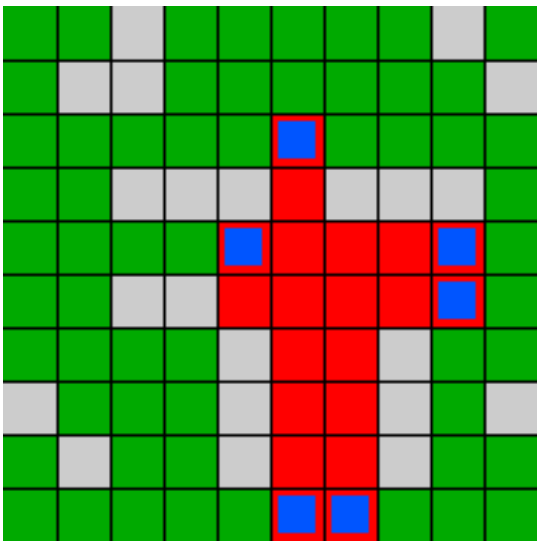
Dokumentation zur Aufgabe 1 - Buschfeuer

Dominic S. Meiser



Inhaltsverzeichnis

1 Lösungsidee - Teilaufgabe 1 **4**
1.1 Bruteforce . . . 4
1.1.1 TeilrasterBruteforce . . . 4
1.2 TreeBuilding . . . 5
1.3 Rasterfunktionen . . . 5
1.3.1 Ist das Raster gelöscht? . . . 6
1.3.2 Das Feuer weiterverbreiten . . . 6

2 Lösungsidee - Teilaufgabe 2 **6**

3 Umsetzung - Teilaufgabe 1 **7**
3.1 Raster . . . 7
3.2 RasterUtils . . . 8
3.2.1 Methode `istGeloescht()` . . . 8
3.2.2 Methode `burn()` . . . 9
3.3 BruteforceFeuerlöscher . . . 9
3.3.1 Laufzeit . . . 10
3.4 TeilrasterBruteforceFeuerlöscher . . . 10
3.4.1 Laufzeit . . . 11
3.5 TreeBuildingFeuerlöscher . . . 11
3.5.1 Laufzeit . . . 12

4 Umsetzung - Teilaufgabe 2 **13**

5 Beispiele - Teilaufgabe 1 **14**
5.1 BruteforceFeuerlöscher . . . 14
5.1.1 Beispiel 1 . . . 14
5.1.2 Beispiel 2 . . . 15
5.1.3 Beispiel 3 . . . 16
5.2 TeilrasterBruteforceFeuerlöscher . . . 17
5.2.1 Beispiel 4 . . . 18
5.3 TreeBuildingFeuerlöscher . . . 19
5.3.1 Beispiel 3 . . . 20
5.3.2 Beispiel 5 . . . 22

6 Quelltext - Teilaufgabe 1 **26**
6.1 Die Klasse `Raster` . . . 26
6.2 Die Klasse `RasterUtils` . . . 27
6.3 Die Klasse `BruteforceFeuerloescher` . . . 28
6.4 Die Klasse `TeilrasterBruteforceFeuerloescher` . . . 30

6.5 Die Klasse `TreeBuildingFeuerloescher` 33

1 Lösungsidee - Teilaufgabe 1

Bei dieser Aufgabe geht es darum, einen Löschweg für ein gegebenes Raster zu finden, bei dem möglichst viel Wald vor dem Feuer gerettet wird. Deshalb bietet es sich an, eine Lösung nach der Anzahl der geretteten Waldstücke zu bewerten. Ansonsten habe ich 2 verschiedene Algorithmen gefunden, die dies ermöglichen. Diese sind weiter unten beschrieben. Erstmal möchte ich noch auf das Raster eingehen. Es ist hier sinnvoll das Raster als zweidimensionales Integer-Array zu implementieren. Außerdem, um eine Überfüllung von Konstanten zu vermeiden, sollten gelöschte Felder einen dazu addierten Wert bekommen, anstatt das für jeden gelöschten Fall neue Konstanten erstellt werden. Bei dieser Teilaufgabe ist dies zwar noch nicht von großer Bedeutung, für die nächste Teilaufgabe ist dies jedoch von Vorteil. Außerdem braucht es noch ein paar Funktionen eines Rasters. Diese habe ich im Unterpunkt Rasterfunktionen beschrieben.

1.1 Bruteforce

Als einfachen und immer perfekten Algorithmus habe ich Bruteforce implementiert. Hierbei bietet es sich an, diesen rekursiv zu gestalten. Dieser sollte an eine Liste von Punkten solange neue Punkte dranhängen, bis das entstandene Raster entweder weniger noch nicht brennende Waldstücke enthält als die bis jetzt beste gefundene Lösung oder bis das Raster vollständig gelöscht ist. Danach muss der letzte Punkt wieder entfernt werden und weiter gesucht werden.

Sobald eine Lösung gefunden wurde, müssen die noch nicht brennenden Waldstücke dieser Lösung herausgefunden werden. Diese sollten gespeichert werden. Wenn ein Raster diese Anzahl unterschreitet, kann diese Lösung abgebrochen werden. Dies kann die Laufzeit des Bruteforce sehr beschleunigen. Wie die meisten Bruteforce-Algorithmen ist auch meiner sehr langsam, kommt aber mit relativ wenig Arbeitsspeicher aus. Die Laufzeit und der Speicherverbrauch ist natürlich von der Rastergröße, der Anzahl der Brandherde und -schneisen abhängig.

1.1.1 TeilrasterBruteforce

Da der Bruteforce bei großen Rastern sehr langsam ist und es bei einigen Rastern reicht, wenn man nur einen kleinen Teil das Rasters anstatt des kompletten Rasters betrachtet, habe ich mir eine kleine Veränderung des Bruteforces überlegt. Es geht hier darum, dass ein kleineres Raster gebildet wird, welches alle Elemente auf dem Raster enthält, aber kleiner ist als das eingegebene Raster. Wenn der Bruteforce-Algorithmus für dieses kleinere Raster eine Lösung findet, bei der sich kein brennendes Waldstück am Rand des Rasters befindet, so kann diese Lösung verwendet werden. Ansonsten muss das Raster vergrößert werden.

1.2 TreeBuilding

Mir ist aufgefallen, dass Bruteforce etwa so Lösungen findet:

```
18: [[0, 3], [0, 6], [1, 6], [2, 6], [3, 6], [4, 3], [4, 2], [3, 7], [3, 1], [2, 8], [2, 0], [1, 9]]
20: [[0, 3], [0, 6], [1, 6], [2, 6], [3, 6], [4, 3], [4, 7], [3, 2], [3, 1], [3, 8], [2, 0], [2, 9]]
23: [[0, 3], [0, 6], [1, 6], [2, 6], [5, 3], [3, 7], [4, 2], [4, 1], [3, 8], [3, 0], [2, 9]]
24: [[0, 3], [0, 6], [1, 6], [2, 6], [5, 3], [3, 7], [4, 2], [4, 8], [3, 1], [3, 0], [3, 9]]
26: [[0, 3], [0, 6], [1, 6], [6, 3], [2, 7], [5, 2], [3, 8], [4, 1], [4, 0], [3, 9]]
27: [[0, 3], [0, 6], [7, 3], [1, 7], [6, 2], [2, 8], [5, 1], [3, 9], [4, 0]]
28: [[0, 3], [8, 3], [0, 7], [7, 2], [1, 8], [6, 1], [2, 9], [5, 0], null]
29: [[0, 3], [8, 3], [8, 2], [0, 8], [7, 1], [1, 9], [6, 0], [3, 9], null]
30: [[0, 3], [8, 3], [8, 2], [8, 1], [0, 9], [7, 0], [2, 9], [3, 9], null]
31: [[0, 3], [8, 3], [8, 2], [8, 1], [8, 0], [1, 9], [2, 9], [3, 9], null]
32: [[0, 5], [0, 2], [7, 5], [7, 6], [7, 7], [2, 0], [6, 8], [6, 9]]
33: [[0, 5], [0, 2], [7, 5], [7, 6], [7, 7], [7, 8], [3, 0], [6, 9]]
34: [[0, 5], [0, 2], [7, 5], [7, 6], [7, 7], [7, 8], [7, 9], null]
35: [[0, 5], [8, 5], [0, 1], [7, 6], [7, 7], [7, 8], [7, 9], null]
36: [[0, 5], [8, 5], [8, 6], [0, 0], [7, 7], [7, 8], [7, 9], null]
37: [[0, 5], [8, 5], [8, 6], [8, 7], [1, 0], [7, 8], [7, 9], null]
38: [[0, 5], [8, 5], [8, 6], [8, 7], [8, 8], [2, 0], [7, 9], null]
39: [[0, 5], [8, 5], [8, 6], [8, 7], [8, 8], [8, 9], [3, 0], null]
```

Die beste Lösung ist fast immer die mit dem kürzesten Löschweg. Deswegen habe ich mir überlegt, dass man das Feuer so löschen sollte, dass zuerst die kürzeste Lösung gefunden wird. Dies ist möglich, indem man einen "Baum" aufbaut, der alle möglichen Wege enthält, die den Löschvorgang beschreiben. Diese werden immer weiter verlängert. Sobald eine Lösung gefunden wurde, können alle Wege, die weniger Waldstücke "überleben" lassen, aus dem Baum entfernt werden. Sobald der Baum leer ist, kann ich davon ausgehen, dass die beste Lösung gefunden wurde. Passiert dies nicht, aber die Anzahl der Schritte im Baum wird größer als die Anzahl der existierenden Felder im Raster, kann ich ebenfalls aufhören weiter zu suchen. Dies ist deutlich schneller als der Bruteforce-Algorithmus, obwohl es sich hierbei um eine veränderte breadth-first-Suche handelt, verbraucht aber dafür verdammt viel Arbeitsspeicher. Deshalb kann es nötig sein, einige Wege, die nur sehr wenig Wald retten können, auszuschließen, obwohl es keine bessere Lösung, aber dafür einen besseren Weg gibt. Dies sollte so passieren, dass die Anzahl der Waldflächen im idealsten Weg gespeichert werden und alle Wege, die eine bestimmte Anzahl an Waldstücken unter diesem Wert liegen, ignoriert werden. Dadurch kann aber nicht garantiert werden, dass die beste Lösung gefunden wird.

1.3 Rasterfunktionen

Hier sind einige Ideen für Rasterfunktionen aufgeführt:

1.3.1 Ist das Raster gelöscht?

Eine wichtige Funktion für Raster ist die Frage: Ist dieses Raster gelöscht? Hierfür ist es am besten, jede einzelne Zelle im Raster zu durchsuchen. Sollte diese brennen und eine der 4 umliegenden Felder (insofern es 4 gibt - könnte ja auch sein dass die Zelle am Rand des Rasters liegt) ungelöschter und nicht brennender Wald sein, so ist das Raster noch nicht gelöscht. Andernfalls ist das Feuer eingegrenzt und kann sich nicht weiterverbreiten - also ist das Raster gelöscht.

