lange, obwohl ihr „Aussehen“ sich stark unterscheidet.

Optimierung

Wenn davon ausgegangen wird, dass sich die Kreiscodes auf einem hellen Hintergrund befinden, lässt sich die Laufzeit stark reduzieren, mit nur minimalem Qualitätsverlust. Denn es können nun alle Ankerpunkte übersprungen werden, die eine durchschnittliche oder überdurchschnittliche Helligkeit haben, da sie (in fast allen Fällen) sowieso nicht als Mittelpunkt in Frage kommen. Durch diese härtere Ankerpunkte-Auswahl verringert sich die Laufzeit der Schablonensuche bei den Fotografien von 20 Sekunden auf 2 Sekunden, bei den S/W-Bildern auf 0,6 Sekunden. Inklusive der restlichen Programmteile (Bild laden: 1,2 Sekunden, Gauß-Weichzeichnen: 0,6 Sekunden, Ellipsen finden: 0,4 Sekunden für 20 Kreiscodes.) kommt das Programm damit bei allen Beispiel-Fotografien auf eine Gesamtlaufzeit von gut 4 Sekunden.

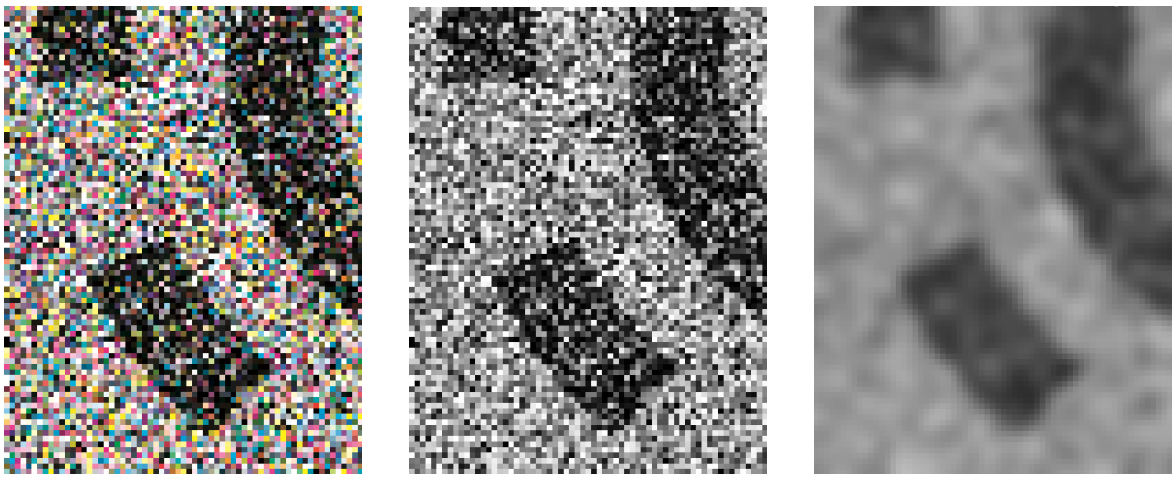
Fehlerresistenz

Es kann geschehen, dass sich die gefundene Bitfolge nicht in der Tabelle wiederfinden lässt. Ein gutes Programm kann hartnäckig bleiben und noch einmal prüfen, ob sich nicht durch eine kleine Korrektur eine korrekte Folge erzielen lässt. Eine sehr allgemeine Korrektur wäre z.B., jeweils ein Bit in der gefundenen Folge zu switchen und dann zu schauen, ob die neu generierte Folge sich in der Tabelle wiederfinden lässt.

Graustufen und Bildglättung

Es reicht aus, nur Helligkeitskontraste zu beachten (statt Farbkontraste). Das lässt sich z.B. dadurch rechtfertigen, dass man auch im echten Leben fast nur dunklen Barcodes und QR-Codes begegnet, die sich vor einem hellen Hintergrund befinden.

