41. Bundeswettbewerb Informatik 2022/2023, 1. Runde

Bundeswettbewerb Informatik

Lösungshinweise

Allgemeines

Es ist sehr erfreulich, dass wieder sehr viele die Aufgaben bearbeitet und am Bundeswettbewerb Informatik teilgenommen haben. Den Veranstaltern ist bewusst, dass in der Regel viel Arbeit hinter der Erstellung einer Einsendung steckt.

Bei der Bewertung wurden die in den Einsendungen gezeigten Leistungen so gut wie möglich gewürdigt. Das ist nicht immer leicht, insbesondere wenn die Dokumentation nicht die im Aufgabenblatt genannten Anforderungen erfüllt. Bevor mögliche Lösungsideen zu den Aufgaben und Einzelheiten zur Bewertung beschrieben werden, soll deshalb im folgenden Abschnitt auf die Dokumentation näher eingegangen werden; vielleicht helfen diese Anmerkungen bei der nächsten Teilnahme.

Wie auch immer die Bewertung einer Einsendung ausfällt, sie sollte nicht entmutigen! Allein durch die Arbeit an den Aufgaben und ihren Lösungen hat jede Teilnehmerin und jeder Teilnehmer einiges gelernt; diesen Effekt sollte man nicht unterschätzen.

Die Bearbeitungszeit für die 1. Runde beträgt etwa drei Monate. Frühzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, zeitliche Engpässe am Ende der Bearbeitungszeit gerade mit der Dokumentation zu vermeiden. Aufgaben sind gelegentlich schwerer zu bearbeiten, als es auf den ersten Blick erscheinen mag. Erst bei der konkreten Umsetzung einer Lösungsidee oder beim Testen von Beispielen kann man auf Besonderheiten oder Schwierigkeiten stoßen, die zusätzlicher Zeit bedürfen.

Noch etwas Organisatorisches: Sollte der Name auf Urkunde oder Teilnahmebescheinigung falsch geschrieben sein, ist er auch im AMS (login.bwinf.de) falsch eingetragen. Die Teilnahmeunterlagen können gerne neu angefordert werden; dann aber bitte vorher den Eintrag im AMS korrigieren.

Dokumentation

Die Zeit für die Bewertung ist leider begrenzt, weshalb es unmöglich ist, alle eingesandten Programme gründlich zu testen. Die Grundlage der Bewertung ist deshalb die Dokumentation, die, wie im Aufgabenblatt beschrieben, für jede bearbeitete Aufgabe aus den Teilen *Lösungsidee*, *Umsetzung*, *Beispielen* und *Quellcode* bestehen soll. Leider sind die Dokumentationen bei vielen Einsendungen sehr knapp ausgefallen, was oft zu Punktabzügen führte, die das Erreichen der 2. Runde verhinderten. Grundsätzlich kann die Dokumentation einer Aufgabe als „sehr unverständlich oder nicht vollständig“ bewertet werden, wenn die schriftliche Darstellung kaum verständlich ist oder Teile wie Lösungsidee, Umsetzung oder Quellcode komplett fehlen.

Die Beschreibung der *Lösungsidee* sollte keine Bedienungsanleitung des Programms oder eine Wiederholung der Aufgabenstellung sein. Stattdessen soll erläutert werden, welches fachliche Problem hinter der Aufgabe steckt und wie dieses Problem grundsätzlich mit einem Algorithmus sowie Datenstrukturen angegangen wurde. Ein einfacher Grundsatz ist, dass Bezeichner von Programmelementen wie Variablen, Methoden etc. nicht in der Beschreibung einer Lösungsidee verwendet werden, da sie unabhängig von solchen technischen Realisierungsdetails sind. Wenn die Beschreibungen in der Dokumentation nicht auf die Lösungsidee eingehen oder bzgl. der Lösungsidee kaum nachvollziehbar sind, kann es einen Punktabzug geben, weil das „Verfahren unzureichend begründet bzw. schlecht nachvollziehbar“ ist.

Ganz besonders wichtig sind die vorgegebenen (und ggf. weitere) *Beispiele*. Wenn Beispiele und die zugehörigen Ergebnisse in der Dokumentation ganz oder teilweise fehlen, führt das zu Punktabzug. Zur Bewertung ist für jede Aufgabe vorgegeben, zu wie vielen (und teils auch zu welchen) Beispielen Programmausgaben oder Ergebnisse in der Dokumentation erwartet werden. Diese Ergebnisse sollten idealerweise korrekt sein. Punktabzug für falsche Ergebnisse gibt es aber nur, wenn nicht schon für den ursächlichen Mangel ein Punkt abgezogen wurde.

Es ist nicht ausreichend, Beispiele nur in gesonderten Dateien abzugeben, ins Programm einzubauen oder den Bewerterinnen und Bewertern sogar das Erfinden und Testen geeigneter Beispiele zu überlassen. Beispiele sollen die Korrektheit der Lösung belegen und auch zeigen, wie das Programm mit Sonderfällen umgeht.

Auch *Quellcode*, zumindest dessen für die Lösung wichtigen Teile, gehört in die Dokumentation; Quellcode soll also nicht nur elektronisch als Code-Dateien (als Teil der Implementierung) der Einsendung beigefügt werden. Schließlich gehören zu einer Einsendung als Teil der Implementierung *Programme*, die durch die genaue Angabe der Programmiersprache und des verwendeten Interpreters bzw. Compilers möglichst problemlos lauffähig sind und hierfür bereits fertig (interpretierbar/kompiliert) vorliegen. Die Programme sollten vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner hinsichtlich ihrer Lauffähigkeit getestet werden.

Hilfreich ist oft, wenn die Erstellung der Dokumentation die Programmierarbeit begleitet oder ihr teilweise sogar vorangeht. Wer es nicht ausreichend schafft, seine Lösungsidee für Dritte verständlich zu formulieren, dem gelingt meist auch keine fehlerlose Implementierung, egal in welcher Programmiersprache. Daher kann es nützlich sein, von Bekannten die Dokumentation auf Verständlichkeit hin lesen zu lassen, selbst wenn sie nicht vom Fach sind.

Wer sich für die 2. Runde qualifiziert hat, beachte bitte, dass dort deutlich kritischer als in der 1. Runde bewertet wird und höhere Anforderungen an die Qualität der Dokumentationen und Programme gestellt werden. So werden in der 2. Runde unter anderem eine klar beschriebene Lösungsidee (mit geeigneten Laufzeitüberlegungen und nachvollziehbaren Begründungen), übersichtlicher Programmcode (mit verständlicher Kommentierung), genügend aussagekräftige Beispiele (zum Testen des Programms) und ggf. spannende eigene Erweiterungen der Aufgabenstellung (für zusätzliche Punkte) erwartet.

Bewertung

Bei den im Folgenden beschriebenen Lösungsideen handelt es sich um Vorschläge, aber sicher nicht um die einzigen Lösungswege, die korrekt sind. Alle Ansätze, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind, werden in der Regel akzeptiert. Einige

Dinge gibt es allerdings, die – unabhängig vom gewählten Lösungsweg – auf jeden Fall erwartet werden. Zu jeder Aufgabe werden deshalb im jeweils letzten Abschnitt die Kriterien näher erläutert, auf die bei der Bewertung dieser Aufgabe besonders geachtet wurde. Die Kriterien sind in der Bewertung, die man im AMS einsehen kann, aufgelistet. Außerdem gibt es aufgabenunabhängig einige Anforderungen an die Dokumentation, die oben erklärt sind.

In der 1. Runde geht die Bewertung von fünf Punkten pro Aufgabe aus, von denen bei Mängeln Punkte abgezogen werden. Da es nur Punktabzüge gibt, sind die Bewertungskriterien meist negativ formuliert. Wenn das (Negativ-)Kriterium erfüllt ist, gibt es einen Punkt oder gelegentlich auch zwei Punkte Abzug; ansonsten ist die Bearbeitung in Bezug auf dieses Kriterium in Ordnung. Wurde die Aufgabe insgesamt nur unzureichend bearbeitet, wird ein gesondertes Kriterium angewandt, bei dem es bis zu fünf Punkten Abzug geben kann. Im schlechtesten Fall wird eine Aufgabenbearbeitung mit 0 Punkten bewertet.

Für die Gesamtpunktzahl sind die drei am besten bewerteten Aufgabenbearbeitungen maßgeblich. Es lassen sich also maximal 15 Punkte erreichen. Einen 1. Preis erreicht man mit 14 oder 15 Punkten, einen 2. Preis mit 12 oder 13 Punkten und einen 3. Preis mit 9 bis 11 Punkten. Mit einem 1. oder 2. Preis ist man für die 2. Runde qualifiziert. Kritische Fälle mit nur 11 Punkten sind bereits sehr gründlich und mit viel Wohlwollen geprüft. Leider ließ sich aber nicht verhindern, dass Teilnehmende nicht weitergekommen sind, die nur drei Aufgaben abgegeben haben in der Hoffnung, dass schon alle richtig sein würden; dies ist ziemlich riskant, da sich leicht Fehler einschleichen.

Auch wurde in einigen Fällen die Regelung zur Bearbeitung von Junioraufgaben als Teil einer Einsendung zum Bundeswettbewerb Informatik nicht beachtet. Hierzu ein Zitat aus dem Mantelbogen des Aufgabenblatts: „Die etwas leichteren Junioraufgaben dürfen nur von SchülerInnen vor der Qualifikationsphase des Abiturs bearbeitet werden.“ Nur unter Einhaltung dieser Bedingung können Bearbeitungen von Junioraufgaben im Bundeswettbewerb gewertet werden.

Danksagung

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

An der Erstellung der im folgenden skizzierten Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Personen mit: Katharina Bade (Junioraufgabe 1), Gabriel Dengler (Junioraufgabe 2), Jannik Kudla (Aufgabe 1), Tim Pokart (Aufgabe 2 und Aufgabe 4), Hans-Martin Bartram (Aufgabe 3), Dr. Martin Schmidt (Aufgabe 5) und Manuel Gundlach. Allen Beteiligten sei für ihre Mitarbeit ganz herzlich gedankt.

Junioraufgabe 1: Reimerei

J1.1 Lösungsidee

Für jedes Paar von Wörtern (w_1, w_2) muss geprüft werden, ob die drei gegebenen Voraussetzungen erfüllt werden.

Dafür muss zunächst für jedes Wort die jeweils maßgebliche Vokalgruppe bestimmt werden. Hierzu kann das gesamte Wort durchlaufen und nach allen möglichen Vokalgruppen durchsucht werden. Wichtig dabei ist, dass es um Vokal*gruppen* geht. Aufeinanderfolgende Vokale (z. B. Wiese) müssen also auch als Gruppe betrachtet werden. Von allen gefunden Vokalgruppen kann dann die vorletzte – sofern es mindestens zwei Vokalgruppen gibt – oder die einzige – falls es nur eine Vokalgruppe gibt – als die maßgebliche Vokalgruppe gespeichert werden. Wurde überhaupt keine Vokalgruppe gefunden, kann das Wort aus der Liste der zu betrachtenden Wörter gestrichen werden. Es genügt, den Index, also die Position im Wort, der maßgeblichen Vokalgruppe im Wort zu speichern. Natürlich kann zusätzlich aber auch die Vokalgruppe an sich gespeichert werden.

Nun können die einzelnen Regeln mithilfe der beiden berechneten maßgeblichen Vokalgruppen überprüft werden. Dazu können beispielsweise beide Wörter rückwärts, beginnend beim letzten Buchstaben, Buchstabe für Buchstabe verglichen werden, sodass bekannt ist, ab welchem Index sich beide Wörter gleichen. Sei $i_{1\to 2}$ der kleinste Index, ab welchem w_1 dem Ende des Worts w_2 gleicht, und entsprechend $i_{2\to 1}$ der kleinste Index, ab welchem w_2 dem Wort w_1 gleicht.[1] Mit dieser Information und den Indizes vok_1, vok_2, an denen die maßgeblichen Vokalgruppen in w_1 und w_2 starten, ist es möglich, alle Regeln schnell zu überprüfen.

1. Regel: Beide Wörter enden gleich.

Dazu wird für jedes der beiden Wörter geprüft, ob der Index des Starts der maßgeblichen Vokalgruppe größer oder gleich dem Index ist, ab dem das Wort dem anderen gleicht. Damit gleichen sie sich mindestens ab der maßgeblichen Vokalgruppe (und ggf. auch noch davor):

$$i_{1\to 2} \leq vok_1 \text{ und } i_{2\to 1} \leq vok_2$$

2. Regel: Die maßgebliche Vokalgruppe und was ihr folgt enthält mindestens die Hälfte der Buchstaben.

Dazu wird für jedes der beiden Wörter geprüft, ob der (*nullbasierte*) Index des Starts der maßgeblichen Vokalgruppe kleiner oder gleich der Hälfte der Länge des dazugehörigen Wortes ist:

$$vok_1 \leq \frac{l_1}{2} \text{ und } vok_2 \leq \frac{l_2}{2}$$

3. Regel: Keines der beiden Wörter darf mit dem kompletten anderen Wort enden.

Dazu wird für jedes der beiden Wörter geprüft, ob der Index, ab dem das Wort dem anderen gleicht, größer als 0 ist. Ist das bei einem der beiden Wörter nicht der Fall, bedeutet das, dass das andere Wort mit diesem endet. Wir prüfen also:

$$i_{1\to 2} > 0 \text{ und } i_{2\to 1} > 0$$

[1] Sind l_1 und l_2 die Längen der Wörter, so gilt also $l_1 - i_{1\to 2} = l_2 - i_{2\to 1}$.

Werden alle drei Regeln erfüllt, wird das aktuell betrachtete Wortpaar als passend im Sinne der Aufgabenstellung ausgegeben.

J1.2 Optimierung

Es ist sinnvoll, die maßgebliche Vokalgruppe eines Wortes nicht für jeden Vergleich des Wortes mit einem anderen Wort neu zu berechnen. Effizienter ist es, zu Beginn des Programms ein Mal die maßgebliche Vokalgruppe jedes Wortes vorzuberechnen und zu speichern, sodass später bei den Vergleichen von zwei Wörtern darauf zurückgegriffen werden kann.

J1.3 Sonderfälle

Die Aufgabenstellung legt nicht fest, welche Buchstaben als Vokale gelten und daher Vokalgruppen bilden können. Wir betrachten selbstverständlich die Buchstaben a, e, i, o, u, aber auch ä, ö und ü als Vokale. Sonderfälle wie qu oder y als Vokale zu zählen wäre ebenfalls denkbar, wir entscheiden uns aber dagegen.

Abkürzungen wie LKW (Lastkraftwagen) stellen einen weiteren Sonderfall dar. In diesen Wörtern sind keine Vokale und dementsprechend sind die Regeln der Reimerei nicht auf sie anwendbar. Wir sortieren diese Wörter daher einfach aus.

Bei Wörtern mit Bindestrich, wie zum Beispiel U-Bahn, sind ebenfalls mehrere Vorgehen denkbar. Wie entscheiden uns dafür, dieses Wort nach den ganz normalen Regeln zu behandeln, sodass das „U“ in diesem Fall die maßgebliche Vokalgruppe ist.

J1.4 Laufzeit

Sei n die Anzahl der Wörter und m die Länge des längsten Wortes.

Da alle Paare von Wörtern geprüft werden müssen, benötigt das eine Laufzeit von $\mathcal{O}(n^2)$. Das Vergleichen der Wortenden der zwei aktuell betrachteten Wörtern hat eine Laufzeit von $\mathcal{O}(m)$, da spätestens dann der Vergleich endet. Um die maßgebliche Vokalgruppe eines Wortes zu berechnen, wird das gesamte Wort durchlaufen, was wieder eine Laufzeit von $\mathcal{O}(m)$ hat.

Für das Vorberechnen der maßgeblichen Vokalgruppen und erst darauf folgende Vergleichen der Wörter wird damit eine Laufzeit von $\mathcal{O}(n^2 \cdot m)$ erreicht, denn das Vorberechnen hat eine Laufzeit von $\mathcal{O}(n \cdot m)$ und das folgende Vergleichen eine Laufzeit von $\mathcal{O}(n^2 \cdot m)$.

Ohne die Vorberechnung würde die asymptotische Laufzeit $\mathcal{O}(n^2 \cdot (2 \cdot m + m)) = \mathcal{O}(n^2 \cdot m)$ betragen, denn in jedem Vergleich müsste für zwei Wörter die maßgebliche Vokalgruppe berechnet und diese Vokalgruppen dann verglichen werden. Die asymptotische Laufzeit wäre also nicht schlechter, das Programm jedoch tatsächlich langsamer, da mehr Instruktionen in jedem Schleifendurchlauf durchgeführt würden.

J1.5 Beispiele

reimerei0.txt

bemühen	glühen
biene	hygiene
biene	schiene
hygiene	schiene
knecht	recht

reimerei1.txt

bildnis	wildnis
brote	note

reimerei2.txt

epsilon	ypsilon

reimerei3.txt

absender	kalender	ansage	frage	ansage	garage
bahn	zahn	bank	dank	baum	raum
bein	wein	bier	tier	bild	schild
bitte	mitte	butter	großmutter	butter	mutter
dame	name	dezember	november	dezember	september
drucker	zucker	durst	wurst	ermäßigung	kündigung
ermäßigung	reinigung	fest	test	feuer	steuer
fisch	tisch	flasche	tasche	frage	garage
fuß	gruß	gas	glas	glück	stück
gleis	kreis	gleis	preis	gleis	reis
gruppe	suppe	hand	land	hand	strand
hose	rose	hund	mund	kündigung	reinigung
kanne	panne	kasse	klasse	kasse	tasse
kassette	kette	kassette	toilette	keller	teller
kette	toilette	kind	rind	kind	wind
klasse	tasse	kopf	topf	kreis	preis
kunde	stunde	land	strand	lohn	sohn
magen	wagen	nachmittag	vormittag	november	september
platz	satz	rind	wind	rock	stock
s-bahn	zahn	sache	sprache	sekunde	stunde
see	tee				

Anmerkung: `reimerei3.txt` war in dem hochgeladenen zip-Archiv leicht anders als bei den einzeln unter Junioraufgabe 1 gelisteten Dateien. Je nachdem, welche Version verwendet wurde, ist (see, tee) bei den Reimpaaren enthalten oder nicht.