1.3.2 Das Feuer weiterverbreiten

Weiterhin ist es wichtig, das Raster weiter brennen zu lassen. Hierfür muss man auch jede mögliche Zelle im Raster durchgehen. Wenn diese brennen sollte, schaut man in der Umgebung (d.h. in dem 4 umliegenden Feldern, insofern diese existieren) nach nicht brennenden und nicht gelöschten Waldfeldern. Jedes Waldfeld in der Umgebung, das existiert, wird als brennend markiert. So breitet sich das Feuer in alle 4 Richtungen aus.

2 Lösungsidee - Teilaufgabe 2

Diese Teilaufgabe beschäftigt sich damit, ob es sinnvoll sein kann, auch Waldflächen zu löschen. Da es nicht gefordert war, habe ich mich entschlossen, hierfür keinen Feuerlöscher zu implementieren. Stattdessen kam es mir sinnvoller vor, per Hand eine Lösung zu erstellen, bei der es sich lohnt, ein Waldstück zu löschen. Da ich auch einen Fall gefunden habe, bei dem dies sinnvoll wäre, komme ich zu dem Schluss: Ja, Scheila hat Recht, es kann in einigen Fällen sinnvoll sein.

3 Umsetzung - Teilaufgabe 1

Ich habe die Lösungsidee in Java umgesetzt. Am Anfang wird der Benutzer aufgefordert, den Feuerlöscher und die Rastergröße einzugeben. Dabei werden alle Feuerlöscher im Paket `buschfeuer.feuerloescher` automatisch erkannt. Dies geschieht wie folgt:

```
String[] feuerloescher_files = new File("buschfeuer"+File.separator+"feuerloescher").list(
        (File dir, String name) ->
        ((new File(dir, name).isFile()) && (name.indexOf('$') == -1) && name.endsWith(".class")) );
```

Hierbei werden also alle Ordner, Unterklassen und Lambdas (die bei Java ja ein '$' im Dateinamen haben) und nicht-Klassen-Dateien ausgeschlossen. Dadurch werden genau alle Klassen, die ein Feuerlöscher sind, gefunden.

Anschließend hat der Benutzer die Wahl, ein gespeichertes Raster zu benutzen oder ein Raster neu zu erstellen. Ein eingegebenes Raster wird automatisch in `<tmp>/buschfeuer/raster.bfr` gespeichert. Aufgrund der Tatsache, dass ein Feuerlöscher sehr lange brauchen kann, legt jeder Feuerlöscher im selben Verzeichnis auch noch ein Raster ab, dass den erstellten Löschweg speichert. Wenn man ein solches Raster öffnet, wird man gefragt, ob eine neue Lösung entstehen soll oder ob die gespeicherte Lösung benutzt werden soll. Wenn man kein vorgegebenes Raster benutzen möchte, kann man beim Startdialog auswählen, wie groß das Raster sein soll. Man sollte hierbei jedoch beachten, dass meine GUI nicht in der Lage ist, Raster, die größer als 38x38 sind, innerhalb des Bildschirms darzustellen. Deshalb habe ich mich entschieden die maximale Rastergröße auf 38x38 zu setzen. Die minimale Rastergröße ist meiner Meinung nach 3x3, da man bei einem kleineren Raster keinen Feuerlöscher benötigt, um das Raster zu löschen. Beim anschließend sich öffnenden Fenster kann man zuerst durch Mausklicks auf das entsprechende Feld im Raster Brandschneisen auswählen; beim Klick auf "Weiter" kann man die Brandflächen auswählen. Anschließend kann man mithilfe des Buttons "Simulation starten" mit diesem Raster und dem vorher ausgewählten Feuerlöscher die Simulation starten.

Sobald die Simulation gestartet wird, wird die `init`-Methode des Feuerlöschers aufgerufen. In dieser kann der Feuerlöscher, falls er dies möchte, ein Ergebnis für das übergebene Raster berechnen. Ist der Feuerlöscher damit fertig, wird der Brand simuliert. Zuwider der Aufgabenstellung habe ich mich entschlossen, keine Stunde zwischen den Löschschritten zu warten. Dies finde ich überflüssig, da niemand daran interessiert ist, mehrere Stunden vorm PC zu sitzen und zu hoffen, dass sich der Simulator dazu bequemt, den nächsten Schritt anzuzeigen. Ich habe diese Wartezeit auf eine Sekunde herunter gesetzt.

Im folgenden werden ein paar relevante Klassen genau beschrieben. Die restlichen Klassen (wie etwa die Klasse `RasterPane`, welche zuständig ist um ein Raster graphisch darzustellen) sind nicht für die Umsetzung der Lösungsidee relevant, weshalb ich diese hier nicht beschreiben werde.

3.1 Raster

Diese Klasse repräsentiert ein Raster als zweidimensionales Integer-Array. Weiterhin definiert diese Klasse die Konstanten für die Rasterwerte:

```
public static final int WALD=1, SCHNEISE=2, BRAND=3, GELOESCHT=4;
```

Außerdem enthält diese Klasse eine Methode namens `count`, welche die Anzahl der Vorkommnisse eines Wertes im Raster herausfindet. Sie durchsucht dafür das komplette Raster und inkrementiert den Counter sobald der entsprechende Wert gefunden wurde:

```
int count = 0;
for (int i = 0; i < raster.length; i++)
      for (int j = 0; j < raster[i].length; j++)
            if (get(i,j) == value)
                  count++;
```

Diese Klasse setzt also diesen kleinen Teil der Lösungsidee um.

3.2 RasterUtils

Diese Klasse definiert die Methoden aus dem 3. Teil der Lösungsidee. Dazu gehören die folgenden beiden Methoden. Außerdem sind in dieser Klasse Methoden zum Speichern und Öffnen von Rastern enthalten. Weiterhin gibt es noch eine Methode um ein Raster zu klonen.

3.2.1 Methode `istGeloescht()`

Diese Methode prüft, wie in der Lösungsidee beschrieben, das Raster darauf, ob sich der Brand noch weiterverbreiten kann. Dazu wird durch jedes Rasterfeld iteriert und geschaut, ob dieses 1.) brennt und 2.) von Wald umgeben ist. Existiert kein solches Feld, so ist das Raster gelöscht. Ein Rasterfeld wird dabei so überprüft:

```
if (raster.get(i, j) == BRAND)
{
      if (i > 0)
            if (raster.get(i-1, j) == WALD)
                  return false;
      if (j > 0)
            if (raster.get(i, j-1) == WALD)
                  return false;
      if (i < raster.getSize().width-1)
            if (raster.get(i+1, j) == WALD)
                  return false;
      if (j < raster.getSize().height-1)
            if (raster.get(i, j+1) == WALD)
                  return false;
}
```

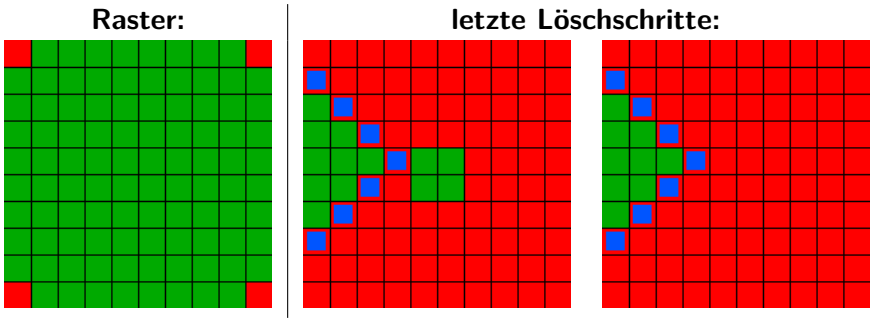
3.2.2 Methode `burn()`

Diese Methode lässt ein Raster weiter brennen. Auch diesen Vorgang habe ich in der Lösungsidee beschrieben. Ich gehe hier ähnlich wie in der `istGeloescht`-Methode vor; ich gehe hierfür auch jedes Rasterfeld durch und schaue, ob es von Wald umgeben ist. Jedes solche Waldfeld wird entzündet. Bevor ich damit beginnen kann, muss ich das Raster noch klonen. Ansonsten würde bei dieser Weiterentzündung in Brand geratene Waldstücke weitere Waldstücke entzünden. Das Weiterentzünden geschieht so:

```
if (bevore.get(i, j) == BRAND)
{
      if (i > 0)
            if (bevore.get(i-1, j) == WALD)
                  raster.set(i-1, j, BRAND);
      if (j > 0)
            if (bevore.get(i, j-1) == WALD)
                  raster.set(i, j-1, BRAND);
      if (i < bevore.getSize().width-1)
            if (bevore.get(i+1, j) == WALD)
                  raster.set(i+1, j, BRAND);
      if (j < bevore.getSize().height-1)
            if (bevore.get(i, j+1) == WALD)
                  raster.set(i, j+1, BRAND);
}
```

3.3 BruteforceFeuerlöscher

Dieser Feuerlöscher implementiert einen normalen Bruteforce-Algorithmus, wie in der Lösungsidee beschrieben. Dabei handelt es sich im Prinzip um breadth-first, wobei ich abbreche, sobald das Raster die Anzahl der Waldstücke der bisher besten Lösung unterschreitet. In der `init`-Methode rufe ich die rekursive Methode namens `test` auf, welche den Algorithmus enthält. Diese schaut zuerst, ob das Raster gelöscht ist. Ist dies so und die Anzahl der Waldflächen im Raster größer als die bisher beste Lösung (welche am Anfang 0 ist), so wird diese Lösung zurückgegeben und die Anzahl der Waldflächen als beste bisher gefundene Möglichkeit gespeichert. Ansonsten schaut die Methode nach löschbaren Brandfeldern; worunter ich in dem Fall ein Brandstück verstehe, welches an mindestens einer der vier Seiten von einem Waldstück umgeben wird. Jedes davon wird an die Liste, die als Parameter übergeben wird, angehängt, und anschließend findet ein rekursiver Aufruf statt. Existiert kein solches Brandstück, der Brand ist aber dennoch noch nicht gelöscht, wird `null` an die Liste angehängt (was bedeutet, dass kein Brandstück gelöscht werden soll). Daraufhin findet ein weiterer rekursiver Aufruf statt. Dieser Fall kann etwa im folgenden Raster auftreten:



In diesem Raster gibt es beim letzten Löschschritt kein sinnvolles Brandfeld mehr, dass der Feuerlöscher löschen könnte. Es wird also nichts mehr gelöscht.

