44. Bundeswettbewerb Informatik 2025/2026, 1. Runde

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; das ist einer der wichtigsten Effekte einer Wettbewerbsteilnahme.

Die Bearbeitungszeit für die 1. Runde beträgt gut zweieinhalb 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 *Quelltext* bestehen soll. Bei dieser 1. Runde wurde außerdem erstmalig erwartet, dass in einem Abschnitt *Werkzeuge* alle zur Bearbeitung einer Aufgabe verwendeten Werkzeuge genannt werden und für jedes Werkzeug erläutert wird, wie es eingesetzt wurde. 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 sein sollte. 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 *Quelltext*, zumindest dessen für die Lösung wichtigen Teile, gehört in die Dokumentation; Quelltext 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, 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 prüfen 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 kritischer Analyse 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 von Bearbeitungen 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 5 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 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 vom Aufgabenblatt: „Im Bundeswettbewerb sind die Junioraufgaben Schülerinnen und Schülern vor der Qualifikationsphase des Abiturs vorbehalten“. 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, und Jürgen Lerner sowie (als Gäste) Greta Niemann, Wolfgang Pohl und Hannah Rauterberg.

An der Erstellung der im Folgenden skizzierten Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Personen mit: Christina Suttrop (Junioraufgabe 1), Tim Pokart (Junioraufgabe 2), Selma Hübner (Aufgabe 1), Martin Bartram (Aufgabe 2), Raphael Gaedtke (Aufgabe 3), Shuheng Wei (Aufgabe 4) und Philip Gilde (Aufgabe 5). Allen Beteiligten sei für ihre Mitarbeit ganz herzlich gedankt.

Junioraufgabe 1: Bällebad

J1.1 Lösungsidee

Ein Möglichkeit, um die maximal benötigte Anzahl an Medizinbällen zu ermitteln, besteht darin, einen Nutzungsplan aus den gegebenen Sporteinheiten zu erstellen.

Dafür erstellen wir einen Stundenplan, der die Anzahl benötigter Bälle pro Sporteinheit eines Schultages beinhaltet. Zunächst ist diese Anzahl für jede Stunde eines Tages 0. Nun betrachten wir jede Sporteinheit und addieren die Zahl an erforderlichen Medizinbällen zur Anzahl der Bälle, die bereits während dieser Stunde benötigt wird. So verfahren wir mit jedem Sporteinheiteneintrag aus der Eingabe, bis wir alle Einheiten in den Nutzungsplan eingetragen haben.

Um nun den Maximalbedarf an Medizinbällen zu bestimmen, gehen wir einmal durch den Nutzungsplan und finden so die Stunden mit der größten Anzahl an benötigten Bällen.

Überlegungen zur Laufzeit

Um die Laufzeit des Algorithmus abzuschätzen, betrachten wir die dessen verschiedene Teile. Für das Eintragen der n Sporteinheiten in den Nutzungsplan benötigen wir n Operationen. Beim Finden der Stundeneinträge mit dem höchsten Ballbedarf untersucht der Algorithmus jede Stunde des Nutzungsplans. Da die Anzahl der Stunden im Nutzungsplan konstant ist, sich also nicht verändert, können diese Operationen bei der Laufzeitabschätzung vernachlässigt werden. Somit kommen wir auf eine Laufzeit von maximal n Operationen. Das können wir schreiben als $\mathcal{O}(n)$.

J1.2 Umsetzung

Bei der Umsetzung ist es wichtig zu beachten, dass der Bällebedarf einer Sporteinheit nicht doppelt gezählt wird. Dies kann man zum Beispiel dadurch realisieren, dass die Startzeit einer Sporteinheit inklusiv, aber die Endzeit exklusiv behandelt wird. Startet eine Einheit beispielsweise um 13:00 und endet um 15:00 Uhr, werden die Bälle nur in den Stunden ab 13:00 und 14:00 Uhr benötigt, nicht aber in der Stunde ab 15:00, da die Sporteinheit bereits beendet ist. Weiterhin können die Bälle am Ende einer Einheit problemlos an eine direkt darauf folgende weitergegeben werden.

Des Weiteren muss der zeitliche Rahmen des Nutzungsplans festgelegt werden. Grundsätzlich wäre ein Plan, der eine gesamte Woche von Montag bis Sonntag täglich von 00:00 bis 23:59 Uhr umfasst, in Ordnung. Effizienter und daher sinnvoller wäre es aber, wenn der Plan nur die Zeiten enthält, zu denen tatsächlich Unterricht stattfinden kann. Mit den gegebenen Beispieleingaben wäre das von Montag bis Freitag, jeweils von 07:00 bis 21:00 Uhr.

Alternativ kann man die gesamte Schulwoche als Liste von Einheiten mit dem jeweiligen Bällebedarf modellieren und implementieren, ohne dass die Zeiten näher betrachtet werden.

Mehrere Stunden mit maximaler Nutzung

Für die Umsetzung muss auch entschieden werden, was im Falle mehrerer Stunden mit maximaler Ballnutzung geschieht. Mehrere solcher Stunden können aus verschiedenen Gründen

auftreten. Das Programm sollte mit diesem Fall umgehen können. Ein Blick in die Aufgabenstellung zeigt, dass nur die Ausgabe von einem Zeitpunkt mit maximaler Nutzung gefordert ist.[1] Von daher ist die Ausgabe einer Stunde mit maximaler Nutzung ausreichend. Für die Schulleitung praktischer wäre jedoch die Angabe aller Stunden mit dem größten Bedarf an Bällen. Dafür wäre dann eine geeignete Datenstruktur zum Speichern mehrerer Stunden von Nöten. Die Anzahl an Bällen kann wie bei der Ausgabe von nur einer Stunde gespeichert werden, da sie über alle Stunden mit maximaler Nutzung gleich ist.

J1.3 Beispiele

ball00.txt

Die maximale Anzahl an benötigten Medizinbällen ist 60. So viele Bälle werden benötigt am:

- Montag um 14 Uhr

ball01.txt

Die maximale Anzahl an benötigten Medizinbällen ist 96. So viele Bälle werden benötigt am:

- Mittwoch um 11 Uhr

ball02.txt

Die maximale Anzahl an benötigten Medizinbällen ist 157. So viele Bälle werden benötigt am:

- Montag um 10 Uhr

ball03.txt

Die maximale Anzahl an benötigten Medizinbällen ist 152. So viele Bälle werden benötigt am:

- Freitag um 09 Uhr

ball04.txt

Die maximale Anzahl an benötigten Medizinbällen ist 31. So viele Bälle werden benötigt am:

- Montag um 10 Uhr und Montag um 11 Uhr

ball05.txt

Die maximale Anzahl an benötigten Medizinbällen ist 60. So viele Bälle werden benötigt am:

- Donnerstag um 08 Uhr und Donnerstag um 09 Uhr

[1] „[Das Programm] soll **einen** Zeitpunkt angeben, an welchem diese Maximalzahl benötigt wird.“ (Aufgabenstellung 1. Runde des 44. Bundeswettbewerb Informatik mit eigenen Hervorhebungen)

ball06.txt

Die maximale Anzahl an benötigten Medizinbällen ist 90. So viele Bälle werden benötigt am:

- Dienstag um 09 Uhr und Dienstag um 10 Uhr

ball07.txt

Die maximale Anzahl an benötigten Medizinbällen ist 59. So viele Bälle werden benötigt am:

- Freitag um 10 Uhr, Freitag um 11 Uhr, Freitag um 12 Uhr und Freitag um 13 Uhr

J1.4 Bewertungskriterien

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

- [−1] **Modellierung ungeeignet**
  - Unabhängig von der gewählten Ausgabe soll die Anzahl maximal benötigter Bälle nur einmal gespeichert werden.
  - Die Sporteinheiten müssen so modelliert werden, dass der Bällebedarf einer Sporteinheit nicht für zwei Einheiten gezählt wird.
  - Es ist unnötig aufwändig, die Zeit in Minuten statt Stunden zu speichern.
- [−1] **Lösungsverfahren fehlerhaft**
  Der maximale Wert mit mindestens einem entsprechenden Zeitpunkt muss korrekt berechnet werden.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Es wird deutlich, wie viele Bälle maximal benötigt werden. Außerdem wird mindestens ein Zeitpunkt (bestehend aus Tag und Uhrzeit) mit maximaler Nutzung ausgegeben.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens fünf der Beispiele (`ball00.txt` bis `ball07.txt`) enthalten.

  Von `ball04.txt` bis `ball07.txt` darf maximal ein Beispiel fehlen.

Beispiel	Maximum	Zeitpunkt
`ball00.txt`	60	Mo 14 Uhr
`ball01.txt`	96	Mi 11 Uhr
`ball02.txt`	157	Mo 10 Uhr
`ball03.txt`	152	Fr 9 Uhr
`ball04.txt`	31	Mo 10 Uhr, Mo 11 Uhr
`ball05.txt`	60	Do 8 Uhr, Do 9 Uhr
`ball06.txt`	90	Di 9 Uhr, Di 10 Uhr
`ball07.txt`	59	Fr 10 Uhr, Fr 11 Uhr, Fr 12 Uhr, Fr 13 Uhr

Junioraufgabe 2: Sil-ben-tren-nung

J2.1 Lösungsidee

In dieser Aufgabe soll Lars geholfen werden, ein Silbentrennungsprogramm zu entwickeln, welches für gegebene Wörter die Schreibsilben markiert. Schreibsilben sind dabei solche, welche von Textverarbeitungsprogrammen für Zeilenumbrüche verwendet werden können. Grundsätzlich ist für eine korrekt funktionierende Silbentrennung ein Wörterbuch erforderlich. So existieren für manche Wörter Sonderregeln; im Deutschen ist das etwa bei zusammengesetzten Wörtern der Fall, die grundsätzlich an der Verbindungsstelle getrennt werden. Beispiel: Lang·schwert. Hier wollen wir uns aber auf einen Lösungsansatz beschränken, der durch einfache Regeln abhängig von einzelnen Buchstaben oder Buchstabenkombinationen Trennstellen vorschlägt.

In dem Programm kann dann in jedem Wort für jeden Buchstaben anhand dessen Vorgängers und einiger Nachfolger entschieden werden, ob eine Textsilbe getrennt werden kann. Für die Lösung werden später Buchstabenbezeichnungen relevant, die wir hier kurz einführen wollen: Die Buchstaben “`a`”, “`e`”, “`i`”, “`o`” und “`u`” sowie deren Umlaute “`ä`”, “`ö`” und “`ü`” behandeln wir als Vokale. Alle Buchstaben, die keine Vokale sind, sind Konsonanten; Ausnahme ist das “`y`”, für das wir beide Fälle – die Behandlung als Vokal und die als Konsonant – betrachten. Als Punktierung betrachten wir “`.`”, “`,`”, “`-`”, “`?`” und “`!`”; diese Zeichen werden nicht verändert und dürfen nicht eigenständig umgebrochen werden. Die Gruppen “`ei`”, “`ai`”, “`ui`”, “`eu`”, “`äu`” und “`au`” sind als Diphthong (Zwielaut) solche Buchstabenkombinationen, die in der Regel einer Silbe zuzuordnen sind. Aus Konsonanten kann man die für uns relevanten Kombinationen “`ch`”, “`sch`”, “`ph`”, “`rh`”, “`sh`”, “`th`” und “`ck`” bilden. Ähnlich den Diphtongen werden diese ebenfalls zwangsläufig einer Silbe zugeordnet.

Zunächst wollen wir Lars' Regeln umsetzen. Dabei sind für Lars' dritte Regel zwei unterschiedliche Interpretationen zulässig. Um diesen Spielraum aufzulösen sind hier Lars' Regeln leicht abgewandelt formuliert, aber inhaltlich gleich denen in der Aufgabenstellung wiedergegeben. Bei der Formulierung der Regeln gehen wir davon aus, dass ein Verfahren zur Ermittlung von Trennstellen eine Buchstabenfolge (ein Wort oder einen Satz) Zeichen für Zeichen durchläuft und dabei nach und nach das Zeichen an einer Position der Folge betrachtet. Die Regeln müssen von unten nach oben priorisiert werden und lauten:

1. Ist an der aktuellen Position ein Konsonant und folgt ebenfalls ein Konsonant, darf nach der Position getrennt werden.
2. Wenn die aktuelle Position die erste oder letzte im Wort ist darf danach bzw. davor nicht abgetrennt werden.

3.[a] Nach einer Position kann nicht getrennt werden, wenn an vorheriger Position ein Konsonant steht.[2]

3.[b] Ist die aktuelle Position am Beginn einer Gruppe von drei oder mehr Konsonanten, wird danach getrennt.

4. Nach der aktuellen Position wird getrennt, wenn sie ein Vokal ist und nicht zwei Konsonanten folgen; insbesondere also wenn vom Wort nur ein Buchstabe bleibt.

[2] Diese Regel versteht etwas strikter die Bedingung, die Lars' mit “kann nur” formuliert. Hier wird “kann nur” so interpretiert, dass die Regel nur eine Silbentrennung verbieten könnte, aber nicht automatisch eine Silbentrennung erlaubt. Mit der gewählten Formulierung wird dann automatisch Lars' Vorgabe erfüllt, dass bei drei oder mehr Konsonanten nur der erste umgebrochen werden kann.

Diskussion

Aus diesen Regeln ergeben sich einige Besonderheiten, die nicht mit den üblichen Silbentrennregeln der deutschen Sprache kompatibel sind. Am auffälligsten ist dabei wahrscheinlich, dass die Regel Nr. 2 nicht die höchste Priorität hat und dadurch manchmal Wörter nach dem ersten oder vor dem letzten Buchstaben getrennt werden. Beispiele hierfür sind "Es" (`silben03.txt`), "im" (`silben04.txt`) und "Mais" (`silben05.txt`). Außerdem werden Blöcke von Konsonanten nach Lars' Regeln immer nach dem ersten Konsonanten getrennt, wobei im Deutschen üblicherweise genau umgekehrt vor dem letzten Konsonanten die Silbe endet – wenn es sich nicht um ein zusammengesetztes Wort handelt, man beachte den Titel der Aufgabe. Lars missachtet bei der Trennung ebenfalls untrennbare Verbindungen wie die oben erwähnten Diphtonge und Konsonantkombinationen.

Erweiterung

Als Verbesserung kommen beispielsweise Umordnungen, Ergänzungen oder Ersetzungen von Lars' Regeln in Frage. Hierfür wollen wir uns am Amtlichen Regelwerk der deutschen Rechtschreibung[3] orientieren. Daraus lässt sich der folgende erweiterte Regelsatz ableiten, der wie folgt formuliert als vollständig neuer Regelsatz implementiert wird:

1. Bei einer Folge von einem oder mehreren Konsonanten wird immer vor dem letzten getrennt, siehe § 87.
2. Die vorher definierten Konsonantkombinationen werden nicht getrennt und stattdessen als ein Konsonant behandelt, siehe § 88.
3. Ist die aktuelle sowie die folgende Position ein Vokal und die beiden Vokale bilden keinen Diphtong, wird eine Silbengrenze eingefügt. Dies implementiert § 86.
4. Am Wortanfang oder -ende darf keine Silbentrennung eingefügt werden, was exakt Lars' Regel Nr. 2 entspricht.

Der § 85 als Regel für zusammengesetzte Wörter sowie der § 89 als Regel für Fremdwörter verfallen, weil das Programm keine Möglichkeit hat solche zu erkennen. Obwohl es sich um einen neuen Regelsatz handelt, sind die Regeln mit denen von Lars verwandt; so ersetzt hier die Regel Nr. 3 Lars' Vokalregel Nr. 4. Aus der Trennung nach dem ersten Konsonanten in Lars' Regel Nr. 3 wird in den erweiterten Regeln eine Trennung nach dem letzten Konsonanten eines Blocks. Die Regel Nr. 2 hat kein Äquivalent bei Lars.

J2.2 Umsetzung

Sowohl Lars' Regeln als auch unsere Erweiterungen haben eine vergleichsweise einfache Struktur. Diese ermöglicht es uns, an jeder Position im Wort bzw. Satz basierend auf dem aktuellen Buchstaben sowie dessen Vorgänger und Nachfolger zu entscheiden, ob eine Silbentrennung möglich ist oder nicht. Formell entspricht dies der Schwierigkeit, eine reguläre Sprache zu erkennen. Bei Lars' Regeln reicht es, hierbei Vokale und Konsonanten zu unterscheiden. In den Regeln der Erweiterung müssen zusätzlich die Konsonantkombinationen und Zwielaute als

[3] Siehe Amtliches Regelwerk 2024, Kapitel F, §§ 86, 87, 88.

untrennbare Einheiten beachtet werden. Für die Konsonantkombinationen wird dies dadurch erreicht, dass diese zunächst durch ein gesondertes Schriftzeichen ersetzt werden, welches dann wie ein Konsonant behandelt werden kann.

Etwas konkreter wollen wir nun beispielhaft die Programmstruktur für Lars' Regeln erklären, mit der Variante 3^b für Lars' dritte Regel. Kernelement der Implemenierung ist eine Funktion SILBEDANACH, die für eine Position in einem Wort TRUE zurückgibt, wenn nach dem dortigen Buchstaben eine Silbentrennung eingefügt werden soll; andernfalls wird FALSE zurückgegeben.

```
function SILBEDANACH(Wort w, Position p)
    Seien b_{p-1}, b_p, b_{p+1} und b_{p+2} der vorherige, aktuelle, nächste und übernächste Buchstabe
    Definiere ZWEIKONSONANTENFOLGEN ← b_{p+1} und b_{p+2} sind Konsonanten
    if b_p ist Vokal und nicht ZWEIKONSONANTENFOLGEN then                ▷ Regel 4
        return TRUE
    end if
    if b_p ist Konsonant then                                            ▷ Regel 3^b
        if b_{p-1} ist Konsonant then                                    ▷ Nur der erste einer Gruppe
            return FALSE
        else if ZWEIKONSONANTENFOLGEN then
            return TRUE
        end if
    end if
    if p ist am Wortanfang oder eins vor Wortende then                   ▷ Regel 2
        return FALSE
    end if
    if b_p ist Konsonant und b_{p+1} ist Konsonant then                  ▷ Regel 1
        return TRUE
    end if
    return FALSE
end function
```

Gibt diese Funktion nun TRUE zurück, wird an der entsprechenden Stelle eine Silbentrennung eingefügt. Danach wird mit der nächsten Position im Wort fortgefahren.