Um beim Rauschen Ausreißer zu erkennen bzw. zu eliminieren bietet es sich an, eine Form von Bildglättung anzuwenden. Wird dies live beim Auslesen bewerkstelligt, könnte man den Betrachtungsradius um den Pixel herum entsprechend der Kreiscodegröße skalieren. Wird stattdessen das gesamte Bild schon zu Beginn geglättet, sollte darauf geachtet werden, dass die Glättung nicht so stark ist, dass sie kleine Kreiscodes zerstört. Ist sie dann zu schwach für



(a) Ursprungsbild (b) Graustufen (c) zweimal geglättet

Abbildung 24: Umwandlung in Graustufen und Bildglättung

große Kreiscodes, kann bei Bedarf bei entsprechender Suchgröße ein zweites oder drittes Mal geglättet werden.

Als Glättverfahren ist ein 5×5 Gaussfilter möglich. Dieser kann z.B. einmal zu Beginn angewendet werden und ein zweites Mal bei einer Schablonenlänge von ca. 20 Pixeln. Hier ein 5×5 Filterkern mit (Integer-gerundeter) Gaussverteilung von Sigma=1,4:

$$\frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Fremde Algorithmen und Bibliotheken

Eigene Implementierungen fremder (zitierter) Ideen sind erlaubt. Hier besteht oft schon eine Leistung darin, die Implementierung korrekt hinzubekommen. Wird stattdessen gleich ein ganzer lauffähiger Code kopiert (bzw. aus einer Bibliothek benutzt), sollte trotzdem für die Bewerter klar erkennbar sein, dass die Funktionsweisen des Codes und die Idee dahinter ausreichend verstanden wurden. Das ließe sich z.B. dadurch erreichen, dass man auf die jeweiligen Parameter des kopierten Codes eingeht und erklärt, ob und warum man diese verändert oder warum man sie auf den Standardwerten gelassen hat.

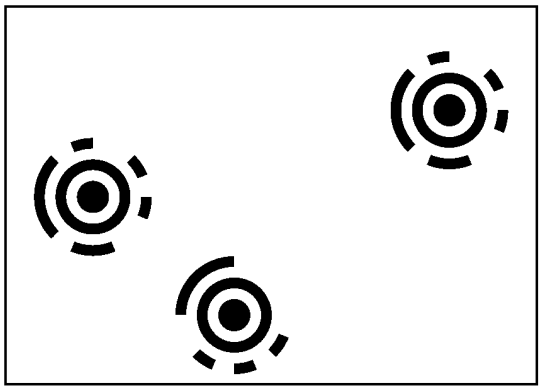
Diese Lösung benutzt z.B. eine fremde Implementierung zum Erzeugen der Ellipsen[8]. Bei dieser ist es besonders praktisch, dass durch die dort gewählte Parameterdarstellung (sin, cos) automatisch der von uns gewünschte Effekt eintritt, dass an weit enfernten Bereichen enger ausgelesen werden soll.

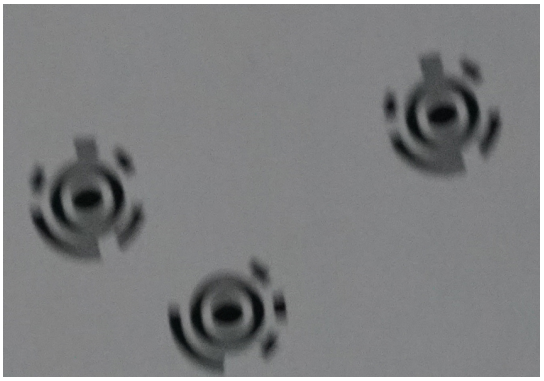
[8] http://www.uni-forst.gwdg.de/~wkurth/cb/html/cg_v05a.htm

Mögliche Alternativen zur Schablonensuche

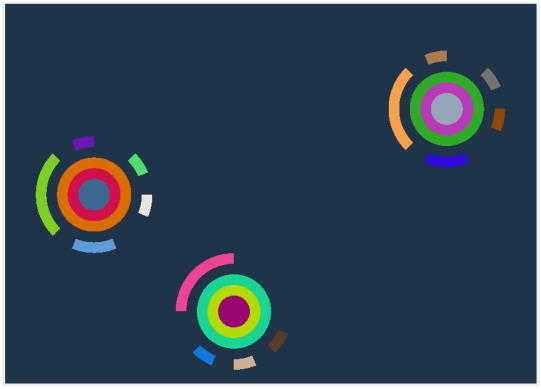
Zwei alternative Ansätze wurden zu Beginn erwähnt. Deshalb hier zumindest jeweils zwei Bilder, um ein Gefühl für die Daten zu bekommen, die statt der Trefferquoten entstehen.