3.3.1 Laufzeit

Dieser Feuerlöscher braucht kaum Arbeitsspeicher. Im Prinzip wird nur die aktuelle Lösung, eine `LinkedList<Point>` und die Liste mit den aktuell hinzugefügten Punkten, die als Parameter übergeben wird, ebenfalls eine `LinkedList<Point>`. Außerdem wird natürlich das Raster gespeichert, welches eine eigene Klasse ist; diese enthält jedoch eigentlich nur ein `int[][]`. Somit ist der Speicherverbrauch relativ konstant. Leichte Änderungen ergeben sich jedoch daraus, dass die rekursive `test`-Methode alle sinnvollen Brandfelder speichert. Da diese Methode ja rekursiv arbeitet, können schon so 20 Listen im Arbeitsspeicher rumgammeln. Dies erhöht den Arbeitsspeicherverbrauch noch etwas. Bei einem 15x15-Raster beträgt der Speicherverbrauch etwa 130 MiB. Gestartet hat der Algorithmus (d.h. kurz nach dem Starten des Löschvorgangs) mit etwas mehr als 110 MiB. Das Raster habe ich selbstverständlich im Beispiel-Ordner beigelegt. Da der Java Heap Space ja nicht immer genau um so viele Bytes wächst wie das Objekt aktuell benötigt, wird der Speicherverbrauch nur sehr wenig gestiegen sein. Dies habe ich oben bereits erklärt.

Die Laufzeit eines Bruteforce-Algorithmus steigt proportional zur Anzahl der Möglichkeiten. So steht es auf Wikipedia. Theoretisch stimmt das auch für meinen Feuerlöscher, wenn die beste Lösung die letzte ist. Ansonsten kann es passieren, dass der Feuerlöscher vorher bereits eine bessere Lösung gefunden hat und manche Möglichkeiten ignoriert. Außerdem ist die Rechenzeit von der Anzahl der Brandflächen und -schneisen und deren Anordnung im Raster abhängig. Es kann also nicht genau vorhergesagt werden, wie lange der Feuerlöscher braucht. Fest steht aber: Je größer das Raster, desto länger braucht der Feuerlöscher.

Aufgrund seiner Langsamkeit sollte man diesen Feuerlöscher nur für kleine Raster verwenden. Da er aber sehr wenig Speicherplatz verbraucht, würde ich ihn auch gerade für kleine Raster empfehlen. Bei größeren Rastern eignet es sich, einen anderen Feuerlöscher zu wählen.

3.4 TeilrasterBruteforceFeuerlöscher

Dieser Feuerlöscher erweitert `BruteforceFeuerloescher`. In der `init`-Methode werden erst mal alle möglichen Teilraster gebildet. Dies geschieht, indem ich durch alle Rasterfelder durchge-

he und wenn das Rasterfeld ein Brandelement ist, dieses, falls es noch nicht im Teilraster liegt, in dieses einfüge, indem ich die Ränder des Teilrasters anpasse:

```
if (raster.get(i, j) == BRAND)
{
      if (minx > i) minx = i;
      if (miny > j) miny = j;
      if (maxx < i) maxx = i+1;
      if (maxy < j) maxy = j+1;
}
```

Anschließend wird die `init`-Methode der Superklasse aufgerufen. Wenn die in der `protected`-Variable `loesung` gespeicherte Lösung das Feuer an den Rand brennen lässt ohne es dort zu löschen, so wird das Raster um 1 Feld in jede Richtung vergrößert, insofern das Raster noch nicht die volle Größe erreicht hat, und der Vorgang wird wiederholt. Anschließend wird der Löschweg "übersetzt", d.h. die Werte, die für das Teilraster angepasst sind, werden auf die entsprechenden Felder im Originalraster zurückgeführt.

3.4.1 Laufzeit

Der Arbeitsspeicherverbrauch ist ähnlich wie beim BruteforceFeuerlöscher, nur dass hier noch eine `LinkedList<Teilraster>` mehr im Arbeitsspeicher rumliegt. Die Laufzeit variiert hier sehr stark von Raster zu Raster. Man kann nicht genau sagen wie lange der Feuerlöscher braucht. Die Größe des Rasters spielt hierbei kaum eine Rolle, es kommt vielmehr darauf an wie viele Brandschneisen und Brandflächen existieren und vor allem wie weit diese voneinander entfernt sind. Der Speicherverbrauch des 15x15-Rasters, welches ich auch schon bei der Laufzeit des BruteforceFeuerlöschers als Beispiel gebracht habe, liegt bei 200 MiB. Die Rechenzeit ist vergleichbar mit dem BruteforceFeuerlöscher. Die Anwendung dieses Feuerlöschers ist für große Raster, die in einem kleinen Teil einen Brandherd und ein paar Brandschneisen haben, angelegt.

3.5 TreeBuildingFeuerlöscher

Dieser Feuerlöscher übernimmt den 2. Feuerlöscher, der in der Lösungsidee beschrieben wurde. Ich habe hier eine etwas veränderte Implementation des Baums gemacht: Ich speichere eine Liste von Möglichkeiten (Klasse `Possibility`, siehe unten). Dies entspricht den Knoten, die mit dem Wurzelknoten verbunden sind. Jede `Possibility` speichert eine Liste von weiteren `Possibility`, also die Knoten, die von diesem Knoten ausgehen. `Possibility`'s, welche eine leere Liste enthalten, sind somit Blätter. Jede Kante im Graphen ist eine Brücke.

Die Klasse `Possibility` speichert die aktuelle Lösung als `Point` und eine Liste mit weiteren `Possibility`'s. Dabei habe ich den `Point` so implementiert, dass jeder Punkt nur einmal im Arbeitsspeicher vorhanden sein kann. Dies habe ich durch eine Unterklasse von `Point` erledigt, die ich `MemorySavePoint` genannt habe. Diese speichert eine Liste von bereits benutzen Punkten. Somit wird z.B. der Punkt (0|0) beim ersten Mal, bei dem er gebraucht wird, neu erstellt. Beim 2. Mal, wo dieser gebraucht wird, ist er bereits vorhanden und derselbe

Punkt wird zurückgegeben. Wenn jedoch beide Punkte (0|0), die bis jetzt existieren, gelöscht werden, so sind sie immer noch in der Liste vorhanden, werden vom GC folglich auch nicht gelöscht. Da letzteres Beispiel nicht besonders häufig vorkommen sollte, kann ich diesen Fall vernachlässigen.

Da es sehr schwer zu überblicken ist, wie lange dieser Feuerlöscher noch braucht, habe ich eine kleine GUI geschrieben, welche dem Benutzer während des TreeBuildings anzeigt, wie lange es schon gebraucht hat, wie viel Speicherplatz es verbraucht, wie viel Wald die bisher beste Lösung noch übrig hat und vor allem bei welcher Ebene, d.h. Suchtiefe, der Algorithmus schon ist. Letzteres ist interessant, da man hiermit einen Überblick über den Löschvorgang bekommen kann. Weiterhin hat diese Klasse eh einen Thread, der alle 500 Millisekunden aufgerufen wird. Da ich einen sehr hohen Speicherverbrauch habe, gibt es einen tree-Ignorance-Wert, der auch ohne bessere Lösung für das Ignorieren eines Wegs verantwortlich sein kann. Diesen habe ich anfangs auf 20 gesetzt. Wenn dieser Thread nun mitkriegt, dass der Speicherverbrauch über die Grenze, welche mit der Systemeigenschaft `treebuilding.maxstore` angegeben werden muss, steigt, wird dieser Wert auf minimal 5 verkleinert. Dadurch sollte der Speicherverbrauch wieder abfallen. Um diese Systemeigenschaft automatisch zu setzen, habe ich ein Shellscript geschrieben, welches als Argument den maximalen Speicherverbrauch in MiB entgegen nimmt. Dieser wird unverändert an die JVM übergeben. Die Systemeigenschaft `treebuilding.maxstore` wird 500 MiB darunter gesetzt; dies hat sich in den meisten Fällen bewährt. Bitte beachten Sie, dass hierfür das Paket `apcalc` zur Verfügung stehen muss, welches in fast allen Linux-Repositories zu finden ist.

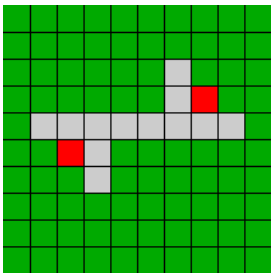
3.5.1 Laufzeit

Die Laufzeit und der Speicherverbrauch kann bei diesem Algorithmus eigentlich überhaupt nicht vorhergesagt werden. Es gibt ziemlich viele `Possibility`'s im Arbeitsspeicher, wie viele ist sehr stark vom Raster und vom TreeIgnorance-Wert abhängig. Die Laufzeit wird kaum von der Rastergröße beeinflusst, sondern von der Anzahl der Schritte im besten Löschweg, insofern dieser noch gespeichert ist und nicht aufgrund des TreeIgnorance-Werts vorher entfernt wurde. Dieser Feuerlöscher eignet sich also vor allem für große Raster, wo absehbar ist, dass der `TeilrasterBruteforceFeuerlöscher` sehr lange für brauchen wird. Allerdings sollte es nicht zu viele gleichwertige Löschwegansätze geben, da der Feuerlöscher sonst mit dem Arbeitsspeicher nicht zurande kommen wird.

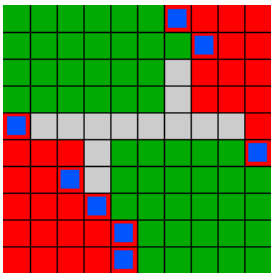
Für das 15x15-Raster, welches ich auch bei den anderen beiden Feuerlöschern benutzt habe, hat dieser Feuerlöscher 1300 MiB Arbeitsspeicher benutzt. Er hat außerdem etwas länger als die anderen beiden Algorithmen gebraucht, dies wäre bei noch größeren Rastern aber anders herum.