J2.3 Beispiele

In den folgenden Beispielen wird die korrekte Trennung aus dem Duden übernommen. Insofern sich Unterschiede durch die unterschiedliche Interpretation der dritten Regel ergeben, sind diese durch eine Unterstreichung markiert.

silben01.txt

Mein Name ist Lars und ich esse schrecklich gerne Sauerkraut.

Korrekte Trennung: Mein Na·me ist Lars und ich es·se schreck·lich ger·ne Sau·er·kraut.

Lars' Regeln:[a] Me·i·n Na·me ist Lars und ich es·se schrec·klich ger·ne Sa·u·er·kra·u·t.

Lars' Regeln:[b] Me·i·n Na·me ist Lars und ich es·se s·chrec·klich ger·ne Sa·u·er·kra·u·t.

Erweiterte Regeln: Mein Na·me ist Lars und ich es·se schreck·lich ger·ne Sau·erk·raut.

silben02.txt

Der Kapitän ist ebenfalls Präsident der örtlichen Dampfschifffahrtsgesellschaft.

Korrekte Trennung: Der Ka·pi·tän ist eben·falls Prä·si·dent
der ört·lichen Dampf·schiff·fahrts·ge·sell·schaft.

Lars' Regeln:[a,b] De·r Ka·pi·tä·n ist e·ben·fal·ls Prä·si·de·nt
de·r ör·tlic·he·n Dam·pfschif·ffah·rtsge·sel·lschaft.

Erweiterte Regeln: Der Ka·pi·tän ist eben·falls Prä·si·dent
der ört·li·chen Dampf·schiff·fahrts·ge·sell·schaft.

silben03.txt

Es ist arschkalt!

Korrekte Trennung: Es ist arsch·kalt!

Lars' Regeln:[a,b] E·s ist ar·schkalt!

Erweiterte Regeln: Es ist arsch·kalt!

silben04.txt

Ist das Audiosignal im Radio schlecht?

Korrekte Trennung: Ist das Au·dio·si·gnal im Ra·dio schlecht?

Lars' Regeln:[a] Ist da·s A·u·di·o·sig·na·l i·m Ra·di·o schlec·ht?

Lars' Regeln:[b] Ist da·s A·u·di·o·sig·na·l i·m Ra·di·o s·chlec·ht?

Erweiterte Regeln: Ist das Au·di·o·sig·nal im Ra·dio schlecht?

silben05.txt

Sein Vater ist Bauer und erntet Mais.

Korrekte Trennung: Sein Va·ter ist Bau·er und ern·tet Mais.

Lars' Regeln:[a,b] Se·i·n Va·te·r ist Ba·u·e·r und er·nte·t Ma·i·s.

Erweiterte Regeln: Sein Va·ter ist Bau·er und ern·tet Mais.

silben06.txt

Ich angle Karpfen

Korrekte Trennung: Ich an·gle Karp·fen

Lars' Regeln:[a,b] Ich an·gle Kar·pfe·n.

Erweiterte Regeln: Ich ang·le Karp·fen.

silben07.txt

Was sind acht Hobbits? Ein Hobbyte!

In der Variante mit “y” als Vokal:

Korrekte Trennung: Was sind acht Hob·bits? Ein Hob·byte!

Lars’ Regeln:[a,b] Wa·s sind ac·ht Hob·bits? E·i·n Hob·by·te!

Erweiterte Regeln: Was sind acht Hob·bits? Ein Hob·by·te!

In der Variante ohne “y” als Vokal:

Lars’ Regeln:[a,b] Wa·s sind ac·ht Hob·bits? E·i·n Hob·byte!

Erweiterte Regeln: Was sind acht Hob·bits? Ein Hobby·te!

silben08.txt

Freude, schöner Götterfunken, Tochter aus Elisium, Wir betreten feuertrunken, Himmlische, dein Heiligthum. Deine Zauber binden wieder, Was die Mode streng getheilt, Alle Menschen werden Brüder, Wo dein sanfter Flügel weilt.[4]

Korrekte Trennung: Freu·de, schö·ner Göt·ter·fun·ken, Toch·ter aus Eli·si·um, Wir be·tre·ten feu·er·trun·ken, Himm·lische, dein Hei·lig·thum. Dei·ne Zau·ber bin·den wie·der, Was die Mo·de streng ge·theilt, Al·le Men·schen wer·den Brü·der, Wo dein sanf·ter Flü·gel weilt.

Lars’ Regeln:[a] Fre·u·de, schö·ne·r Göt·ter·fun·ke·n, Toc·hte·r a·u·s E·li·si·u·m, Wi·r bet·re·te·n fe·u·er·trun·ke·n, Him·mlis·che, de·i·n He·i·lig·thu·m. De·i·ne Za·u·be·r bin·de·n wi·e·de·r, Wa·s di·e Mo·de streng get·he·ilt, Al·le Men·sche·n wer·de·n Brü·de·r, Wo de·i·n san·fte·r Flü·ge·l we·ilt.

Lars’ Regeln:[b] Fre·u·de, s·chö·ne·r Göt·ter·fun·ke·n, Toc·hte·r a·u·s E·li·si·u·m, Wi·r bet·re·te·n fe·u·er·trun·ke·n, Him·mlis·che, de·i·n He·i·lig·thu·m. De·i·ne Za·u·be·r bin·de·n wi·e·de·r, Wa·s di·e Mo·de s·treng get·he·ilt, Al·le Men·sche·n wer·de·n Brü·de·r, Wo de·i·n san·fte·r Flü·ge·l we·ilt.

Erweiterte Regeln: Freu·de, schö·ner Göt·ter·fun·ken, Toch·ter aus Eli·si·um, Wir bet·re·ten feu·ert·run·ken, Himm·li·sche, dein Hei·lig·thum. Dei·ne Zau·ber bin·den wi·e·der, Was die Mo·de st·reng ge·theilt, Al·le Men·schen wer·den Brü·der, Wo dein sanf·ter Flü·gel weilt.

[4] Hierbei handelt es sich um die Ode *An die Freude* von Friedrich Schiller, in ihrer späten Fassung.

J2.4 Bewertungskriterien

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

- $[-1]$ **Modellierung ungeeignet**
  Die Texte sollen korrekt eingelesen werden, dabei sind insbesondere die Umlaute zu beachten. Sonder- und Satzzeichen dürfen kein Teil der Trennung sein. Die Textposition sollte so modelliert werden, dass die Regeln gut umsetzbar sind.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Die Implementierung soll deterministisch alle möglichen Trennstellen bestimmen. Dabei müssen die von Lars vorgegebenen Regeln beziehungsweise die eigene Erweiterung im Wesentlichen korrekt umgesetzt werden. Wichtig ist auch, dass die Priorität der Regeln eingehalten wird.
- $[-1]$ **Unzureichende Diskussion**
  - Ideen für eigene (eventuell verworfene) Regeln ggf. mit Prioritäten sollen beschrieben sein.
  - Die Ergebnisse der Trennung müssen diskutiert werden. Es genügt, wenn eine grundsätzliche Aussage zur Qualität der Ergebnisse der genutzten Trennregeln vorhanden ist.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Die Trennstellen müssen klar gekennzeichnet sein. Außerdem muss klar sein, welcher Regelsatz zur Ermittlung der Ergebnisse angewandt wurde.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens sechs der vorgegebenen Beispiele (`silben01.txt` bis `silben08.txt`) enthalten.

Aufgabe 1: Drehfreudig?

1.1 Lösungsidee

Ohne große Einkleidung wird in dieser Aufgabe der informatische *Baum* eingeführt und nach einer ganz besonderen Eigenschaft solcher Bäume gefragt, der Drehfreudigkeit. Wie die Aufgabe erläutert, bestehen Bäume aus Knoten sowie aus Kanten zwischen einigen Knoten. Wenn n die Anzahl der Knoten ist, gibt es genau $n-1$ Kanten: Jeder Knoten ist mit genau einem *Elternknoten* verbunden, mit Ausnahme des *Wurzelknotens*. Die Knoten mit dem gleichen Elternknoten werden auch als dessen *Kinderknoten* bezeichnet. Bei der Bestimmung der Drehfreudigkeit wird jedem Knoten ein Rechteck zugeordnet, dessen Breite eine besondere Rolle spielt. Die Breite eines Knotenrechtecks nennen wir kurz auch die Breite des Knotens.

Für jeden Knoten werden der Elternknoten und die Anzahl der Kinderknoten bestimmt, um danach für jeden Knoten seine Breite zu berechnen. Der Wurzelknoten hat die Breite 1. Die Breite aller anderen Knoten wird berechnet, indem die Breite des Elternknotens durch die Anzahl der Kinder des Elternknotens geteilt wird. Die Berechnung muss in einer bestimmten Reihenfolge geschehen, da die Breite der tiefer im Baum liegenden Knoten von ihren Elternknoten abhängt. Somit muss die Breite eines Elternknotens immer vor der seiner Kinderknoten berechnet werden.

Ein Baum ist drehfreudig, wenn die Blätter von vorne und hinten die gleichen Breiten haben (genauer: wenn die Folge der Breiten ein Palindrom ist), da dann nach Drehen des Baumes immer gleich breite Blätter aufeinander liegen. Haben die Blätter von vorne und hinten unterschiedliche Breiten, ist der Baum nicht drehfreudig.

Eine strengere Definition drehfreudiger Bäume setzt nicht nur gleiche Breiten, sondern auch gleiche Höhe der Blätter voraus. Diese Interpretation ist ebenfalls zulässig, wird aber in dieser Lösung nicht verwendet.

Um ein Bild eines drehfreudigen Baumes auszugeben, müssen wir die Knoten auf jeder Tiefe mit ihrer Breite proportional zur Gesamtbreite des Baumes als Rechtecke ausgeben. Rechtecke von Blättern, die sich nicht auf der tiefsten Ebene befinden, sollen sich bis dorthin erstrecken.

1.2 Umsetzung

Drehfreudigkeit bestimmen

Um die Breiten möglichst schnell in der richtigen Reihenfolge zu berechnen, benutzen wir einen Stack (Stapelspeicher). Bei einem Stack handelt es sich um eine Datenstruktur, bei der nur das oberste Element entnommen werden kann und Elemente nur ganz oben hinzugefügt werden können, ähnlich einem Bücherstapel. Ein Stack kann auch durch ein Array simuliert werden, bei dem ein Index gespeichert wird, der auf das oberste Element verweist. Da jeder Knoten von zwei Klammern, einer öffnenden und einer schließenden, dargestellt wird, nutzen wir nur den Index der öffnenden Klammer, um einen Knoten zu identifizieren. Wir nutzen zwei Arrays `Elternknoten` und `AnzahlKinder`, um den Elternknoten und die Anzahl der Kinder aller Knoten zu speichern. Nun gehen wir die Klammerfolge von links nach rechts durch.

Ist die aktuelle Klammer am Index u in der Klammerfolge öffnend, so entspricht der oberste Wert auf dem Stack dem Elternknoten v und wir können diesen in `Elternknoten`$[u]$ speichern.

Außerdem können wir `AnzahlKinder`$[v]$ um 1 erhöhen, da wir ein neues Kind von v gefunden haben. Ist der Stack leer, handelt es sich um die Wurzel des Baumes, die keinen Elternknoten hat.

Ist die aktuelle Klammer hingegen schließend, so verweist der oberste Wert auf dem Stack auf die korrespondierende öffnende Klammer u. Den Elternknoten müssen wir nicht erneut ermitteln, da dies bereits bei der öffnenden Klammer geschehen ist. Jedoch wissen wir nun, dass wir alle Nachfahren von u betrachtet haben. Falls `AnzahlKinder`$[u] = 0$ gilt, haben wir u als Blatt identifiziert, welches wir zum Array `Blätter` hinzufügen können.

Handelt es sich um eine öffnende Klammer, fügen wir ihren Index nun zum Stack hinzu, da wir bis zur korrespondierenden schließenden Klammer die Nachfahren des Knotens betrachten. Bei einer schließenden Klammer entfernen wir das oberste Element vom Stack, den Index der korrespondierenden öffnenden Klammer, weil wir alle Nachfahren des Knotens betrachtet haben.

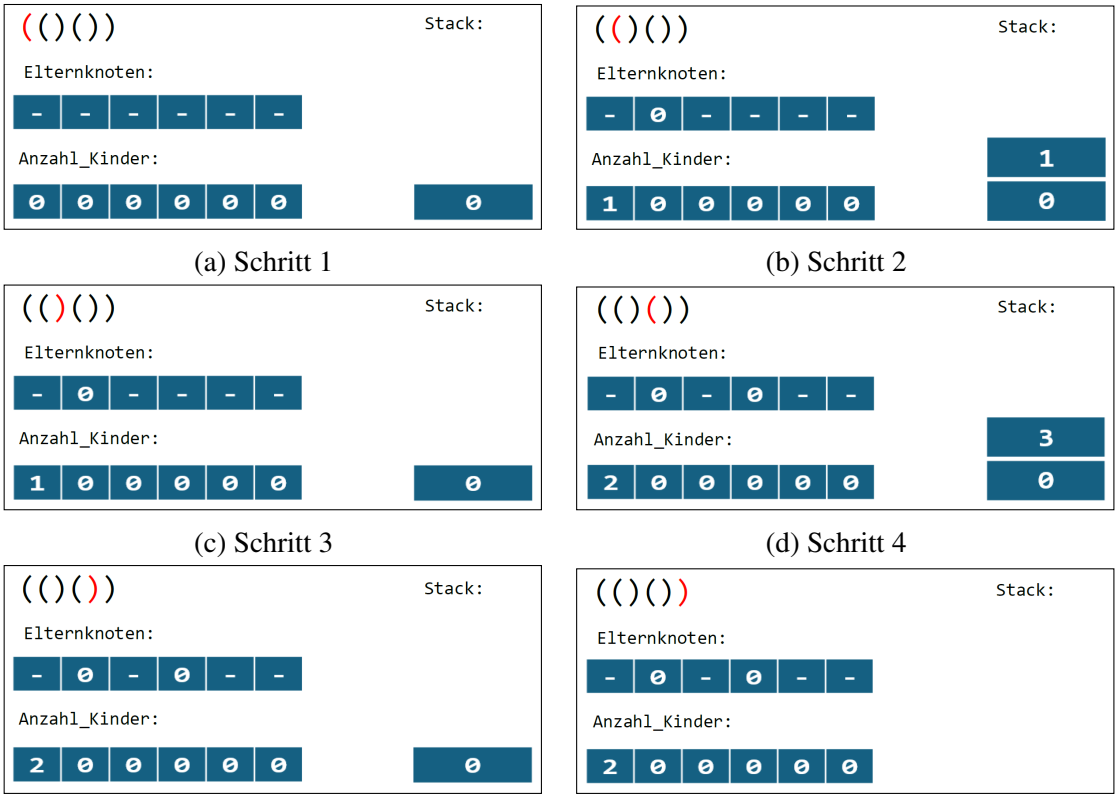


Abbildung 1.1: Die Abbildungen (a) bis (f) stellen das Vorgehen bei der schrittweisen Verarbeitung der Eingabe $(()())$ dar. Es ist jeweils zu sehen, welche Eingabe gelesen wird, wie sich das Array der `Elternknoten` und `Kindknoten` verändert und welche Werte auf dem Stack liegen.

Nun können wir die Breiten aller Knoten berechnen und im Array `Breite` speichern. Der Wurzelknoten hat die Breite 1. Die Breite `Breite`$[u]$ eines beliebigen anderen Knotens u mit Elternknoten `Elternknoten`$[u]$ entspricht:

$$\frac{\texttt{Breite}[\texttt{Elternknoten}[u]]}{\texttt{AnzahlKinder}[\texttt{Elternknoten}[u]]}$$

Um `Breite`$[$`Elternknoten`$[u]]$ immer vor `Breite`$[u]$ zu berechnen, reicht es die Klammerfolge von links nach rechts durchzugehen, da die öffnende Klammer eines Elternknotens immer vor der eines Kinderknotens erscheint.

Um Gleitkomma-Ungenauigkeiten zu vermeiden, ist es sinnvoll, die Breite vom Knoten am Index u, die als Bruch mit dem Zähler 1 dargestellt werden kann, nur anhand des Nenners zu speichern. In diesem Fall speichert man:

$$\texttt{Breite}[u] = \texttt{Breite}[\texttt{Elternknoten}[u] \times \texttt{AnzahlKinder}[\texttt{Elternknoten}[u]]]$$

Abbildung 1.2: Berechnete Knotenbreiten für das Beispiel $(()())$. Im Falle eines Bruches, werden nur die Nenner abgespeichert.

Wenn wir alle Breiten berechnet haben, können wir die Drehfreudigkeit überprüfen, indem wir `Breite`$[$`Blätter`$[i]]$ mit `Breite`$[$`Blätter`$[n-1-i]]$ vergleichen, `Breite`$[$`Blätter`$[i+1]]$ mit `Breite`$[$`Blätter`$[n-2-i]]$ usw., wobei n die Anzahl an Blättern ist.

Drehfreudigen Baum visualisieren

Handelt es sich um einen drehfreudigen Baum, gibt es verschiedene Möglichkeiten ihn visuell darzustellen. Z.B. können wir die Rechtecke mit Bibliotheken wie *turtle* oder *pygame* zeichnen lassen. Die Breite eines Rechteckes für einen Knoten u entspricht hier `Breite`$[u]$. Die Höhe des Rechteckes beträgt `MaximaleTiefe` $-$ `Tiefe`$[u]+1$ für Blätter und 1 für alle anderen Knoten. Die Tiefen aller Knoten können mit einer Tiefensuche ermittelt werden. Wenn wir die Klammerfolge von links nach rechts durchgehen und die Rechtecke zeichnen, liegt die linke obere Ecke des Rechteckes für u an der Ecke der unteren Kante von `Elternknoten`$[u]$ zu der rechten Kante des am weitesten rechts gezeichneten Rechtecks auf dieser Ebene, falls vorhanden, und der y-Achse anderenfalls. Mit dem gleichen Verfahren können wir auch den umgedrehten Baum ausgeben, indem wir von unten rechts statt von oben links ausgehen.