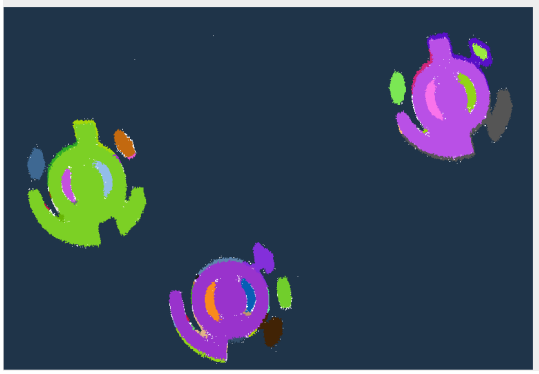
Ausgangsbilder:



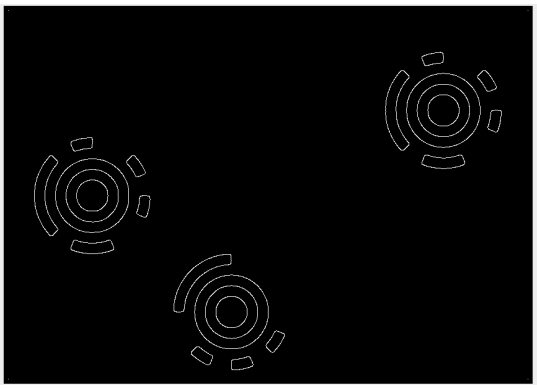


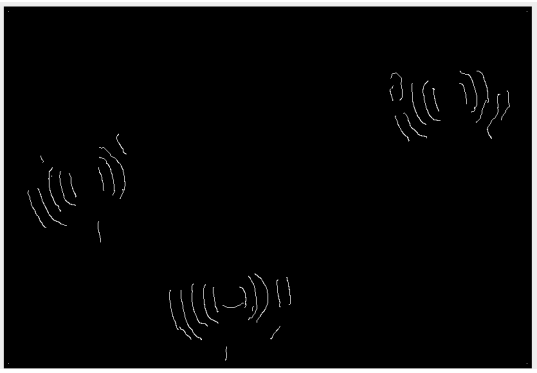
Blob-Aufteilung:





Kantenerkennung:





3.4 Bewertungskriterien

- Angesichts der Größe der vorgegebenen Bilder sollte die Laufzeit des Verfahrens begrenzt bleiben. Die meisten sinnvoll anwendbaren Verfahren sind linear in der Anzahl n