4 Umsetzung - Teilaufgabe 2

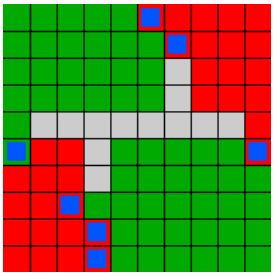
Ich habe hier folgendes Raster gefunden, bei dem es sich lohnt, Wald zu löschen:



Beide Algorithmen aus der Teilaufgabe 1 kommen auf folgendes Ergebnis:



Dabei werden 52 von 86 Waldstücken gerettet. Man kann jedoch leicht erkennen, dass ein Schritt völlig überflüssig war. Man hätte jedoch auch keinen besseres Brandstück löschen können. Per Hand konnte ich durch Löschen eines Waldstückes anstelle dieses Brandstücks folgende Lösung erreichen (mit 54 geretteten Waldstücken):



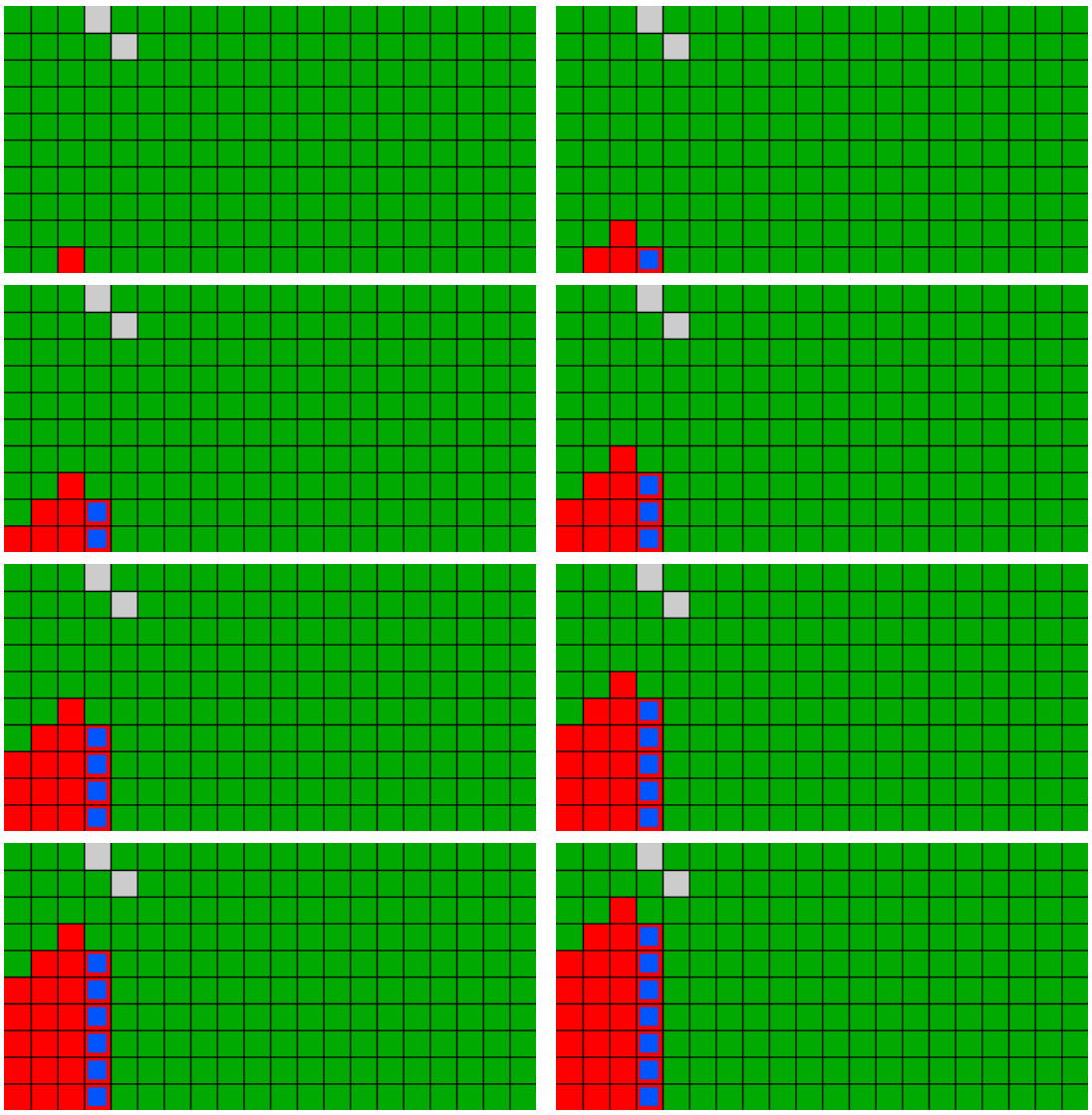
Dieses Beispiel zeigt eindeutig, wie ich in der Lösungsidee dieser Teilaufgabe auch schon geschrieben hab, dass es sich in einigen Fällen lohnt, Wald zu löschen. Die Dateien, mit denen man den Löschvorgang reproduzieren kann, habe ich im Ordner "Beispiele" mitgeliefert.

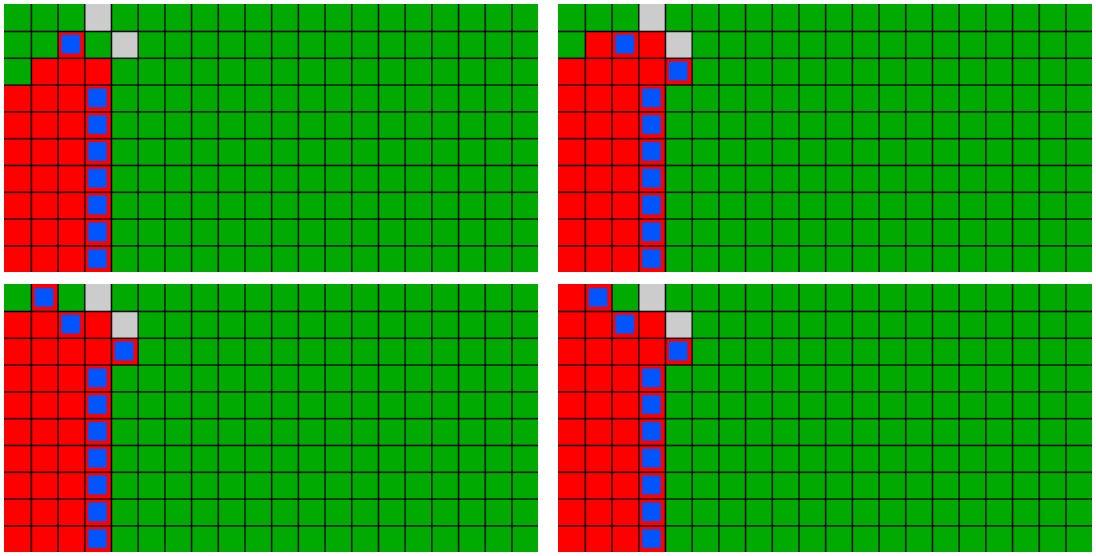
5 Beispiele - Teilaufgabe 1

5.1 BruteforceFeuerlöscher

Hier sind ein paar Raster und deren kompletter Löschweg inklusive gebrauchte Zeit des BruteforceFeuerlöschers (2.5 GHz):