J1.6 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- [−1] **Regeln werden nicht eingehalten**
  Es müssen alle drei Regeln zur Überprüfung eines Reimpaares beachtet werden:
  - Ab der maßgeblichen Vokalgruppe enthalten beide Wörter dieselben Buchstaben in derselben Reihenfolge.
  - Der Teil ab der maßgeblichen Vokalgruppe umfasst jeweils mindestens die Hälfte der Buchstaben.
  - Keines der beiden Wörter darf mit dem kompletten anderen Wort enden.
- [−1] **Lösungsverfahren anderweitig fehlerhaft**
  Insbesondere muss darauf geachtet werden, dass die maßgebliche Vokal*gruppe* korrekt bestimmt wird. Sofern es mindestens zwei Vokalgruppen in einem Wort gibt, muss die vorletzte als maßgebliche verwendet werden.

  Die Buchstaben a, e, i, o und u sollten auf jeden Fall als Vokale gezählt werden. Umlaute nicht als Vokale zu zählen, führt nicht zu Punktabzug. Sonderfälle wie qu oder y als Vokale zu zählen, ist erlaubt und richtig, sofern dies in der Dokumentation entsprechend begründet wurde.

  Wie Abkürzungen wie LKW behandelt werden, ist gänzlich freigestellt, solange die anderen Wortpaare korrekt geprüft werden.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Alle Reimpaare sollten erkennbar und voneinander getrennt ausgegeben werden. Es ist dabei in Ordnung, wenn dasselbe Paar zwei Mal, aber in unterschiedliche Richtungen ausgegeben wird (z. B. `recht - knecht`, `knecht - recht`). Die Liste muss allerdings angemessen sortiert sein (z. B. lexikographisch). Ist dies nicht der Fall, darf hier ein Punkt abgezogen werden.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu allen vorgegebenen Beispielen (`reimerei0.txt` bis `reimerei3.txt`) enthalten.

Junioraufgabe 2: Container

J2.1 Lösungsidee

In dieser Aufgabe gibt es mehrere unterschiedlich schwere Container, von denen wir allerdings nicht wissen, wie schwer diese sind. Wir haben lediglich als Information eine Liste von paarweisen Wägungen gegeben, die jeweils angeben, ob ein Container schwerer als ein anderer ist. Weiterhin ist bekannt, dass jeder Container mindestens einmal zusammen mit einem anderen Container auf der Containerpaarwaage war. Nun soll man herausfinden, ob man ausgehend von diesen Informationen sicher sagen kann, welcher Container der schwerste ist.

Zur Veranschaulichung bietet es sich an, diese Wägungen wie in Abbildung J2.1 in einem gerichteten azyklischen Graphen[2] zu modellieren, wobei für jeden Container ein Knoten reserviert wird. Hierbei bedeutet ein Pfeil von einem Container A zu einem Container B, dass A mit B auf der Waage war und A schwerer als B ist.

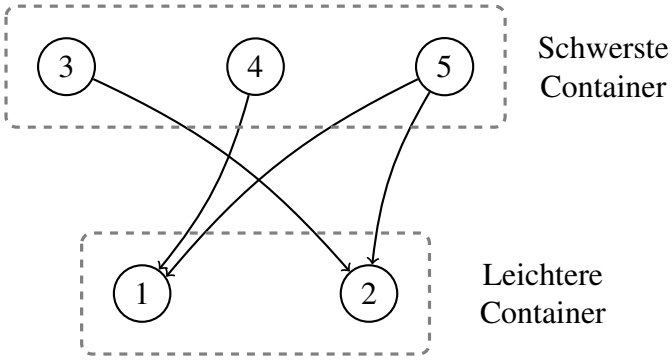


Abbildung J2.1: Darstellung der Abhängigkeiten im Beispiel `container0.txt`

Zunächst fällt einem auf, dass in diesem Beispiel die Container 1 und 2 nicht die schwersten sein können, denn Container 1 ist leichter als die Container 4 und 5 und Container 2 ist leichter als die Container 3 und 5. Allgemein gilt, dass ein Container, der bei einer Wägung leichter als irgendein anderer Container ist, auf jeden Fall nicht der schwerste sein kann.

Es gilt also, dass es zu einem schwersten Container nie einen schwereren geben darf. Anhand der gegebenen Vergleiche kann man alle Container sammeln, die diese Eigenschaft erfüllen, nämlich dass es jeweils keinen als schwerer bekannten Container gibt. Wir bezeichnen diese Container im Folgenden als *Kandidaten*. Hier gibt es zwei Fälle:

- Entweder es gibt nur genau einen möglichen Kandidaten. Dann muss dieser Container der schwerste sein, denn jeder andere Container war zumindest einmal der leichtere bei einer Wägung.
- Andernfalls gibt es mehrere Kandidaten. Hier ist nicht eindeutig, welcher dieser Kandidaten der schwerste Container ist. Ein Beispiel zeigt bereits die Abbildung J2.1: Die Container 3, 4 und 5 sind bei keiner Wägung leichter als ein anderer Container. Allerdings müssen diese nach Aufgabenstellung unterschiedlich schwer sein. Nachdem aber zwischen den Containern 3, 4 und 5 keine Wägungen vorgenommen wurden, kann man nicht wissen, welcher der schwerste ist.

[2] Für eine Erklärung, was ein gerichteter Graph ist, siehe `https://de.wikipedia.org/wiki/Gerichteter_Graph`. Ein azyklischer Graph ist ein Graph ohne Zyklen. Für mehr Informationen zu Zyklen siehe `https://de.wikipedia.org/wiki/Graph_(Graphentheorie)#Teilgraphen,_Wege_und_Zyklen`.

Algorithmische Umsetzung

Ausgehend von den vorherigen Überlegungen genügt es somit zur Lösung der gestellten Aufgabe, die Container zu finden, die in keiner Wägung leichter waren. Effizient lässt sich das beispielsweise wie in Algorithmus 1 realisieren, der als Eingaben die Container (containers) sowie die Wägungen (relations, jeweils Paare der Form $(\text{larger}, \text{smaller})$) entgegennimmt.

Algorithmus 1 Bestimmen möglicher Kandidaten mit größtem Gewicht

```
procedure FINDCANDIDATES(containers, relations)
    candidates ← containers
    for (larger, smaller) in relations do
        candidates ← candidates \ {smaller}
    end for
    return candidates
end procedure
```

Die Idee ist hier, dass wir zunächst davon ausgehen, dass noch keine Wägungen durchgeführt wurden. Somit kann jeder Container der schwerste Container sein und gilt als Kandidat. Danach wird jede Wägung überprüft: Ist hierbei ein Container leichter als ein anderer, ist er kein möglicher Kandidat mehr und wird aus der Liste der Kandidaten, falls noch darin vorhanden, entfernt. Falls anschließend nur noch ein Element in der Kandidatenliste enthalten ist, so ist eindeutig, welcher Container der schwerste ist. Nämlich eben dieser letzte Kandidat, der noch in der Liste ist. Falls noch mehrere Elemente in der Kandidatenliste enthalten sind, so ist nicht eindeutig feststellbar, welcher Container der schwerste Container ist.

Laufzeit

Als Eingaben des Algorithmus haben wir die Menge der Container (containers) sowie die gegebenen Vergleiche (relations). Der vorgestellte Algorithmus besteht im Wesentlichen aus zwei Phasen:

1. Zunächst nehmen wir an, dass alle Container potenziell die schwersten sein können und erstellen eine entsprechende Menge. Dann entfernen wir für jeden gegebenen Vergleich maximal ein Element von den Kandidaten.

   Die Laufzeit hängt also hierfür vor allem davon ab, welche Datenstruktur man für die Verwaltung einer Menge M verwendet. Die bekanntesten sind Hashtabellen[3], balancierte Binärbäume[4] oder Bitarrays[5]. Im Regelfall weist die Hashtabelle für die üblichen Operationen wie Finden, Einfügen oder Löschen eines Elementes eine konstante Laufzeit auf, diese kann aber im seltenen Fall auch $\mathcal{O}(|M|)$ sein. Ein balancierter Suchbaum hat für diese Operationen stets eine Laufzeitkomplexität von $\mathcal{O}(\log(|M|))$. Ein Bitarray (also ein Array von Bits, dessen i-tes Bit angibt, ob das mit i indizierte Element in M enthalten ist) hat garantiert konstante Laufzeit, setzt allerdings voraus, dass man zu jedem Element schnell einen zugehörigen Index finden kann. Dies trifft beispielsweise dann zu, wenn die Container bereits aufsteigend mit den natürlichen Zahlen $(1, 2, 3, \ldots)$ benannt worden

[3] `https://de.wikipedia.org/wiki/Hashtabelle`
[4] `https://de.wikipedia.org/wiki/Balancierter_Baum`
[5] `https://de.wikipedia.org/wiki/Bitkette`

sind. Eine solche Nummerierung gibt ein Eingabeformat wie das der Beispieleingaben bereits vor.

Wir nehmen im Folgenden an, dass wir die üblichen Operationen auf Mengen in konstanter Zeit absolvieren können. Damit erhalten wir für das Erzeugen der Menge einen Laufzeitaufwand von $\mathcal{O}(|\text{containers}|)$ und für die Entfernung der Elemente einen Laufzeitaufwand von $\mathcal{O}(|\text{relations}|)$.

2. Anschließend muss man noch überprüfen, ob die Menge der Kandidaten die Mächtigkeit 1 hat, also nur noch ein Element enthalten ist und das Ergebnis eindeutig ist. Andernfalls sollte die Mächtigkeit ggf. ausgegeben werden. Das geschieht in konstanter Zeit.

Zusammengefasst erhalten wir also eine Laufzeit von $\mathcal{O}(|\text{containers}| + |\text{relations}|)$.

Anmerkung: Wir gehen in unserer Lösung davon aus, dass alle gegebenen paarweisen Wägungen konsistente Abhängigkeiten verursachen. Beispielsweise darf für zwei Container A und B nicht sowohl „A schwerer als B“ als auch „B schwerer als A“ gewogen worden sein. In diesem Fall hätte man in dem Abhängigkeitsgraphen einen Zyklus, den man erkennen und damit dem Nutzer eine entsprechende Warnung ausgeben könnte.

J2.2 Ein anderer Lösungsweg

Anstatt die Eigenschaft auszunutzen, dass ein schwerster Container in einer Wägung nie der leichtere ist, kann man die Eigenschaft auch expliziter überprüfen: Wir verwenden hierbei, dass Gewichtsrelationen transitiv sind, d. h. wenn für drei Container A, B und C sowohl „A größer als B“ als auch „B größer als C“ gemessen wird, muss auch „A größer als C“ gelten. Können wir also mithilfe der transitiven Eigenschaft für einen Container zeigen, dass dieser schwerer als alle anderen ist, haben wir den schwersten Container gefunden.

Algorithmus 2 Finden aller definitiv leichteren Container mittels Tiefensuche

```
procedure FINDSMALLER(curr, relations, visited)
    if curr ∈ visited then
        return visited
    end if
    for neigh ∈ {x | (curr, x) ∈ relations} do
        visited ← FINDSMALLER(neigh, relations, visited)
        visited ← visited ∪ {neigh}
    end for
    return visited
end procedure
```

Eine mögliche Realisierung mittels Tiefensuche[6] beschreibt Algorithmus 2. Wir betrachten zunächst Container, von denen man ausgehend von der Information in relations direkt weiß, dass diese kleiner als der aktuelle Container sind. Für jedes kleinere Element kann man nun aufgrund der transitiven Eigenschaft der Größer-Relation rekursiv weitergehen und weitere Container sammeln. Bereits besuchte Container im rekursiven Aufruf lassen sich überspringen, um dadurch Laufzeit zu sparen. Wenn man für einen Container alle anderen Container in visited gesammelt hat, weiß man, dass dieser der schwerste ist. Die Laufzeitkomplexität der Tiefensuche

[6] `https://de.wikipedia.org/wiki/Tiefensuche`

ist bekanntlich $\mathcal{O}(|\text{containers}| + |\text{relations}|)$, unter der Voraussetzung, dass die Datenstruktur relations die leichteren Container zu einem Container effizient herausfinden kann.

Dieses Prozedere müssen wir für alle Container wiederholen. Im schlimmsten Fall hat man also eine Laufzeitkomplexität von $\mathcal{O}(|\text{containers}| \cdot (|\text{containers}| + |\text{relations}|))$, was zwar eine Verschlechterung gegenüber der anderen vorgestellten Lösungsidee bedeutet, aber trotzdem noch eine akzeptable Laufzeit ist.

J2.3 Beispiele

Die folgende Tabelle zeigt die Ergebnisse für die vorgegebenen Beispiele. Es ist jeweils angegeben, welche Container potenziell die schwersten nach dem Kriterium aus der Lösungsidee in Abschnitt J2.1 sind, sowie, ob man eindeutig den schwersten Container identifizieren konnte (es also nur einen Kandidaten gibt).

Beispiel	**Mögliche Kandidaten**	**Eindeutig?**
`container0.txt`	$\{3,4,5\}$	nein
`container1.txt`	$\{4\}$	ja
`container2.txt`	$\{1,3\}$	nein
`container3.txt`	$\{5,7\}$	nein
`container4.txt`	$\{5\}$	ja

Für die Eingaben `container0.txt` und `container1.txt` lassen sich die Ergebnisse leicht exemplarisch nachvollziehen. Wie bereits in Abbildung J2.1 dargestellt, kann man die Container 1 und 2 als schwerste Container ausschließen, da diese kleiner sind als andere. Zwischen den Containern 3, 4 und 5 ist allerdings keine Relation gegeben. Jeder von diesen könnte der schwerste sein, wodurch wir kein eindeutiges Ergebnis erhalten.

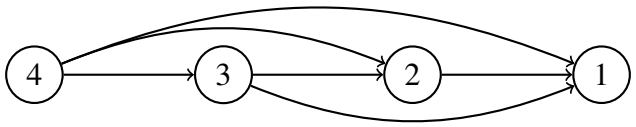


Abbildung J2.2: Darstellung der Abhängigkeiten im Beispiel `container1.txt`

Bei der zweiten Eingabe (Abbildung J2.2) sieht man hingegen, dass es für alle Container, bis auf einen, einen anderen Container gibt, der schwerer ist. Insbesondere wurde hier das Gewicht jedes Containers mit dem jedes anderen verglichen. Container 4 ist also eindeutig der schwerste.

J2.4 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- [−1] **Eindeutigkeit nicht korrekt festgestellt**
  Das Lösungsverfahren muss erkennen können, ob es mehrere Container gibt, die die schwersten sein können, und dann feststellen, dass die Lösung nicht eindeutig ist.
- [−1] **Lösungsverfahren anderweitig fehlerhaft**
  Insbesondere soll die Menge der Kandidaten für den schwersten Container – ob eindeutig oder nicht – keinen falschen Container enthalten.

  Falls schon bei *Eindeutigkeit nicht korrekt festgestellt* ein Punkt abgezogen wurde, so soll hier nicht erneut ein Punkt abgezogen werden.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Es muss ausgegeben werden, ob der schwerste Container eindeutig bestimmt werden kann, und, wenn ja, welcher dies ist. Lediglich auszugeben, ob der schwerste Container eindeutig bestimmt werden kann, genügt nicht. Falls die Lösung nicht eindeutig ist, so muss nicht die Menge an Kandidaten ausgegeben werden, sondern es genügt ein Hinweis, dass die Lösung nicht eindeutig ist.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens 4 der vorgegebenen Beispiele (`container0.txt` bis `container4.txt`) enthalten.

Aufgabe 1: Störung

Im Kern verlangt die Aufgabe einen Algorithmus, der einen gegebenen Suchtext T nach allen Vorkommen eines gegebenen Musters P durchsucht. Die Besonderheit ist, dass P Unterstriche als Platzhalter enthalten kann, die jeweils für ein beliebiges Wort stehen. Damit ist die Aufgabe verwandt mit dem *String-Matching-Problem mit Don't Cares* und kann auch darauf reduziert werden.

Für eine gute Lösung müssen wir einige Definitions- und Modellierungsentscheidungen treffen, was im folgenden Abschnitt geschieht. Da zum Lösen der Beispieleingaben für das eigentliche String-Matching bereits eine einfache Lösung ausreichend ist, liegt die Hauptschwierigkeit der Aufgabe darin, die Eingabe korrekt zu verarbeiten und einige Feinheiten zu berücksichtigen (siehe auch Abschnitt 1.2). Die einfache Lösung aus Abschnitt 1.3 lässt sich recht einfach durch Hashing optimieren, was wir in Abschnitt 1.4 erklären. Schließlich gehen wir in Abschnitt 1.5 kurz auf weitere Lösungsansätze ein.

1.1 Modellierung

Strings und Wörter

Die Eingabe können wir als zwei Strings $T[1:n]$ (Suchtext) und $P[1:m]$ (Muster) mit Zeichen über einem Alphabet Σ darstellen. Als dieses Alphabet kann z. B. die Menge aller ASCII-Zeichen gewählt werden. Da wir im Muster P zusätzlich Platzhalter erlauben, fordern wir $P[i] \in \Sigma \cup \{\_\}$ für $1 \leq i \leq m$ und $T[i] \in \Sigma$ für $1 \leq i \leq n$, wobei $\_ \notin \Sigma$ ausschließlich als Platzhalter dient.

Im nächsten Schritt müssen wir uns überlegen, was als *Vorkommen* von P in T zählt. Hier lässt die Aufgabe etwas Interpretationsspielraum. Ist zum Beispiel

```
das _ mir _ _ _ vor
```

das Muster, so stellen wir fest, dass nur

```
Das kommt mir gar nicht richtig vor
```

im Beispieltext „Alice im Wunderland“ vorkommt. Wir entscheiden uns also, nicht zwischen *Groß- und Kleinschreibung* zu unterscheiden.

Anknüpfend an die Frage, was als Vorkommen zählt, müssen wir genauer überlegen, wie Platzhalter ersetzt werden. In der Aufgabenstellung heißt es, Trudi habe „einen Weg gefunden, Lücken in die Sätze zu reißen und ganze Wörter verschwinden zu lassen.“ Daher nehmen wir an, dass ein Platzhalter _ für genau ein beliebiges Wort steht. Hier sind einige Feinheiten zu beachten:

- *Satzzeichen:* Wir zählen Satzzeichen nicht als Teil eines Wortes. Diese können also niemals durch einen Platzhalter „verschluckt“ werden und stehen im Muster genau wie im Suchtext.
- *Bindestriche:* Bindestriche sind Teil eines Wortes, sofern sie nicht am Zeilenende auftreten. Ist Letzteres der Fall, interpretieren wir sie als „Trennstriche“ und fügen die Wortteile zu einem Wort zusammen.

Um zwischen Satzzeichen und Zeichen, die Teil eines Wortes sind, zu unterscheiden, definieren wir ein *Wort* als Folge von Zeichen über dem Alphabet

$$\Sigma_\alpha := \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}, \mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}, \ddot{\mathtt{a}}, \ddot{\mathtt{o}}, \ddot{\mathtt{u}}, \ddot{\mathtt{A}}, \ddot{\mathtt{O}}, \ddot{\mathtt{U}}, \mathtt{ß}, -\}.$$

Dem liegt natürlich die Annahme zugrunde, dass wir es hier mit deutschen Texten zu tun haben. Für andere Sprachen müsste Σ_α ggf. angepasst werden.

