Aufgabe 2: Alles Käse

Teilnahme-ID: 64650

Bearbeiter dieser Aufgabe:
Arne de Borman

17. April 2023

Inhaltsverzeichnis

1 Lösungsidee

a)

Die Startscheibe ist die Scheibe, die beim Abschneiden der Käsescheiben vom Käse als letztes überbleibt. Ein potenzieller Käse ist ein Käse, der möglicherweise Teil des ursprünglichen Käses ist. Die restlichen Scheiben sind die Scheiben, die noch nicht für den möglichen Käse verwendet wurden. Um den Käse wiederherzustellen, wird mit einer Startscheibe als möglicher Käse begonnen und dann werden kontinuierlich Käsescheiben hinzugefügt, die zu einer Seite des Käses passen, bis der Ursprüngliche Käse wieder zusammengefügt wurde. Da nicht bekannt ist, welche Scheibe die tatsächliche Startscheibe ist, werden alle Scheiben als Startscheiben ausprobiert. Dazu werden sie als möglicher Käse in einer Liste gespeichert. An jeden dieser Käse werden, falls möglich, passende Scheiben an die Seiten angefügt und somit neue Käse erzeugt. Die Ergebnisse werden in einer neuen Liste gespeichert.

Nachdem alle Käse mit passenden Scheiben erweitert wurden, wird die alte Liste mit dieser neuen Liste ersetzt. Dieser Vorgang wird kontinuierlich wiederholt. Wenn für einen Käse alle Scheiben benutzt wurde, so wird er zurückgegeben.

```
poss_paths = mögliche Käse aus Startscheiben und restliche Scheiben, ohne Startscheibe
SOLANGE poss_paths nicht leer ist TUE
        neue_mögliche_pfade = []
        FUER käse, verbleibende_stücke IN poss_paths TUE
                FUER seite IN käse TUE
                        WENN seite in verbleibende_stücke TUE
                                neuer_käse = käse + seite
                                neue_stücke = verbleibende_stücke ohne seite
                                neue_mögliche_pfade.push(neuer_käse, neue_stücke)
    WENN schon alle Scheiben benutzt wurden TUE
        // => Der Käse muss vollständig sein
        WENN es nur einen möglichen vollständigen Käse gibt TUE
            (käse, _) = neue_mögliche_pfade[0]
            RETURN käse //ursprünglicher Käse gefunden
        SONST
            //Fehler, mehr als ein möglicher Käse wurde gefunden
        poss_paths = neue_mögliche_pfade
//Fehler, kein Käse wurde gefunden
```

Um schnell für einen Käse mögliche passende Stücke finden zu können, werden die restliche Scheiben für einen Käse in einer HashMap gespeichert. In der HashMap ist der Key das gesuchte Stück und die Value die Anzahl dieser Stücke, da es von einem Stück mehr als eins geben kann.

b)

Hier wird die ursprüngliche Aufgabenstellung sowohl durch das Vermischen der Scheiben mehrerer Käse als auch das Entfernen einiger Käsescheiben erweitert.

1.0.1 Mehrere Käse

Es wird anhand der Gesamtanzahl der Käsescheiben eine angenommene Mindestlänge für die Käse definiert. Die Mindestlänge gibt an, wie viele Scheiben mindestens für einen Käse verwendet werden müssen, damit er akzeptiert wird. Können nun für einen Pfad keine neuen Scheiben mehr gefunden werden, es wurden aber schon mehr Scheiben für einen Käse benutzt, als die Mindestanzahl wird ein neuer Käse vermutet. Die benutzten Scheiben werden dann aus der Scheibenliste entfernt und die Mindestlänge wird dementsprechend heruntergesetzt. Die Suche nach Käse wird wieder mit weniger Scheiben begonnen. Kann bei einem Durchlauf kein Käse gefunden werden, so wird die Mindestlänge heruntergesetzt. Dadurch werden die Scheiben nach und nach zuerst zu den größeren und dann den kleineren Käse zusammengesetzt.

1.0.2 Finden gegessener Scheiben

Hier wird davon ausgegangen, dass beim Abschneiden nie zwei Käsescheiben hintereinander aufgegessen werden. Wenn für einen möglichen Käse keine passenden Scheiben gefunden werden können, so wird nach gegessenen Scheiben gesucht. Um festzustellen, ob wirklich eine Scheibe gegessen wurde, werden den Seitenlängen des Käses nacheinander 1 (mm) hinzugefügt. Kann nun eine Scheibe für einen dieser Käse gefunden werden, so wird vermutet, dass an der Seite, an der der Millimeter hinzugefügt wurde, eine Scheibe fehlt, die aufgegessen wurde.

2 Umsetzung

Die Lösungsidee wird in der Programmiersprache Rust implementiert um die hohe Anzahl an Käsescheiben einiger Beispiele schnell zusammensetzen zu können.

2.1 Pfade

Ein möglicher Pfad hat drei Werte: den möglichen Käse, die Reihenfolge der bisher benutzten Scheiben und die übrigen Scheiben.

```
pub struct PossPath { //Ein möglicher Pfad
    cheese: Cheese,
    prev_pieces: PrevPieces,
    pieces_left: Box<PiecesMap>,
}
```

Die übriggebliebenen Scheiben werden im Heap gespeichert, sodass sie ohne Kopieren zwischen Funktionen hin und hergegeben werden können. Das PrevPieces Objekt speichert die Scheiben in einer Linked-List artigen Datenstruktur, bei der mehrere Knoten auf denselben Knoten zeigen können. Dadurch können mehrere Pfade die gleiche Reihenfolge an Scheiben teilen, ohne sie kopieren zu müssen, was die Laufzeit erheblich verschlechtern würde.

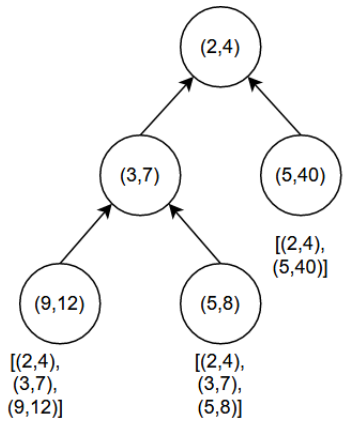

Abbildung 1: Speicherung der benutzten Scheiben in einer Linked-List artigen Struktur

Es wird davon ausgegangen, dass verschiedene Käse auch verschiedene Startscheiben haben. Deshalb werden die Pfade getrennt nach Startscheibe gespeichert, um diese Käse unterscheiden zu können. Es gibt top-Paths, die nach Startscheibe getrennt sind und jeweils mehrere sub-Paths haben. Diese speichern dann die PossPath-Objekte. In einem Schritt werden für alle Käse der Pfade neue Scheiben gefunden und neue Pfade erzeugt.

```
SOLANGE top_paths nicht leer ist TUE
    new_top_paths = []
    FUER sub_path in top_paths TUE
        new_paths = []
        FUER poss_path in sub_paths TUE
            new_paths.extend(poss_path.gen_new_paths())
        new_top_paths.push(new_paths)
    top_paths = new_top_paths
```

Dies geschieht in der construct_cheese Funktion.