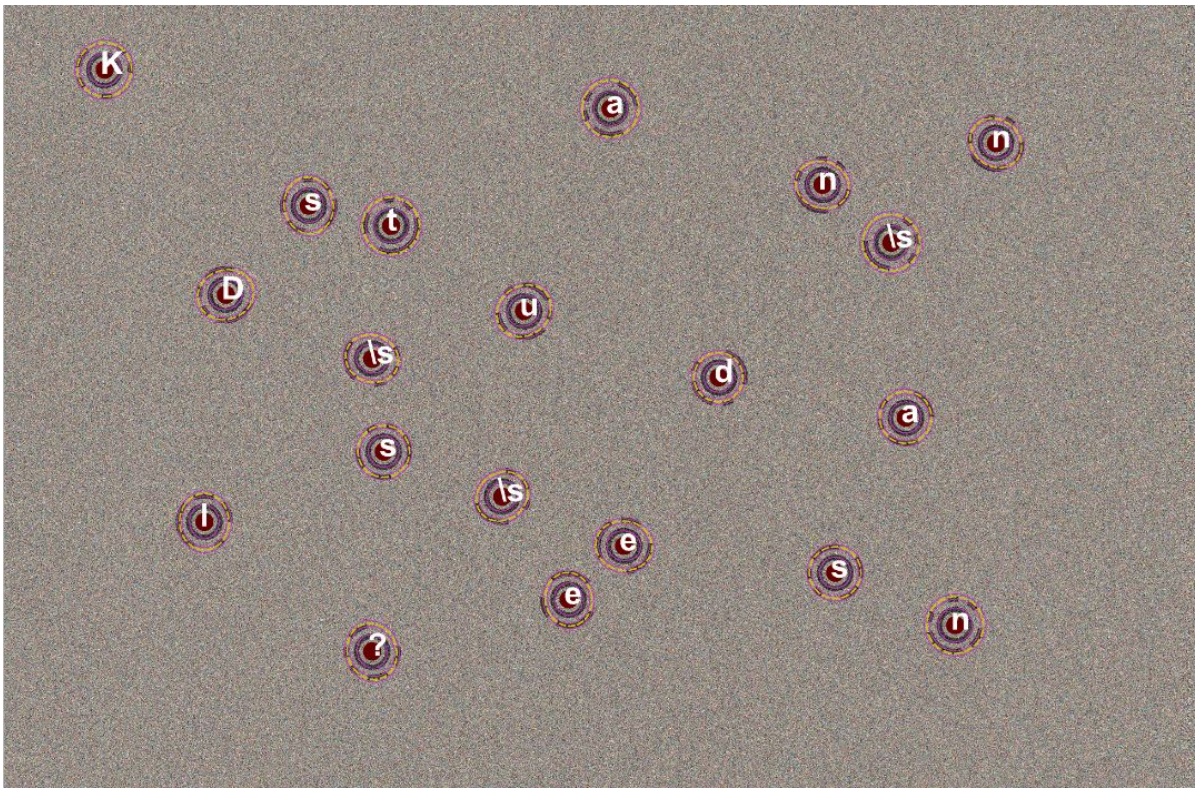
der Bildpunkte. Insgesamt sollte die Laufzeitkomplexität maximal in $\mathbf{O}(n \cdot ln(n))$ liegen. Fallen bei aufwändigen Analysen große Datenmengen an, kann dem durch Rasterung, Clustern oder Durchschnittsbildungen begegnet werden.

- Die Teilaufgaben 1 und 2, also Mittelpunktbestimmung und Dekodierung für ein Schwarz-Weiß-Bild mit sauber kreisförmigen Codes, sollten erfolgreich bearbeitet worden sein.
- In Teilaufgabe 3 wird es schwieriger. Die Bewertung hängt davon ab, mit welchen Schwierigkeiten die Lösung zurecht kommt. Kritisch sind folgende Aspekte:
  - Verzerrungen: Wer nur (Zitat aus einer Einsendung) „runde Kreise“ erkennt, kommt mit Verzerrungen z.B. bei Aufnahmen von der Seite nicht zurecht.
  - Unschärfe und Rauschen erfordert geeignete Vorverarbeitung durch Filtern und Glätten etc. Es sollte darauf geachtet werden, dass bei anderen Bildern potenziell sehr kleine Kreiscodes nicht kaputt geglättet werden.
  - Die mögliche Rotation der Segmentübergänge sollte berücksichtigt werden.
  - Schwankende Helligkeiten z.B. durch Schatten machen u.a. erfoderlich, dass eine Bitzuweisung je Segment relativ zum Umfeld geschieht und nicht anhand einer vorher festgelegten Helligkeitsschwelle.
  - Bonuspunkte werden vergeben, wenn das Programm überdurchschnittlich gut ist und auch mit schwierigen Störeffekten bzw. Testbildern zurechtkommt.
- Laufzeitüberlegungen werden auch bei dieser Aufgabe erwartet; hier werden allerdings häufig Schätzungen oder qualitative Bewertungen der Auswirkungen bestimmter Parameter einfließen müssen. Deshalb sollten zusätzlich auch Laufzeitmessungen durchgeführt werden, die die Schätzungen (hoffentlich) bestätigen.
- Auch bei dieser Aufgabe können bekannte Verfahren eingesetzt werden. Ähnlich wie in Aufgabe 1 muss gut begründet werden, dass diese Verfahren funktionieren; es soll auch erkennbar werden, dass die Effekte übernommener Verfahren verstanden wurden.
- Die Wahl der zentralen Parameter und Toleranzgrenzen für die Identifikation von Kreiscode-Zeichen sollte geeignet sein und begründet werden.
- Die vorgegebenen Beispiele sollten bearbeitet sein. Weitere, selbst erstellte Beispiele sind insbesondere dann sinnvoll, wenn damit weitergehende Leistungen, aber auch die Grenzen der Lösung aufgezeigt werden. In der Informatik sollte man sich immer darüber im klaren sein, was ein System kann – und was es nicht kann.
- Es ist nicht nötig, die gefundenen Dekodierungen in die Bilder einzubauen. Aber es genügt auch nicht, einfach den dekodierten Text auszugeben (der in den Beispielen ohnehin immer gleich ist). Die Ausgabe sollte nachvollziehbar machen, wie das Verfahren zur Dekodierung gekommen ist. Zwischenausgaben nach verschiedenen Arbeitsschritten (Filterung, Mittelpunktbestimmung etc.) sind sinnvoll.
- Da die Aufgabe ohnehin Kreativität erfordert, eröffnet sich auch Spielraum für Erweiterungen. Neue Probleme entstehen, wenn die Codes andere Formen annehmen können. Ansprüche an die Bitzuweisung erhöhen sich, wenn eine deutlich größere Code-Tabelle und damit eine deutlich höhere Auflösung der Bit-Codierung angenommen wird.