Muster und Suchtext direkt als Strings zu speichern, ist die wohl einfachste Modellierung und in den meisten Programmiersprachen schnell umgesetzt. Allerdings sind String-Matching-Algorithmen in der Literatur meist so beschrieben, dass der Platzhalter _ einem einzelnen Zeichen entspricht. In unserem Fall soll _ jedoch ein ganzes Wort ersetzen, was wir dann spätestens bei der Implementierung zu beachten haben.

Liste von Tokens

Alternativ können wir Suchtext und Muster jeweils in eine Liste von *Tokens* zerlegen. Nach unseren bisherigen Überlegungen erscheint es sinnvoll, vier Typen von Tokens einzuführen: Platzhalter (`PLACEHOLDER`), Leerzeichen (`SPACE`), Wort (`WORD`) und Satzzeichen (`PUNCTUATION`). Dabei sind die letzten zwei Tokens parametrisiert, d. h. sie beinhalten das Wort bzw. das Satzzeichen, das sie repräsentieren, als String. Muster und Suchtext lassen sich dann als Folge von Tokens auffassen. Das könnte für den Satz „Ich muß _ Clara _!“ zum Beispiel so aussehen:

```
[WORD(Ich), SPACE, WORD(muß), SPACE, PLACEHOLDER, SPACE, WORD(Clara),
SPACE, PLACEHOLDER, PUNCTUATION(!)]
```

Für uns Menschen ist eine solche Liste von Tokens eher mühsam zu lesen. Schreibt man jedoch ein Computerprogramm, erweist sich diese schnell als Segen. Die Einführung von Tokens und die Transformation eines Strings in eine Folge von Tokens ist so verbreitet, dass es dafür einen Namen gibt: Das *Lexen*. Ein Programm für diese Transformation heißt *Lexer*. Quellcode in den meisten Programmiersprachen wird zunächst durch einen Lexer (oft Teil des Compilers) in eine Folge von Tokens überführt.

In vielen Programmiersprachen, insbesondere – aber nicht nur – objektorientierten Sprachen, lassen sich diese Tokens und Texte (Letztere z. B. als Listen von Tokens) sehr gut modellieren. In unseren Experimenten hat sich diese Modellierung im Vergleich zur Darstellung durch reine Strings als eleganter bei der Implementierung erwiesen.

1.2 Lexer und Token-Matching

Lexen der Eingabe

Nun geht es darum, die Eingabe (sowohl den Suchtext T als auch das Muster P) in eine Liste von Tokens zu überführen. Hierzu definieren wir eine rekursive Funktion, die den String Stück für Stück „frisst“, bis dieser vollständig gelext ist. Dabei wird immer anhand der ersten Zeichen des aktuellen Strings bestimmt, welches Token als nächstes auftritt. Dieses Token wird vollständig eingelesen, um die Funktion anschließend rekursiv mit dem Rest-String aufzurufen. Eine formale Beschreibung dieses Vorgehens ist in Algorithmus 3 beschrieben.

Zur Notation: Mit `[]` bezeichnen wir die leere Liste, mit :: den *Cons-Operator*, der ein Token an den Anfang einer Liste anfügt, z. B.

$$\texttt{SPACE} :: [\texttt{PLACEHOLDER}, \texttt{WORD(BWINF)}] = [\texttt{SPACE}, \texttt{PLACEHOLDER}, \texttt{WORD(BWINF)}],$$

und $\circ$ bezeichnet die *Konkatenation* zweier Strings, z. B. 'H-' $\circ$ 'Milch' = 'H-Milch'.

Algorithmus 3 Lexer für „Störung“

```
1:  procedure LEX(U = U[1 : l])
2:      if U = '' then                                          ▷ leerer String
3:          return []                                           ▷ leere Liste
4:      else
5:          if U[1] = ' ' ∨ U[1] = '\n' then     ▷ Leerzeichen und Zeilenumbrüche sind SPACE
6:              return SPACE :: LEX(U[2 : l])
7:          else if U[1] = '_' then
8:              return PLACEHOLDER :: LEX(U[2 : l])
9:          else if U[1] ∈ Σ \ Σ_α then
10:             return PUNCTUATION(U[1]) :: LEX(U[2 : l])
11:         else                                                ▷ U[1] ∈ Σ_α
12:             i ← 1
13:             W ← ''                                          ▷ leerer String
14:             while true do
15:                 if U[i] = '-' ∧ U[i+1] = '\n' then   ▷ Trennstrich bei Zeilenumbruch
16:                     i ← i + 2
17:                 else if U[i] ∈ Σ_α then
18:                     W ← W ∘ U[i]                  ▷ Füge Zeichen U[i] an String W an
19:                     i ← i + 1
20:                 else
21:                     break                                   ▷ Ende des Worts
22:                 end if
23:             end while
24:             return WORD(W) :: LEX(U[i : l])
25:         end if
26:     end if
27: end procedure
```

Gleichheit auf Tokens

Nachdem wir sowohl Muster als auch Suchtext in eine Liste von Tokens überführt haben, können wir mit dem Matching beginnnen und versuchen, das Muster im Suchtext zu finden. Zunächst stellt sich die Frage, wann zwei Tokens als *gleich* angesehen werden. Wir haben diese Frage schon in Abschnitt 1.1 beantwortet: Zunächst sollen zwei `WORD`-Tokens gleich sein, wenn sie bis auf Groß- und Kleinschreibung den gleichen String als Parameter tragen. Ein `PLACEHOLDER`-Token kann mit einem beliebigen `WORD`-Token übereinstimmen. Schließlich passen Satzzeichen, d. h. `PUNCTUATION`-Tokens, nur zu `PUNCTUATION` mit dem gleichen Parameter, und `SPACE`-Tokens nur zu `SPACE`-Tokens. Übersichtlich dargestellt sind diese Definitionen in Tabelle 1.

[7] Wir erinnern uns an die Annahme, dass $\_ \notin \Sigma$, d. h. der Platzhalter kommt nur im Muster vor, niemals aber im Suchtext.

	WORD(T)	PUNCTUATION(T)	SPACE	PLACEHOLDER
WORD(S)	wenn $S = T$	Nein	Nein	Ja
PUNCTUATION(S)	Nein	wenn $S = T$	Nein	Nein
SPACE	Nein	Nein	Ja	Nein
PLACEHOLDER	Ja	Nein	Nein	N/A[7]

Tabelle 1: Definition von Gleichheit auf Tokens

1.3 Einfache Lösung

Algorithmus

Wie beim klassischen String-Matching-Problem können wir auch hier das Muster an jeder möglichen Position im Suchtext anlegen und die Tokens vergleichen. Die korrekte Interpretation von Platzhaltern ist dabei dank unserer Definition der Gleichheit auf Tokens automatisch gewährleistet. Sei $T = [T_1, \ldots, T_n]$ die Token-Liste des Suchtexts und $P = [P_1, \ldots, P_m]$ die Token-Liste des Musters. Algorithmus 4 beschreibt ein solches einfaches Vorgehen in Pseudocode. Hierbei ist TOKENS-EQUAL der Gleichheitstest für zwei Tokens gemäß Tabelle 1.

Algorithmus 4 Muster im Suchtext finden

```
 1: procedure FIND-PATTERN(T = [T_1, ..., T_n], P = [P_1, ..., P_m])
 2:     for i = 1 to n − m + 1 do
 3:         matches ← true
 4:         for j = 1 to m do
 5:             if not TOKENS-EQUAL(T_{i+j−1}, P_j) then
 6:                 matches ← false
 7:                 break
 8:             end if
 9:         end for
10:         if matches then
11:             print [T_i, T_{i+1}, ..., T_{i+m−1}]          ▷ „pretty print“
12:         end if
13:     end for
14: end procedure
```

Laufzeitbetrachtung

Zunächst führen wir für ein Token S die Notation $|S|$ ein: Damit soll die Anzahl der Zeichen, die das Token im ursprünglich Text lang ist, gemeint sein. Ist S vom Typ SPACE, PLACEHOLDER oder PUNCTUATION, gilt also $|S| = 1$. Für WORD ist S die Länge des Worts, z. B. $|S| = 5$, falls $S = \texttt{WORD}(\text{BWINF})$ gilt.

Durch die zwei **for**-Schleifen und die Logik innerhalb dieser erkennen wir, dass sich die Laufzeit von Algorithmus 4 proportional zu $\sum_{i=1}^{n-m+1} \sum_{j=1}^{m} \left(|T_{i+j-1}| + |P_j| \right)$ nach oben abschätzen lässt. Dabei ist n die Länge der Token-Liste T des Suchtextes und m die Länge der Token-Liste

P des Musters. Dabei kommt der Term $|T_{i+j-1}| + |P_j|$ vom Vergleichen der Strings, mit denen T_{i+j-1} und P_j parametrisiert sind.

Man beachte, dass dies gewissermaßen eine grobe Überschätzung ist: Wenn Suchtext und Muster nicht besonders ungünstig sind, wird in den meisten Fällen schon das erste Token nicht übereinstimmen und wegen der **break**-Instruktion der übrige Teil des Musters gar nicht erst an der jeweiligen Stelle im Suchtext abgeglichen. Für eine einfache Analyse, die auch den Worst-Case abdeckt, verwenden wir jedoch diese Überschätzung. Ein paar Umformungen bringen den Ausdruck auf eine für uns günstigere Form:

$$\sum_{i=1}^{n-m+1} \sum_{j=1}^{m} \left(|T_{i+j-1}| + |P_j| \right) = \sum_{j=1}^{m} \sum_{i=1}^{n-m+1} |T_{i+j-1}| + \sum_{i=1}^{n-m+1} \sum_{j=1}^{m} |P_j| \tag{1.1}$$

Wir bemerken, dass $\sum_{i=1}^{n-m+1} |T_{i+j-1}| \leq \sum_{i=1}^{n} |T_i|$ gilt, da die Summe jedes Token höchstens einmal betrachtet. Somit erhalten wir, dass die obige Summe durch

$$m \sum_{i=1}^{n} |T_i| + n \sum_{j=1}^{m} |P_j| \tag{1.2}$$

beschränkt ist. Ist M die Länge des Musters (in Zeichen) und N die Länge des Suchtexts, so ist die Gesamtlaufzeit in $\mathcal{O}(mN + nM)$, was wiederum in $\mathcal{O}(NM)$ liegt, wobei die Eingabelänge nur $\mathcal{O}(N + M)$ ist. Hier gibt es also Verbesserungspotential, wie in Abschnitt 1.4 und 1.5 näher betrachtet.

Für die Beispieleingaben ist diese einfache Lösung schnell genug. Das liegt unter anderem daran, dass stets sehr kurze Muster verwendet werden. Tatsächlich gilt $M \leq 30$ für alle Beispieleingaben. Ist M unter einer festen Größe, wie es für die meisten Sätze der Fall sein dürfte, so haben wir $\mathcal{O}(M) = \mathcal{O}(1)$ und daher eine Laufzeit von $\mathcal{O}(N)$, was asymptotisch optimal ist. Das ist natürlich nur die halbe Wahrheit: Selbstverständlich wirkt sich der größere konstante Faktor in der Praxis auf die reale Laufzeit aus.

1.4 Optimierung durch Hashing

Eine mögliche und recht einfache Optimierung der einfachen Lösung besteht darin, die Vergleiche von Tokens effizienter zu machen. In den meisten Fällen laufen diese bereits in Zeit $\mathcal{O}(1)$, jedoch nicht, wenn zwei `WORD`-Tokens miteinander zu vergleichen sind. Statt hier die beiden Parameter per String-Matching naiv miteinander zu vergleichen, können wir zunächst alle Wörter in der Eingabe *hashen*. In den Tokens speichern wir anschließend nur noch die Hashwerte, welche sich – solange wir nicht Milliarden von verschiedenen Wörtern haben[8] – als Ganzzahlen darstellen und auf den meisten Maschinen in Zeit $\mathcal{O}(1)$ vergleichen lassen.

Die Gesamtlaufzeit reduziert sich durch diese Optimierung auf $\mathcal{O}(mn)$, wobei m und n die Anzahlen der Tokens in Muster und Suchtext sind. Solange es keine beliebig langen Wörter gibt, bringt diese Optimierung zwar keine echte asymptotische Verbesserung der Laufzeit, in der Praxis aber einen nützlichen konstanten Faktor.

[8] Das wird es in kaum einer Sprache geben…

1.5 Alternative Lösungen

Während es für das *String-Matching-Problem ohne Don't Cares* sehr bekannte Algorithmen gibt, die in Linearzeit laufen, z. B. den Algorithmus von Knuth-Morris-Pratt (KMP), ist die Situation für *String-Matching mit Don't Cares* etwas komplizierter.

Verbesserungen der asymptotischen Laufzeit können z. B. durch String-Matching mit deterministischen endlichen Automaten (DFA) erreicht werden. Wir bemerken außerdem, dass es nicht schwierig ist, für ein Muster mit Platzhaltern einen regulären Ausdruck aufzuschreiben, der genau die Strings beschreibt, die auf das Muster mit Platzhaltern passen. Dies ist kein Zufall, denn DFA-erkennbare Sprachen und reguläre Sprachen sind das Gleiche.

Für das String-Matching-Problem mit Platzhaltern gibt es noch weitere Algorithmen, etwa basierend auf *Fingerprinting*. Interessierten mag der Artikel „Efficient Pattern-Matching with Don't Cares“[9] von Adam Kalai als guter Einstiegspunkt in die Literatur dienen. Darin wird ein randomisierter Algorithmus mit Laufzeit in $\mathcal{O}(n \log m)$ beschrieben.

1.6 Beispiele

stoerung0.txt

```
Muster: Das kommt _ _ nicht _ vor
Muster mit einer Länge von 25 Zeichen in 13 Tokens gelext
Suchtext mit einer Länge von 159416 Zeichen in 62747 Tokens gelext

Das kommt mir gar nicht richtig vor

Matching beendet in 28 ms
```

stoerung1.txt

```
Muster: Ich muß _ Clara _
Muster mit einer Länge von 17 Zeichen in 9 Tokens gelext
Suchtext mit einer Länge von 159416 Zeichen in 62747 Tokens gelext

Ich muß in Clara verwandelt
Ich muß doch Clara sein

Matching beendet in 28 ms
```

stoerung2.txt

Wie sich herausstellt, erscheint der Ausdruck `Fressen Katzen gern Spatzen` im Suchtext zweimal. Unser Programm gibt ihn daher auch zweimal aus. Es wäre der Aufgabe durchaus ebenso angemessen, das Programm so zu erweitern, dass jeder gefundene Ausdruck nur einmal ausgegeben wird.

[9] `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=06253463543a0e95c006cac321df7b6598a32f24`

```
Muster: fressen _ gern _
Muster mit einer Länge von 16 Zeichen in 7 Tokens gelext
Suchtext mit einer Länge von 159416 Zeichen in 62747 Tokens gelext

Fressen Katzen gern Spatzen
Fressen Katzen gern Spatzen
Fressen Spatzen gern Katzen

Matching beendet in 28 ms
```

stoerung3.txt

```
Muster: das _ fing _
Muster mit einer Länge von 12 Zeichen in 7 Tokens gelext
Suchtext mit einer Länge von 159416 Zeichen in 62747 Tokens gelext

das Spiel fing an
Das Publikum fing an

Matching beendet in 33 ms
```

stoerung4.txt

```
Muster: ein _ _ tag
Muster mit einer Länge von 11 Zeichen in 7 Tokens gelext
Suchtext mit einer Länge von 159416 Zeichen in 62747 Tokens gelext

ein sehr schöner Tag

Matching beendet in 37 ms
```

stoerung5.txt

```
Muster: wollen _ so _ sein
Muster mit einer Länge von 18 Zeichen in 9 Tokens gelext
Suchtext mit einer Länge von 159416 Zeichen in 62747 Tokens gelext

Wollen Sie so gut sein

Matching beendet in 26 ms
```

1.7 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Modellierung fehlerhaft**
  Je nach Verfahren muss die Eingabe korrekt in eine Datenstruktur überführt und verwendet werden, es ist aber auch in Ordnung, wenn Ansätze verwendet werden, die den Text unverändert verwenden, insbesondere beim Matchen mit regulären Ausdrücken.

  Es muss klar werden, wie mit Groß- und Kleinbuchstaben, Satzzeichen und Bindestrichen (getrennte Wörter) umgegangen wird. Es sollte aus der Dokumentation klar werden, was als Wort zählt bzw. wodurch Platzhalter ersetzt werden können. Eine formale Grammatik ist hierzu nicht gefordert.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Das Lösungsverfahren muss *alle* Vorkommen des Musters im Suchtext finden. Platzhalter werden dabei stets durch ganze Wörter und nicht etwa durch einzelne Zeichen ersetzt.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Die Laufzeit sollte durch $\mathcal{O}(mn)$ beschränkt sein. Linearzeit $\mathcal{O}(m+n)$ wird nicht erwartet. Ein (zusätzlicher) Faktor $|\Sigma|$ ist akzeptabel, wenn der Algorithmus im Kontext der Aufgabe sinnvoll ist.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Die bloße Anzahl der Vorkommen genügt nicht als Ausgabe des Programms. Es muss ersichtlich sein, durch welche Wörter die Platzhalter jeweils ersetzt werden.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens 4 der vorgegebenen Beispiele (`stoerung0.txt` bis `stoerung5.txt`) enthalten, darunter `stoerung1.txt` oder `stoerung2.txt`.

Aufgabe 2: Verzinkt

In dieser Aufgabe soll die heterogene Struktur von Zinkkristallen reproduziert werden, welche in der Realität beispielsweise bei feuerverzinkten Oberflächen auftreten, wie in Abbildung 2.1 sichtbar wird. Diese in verschiedenen Grautönen erscheinenden Flächen entstehen dabei durch den folgenden Prozess: Zunächst muss ein Stoff, wie hier das flüssige Zink auf einer Metalloberfläche, in einen Zustand gebracht werden, der nah einem Phasenübergangspunkt ist.[10] Der Wechsel zwischen zwei Phasen kann nun nicht überall auf einmal vonstattengehen, sondern beginnt an sogenannten Keimzellen.[11] Bei der Kristallbildung aus einer Zinkschmelze sorgen die vielen kleinen Keimstellen dafür, dass sich verschieden orientierte Kristalle ausbilden, welche wiederum Flächen mit unterschiedlichen Reflexionseigenschaften erzeugen, die in verschiedenen Grautönen erscheinen.

https://de.wikipedia.org/wiki/Datei:Feuerverzinkte_Oberfläche.jpg

Abbildung 2.1: Kristalline Oberfläche eines noch wenig oxidierten, feuerverzinkten Stahlgeländers.