2.2 Finden passender Scheiben für einen möglichen Käse

Eine Scheibe wird in folgender Klasse gespeichert:

```
pub struct Piece(pub u32, pub u32);
```

Scheiben werden immer so gespeichert, dass der nullte Wert größer ist als der Erste. Wenn dies beim Auslesen der Datei für ein Käsestück nicht der Fall ist, wird die Käsescheibe gedreht, indem die beiden Werte

vertauscht werden. Dadurch können gleiche Käsescheiben gefunden werden, ohne dass beide Ausrichtungen auszuprobiert werden müssen. Dies ist jedoch eigentlich bei den BwInf-Beispieleingaben nicht nötig, da hier sowieso immer der größere Wert als erstes angegeben wird. Da ich das aber nicht voraussetzten will überprüfe ich trotdem die Orientierung. Ein Käsequader wird folgendermaßen gespeichert:

```
pub struct Cheese {
    //a>=b>=c
    pub size: [u32; 3],
}
```

Hier werden die Seitenlängen auch der Größe nach abgespeichert, sodass die Seiten des Quaders den Scheiben entsprechen. Da ein Quader sechs Seiten hat, von denen jeweils zwei gleich sind, werden drei der Seiten benutzt, um nach Scheiben zu suchen. Um die Größe dieser gesuchten Scheiben zu bestimmen werden die Seitenlängen des möglichen Käses benutzt.

```
impl Cheese{
    /* ... */
    ///gibt die Seitenlängen zurück, die für die Erzeugung einer Seite verwendet werden
    pub fn get_sides_n() -> Vec<(usize, usize)> {
        vec![(1, 2), (0, 2), (0, 1)]
    }
    ///gibt die Seiten des Käsestücks zurück
    pub fn get_sides(&self) -> Vec<Piece> {
        Cheese::get_sides_n()
            .into_iter()
            .map(|(a, b)| Piece(self.size[a], self.size[b]))
            .collect()
    }
    /* ... */
}
```

Das Finden dieser gesuchten Seiten geschieht in der find_new_sides Methode. Es kann vorkommen, dass von diesen drei Seiten einige gleich sind. Damit nicht zwei identische neue Pfade erzeugt werden, werden Seiten gespeichert und dann übersprungen wenn sie schon gesehen wurden. Danach wird überprüft, ob diese Seiten sich in den restlichen Käsescheiben befinden (mehr dazu unter „Speicherung der Scheiben“). Wenn ja, werden sie in einem NewSide-Objekt zurückgegeben. Dieses Objekt speichert neben der Scheibe die Nummer der Seite an die die Käsescheibe angefügt werden kann. Dies geschieht in der Cheese::find_new_sides Methode.

Danach werden mithilfe dieser Informationen neue PossPath-Objekte erzeugt. Dies geschieht in der new_sides_to_path Methode. Dafür wird ein neuer Käse erzeugt, der auf der gefundenen Seiten erweitert ist. Hier wird wieder die Reihenfolge der Seitenlängen überprüft.

2.3 Speicherung der Scheiben

Die übergebliebenen Scheiben für jeden möglichen Käse werden wie oben angegeben in einer HashMap gespeichert. Da ein möglicher Käse eventuell mehrere Scheiben hat die an ihn passen, müssten die HashMaps in jedem Schritt kopiert werden, was bei einer hohen Anzahl an Scheiben lange dauern kann. Deshalb wird hier eine etwas effizientere Lösung benötigt. Die übrigen Scheiben werden in einer Klasse namens PiecesMap gespeichert. Die Klasse hat einmal die base-Hashmap die gleichbleibt, wenn man eine Scheibe benutzt und die Werte verändert werden. Verändert werden die Werte nur in der added-Hashmap, die am Anfang leer ist.

```
struct PiecesMap {
    base: Rc<FxHashMap<Piece, u32>>,
    base_id: Uuid,
    added: FxHashMap<Piece, u32>,
}
```

Wenn man nach einer Scheibe in einer PiecesMap-Instanz sucht, so wird zuerst nach ihr in der added Hashmap gesucht. Wird in ihr nichts gefunden, so wird der Wert aus der Base-Hashmap zurückgegeben.

Dadurch muss beim Kopieren einer Instanz nur die added-Hashmap kopiert werden und die base-Hashmap kann über mehrere Instanzen hinweg geteilt werden.

Wenn die added HashMap zu groß wird, so wird eine neue base-Hashmap erzeugt, um die Zeit, die für das Kopieren benötigt wird, möglichst klein zu halten.

```
impl PiecesMap{
    /* ... */
    fn make_copy(&mut self) -> Self {
        if self.added.len() > self.base.len() / 10 {
            //Die Länge der added-Hashmap auf ein 10-tel der base Hashmap einzuschränken
            //scheint beim ausprobieren zu einem guten kompromiss zwischen kopier-
            //und merge-dauer zu ermöglichen
            self.merge();
        }
        self.clone()
    }
    /* ... */
}
```

Außerdem muss nicht für jeden neuen Käse die Scheibenliste kopiert werden. Einer der Käse kann die Scheibenliste übernehmen, was besonders wenn es nur eine mögliche Scheibe gibt die angefügt werden kann viel Zeit spart.

```
//wandle zuerst die überbleibenden Scheibenliste in einen Pointer um,
//damit sie einmal weniger kopiert werden müssen
let pieces_ptr = Box::into_raw(pieces);
let n_side = new_sides.len();
new_sides
    .into_iter()
    .enumerate()
    .map(move |(i, new_side)| {
        //wenn es die letzte neue Seite ist, verwende die Scheibenliste,
        //ohne sie zu kopieren
        let mut pieces = if i == n_side - 1 {
            unsafe { Box::from_raw(pieces_ptr) }
        } else {
            //ansonsten kopiere die Liste
             Box::new(unsafe { pieces_ptr.as_mut() }.unwrap().make_copy())
        };
        /*
        ...
        */
    })
```

2.4 Mehrere Käse

Um mehrer Käse zusammensetzen zu können wird, wie in der Lösungsidee beschrieben, eine Mindestlänge definiert. Diese definiert, ab welcher Länge ein Käse akzeptiert wird. Diese ist anfangs 75% der Scheibenanzahl. Dieser Anteil hat sich bei den Beispieleingaben als guter Wert erwiesen, da hier die größeren Käse um einiges mehr Scheiben haben als die kleineren Käse. Wird ein Käse gefunden wird anhand der Anzahl der verbleibenden Scheiben eine neue Mindestanzahl berechnet. Wenn kein Käse gefunden werden kann wird die Mindestanzahl halbiert, um kleinere Käse finden zu können. Dies geschieht in der construct_cheeses Funktion.