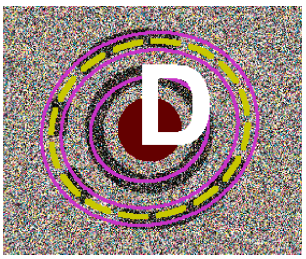
3.5 Beispiele

In allen vorgegebenen Beispielen ist Folgendes kodiert: Kannst Du das lesen?

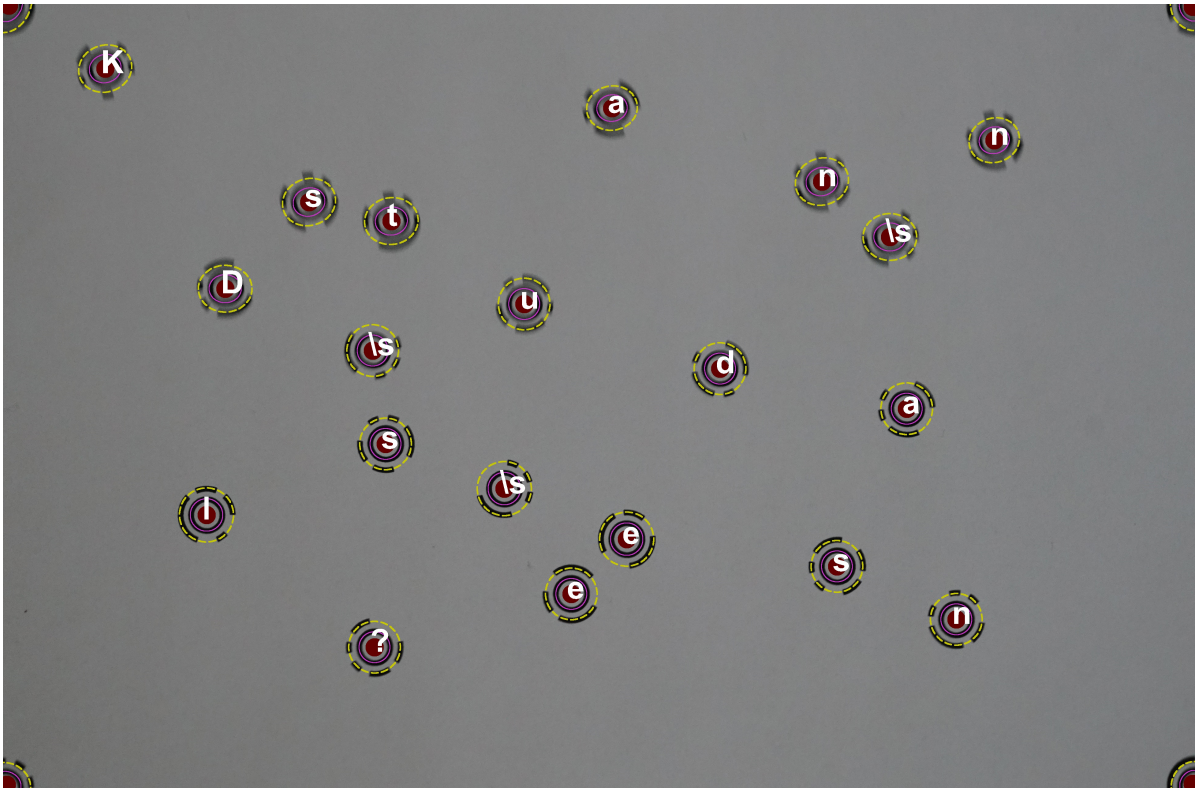
noise50.jpg




```
.0001.0010.1100.0101 (x: 328, y: 211): K
.0011.1010.1000.1110 (x: 1981, y: 338): a
.1010.0101.0011.0111 (x: 3240, y: 451): n
.1101.0010.1001.1011 (x: 2673, y: 587): n
.1001.0101.1100.1101 (x: 995, y: 655): s
.1111.1001.1001.0101 (x: 1264, y: 721): t
.0100.1111.0000.0101 (x: 2898, y: 776): \s
.1010.1011.1000.0111 (x: 724, y: 946): D
.1101.1001.0111.0101 (x: 1698, y: 999): u
.0111.1000.0010.1010 (x: 1202, y: 1155): \s
.0010.1110.1000.1111 (x: 2334, y: 1216): d
.0011.1010.1000.1110 (x: 2945, y: 1347): a
.1001.1011.0010.1011 (x: 1240, y: 1458): s
.1111.0000.0101.0100 (x: 1627, y: 1605): \s
.0100.1111.1011.0110 (x: 654, y: 1688): l
.0011.1101.1001.1110 (x: 2025, y: 1766): e
.0110.0101.0111.0011 (x: 2715, y: 1854): s
.0011.1100.0111.1011 (x: 1844, y: 1941): e
.1101.1110.1001.0100 (x: 3107, y: 2025): n
.1011.0000.1101.1010 (x: 1202, y: 2111): ?
```