Zeichnen wir den Baum hingegen in der Konsole, müssen wir die Breiten zunächst normieren, damit wir sie als ein Vielfaches von Leerzeichen und Strichen darstellen können. Dafür berechnen wir das kleinste gemeinsame Vielfache (kgV) der Nenner der Breiten aller Blätter. Um das kgV zweier Zahlen a und b zu berechnen benutzen wir die Formel $kgV(a,b) = \frac{a \cdot b}{ggT(a,b)}$. ggT bezeichnet den größten gemeinsamen Teiler zweier Zahlen. Dieser kann mit dem euklidischen Algorithmus berechnet werden: Der $ggT(a,b)$ für $a \geq b$ entspricht $ggT(b, a\%b)$. Wenn $a\%b = 0$, dann ist b der ggT und kann über alle rekursiven Aufrufe zurückgegeben werden.

Wir berechnen nun das kgV aller Blätter, wobei wir für a immer das letzte Ergebnis nehmen und für b ein neues Blatt. Dann passen wir die Breiten an, indem wir sie mit dem kgV multiplizieren, falls wir sie als Bruch gespeichert haben, oder das kgV durch sie teilen, falls wir nur die Nenner gespeichert haben.

Jetzt entspricht jede Breite einem ganzzahligen Wert: Der Anzahl an Leerzeichen, die das Rechteck breit ist. Wir können einen zweidimensionalen Vektor verwenden, um ihn nacheinander mit den Rechtecken zu befüllen, bevor wir ihn am Ende ausgeben. Dabei müssen wir beachten, dass Trennstriche auch ein Zeichen Platz einnehmen und die Breiten, von Elternrechtecken entsprechend anpassen. Um den Baum umgedreht auszugeben, geben wir den zweidimensionalen Vektor einfach rückwärts aus.

1.3 Laufzeit

Die Berechnung der Breiten hat eine Zeitkomplexität von $\mathcal{O}(n)$, wobei n der Länge der Klammerfolge entspricht. Wir gehen die Klammerfolge zweimal linear durch: Einmal zur Ermittlung der Elternknoten, Anzahl an Kinderknoten und Blätter und einmal zur Berechnung der Breiten. Stack-Operationen geschehen in konstanter Zeit und sind somit zu vernachlässigen. Außerdem gehen wir die Blätter, deren Anzahl geringer als n ist, einmal linear durch.

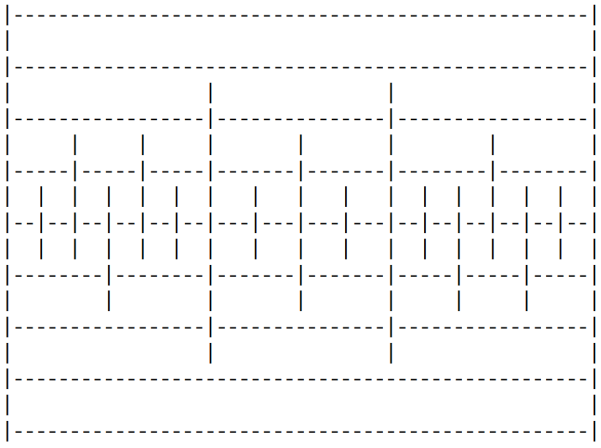
Die Laufzeitkomplexität der Ausgabe variiert je nach Verfahren. Die Laufzeit zur Berechnung des kgVs zweier Zahlen ist z.B. ungefähr logarithmisch, was zu einer leicht höheren Laufzeit insgesamt führt. Jedoch ist diese Erhöhung in unseren Beispielen irrelevant, da die zu betrachtenden Bäume nicht sonderlich groß sind.

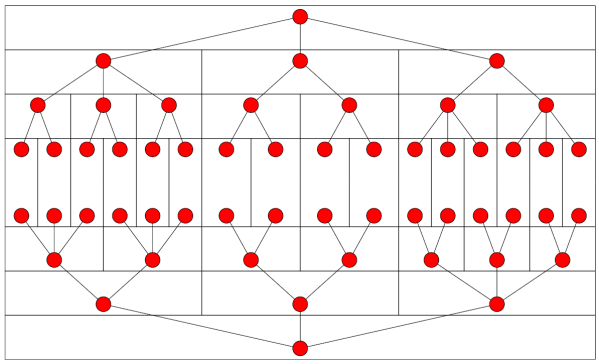
1.4 Beispiele

Wird die Interpretation von drehfreudigen Bäumen verwendet, nach der nicht nur die Breiten, sondern auch die Höhen der Blattrechtecke übereinstimmen müssen, sind, zusätzlich zu den nicht-drehfreudig dokumentierten Beispielen, auch die Beispiele `drehfreudig06.txt`, `drehfreudig08.txt`, `drehfreudig13.txt` und `drehfreudig15.txt` nicht drehfreudig.