2.5 Finden gegessener Scheiben

Um möglicherweise gegessene Scheiben zu finden, wird, wie in der Lösungsidee erwähnt den Seiten des Käses ein Millimeter hinzugefügt und dann geschaut, ob nun eine Scheibe für die Seiten des Käses gefunden wird. Um das zu ermöglichen, wird der Käse hintereinander von den drei verschiedenen Seiten betrachtet. Es wird eine x,y und z Seitenlänge bestimmt, wobei x und y die Seite ausmachen, die gerade betrachtet wird. Dann wird ein Stück erzeugt, bei dem die x-Seite um eins größer ist, und Eins, bei dem die y-Seite Seite um eins größer ist. Wenn man nun eines dieser vergrößerten Stücke in den übrigbleibenden Scheiben findet, kann man vermuten, dass ein Stück des Käses fehlt, das die Dimensionen der erweiterten Seite hat. Wenn man also ein Stück mit erweiterter x-Seite in den restlichen Scheiben wiederfindet, wird vermutet, dass ein Stück der Größe (x|y) fehlt (siehe Abb. 2). Die gefundene Scheibe wäre dann die Scheibe die nach dem fehlendem Stück an den Käse angefügt wird. Bei der Suche werden nur Seiten

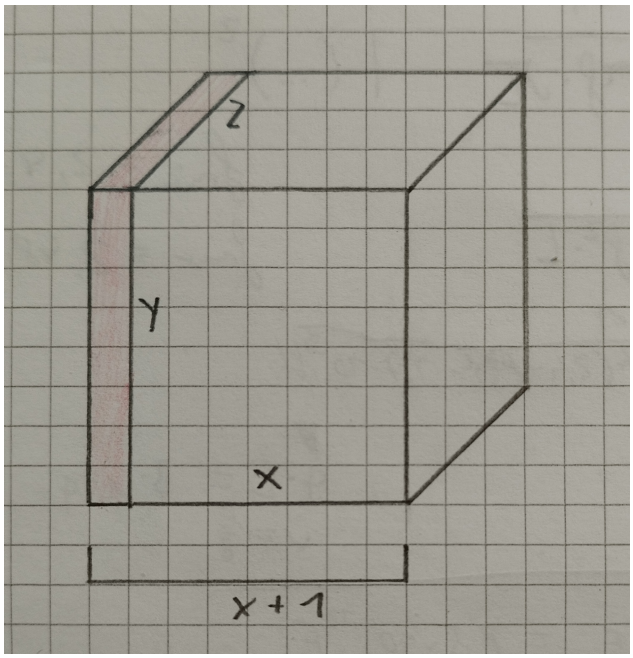

Abbildung 2: Finden einer fehlenden Scheibe, vermutete fehlende Scheibe in Rot, gesuchte Seite ist die Vorderseite

erweitert, für welche bei der normalen Scheibensuche nichts gefunden wurde. Die Suche nach fehlenden Scheiben findet in der Cheese::find_missing Methode statt.

Hier können auch fehlende Scheiben vermutet werden, die nicht tatsächlich fehlen. Deshalb werden in jedem Schritt bei den möglichen Pfaden mit derselben Startscheibe die Anzahl hinzugefügter Scheiben verglichen und nur der Pfad übergelassen, der die wenigsten hinzugefügten Startscheiben hat.

Da es hier auch sein kann, dass Käse mit der falschen Startscheibe begonnen werden und dann mit vielen fehlenden Scheiben trotzdem vervollständigt werden, werden nach einer bestimmten Schrittanzahl die top-Paths gefiltert wie die sub-Paths

```
if i == min_path_len / 2 && i > 4 && find_missing {
    // Wenn etwas Zeit vergangen ist, wird nur der Pfad mit den wenigsten
    // hinzugefügten Stücken weiterverfolgt
    top_paths = filter_top_paths(top_paths);
}
```

3 Laufzeitanalyse

Der Teil eines Schritts der am längsten dauert ist das kopieren der noch übrigbleibenden Scheiben. Die Dauer des Kopierens ist egal, ob nur die added-HashMap kopiert oder die beiden HashMaps (base und added) zusammengeführt werden linear von der Stückanzahl abhängig. Da die Anzahl der Schritte auch linear von der Stückanzahl abhängt ist die Gesamt-Laufzeitkomplexität $O(n^2)$.

4 Beispiele

Das Programm wird mit dem „$ cargo run –release“- Befehl ausgeführt. Man muss sich dabei im Ordner für diese Aufgabe befinden. Hier wird davon ausgegangen, dass Rust installiert ist. Die „–release“-Endung ermöglicht Optimisierungen die erlauben, dass das Programm in einer akzeptierbaren Zeit läuft. Alternativ kann auch die vorkompilierte Datei für die jeweilige Plattform verwendet werden, zum Beispiel „$ kaese-windows.exe“. Mögliche Argumente können mithilfe des „–help“ parameters angezeigt werden.

```
~/bwinf/kaese $ cargo run -- --help
Ein Programm, das Käsequader aus Scheiben wieder zusammensetzen kann.

Usage: kaese [OPTIONS] <FILES>...

Arguments:
  <FILES>...  Die Dateien, aus denen die Stücke geladen werden sollen

Options:
      --eat-prob <EAT_PROB>  Wahrscheinlichkeit, dass ein stück entfernt wird
      --find-missing         Wird automatisch gesetzt wenn prob gesetzt ist
  -h, --help                 Print help
  -V, --version              Print version
```

Da die Anzahl an Käsescheiben bei einigen Beispielen sehr groß ist gibt das Programm im Terminal nur den vollständigen Käse sowie die erste und letzte Scheibe aus. Die vollständige Liste der benutzten Scheiben wird in eine Datei geschrieben. Diese Dateien sind im Ordner für die Aufgabe unter dem Namen „out_n.txt“ auffindbar, wobei n die Nummer der Beispieleingabe ist.

a)

Das Programm nimmt den Dataipfad der Scheiben als Argument und setzt dann den Käse zusammen. Nach dem einlesen der Scheiben gibt das Programm einige Informationen über die Scheiben aus.

```
~/bwinf/kaese $ cargo run --release data/kaese2.txt
Lade Scheiben...
        5 Scheiben aus data/kaese2.txt gelesen

Mische Scheiben...

Informationen über die Käsescheiben:
        Maximale Anzahl eines einzelnen Stücks: 2
        Mehrfache Scheiben: 2
        Anzahl verschiener Scheiben: 3

1 Käse gefunden:
        Käse: [1000, 1000, 2]
        Startscheibe: Piece(999, 998)
        Letzte Scheibe: Piece(1000, 2)
        Scheibenreihenfolge in solution.txt gespeichert

Suche hat 38.9µs gedauert
```

Wie lange die Suche dauert hängt sowohl von der Scheibenanzahl als auch von der Hardware ab, auf der das Programm läuft.

```
~/bwinf/kaese $ cargo run --release data/kaese7.txt
Lade Scheiben...
        1529903 Scheiben aus data/kaese7.txt gelesen

Mische Scheiben...

Informationen über die Käsescheiben:
```

```
        Maximale Anzahl eines einzelnen Stücks: 14
        Mehrfache Scheiben: 511076
        Anzahl verschiener Scheiben: 1018827

1 Käse gefunden:
        Käse: [510510, 510510, 510510]
        Startscheibe: Piece(962, 665)
        Letzte Scheibe: Piece(510510, 510510)
        Scheibenreihenfolge in solution.txt gespeichert

Suche hat 2.696156624s gedauert
```

b)