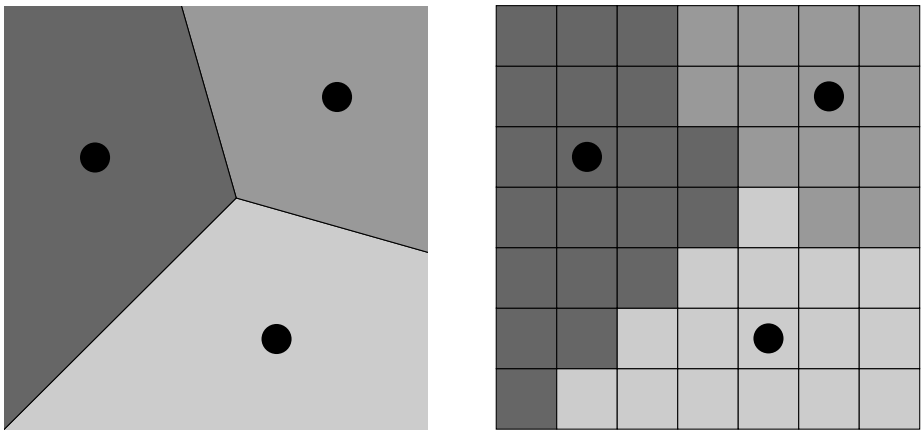
2.1 Lösungsidee

Um eben solche Kristallmuster mit dem Computer erzeugen zu können, gehen wir anhand der Vorgaben der Aufgabenstellung vor. Diese sind,

1. dass der Ort eines Kristallisationskeims als Punkt in einem zweidimensionalen Raster dargestellt wird,
2. dass ein Keim schrittweise in die vier Raster-Richtungen links, rechts, oben und unten mit jeweils eigenen Wachstumsgeschwindigkeiten wächst,
3. dass ein Kristall nicht in einen anderen hineinwachsen kann und
4. dass die jeweiligen Kristalle entsprechend ihrer Orientierung mit Grautönen dargestellt werden.

[10] Konkrete Beispiele für einen „Stoff, der nah an einem Phasenübergangspunkt ist“, sind unter anderem Wasser, das entweder gerade bei ungefähr 100 °C zwischen flüssig und gasförmig oder bei ungefähr 0 °C zwischen flüssig und fest schwebt, eine übersättigte Salzlösung, in der sich Salzkristalle ausbilden oder eben Zink bei ungefähr 420 °C, das zwischen flüssig und fest wechselt.

[11] Ein gutes Beispiel hierfür sind Wasserdampfblasen, die beim Kochen von Wasser entstehen. Es wird also nicht das gesamte Wasser ruckartig gasförmig, sondern nur in kleinen Teilen.



(a) Analytischer Ansatz (b) Gerasterter Ansatz

Abbildung 2.2: Gegenüberstellung zwischen dem analytischen Ansatz und dem auf einer gerasterten Oberfläche basierten Ansatz. Die Keimzellen sind jeweils als schwarze Punkte dargestellt, ihre dazugehörigen Kristallflächen in Grautönen.

In Einklang mit den Bedingungen steht ein auf einem zweidimensionalen Raster simuliertes Verfahren. Grundsätzlich lassen sich zur Lösung des Problems (der Nachbildung der Zinkkristallstruktur) aber zwei grundlegend verschiedene Lösungsansätze motivieren, welche in Abbildung 2.2 illustriert sind. Einerseits wäre ein analytischer Ansatz denkbar. Dabei würden zunächst alle relevanten Keimzellen sowie deren Wachstumsgeschwindigkeiten entlang der Gitterkoordinaten festgelegt. Die einzelnen Kristallflächen ergeben sich dann aus dem zu den Keimzellen gehörenden Voronoi-Diagramm. Dieses fasst alle Punkte zu einer Region zusammen, die der Keimzelle am nächsten sind. Um die Vorgaben der Aufgabenstellung zu erfüllen, müsste man dabei die Definition von „am nächsten“ sorgfältig wählen, um die verschiedenen Wachstumsgeschwindigkeiten zu berücksichtigen. Als klarer Vorteil ergibt sich hierbei eine beliebig detaillierte Auflösung und eine Laufzeit, die prinzipiell lediglich von der Anzahl der Keimzellen abhängt.

Im Einklang mit der Aufgabenstellung wird in dieser Lösung jedoch ein gerasterter Ansatz auf einem zweidimensionalen Raster gewählt.[12] Auf diesem werden zunächst eine vorgegebene Anzahl N von Keimzellen verteilt. Jeder Keimzelle wird jeweils eine Wachstumsgeschwindigkeit für die verschiedenen Gitterrichtungen (links, rechts, oben und unten) zugeordnet. Ausgehend von den Keimzellen können sich die Kristallflächen dann, wie in Abbildung 2.3 dargestellt, auf dem Gitter ausbreiten.

Ist eine Zelle des Rasters bereits durch einen anderen Kristall belegt, kann diese nicht durch einen neuen Kristall eingenommen werden. Für die Umsetzung ist nun noch zu klären, wie das Wachstum der einzelnen Kristallflächen realisiert wird. Dafür sind mehrere Ansätze denkbar:

- In einem einfachen Floodfill-Verfahren könnte man ausgehend von den schon belegten Flächen neue hinzufügen. Hierbei wäre jedoch die Anforderung variabler Ausbreitungsgeschwindigkeiten massiv reduziert, da in jedem Iterationsschritt lediglich eine ganzzahlige Anzahl von neuen Kästchen im Raster einer Kristallfläche zugeordnet werden kann. Außerdem ist fraglich, wie Kästchen, die in einem Iterationsschritt von mehreren Kristallflächen beansprucht werden, behandelt werden. Aufgrund dieser Probleme wird der Floodfill-Ansatz verworfen.

[12] Dieses kann in der Implementierung durch einen zweidimensionalen Array realisiert werden, in dem die Zugehörigkeit zu einer Keimzelle beispielsweise durch eine Kennnummer markiert wird.

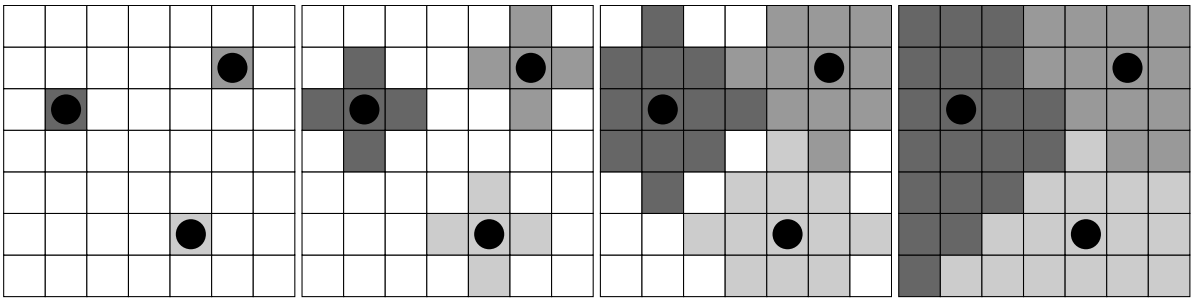


Abbildung 2.3: Ausbreitung der Kristallflächen von den Keimzellen mit fortlaufenden Iterationen.

- Um diese Probleme zu umgehen, ist ein anderer Ansatz denkbar: Für jedes Kästchen des Rasters berechnet man die Zeit, welche die einzelnen Keimzellen bräuchten, um in dieses hinein zu wachsen. Das entsprechende Kästchen wird dann der Keimzelle zugeordnet, die am schnellsten bis zu ihm wächst. Hierbei riskiert man bei einer naiven Implementierung allerdings, dass eben dieser kürzeste Pfad zu einem Kästchen des Rasters überhaupt nicht existiert, weil dieser teilweise von anderen Kristallflächen eingenommen ist. Insbesondere bei schnell wachsenden Keimzellen führt dies dazu, dass diese durch bereits existierende Kristalle hindurchwachsen.
- Um auch diesen Problemen aus dem Weg zu gehen, empfiehlt es sich, einen zeitaufgelösten Floodfill-Ansatz zu wählen. Hierbei wird anfangs ausgehend von den Keimzellen jedes Kästchen ermittelt, welches von diesen bewachsen werden kann. Entsprechend der Geschwindigkeiten entlang der Gitterrichtung kann ebenfalls ermittelt werden, wie lange die Kristallfläche braucht, dorthin zu wachsen. Diese Kästchen, sowie die Zeit, bis wann der Kristall sich zu ihnen ausgebreitet hätte, werden vermerkt. Dann wird sukzessive das Kästchen einer Kristallfläche zugeordnet, welches zeitlich als nächstes bewachsen wird. Für dessen Nachbarn wird wiederum ermittelt und vermerkt, wie lange die Kristallfläche braucht, in sie zu wachsen. Dies wird solange wiederholt, bis keine leeren Kästchen mehr verfügbar sind.

Der entsprechende Algorithmus ist als Pseudocode in Algorithmus 5 aufgeführt. Als Datenstruktur bietet es sich an, eine Prioritätswarteschlange zu verwenden. In dieser wird den Elementen mit der kleinsten Zeit zum Bewachsen die größte Priorität zugeordnet.

2.2 Erweiterungen

Zusätzlich zu dem oben beschriebenen Algorithmus ist eine Vielzahl von Verbesserungen denkbar.

Stochastisch verzögerte Keimzellen

Richtige Kristalle wachsen nicht, wie vorher angenommen, indem anfangs eine bestimmte Anzahl von Keimzellen vorhanden ist, von denen ausgehend dann die einzelnen Flächen erscheinen. Stattdessen tauchen die Keime eher statistisch verteilt auf und werden immer mehr. Um dies zu modellieren, treffen wir die Annahme, dass die Wahrscheinlichkeit, dass sich in einem Kästchen eine Keimzelle bildet, über den gesamten Wachstumsprozess konstant ist. Damit ist

Algorithmus 5 Zeitaufgelöster Floodfill-Algorithmus

Input: N Keimzellen an Positionen $\vec{x}_i$ mit Ausbreitungszeiten t_o, t_l, t_u, t_r

```
1:  Initialisiere Prioritätswarteschlange Q = [ ]
2:  for each Keimzelle i do
3:      Q ← (t = 0, x_i, i)              ▷ Füge alle Keimzellen als Startpunkte (t = 0) in Q ein
4:  end for
5:  while Q nicht leer do
6:      t_now, x, i ← pop Q              ▷ Entferne das als nächstes bewachsene Kästchen aus Q
7:      if x noch nicht besetzt then
8:          Ordne Kästchen an x der Keimzelle i zu
9:          Q ← (t_now + t_o, x + (0,1), i)        ▷ Füge Nachbarn von x zu Q hinzu
10:         Q ← (t_now + t_l, x + (−1,0), i)
11:         Q ← (t_now + t_u, x + (0,−1), i)
12:         Q ← (t_now + t_r, x + (1,0), i)
13:     end if
14: end while
```

die Anzahl von auftauchenden Keimzellen direkt von der noch verbleibenden Fläche im Raster A_{frei} abhängig. Zusammen mit einer Keimbildungsrate pro Flächen- und Zeiteinheit μ kann man nun die Anzahl von erschienenen Keimzellen nach einer Zeit t ermitteln. Diese ist im Mittel $\lambda = \mu A_{\text{frei}} t$. Solch ein Prozess kann beispielsweise als poissonverteilt angenommen werden. Für die Anzahl von dazugekommenen Keimzellen N_{neu} gilt dann

$$N_{\text{neu}} \sim \text{Poisson}(\lambda = \mu A_{\text{frei}} t)$$

nach jedem Zeitschritt t. Dies kann in unserem Algorithmus in jedem Schritt bestimmt werden.

Hexagonales Gitter

(a) Quadratisches Gitter (b) Hexagonales Gitter

Abbildung 2.4: Gegenüberstellung des bisher verwendeten quadratischen Gitters und des hexagonalen Gitters von Zink. Die jeweiligen Ausbreitungsmöglichkeiten zu nächsten Nachbarn ist durch Pfeile dargestellt.

Im Gegensatz zum vorgeschlagenen Algorithmus wächst Zink nicht in die Raumrichtungen oben, unten, links und rechts, sondern auf einem hexagonal-dicht-gepackten-Gitter. Eine Erweiterung kann also sein, die Keimzellen in der Simulation nicht nur in die quadratischen Raumrichtungen, sondern wie in einer Ebene des hexagonalen Gitters nach links-oben, rechts-oben, links, rechts, links-unten und rechts-unten wachsen zu lassen. Der Unterschied ist in

Abbildung 2.4 dargestellt.[13] Die entsprechenden vier Ausbreitungsgeschwindigkeiten können hier auf entweder sechs oder drei (horizontal, links-oben und rechts-unten sowie rechts-oben und links-unten) angepasst werden.

Relation zwischen den relevanten Größen

Zusäzlich empfiehlt es sich, die beiden relevanten Größen Ausbreitungsgeschwindigkeit und Keimzellenrate einheitenlos zu machen. Damit ist gemeint, dass diese unabhängig von der konkreten Größe des Rasters als eine charakteristische Größe das Ergebnis beschreiben. Die Geschwindigkeit wählen wir dabei so, dass die Ausbreitungszeit t in eine benachbarte Zelle gutartig gleichverteilt im Intervall $[t_{\min}, 1]$ liegt.[14] Damit ist diese bereits ohne Einheit gewählt. Die Keimzellenrate μ kann ähnlich angepasst werden, indem in diese die Gesamtgröße des Rasters $A_{\max}$ absorbiert wird. Definiert man $\tilde{\mu} \equiv \mu A_{\max}$, findet man, dass

$$\lambda = \tilde{\mu} \frac{A_{\text{frei}}}{A_{\max}} t$$

nur noch vom Anteil der freien Fläche $\frac{A_{\text{frei}}}{A_{\max}}$ und nicht mehr von der konkreten Größe abhängt.

Kristallwachstum auf anderen Geometrien

Zu guter Letzt kann man sich noch Gedanken über die zugrundeliegenden Eigenschaften der Geometrie machen. Abweichend von der planaren Platte, die durch ihre vier Ränder begrenzt ist, wären beispielsweise auch ein Zylinder[15], ein Torus[16] oder eine Kugeloberfläche als Wachstumsunterlage denkbar. Diese können über eine geschickte Wahl der Randbedingungen des zweidimensionalen Gitters umgesetzt werden.

2.3 Beispiele

Um die Qualität der produzierten Struktur einschätzen zu können, müssen zunächst dafür relevante Kriterien aufgestellt werden. Diese ergeben sich aus Beobachtungen in Abbildung 2.5:

A) Das Bild wird von mittelgroßen Kristallstrukturen dominiert, die eine scheinbar „beliebige“ Form mit größtenteils geradlinigen Kanten haben.

B) Aus diesen Kanten brechen teilweise einzelne Zacken heraus.

C) Verstreut tauchen kleinere Kristallflächen auf.

D) Die einzelnen Kristallflächen laufen oft ähnlich einem Bleistift spitz zu.

Zur Darstellung wird eine Farbpalette zwischen grau-blau und weiß-grau verwendet, um dem Original möglichst nahe zu kommen. Für N Kristalloberflächen werden auf dieser in gleichen

[13] Die weiteren Raumrichtungen können hier durch eine geschickte Nummerierung eines zweidimensionalen Arrays ermöglicht werden. Hierbei bedarf es also nicht einmal neuer Datenstrukturen.

[14] Wählt man beispielsweise die Geschwindigkeit $v \in [0, 1]$ und damit $t \sim \frac{1}{v}$, wäre t Pareto-verteilt. Damit entsteht das grundlegende Problem, dass entgegen der Beobachtung einige Keimzellen viel langsamer als andere wären. Eine solche Dominanz von wenigen ausreißenden Keimzellen ist allerdings nicht im Vorbild sichtbar.

[15] Beispiele für eine Zylindergeometrie sind Laternen- und Straßenschildmasten, Geländer und Bolzen.

[16] Beispiele für eine Torusgeometrie sind Muttern und Ringe.

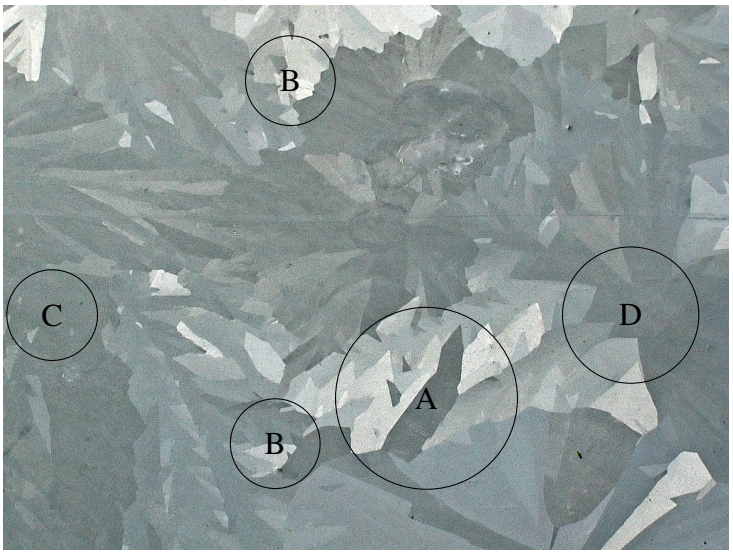
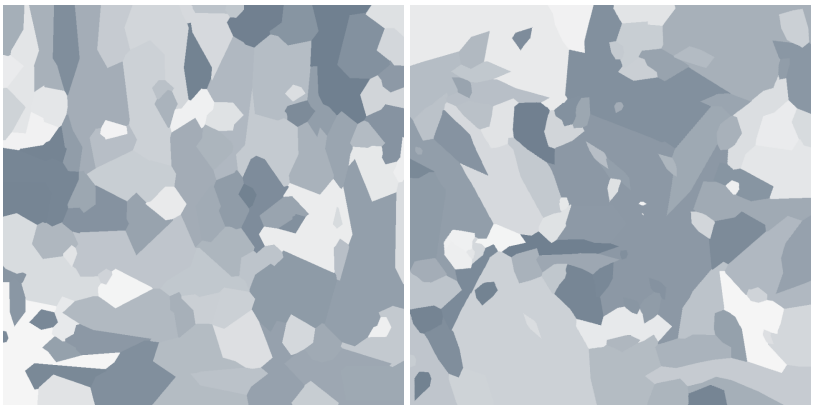


Abbildung 2.5: Beobachtete Strukturen auf der feuerverzinkten Oberfläche

Abständen N Farben festgelegt und den Kristallflächen willkürlich zugeordnet. Hier wären ebenfalls Verbesserungen durch eine Zuordnung ähnlicher Farben für Nachbarn oder eine auf die Wachstumsrichtung gestützte Zuordnung denkbar, welche hier jedoch nicht angewendet werden.