drehfreudig01.txt

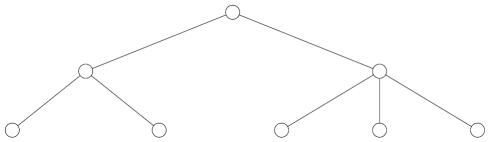
`drehfreudig`





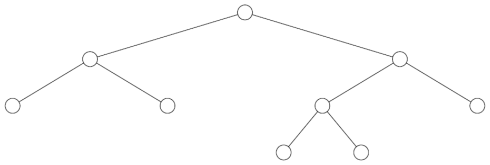
drehfreudig02.txt

`nicht drehfreudig`



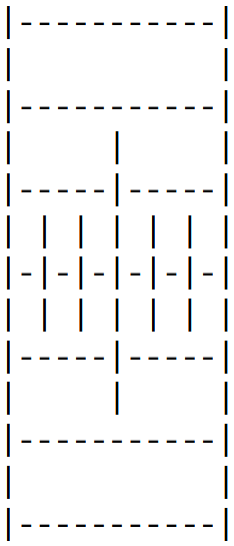
drehfreudig03.txt

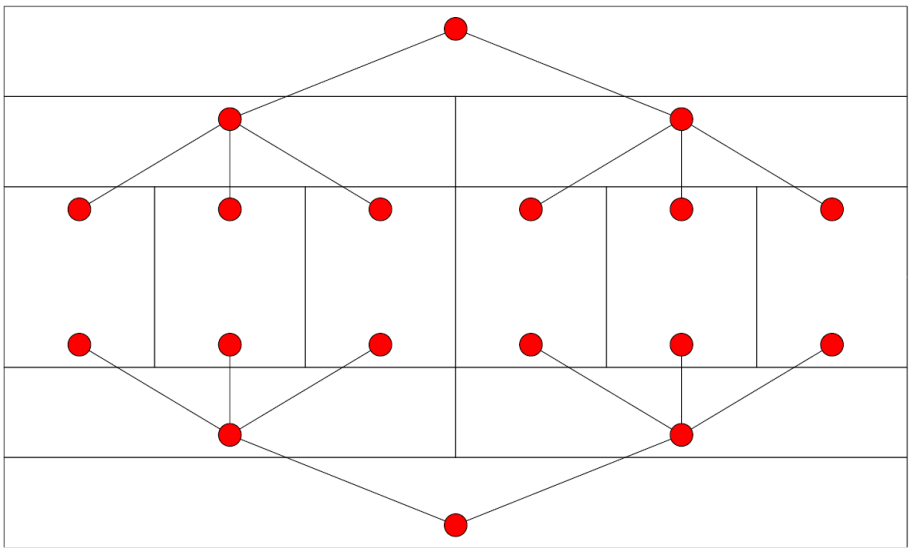
`nicht drehfreudig`



drehfreudig04.txt

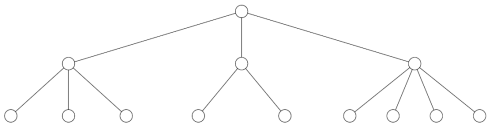
```
drehfreudig
```





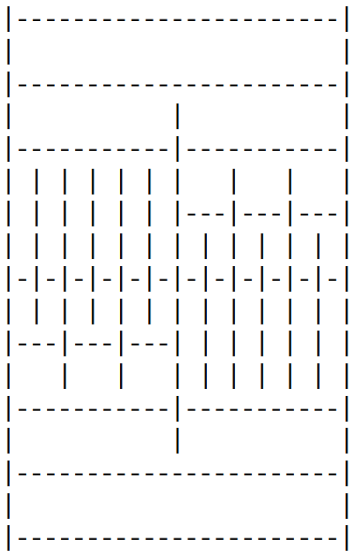
drehfreudig05.txt

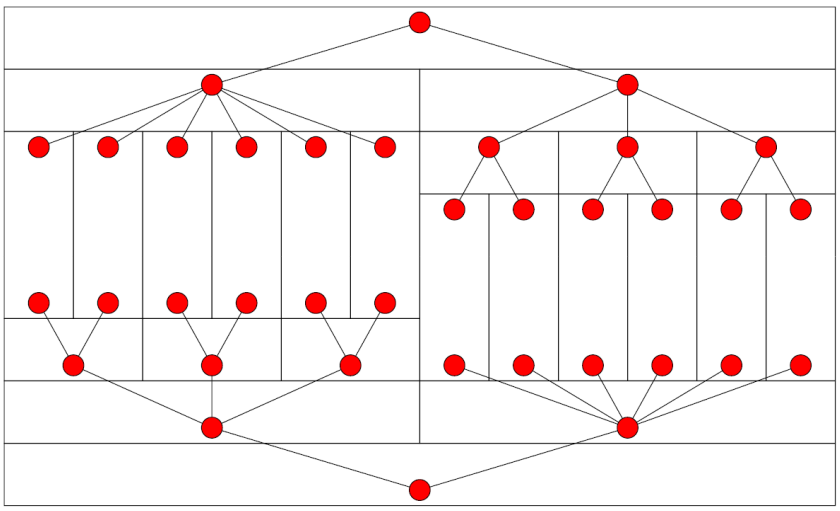
```
nicht drehfreudig
```



drehfreudig06.txt

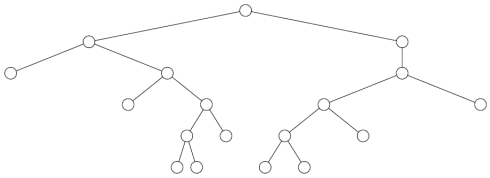
```
drehfreudig
```





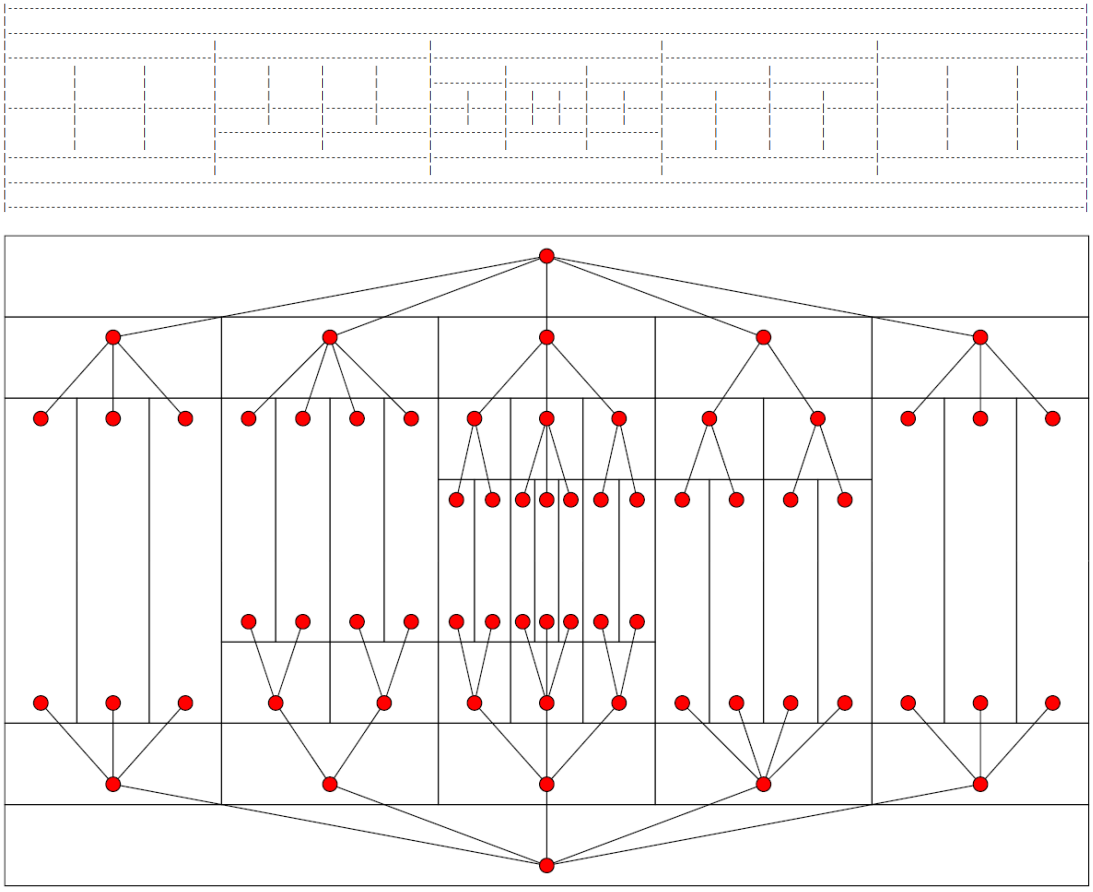
drehfreudig07.txt

`nicht drehfreudig`



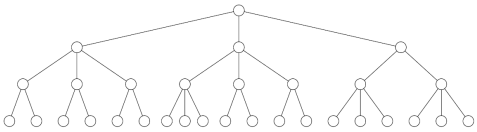
drehfreudig08.txt

`drehfreudig`



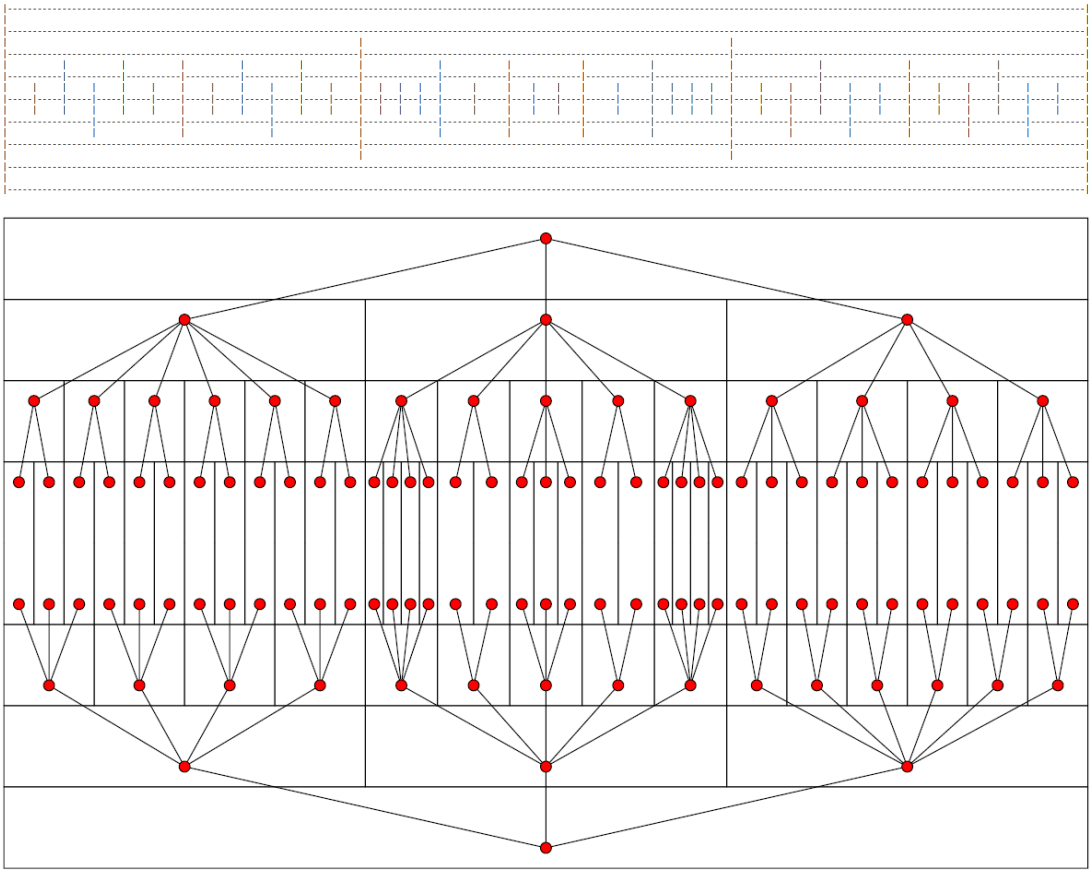
drehfreudig09.txt

`nicht drehfreudig`



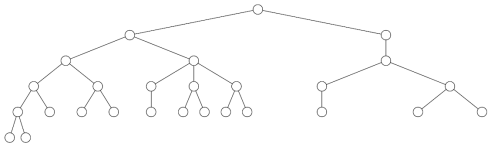
drehfreudig10.txt

`drehfreudig`



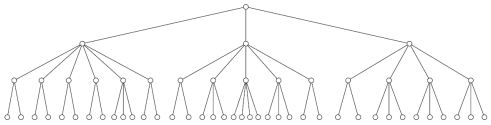
drehfreudig11.txt

`nicht drehfreudig`



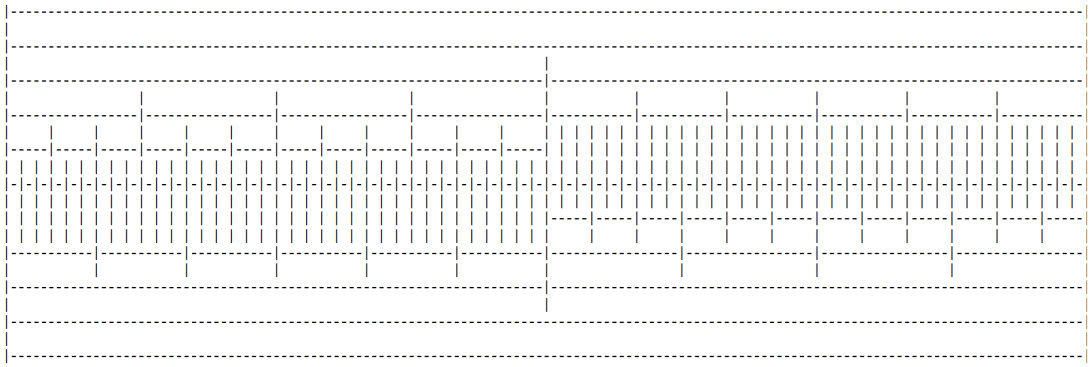
drehfreudig12.txt

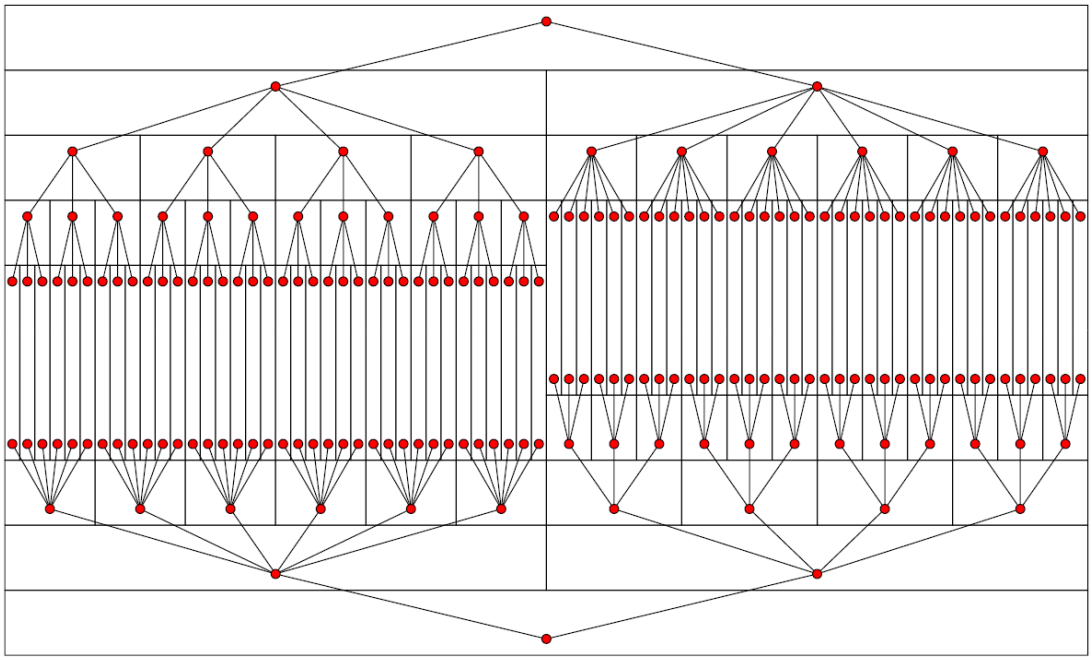
`nicht drehfreudig`



drehfreudig13.txt

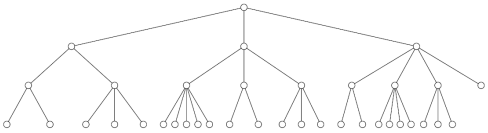
`drehfreudig`





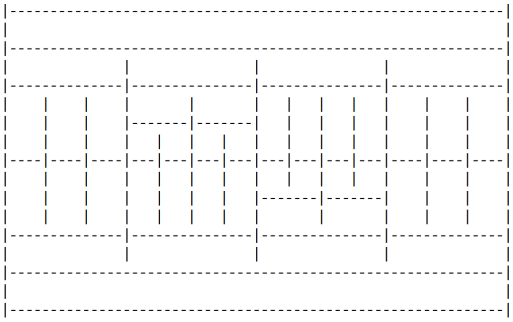
drehfreudig14.txt

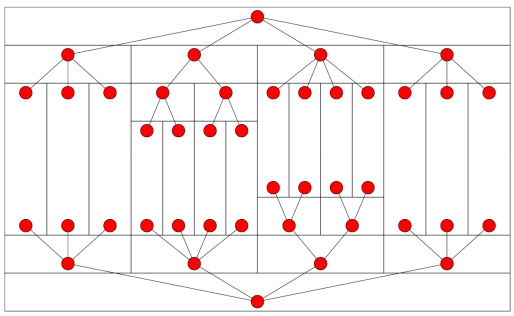
`nicht drehfreudig`



drehfreudig15.txt

`drehfreudig`





1.5 Bewertungskriterien

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

- [−1] **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**
  Die allgemeinen Anforderungen sind im Teil „Allgemeines“ im Abschnitt „Dokumentation“ erläutert.
- [−1] **Modellierung ungeeignet**
  Die gewählte Datenstruktur sollte die unkomplizierte Berechnung der Rechteckbreiten zulassen.
- [−1] **Lösungsverfahren fehlerhaft**
  Die Breiten aller Knoten (genauer gesagt die Breiten der zugehörigen Rechtecke) müssen korrekt berechnet werden. D.h. die Breite eines Elternknotens entspricht immer der Summe der Breiten seiner Kindknoten. Es muss als Kriterium für die Drehfreudigkeit erkannt werden, dass die Folge der Blätterbreiten ein Palindrom ist. Andere Kriterien, wie zum Beispiel Symmetrieüberlegungen, führen nicht immer zu richtigen Ergebnissen.
- [−1] **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Das Auslesen der Baumstruktur aus der Klammerfolge sollte mit linearer Zeitkomplexität erreicht werden.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Es muss ausgegeben werden, ob der Baum drehfreudig ist.

  Für alle drehfreudigen Bäume muss eine visuelle Darstellung durch Rechtecke ausgeben werden, wobei die Baumstruktur deutlich erkennbar sein muss. Solange die letztere Bedingung erfüllt ist, ist auch eine textbasierte Darstellung in Ordnung. Darstellungen für drehfreudige Bäume müssen den Baum zweifach, einmal normal und einmal um 180° gedreht, abbilden, sodass die passgenauen Rechtecke übereinander liegen. Die Breiten aller Rechtecke müssen proportional zu den zuvor berechneten Breiten sein.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu mindestens acht der fünfzehn vorgegebenen Beispiele (`drehfreudig01.txt` bis `drehfreudig15.txt`) enthalten. Darin müssen Beispiele mit und ohne drehfreudige Bäume enthalten sein.

Aufgabe 2: ChoreoGraph

Let's Dance! Und wer nicht tanzen will, muss eben Line-Dance-artige Tanzfiguren zu Choreographien zusammenfügen, die den Ansprüchen des Bundeswettbewerbs Informatik genügen. Jede Figur entspricht einer Permutation von 16 Elementen und benötigt eine bestimmte Anzahl von Takten. Konkret müssen nun bei dieser Aufgabe Figuren aneinander gereiht werden, sodass jede Person am Ende der Choreographie wieder an ihrer Startposition steht und die Länge der Choreographie der Liedlänge entspricht. Eine Choreographie, welche diese Bedingungen erfüllt, nennen wir *gültig*. Weiterhin sollen die besten Choreographien nach den folgenden Kriterien gefunden werden:

- Die Choreographie hat möglichst viele unterschiedliche Figuren.
- Die Choreographie hat möglichst viele Figuren.
- Die Choreographie hat möglichst wenige Figuren.
- Die von allen Tanzenden insgesamt zurückgelegte Strecke ist möglichst groß.
- Die von allen Tanzenden insgesamt zurückgelegte Strecke ist möglichst gering.

2.1 Lösungsidee

Ein erster Ansatz für dieses Problem ist ein Brute-Force-Backtracking-Algorithmus, der alle möglichen Kombinationen von Figuren ausprobiert und die jeweils besten Choreographien nach den oben genannten Kriterien speichert. Hierbei werden rekursiv alle Figuren an einander gereiht, bis die Gesamtlänge der Choreographie der Liedlänge entspricht oder sie überschreitet. Daraufhin wird überprüft, ob die Choreographie die beiden Bedingungen zur Länge und Permutation erfüllt und gegebenenfalls werden die besten Choreographien aktualisiert.

Die Anzahl der unterschiedlichen Figuren einer Choreographie kann mithilfe eines bool-Arrays ausgerechnet werden. Die zurückgelegte Strecke bei einer einzelnen Figur berechnet sich aus der absoluten Differenz zwischen der Position vor und nach der Figur. In dem folgenden Beispiel wird eine Gesamtstrecke von 6 zurückgelegt.

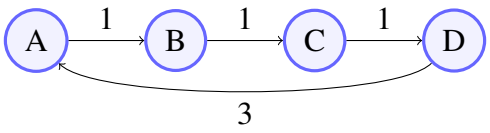


Abbildung 2.1: Distanzberechnung

Der Algorithmus kann nun durch verschiedene Beobachtungen optimiert werden. Die folgenden Abschnitte beschreiben zwei Ansätze für mögliche Optimierungen.

Auswahl der Figurenanzahl

Die Reihenfolge der verwendeten Figuren hat keinen Einfluss auf die Kriterien der Choreographie und kann lediglich ihre Gültigkeit beeinflussen. Daher ist es sinnvoll, zunächst zu entscheiden, wie oft jede Figur verwendet wird (im Folgenden *Auswahl* genannt), und danach die Reihenfolge zu bestimmen. Wenn für eine bestimmte Auswahl an Figuren eine gültige Reihenfolge gefunden wurde, so ist es nicht notwendig, weitere Reihenfolgen für diese Auswahl

zu testen. Weiterhin kann bevor eine gültige Reihenfolge für eine Auswahl gesucht wird, überprüft werden, ob diese Auswahl überhaupt in einer der Kriterien eine Verbesserung erzeugen könnte.

Rotation der Choreographie

Eine weitere Beobachtung ist, dass die Reihenfolge der Figuren in einer Choreographie zyklisch rotiert werden kann ohne die Gültigkeit der Choreographie zu beeinflussen. Das bedeutet, wenn eine gültige Choreographie aus den Figuren *abc* besteht, so sind auch *bca* und *cab* gültige Choreographien. Hierfür kann eine gültige Choreographie als Kreis dargestellt werden, bei dem jede Figur einen Abschnitt des Kreises einnimmt.

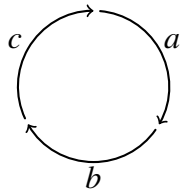


Abbildung 2.2: Kreisdarstellung einer Choreographie

Dies bedeutet, dass bei dem Generieren der Choreographien im besten Fall nur eine der möglichen Rotationen erstellt und getestet werden muss. Da Figuren mehrfach vorkommen können und die Anzahl der Figuren vorher bestimmt wird, handelt es sich hier aus kombinatorischer Sicht um Necklaces[5] (Perlenketten). Das effiziente Generieren dieser Perlenketten ist nicht trivial, aber es existieren verschiedene Algorithmen dafür, wie z.B. in dem Paper Frank Ruskey: Generating Necklaces[6] beschrieben.

Für die Umsetzung hier wurde ein einfacherer Ansatz gewählt, bei dem für eine Auswahl eine Figur als Startfigur festgelegt wird und nur Reihenfolgen generiert werden, die mit dieser Figur beginnen. Hier ist es sinnvoll eine Figur zu wählen die möglichst selten vorkommt, um die Anzahl der möglichen Reihenfolgen zu minimieren. Auch wenn bei diesem Ansatz immer noch einige unter Rotation äquivalente Permutationen mehrfach erzeugt werden, wird die Anzahl der zu testenden Reihenfolgen deutlich reduziert.

Algorithmus insgesamt

Der Algorithmus besteht insgesamt aus zwei Phasen. In Phase 1 wird eine Auswahl an Figuren getroffen, und in Phase 2 wird eine gültige Reihenfolge für diese Auswahl gesucht. Mithilfe von Backtracking wird in Phase 1 rekursiv eine Auswahl an Figuren getroffen, bis die Gesamtlänge der Figuren der Liedlänge entspricht. Danach wird getestet, ob diese Auswahl die beste bislang gefundene Choreographie für eines der Kriterien verbessert. Wenn dies der Fall ist, wird eine Reihenfolge für diese Auswahl gesucht, bei der alle Personen am Ende der Choreographie wieder in ihrer Startposition stehen. Hierfür wird zunächst eine Figur gesucht, die möglichst selten vorkommt und als Startfigur festgelegt. Daraufhin wird mithilfe Backtracking eine Reihenfolge der restlichen Figuren gesucht, die eine gültige Choreographie ist. Der Pseudocode in Algorithmus 2.1 beschreibt dieses Verfahren grob.

[5] `https://en.wikipedia.org/wiki/Necklace_(combinatorics)`
[6] `https://www.sciencedirect.com/science/article/pii/0196677492900476`

Algorithmus 2.1 2-Phase-Algorithm

```
AuswahlBacktrack(0, 0, [0, 0, ..., 0])          ▷ Initiale Auswahl mit Länge 0 und Index 0
procedure AUSWAHLBACKTRACK(länge, index, auswahl)
    if länge == liedlänge then
        if istBesserAlsBesteChoreographie(auswahl) then
            startFigur = findeStartFigur(auswahl)
            auswahl[startFigur] -= 1
            ReihenfolgeBacktrack([startFigur], auswahl)
        end if
    end if
    if länge >= liedlänge or index == figuren.length then
        return
    end if
    anzahl = 0
    while länge <= liedlänge do
        auswahl[index] = anzahl
        AuswahlBacktrack(länge, index + 1, auswahl)
        anzahl += 1
        länge += figuren[index].länge
    end while
    return
end procedure

procedure REIHENFOLGEBACKTRACK(liste, auswahl)
    if sum(auswahl) == 0 then                    ▷ Alle Figuren verwendet
        if istGültigeChoreographie(reihenfolge) then
            aktualisiereBesteChoreographien(reihenfolge)
        end if
        return true
    end if
    for i = 0 to auswahl.length - 1 do
        if auswahl[i] > 0 then
            auswahl[i] -= 1
            reihenfolge.push(figuren[i])
            if ReihenfolgeBacktrack(reihenfolge, auswahl) then
                return true
            end if
            reihenfolge.pop()
            auswahl[i] += 1
        end if
    end for
    return false
end procedure
```

Die Laufdistanz jeder Figur wird aus der absoluten Differenz der Position vor und nach der Figur berechnet. Hierfür muss das Array, welches die Positionen der Personen nach der Figur speichert, nur einmal durchlaufen werden.

Da beim Aufruf der *ReihenfolgeBacktrack* Funktion die Liedlänge bereits getestet wurde, ist es nurnoch notwendig zu überprüfen ob die Permutation der Choreographie die Identität ist. Hierfür wird die derzeitige Position alle Tänzer nach jeder Figur aktualisiert und am Ende überprüft ob alle Tänzer wieder an ihrer Startposition sind.

2.2 Laufzeit

Die Laufzeit des Algorithmus hängt sowohl von der Liedlänge als auch von der Anzahl und Länge der einzelnen Figuren ab. Angenommen, es gibt n Figuren der Länge 1 und die Liedlänge beträgt t Takte, dann ist der Aufwand des Brute-Force-Backtracking-Algorithmus $\mathcal{O}(n^t)$. Der 2-Phasen-Algorithmus hat die selbe Worst-Case-Laufzeit, da alle möglichen Kombinationen an Figuren ausprobiert werden müssen. Allerdings löst er die Beispiele in der Praxis deutlich schneller als der Brute-Force-Backtracking-Algorithmus.

2.3 Beispiele

Zusamenfassung

In der folgenden Tabelle sind die Ergebnisse mit ihren Werten für die Optimierungskriterien zusammengefasst. Es werden jeweils die (minimale, maximale) Anzahl an Figuren, Anzahl einzigartiger Figuren und Distanzen angegeben. Zusätzlich stehen in den letzten beiden Spalten die Laufzeiten des Brute-Force-Backtracking-Algorithmus und des 2-Phasen-Algorithmus im Vergleich.

Beispiel	Figuren	Einzig.	Distanz	Brute Force	2 Phase
choreo01.txt	(12, 16)	(1, 2)	(312, 384)	0.6 ms	0.2 ms
choreo02.txt	-	-	-	7.3 s	135 ms
choreo03.txt	-	-	-	1.6 ms	0.06 ms
choreo04.txt	12	6	1154	20.1 s	453 ms
choreo05.txt	(6, 24)	(1, 2)	(504, 1728)	4.4 s	145 ms
choreo06.txt	(4, 16)	(1, 4)	(256, 768)	1.3 s	14 ms

In den folgenden Abschnitten werden die Ergebnisse der einzelnen Beispiele genauer beschrieben. In der letzten Spalte sind die gefundenen Choreographien aufgelistet. Sollte eine Figur mehrfach hintereinander vorkommen, so wird dies mit Klammern und der Anzahl dahinter dargestellt.

choreo01.txt

Länge des Liedes: 32 Takte

Anzahl Figuren	Einzigartige Figuren	Distanz	Choreo
16	1	384	CastFour(16)
12	2	312	CastFour(4) CastFour CastUp CastFour CastUp CastFour CastUp CastFour CastUp

choreo02.txt

Länge des Liedes: 24 Takte

Keine gültige Choreographie möglich.

choreo03.txt

Länge des Liedes: 48 Takte

Keine gültige Choreographie möglich.

choreo04.txt

Länge des Liedes: 32 Takte

Anzahl Figuren	Einzigartige Figuren	Distanz	Choreo
12	6	1154	Aida Fan(2) Hover Fan HockeyStick Fan(4) Appel Bounce

choreo05.txt

Länge des Liedes: 24 Takte

Anzahl Figuren	Einzigartige Figuren	Distanz	Choreo
24	1	1728	HipTwist(24)
21	2	1524	HipTwist(20) SpotTurn
6	1	504	SpotTurn(6)

choreo06.txt

Länge des Liedes: 16 Takte

Anzahl Figuren	Einzigartige Figuren	Distanz	Choreo
8	1	768	Cycle4(8)
10	4	568	Cycle4(3) Pairs(2) Sides CycleBack(4)
4	1	512	Sides(4)
16	1	256	Pairs(16)

2.4 Bewertungskriterien

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

- [−1] **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**
  Die allgemeinen Anforderungen sind im Teil „Allgemeines“ im Abschnitt „Dokumentation“ erläutert.
- [−1] **Modellierung ungeeignet**
  Durch die gewählte Modellierung sollten Distanzen zwischen den Positionen und Permutationen der Positionen abbildbar sein.
- [−1] **Lösungsverfahren fehlerhaft**
  Das Verfahren muss korrekt entscheiden können, ob es eine gültige Choreographie gibt. Die gefundenen Choreographien müssen nach den angelegten Kriterien optimal sein.
- [−1] **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Ein Brute-Force-Ansatz, welcher alle möglichen Figurenkombinationen testet, ist ausreichend. Aber es sollten keine Choreographien mehrfach getestet werden.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Wenn keine Choreographie möglich ist, muss das Programm dies melden. Falls mehrere Choreographien möglich sind, muss erkennbar sein, welche Choreographie welches Kriterium am besten erfüllt. Die für die Kriterien relevanten Werte müssen angegeben oder aus der Darstellung der Choreographien erkennbar sein.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu allen vorgegebenen Beispielen (`choreo01.txt` bis `choreo06.txt`) enthalten.