4.0.1 Mehrere Käse

Um die Scheiben mehrer Käse zusammenzumischen können mehrere Dateien als Programmargument gegeben werden. Alternativ können die Scheiben auch manuell in einer Datei zusammengefügt werden. Zusammenmischen der Beispieleingaben (hier wurde hinterher von mir mit „//“ dazugeschrieben, aus welcher Beispieleingabe der Käse stammt):

```
~/bwinf/kaese $ cargo run --release data/kaese1.txt data/kaese2.txt data/kaese3.txt data/kaese4.txt d
Lade Scheiben...
        12 Scheiben aus data/kaese1.txt gelesen
        5 Scheiben aus data/kaese2.txt gelesen
        23 Scheiben aus data/kaese3.txt gelesen
        550 Scheiben aus data/kaese4.txt gelesen
        6848 Scheiben aus data/kaese5.txt gelesen
        90300 Scheiben aus data/kaese6.txt gelesen
        1529903 Scheiben aus data/kaese7.txt gelesen

Mische Scheiben...

Informationen über die Käsescheiben:
        Maximale Anzahl eines einzelnen Stücks: 14
        Mehrfache Scheiben: 543460
        Anzahl verschiener Scheiben: 1084181

7 Käse gefunden:
   0:
        Käse: [510510, 510510, 510510] //kaese7.txt
        Startscheibe: Piece(962, 665)
        Letzte Scheibe: Piece(510510, 510510)
        Scheibenreihenfolge in solution_0.txt gespeichert

   1:
        Käse: [510510, 39270, 30030] //kaese6.txt
        Startscheibe: Piece(480255, 9255)
        Letzte Scheibe: Piece(510510, 39270)
        Scheibenreihenfolge in solution_1.txt gespeichert

   2:
        Käse: [3570, 2730, 2310] //kaese5.txt
        Startscheibe: Piece(1325, 437)
        Letzte Scheibe: Piece(3570, 2730)
        Scheibenreihenfolge in solution_2.txt gespeichert

   3:
        Käse: [210, 210, 210] //kaese4.txt
```

```
        Startscheibe: Piece(51, 29)
        Letzte Scheibe: Piece(210, 210)
        Scheibenreihenfolge in solution_3.txt gespeichert

   4:
        Käse: [1000, 1000, 10] //kaese3.txt
        Startscheibe: Piece(995, 992)
        Letzte Scheibe: Piece(1000, 1000)
        Scheibenreihenfolge in solution_4.txt gespeichert

   5:
        Käse: [6, 6, 6] //kaese1.txt
        Startscheibe: Piece(4, 2)
        Letzte Scheibe: Piece(6, 6)
        Scheibenreihenfolge in solution_5.txt gespeichert

   6:
        Käse: [1000, 1000, 2] //kaese2.txt
        Startscheibe: Piece(999, 998)
        Letzte Scheibe: Piece(1000, 2)
        Scheibenreihenfolge in solution_6.txt gespeichert

Suche hat 3.026044521s gedauert
```

4.0.2 Gegessene Scheiben

Mithilfe der –eat-prob Flag kann angegeben werden, ob und mit welcher Wahrscheinlichkeit Scheiben entfernt werden sollen. Das Programm zeigt dann an wie viele entfernt wurden. Wenn diese Flag gesetzt ist wird auch automatisch nach fehlenden Scheiben gesucht. Alternativ kann dies auch mit –find-missing Flag aktiviert werden.

```
~/bwinf/kaese $ cargo run --release data/kaese5.txt --eat-prob 0.01
Lade Scheiben...
        6848 Scheiben aus data/kaese5.txt gelesen

Mische Scheiben...

47 Scheiben wurden gegessen

Informationen über die Käsescheiben:
        Maximale Anzahl eines einzelnen Stücks: 9
        Mehrfache Scheiben: 2226
        Anzahl verschiener Scheiben: 4575

1 Käse gefunden:
        Käse: [3570, 2730, 2310]
        47 Scheiben wurden hinzugefügt
        Startscheibe: Piece(1325, 437)
        Letzte Scheibe: Piece(3570, 2730)
        Scheibenreihenfolge in solution.txt gespeichert

Suche hat 12.434696ms gedauert
```

Wenn die Esswahrscheinlichkeit zu hoch gewählt wird kann es passieren, dass zwei Scheiben gegessen werden die hintereinander an den Käse angefügt werden müssen, weshalb die fehlenden Scheiben nicht gefunden werden können. Dadurch kann kein Käse gefunden werden.

```
{rust}
~/bwinf/kaese $ cargo run --release data/kaese5.txt --eat-prob 0.1
Lade Scheiben...
```

```
        6848 Scheiben aus data/kaese5.txt gelesen

Mische Scheiben...

644 Scheiben wurden gegessen

Informationen über die Käsescheiben:
        Maximale Anzahl eines einzelnen Stücks: 8
        Mehrfache Scheiben: 1926
        Anzahl verschiener Scheiben: 4278

Kein Käse gefunden
Suche hat 14.306967ms gedauert
```

Bei kleinen Käse kann das entfernen von Scheiben dazu führen, dass der Käse nicht mehr richtig erkannt wird.

```
~/bwinf/kaese $ cargo run --release data/kaese1.txt --eat-prob 0.1
Lade Scheiben...
        12 Scheiben aus data/kaese1.txt gelesen

Mische Scheiben...

1 Scheiben wurden gegessen

Informationen über die Käsescheiben:
        Maximale Anzahl eines einzelnen Stücks: 3
        Mehrfache Scheiben: 5
        Anzahl verschiener Scheiben: 6

3 Käse gefunden:
   0:
        Käse: [6, 6, 5]
        1 Scheiben wurden hinzugefügt
        Startscheibe: Piece(6, 4)
        Letzte Scheibe: Piece(6, 6)
        Scheibenreihenfolge in solution_0.txt gespeichert

   1:
        Käse: [5, 4, 2]
        1 Scheiben wurden hinzugefügt
        Startscheibe: Piece(4, 3)
        Letzte Scheibe: Piece(4, 2)
        Scheibenreihenfolge in solution_1.txt gespeichert

   2:
        Käse: [3, 3, 2]
        0 Scheiben wurden hinzugefügt
        Startscheibe: Piece(3, 3)
        Letzte Scheibe: Piece(3, 3)
        Scheibenreihenfolge in solution_2.txt gespeichert

Suche hat 144.4µs gedauert
```

Meistens klappt aber das zusammensetzten auch hier

```
~/bwinf/kaese $ cargo run --release data/kaese1.txt --eat-prob 0.1
Lade Scheiben...
        12 Scheiben aus data/kaese1.txt gelesen
```

```
Mische Scheiben...

1 Scheiben wurden gegessen

Informationen über die Käsescheiben:
        Maximale Anzahl eines einzelnen Stücks: 3
        Mehrfache Scheiben: 5
        Anzahl verschiener Scheiben: 6

1 Käse gefunden:
        Käse: [6, 6, 6]
        1 Scheiben wurden hinzugefügt
        Startscheibe: Piece(4, 2)
        Letzte Scheibe: Piece(6, 6)
        Scheibenreihenfolge in solution.txt gespeichert

Suche hat 72.399µs gedauert
```

Das Zusammensetzten dauert etwas länger, wenn nach gegessenen Scheiben gesucht wird, selbst wenn keine Scheiben entfernt wurden. Das liegt daran, dass manchmal fehlende Scheiben vermutet werden wo keine fehlen und dadurch Pfade erzeugt werden, die auch berechnet werden.