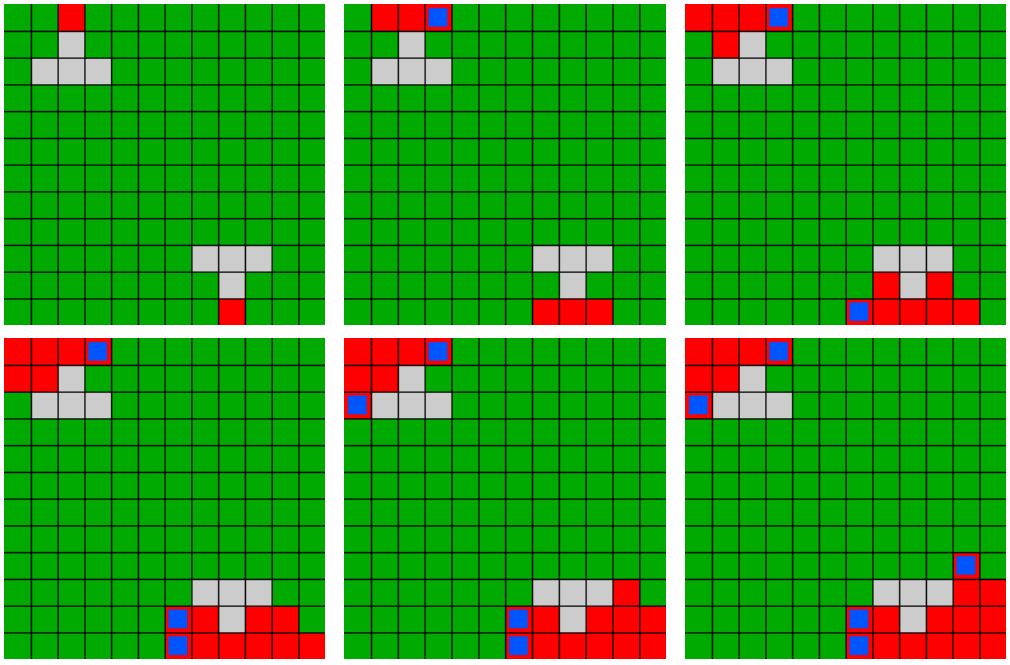
5.1.1 Beispiel 1

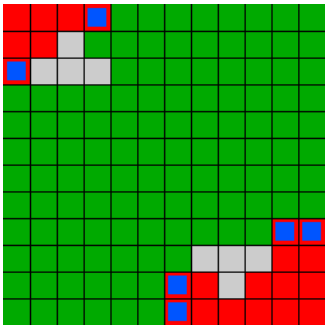




Benutzte Zeit: 00:04:38

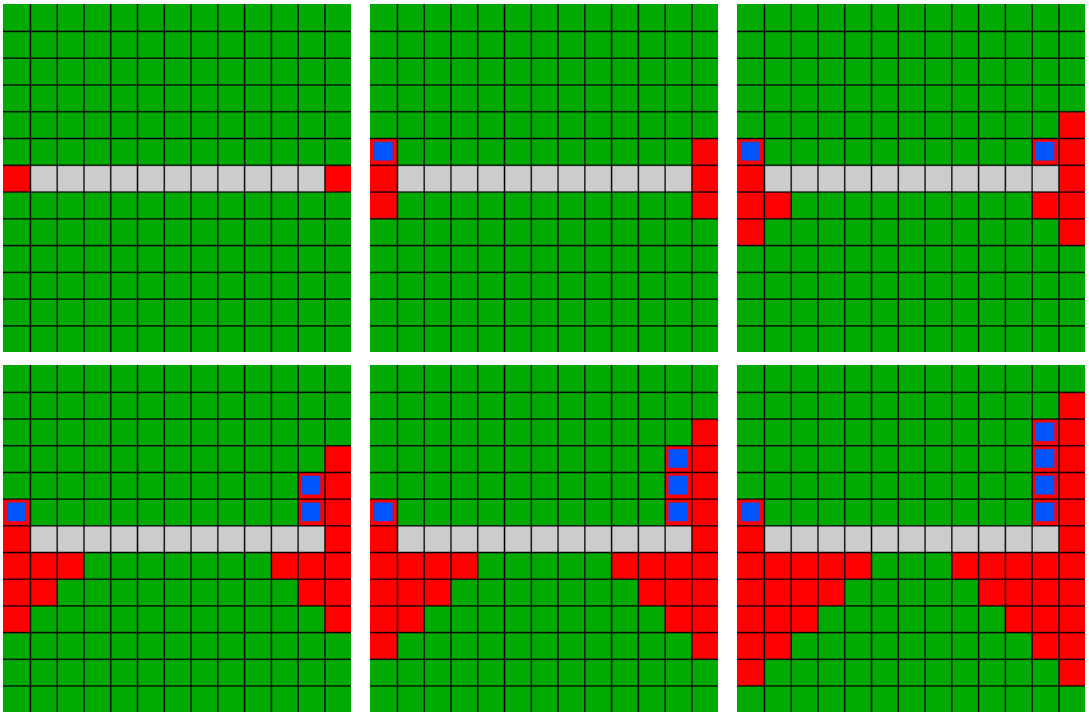
5.1.2 Beispiel 2

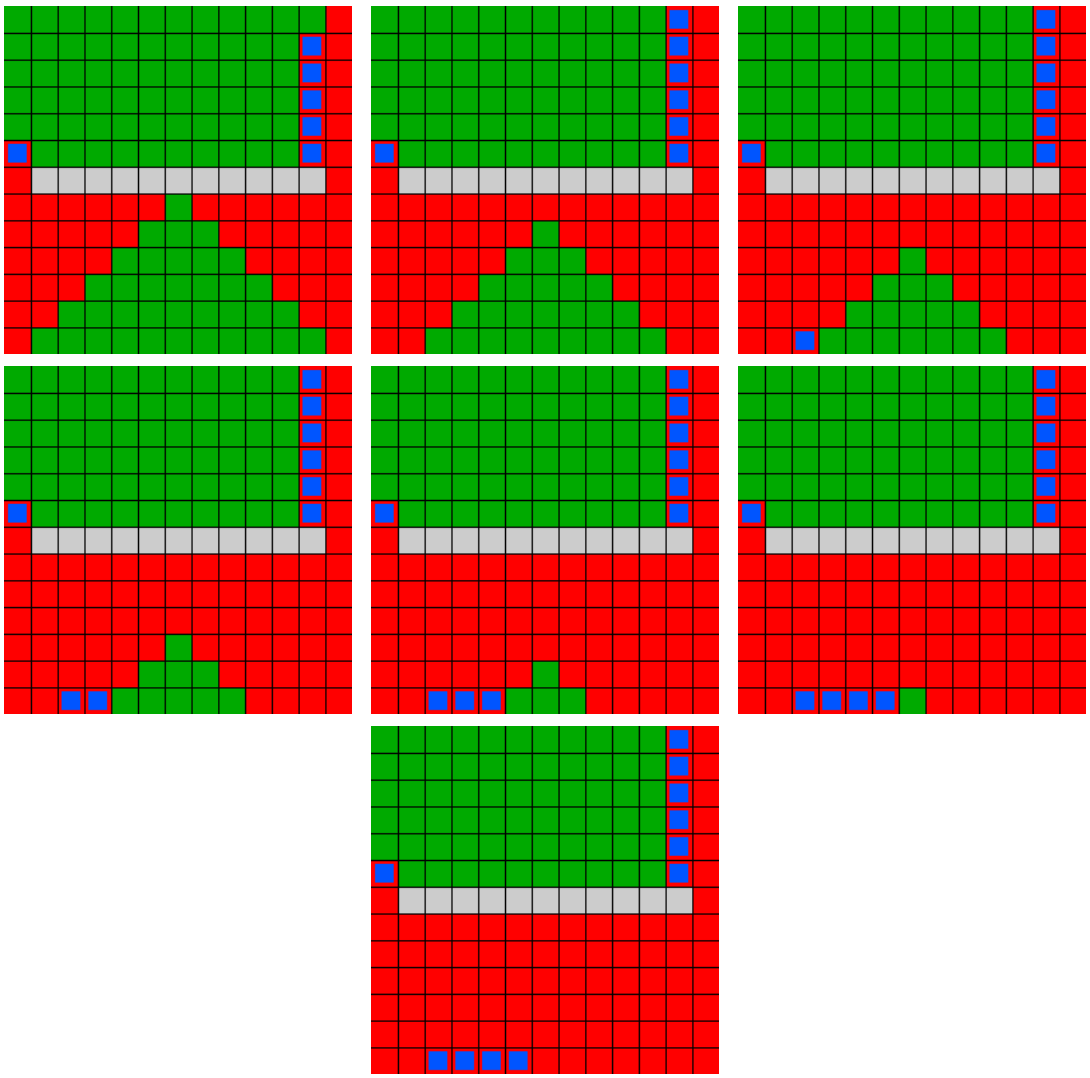




Benutzte Zeit: 00:04:03

5.1.3 Beispiel 3



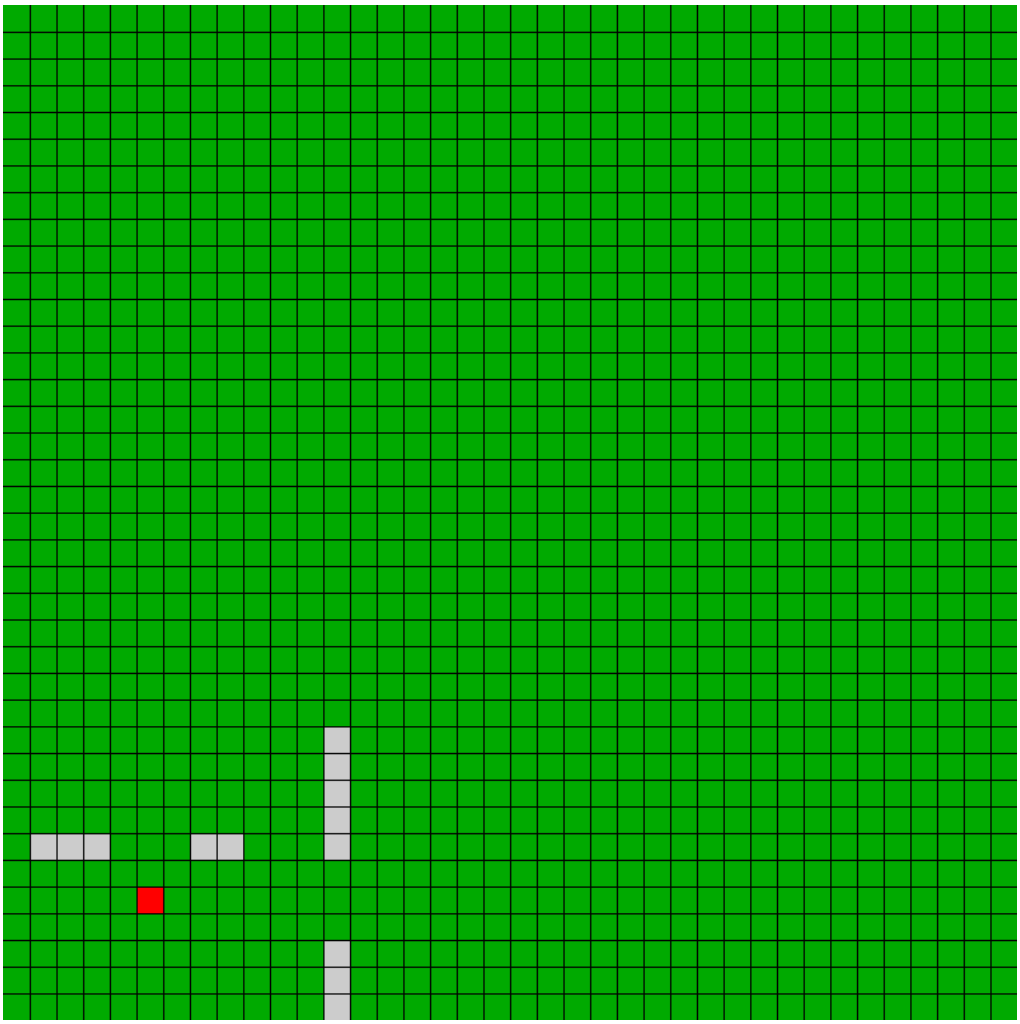


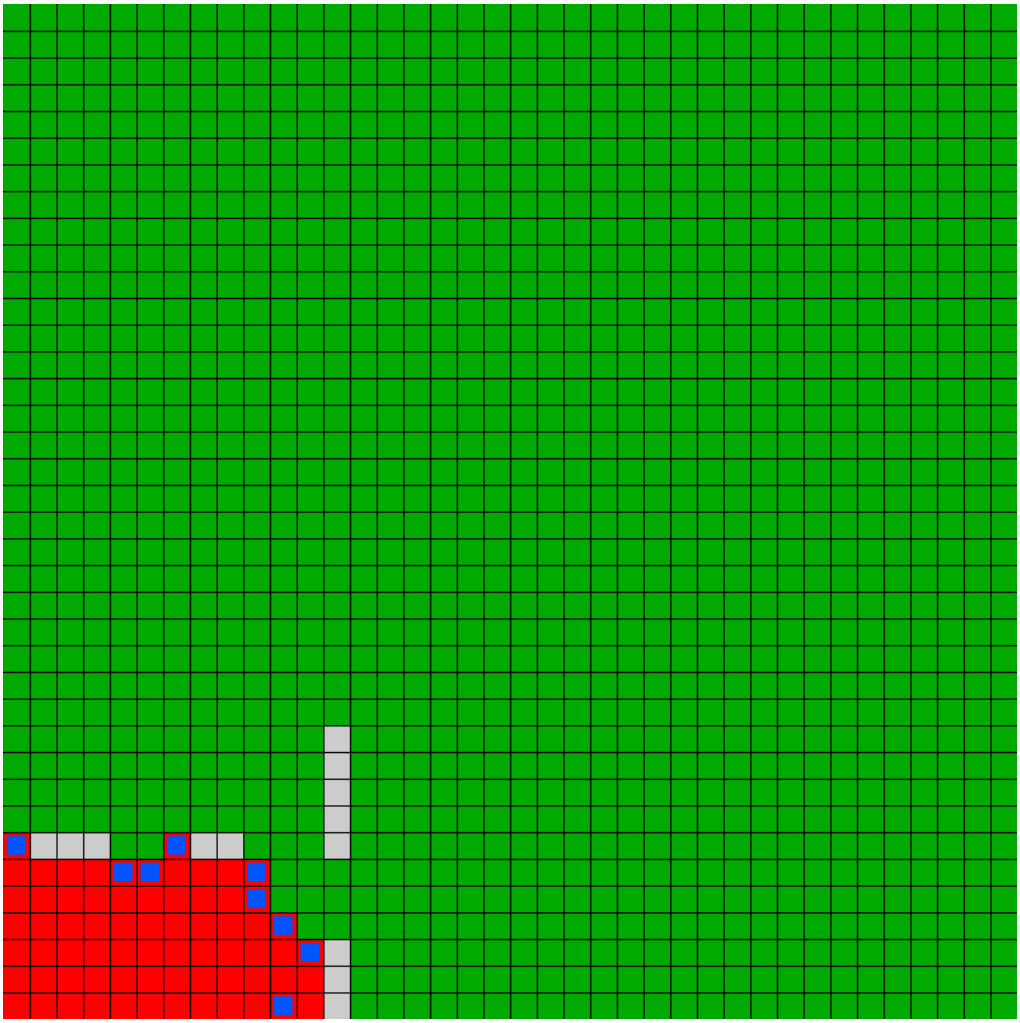
Benutzte Zeit: 00:05:06

5.2 TeilrasterBruteforceFeuerlöscher

Aus Platzgründen ist hier nur das Raster und das Ergebnis abgebildet. Außerdem natürlich wieder die verbrauchte Zeit bei 2.5 GHz.