Beispiel	**Anzahl Figuren**		**Maximale Anzahl**	**Zurückgelegte Distanz**	
	min	max	**einzigartiger Figuren**	min	max
`choreo01.txt`	12	16	2	312	384
`choreo02.txt`	-	-	-	-	-
`choreo03.txt`	-	-	-	-	-
`choreo04.txt`	12	12	6	1154	1154
`choreo05.txt`	6	24	2	504	1728
`choreo06.txt`	4	16	4	256	768

Aufgabe 3: Nasser Hund

3.1 Lösungsidee

In dieser Aufgabe sind die Positionen von Wegen und Seen gegeben. Hilda möchte auf diesen Wegen mit ihrem Hund Wuffi Gassi gehen. Dabei kann sie sich an jedem Punkt eines jeden Weges aufhalten. Gesucht ist die maximale Länge einer Hundeleine, die es Wuffi unmöglich macht, in irgendeinen See zu springen. Könnte Wuffi das Wasser eines Sees erreichen, so würde die Leine die Begrenzung dieses Sees überqueren. Schließlich kann angenommen werden, dass alle Wege im Trockenen verlaufen. Also ist in Wirklichkeit der minimale Abstand zwischen zwei Punkten, von denen einer auf einem Weg und einer auf der Uferkante eines Sees liegt, gefragt.

In der Eingabe werden alle Wege durch (eindimensionale) Liniensegmente und alle Seen durch (einfache) Polygone dargestellt. Im Sinne dieser Aufgabe sind Hilda und Wuffi beide punktförmig. Gelöst werden muss somit die folgende geometrische Aufgabe: Gegeben sind zwei Mengen von Liniensegmenten, P (die Kanten der die Seen repräsentierenden Polygone) und W (die Wegstrecken). Gesucht ist die minimale euklidische Distanz zwischen zwei Punkten, von denen einer auf einem Liniensegment aus P und der andere auf einem Liniensegment aus W liegt. Definitionsgemäß ist dies der minimale Abstand zwischen einem Element von P und einem Element von W. Dieser lässt sich mit Algorithmus 3.1 ermitteln.

Algorithmus 3.1 Maximale Leinenlänge

Eingabe: Mengen P, W von Liniensegmenten
Ausgabe: $\min_{p\in P, w\in W} \text{dist}(p,w)$
$d \leftarrow \infty$
for $i = 0$ **to** $|P|-1$ **do**
  for $j = 0$ **to** $|W|-1$ **do**
    $d \leftarrow \min(d, \text{dist}(P[i], W[j]))$
  end for
end for
return d

Dabei soll die Funktion dist den Abstand zweier Liniensegmente ermitteln. Der letzte und entscheidende Schritt zu einer vollständigen Lösung ist also die Entwicklung dieser Funktion.

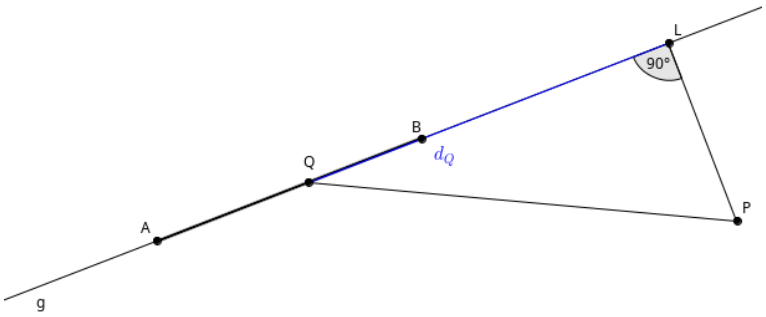
Abstand zweier Liniensegmente

Den Abstand zweier Liniensegmente zu finden ist ein Standardproblem. Für viele Programmiersprachen gibt es deswegen Bibliotheken, die Implementierungen der Funktion dist enthalten.[7] Dies trifft auch auf den Abstand eines Polygons von einem Liniensegment zu. Im Bundeswettbewerb Informatik ist es Teilnehmenden natürlich gestattet, solche Bibliotheken zu verwenden und ihre Lösung auf diesem Wege zu vereinfachen. Im Folgenden wird trotzdem beschrieben, wie die Funktion dist eigenständig umgesetzt werden kann.

Dafür wird zunächst der Abstand $\text{dist}(P, \overline{AB})$ zwischen einem Punkt P und einem Liniensegment $\overline{AB}$ betrachtet. Es sei g die Gerade, die das Liniensegment $\overline{AB}$ enthält. Aus der Schule ist

[7] Ein Beispiel dafür ist das Python-Modul *shapely*.

bekannt, wie sich der Abstand zwischen P und g berechnen lässt:[8] Zuerst wird der *Lotfußpunkt* von P auf g berechnet. Das ist der (eindeutige) Punkt $L \in g$, für den das Liniensegment $\overline{LP}$ senkrecht zur Geraden g steht. Der Abstand von P zu g ist dann die Länge der Strecke $\overline{LP}$.



Liegt L nun auf dem Liniensegment $\overline{AB}$, so ist $\text{dist}(P,\overline{AB})$ genau diese Länge. Andernfalls ist $\text{dist}(P,\overline{AB})$ der Abstand von P zu einem Punkt $Q \in \overline{AB}$ mit Abstand $d_Q > 0$ zu L. Nach dem Satz des Pythagoras ist $|\overline{PQ}|$ genau $\sqrt{d_Q^2 + |\overline{LP}|^2}$. Der optimale Punkt Q ist also derjenige, der d_Q minimiert. Das ist sicherlich einer der beiden Endpunkte A und B.

Insgesamt lässt sich der Abstand von P zu $\overline{AB}$ mit Algorithmus 3.2 berechnen.

Algorithmus 3.2 Abstand von Punkt zu Liniensegment

Eingabe: Punkt P, Liniensegment $\overline{AB}$
Ausgabe: $\text{dist}(P,\overline{AB})$
Berechne Gerade g durch A und B
Berechne Lotfußpunkt L von P auf g
if $L \in \overline{AB}$ **then**
    return $|\overline{LP}|$
else
    return $\min(|\overline{AP}|, |\overline{BP}|)$
end if

Nun seien zwei Liniensegmente $\overline{AB}$ und $\overline{CD}$ gegeben. Weiterhin seien $R \in \overline{AB}$ und $S \in \overline{CD}$ zwei Punkte, die den minimalen Abstand dieser beiden Liniensegmente haben. Aus den obigen Überlegungen folgt: R ist einer der beiden Entpunkte von $\overline{AB}$ oder der Lotfußpunkt von S auf der Geraden durch A und B. Eine analoge Aussage gilt für S. Ist keiner der beiden Punkte R und S ein Endpunkt seines jeweiligen Liniensegments, so steht die Strecke $\overline{RS}$ senkrecht zu $\overline{AB}$ und zu $\overline{CD}$. Diese beiden Liniensegmente sind dann parallel zueinander. Also können die beiden Punkte R und S auf diesen Liniensegmenten parallel verschoben werden, bis einer von ihnen ein Endpunkt seines jeweiligen Liniensegments ist.

[8] Für eine Beschreibung des dreidimensionalen Falles siehe beispielsweise `https://www.mathematik-oberstufe.de/vektoren/a/abstand-punkt-gerade-lot.html`

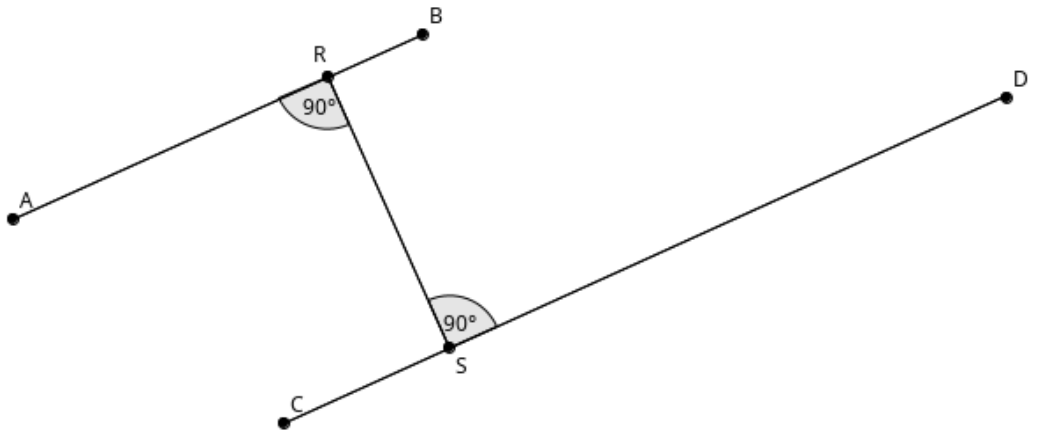


Abbildung 3.1: Sind R und S beide Lotfußpunkte des jeweils anderen Punktes, so sind $\overline{AB}$ und $\overline{CD}$ parallel zueinander.

Allgemein gilt deswegen: Der Wert von $\text{dist}(\overline{AB},\overline{CD})$ ist der Abstand eines der vier Punkte A, B, C und D zum jeweils anderen Liniensegment. Es ergibt sich Algorithmus 3.3 zur Berechnung der Funktion dist für zwei Liniensegmente.

Algorithmus 3.3 Abstand zweier Liniensegmente

Eingabe: Liniensegmente $\overline{AB}$, $\overline{CD}$
Ausgabe: $\text{dist}(\overline{AB},\overline{CD})$
$d \leftarrow \text{dist}(C,\overline{AB})$
$d \leftarrow \min(d,\text{dist}(D,\overline{AB}))$
$d \leftarrow \min(d,\text{dist}(A,\overline{CD}))$
$d \leftarrow \min(d,\text{dist}(B,\overline{CD}))$
return d

Laufzeit

Algorithmus 3.2 und Algorithmus 3.3 und damit die Berechnung der Funktion dist durchlaufen alle konstant viele Schritte, die jeweils in konstanter Zeit ausgeführt werden können. Die Laufzeit von Algorithmus 3.1 ist also proportional zur Anzahl der Aufrufe von $\text{dist}(P[i],W[j])$. Diese Anzahl ist genau $|P|\cdot|W|$. Entsprechend liegt die Laufzeitkomplexität dieses Algorithmus in $\mathcal{O}(|P|\cdot|W|)$.

Optimierung

Das Problem lässt sich auch in einer Laufzeitkomplexität von $\mathcal{O}(n\log n)$ mit $n := |P|+|W|$ lösen. Dafür sind allerdings fortgeschrittene Konzepte und ein hoher Implementierungsaufwand nötig, weshalb diese Lösung hier nur grob skizziert und ihre Laufzeitkomplexität auch nicht von den Teilnehmenden erwartet wird. Eine etwas ausführlichere Erklärung der verwendeten Konzepte und Algorithmen findet sich beispielsweise unter `https://www.cise.ufl.edu/~sitharam/COURSES/CG/kreveldmorevoronoi.pdf`.

Wie im Abschnitt über die Laufzeit aufgeführt, ist die Laufzeit von Algorithmus 3.1 proportional zur Anzahl der Paare von Strecken, deren Abstand berechnet wird. Darunter sind allerdings auch viele Paare, die schon deswegen nicht den minimalen Abstand haben können, weil zwischen ihnen viele weitere Strecken liegen. Die hier beschriebene Idee zielt darauf ab, die Anzahl der betrachteten Paare zu reduzieren. Dafür werden zunächst zwei Beobachtungen gemacht:

1. Haben zwei Punkte $R \in \overline{AB}$ und $S \in \overline{CD}$ den geringstmöglichen Abstand $|\overline{RS}|$ der beiden Liniensegmente $\overline{AB}$ und $\overline{CD}$, so hat der Mittelpunkt M der Strecke $\overline{RS}$ zu beiden Liniensegmenten einen Abstand von jeweils $\frac{|\overline{RS}|}{2}$. Der Punkt M liegt also auf dem *Bisektor* der Liniensegmente $\overline{AB}$ und $\overline{CD}$.
   Der Bisektor zweier Liniensegmente ist zusammengesetzt aus konstant vielen Halbgeraden und Parabelstücken und beschreibt die Menge der Punkte, die zu beiden Liniensegmenten genau den gleichen Abstand haben.
2. Liegt „zwischen“ $\overline{AB}$ und M ein weiterer Bisektor (eines beliebigen Paares von Strecken!), so ist $|\overline{RS}|$ nicht der gesuchte minimale Abstand.

Nun werden alle Strecken zu einer Menge $N = P \cup W$ mit $|N| = n$ zusammengefasst und das Voronoi-Diagramm dieser Mengen berechnet. Das Voronoi-Diagramm einer Menge von Liniensegmenten teilt die Ebene in verschiedene Regionen ein. Jede Region gehört dabei zu genau einem Liniensegment und umfasst die Menge aller Punkte, die zu diesem Liniensegment eine geringere Distanz haben als zu jedem anderen.

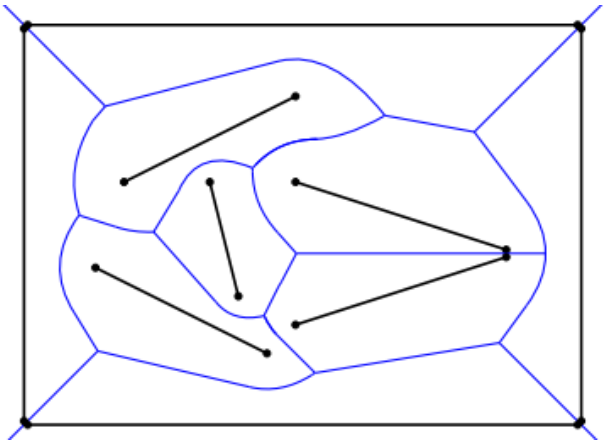


Abbildung 3.2: Beispiel für ein Voronoi-Diagramm von Liniensegmenten.
Quelle: `https://www.cise.ufl.edu/~sitharam/COURSES/CG/kreveldmorevoronoi.pdf`

Die Grenzen dieser Regionen setzen sich zusammen aus den Bisektoren von Paaren von Liniensegmenten. Im Unterschied zu Algorithmus 3.1 genügt es nun, den Abstand aller Paare von Liniensegmenten zu berechnen, die

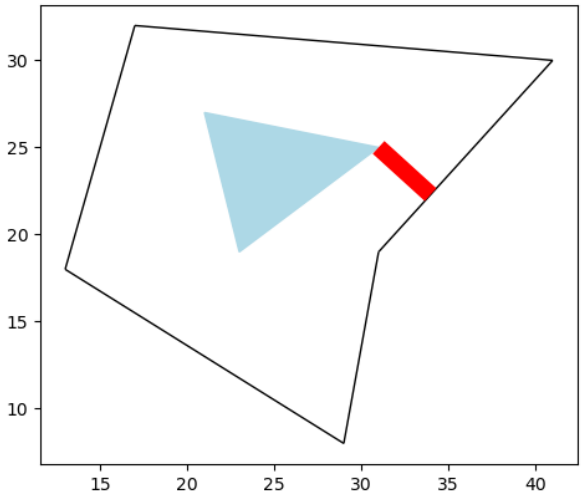
1. aus unterschiedlichen Mengen (P und W) kommen und
2. deren Bisektor Teil des Voronoi-Diagramms ist.

Mit den beiden vorherigen Beobachtungen lässt sich in einer umfangreicheren Beweisführung begründen, dass darunter tatsächlich ein Paar mit minimalem Abstand ist.

Das Voronoi-Diagramm der Liniensegmente kann in $\mathcal{O}(n \log n)$ berechnet werden. Außerdem enthält es maximal $\mathcal{O}(n)$ Bisektoren, womit maximal $\mathcal{O}(n)$ Paare von Liniensegmenten betrachtet werden müssen. Insgesamt liegt die Laufzeitkomplexität dieses Algorithmus also in $\mathcal{O}(n \log n)$.