Tabelle 1: Suchzeit für verschidene Beispieleingaben. Zeit schwankt bei kleiner Scheibenanzahl stark weshalb nur die größeren Beispieleingaben verwendet wurden

Beispiel Nummer	4	5	6	7
Scheibenanzahl	550	6848	90300	1529903
mit Suche	1.809791ms	12.64834ms	198.674711ms	4.268454533s
ohne Suche	1.158492ms	7.46266ms	103.731686ms	2.628977384s

4.0.3 Beides

Es funktioniert generell auch mehrere Käse zusammenzumischen und einige Scheiben zu entfernen, es gibt aber leider auch manchmal Fehler. Vermutlich werden die Scheiben der Käse vertauscht und gegessene Scheiben falsch vermutet. Die Probleme treten auf, wenn man Scheiben der ersten Beispieleingabe mit entweder der zweiten oder dritten Beispieleingabe vermischt. Ansonsten klappt das vermischen und finden gegessener Scheiben.

```
~/bwinf/kaese $ cargo run --release data/kaese2.txt data/kaese3.txt data/kaese4.txt data/kaese5.txt d
Lade Scheiben...
        5 Scheiben aus data/kaese2.txt gelesen
        23 Scheiben aus data/kaese3.txt gelesen
        550 Scheiben aus data/kaese4.txt gelesen
        6848 Scheiben aus data/kaese5.txt gelesen
        90300 Scheiben aus data/kaese6.txt gelesen
        1529903 Scheiben aus data/kaese7.txt gelesen

Mische Scheiben...

156 Scheiben wurden gegessen

Informationen über die Käsescheiben:
        Maximale Anzahl eines einzelnen Stücks: 14
        Mehrfache Scheiben: 543358
        Anzahl verschiener Scheiben: 1084115

6 Käse gefunden:
   0:
        Käse: [510510, 510510, 510510]
```

```
        147 Scheiben wurden hinzugefügt
        Startscheibe: Piece(962, 665)
        Letzte Scheibe: Piece(510510, 510510)
        Scheibenreihenfolge in solution_0.txt gespeichert

    1:
        Käse: [510510, 39270, 30030]
        8 Scheiben wurden hinzugefügt
        Startscheibe: Piece(480255, 9255)
        Letzte Scheibe: Piece(510510, 39270)
        Scheibenreihenfolge in solution_1.txt gespeichert

    2:
        Käse: [3570, 2730, 2310]
        1 Scheiben wurden hinzugefügt
        Startscheibe: Piece(1325, 437)
        Letzte Scheibe: Piece(3570, 2730)
        Scheibenreihenfolge in solution_2.txt gespeichert

    3:
        Käse: [210, 210, 210]
        0 Scheiben wurden hinzugefügt
        Startscheibe: Piece(51, 29)
        Letzte Scheibe: Piece(210, 210)
        Scheibenreihenfolge in solution_3.txt gespeichert

    4:
        Käse: [1000, 1000, 10]
        0 Scheiben wurden hinzugefügt
        Startscheibe: Piece(995, 992)
        Letzte Scheibe: Piece(1000, 1000)
        Scheibenreihenfolge in solution_4.txt gespeichert

    5:
        Käse: [1000, 1000, 2]
        0 Scheiben wurden hinzugefügt
        Startscheibe: Piece(999, 998)
        Letzte Scheibe: Piece(1000, 2)
        Scheibenreihenfolge in solution_5.txt gespeichert

Suche hat 4.983174228s gedauert
```

5 Quellcode

Der Quellcode ist in mehrere Dateien gegliedert. „main.rs“ enthält Code zum Auslesen der Dateien sowie dem Verarbeiten der Programmargumente, ist also nicht weiter für die Lösung der Aufgabenstellung interessant und hier deshalb nicht mit eingezogen. Die anderen Dateien enthalten folgenden Code:

- cheese_builder.rs enthält die PossPath Klasse und Funktionen wie construct_cheeses, die mithilfe mehrerer Pfade die Käse finden.
- cheese.rs enthält die Definition für die Cheese und Piece Klassen, außerdem ist hier der Code für das finden von Seiten, die zu dem Käse passen
- pieces_map.rs Enthält die PiecesMap Klasse
- prev_pieces.rs Enthält Code zum Speichern der für einen Käse benutzten Scheiben.

Da der Quellcode ziemlich lang ist habe ich einige Funktionen entfernt. Der gesamte Quellcode kann im Ordner für die Aufgabe unter „src/“ gefunden werden.