5.2.1 Beispiel 4



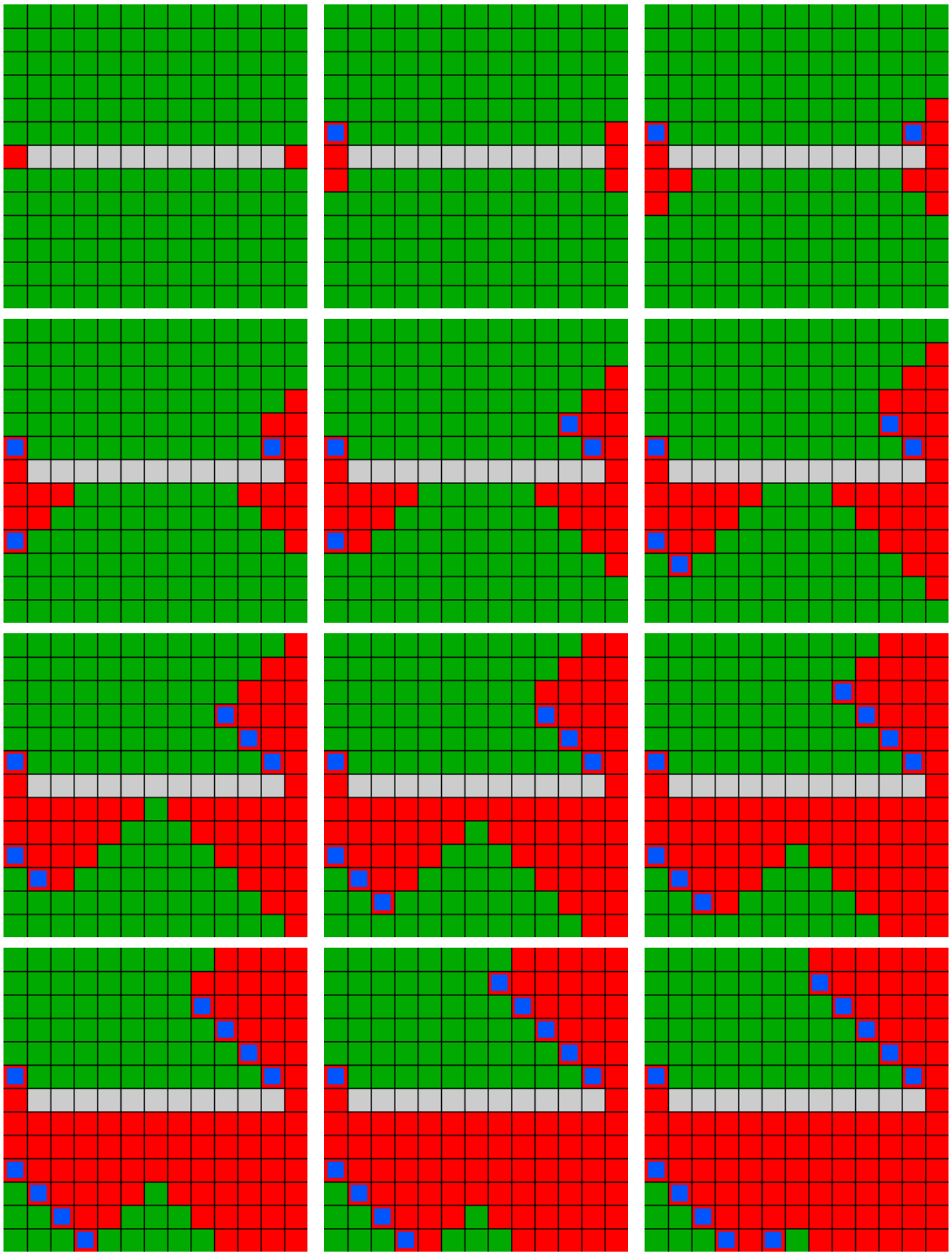


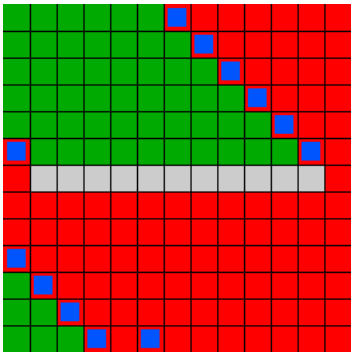
Benutzte Zeit: 00:01:32

5.3 TreeBuildingFeuerlöscher

Hier sind ein paar Raster und deren kompletter Löschweg inklusive gebrauchte Zeit des BruteforceFeuerlöschers (2.5 GHz):