Untersuchung zwischen Hexagonalem und Quadratischem Gitter



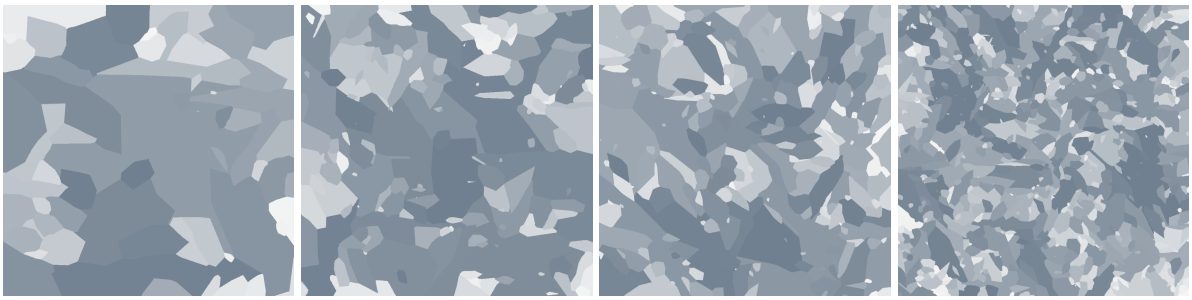
(a) Quadratisches Gitter (b) Hexagonales Gitter

Abbildung 2.6: Vergleich zwischen quadratischem und hexagonalem Gitter

Um den Unterschied zwischen quadratischen und hexagonalen Ausbreitungsrichtungen zu untersuchen, sind in Abbildung 2.6 zwei bei sonst gleichen Parametern aus $N = 100$ anfänglichen, auf der Oberfläche gleichverteilten Keimzellen generierte Kristallflächen dargestellt. Beide reproduzieren Strukturen mit geraden Kanten (A) und ermöglichen das Ausbrechen von Zacken (B). Die Kristallstrukturen laufen außerdem jeweils spitz zu (D), beim hexagonalen Gitter sind diese Spitzen jedoch ähnlicher zu denen aus Abbildung 2.5 ausgeprägt, weshalb dieses die besseren Ergebnisse produziert. Dementsprehend werden fortan die hexagonalen Raumrichtungen verwendet.

Untersuchung der Keimzellenrate

Um verstreut kleinere Keimzellen einzubringen, wird in Abbildung 2.7 die Auswirkung von stochastisch erscheinenden Keimzellen dargestellt.



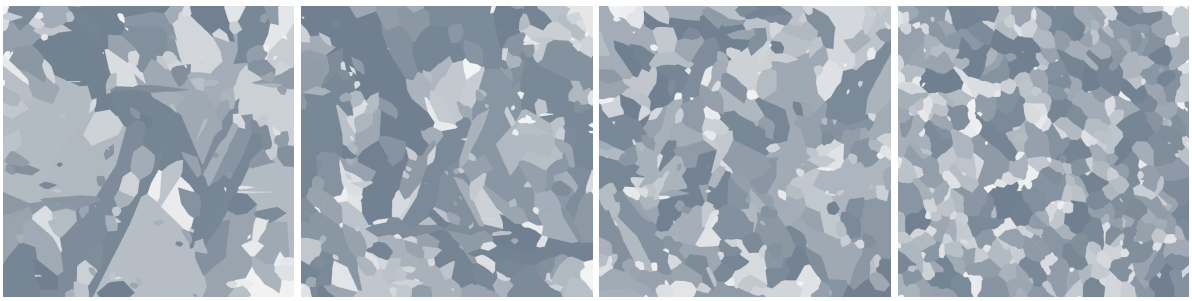
(a) $\tilde{\mu} = 1.6$ (b) $\tilde{\mu} = 8$ (c) $\tilde{\mu} = 16$ (d) $\tilde{\mu} = 80$

Abbildung 2.7: Vergleich verschiedener Keimzellenraten $\tilde{\mu}$.

Mit zunehmender Keimzellenrate $\tilde{\mu}$ bilden sich immer mehr kleine Strukturen aus. Dabei sind insbesondere auch winzige Keimzellen (C) sichtbar. Für $\tilde{\mu} = 16$ scheinen diese am nächsten am Vorbild in Abbildung 2.5, weswegen fortlaufend dieser Wert verwendet wird.

Untersuchung der minimalen Ausbreitungsdauer

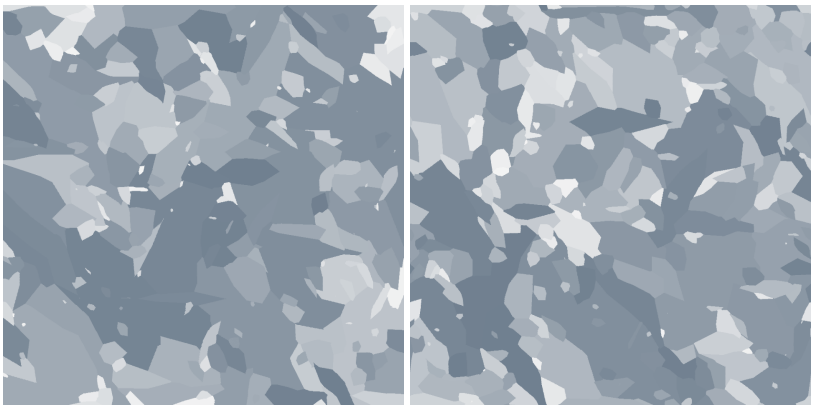
Die eingeführte Mindestausbreitungszeit $t_{\min}$ kann ebenfalls verglichen werden. Diese kontrolliert, wie in Abbildung 2.8 dargestellt, die realisierten Größenunterschiede zwischen großen und kleinen Kristallflächen. Ist der Wert sehr klein, dominieren fast ausschließlich einige sich schnell ausbreitende Keimzellen. Ist er wiederum sehr groß, gibt es kaum kleine Keimzellen. Der Wert $t_{\min} = 0.05$ scheint hier eine gute Übereinstimmung mit dem Original zu liefern.



(a) $t_{\min} = 0.01$ (b) $t_{\min} = 0.05$ (c) $t_{\min} = 0.10$ (d) $t_{\min} = 0.40$

Abbildung 2.8: Vergleich verschiedener Mindestausbreitungszeiten $t_{\min}$.

Untersuchung des Wachstums auf einem Torus



(a) Ohne Toruseigenschaft (b) Mit Toruseigenschaft

Abbildung 2.9: Auswirkung der Toruseigenschaft

Zuletzt sind in Abbildung 2.9 die Auswirkungen der Toruseigenschaft dargestellt. Klar erkennbar sind die Überwüchse am Rand des Gitters (eine Keimzelle auf der linken Seite kann in die rechte Seite hineinwachsen). Dies bringt aber keine bessere Übereinstimmung mit der Vorlage und wird deshalb nicht weiter verwendet.

Endergebnis

In Abbildung 2.10 ist eine Kristallfläche mit den finalen Parametern dargestellt.

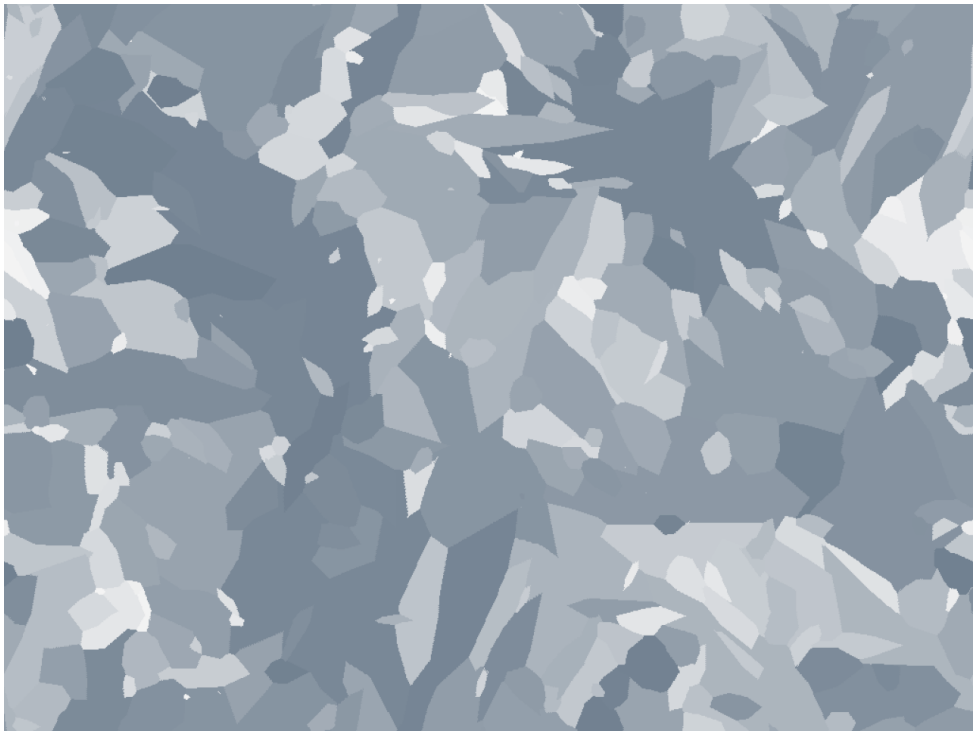


Abbildung 2.10: Endergebnis: $N = 100$, hexagonales Gitter, $\tilde{\mu} = 16$, $t_{\min} = 0.05$

2.4 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-2]$ **Simulation fehlerhaft**
  Die Simulation muss sich an die Vorgaben halten:
  - Es muss mehrere Kristallisationskeime geben. Es ist nicht notwendig, dass diese zufällig verteilt werden.
  - Jeder Kristallisationskeim muss in mindestens vier Richtungen wachsen.
  - Jeder Kristallisationskeim sollte in jede Richtung mit einer eigenen Geschwindigkeit wachsen. Es muss also vier potenziell verschiedene Wachstumsgeschwindigkeiten je Keim geben.
  - Es muss garantiert sein, dass ein Kristall nicht in einen anderen Kristall hinein oder durch einen anderen Kristall durch wachsen kann.

  Wenn eine Vorgabe nicht eingehalten wird, dann wird ein Punkt abgezogen. Ab zwei nicht eingehaltenen Vorgaben, werden zwei Punkte abgezogen.
- $[-1]$ **Parameter nicht ausreichend variierbar**
  Die Aufgabenstellung nennt explizit vier Parameter, die variierbar sein sollen:
  - Die Anzahl der Keime – Die Zahl muss einstellbar sein, dafür reicht es aus, wenn dies an bequemer Stelle im Quellcode möglich ist.
  - Die Orte der Keime – Es ist ausreichend, wenn die Orte zufällig gewählt werden.
  - Entstehungszeitpunkte der Keime – Es wird akzeptiert, wenn alle Keime zu Anfang der Simulation platziert werden.
  - Wachstumsgeschwindigkeiten – Auch hier ist eine zufällige Auswahl der vier Geschwindigkeiten je Keim in Ordnung.

  Wenn bei mindestens zwei Parametern die Variierbarkeit nicht ausreichend ist, so wird ein Punkt abgezogen.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Die bei der Simulation entstandenen Muster müssen als Bilder mit unterschiedlichen, sich voneinander abgrenzenden Farben visualisiert worden sein. Es muss erkennbar sein, mit welchen Parametern welches Bild erstellt wurde.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll mindestens 3 Bilder enthalten, welche mit verschiedenen Parametern simuliert wurden. Mindestens eines der Bilder soll vergleichbar zum Beispiel aus der Aufgabenstellung sein.

Aufgabe 3: Sudokopie

3.1 Lösungsidee

Um zu vergleichen, ob zwei Sudokus bis auf die vorgegebenen Transformationen gleich sind, können auf eines der beiden Sudokus nacheinander alle möglichen Transformationen angewendet (oder ausgelassen) und das Ergebnis daraufhin mit dem anderen Sudoku verglichen werden. Dieses Vorgehen folgt dem Prinzip der *vollständigen Suche* (*Brute Force*). Die Reihenfolge dieser Transformationen ist hierbei entscheidend. In dieser Lösung legen wir die folgende Reihenfolge für die Transformationen fest:

1. Rotation um 90°
2. Permutation der Blockspalten
3. Permutation der Blockzeilen
4. Permutation der einzelnen Spalten
5. Permutation der einzelnen Zeilen
6. Umbenennung der Zahlen

Es lässt sich nachprüfen, dass alle Sudokus, die sich mit einer anderen Reihenfolge der Transformationen erzeugen ließen, auch mit dieser Reihenfolge erzeugen lassen. Dies wird mit einem Beispiel deutlich: Mit der oben genannten Reihenfolge an Transformationen kann erst ein vertauschen der Spalten 1 und 2, gefolgt von einem Tausch der Blockspalten 1 und 2 nicht dargestellt werden, da immer zuerst die Blockspalten betrachtet werden und dann die Spalten. Allerdings ist das Tauschen der Blockspalten 1 und 2 gefolgt von einem Tausch der Zeilen 4 und 5 äquivalent.

Weiterhin ist es ausreichend, eine 90°-Rotation zu überprüfen, da sich zwei aufeinanderfolgende 90°-Rotationen (also eine 180°-Rotation) durch Permutationen von Zeilen, gefolgt von Permutationen von Spalten erreichen ließen.

Es wurde diese bestimmte Reihenfolge gewählt, weil sie bestimmte Optimierungen zulässt, welche den Algorithmus beschleunigen können. Diese Optimierungen werden im Abschnitt 3.3 erklärt. Der folgende Pseudocode zeigt den Ablauf des Algorithmus. Hierbei werden zunächst alle Permutationen aufgezählt und dann wird eine Kopie des ersten Sudokus erstellt. Auf diese Kopie werden dann alle in den Schleifen festgelegten Transformationen angewendet. Das Ergebnis wird daraufhin sowohl ohne als auch mit der 90°-Rotation auf Gleichheit bis auf Umbenennung (s. u.) zum zweiten Sudoku getestet.

Algorithmus 6 Vergleichen zweier Sudokus durch Generieren aller Varianten des einen

```
 1: procedure IstVariante(Sudoku1, Sudoku2)
 2:     for all Permutationen von Blockspalten do
 3:         for all Permutationen von Blockzeilen do
 4:             for all Permutationen von Spalten do
 5:                 for all Permutationen von Zeilen do
 6:                     buffer ← Transform(Sudoku1, Permutationen)
 7:                     if Umbenennung(buffer, Sudoku2) then
 8:                         return true
 9:                     end if
10:                     buffer ← Rotate90(buffer)
11:                     if Umbenennung(buffer, Sudoku2) then
12:                         return true
13:                     end if
14:                 end for
15:             end for
16:         end for
17:     end for
18:     return false
19: end procedure
```

Die Funktion Umbenennung testet nun, ob diese zwei Sudokus bis auf die Umbenennung der Zahlen gleich sind.

Hierfür wird ein Mapping (eine Zuordnung) erstellt, welches Zahlen des ersten Sudokus auf Zahlen des zweiten Sudokus abbildet. Sollten zwei Einträge zu unterschiedlichen Mappings führen, so kann geschlossen werden, dass es keine Umbenennung der Zahlen gibt, welche zur Gleichheit der beiden Sudokus führen würde. Hierfür wird immer das erste Aufkommen einer Zahl zusammen mit ihrem korrespondierenden Eintrag im Mapping gespeichert.

Ebenso gäbe es keine zur Gleichheit der Sudokus führende Umbenennung, wenn zwei Zahlen auf die gleiche Zahl abgebildet würden. Um dies zu überprüfen, speichern wir im Bitarray Zugewiesen, auf welche Zahlen bereits eine Zahl abgebildet wird.

Algorithmus 7 Vergleichen zweier Sudokus in Bezug auf Umbenennung

```
 1: procedure UMBENENNUNG(Sudoku1, Sudoku2)
 2:     Mapping ← {}
 3:     Zugewiesen ← Array(10, false)
 4:     Mapping[0] ← 0                          ▷ leeres Feld ↦ leeres Feld
 5:     for all Felder i in einem 9 × 9 Sudoku do
 6:         if Mapping[Sudoku1[i]] schon gefunden then
 7:             if Mapping[Sudoku1[i]] ≠ Sudoku2[i] then
 8:                 return false
 9:             end if
10:         else
11:             Mapping[Sudoku1[i]] ← Sudoku2[i]
12:             if Zugewiesen[Sudoku2[i]] then
13:                 return false
14:             end if
15:             Zugewiesen[Sudoku2[i]] ← true
16:         end if
17:     end for
18:     return true
19: end procedure
```

3.2 Laufzeit

Es gibt nur eine begrenzte Anzahl an möglichen Transformationen. Diese kann berechnet werden, indem die Anzahlen der Möglichkeiten für die einzelnen Permutationen miteinander multipliziert werden. Es gibt je 3! mögliche Permutationen von Blockzeilen bzw. Blockspalten und 3! Permutationen für jede der 3 Gruppen von Zeilen bzw. Spalten. Dazu kommen zwei Möglichkeiten für die Verwendung der 90°-Drehung.

Die Anzahl der möglichen Transformationen beträgt also

$$3!^8 \cdot 2 = 3359232.$$

So oft werden im schlechtesten Fall die Funktionen TRANSFORM und UMBENENNUNG aufgerufen. Diese haben einen linearen Aufwand in Abhängigkeit von der Anzahl der Felder.

Für beliebig große $n^2 \times n^2$-Sudokus wäre der Aufwand zum Prüfen aller Transformationen

$$\mathcal{O}(n!^{2n+2} \cdot n^4).$$

Hier ist n die Seitenlänge eines Blocks, welche bei einem gewöhnlichen 9×9-Sudoku 3 entspricht. Der Faktor n^4 ist die Anzahl der Felder im Sudoku.