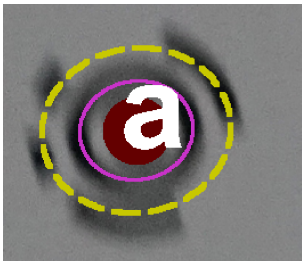


cam8.jpg



Die „Viertel-Kreiscodes“ in den Ecken werden erkannt, die (naturgemäß erfolglose) Decodierung wird hier nicht angegeben.

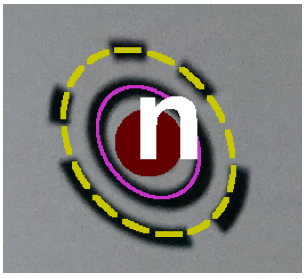
```
.1001.0110.0010.1000 (x: 334, y: 211): K
.1101.0100.0111.0001 (x: 1990, y: 340): a
.1110.1001.0100.1101 (x: 3239, y: 445): n
.1111.0100.1010.0110 (x: 2676, y: 580): n
.0110.1100.1010.1110 (x: 1000, y: 647): s
.1001.1001.0101.1111 (x: 1269, y: 709): t
.1111.0000.0101.0100 (x: 2898, y: 761): \s
.0101.0111.0000.1111 (x: 726, y: 930): D
.0111.0110.0101.1101 (x: 1703, y: 980): u
.1010.0111.1000.0010 (x: 1208, y: 1132): \s
.0010.1110.1000.1111 (x: 2343, y: 1191): d
.0011.1010.1000.1110 (x: 2953, y: 1322): a
.1001.1011.0010.1011 (x: 1250, y: 1437): s
.1001.1110.0000.1010 (x: 1638, y: 1583): \s
.0100.1111.1011.0110 (x: 665, y: 1669): l
.1110.0011.1101.1001 (x: 2038, y: 1748): e
.0110.0101.0111.0011 (x: 2725, y: 1837): s
.0111.1011.0011.1100 (x: 1857, y: 1927): e
.1101.1110.1001.0100 (x: 3115, y: 2010): n
.1011.0101.0110.0001 (x: 1214, y: 2101): ?
```