5.1 cheese_builder.rs

```rust
//Ein möglicher Pfad
pub struct PossPath {
    cheese: Cheese, //Der mögliche Käse
    prev_pieces: PrevPieces,
    pieces_left: Box<PiecesMap>,
}

impl PossPath {
    ///erzeugt einen neuen möglichen Pfad
    pub fn new(cheese: Cheese, path: PrevPieces, pieces: Box<PiecesMap>) -> Self {
        Self {
            cheese,
            prev_pieces: path,
            pieces_left: pieces,
        }
    }
    ///erzeugt neue Pfade, indem es die neuen Seiten an den Käse anfügt
    fn gen_new_paths(self, find_missing: bool) -> Vec<PossPath> {
        self.cheese
            .gen_poss_paths(self.prev_pieces, self.pieces_left, find_missing)
    }
}

// Setzt nur einen Käse zusammen, wird immer wieder von construct_cheeses aufgerufen
fn construct_cheese(
    mut top_paths: Vec<Vec<PossPath>>, // Die Pfade, nach Startstück getrennt
    min_path_len: usize,               // Die minimale Länge eines Pfades
    find_missing: bool,                // Ob nach fehlenden Stücken gesucht werden soll
) -> Option<(Cheese, PrevPieces)> {
    let mut i = 0; // Die aktuelle Länge der Pfade
    while !top_paths.is_empty() {
        if i == min_path_len / 2 && i > 3 && find_missing {
            // Wenn etwas Zeit vergangen ist, wird nur der Pfad mit den wenigsten
            // hinzugefügten Stücken weiterverfolgt
            top_paths = filter_top_paths(top_paths);
        }
        let mut new_top_paths = vec![];
        for sub_paths in top_paths {
            let mut new_paths = vec![];
            // Der aktuelle Pfad wird gespeichert, sodass er als Ergebnis zurückgegeben werden kann,
            // wenn keine neuen Pfade gefunden werden
            let curr_result = {
                let PossPath {
                    cheese,
                    prev_pieces: path,
                    ..
                } = &sub_paths[0];
                (*cheese, path.clone())
            };
            //erzeugt neue Pfade
            for poss_path in sub_paths.into_iter() {
                let paths = poss_path.gen_new_paths(find_missing);
                new_paths.extend(paths);
            }
            // Entfernt Pfade mit mehr hinzugefügten Stücken
            if new_paths.len() > 1 {
                new_paths = filter_sub_paths(new_paths);
```

```
            }
            if new_paths.is_empty() {
                if i >= min_path_len {
                    return Some(curr_result);
                }
            } else {
                new_top_paths.push(new_paths);
            }
        }
        top_paths = new_top_paths;
        i += 1;
    }
    None
}
/// Findet alle möglichen Käse
pub fn construct_cheeses(
    pieces: Box<PiecesMap>,
    // Anzahl der Stücke, wird benötigt da gleiche Stücke
    // im Pieces-Objekt zusammengefasst werden
    n_pieces: usize,
    find_missing: bool,
) -> Vec<(Cheese, PrevPieces)> {
    let mut pieces_map = pieces;
    // Die gefundenen Käse
    let mut results = vec![];
    // Die Stücke, die bereits verwendet wurden
    // Wird verwendet um zu überprüfen ob alle Stücke verwendet wurden
    let mut used_pieces = vec![];
    let mut min_path_len = n_pieces * 3 / 4; // Die minimale Länge eines Pfades
                                             // Es wird davon ausgegangen, dass ein Käse mindestens
    while min_path_len > (n_pieces - used_pieces.len()) / 5 {
        // Die Stücke, die noch nicht verwendet wurden
        // werden als Startstücke verwendet
        let keys = pieces_map.base.keys().cloned().collect::<Vec<Piece>>();
        let top_paths = keys
            .iter()
            .map(|piece| {
                let cheese = Cheese::new([piece.0, piece.1, 0]);
                let start = PossPath::new(cheese, PrevPieces::new(*piece), pieces_map.clone());
                vec![start]
            })
            .collect::<Vec<_>>();
        // Es wird versucht einen Käse zu finden
        if let Some((cheese, path)) = construct_cheese(top_paths, min_path_len, find_missing) {
            // Wenn ein Käse gefunden wurde, werden die Stücke aus dem Pieces-Objekt entfernt
            let new_used_pieces = path.curr.get_real_pieces();
            pieces_map = Box::new(pieces_map.clone_without(&new_used_pieces));
            used_pieces.extend(new_used_pieces);

            // Die minimale Pfadlänge wird angepasst
            min_path_len = (n_pieces - used_pieces.len()) * 3 / 4;
            results.push((cheese, path));
            // found_cheese = true;
        } else {
            // Es wurde mit der der aktuellen Mindestlänge kein Käse gefunden,
            // Weshalb die Mindestlänge halbiert wird
            min_path_len /= 2;
        }
```

```
    }
    if !results.is_empty() {
        // Es wird überprüft ob alle Stücke verwendet wurden
        match used_pieces.len().cmp(&n_pieces) {
            Ordering::Less => {
                panic!(
                    "not all pieces used!! pieces left: {}",
                    pieces_map.base.values().sum::<u32>()
                );
            }
            Ordering::Greater => {
                panic!("error, used {} out of {}", used_pieces.len(), n_pieces);
            }
            Ordering::Equal => {}
        }
    }
    results
}
```