3.3 Optimierungen

Eine Möglichkeit, frühzeitig zu testen, ob eine Permutation zielführend sein kann, ist, zu zählen, wie viele Einträge in einer bestimmten Zeile oder Spalte oder einem bestimmten Block auftreten. So kann eine Spalte mit 4 Einträgen nie auf eine Spalte mit 3 Einträgen abgebildet

werden, da sich diese Anzahl nicht durch Permutation von Zeilen, Permutation von Blockzeilen oder Umbenennung verändern kann. Hierbei muss jedoch beachtet werden, dass bei einer 90°-Rotation Zeilen auf Spalten abgebildet werden und sich die Abbildung der 3×3-Blöcke aufeinander auch ändert. In der hier verwendeten Implementierung wurde diese Optimierung auf Blockebene eingebracht. So wird nach jeder Permutation von Blockzeilen oder Blockspalten zunächst getestet, ob die Blöcke, die hierbei aufeinander abgebildet wurden, die gleiche Anzahl an Einträgen haben. Dies muss hier auch für die 90°-Rotation getestet werden.

Algorithmus 8 Vergleichen zweier Sudokus durch optimiertes Generieren von Varianten des einen

```
 1: procedure IstVariante(Sudoku1, Sudoku2)
 2:     for all Permutationen von Blockspalten do
 3:         for all Permutationen von Blockzeilen do
 4:             if Anzahl der Einträge in Blöcken ist ungleich then
 5:                 continue
 6:             end if
 7:             for all Permutationen von Spalten do
 8:                 for all Permutationen von Zeilen do
 9:                     buffer ← Transform(Sudoku1, Permutationen)
10:                     if Umbenennung(buffer, Sudoku2) then
11:                         return true
12:                     end if
13:                     buffer ← Rotate90(buffer)
14:                     if Umbenennung(buffer, Sudoku2) then
15:                         return true
16:                     end if
17:                 end for
18:             end for
19:         end for
20:     end for
21:     return false
22: end procedure
```

3.4 Beispiele

Im Folgenden werden die Beispiele und eine ihrer Lösungen vorgestellt. Lösungen können auch anderen Reihenfolgen folgen und deutlich anders aussehen. Diese Festlegung ändert jedoch nichts daran, ob sich ein Sudoku in ein anderes umformen lässt.

sudoku0.txt

Im ersten Beispiel werden nur einzelne Zeilen und Spalten vertauscht, um ein Sudoku in das andere zu überführen.

6				8				1
	9		4			6		
				9				2
					1			
			6					5
3	2	7	5					8
				7				
		6	8		3	9	7	
							8	

Startsudoku

		9			4			6
	6		8			1		
			9			2		
				1				
					6	5		
7	3	2			5	8		
			7					
							8	
6				3	8		7	9

Zielsudoku

Zeilen 1-3	2 1 3
Zeilen 7-9	1 3 2
Spalten 1-3	3 1 2
Spalten 4-6	2 3 1
Spalten 7-9	3 2 1

Als Vertauschungen formuliert:

- Zeilen: 1 mit 2, 8 mit 9
- Spalten: 1 mit 2, 1 mit 3, 4 mit 5, 5 mit 6, 7 mit 9

Laufzeit: 40 ms

sudoku1.txt

			2					
	1			6		9	2	
					1	5		
8	9		6		2			1
		5						4
		4		7			6	
					8	3		
5		7	4					
				1				

Startsudoku

		6			2		4	
7				6		1		
		2	1					8
			5	9				3
6				2				
	4	1						
		8					5	
		9		1				
4	5						7	

Zielsudoku

Bei diesem Beispiel wurden nur Blockspalten und Blockzeilen vertauscht sowie eine Rotation um 90° ausgeführt.

Rotation	
Blockzeilen	2 3 1
Blockspalten	2 3 1

Als Vertauschungen formuliert:

- Rotation
- Zeilenblöcke: 1 mit 3, 1 mit 2
- Spaltenblöcke: 1 mit 3, 1 mit 2

Laufzeit: < 1 ms

sudoku2.txt

5				8			4	9
			5				3	
	6	7	3					1
1	5							
			2		8			
							1	8
7					4	1	5	
	3				2			
4	9			5				3

Startsudoku

	5	1		9		6		
	4		6					
		2	4				7	8
						2	6	
	2	9						
			3		9			
2	6				5	8		
					3		4	
		4		6		5	1	

Zielsudoku

Dies ist das Beispiel aus der Aufgabenstellung.

Blockspalten	3 2 1
Zeilen 4–6	1 3 2
Umbenennung	2 3 4 5 6 7 8 9 1

Als Vertauschungen formuliert:

- Spaltenblöcke: 1 mit 3
- Zeilen: 5 mit 6
- Umbenennung: 1 > 2, 2 > 3, 3 > 4, 4 > 5, 5 > 6, 6 > 7, 7 > 8, 8 > 9, 9 > 1

Laufzeit: 35 ms

sudoku3.txt

					6		7	
	1	4						5
6			8	3		9		
			9					4
2		9				1		
	7			1	5	2		
	4		6					2
1				8	7	5		3
7								

Startsudoku

3		6					5	
					1	8		
	5	1	7				6	
		4					3	
	7		5		6		4	
9				8		6		
						7		
				9				6
7	2							

Zielsudoku

Bei diesem Beispiel kann das Startsudoku nicht zum Zielsudoku transformiert werden.

Laufzeit: 35 ms

sudoku4.txt

7							1	
		8			2		9	
	2		7	8			3	
		4						
	5			9		6		
	9			7	6		2	4
		5	2			7	4	
	7	3	1		5			2

Startsudoku

9					3	2		1
	2	6		8				7
			7					
		9						8
		8	4	1	6	9		
	5					7		
		5			8			2
	6	7		3				
				7		8		4

Zielsudoku

Bei diesem Beispiel treten alle Arten von Transformationen auf.

Rotation	
Blockzeilen	1 3 2
Blockspalten	2 3 1
Zeilen 1–3	3 2 1
Zeilen 4–6	3 2 1
Zeilen 7–9	3 2 1
Spalten 1–3	3 2 1
Spalten 4–6	3 1 2
Spalten 7–9	3 1 2
Umbenennung	4 8 1 9 2 5 7 3 6

Als Vertauschungen formuliert:

- Rotation
- Zeilenblöcke: 2 mit 3
- Spaltenblöcke: 1 mit 2, 2 mit 3
- Zeilen: 1 mit 3, 4 mit 6, 7 mit 9
- Spalten: 1 mit 3, 4 mit 5, 4 mit 6, 7 mit 8, 7 mit 9
- Umbenennung: 1 > 4, 2 > 8, 3 > 1, 4 > 9, 5 > 2, 6 > 5, 7 > 7, 8> 3, 9 > 6

Laufzeit: 341 ms

3.5 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Nicht alle Transformationen korrekt umgesetzt**
  Das Lösungsverfahren muss alle Transformationen umsetzen. Dazu gehören:
  - Permutation der drei Spalten innerhalb der Spaltenblöcke
  - Permutation der drei Spaltenblöcke
  - Permutation der drei Zeilen innerhalb der Zeilenblöcke
  - Permutation der drei Zeilenblöcke
  - 90-Grad-Rotation im Uhrzeigersinn
  - Umbenennung der Ziffern 1 bis 9

  Insbesondere muss auch die Umbenennung der Ziffern bzw. die Überprüfung, ob eine konsistente Umbenennung existiert, korrekt implementiert sein.
- $[-1]$ **Lösungsverfahren anderweitig fehlerhaft**
  Die Transformationen müssen nicht der Reihenfolge der Aufgabenstellung folgen. Sollte eine Reihenfolge festgelegt werden, die dazu führt, dass nicht alle möglichen Varianten erreicht werden können, so wird ein Punkt abgezogen. Die Reihenfolge der Transformationen muss aus der Dokumentation ersichtlich werden. Ist das nicht der Fall, kann bei Kriterium 2 abgezogen werden.

  Ein Beispiel hierfür wäre, die Rotation zwischen den Zeilen- und den Spaltenpermutationen auszuführen: Die Spaltenpermutationen wären dann bedeutungslos, da sie keine zusätzlichen Varianten mehr erzeugen würden; stattdessen müssten weitere Zeilenpermutationen durchgeführt werden können.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Der Aufwand des Verfahrens sollte nicht schlechter sein als der hier beschriebene Worst-Case-Aufwand von $\mathcal{O}(n!^{2n+2} \cdot n^4)$. Wenn die Transformationen blind durchprobiert werden, sodass die Laufzeit nicht erkennbar begrenzt ist, führt dies zu Punktabzug. Alle Beispieleingaben sollten sich in unter einer Minute lösen lassen.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Das Programm soll ausgeben, ob die zwei Sudokus Varianten voneinander sind. Falls ja, dann müssen die Transformationen nachvollziehbar dargestellt werden. Dabei ist auch eine Beschreibung der Transformationen mit Worten erlaubt.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens 3 der vorgegebenen Beispiele (`sudoku0.txt` bis `sudoku4.txt`) enthalten, darunter `sudoku3.txt`.

Aufgabe 4: Fahrradwerkstatt

In dieser Aufgabe soll Marc dabei unterstützt werden, die Aufträge seiner Fahrradwerkstatt so abzuarbeiten, dass die Kunden möglichst wenig mit langen Wartezeiten konfrontiert werden.

4.1 Lösungsidee

Die Aufgabe kann dadurch gelöst werden, dass alle zu einem bestimmten Zeitpunkt vorhandenen Aufträge entsprechend einer vorher festgelegten Priorisierung abgearbeitet werden. Dabei muss ein Auftrag zunächst vollständig abgeschlossen werden, bevor ein neuer begonnen werden kann. Die zwei vorgegebenen Priorisierungen sind:

A) Priorisierung nach dem Eingangszeitpunkt. Hierbei werden zuerst eingetroffene Aufträge auch zuerst bearbeitet.

B) Priorisierung nach der Bearbeitungsdauer. Hierbei wird der kürzeste vorhandene Auftrag zuerst bearbeitet.

Eine mögliche Abfolge der Simulationsschritte ist in Algorithmus 9 gegeben. In einer effizienten Umsetzung sollte beachtet werden, dass die Größe, nach der die Aufträge priorisiert werden, bei den vorgegebenen Priorisierungen bereits bei deren Eingang feststeht und sich nicht mehr ändert.[17] Dementsprechend ändert sich auch die Abarbeitungsreihenfolge bereits eingegangener Aufträge nicht mehr und neu hinzukommende Einträge müssen lediglich „dazwischengeschoben“ werden. Dadurch muss die Liste mit bereits eingegangenen Aufträgen nicht in jeder Iteration neu sortiert werden, stattdessen können die neuen Aufträge bei geschickter Wahl der zugrunde liegenden Datenstruktur[18] für die Liste L' jeweils mit einer Laufzeit von $\mathcal{O}(\log n)$ eingefügt werden.

Algorithmus 9 Fahrradwerkstatt-Simulation

Input: Liste L von Aufträgen mit Startzeit t_0 und Bearbeitungsdauer t_b.

```
1:  Initialisiere die aktuelle Uhrzeit T auf den Beginn von Tag 0
2:  Initialisiere die leere Liste L' mit nicht-bearbeiteten Aufträgen
3:  while nicht alle Aufträge abgearbeitet do
4:      if in der Liste L' sind nicht-bearbeitete Aufträge vorhanden then
5:          Sortiere eingegangene (t_0 < T) Aufträge in Liste L' nach ihrer Priorisierung ein
6:          Passe, wenn nötig, Priorisierung alter Aufträge an
7:          Bearbeite den Auftrag mit der höchsten Priorität
8:          Spule Zeit T um dessen Bearbeitungszeit t_b vor      ▷ Hier die Arbeitszeit beachten
9:      else
10:         Spule Zeit T vor, bis der nächste Auftrag vorhanden ist
11:     end if
12: end while
```

Bei der Umsetzung muss außerdem besonders beachtet werden, dass Marc nur von 9 bis 17 Uhr arbeitet und dass manche Aufträge dementsprechend mehr als einen Arbeitstag beanspruchen können.

[17] Damit flexible Erweiterungen wie die später eingeführte Priorisierung C möglich sind, müssen unter Umständen bereits eingegangene Aufträge umsortiert werden. Dafür ist in Algorithmus 9 die Zeile 6 vorgesehen. Hierbei ist die Umsortierung alter Aufträge i. A. effizienter als eine komplette Neuaufstellung der Liste L'.

[18] Hierfür bietet sich etwa eine auf einem Heap implementierte Prioritätenliste an.

Die relevante Kenngröße, um zu beurteilen, wie gut die verschiedenen Verfahren sind, ist die Wartezeit der Aufträge. Sie berechnet sich aus Eingangszeitpunkt und Ausgangszeitpunkt als

$$\text{Wartezeit der Aufträge} = \text{Ausgangszeitpunkt} - \text{Eingangszeitpunkt}.$$

Weitere Kenngrößen

Zusätzlich zur Wartezeit der Aufträge ergeben sich aus Marcs Arbeitsprozess einige weitere Kenngrößen, welche ebenfalls untersucht werden können.

- Bei Annahme eines Auftrags weiß Marc bereits, wie lange dessen Bearbeitung benötigt. Die tatsächliche Verzögerung, die ein Kunde wahrnimmt, ist dann eben die Dauer, die der Auftrag länger als vorher bekannt braucht:

$$\text{Verzögerung} = \text{Ausgangszeitpunkt} - \text{Eingangszeitpunkt} - \text{Bearbeitungsdauer}.$$

- Außerdem kann man die durch die Aufgabenstellung berechnete Wartezeit auch lediglich auf die vergangene Arbeitszeit beziehen und die Totzeit, jeweils die Zeit außerhalb von Marcs Arbeitszeiten, ignorieren.
- Dadurch, dass Marc gelegentlich warten muss, bis ein neuer Auftrag im Laden eintrifft, kann es zu nicht genutzter Arbeitszeit kommen. Diese kann in der relativen Kenngröße der Auslastung abgelesen werden:

$$\text{Auslastung} = \frac{\text{Tatsächlich genutzte Arbeitszeit}}{\text{Vergangene Arbeitszeit}}.$$

- Auch nachdem keine neuen Aufträge mehr eingehen, gibt es eine gewisse Menge an Arbeit, die Marc noch abarbeiten muss. Die Zeit, die dafür nötig ist, bezeichnen wir als Überhang:

$$\text{Überhang} = \text{Arbeitsende} - \text{Eingangszeit des allerletzten Auftrags}.$$

4.2 Erweiterungen

Eine wichtige und von der Aufgabenstellung geforderte Erweiterung ist das Hinzufügen neuer Priorisierungsverfahren. Hierfür ist beispielsweise das folgende Verfahren denkbar:

C) Priorisierung nach Bearbeitungsdauer mit einer Höchstwartezeit. Priorisiert werden die Aufträge hierbei zunächst nach ihrer Bearbeitungsdauer. Sobald jedoch ein Auftrag eine gewisse Maximalwartedauer überschreitet[19], wird dieser mit höherer Priorität vor den anderen bearbeitet.

Außerdem könnte Marc auch einen weiteren Angestellten einstellen, damit mehrere Aufträge gleichzeitig abgearbeitet werden können.

[19] Diese Maximalwartedauer wird hier auf 50 000 min ≈ 35 d festgelegt. Der Wert wurde durch Probieren bestimmt.

4.3 Beispiele

		Methode		
		A	**B**	**C**
`-statt0.txt`	Mittlere Wartezeit	32 754 (10 757)	16 986 (5497)	22 344 (7324)
	Standardabweichung	20 121 (6761)	27 132 (9041)	23 916 (8005)
	Maximale Wartezeit	68 771 (23 475)	188 734 (62 465)	89 663 (29 731)
	Überhang	56 234	51 467	56 234
	Effektivität		91,1%	
`-statt1.txt`	Mittlere Wartezeit	63 536 (21 029)	11 908(3800)	35 761 (11 773)
	Standardabweichung	43 405 (14 491)	47 673 (15 874)	37 221 (12 426)
	Maximale Wartezeit	128 321 (43 121)	433 563 (144 104)	331 601 (110 395)
	Überhang	86 506	84 867	84 881
	Effektivität		86,8%	
`-statt2.txt`	Mittlere Wartezeit	51 200 (16 916)	14 862 (4799)	32 285 (10 595)
	Standardabweichung	26 713 (8953)	36 402 (12 127)	28 681 (9565)
	Maximale Wartezeit	110 973 (37 731)	327 087 (108 696)	145 614 (48 172)
	Überhang	27 429	21 052	23 529
	Effektivität		95,4%	
`-statt3.txt`	Mittlere Wartezeit	30 029 (9854)	17 246 (5594)	20 486 (6661)
	Standardabweichung	17 244 (5780)	53 697 (17 884)	22 484 (7502)
	Maximale Wartezeit	60 821 (20 323)	382 016 (127 215)	88 135 (29 133)
	Überhang	27 385	27 183	27 241
	Effektivität		95,0%	
`-statt4.txt`	Mittlere Wartezeit	74 428 (24 633)	42 201 (13 904)	56 887 (18 861)
	Standardabweichung	45 330 (15 129)	78 756 (26 230)	50 253 (16 781)
	Maximale Wartezeit	167 059 (55 165)	363 155 (120 787)	253 641 (84 208)
	Überhang	62 142	61 235	62 142
	Effektivität		91,5%	

Tabelle 2: Kenngrößen in Minuten der verschiedenen Methoden bei den Beispieleingaben `fahrradwerkstatt*.txt`. In Klammern sind die Größen ohne Marcs Freizeit angegeben. Der beste Wert ist jeweils in grün, der schlechteste in rot markiert.