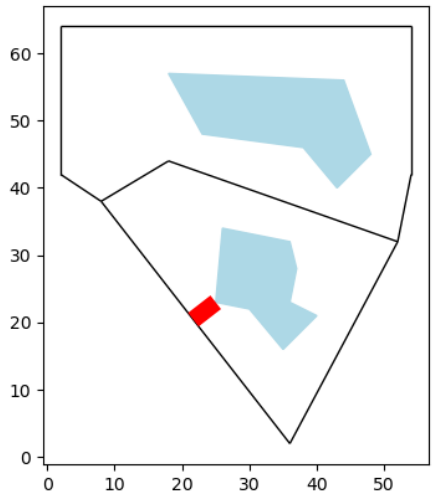
3.2 Beispiele

hund01.txt



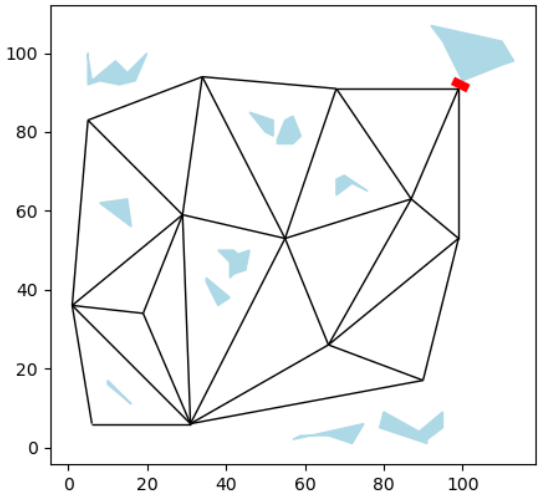
Die Hundeleine darf in diesem Beispiel eine Länge von maximal 4.036036763977875 haben. Das ist der Abstand der Punkte $(33.98642533936652, 22.285067873303166)$ und $(31, 25)$. Dieses Punktepaar ist eindeutig.

hund02.txt



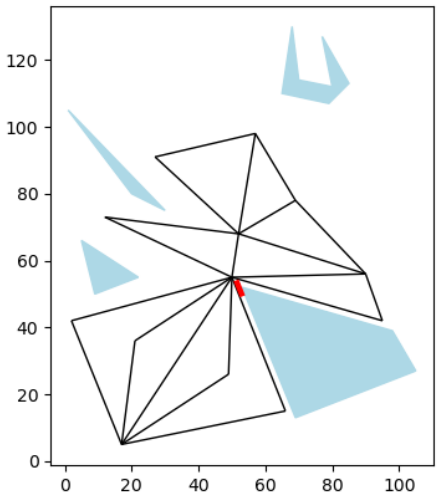
Die Hundeleine darf in diesem Beispiel eine Länge von maximal 4.20987849267374 haben. Das ist der Abstand der Punkte $(21.676923076923075, 20.415384615384617)$ und $(25, 23)$. Dieses Punktepaar ist eindeutig.

hund03.txt



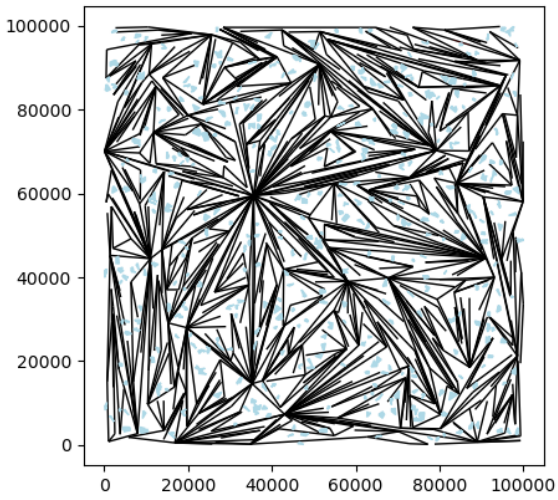
Die Hundeleine darf in diesem Beispiel eine Länge von maximal 2.23606797749979 haben. Das ist der Abstand der Punkte $(99, 91)$ und $(100, 93)$. Dieses Punktepaar ist eindeutig.

hund04.txt



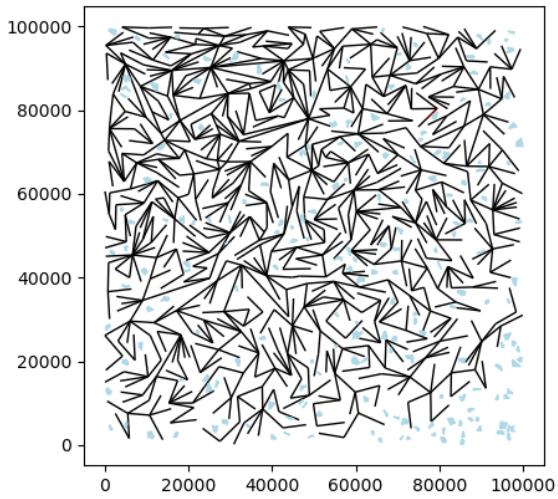
Die Hundeleine darf in diesem Beispiel eine Länge von maximal 1.6712580435934667 haben. Das ist der Abstand der Punkte $(51.44827586206897, 51.37931034482759)$ und $(53, 52)$. Dieses Punktepaar ist nicht eindeutig, weil es auf zwei zueinander parallelen Liniensegmente liegt.

hund05.txt



Die Hundeleine darf in diesem Beispiel eine Länge von maximal 50.3203503129886 haben. Das ist der Abstand der Punkte $(85194.81557780845, 96077.08987741536)$ und $(85180, 96029)$. Dieses Punktepaar ist eindeutig.

hund06.txt



Die Hundeleine darf in diesem Beispiel eine Länge von maximal 199.57250236268962 haben. Das ist der Abstand der Punkte $(78210.27511201614, 79189.49961184931)$ und $(78350, 79047)$. Dieses Punktepaar ist eindeutig.

3.3 Bewertungskriterien

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

- [−1] **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**
  Die allgemeinen Anforderungen sind im Teil „Allgemeines“ im Abschnitt „Dokumentation“ erläutert. Bei dieser Aufgabe muss insbesondere nachvollziehbar sein, welche Methoden zur Abstandsberechnung verwendet werden. Werden dafür externe Bibliotheken verwendet, muss dies erwähnt und deren Wahl begründet werden.
- [−1] **Modellierung ungeeignet**
  Es darf angenommen werden, dass sich zwei Liniensegmente aus der Eingabe maximal in ihren Endpunkten schneiden, sowie, dass es immer mindestens einen Weg und einen See gibt. Die zur Berechnung des Abstands gewählten Methoden dürfen nicht scheitern, wenn eine der Kanten parallel zu einer Koordinatenachse verläuft. Mit Rundungsungenauigkeiten muss sinnvoll umgegangen werden.
- [−1] **Lösungsverfahren fehlerhaft**
  Es muss das richtige geometrische Problem gelöst werden: Gesucht ist der minimale Abstand zweier Liniensegmente, von denen eines eine Uferkante und eines eine Wegstrecke repräsentiert. Die genutzten geometrischen Strategien müssen korrekt umgesetzt sein.
- [−1] **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Das Lösungsverfahren darf nicht lediglich ohne Strategie möglichst viele Paare von Punkten auf den Liniensegmenten ausprobieren.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Für jede Eingabe muss die maximale Leinenlänge ausgegeben werden, und zwar mit mindestens drei Nachkommastellen. Eine graphische Darstellung ist hilfreich, aber nicht erforderlich.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu allen vorgegebenen Beispielen (`hund01.txt` bis `hund06.txt`) enthalten. Die maximale Leinenlänge muss innerhalb der erlaubten Abweichung der Maximallänge der Leine liegen. Bei einem brauchbaren Umgang mit Rundungsfehlern sollten die Werte innerhalb der erlaubten Abweichung liegen.

Beispiel	**Maximallänge**	**Erlaubte Abweichung**	
		Untere Grenze	Obere Grenze
`hund01.txt`	4.036036763977	4.035	4.037
`hund02.txt`	4.209878492673	4.209	4.210
`hund03.txt`	2.236067977499	2.235	2.237
`hund04.txt`	1.671258043593	1.671	1.672
`hund05.txt`	50.320350312988	50.320	50.321
`hund06.txt`	199.572502362689	199.572	199.573

Aufgabe 4: Bibertomograph

4.1 Lösungsidee

Gegeben ist eine quadratische Fläche, die in einem Raster angeordnete Felder hat. Diese Felder können belegt oder unbelegt sein. Durch eine Belegung der Felder erhält man eine *Figur*. Als Eingabe erhält man nun für jede Reihe die Anzahl an belegten Feldern. Eine *Reihe* kann eine Zeile, Spalte, Diagonale von links unten nach rechts oben oder eine Diagonale von links oben nach rechts unten sein. Im Folgenden wird eine Figur *passend* genannt, wenn in allen Reihen die Anzahl belegter Felder mit der Eingabe übereinstimmen.

Das Programm soll nun die Eingabe auswerten und eine passende Figur rekonstruieren. Falls das eindeutig möglich ist, soll die passende Figur ausgegeben werden. Sonst sollen zusätzlich zu einer passenden Figur auch noch die Felder, deren Belegung nicht eindeutig bestimmbar ist, mit einem Fragezeichen gekennzeichnet werden. Die *Auswertung* der Eingabe kann also (in unserem Fall) für jedes Feld eine **1** (für „belegt“), eine **0** (für „unbelegt“) oder ein **?** (für „nicht eindeutig“) ergeben. Enthält die Auswertung kein **?**, gibt es nur eine zur Eingabe passende Figur.

4.1.1 Eindeutigkeit der Belegung

Zum Lösen des Problems ist zunächst eine präzisere Beschreibung hilfreich, wann die Belegung eines Feldes eindeutig ist.

- Eindeutig ist die Belegung eines Feldes, wenn es in allen passenden Figuren die gleiche Belegung (belegt bzw. unbelegt) hat. Dann kann man nämlich sicher sagen, ob das Feld belegt bzw. unbelegt sein muss.
- Uneindeutig ist die Belegung des Feldes, wenn es jeweils mindestens eine passende Figur gibt, in der das Feld belegt bzw. unbelegt ist. Dann kann man aus der Eingabe nicht schließen, welche Belegung das Feld haben muss.

4.1.2 Einzelne Betrachtung der Felder

Die gesuchte Auswertung (der Eingabe) kann man für jedes Feld unabhängig von anderen Feldern bestimmen. Basierend auf der vorherigen Beschreibung benötigen wir für jedes Feld nur zwei Informationen, um die Ausgabe zu berechnen: nämlich (1) ob es eine passende Figur gibt, bei der das Feld belegt ist, und (2) ob es eine passende Figur gibt, bei der das Feld unbelegt ist.

Anders ausgedrückt müssen wir wissen, ob das Feld belegt oder unbelegt sein kann. Wenn beides möglich ist, dann ist die Belegung des Feldes uneindeutig. Sonst ist die mögliche Belegung die eindeutige Belegung für das Feld. Dies vereinfacht unsere Berechnung: Wir brauchen einen Algorithmus, der uns für eine bestimmte Belegung des Feldes sagt, ob es eine passende Figur gibt. Diesen Algorithmus führen wir zwei mal aus: einmal mit dem Feld belegt und einmal mit dem Feld unbelegt. Dadurch bekommen wir, was wir brauchen.

Für jedes Feld wiederholt man das Ganze und erhält somit die gesamte Ausgabe. Zusätzlich benötigt man irgendeine passende Figur. Die im Folgenden beschriebenen Verfahren finden aber immer auch passende Figuren, von denen man ohne größeren Aufwand eine beliebige speichern kann. Daher wird hierauf nicht mehr eingegangen.

4.1.3 Lösung mit Backtracking

Ein naiver Brute-Force Ansatz

Wir wollen herausfinden, ob es eine passende Figur gibt, wobei die Belegung eines bestimmten Feldes vorgegeben ist. Ein sehr simpler Ansatz ist es, sämtliche Figuren auszuprobieren, bei denen das bestimmte Feld die vorgegebene Belegung hat. Für jede Figur testen wir, ob sie passend ist. Da das Verfahren nicht sonderlich geschickt einfach alles ausprobiert, spricht man hier auch von Brute-Force.

Dieses Verfahren hat jedoch einen großen Nachteil: es ist zu langsam. Gegeben sei ein $n \times n$ Feld. Dieses hat n^2 Felder, die jeweils zwei Zustände annehmen können. Daher gibt es 2^{n^2} verschiedene Figuren, zu viele, um alle auszuprobieren.

Backtracking

Als eine Verbesserung kann man *Backtracking* anwenden. Die Idee ist, alle möglichen Figuren schrittweise aufzubauen. Ein Feld ist zunächst vorgegeben, das wir bereits *fixieren* können. Heißt: Das Feld hat eine feste Belegung und wir probieren nur die anderen. Ein Feld nach dem anderen bekommt einen Wert zugewiesen und wird dadurch fixiert.

Für jedes Feld gibt es eine Abzweigung mit zwei Optionen: belegt und unbelegt. Zunächst weisen wir belegt zu und machen mit den nächsten Feldern weiter. Irgendwann wurden alle möglichen Figuren mit den bisherigen Zuweisungen ausprobiert. Wir weisen dann dem Feld unbelegt zu und suchen nun die andere Option ab. Nachdem auch hier alle möglichen Figuren abgesucht wurden, gehen wir zurück zur letzten Abzweigung.

Vorzeitiger Abbruch im Backtracking

Das ist erst einmal nur eine spezielle Weise, alle möglichen Figuren zu finden. Mit ihr kann allerdings die (best-case) Laufzeit wesentlich verkleinert werden. Wir sind nämlich nur an passenden Figuren interessiert. Oft kann die bisherige Wahl der Belegungen, noch bevor man alle Felder fixiert hat, schon keine passende Figur mehr ergeben. Hier kann man zwei Fälle unterscheiden:

- In einer Reihe wurden bereits mehr Felder als belegt gewählt, als in der Eingabe gefordert

$$\text{Anzahl belegte Felder} > \text{Anzahl belegte Felder nach Eingabe}$$

- In einer Reihe wurden so wenige Felder als unbelegt gewählt, dass mit den noch nicht fixierten Feldern die geforderte Anzahl nicht erreicht werden kann.

$$\text{nicht fixierte Felder} + \text{belegte Felder} < \text{Anzahl belegte Felder nach Eingabe}$$

Ist eine dieser Bedingungen erfüllt, dann kann keine passende Figur mehr entstehen. Egal wie die restlichen Felder gewählt werden, wird es in einer Reihe nicht die richtige Anzahl an belegten Feldern geben. Deswegen können wir diese Abzweigung direkt im Backtracking verwerfen und zur vorherigen Abzweigung zurückkehren. Dies macht Backtracking schneller als Brute-Force.

Zusammenfassen von Reihen

Als Vereinfachung kann man, statt die Belegung der Felder einzeln zu betrachten, immer eine ganze Reihe füllen. Statt an jedem Feld sich für eine Belegung zu entscheiden, gehen wir Reihen durch (z.B. alle Zeilen) und betrachten die möglichen Belegungen einer ganzen Reihe. Denn wir kennen für eine Reihe die Anzahl der belegten Felder und müssen nur noch diese auf die noch nicht zugewiesenen Felder der Reihe verteilen.

Sei n die Anzahl an Feldern in einer Reihe und k die Anzahl an belegten Feldern unter diesen. Probiert man alle Belegungen aus, gibt es 2^n Möglichkeiten. Es können aber nur die Möglichkeiten passend sein, in denen genau k Felder als belegt gewählt werden. Davon gibt es nur $\binom{n}{k}$, was weniger als 2^n ist. Durch das Zusammenfassen einer Reihe wird somit die Anzahl an betrachteten Optionen reduziert.

Dieser Ansatz ist bereits geeignet, um alle Beispiele außer dem letzten zu berechnen. Der Binomialkoeffizient $\binom{n}{k}$ ist am größten, wenn es ungefähr genau so viele belegte Felder wie unbelegte in einer Reihe gibt. Im letzten Beispiel ist das der Fall mit $\binom{12}{6} = 924$ Möglichkeiten pro Reihe. Alleine wenn wir vier Reihen ausprobieren, wären das $924^4 \approx 7 \cdot 10^{11}$. Bei nur vier Reihen wird man noch nicht so häufig abbrechen können, insbesondere weil auch die Diagonalen ungefähr die gleiche Anzahl an unbelegten und belegten Feldern haben. Das ist also immer noch zu langsam für das letzte Beispiel.

Reihenfolge der Reihen

Aus den Berechnungen folgt allerdings auch Folgendes: Besonders wenige Möglichkeiten gibt es, wenn k sehr klein ist. Zum Beispiel, wenn es gar keine belegten Felder geben soll. Dann gibt es nur eine Möglichkeit. Ebenso, wenn k sehr groß ist. Wenn zum Beispiel alle Felder belegt sein müssen, dann gibt es auch nur eine Möglichkeit.

Beim Backtracking kann man frei die Reihenfolge der Reihen wählen. Es kann also sinnvoll sein, zuerst die Reihen zu nehmen, die wenige Möglichkeiten haben. Denn diese stehen fast schon fest und es ist wahrscheinlicher, dass eine Wahl zu einer passenden Figur führt. Wir nehmen in jedem Schritt also die Reihe mit der kleinsten Zahl an möglichen Belegungen $\binom{n}{k}$.

Man beachte, dass n und k nicht notwendigerweise auf die ganze Reihe bezogen sind. Es kann sein, dass Felder bereits gefüllt wurden. Das aktuell betrachtete Feld (was wir testen) ist zum Beispiel fixiert. Im nächsten Abschnitt werden wir noch eine weitere Art erhalten, auf die Felder bereits fixiert werden können. Bei dieser Berechnung verwendet man die Anzahl an noch zu füllenden Feldern n und die Anzahl an darunter belegten Feldern k.

Zeilen und Spalten

Eine weitere Erkenntnis ist, dass man „kreuz und quer“ Reihen belegen kann, also auch abwechselnd horizontal oder vertikal. Wenn bei der Belegung Teile der Reihe bereits fixiert wurden, betrachtet man nur die noch nicht fixierten Felder. Um diese zu belegen, benötigt man unter diesen die Anzahl an belegten Feldern. Man kann die Anzahl berechnen durch Abziehen der Anzahl der bereits als belegt fixierten Felder von der Anzahl an Feldern, die belegt sein sollen in der ganzen Reihe.

$$\begin{aligned}\text{verbleibende belegte Felder} = {}& \text{als belegt zu wählende Felder in der ganzen Reihe} \\ & - \text{bereits als belegt gewählte Felder}\end{aligned}$$

Von allen möglichen Zeilen und Spalten nimmt man nun die, für die die wenigsten Belegungen in Frage kommen. Hiermit kann man auch das vorletzte Beispiel schnell berechnen. Denn darin kommen auch Zeilen und Spalten mit wenigen oder vielen belegten Feldern vor.