5.2 cheese.rs

```
pub struct NewSide {
    //neue Seite, die an das Käsestück angefügt wird
    pub side_n: usize,  //Seitennummer
    pub piece: Piece,   //Das Stück, das an die Seite angefügt wird
    pub is_added: bool, //wurde das Stück aufgegessen und ist hypothetisch?
}

impl NewSide {
    fn new(side_n: usize, piece: Piece, is_added: bool) -> Self {
        Self {
            side_n,
            piece,
            is_added,
        }
    }
}
#[derive(Debug, PartialEq, Eq, Hash, Clone, Copy)]
pub struct Piece(pub u32, pub u32);
impl TryFrom<Vec<&str>> for Piece {
    type Error = std::num::ParseIntError;
    fn try_from(value: Vec<&str>) -> Result<Self, Self::Error> {
        assert!(value.len() == 2);
        let width = value[0].parse()?;
        let height = value[1].parse()?;
        if width > height {
            // panic!();
            Ok(Self(width, height))
        } else {
            Ok(Self(height, width))
        }
    }
}

#[derive(Debug, PartialEq, Eq, Hash, Clone, Copy)]
pub struct Cheese {
    //a>=b>=c
```

```
    pub size: [u32; 3],
}

impl Cheese {
    ///erzeugt ein neues Käsestück
    pub fn new(size: [u32; 3]) -> Self {
        Self { size }
    }
    ///gibt die Seitenlängen zurück, die für die Erzeugung einer Seite verwendet werden
    pub fn get_sides_n() -> Vec<(usize, usize)> {
        vec![(1, 2), (0, 2), (0, 1)]
    }
    ///gibt die Seiten des Käsestücks zurück
    pub fn get_sides(&self) -> Vec<Piece> {
        Cheese::get_sides_n()
            .into_iter()
            .map(|(a, b)| Piece(self.size[a], self.size[b]))
            .collect()
    }
    ///fügt eine Scheibe zum Käse hinzu, indem eine Seite vergrößert wird
    fn expand_side(&self, n: usize) -> Cheese {
        let mut size = self.size;
        size[n] += 1;
        //sortiert die Seitenlängen, damit die Käsestücke eindeutig wiederfindbar sind
        //wie lange das dauert ist relativ egal, da das kopieren der Käsestücke eh
        //viel länger dauert
        size.sort_unstable_by_key(|size| std::cmp::Reverse(*size));
        Cheese { size }
    }
    ///findet fehlende Scheiben (siehe Dokumentation)
    fn find_missing(
        &self,
        updated_sides: Vec<bool>, //welche Seiten bereits vergrößert wurden
        pieces: &PiecesMap,       //welche Scheiben noch vorhanden sind
    ) -> Vec<NewSide> {
        //sucht in den verbleibenden Scheiben nach einer die zu einem vergrößerten Käsestück passt
        Cheese::get_sides_n() //welche Seitenlängen für vergrößert werden
            .into_iter()
            .enumerate() //seite, die in den Scheiben gesucht wird
            .flat_map(|(i_searched, (other_a, other_b))| {
                if updated_sides[other_a] && updated_sides[other_b] {
                    //falls beide Seiten bereits vergrößert wurden, kann es hier keine fehlende Sche
                    return vec![];
                }
                //Der Käse aus der Sicht der Scheibe die nach der fehlenden Scheibe kommt
                let len_x = self.size[other_a];
                let len_y = self.size[other_b];
                let len_z = self.size[i_searched];

                //möglicherweise fehlende Scheiben, sowie die gesuchte Scheibe danach
                let mut poss_missing = vec![];
                if !updated_sides[other_a] {
                    let missing = Piece(len_y, len_z);
                    let searched = Piece(len_x + 1, len_y);
                    poss_missing.push((other_a, missing, searched))
                }
                if !updated_sides[other_b] {
                    let missing = Piece(len_x, len_z);
```

```
                let searched = Piece(len_x, len_y + 1);
                poss_missing.push((other_b, missing, searched))
            }
            poss_missing
        })
        //gibt nur Scheiben zurück, die wirklich fehlen,
        //d.h. es wird auch die Scheibe gefunden, die danach kommt
        .filter_map(|(added_i, added, next)| {
            //sortiert Scheiben aus, für die es keine darauf folgende Scheibe gibt
            let n_pieces = pieces.get(&next)?;
            if n_pieces == &0 {
                return None;
            }
            //gibt die fehlende Scheibe als NewSide zurück
            //true gibt an, dass die Scheibe hinzugefügt ist
            Some(NewSide::new(added_i, added, true))
        })
        .collect::<Vec<_>>()
}
//findet neue Seiten, die an den Käse angefügt werden können
pub fn find_new_sides(&self, pieces: &PiecesMap) -> (Vec<bool>, Vec<NewSide>) {
    //Seiten des Käses die bereits gesehen wurden, um nicht doppelte Pfade zu erzeugen
    //falls zwei Seiten gleich sind, wird nur eine davon verwendet
    let mut sides_seen = vec![];
    //welche Seiten bereits vergrößert wurden
    //wird nur benötigt, wenn man bei der Suche nach fehlenden Scheiben auch sucht,
    //wenn Seiten gefunden wurden (also momentan nicht, villeicht aber in Zukunft)
    let mut updated_sides = vec![false; 3];
    //finde neue Seiten, die an den Käse angefügt werden können
    let new_sides = self
        .get_sides()
        .into_iter()
        .enumerate()
        .filter(|(_, side)| {
            //filtert gleiche Seiten des Käses aus, um doppelte Pfade zu vermeiden
            if sides_seen.iter().any(|other| other == side) {
                false
            } else {
                sides_seen.push(*side);
                true
            }
        })
        .filter_map(|(i, side)| {
            //überprüft, ob die Scheibe die zur Seite passt vorhanden ist
            let n_pieces = pieces.get(&side)?;
            if n_pieces == &0 {
                return None;
            }
            //markiert die Seite als vergrößert
            updated_sides[i] = true;
            //gibt die Seite zurück, die an den Käse angefügt werden kann
            //false gibt an, dass die Seite nicht vergrößert wurde
            Some(NewSide::new(i, side, false))
        })
        .collect::<Vec<_>>();
    (updated_sides, new_sides)
}
///erzeugt mögliche Pfade, indem es die neuen Seiten an den Käse anfügt
```

```rust
    pub fn new_sides_to_path(
        &self,
        new_sides: Vec<NewSide>, //neue Seiten, die an den Käse angefügt werden
        path: PrevPieces,        //der Pfad, der bis hierher gefolgt wurde
        pieces: Box<PiecesMap>,  //welche Scheiben noch vorhanden sind
    ) -> Vec<PossPath> {
        //wandle zuerst die überbleibenden Scheibenliste in einen Pointer um,
        //damit sie einmal weniger kopiert werden müssen (siehe Dokumentation)
        let pieces_ptr = Box::into_raw(pieces);
        let n_side = new_sides.len();
        new_sides
            .into_iter()
            .enumerate()
            .map(move |(i, new_side)| {
                //wenn es die letzte neue Seite ist, verwende die Scheibenliste,
                //ohne sie zu kopieren
                let mut pieces = if i == n_side - 1 {
                    unsafe { Box::from_raw(pieces_ptr) }
                } else {
                    //ansonsten kopiere die Liste
                    Box::new(unsafe { pieces_ptr.as_mut() }.unwrap().make_copy())
                };
                //erzeuge neuen Pfad und entferne die Scheibe aus der Liste
                let new_path = if new_side.is_added {
                    //Scheibe wurde hinzugefügt (wurde aufgegessen)
                    path.extend_added(new_side.piece) //füge sie dem Pfad hinzu
                } else {
                    //Scheibe ist echt
                    //entferne die Scheibe aus der Liste
                    let mut n_pieces = *pieces.get(&new_side.piece).unwrap();
                    n_pieces -= 1;
                    pieces.insert(new_side.piece, n_pieces);

                    path.extend_real(new_side.piece) //füge die Scheibe dem Pfad hinzu
                };
                //erzeuge neuen Käse
                let new_cheese = self.expand_side(new_side.side_n);
                //gebe neuen Pfad zurück
                PossPath::new(new_cheese, new_path, pieces)
            })
            .collect()
    }
    ///erzeugt mögliche neue Pfade
    pub fn gen_poss_paths(
        &self,
        path: PrevPieces,
        pieces: Box<PiecesMap>,
        find_missing: bool,
    ) -> Vec<PossPath> {
        //findet neue Seiten, die an den Käse angefügt werden können
        let (updated_sides, mut new_sides) = self.find_new_sides(&pieces);
        // sucht nach fehlenden Scheiben, falls keine neuen Seiten gefunden wurden und
        // es möglicherweise fehlende Scheiben gibt
        if new_sides.is_empty() && find_missing {
            new_sides = self.find_missing(updated_sides, &pieces);
        }
        if !new_sides.is_empty() {
            //erzeugt mögliche Pfade, indem es Scheiben an die Seiten des Käses anfügt
```

```
            self.new_sides_to_path(new_sides, path, pieces)
        } else {
            vec![]
        }
    }
}

//Nimmt die Scheibe als Startscheibe
//kann durch die into() Funktion genutzt werden
impl From<Piece> for Cheese {
    fn from(value: Piece) -> Self {
        Cheese::new([value.0, value.1, 1])
    }
}
```