			Methode A (Ursprüngliches Verfahren)				
			`werkstatt0.txt`	`werkstatt1.txt`	`werkstatt2.txt`	`werkstatt3.txt`	`werkstatt4.txt`
Minuten	mit Totzeit	Durchschnitt	32754 min	63536 min	51200 min	30029 min	74428 min
		Maximum	68771 min	128321 min	110973 min	60821 min	167059 min
	ohne Totzeit	Durchschnitt	10757 min	21029 min	16916 min	9854 min	24633 min
		Maximum	23475 min	43121 min	37731 min	20323 min	55165 min
Stunden	mit Totzeit	Durchschnitt	545,9 h	1058,9 h	853,3 h	500,5 h	1240,5 h
		Maximum	1146,2 h	2138,7 h	1849,6 h	1013,7 h	2784,3 h
	ohne Totzeit	Durchschnitt	179,3 h	350,5 h	281,9 h	164,2 h	410,6 h
		Maximum	391,3 h	718,7 h	628,9 h	338,7 h	919,4 h
Tage	mit Totzeit	Durchschnitt	22,746 d	44,122 d	35,556 d	20,853 d	51,686 d
		Maximum	47,758 d	89,112 d	77,065 d	42,237 d	116,013 d
	ohne Totzeit	Durchschnitt	7,470 d	14,603 d	11,747 d	6,843 d	17,106 d
		Maximum	16,302 d	29,945 d	26,202 d	14,113 d	38,309 d
			Methode B (Neue Idee)				
			`werkstatt0.txt`	`werkstatt1.txt`	`werkstatt2.txt`	`werkstatt3.txt`	`werkstatt4.txt`
Minuten	mit Totzeit	Durchschnitt	16986 min	11908 min	14862 min	17246 min	42201 min
		Maximum	188734 min	433563 min	327087 min	382016 min	363155 min
	ohne Totzeit	Durchschnitt	5497 min	3800 min	4799 min	5594 min	13904 min
		Maximum	62465 min	144104 min	108696 min	127215 min	120787 min
Stunden	mit Totzeit	Durchschnitt	283,1 h	198,5 h	247,7 h	287,4 h	703,4 h
		Maximum	3145,6 h	7226,1 h	5451,5 h	6366,9 h	6052,58 h
	ohne Totzeit	Durchschnitt	91,6 h	63,3 h	80,0 h	93,2 h	231,7 h
		Maximum	1041,1 h	2401,7 h	1811,6 h	2120,3 h	2013,1 h
Tage	mit Totzeit	Durchschnitt	11,796 d	8,269 d	10,321 d	11,976 d	29,306 d
		Maximum	131,065 d	301,085 d	227,144 d	265,289 d	252,19 d
	ohne Totzeit	Druchschnitt	3,817 d	2,639 d	3,333 d	3,885 d	9,656 d
		Maximum	16,302 d	29,945 d	26,202 d	14,113 d	38,309 d

Tabelle 3: Kenngrößen in Minuten, Stunden und Tagen der verschiedenen Methoden bei den Beispieleingaben.

In Tabelle 2 werden die entsprechenden Kenngrößen für die Beispieldateien aufgeführt. Dabei fällt vor allem auf, dass die mittlere Wartezeit bei Methode A durchweg größer als bei Methode B ist. Dafür ist ihre Standardabweichung und maximale Wartezeit kleiner, was bedeutet, dass Marcs Kunden ihre Wartezeit verlässlicher einschätzen könnten und es keine exorbitant lange wartenden Aufträge gäbe.

Hier gilt es nun eine Abwägung vorzunehmen. Verwendet Marc die Methode B, ist es sehr wahrscheinlich, dass er mit einigen sehr wütenden Kunden (eben denen, die bedeutend länger als vorgesehen warten) eine lange Zeit zu tun hat. Sowohl für die Kunden als auch für Marc stellt sich dadurch Methode A als befriedigender heraus. Darauf basierend scheint die Methode C das beste aus beiden Welten zu kombinieren, indem extreme Wartezeiten fast vollständig verhindert werden und zugleich die mittlere Wartezeit im Vergleich zu Methode A sinkt.

Dieser Sachverhalt wird ebenfalls in den Abbildungen 4.1 bis 4.5 deutlich. Dort ist jeweils der Anteil $F(\text{Länge} \leq \text{Wartezeit})$ der Aufträge, die höchstens eine gewisse Zeit warten mussten, dargestellt. Ist diese Funktion bei einer Wartezeit von 30 000 min bei 0.5, bedeutet das, dass die Hälfte (50%) der Aufträge nach einer Wartezeit von eben 30 000 min Marcs Laden verlassen hat.[20] Hier zeigt sich deutlich, dass bei Methode A zwar die meisten Aufträge etwas länger als bei Methode B benötigen, weil Methode B zunächst (kurze) Aufträge sehr schnell abarbeitet, bei Methode B aber eben ungefähr 10% der Aufträge sehr lange warten müssen. Die Methode C scheint auch in dieser Darstellung ein guter Kompromiss, da sie ebenfalls (wie Methode B) über einen steilen Anstieg verfügt, aber gleichzeitig weniger sehr lange wartende Aufträge erzeugt.

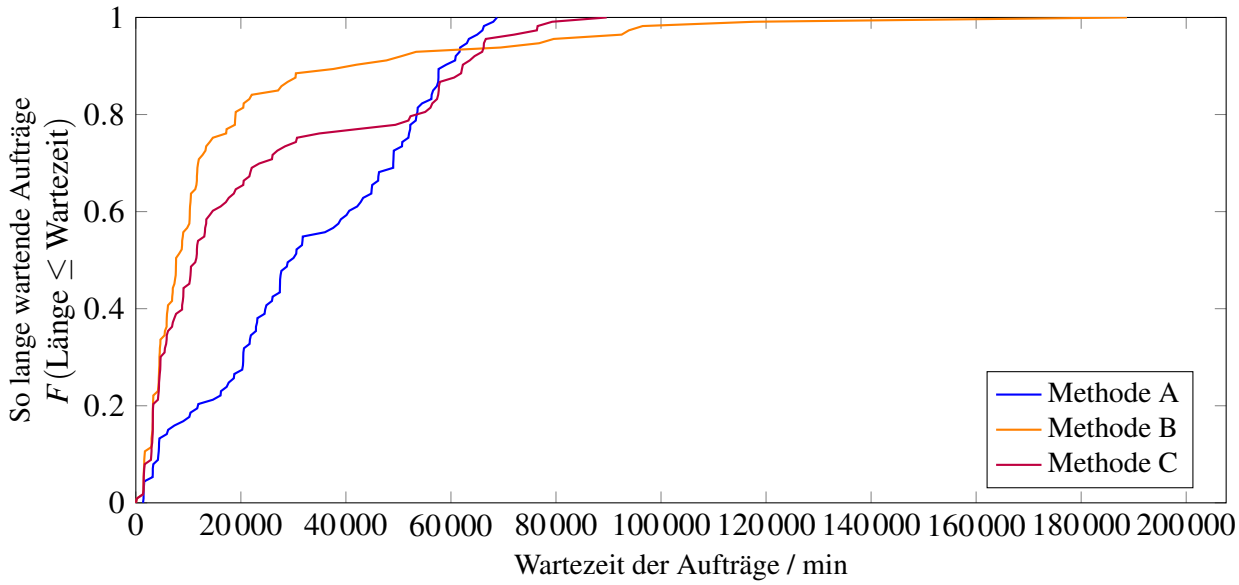


Abbildung 4.1: `fahrradwerkstatt0.txt`

[20] Dies entspricht der kumulierten Verteilungsfunktion. Sie ist nah mit der Wahrscheinlichkeitsverteilungsfunktion verwandt.

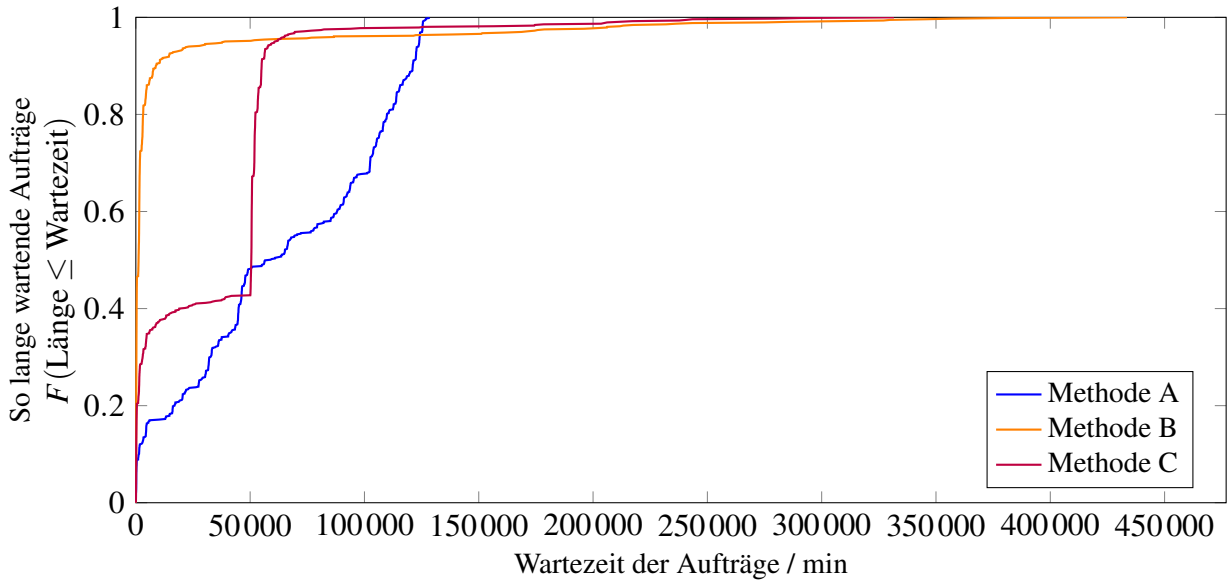


Abbildung 4.2: `fahrradwerkstatt1.txt`

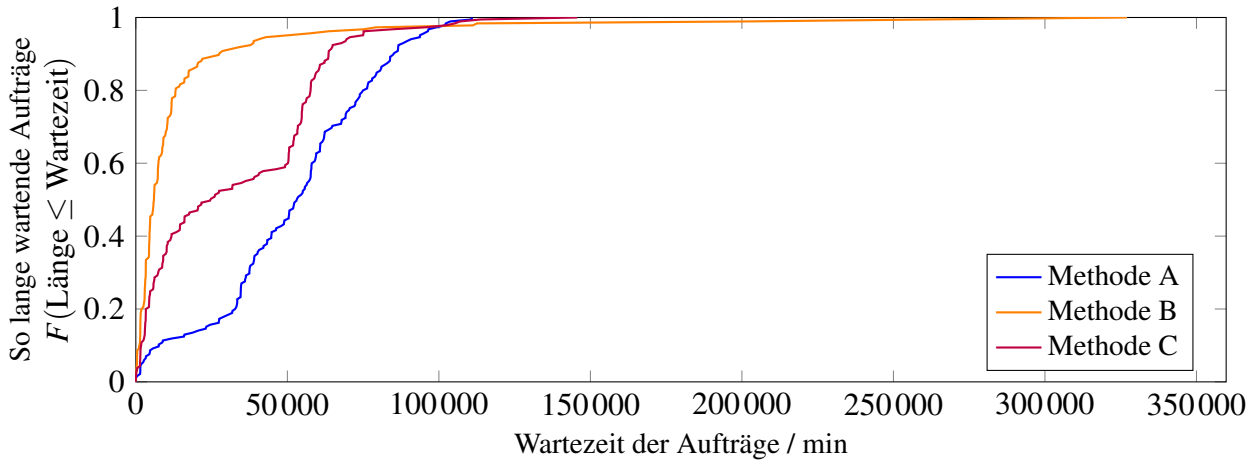


Abbildung 4.3: `fahrradwerkstatt2.txt`

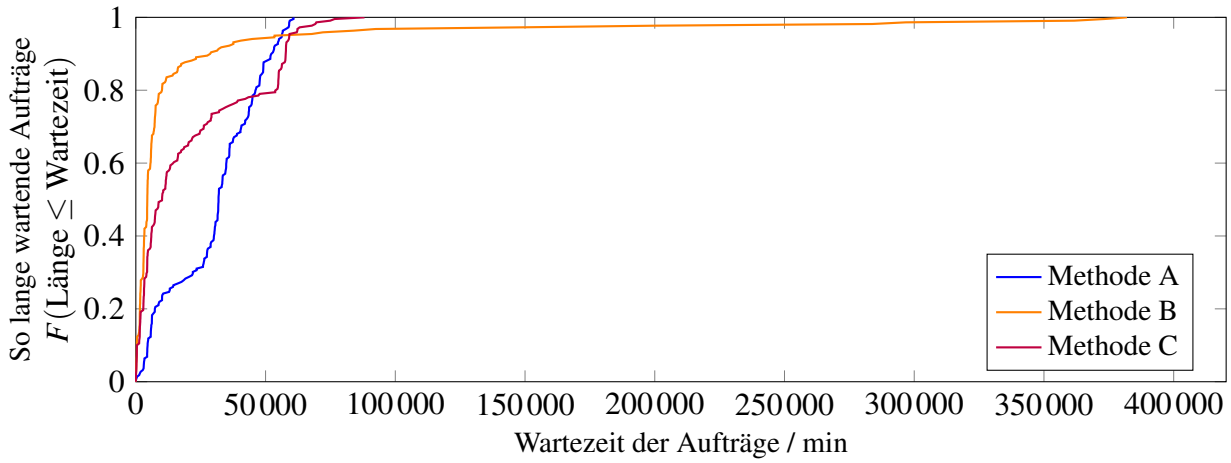


Abbildung 4.4: `fahrradwerkstatt3.txt`

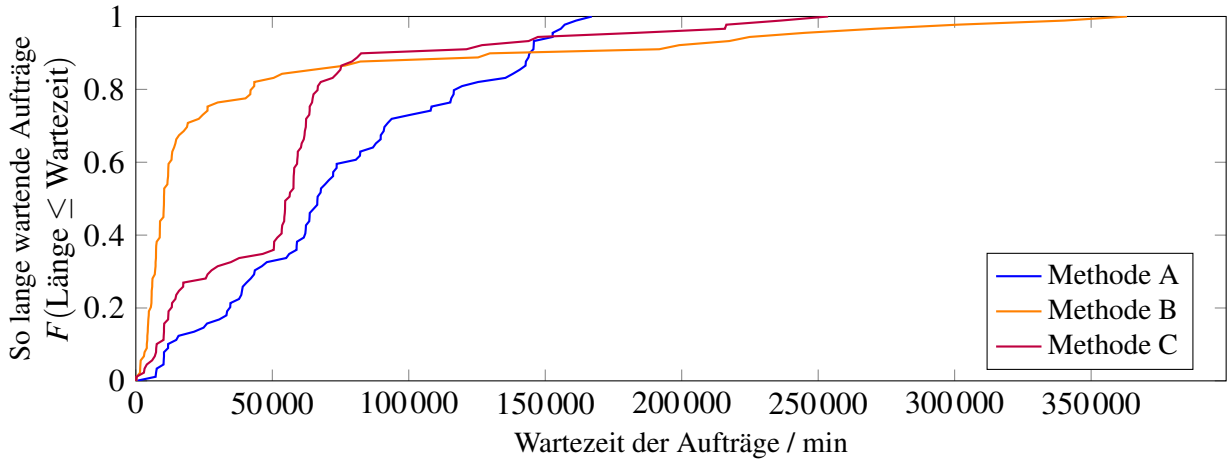


Abbildung 4.5: `fahrradwerkstatt4.txt`

Ob es sich für Marc lohnen würde, einen weiteren Kollegen einzustellen, ist jedoch fraglich. Zwar sind alle Kenngrößen deutlich besser und die Aufträge werden allesamt bedeutend schneller abgearbeitet, wie in Abbildung 4.6 dargestellt ist, allerdings haben sowohl Marc als auch sein neuer Kollege nur eine Auslastung von um die 50%. Die guten Ergebnisse sind also schlichtweg dadurch möglich, dass beide einen Großteil der Zeit warten und jeden Auftrag sofort beginnen können.

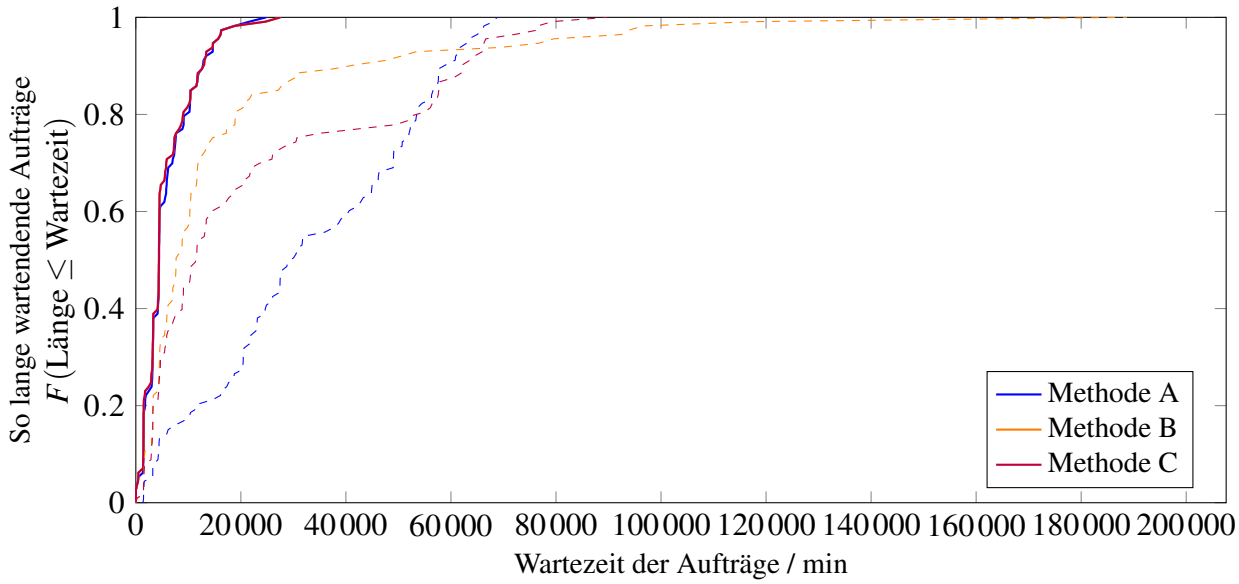


Abbildung 4.6: `fahrradwerkstatt0.txt` mit zwei Arbeitern. Die Ergebnisse mit nur einem Arbeiter sind als gestrichelte Linie hinterlegt.

4.4 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- [−2] **Simulation fehlerhaft**
  Es müssen mehrere Bedingungen eingehalten werden, insbesondere:
  - Der 9–17-Uhr-Arbeitstag von Marc.
  - Die Aufträge sind erst bekannt, wenn sie eingehen. Marc ist nicht „allwissend“ und kann daher nicht vorausplanen.
  - Aufträge dürfen nicht unterbrochen werden, wenn sie begonnen wurden.
  - Die Reihenfolge der Abarbeitung der Aufträge muss dem simulierten Verfahren entsprechen.

  Wenn die berechneten Werte für beide Verfahren denen der Beispiellösung mit einer Abweichung von höchstens $\approx 1\%$ entsprechen, dann kann davon ausgegangen werden, dass alle Bedingungen korrekt umgesetzt sind. Wenn eine Bedingung nicht korrekt umgesetzt wird, dann kommt es zu einem Punkt Abzug, bei mehreren Bedingungen und sich daraus ergebenden deutlichen Abweichungen werden zwei Punkte abgezogen.