Ausnutzen der ganzen Figur

Die letzte Beispieleingabe kann als eine Herausforderung angesehen werden. Man kann sie mit einer weiteren Optimierung lösen. Momentan wird vom Backtracking nur die Information genutzt, ob eine passende Figur gefunden wurde. Durch das Backtracking erhalten wir jedoch nicht nur das, sondern auch eine Figur, wenn es möglich ist. Für alle Felder erhalten wir also eine mögliche Belegung. Wir speichern für alle Felder, dass man mit der jeweiligen Belegung eine passende Figur finden kann. Wird das Feld später bearbeitet, wissen wir für die Belegung schon, dass es eine passende Figur gibt. Das Backtracking müssen wir für die Belegung dann nicht nochmal ausführen.

Randomisierung der Reihenfolge

Im Backtracking werden Belegungen der Reihen ausprobiert, die die gewünschte Anzahl an belegten Feldern haben. Am einfachsten zu implementieren wäre eine lexikographische Reihenfolge. Die erste Belegung wäre, dass in der Reihe zuerst alle unbelegten Felder kommen und dann alle belegten. Dies ist jedoch ein sehr unwahrscheinlicher Fall. Man kann somit in einer zufälligen Reihenfolge die möglichen Belegungen durchgehen. Hierdurch verbessert sich insbesondere beim letzten Beispiel die Laufzeit enorm.

4.1.4 Lösung mit ILP

Eine alternative Lösung ergibt sich mit Integer Linear Programming (ILP). Das ganze Problem lässt sich auch in Form von Gleichungen formulieren. Jedem Feld weisen wir eine Variable zu, die nur 0 oder 1 sein kann. 0 steht für unbelegt und 1 für belegt. Die Figuren werden so auf Belegungen der Variablen abgebildet.

Es muss nun eine Belegung der Variablen gefunden werden, die bestimmte Vorgaben erfüllt. Diese sind immer eine Aussage, dass die Anzahl aller belegten Felder in einer Reihe eine bestimmte Zahl sein muss. Mit den Variablen ausgedrückt ist es eine Gleichung: Die Summe aus den zugehörigen Variablen muss gleich eine Konstante sein. Die Bedingungen für eine passende Figur können also als ein Gleichungssystem aus binären Variablen ausgedrückt werden. Verwendet man die beschriebene Vorgehensweise, Felder einzeln zu bearbeiten, kommt noch eine neue Bedingung hinzu: Der Wert von einer Variablen ist schon bekannt – nämlich vom Feld, das aktuell betrachtet wird. Nun muss man nur noch herausfinden, ob es eine Lösung des Gleichungssystem gibt.

Solche Gleichungssysteme tauchen auch bei Integer Linear Programming (ILP) auf. ILP ist ein sehr bekanntes Problem. Grob gesagt geht es darum, ein lineares Ungleichungssystem zu lösen und dabei eine (ebenfalls lineare) Zielfunktion zu maximieren. Die Variablen sind ganzzahlig. Für dieses Problem existieren bereits viele vorgefertigte Programme, sodass man nur noch das eigentliche Problem in ein ILP formulieren muss. Wir haben nun ein Gleichungssystem, das wir lösen wollen. Dies entspricht einem ILP ohne Zielfunktion. Wir können also einen ILP Solver verwenden, um herauszufinden, ob das Gleichungssystem lösbar ist.

Eine konkrete Formulierung als ILP wäre die folgende: Es gibt für jedes Feld in Zeile i und Spalte j eine ganzzahlige Unbekannte $x_{i,j}$. Diese sind beschränkt durch:

- $x_{i,j} \leq 1$ und $-1 \cdot x_{i,j} \leq 0$ damit $x_{i,j}$ binär ist.
- $\sum_{j=1}^{n} x_{i,j} = r_j$ für jede Zeile j mit r_j belegten Feldern
- $\sum_{i=1}^{n} x_{i,j} = c_i$ für jede Spalte i mit c_i belegten Feldern
- $\sum_{(i,j) \in \text{Diagonale l. u.}} x_{i,j} = d_k$ für jede Diagonale von links unten mit d_k belegten Feldern
- $\sum_{(i,j) \in \text{Diagonale l. o.}} x_{i,j} = e_k$ für jede Diagonale von links oben mit e_k belegten Feldern.
- $x_{s,t} = 0$ bzw. 1 für das betrachtete Feld $x_{s,t}$ mit fixierter Belegung

Die Gleichheiten lassen sich umwandeln zu Ungleichungen, indem man $\leq$ und $\geq$ fordert. Letzteres lässt sich mit $\leq$ darstellen, indem man beide Seiten negiert. Zum Beispiel sind die Bedingungen für die Zeilen (2. Stichpunkt) äquivalent zu:

$$\sum_{j=1}^{n} x_{i,j} \leq r_j \text{ und } \sum_{j=1}^{n} (-1) \cdot x_{i,j} \leq -r_j$$

4.1.5 Alternative Vorgehensweise

Statt wie am Anfang geschildert die Felder einzeln zu betrachten, kann man auch mit Backtracking alle passenden Figuren durchgehen. (Ohne Einschränkungen bei einem Feld.) Für alle passende Figuren speichert man für jedes Feld, dass die jeweilige Belegung möglich ist. Nachdem alle passenden Figuren gefunden wurden weiß man für jedes Feld, ob es immer belegt, unbelegt oder beides sein kann. Also würde man die gleiche Information auch erhalten. Dies kan schneller sein, wenn es nur wenige passende Figuren gibt. Vor allem ist dies bei den komplett eindeutigen Beispielen der Fall. Allerdings lässt sich das letzte Beispiel so nicht lösen, denn das hat sehr viele passende Figuren.

4.2 Umsetzung

Ein einfacher Weg, das Backtracking zu implementieren, ist Rekursion. Dabei sollte man auf konstante Faktoren achten und möglichst alles vorberechnen (z.B. Binomialkoeffizienten). Die bisherigen Belegungen kann man als ein 2D-Array übergeben. Darin kann man drei verschiedene Zustände für jedes Feld speichern: noch nicht fixiert, belegt, oder unbelegt. Da man an mehreren Stellen für jede Reihe die Anzahl an bereits als belegt/unbelegt fixierten Felder benötigt, kann man diese Zähler für eine effizientere Berechnung auch als Parameter übergeben.

Das Aktualisieren der Zähler und die Überprüfung der Abbruchbedingungen lassen sich zusammen erledigen. Die Zähler ändern sich, wenn man ein Feld fixiert. Und zwar nur in den Reihen, denen das Feld angehört. Das ist auch der einzige Zeitpunkt, in dem man in einen Zustand mit erfüllten Abbruchbedingungen gelangen kann, denn nur hier ändern sich die Zähler. Das Programm muss folglich nur beim Fixieren eines Feldes die Abbruchbedingungen überprüfen, und das auch nur für die Reihen, in denen das Feld vorkommt.

Fürs Backtracking müssen alle möglichen Belegungen einer ganzen Reihe durchgegangen werden. Hierfür kann man die Belegungen als Permutationen auffassen. Gesucht sind alle Anordnungen von 0 und 1 für unbelegte und belegte Felder, wobei die Anzahlen festgelegt sind.

Dafür bietet z.B. die C++ Standard Library eine Funktion an, Permutationen in lexikographischer Reihenfolge durchzugehen. Um nun in einer zufälligen Reihenfolge die Permutationen durchzugehen, kann man eine zufällige Permutation vorgenerieren. Mit dieser Permutation vertauscht man jede Anordnung und erhält die wirkliche Anordnung. Da es sich um eine Permutation handelt, geht man alle Anordnungen genau einmal durch.

4.3 Beispiele

Nachfolgend gibt es für jedes Beispiel eine Darstellung der eindeutigen Felder. „0“ steht für ein eindeutig unbelegtes Feld, „1“ für ein eindeutig belegtes Feld und „?“ für ein uneindeutiges Feld. Für die uneindeutigen Beispiele folgt jeweils eine mögliche passende Figur. In dieser wurden die Fragezeichen durch mögliche Belegungen in Lila ersetzt.

tomograph00.txt

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

tomograph01.txt

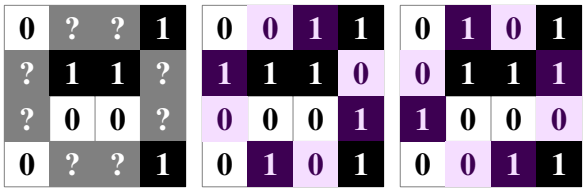
0	0
1	0

tomograph02.txt

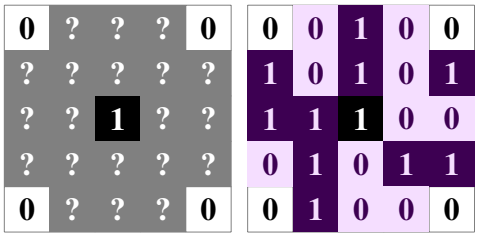
0	0	1	0
1	1	0	0
1	1	1	1
1	0	0	1

tomograph03.txt

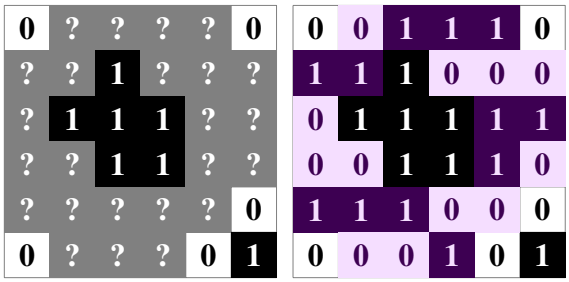
Dies ist ein kleinstmögliches Beispiel mit uneindeutigen Feldern. Es gibt nur zwei passende Figuren, die beide abgebildet werden.



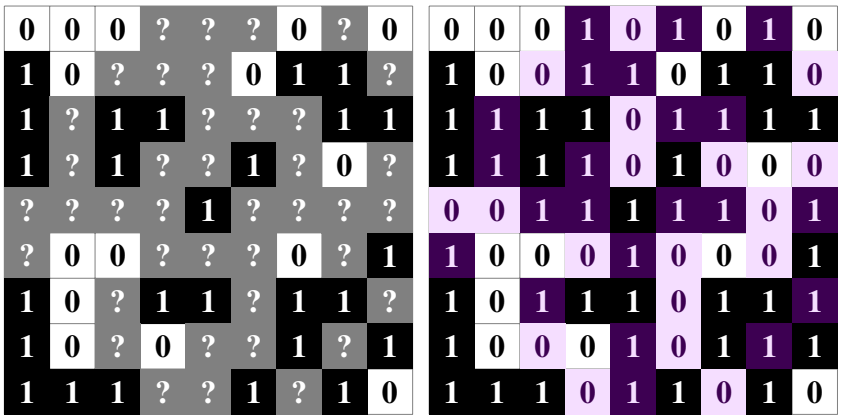
tomograph04.txt



tomograph05.txt



tomograph06.txt



tomograph07.txt

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

tomograph08.txt

Dieses Beispiel wurde zufällig generiert, indem immer eine Reihe eine einheitliche Belegung bekommen hat. Dabei werden mögliche vorherige Belegungen übermalt. Von der Konstruktion her ist es eindeutig lösbar durch „Entfernen“ von Reihen, die komplett belegt oder unbelegt sind. Durch diese Eigenschaft sollte es mit Optimierungen schnell berechnet werden können.

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

tomograph09.txt

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

tomograph10.txt

Zu dieser Eingabe gibt es sehr viele unterschiedliche passende Figuren. Es ist schwer schnell zu berechnen, da es in jeder Reihe fast genau so viele belegte wie unbelegte Felder geben soll. Dafür gibt es sehr viele mögliche Anordnungen in jeder Reihe.

0	?	?	?	?	?	?	?	?	?	?	1
?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?
1	?	?	?	?	?	?	?	?	?	?	0

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

4.4 Bewertungskriterien

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

- $[-1]$ **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**
  Die allgemeinen Anforderungen sind im Teil „Allgemeines“ im Abschnitt „Dokumentation“ erläutert.
- $[-1]$ **Modellierung ungeeignet**
  Die Felder müssen die Zustände *belegt*, *unbelegt* und *nicht eindeutig* annehmen können.
- $[-1]$ **Lösungsverfahren fehlerhaft**
  Das Verfahren muss korrekt sowohl eine passende Figur berechnen können, als auch, welche Feldbelegungen in dieser Figur eindeutig sind. Heuristische Ansätze, die nicht in allen Fällen die geforderten Ergebnisse korrekt berechnen können, führen zu Punktabzug.
- $[-1]$ **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Ein reines Brute-Force-Verfahren ist nicht ausreichend. Zusätzliche Optimierungen müssen vorhanden sein. Bei Backtracking kann z.B. ausgenutzt werden, dass die Anzahl an belegten Feldern bekannt ist. Das Programm muss für alle Beispiele bis einschließlich `tomograph09.txt` in angemessener Zeit eine Ausgabe erzeugen können.
- $[-1]$ **Ergebnisse schlecht nachvollziehbar**
  Für jedes Feld sollte klar sein, welche eindeutige Belegung es hat oder ob es uneindeutig ist. Diese Informationen müssen für alle Felder in einem sinnvollen Format (z.B. in einem Raster) ausgegeben werden.

  In der Ausgabe sollte mindestens eine nach den Regeln passende Figur enthalten sein. Außerdem muss erkennbar sein, welche Felder nicht eindeutig belegt sind.
- $[-1]$ **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll Ergebnisse zu den Beispielen `tomograph01.txt` bis `tomograph09.txt` enthalten. Es ist in Ordnung, wenn zu `tomograph10.txt` kein Ergebnis dokumentiert wird oder ihm eine sehr lange Rechenzeit zu Grunde liegt. Bei der Verwendung von ILP-Solvern muss für alle vorgegebenen Beispiele das Ergebnis dokumentiert sein.

Aufgabe 5: Großstadtbauern

5.1 Lösungsidee

Niemand möchte mit Nemo tauschen, denn der Arme soll für seinen Wohnblock einen Gemüse-Anbauplan entwickeln. Dabei ist die Grundstruktur des Plans vorgegeben, und die in jedem Monat wechselnden Gemüsekombinationen des Plans sollen möglichst viele Rezepte aus einer Wunschliste abdecken.

Brute-Force

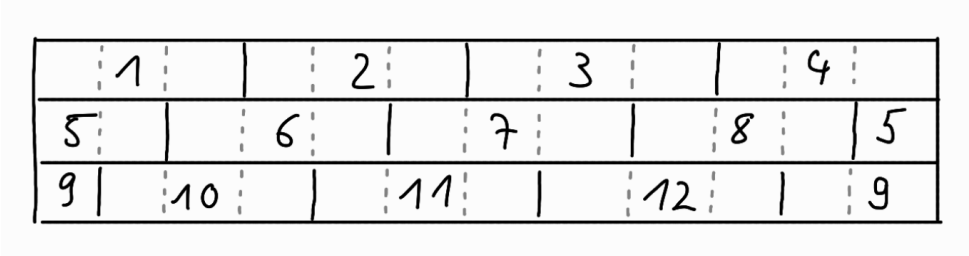


Abbildung 5.1: Reihenfolge, in der wir die Anbauslots speichern.

Ein einfacher Lösungsansatz für die Aufgabe ist ein Brute-Force-Algorithmus, der alle Anbaupläne durchprobiert, um den besten zu finden. Dabei speichert der Algorithmus in einer Variable den bisher besten Anbauplan und in einer weiteren Variable, wie viele der Rezepte mit dem bisher besten Plan gekocht werden können. Falls nun ein Anbauplan ausprobiert wird, mit dem mehr Rezepte gekocht werden können, wird dieser als neuer bester Anbauplan gespeichert.

Ein Anbauslot sind drei Anbauplätze, in denen durch die Einschränkungen der Aufgabenstellung die gleichen Zutaten stehen müssen. Einen Anbauplan stellen wir als 12-Tupel von den Zutaten in den Slots in der Reihenfolge aus Abbildung 5.1 dar. (Jede andere Reihenfolge würde auch funktionieren, wir müssen aber eine festlegen.) Dabei seien $I(m)$ die Indizes der drei Slots, die im Monat m bepflanzt werden. Beispielsweise wäre $I(1) = (1,5,9)$. Das ganze Verfahren ist Algorithmus 5.1.

Dieses Verfahren ist zwar einfach, aber für größere Eingaben zu langsam. Das liegt daran, dass wir bei n Zutaten n^{12} Anbaupläne durchprobieren müssen. Um die größeren Eingaben lösen zu können, müssen wir uns also überlegen, wie wir die zu probierenden Anbaupläne reduzieren können.

Rezepte statt Pflanzen durchprobieren

Eine erste Überlegung ist, dass wir bereits bei der Wahl der Pflanzen für einen Monat versuchen, Rezepte zu kochen. Statt für einen Monat also alle Kombinationen von Pflanzen durchzuprobieren, probieren wir nur die Pflanzen aus, mit denen ein neues Rezept gekocht werden kann, oder wir kochen in einem Monat kein Rezept und legen damit auch keine Pflanzen fest. Falls wir in einem Monat die Möglichkeit haben, für ein Rezept die Pflanzen in mehreren Reihenfolgen zu pflanzen, müssen wir alle diese Reihenfolgen ausprobieren.

Algorithmus 5.1 Brute-Force

```
procedure BRUTE FORCE(Rezepte R, Zutaten Z)
    n_max ← 0                ▷ n_max gibt an, was die bisher größte Anzahl kochbarer Rezepte ist
    B_max ← 0                ▷ B_max gibt die Belegung an, die n_max realisiert
    for B ∈ Z^12 do
        n ← ERFÜLLTEREZEPTE(B, R)
        if n > n_max then
            n_max ← n
            B_max ← B
        end if
    end for
    return B_max
end procedure
procedure ERFÜLLTEREZEPTE(Anbauplan B, Rezepte R)
    K ← ∅                    ▷ In K speichern wir alle verfügbaren Zutatenkombinationen
    for m ∈ {0, ..., 11} do
        i_1, i_2, i_3 ← I(m)
        z_1 ← B[i_1]
        z_2 ← B[i_2]
        z_3 ← B[i_3]
        K ← K ∪ {{z_1, z_2, z_3}}
    end for
    return |K ∩ R|
end procedure
```

Im Folgenden sagen wir, ein (unvollständiger) Anbauplan realisiert ein Rezept, wenn es bereits in einem Monat kochbar ist, und ein Rezept ist in einem (unvollständigen) Anbauplan realisierbar, wenn die freien Slots des Anbauplans so belegt werden können, dass das Rezept gekocht werden kann. Genau so bezeichnen wir eine Menge von Rezepten in einem Anbauplan realisierbar, wenn es eine Belegung der freien Slots gibt, mit der alle Rezepte in der Menge kochbar sind.