camB.jpg

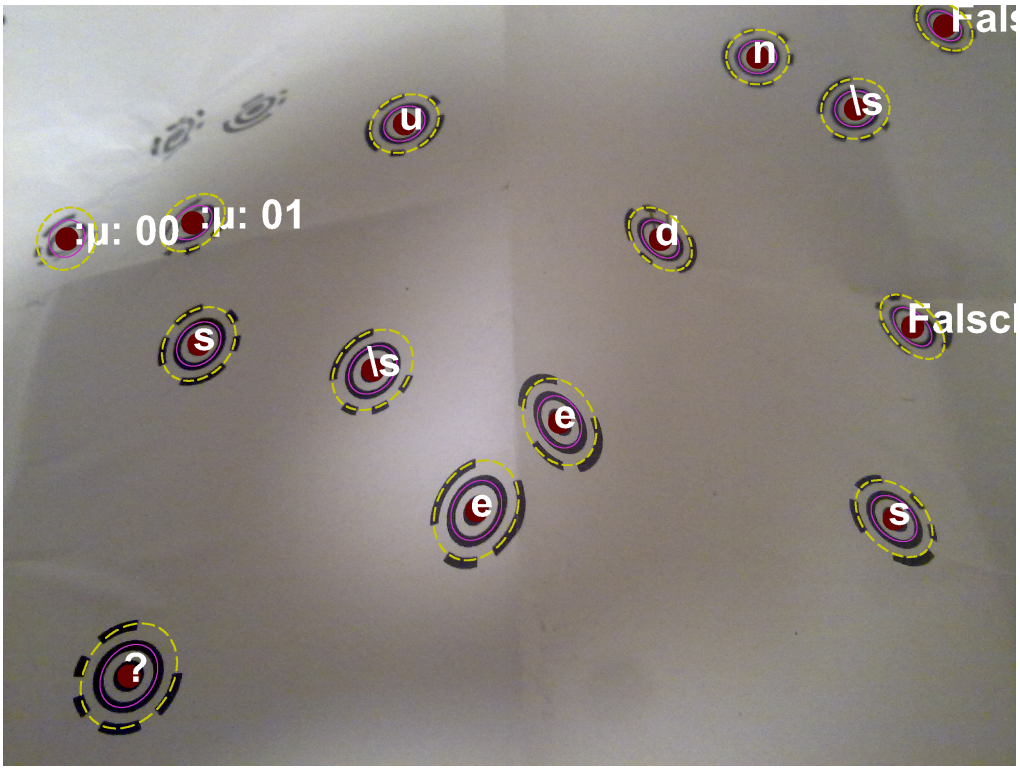
Die „Viertel-Kreiscodes“ in den Ecken werden erkannt, die (naturgemäß erfolglose) Decodierung wird hier nicht angegeben.

```
.0001.0010.1100.0101 (x: 321, y: 217): K
.0011.1010.1000.1110 (x: 1983, y: 325): a
.0100.1010.0110.1111 (x: 3265, y: 330): n
.0111.1010.0101.0011 (x: 2688, y: 483): n
.0011.1100.0001.0101 (x: 2922, y: 641): \s
.1100.1101.1001.0101 (x: 984, y: 663): s
.1111.1001.1001.0101 (x: 1253, y: 724): t
.1111.0101.0111.0000 (x: 710, y: 967): D
.0111.0110.0101.1101 (x: 1688, y: 1003): u
.1011.1010.0011.1100 (x: 2334, y: 1150): d
.1010.0111.1000.0010 (x: 1186, y: 1168): \s
.0111.0101.0001.1100 (x: 2967, y: 1209): a
.1001.1011.0010.1011 (x: 1222, y: 1475): s
.1111.0000.0101.0100 (x: 1608, y: 1598): \s
.1000.1111.0110.0111 (x: 2010, y: 1698): e
.0100.1111.1011.0110 (x: 638, y: 1714): l
.0110.0101.0111.0011 (x: 2702, y: 1801): s
.1111.0110.0111.1000 (x: 1820, y: 1894): e
.1101.1110.1001.0100 (x: 3095, y: 1996): n
.1011.0101.0110.0001 (x: 1176, y: 2091): ?
```