  Es müssen mindestens die durchschnittliche und maximale Wartezeit berechnet werden, ansonsten wird ein Punkt abgezogen.
- [−1] **Geforderte Verfahren nicht implementiert**
  Es müssen auf jeden Fall die beiden Verfahren aus der Aufgabenstellung umgesetzt worden sein, sowie ein weiteres.
- [−1] **Verfahren nicht diskutiert**
  Vor allem muss abgewogen werden, welche Vor- und Nachteile die Verfahren jeweils haben. Für das zusätzlich entwickelte Verfahren wäre eine Diskussion schön, aber das Fehlen einer solchen führt nicht zu Punktabzug.
- [−1] **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Es ist nicht schwer, eine quasilineare Laufzeit der Simulation zu erreichen. Eine quadratische Laufzeit ist aber schnell genug, um die BWINF-Beispiele zu lösen, und wird deshalb auch akzeptiert. Nur bei noch schlechteren Laufzeiten wird hier ein Punkt abgezogen.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Es führt nicht zu Punktabzug, wenn die Kennzahlen in Stunden angegeben sind, sofern die Ergebnisse nachvollziehbar sind. Sind die Kennzahlen hingegen nur in Tagen ohne Nachkommastellen angegeben, so wird ein Punkt abgezogen.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens 3 der vorgegebenen Beispiele (`fahrradwerkstatt0.txt` bis `fahrradwerkstatt4.txt`) enthalten.

Aufgabe 5: Hüpfburg

5.1 Lösungsidee

Der Parcours kann als Graph interpretiert werden. Ein Graph ist eine mathematische Struktur aus Knoten und Kanten, wobei Knoten beliebige Objekte sein können und die Kanten Verbindungen zwischen diesen Objekten beschreiben.

Auf den Parcours übertragen entspricht jedes Feld einem Knoten und jeder Pfeil einer Kante. Typische Algorithmen und Methoden zum Umgang mit Graphen lassen sich jedoch nicht direkt auf die Aufgabenstellung anwenden, da wir zwei Akteure haben, die sich auf diesem Graphen bewegen sollen.

Eine Möglichkeit zum Lösen der Aufgabe besteht darin, die Anzahl der Akteure zu reduzieren. Dazu schauen wir uns nicht einzelne Felder, sondern immer Paare von Feldern an. Wenn wir mit n die Anzahl der Felder bezeichnen, gibt es für die beiden Spieler genau n^2 verschiedene Positionen auf dem Parcours.

Die Frage ist dann, ob Sasha und Mika von ihren gegebenen Anfangsfeldern tatsächlich eine der n Positionen erreichen können, bei der beide auf dem gleichen Feld stehen. Dazu können wir einen gerichteten Graphen konstruieren, bei dem die n^2 Positionen von Sasha und Mika die Knoten bilden und eine Kante genau dann zwischen zwei Knoten vorhanden ist, wenn beide Spieler in einem Schritt von dem Feld im ersten Knoten zum Feld im zweiten Knoten gelangen können.

Da die Startfelder von Sasha und Mika mit 1 bzw. 2 bekannt sind, können wir für diesen neu konstruierten Graphen die kürzesten Abstände zwischen dem Knoten $(1,2)$ und allen anderen Knoten berechnen. Insbesondere sehen wir dann auch, ob einer der Knoten

$$(1,1),(2,2),\ldots,(n,n)$$

erreicht werden kann. Falls ja, lässt sich mit einem Algorithmus zur Ermittlung des kürzesten Pfades von $(1,2)$ zu einem solchen Knoten auch die Folge von Sprüngen für Sasha und Mika ableiten, die benötigt wird, um auf dem gleichen Feld anzukommen.

Alternative Lösungsidee

Der als Graph modellierte Parcours kann auch mithilfe einer Adjazenzmatrix dargestellt werden. Für die Positionen von Sasha und Mika wählen wir einen probabilistischen Ansatz und modellieren die jeweiligen Standorte auf dem Graphen als Einheitsvektor über alle Felder. Per Matrix-Vektor-Multiplikation der Standortsvektoren mit der normierten Adjazenzmatrix können wir die Wahrscheinlichkeiten berechnen, mit denen sich Sasha bzw. Mika nach einem Schritt auf einem Feld befinden, wenn sie einen Pfeil von der jeweils aktuellen Position zufällig auswählen. Wir können also wiederholt die normierte Adjazenzmatrix mit den Standortvektoren von Sasha und Mika multiplizieren und nach jedem Schritt die Standortvektoren auf einen Nichtnull-Eintrag für das gleiche Feld überprüfen. Finden wir einen solchen Eintrag, heißt das, dass sich die beiden Spieler mit einer positiven Wahrscheinlichkeit nach der gleichen Anzahl an Schritten auf dem gleichen Feld befinden können – es muss also eine Lösung geben.

Grundsätzlich ist ein solcher Lösungsweg möglich und akzeptabel. Insbesondere muss aber auch die Abbruchbedingung bei Nichtexistenz einer Lösung begründet werden. Auch die Rekonstruktion einer Folge von Sprüngen ist in diesem Fall wesentlich aufwendiger.

5.2 Umsetzung

Der mit Kreide auf dem Schulhof gemalte Parcours kann als gerichteter Graph $G = (V,E)$ mit Knoten V und Kanten E interpretiert werden. Jedes Feld des Parcours entspricht genau einem Knoten $v \in V$ und jeder Pfeil genau einer Kante $e \in E \subseteq V \times V$.

Somit lässt sich ein neuer Graph $G' = (V',E')$ mit

$$V' := V \times V \text{ und}$$
$$E' := \{((v_1,w_1),(v_2,w_2)) \mid (v_1,v_2),(w_1,w_2) \in E\} \subseteq V' \times V'$$

konstruieren.

Der Graph G' ist ungewichtet. Somit können kürzeste Pfade vom Startknoten $(1,2)$ zu allen anderen Knoten über eine Breitensuche ermittelt werden.

Bei einer Breitensuche wird eine Liste oder Warteschlange noch zu besuchender Knoten angelegt. Anfangs besteht diese Liste nur aus dem Startknoten. Jeweils der erste Knoten aus dieser Liste wird besucht und damit entfernt. Alle von diesem Knoten aus erreichbaren Knoten werden der Liste am Ende hinzugefügt. Um einen Pfad zu den angehängten Knoten rekonstruieren zu können, speichern wir für jeden angehängten Knoten den aktuell besuchten Knoten als Vorgänger. Dies stellt sicher, dass der erste Besuch eines Knotens über einen kürzest möglichen Pfad erfolgt. Um zu verhindern, dass Knoten mehrfach besucht werden, speichern wir außerdem für jeden Knoten, ob wir diesen bereits besucht haben.

Da die Breitensuche alle vom Startknoten aus erreichbaren Knoten besucht, wissen wir mit Sicherheit, dass keine Lösung für die Aufgabenstellung existiert, wenn kein geeigneter Knoten während der Breitensuche gefunden wurde.

Zusammenfassend beschreiben wir die Lösung der Aufgabe in Algorithmus 10.

5.3 Laufzeit

Neben der Frage, ob eine Lösung möglich ist, ist es auch interessant und meist wichtig zu wissen, wie gut der Algorithmus skaliert bzw. wie sich die Größe der Eingaben auf die Laufzeit auswirkt. Dazu benutzen wir die in der Informatik gebräuchliche $\mathcal{O}$-Notation.

Wir analysieren die Laufzeit anhand von Algorithmus 10 unter Bezugnahme auf die Graph-Darstellung $G = (V,E)$:

- Für die Konstruktion des Graphen G' müssen wir alle Knoten in V und alle Paare von Kanten in E durchlaufen, also ergibt sich eine Laufzeit von $\mathcal{O}(|V| + |E|^2)$.
- Für die Breitensuche müssen wir im schlechtesten Fall alle Knoten und alle Kanten in G' nochmals betrachten. Da $|V'| = |V|^2$ und $|E'| = |E|^2$ also $\mathcal{O}(|V|^2 + |E|^2)$.
- Für die Rekonstruktion des Pfades müssen die auf dem Weg dahin besuchten Knoten nochmals abgelaufen werden, also $\mathcal{O}(|V|^2)$.

Insgesamt ergibt sich damit für den Algorithmus eine Worst-Case-Laufzeit von

$$\mathcal{O}(|V| + |E|^2) + \mathcal{O}(|V|^2 + |E|^2) + \mathcal{O}(|V|^2) = \mathcal{O}(|V|^2 + |E|^2).$$

Algorithmus 10 Hüpfburg-Algorithmus

Require: $G = (V, E)$ als Repräsentation des Parcours
1: $V' \leftarrow \{(v, v)) \mid v \in V\}$ ▷ $G' = (V, E')$ konstruieren
2: $E' \leftarrow \{((v_1, w_1), (v_2, w_2)) \mid (v_1, v_2), (w_1, w_2) \in E\}$
3: **for all** $v' \in V'$ **do**
4: visited$[v']$ ← **false**
5: visited_from$[v']$ ← **null**
6: **end for**
7: queue ← $\{(1, 2)\}$ ▷ Breitensuche auf G' durchführen
8: **while** queue ≠ ∅ **do**
9: node ← queue.popLeft()
10: neighbors ← $\{v' \in V' \mid (\text{node}, v') \in E'\}$
11: **for all** neighbor ∈ neighbors **do**
12: **if not** visited[neighbor] **then**
13: visited[neighbor] ← **true**
14: visited_from[neighbor] ← node
15: queue.appendRight(neighbor)
16: **end if**
17: **end for**
18: **if** node ∈ $\{(v, v) \mid v \in V\}$ **then** ▷ Lösung gefunden
19: current_node ← node ▷ Folge von Sprüngen konstruieren
20: path_sasha ← []
21: path_mika ← []
22: **while** current_node ≠ $(1, 2)$ **do**
23: (v, w) ← current_node
24: path_sasha ← $[v]$ + path_sasha
25: path_mika ← $[w]$ + path_mika
26: current_node ← visited_from[current_node]
27: **end while**
28: **Output:** „Lösung gefunden:“, path_sasha, path_mika
29: **return**
30: **end if**
31: **end while**
32: **Output:** „Keine Lösung gefunden“

5.4 Beispiele

Für die vorgegebenen Beispiele ergeben sich folgende Lösungen.

huepfburg0.txt

Lösung vorhanden

Länge des Pfades: 3

Pfad Sasha: $1 \to 18 \to 13 \to 10$

Pfad Mika: $2 \to 19 \to 20 \to 10$

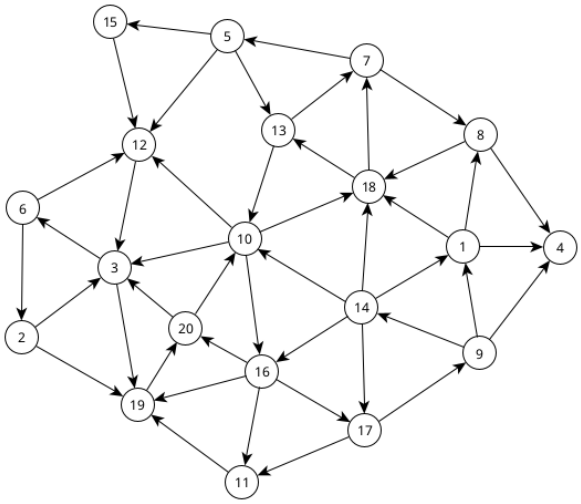


Abbildung 5.1: Hüpfburg 0

huepfburg1.txt

Lösung vorhanden

Länge des Pfades: 121

Pfad Sasha: 1 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 4

Pfad Mika: 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12 → 13 → 14 → 15 → 16 → 17 → 1 → 2 → 3 → 4

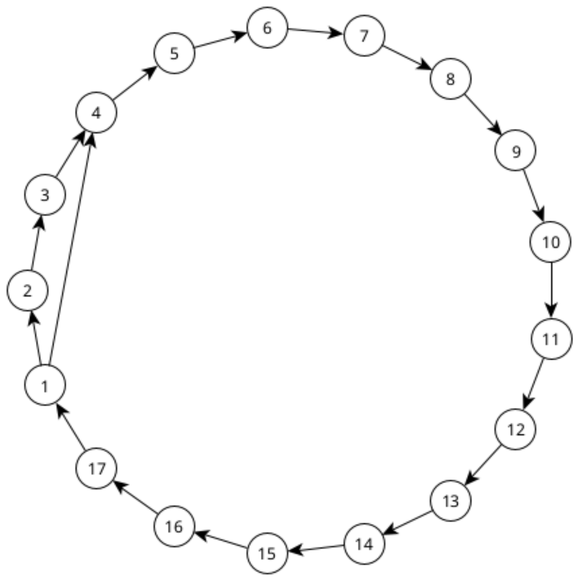


Abbildung 5.2: Hüpfburg 1

huepfburg2.txt

Lösung vorhanden

Länge des Pfades: 8

Pfad Sasha: $1 \to 51 \to 76 \to 59 \to 42 \to 65 \to 54 \to 92 \to 27$

Pfad Mika: $2 \to 24 \to 53 \to 2 \to 106 \to 136 \to 108 \to 100 \to 27$

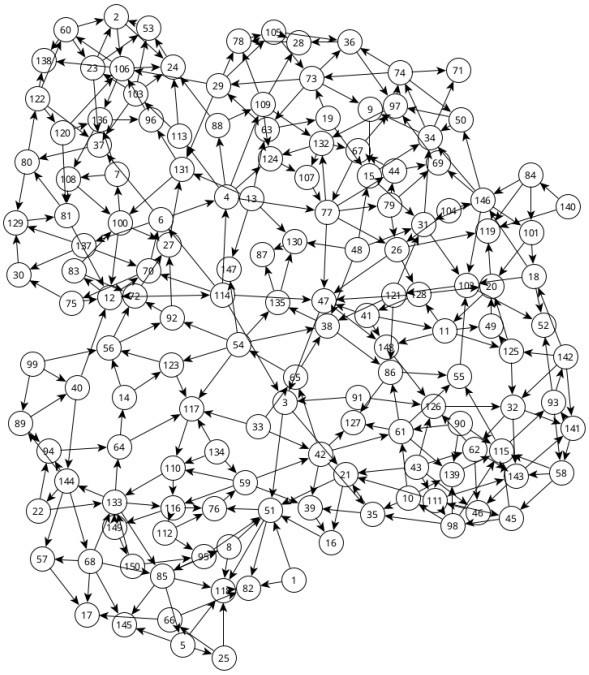


Abbildung 5.3: Hüpfburg 2

huepfburg3.txt

Keine Lösung vorhanden

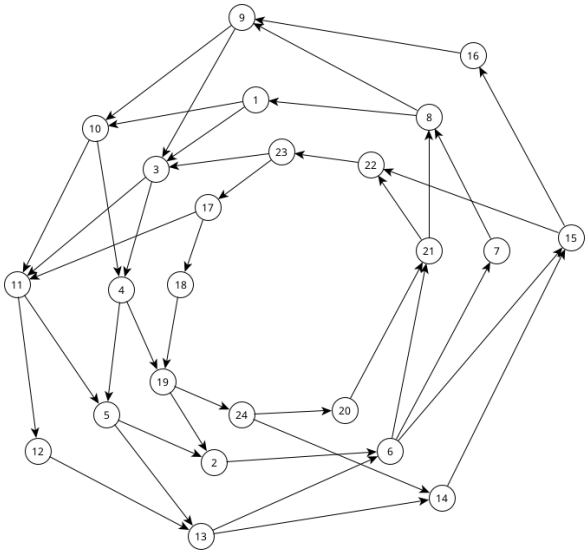


Abbildung 5.4: Hüpfburg 3

huepfburg4.txt

Lösung vorhanden

Länge des Pfades: 16

Pfad Sasha: $1 \to 99 \to 89 \to 79 \to 78 \to 77 \to 76 \to 66 \to 56 \to 55 \to 54 \to 44 \to 43 \to 33 \to 23 \to 13 \to 12$

Pfad Mika: $2 \to 12 \to 11 \to 100 \to 2 \to 12 \to 11 \to 100 \to 2 \to 12 \to 11 \to 100 \to 2 \to 12 \to 11 \to 100 \to 12$

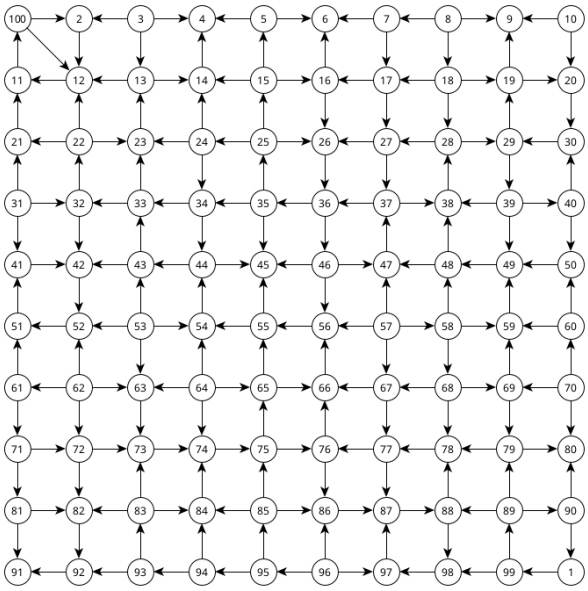


Abbildung 5.5: Hüpfburg 4

5.5 Bewertungskriterien

Die Bewertungskriterien vom Bewertungsbogen werden hier erläutert (Punktabzug in []).

- $[-1]$ **Modellierung fehlerhaft**
  Die Eingabe muss korrekt in eine Datenstruktur überführt und verwendet werden. Es muss insbesondere beachtet werden, dass die Felder durch Pfeile verbunden sind und nur in eine Richtung gesprungen werden kann.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Das Lösungsverfahren muss korrekt bestimmen, ob es eine Lösung gibt.

  Es ist zum Beispiel nicht erlaubt, eine Person „zu parken“, d. h. sie auf einem Knoten warten statt springen zu lassen, selbst bei einer etwaigen Senke im Graphen. Dies würde Regel 2) verletzen und ein Punkt würde abgezogen.

  Bei iterativen Verfahren muss eine Abbruchbedingung eingeführt werden.

  Die bei vorhandener Lösung ausgegebenen Sprungfolgen für Sasha und Mika müssen korrekt sein, d. h. möglich und so, dass am Ende beide auf demselben Feld stehen. Dabei ist nicht gefordert, dass die Anzahl an Sprüngen minimal ist.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Der Aufwand des Verfahrens soll nicht schlechter sein als $\mathcal{O}(|V|^4 + |E|^3)$.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Wenn für Sasha und Mika jeweils eine Folge von Sprüngen existiert, an deren Ende beide auf demselben Feld stehen, müssen diese ausgegeben werden. Ansonsten muss explizit gesagt werden, dass keine Lösung existiert. Die Länge des Pfades kann angegeben werden, ist aber nicht gefordert.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens 3 der vorgegebenen Beispiele (`huepfburg0.txt` bis `huepfburg4.txt`) enthalten, darunter `huepfburg3.txt`.