Wir stellen die unvollständigen Anbaupläne wie oben als 12-Tupel dar, die aber neben Pflanzen auch den Platzhalter -1 enthalten dürfen. Dieser gibt an, dass in diesem Slot noch keine Pflanze festgelegt wurde. Um den Pseudocode zu vereinfachen, greifen wir für einen Bepflanzungsplan B mit $B[i]$ nicht auf die i-te Pflanze zu, sondern auf die drei Pflanzen beziehungsweise Platzhalter, die im i-ten Monat angepflanzt werden. Wir erhalten damit Algorithmus 5.2. Dort bezeichnet $\text{Perm}(n)$ alle Permutationsfunktionen von n-Tupeln.

Algorithmus 5.2 Rezepte probieren

```
procedure REZEPTE PROBIEREN(Bepflanzungsplan B, Übrige Rezepte R, Monat m)
    if m = 13 then                          ▷ Falls Prozedur vom letzten Monat aus aufgerufen wird,
        return []                                                  ▷ gib leere Liste zurück.
    end if
    R_max ← []
    n_max ← 0
    for r_neu ∈ R do
        for p ∈ Perm(3) do          ▷ Perm(3) bezeichnet alle Permutationen von 3 Elementen.
            r_ord ← p(r_neu)  ▷ p ∈ Perm(3) ist Funktion, die 3-Tupel entsprechend permutiert.
            if Für alle i ∈ {1,2,3} : B[m][i] = −1 ∨ r_ord[i] = B[m][i] then
                alt ← B[m]
                B[m] ← r_ord
                R_neu ← REZEPTE PROBIEREN(B, R \ {r_neu}, m+1)
                B[m] ← alt
                R_neu ← [r_ord] + R_neu
                if |{r ∈ R_neu | r ≠ −1}| > n_max then          ▷ Falls in der neuen Lösung mehr
                    R_max ← R_neu       ▷ Rezepte umgesetzt werden als in der bisher besten, ist
                    n_max ← |{r ∈ R_neu | r ≠ −1}|                ▷ diese die neue beste Lösung.
                end if
            end if
        end for
    end for
    R_neu ← REZEPTE PROBIEREN(B, R, m+1)      ▷ Wir probieren auch, ob wir einen guten
    R_neu ← [−1] + R_neu     ▷ Anbauplan erhalten, wenn wir im Monat m kein Rezept kochen.
    if |{r ∈ R_neu | r ≠ −1}| > n_max then
        R_max ← R_neu
    end if
    return R_max
end procedure
```

Diese rekursive Methode kann nun mit einem leeren Anbauplan A, allen Rezepten R und dem Monat 1 aufgerufen werden. Der Rückgabewert ist eine Liste, die für jeden Monat das in diesem Monat umgesetzte Rezept in der richtigen Reihenfolge enthält, oder -1, falls in dem Monat kein Rezept gekocht werden kann.

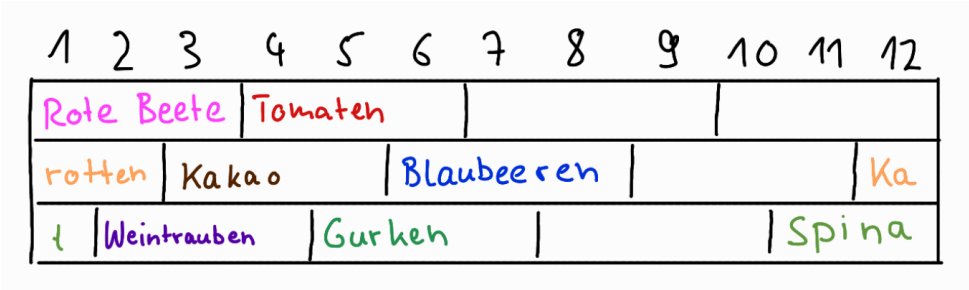


Abbildung 5.2: In dieser Teillösung können wir noch nicht realisierte Rezepte nur noch in den 6 Monaten Juli bis Dezember realisiert werden. Wir können also höchstens 6 weitere Rezepte realisieren.

Pruning

Leider haben wir mit dieser Vereinfachung noch immer sehr viele Pläne durchzuprobieren. Um diesen Suchraum zu verkleinern, überlegen wir uns, wie wir vorzeitig feststellen können, ob eine Teillösung zu einer besseren Lösung als der bisher besten führen kann oder nicht, um diese dann eventuell nicht erweitern zu müssen.

Für ein vorzeitiges Abbrechen der Suche nutzen wir folgende Beobachtung: Eine Teillösung, die bereits k Rezepte realisiert und in m Monaten noch nicht alle Pflanzen festgelegt hat, kann als Gesamtlösung höchstens $k+m$ Rezepte realisieren, nämlich wenn wir in allen übrigen Monaten ein neues Rezept realisieren (siehe Abbildung 5.2). Wenn wir also bereits eine Lösung gefunden haben, die $n \geq k+m$ Rezepte realisiert, brauchen wir diese Teillösung nicht zu erweitern.

Wir können nun Algorithmus 5.2 modifizieren, in dem wir ein weiteres Argument z einführen, welches angibt, wie viele weitere Rezepte mindestens realisiert werden sollen. Dann können wir einfach ausrechnen, ob dieses Ziel bei perfekter Ausnutzung der Monate noch erreicht werden kann und, falls nicht, die Suche als erfolglos abbrechen. Falls wir in der aktuellen Rekursionsstufe noch kein $R_{\max}$ gefunden haben, so dass $n_{\max} \geq z$, dann geben wir an den Rekursionsaufruf $z-1$ weiter, falls wir im Monat der Rekursionsstufe ein Rezept kochen, und z, falls wir keines kochen. Falls wir ein $R_{\max}$ mit $n_{\max} \geq z$ gefunden haben, geben wir $n_{\max}$ weiter, falls wir in der Rekursionsstufe ein Rezept kochen, und $n_{\max}+1$, falls wir keines kochen.

Verschiebungssymmetrie

Wir brauchen für die folgenden Abschnitte eine wichtige Beobachtung: Wir können einen Anbauplan monateweise verschieben, ohne die kochbaren Rezepte zu verändern. Dafür verschieben wir für jeden Slot die Zutat in den Slot, der einen Monat später anfängt (siehe Abbildung 5.3). Für eine Verschiebung um mehrere Monate können wir die einfache Verschiebung beliebig oft wiederholen. Die Lösungen haben also eine Verschiebungssymmetrie.

Symmetrien ausnutzen

Die Verschiebungssymmetrie bringt uns, dass wir viele potentielle Lösungen bei der Suche nicht betrachten müssen: Zunächst erlaubt sie uns festzulegen, dass das erste Rezept einer Liste von Rezepten, die realisiert werden soll, immer im Januar realisiert wird. Weiterhin, wenn wir in der ersten Rekursionsstufe ein Rezept auswählen, probieren wir bis auf Verschiebung

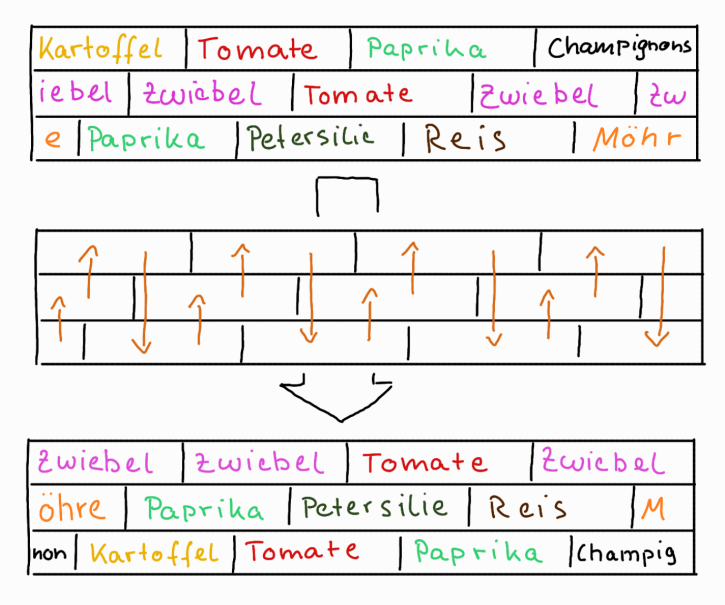


Abbildung 5.3: Aus dem Anbauplan oben wird durch die Symmetrieverschiebung in der Mitte der Anbauplan unten. In beiden Anbauplänen können die gleichen Rezepte gekocht werden.

alle Lösungen aus, die dieses Rezept realisieren. Deshalb müssen wir, wenn wir in der ersten Rekursionsstufe das nächste Rezept auswählen, keine weiteren Lösungen ausprobieren, die das erste Rezept enthalten. Diese haben wir ja bis auf Verschiebung schon ausprobiert. Allgemeiner müssen wir, wenn wir ein Rezept in der ersten Rekursionsstufe auswählen, nach diesem Rezept keines der zuvor in der ersten Stufe ausgewählten Rezepte versuchen zu realisieren.

5.2 Umsetzung

Die Pseudocodes lassen sich ziemlich direkt in Python-Code umsetzen. Anbaupläne lassen sich als Liste speichern. Um wie oben auf die Pflanzen den Monaten nach zuzugreifen, ist es sinnvoll, entweder die Funktionen `set_month` und `get_month` zu implementieren oder diese Funktionalität in einer Klasse zu bündeln.

Für die Permutationen der Rezepte sollte man nicht wie im Pseudocode eine Menge von Funktionen führen. Stattdessen ist es sinnvoller, eine Funktion `reorder` zu implementieren, die eine Reihenfolge (zum Beispiel kodiert als Tupel der Werte 0, 1, 2 in der gewünschten Reihenfolge) und ein Rezept entgegennimmt und das Rezept in der gewünschten Reihenfolge zurückgibt. Da es nicht sehr viele Rezepte und nicht sehr viele Reihenfolgen gibt, können wir für `reorder` den `lru_cache` aus dem Modul `functools` benutzen. Dieser speichert nach jedem Aufruf der Funktion den Rückgabewert für die Funktionsargumente in einem `dict`, um sie beim nächsten mal nicht wieder berechnen zu müssen. Um Rezepte und Mengen von Rezepten darzustellen, kann man den Datentyp `set` für endliche Mengen benutzen.

5.3 Beispiele

Durch die Verschiebungssymmetrie der Lösungen gibt es fast immer mehr als eine optimale Lösung. Bei mehreren Beispieleingaben gibt es sogar verschiedene optimale Lösungen, die nicht durch Verschiebungen ineinander überführt werden können. Dennoch ist zu jeder Eingabe nur eine optimale Lösung angegeben. Wenn eine andere Lösung genauso viele Rezepte realisiert, ist sie auch optimal. Die Namen der Pflanzen wurden der Übersichtlichkeit halber abgekürzt.

bauern1.txt

Monat	1	2	3	4	5	6	7	8	9	10	11	12
GH 1	K	K	K	K	K	K	Z	Z	Z	Z	Z	Z
GH 2	M	M	R	R	R	M	M	M	M	M	M	M
GH 3	R	Z	Z	Z	R	R	R	R	R	R	R	R

Insgesamt 4 kochbare Rezepte:

```
Kartoffel, Möhre, Reis
Kartoffel, Zwiebel, Möhre
Kartoffel, Zwiebel, Reis
Zwiebel, Möhre, Reis
```

bauern2.txt

Monat	1	2	3	4	5	6	7	8	9	10	11	12
GH 1	Pa	Pa	Pa	T	T	T	K	K	K	C	C	C
GH 2	T	T	K	K	K	Z	Z	Z	Z	Z	Z	T
GH 3	R	Z	Z	Z	Pe	Pe	Pe	M	M	M	R	R

Insgesamt 6 kochbare Rezepte:

```
Kartoffel, Paprika, Zwiebel
Kartoffel, Zwiebel, Möhre
Paprika, Zwiebel, Tomate
Reis, Champignons, Zwiebel
Reis, Paprika, Tomate
Tomate, Zwiebel, Petersilie
```

bauern3.txt

Monat	1	2	3	4	5	6	7	8	9	10	11	12
GH 1	Zuc	Zuc	Zuc	Zwi	Zwi	Zwi	Tom	Tom	Tom	Kar	Kar	Kar
GH 2	Lin	Lin	Kar	Kar	Kar	Pap	Pap	Pap	Ros	Ros	Ros	Lin
GH 3	Pas	Pap	Pap	Pap	Möh	Möh	Möh	Zwi	Zwi	Zwi	Pas	Pas

Insgesamt 8 kochbare Rezepte:

```
Kartoffel, Pastinake, Rosenkohl
Kartoffel, Zucchini, Paprika
Kartoffel, Zwiebel, Möhre
Kartoffel, Zwiebel, Paprika
Linse, Zucchini, Paprika
Linse, Zucchini, Pastinake
Möhre, Paprika, Tomate
Rosenkohl, Tomate, Zwiebel
```

bauern4.txt

Monat	1	2	3	4	5	6	7	8	9	10	11	12
GH 1	Apf	Apf	Apf	Ban	Ban	Ban	Pap	Pap	Pap	Pas	Pas	Pas
GH 2	Kar	Kar	Koh	Koh	Koh	Kar	Kar	Kar	Lin	Lin	Lin	Kar
GH 3	Tom	Pap	Pap	Pap	Zwi	Zwi	Zwi	Zuc	Zuc	Zuc	Tom	Tom

Insgesamt 9 kochbare Rezepte:

```
Apfel, Kartoffel, Paprika
Apfel, Kartoffel, Tomate
Apfel, Kohlrabi, Paprika
Banane, Kohlrabi, Paprika
Banane, Kohlrabi, Zwiebel
Kartoffel, Zucchini, Paprika
Kartoffel, Zwiebel, Paprika
Linsen, Zucchini, Paprika
Linsen, Zucchini, Pastinake
```

bauern5.txt

Monat	1	2	3	4	5	6	7	8	9	10	11	12
GH 1	W	W	W	J	J	J	T	T	T	J	J	J
GH 2	V	C	C	C	S	S	S	N	N	N	V	V
GH 3	L	L	B	B	B	U	U	U	Z	Z	Z	L

Insgesamt 9 kochbare Rezepte:

```
W, C, B
J, C, B
J, S, B
J, S, U
T, S, U
T, N, Z
J, N, Z
J, V, Z
J, V, L
```

bauern6.txt

Monat	1	2	3	4	5	6	7	8	9	10	11	12
GH 1	V	V	V	P	P	P	O	O	O	W	W	W
GH 2	K	G	G	G	A	A	A	I	I	I	K	K
GH 3	C	C	K	K	K	B	B	B	S	S	S	C

Insgesamt 10 kochbare Rezepte:

```
C, K, V
G, K, V
G, K, P
A, K, P
A, B, P
A, B, O
B, I, O
I, O, S
K, S, W
C, K, W
```

5.4 Bewertungskriterien

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

- [−1] **Verfahren unzureichend begründet bzw. schlecht nachvollziehbar**
  Die allgemeinen Anforderungen sind im Teil „Allgemeines“ im Abschnitt „Dokumentation“ erläutert.
- [−1] **Modellierung ungeeignet**
  Die Überlappung der Anbauslots, auch über den Jahreswechsel hinweg, muss geeignet modelliert werden. Die Anbaupläne müssen die Vorgaben der Aufgabenstellung erfüllen, also genau 12 Monate mit je drei Anbauslots pro Monat umfassen.
- [−1] **Lösungsverfahren fehlerhaft**
  Das Lösungsverfahren sollte die Anzahl kochbarer Rezepte maximieren und eine optimale Lösung für die Beispiele `bauern1.txt` bis `bauern4.txt` finden. Abkürzungen der Suche sollten nicht dazu führen können, dass keine optimale Lösung gefunden wird.
- [−1] **Verfahren bzw. Implementierung unnötig aufwendig / ineffizient**
  Ein unoptimierter Brute-Force-Ansatz, der alle möglichen Anbaupläne generiert und anschließend auf kochbare Rezepte testet, ist nicht ausreichend. Das Programm muss für alle Beispiele bis einschließlich `bauern4.txt` in angemessener Zeit eine Ausgabe erzeugen können.
- [−1] **Ergebnisse schlecht nachvollziehbar**
  Es muss jeweils ein Anbauplan und die Maximalanzahl der kochbaren Rezepte dokumentiert werden. Es genügt, wenn die Maximalanzahl aus der Ausgabe leicht abgelesen werden kann. Aus dem Anbauplan muss ablesbar sein, in welchem Monat welche Pflanze in welchem Gewächshaus angebaut wird. Die Namen der Pflanzen dürfen der Übersichtlichkeit halber gekürzt werden.
- [−1] **Beispiele fehlerhaft bzw. zu wenige oder ungeeignete Beispiele**
  Die Dokumentation soll mindestens Ergebnisse zu den Beispielen `bauern1.txt` bis `bauern4.txt` enthalten. Die Beispiele `bauern5.txt` und `bauern6.txt` müssen nicht enthalten sein.

Beispiel	**Gesamtanzahl Rezepte**	**Maximalanzahl kochbarer Rezepte**
`bauern1.txt`	4	4
`bauern2.txt`	6	6
`bauern3.txt`	19	8
`bauern4.txt`	36	9
`bauern5.txt`	50	9
`bauern6.txt`	100	10

Aus den Einsendungen: Perlen der Informatik

Nach den Perlen der Informatik wird zur Zeit noch getaucht.