5.3 pieces_map.rs

```
//Speichert die Käsescheiben, die noch über sind
#[derive(Debug, Clone)]
pub struct PiecesMap {
    //base-HashMap bleibt unverändert,
    //wird nicht geklont und zwischen verschiedenen Instanzen geteilt
    pub base: Rc<FxHashMap<Piece, u32>>,
    //base_id wird benutzt, um schnell zu überprüfen,
    //ob zwei Instanzen die selbe base-HashMap verwenden
    pub base_id: Uuid,
    //wird geklont und nicht zwischen verschiedenen Instanzen geteilt
    //added-HashMap wird bei Bedarf mit base-HashMap zusammengeführt
    pub added: FxHashMap<Piece, u32>,
}

impl PiecesMap {
    //erzeugt eine PiecesMap aus einer Liste von Käsescheiben
    pub fn new(pieces: &Vec<Piece>) -> PiecesMap {
        //die base-HashMap,
        //benuzt FxHashMap, da diese schneller ist als die Standard-HashMap
        let mut pieces_map: FxHashMap<Piece, u32> = FxHashMap::default();
        //die größte Anzahl derselben Käsescheibe
        let mut max_n = 0;
        //wie viele Käsescheiben mehrfach vorkommen
        let mut n_multiple = 0;
        for piece in pieces {
            //überprüft, ob die Käsescheibe schon in der HashMap ist
            if let Some(n) = pieces_map.get_mut(piece) {
                //wenn ja, wird die Anzahl der Scheibe in der HashMap um 1 erhöht
                *n += 1;
                n_multiple += 1;
                if n > &mut max_n {
                    max_n = *n;
                }
            } else {
                //wenn nicht, wird die Käsescheibe neu in die HashMap eingefügt
                pieces_map.insert(*piece, 1);
            }
        }
        //gebe Informationen über die Scheiben aus
        println!("Informationen über die Käsescheiben:");
        println!(
```

```
            "\tMaximale Anzahl eines einzelnen Stücks: {}\n\tMehrfache Scheiben: {}\n\tAnzahl verschi
            max_n,
            n_multiple,
            pieces_map.len()
        );
        println!();
        PiecesMap::new_from_map(pieces_map)
    }

    //erzeugt eine neue Instanz aus einer base-HashMap
    pub fn new_from_map(map: FxHashMap<Piece, u32>) -> Self {
        let base_id = Uuid::new_v4();
        Self {
            base: Rc::new(map),
            base_id,
            added: FxHashMap::default(),
        }
    }
    //gibt falls vorhanden die Anzahl der Käsescheiben zurück, die für k gefunden wurden
    pub fn get(&self, k: &Piece) -> Option<&u32> {
        if let Some(result) = self.added.get(k) {
            Some(result)
        } else {
            self.base.get(k)
        }
    }
    //fügt eine Käsescheibe hinzu oder verändert die Anzahl der Käsescheibe
    //es wird nur die added-HashMap verändert, um Zeit beim Klonen zu sparen
    pub fn insert(&mut self, k: Piece, v: u32) -> Option<u32> {
        self.added.insert(k, v)
    }
    //kombiniert die added-HashMap mit der base-HashMap zu einer neuen base-HashMap
    fn merge_hashmaps(&self) -> FxHashMap<Piece, u32> {
        //die neue base-HashMap
        let mut new_map = FxHashMap::with_capacity_and_hasher(
            self.base.len(),
            BuildHasherDefault::<FxHasher>::default(),
        );
        //fügt alle Käsescheiben aus den beiden HashMaps zusammen
        for (k, v) in self.base.as_ref() {
            let v = if let Some(v) = self.added.get(k) {
                //falls ein Eintrag in der added-HashMap vorhanden ist, wird dieser verwendet
                *v
            } else {
                //ansonsten wird der Eintrag aus der base-HashMap verwendet
                *v
            };
            if v == 0 {
                //falls von einer Käsescheibe keine mehr vorhanden sind, wird sie übersprungen
                continue;
            }
            new_map.insert(*k, v);
        }
        new_map
    }
    //erzeugt eine neue Instanz,
    //in der die added-HashMap mit der base-HashMap zusammenführt wurde
    fn merge(&mut self) {
```

```
        let base = self.merge_hashmaps();
        self.base = Rc::new(base);
        self.added = FxHashMap::default();
    }
    //erzeugt eine neue Instanz, meistens einfach als Kopie
    //wenn die added-HashMap zu groß ist, wird sie mit der base-HashMap zusammengeführt
    pub fn make_copy(&mut self) -> Self {
        if self.added.len() > self.base.len() / 10 {
            //10% hat sich beim Ausprobieren als gute Größe herausgestellt
            self.merge();
        }
        self.clone()
    }
    //erzeugt eine neue Instanz ohne die Käsescheiben in removed
    //wird verwendet, um die Käsescheiben zu entfernen, die bereits verwendet wurden
    pub fn clone_without(&self, removed: &Vec<Piece>) -> Self {
        if removed.is_empty() {
            return self.clone();
        }
        //die neue base-HashMap, noch mit allen Käsescheiben
        let mut new_map = self.merge_hashmaps();
        //entfernt iterativ die Käsescheiben aus der neuen base-HashMap
        for piece in removed {
            //ob der Eintrag für die Käsescheibe entfernt werden soll
            let mut do_remove = false;
            if let Some(v) = new_map.get_mut(piece) {
                //entfernt eine Käsescheibe
                *v -= 1;
                if *v == 0 {
                    //wenn keine Käsescheibe mehr vorhanden ist, wird sie entfernt
                    do_remove = true;
                }
            }
            if do_remove {
                //entfernt den Eintrag für die Käsescheibe
                new_map.remove(piece);
            }
        }
        Self::new_from_map(new_map)
    }
}
```

5.4 prev_pieces.rs

```
//Eine Art Linked-List für zu einem Käse hinzugefügte Scheiben
//Hier wird die Liste nicht als Liste implementiert, sondern als Baum
//Das heißt, dass mehrere Path-Instanzen auf Teile der selben Scheiben-Liste zeigen können
//(siehe Dokumentation)
#[derive(Clone)]
pub struct PrevPieces {
    pub curr: Rc<HistPoint>, //die Head-Node der Liste
    pub start_piece: Piece,  //die erste Scheibe die zum Käse hinzugefügt wurde
    pub len: usize,          //die Anzahl der Scheiben, die zum Käse hinzugefügt wurden (nur echte)
    //die Anzahl der hypothetischen fehlenden Scheiben,
    //die zum Käse hinzugefügt wurden
    pub n_added: u32,
}
```

```rust
impl PrevPieces {
    //erzeugt eine neue Instanz
    pub fn new(value: Piece) -> Self {
        Self {
            curr: Rc::new(HistPoint::new(value)),
            start_piece: value,
            len: 1,
            n_added: 0,
        }
    }
}
//ein Knoten der Liste
#[derive(Debug, Clone)]
pub struct HistPoint {
    //der Vorgänger-Knoten wird als Rc gespeichert,
    //da mehrere Instanzen auf den selben Vorgänger-Knoten zeigen können
    prev: Option<Rc<HistPoint>>,
    //die Käsescheibe, die zum Käse hinzugefügt wurde
    value: Piece,
    //wenn true, handelt es sich um eine hypothetische Scheibe
    is_added: bool,
}

impl HistPoint {
    //erzeugt eine neue Instanz
    fn new(value: Piece) -> Self {
        Self {
            prev: None,
            value,
            is_added: false,
        }
    }
    //erzeugt eine neue Instanz, die auf den Vorgänger-Knoten zeigt
    fn make_next(self: &Rc<Self>, value: Piece, is_added: bool) -> Rc<Self> {
        Rc::new(Self {
            prev: Some(self.clone()),
            value,
            is_added,
        })
    }
    ///gibt die Liste als Array von Scheiben zurück,
    /// egal ob sie echte oder hypothetische Scheiben sind
    pub fn get_pieces(self: &Rc<HistPoint>) -> Vec<Piece> {
        self.to_array().iter().map(|pt| pt.value).collect()
    }
}
impl Drop for PrevPieces {
    //wird aufgerufen, wenn eine Instanz von Path gelöscht wird
    //ansonsten würde die Liste rekursiv gelöscht werden
    //was bei einer großen Länge zu einem (Call) Stack Overflow führen kann
    fn drop(&mut self) {
        /* gekürzt... */
    }
}
```