Eigenes Beispiel

Dieses Foto ist durch stark unterschiedliche Lichtverhältnisse, Rauschen und Verformungen des Papiers besonders schwierig und zeigt selbst sehr guten Lösungen ihre Grenzen auf.



Perlen der Informatik – aus den Einsendungen

Teilweise mit Kommentaren von der Bewertung

Allgemeines

Brute Force hat nichts gefunden. *Kein Wunder; Brute, der alte Versager!*

Außerdem sollte der aktuelle Quelltext noch einmal überdacht werden, da er ziemlich unübersichtlich und teilweise auch wahrscheinlich redundant ist.

```
for(Knoten x:kanten)
```

Aufgabe 1: Rosinen picken

Abfressen der Teilbäume

Summer der Gewichtungen

Wer ein perfektes Ergebnis sucht und dafür bei komplexeren Aufgaben etwas mehr Berechnungszeit in Kauf nehmen kann, ist beim komplexen Verfahren richtig.

Das bedeutet, dass es auch günstiger sein könnte, nur die Zuckerraffinerie anstelle der Kombination von Gummibärenfabrik, Altreifenwiederverwertung und Raffinerie zu kaufen.

Die Laufzeit meines Programms ist so verdammt kurz, dass man es nicht mit einer einfachen Stoppuhr hätte zählen können.

Damit die einzelne Zahl ein positiver Summand ist, muss deren Wert positiv sein.

Aus dem Quellcode:

```
// Wir brauchen für diese vollständige Suche
// eine Liste der bekannten Statusse.
// Duden sagt es heißt Status, aber das ist mir egal.
// Wer kann dann noch unterscheiden, ob es Plural oder Singular ist?
// Deutsch ist nichtdeterministisch!
```

Aufgabe 2: Rechtsrum in Rechthausen

restregierte Zusammenhangskomponente

Zum Berechnen des kürzesten Weges zwischen zwei Kreuzungen muss nun einfach nur der richtige Algorithmus [...] angewandt werden.

Das Problem ist auf jeden Fall durch Brute-Force zu löschen.

Ebenfalls nicht erlaubt sollten 360°-Wendungen an Kreuzungen sein.

Es ist vielleicht besser, Rechthausen zu richten.

Allerdings kann man leicht Beispiele konstruieren, wo diese Definition trotz $d > 3$ einen Autofriedhof herstellt.

In der EI-Community findet sich auch ein Beispiel mit 10000 Knoten. Auf eine graphische Darstellung wird der sterbenden Bäume zuliebe verzichtet.

... der rechteste aller Wege, die nach links gehen, sofern dieser weniger als halb so links ist als der linkeste aller rechten Wege rechts.

Es muss schließlich aus jedem Punkt einen legalen Ausweg geben.

Die priority queue wird mit Hilfe von `std::multimap<float, const int>` umgesetzt. *Anmerkung: Es gibt auch* `std::priority_queue`...

Die Anwohner von Rechthausen sind unzufrieden. Obwohl der Bürgermeister die Berichte, das Linksabbiegeverbot würde die Fahrzeit um bis zu 200% erhöhen, abstreitet und als Fake-News betitelt, sträuben sich die Bewohner gegen die neue Verkehrsführung. Nachdem das durchgeführte Referendum klar gegen das Linksabbiegeverbot ausfiel, steht der Bürgermeister unter Zugzwang. Ihm kommt die Idee, Kreuzungen durch Kreisverkehre zu ersetzen.

Aufgabe 3: Kreis-Code

Es handelt sich nicht um runde Kreise.

Bild wird binarisiert.

Diese Funktion ist der Gimp-Funktion Codes -> Thresholds nachempfunden.