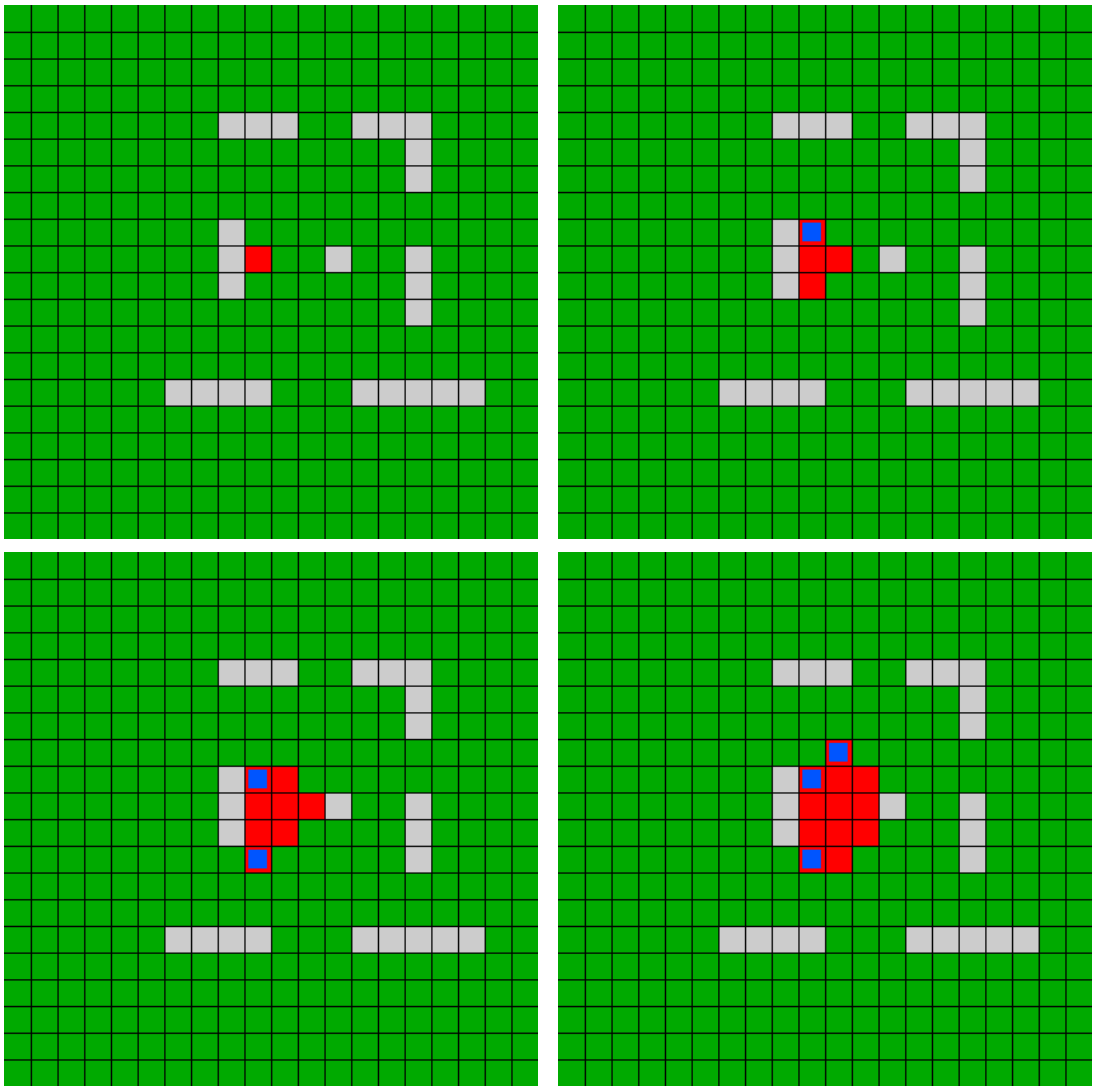
5.3.1 Beispiel 3

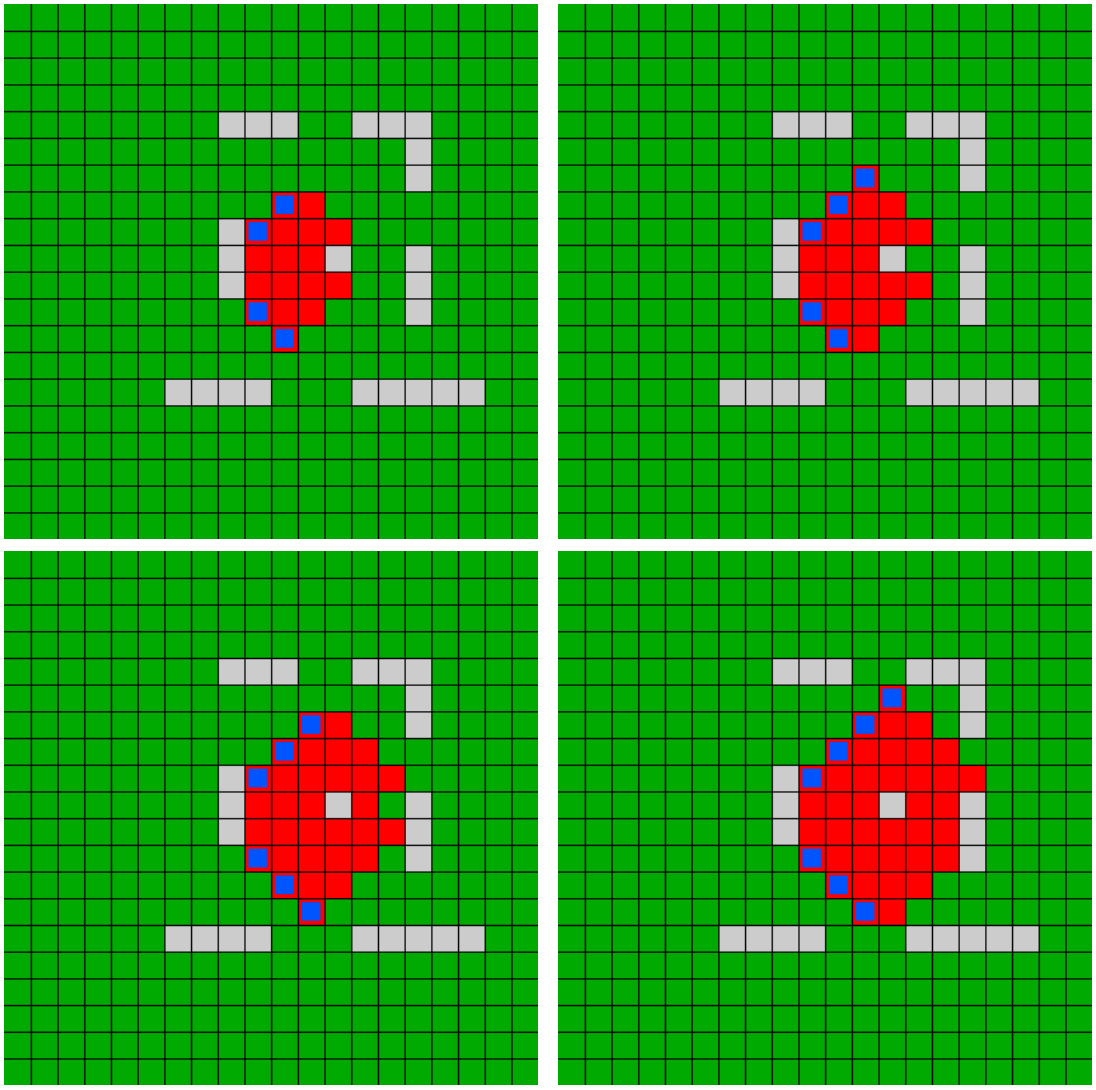


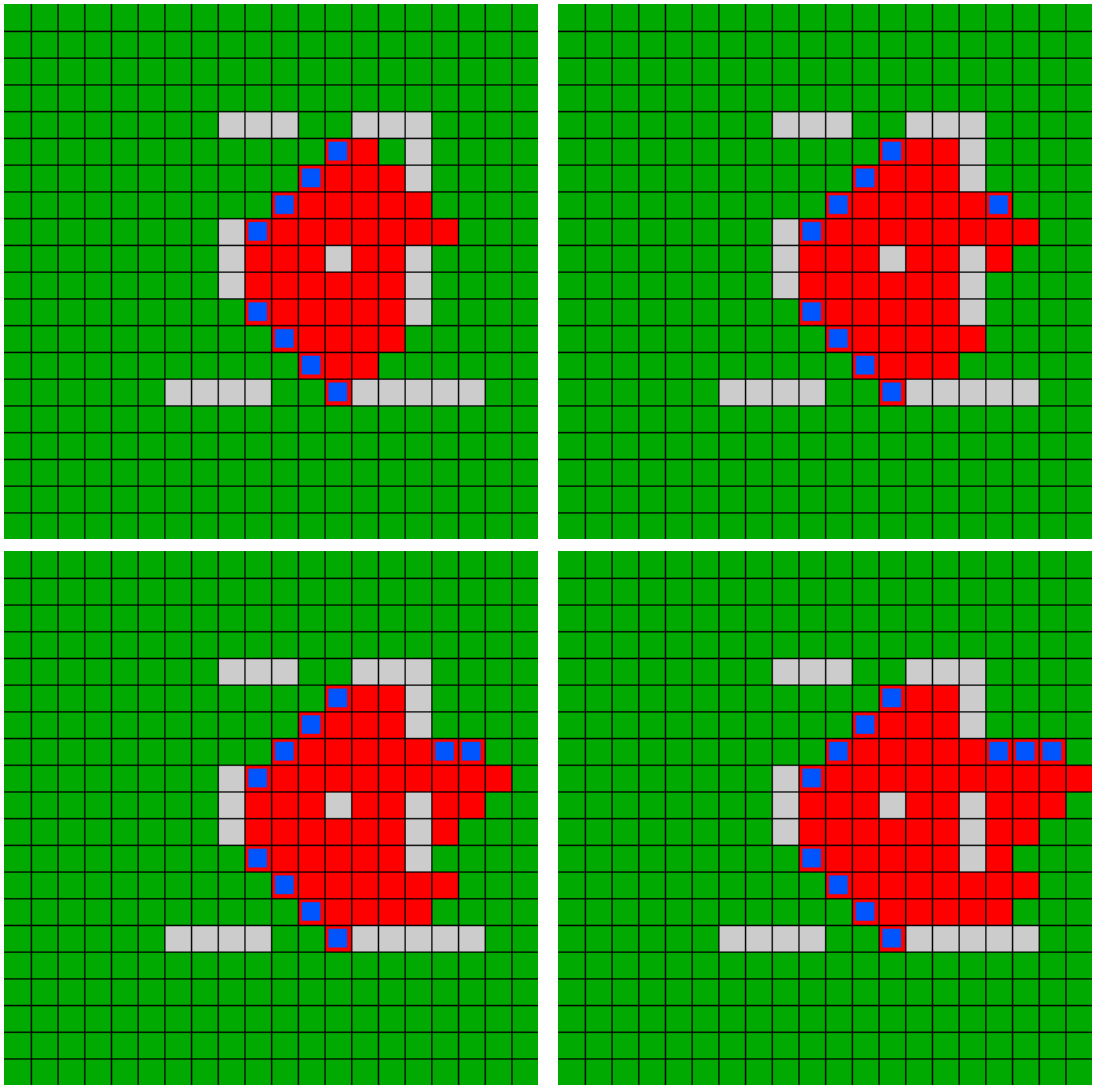


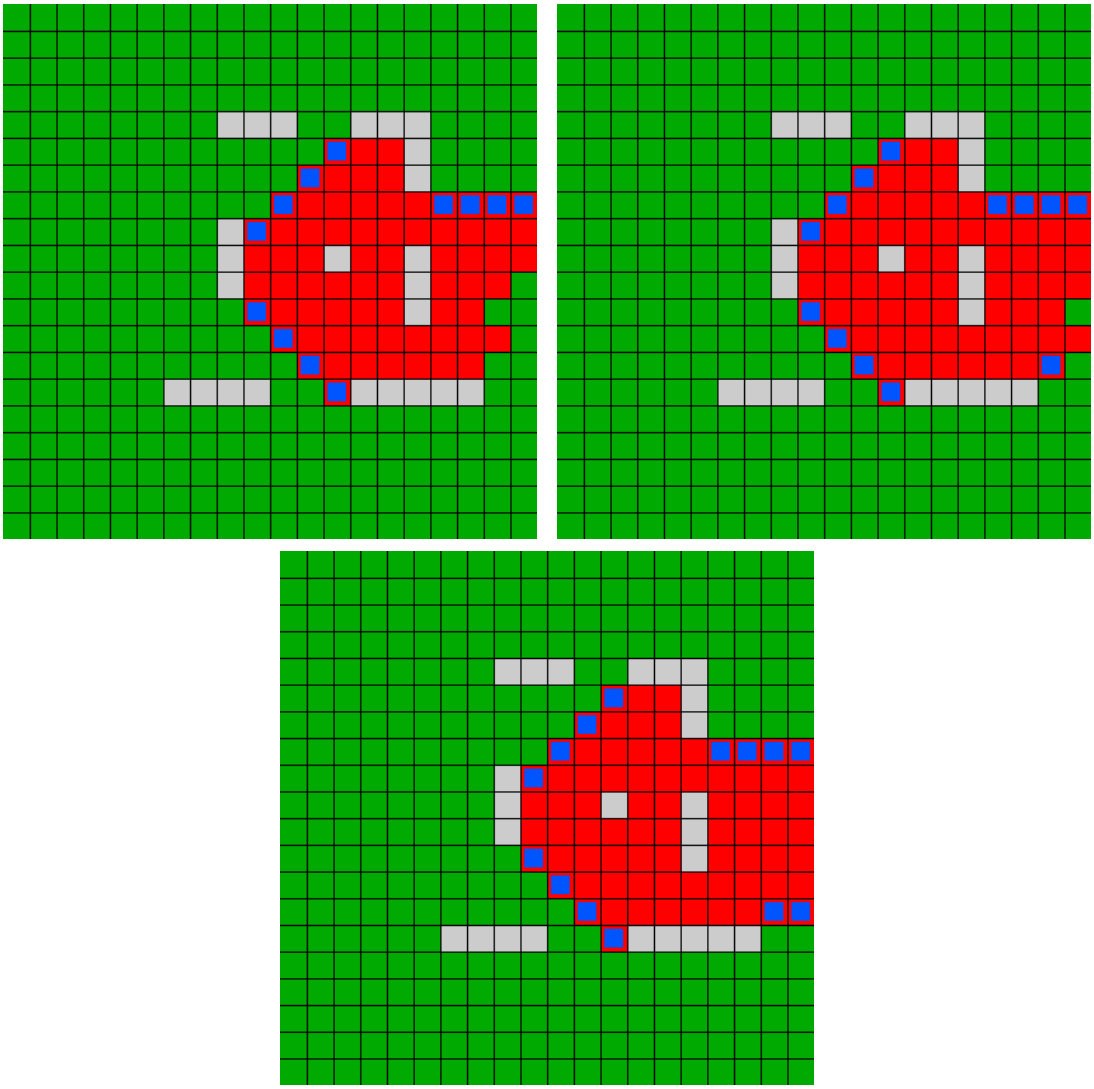
Benutzte Zeit: 00:03:32

5.3.2 Beispiel 5









Benutzte Zeit: 00:02:06

6 Quelltext - Teilaufgabe 1

Hier sind einige wichtige Quelltextausschnitte:

6.1 Die Klasse `Raster`

```
/**
* Diese Klasse speichert ein Raster.
* @author Dominic S. Meiser
*/
public class Raster
{
      public static final int WALD=1, SCHNEISE=2, BRAND=3, GELOESCHT=4;

      /**
      * Erstellt ein Raster mit der angegebenen Dimension. Alle Felder haben am Anfang
      * startValue.
      */
      public Raster (int width, int height, int startValue)
      {
            super();
            if ((width < 1) || (height < 1))
                  throw new IllegalArgumentException("Das Raster ist zu klein!");
            raster = new int[width][height];
            for (int i = 0; i < width; i++)
                  for (int j = 0; j < height; j++)
                        set(i, j, startValue);
      }
      private int[][] raster;

      /**
      * Gibt das raster zurück.
      */
      public int[][] getRaster ()
      {
            return raster;
      }
      /**
      * Gibt den Wert des Rasters an der angegebenen Position zurück.
      */
      public int get (int i0, int i1)
      {
            return getRaster()[i0][i1];
      }
      /**
      * Setzt den Wert des Rasters an der angegebenen Position auf value.
      */
      public void set (int i0, int i1, int value)
      {
            if (!((value == WALD) || (value == SCHNEISE) || (value == BRAND) ||
                        (value == WALD+GELOESCHT) || (value == BRAND+GELOESCHT)))
                  throw new IllegalArgumentException("Kein Rasterwert: "+value);
            raster[i0][i1] = value;
      }
      /**
      * Gibt die Größe des Rasters an.
      */
      public Dimension getSize ()
```

```
	{
		return new Dimension(getRaster().length, getRaster()[0].length);
	}

	/**
	* Zählt, wie oft value im Raster vorkommt.
	*/
	public int count (int value)
	{
		int count = 0;
		for (int i = 0; i < raster.length; i++)
			for (int j = 0; j < raster[i].length; j++)
				if (get(i,j) == value)
					count++;
		return count;
	}


	/** Gibt eine Beschreibung des Rasters an. */
	public String toString ()
	{
		StringBuilder str = new StringBuilder(getClass().getCanonicalName());
		str.append("[");

		for (int i = 0; i < raster.length; i++)
		{
			str.append(Arrays.toString(raster[i]));
			if (i != raster.length-1)
				str.append(", ");
		}

		str.append("]");
		return str.toString();
	}
}
```

6.2 Die Klasse `RasterUtils`

```
/**
* Gibt an, ob der Brand gelöscht wurde.
*/
public static boolean istGeloescht (Raster raster)
{
	// Der Brand ist gelöscht, wenn sich keine Brandfläche mehr ausbreiten
	// kann.
	for (int i = 0; i < raster.getSize().width; i++)
	{
		for (int j = 0; j < raster.getSize().height; j++)
		{
			if (raster.get(i, j) == BRAND)
			{
				if (i > 0)
					if (raster.get(i-1, j) == WALD)
						return false;
				if (j > 0)
					if (raster.get(i, j-1) == WALD)
						return false;
				if (i < raster.getSize().width-1)
```

```
                        if (raster.get(i+1, j) == WALD)
                              return false;
                  if (j < raster.getSize().height-1)
                        if (raster.get(i, j+1) == WALD)
                              return false;
            }
      }
      return true;
}

    /**
* Läst das Raster weiterbrennen.
*/
public static void burn (Raster raster)
{
      // Zuerst das Array clonen, damit ich nicht die in Brand geratenen Waldstücke
      // nochmals weiter entzünden lasse
      Raster bevore = clone(raster);
      // Jetzt Wald entzünden
      for (int i = 0; i < bevore.getSize().width; i++)
      {
            for (int j = 0; j < bevore.getSize().height; j++)
            {
                  if (bevore.get(i, j) == BRAND)
                  {
                        if (i > 0)
                              if (bevore.get(i-1, j) == WALD)
                                    raster.set(i-1, j, BRAND);
                        if (j > 0)
                              if (bevore.get(i, j-1) == WALD)
                                    raster.set(i, j-1, BRAND);
                        if (i < bevore.getSize().width-1)
                              if (bevore.get(i+1, j) == WALD)
                                    raster.set(i+1, j, BRAND);
                        if (j < bevore.getSize().height-1)
                              if (bevore.get(i, j+1) == WALD)
                                    raster.set(i, j+1, BRAND);
                  }
            }
      }
}
```

6.3 Die Klasse `BruteforceFeuerloescher`

```
public class BruteforceFeuerloescher extends Feuerloescher
{
      protected List<Point> loeschen;
      protected int bestSolution = 0;
      public void init (Raster raster)
      {
            // Lösung rekursiv bestimmen
            long time = System.currentTimeMillis();
            System.out.println();
            loeschen = test(raster, new LinkedList<Point>());
            time = System.currentTimeMillis() - time;
            System.out.println("[ERGEBNIS] "+loeschen);
            System.out.println("[ERGEBNIS] "+bestSolution);
```

```
      System.out.print("[ERGEBNIS] "); printBashText("done ("+time+" ms)\n", "32");

      // Raster speichern
      File file = new File(System.getProperty("java.io.tmpdir")+File.separator+"buschfeuer",
                  "raster_bruteforce.bfr");
      try { RasterUtils.save(raster, loeschen, file); }
      catch (IOException ioe) { ioe.printStackTrace(); }
}

/**
* Durchsucht das Raster rekursiv mittels Bruteforce. vorhanden gibt die bereits
* in der Liste vorhandenen Brandstücke an.
*/
protected List<Point> test (Raster raster, List<Point> vorhanden)
{
      int wald = raster.count(WALD);
      if ((wald > bestSolution) && RasterUtils.istGeloescht(raster))
      {
            System.out.println("[GELÖSCHT] "+wald+": "+vorhanden);
            bestSolution = wald;
            return vorhanden;
      }

      // Abbruch, wenn Anzahl der Waldstücke kleiner als bestSolution. Wenn die
      // Anzahl der Waldstücke gleich bestSolution ist, dann ist die Lösung
      // uninteressant, da bereits eine gleichwertige oder bessere Lösung gefunden
      // wurde.
      if (wald <= bestSolution)
            return null;

      // Das Feuer einmal sich weiterverbreiten lassen
      Raster neuesRaster = RasterUtils.clone(raster);
      RasterUtils.burn(neuesRaster);

      // Zuerst die Brandherde mit der Anzahl der in der Umgebung liegenden
      // Waldstücken ermitteln
      List<RasterValue> braende = new LinkedList<>();
      for (int i = 0; i < neuesRaster.getSize().width; i++)
      {
            for (int j = 0; j < neuesRaster.getSize().height; j++)
            {
                  // Brennt das Waldstück überhaupt?
                  if (neuesRaster.get(i, j) == BRAND)
                  {
                        // nicht brenndenden Wald in Umgebung suchen
                        int wald_in_umgebung = 0;
                        if (i > 0)
                              if (neuesRaster.get(i-1, j) == WALD)
                                    wald_in_umgebung++;
                        if (j > 0)
                              if (neuesRaster.get(i, j-1) == WALD)
                                    wald_in_umgebung++;
                        if (i < neuesRaster.getSize().width-1)
                              if (neuesRaster.get(i+1, j) == WALD)
                                    wald_in_umgebung++;
                        if (j < neuesRaster.getSize().height-1)
                              if (neuesRaster.get(i, j+1) == WALD)
                                    wald_in_umgebung++;
                        // Brandstück hinzufügen, wenn mindestens 1 Waldstück in der
```

```
                        // Umgebung liegt
                        if (wald_in_umgebung != 0)
                              braende.add(new RasterValue(i, j, BRAND, wald_in_umgebung));
                  }
            }
      }

      if (braende.size() == 0)
      {
            vorhanden.add(null);
            List<Point> erg = test(neuesRaster, vorhanden);
            vorhanden.remove(vorhanden.size()-1);
            return erg;
      }

      // Jetzt alle Brandstücke durchgehen und rekursiv weiterarbeiten.
      List<Point> best = null;
      for (RasterValue elem : braende)
      {
            vorhanden.add(elem.pos);
            neuesRaster.set(elem.pos.x, elem.pos.y, neuesRaster.get(elem.pos.x, elem.pos.y)+GELOESCHT);
            int bestBefore = bestSolution;
            List<Point> ergebnis = test(neuesRaster, vorhanden);
            if ((ergebnis != null) && (bestSolution > bestBefore))
                  best = clone(ergebnis);
            neuesRaster.set(elem.pos.x, elem.pos.y, neuesRaster.get(elem.pos.x, elem.pos.y)-GELOESCHT);
            vorhanden.remove(elem.pos);
      }
      return best;
}


int i = -1;
public Point delete (Raster raster)
{
      if (i >= loeschen.size()-1) return null;
      return loeschen.get(++i);
}


/** Klont die angegebene Liste. */
protected static List<Point> clone (List<Point> list0)
{
      List<Point> list1 = new LinkedList<>();
      for (Point e : list0)
            list1.add(e);
      return list1;
}
}
```

6.4 Die Klasse `TeilrasterBruteforceFeuerloescher`

```
public class TeilrasterBruteforceFeuerloescher extends BruteforceFeuerloescher
{
      /**
      * Zerlegt das Raster und lässt die Superklasse einen Bruteforce darauf anwenden.
      */
      public void init (Raster raster)
```

```
{
	long time = System.currentTimeMillis();
	System.out.print("[FEUERLÖSCHER] teile Raster ...");

	List<Teilraster> teilraster = new LinkedList<>();

	// Zuerst alle Teilraster bilden, in denen alle Brand-Elemente vorhanden sind
	int minx=raster.getSize().width-1, miny=raster.getSize().height-1, maxx=0, maxy=0;
	for (int i = 0; i < raster.getSize().width; i++)
	{
		for (int j = 0; j < raster.getSize().height; j++)
		{
			if (raster.get(i, j) == BRAND)
			{
				if (minx > i) minx = i;
				if (miny > j) miny = j;
				if (maxx < i) maxx = i+1;
				if (maxy < j) maxy = j+1;
			}
		}
	}
	if ((minx >= maxx) || (miny >= maxy)) // negative Arraygrößen ausschließen
	{
		maxx = minx+1;
		maxy = miny+1;
	}
	// Teilraster bilden
	Teilraster r = null;
	do
	{
		r = new Teilraster (minx, maxx, miny, maxy, WALD);
		for (int i = minx; i < maxx; i++)
			for (int j = miny; j < maxy; j++)
				if (raster.get(i, j) != WALD)
					r.set(i-minx, j-miny, raster.get(i, j));
		teilraster.add(r);
		if (minx > 0) minx-;
		if (miny > 0) miny-;
		if (maxx < raster.getSize().width) maxx++;
		if (maxy < raster.getSize().height) maxy++;
	} while (!r.getSize().equals(raster.getSize()));

	printBashText(" done ("+(System.currentTimeMillis()-time)+" ms, "+teilraster.size()+" Teilraster)\n", "32");

	// Jetzt Bruteforce auf das möglichste, kleinste Teilraster anwenden. Sobald das Feuer
	// an den Rand stößt und dort nicht gelöscht wurde, muss die nächstgrößere Rasterversion
	// verwendet werden
	for (Teilraster teilr : teilraster)
	{
		super.init(teilr);
		printBashText(teilr+" -> "+super.loeschen+"\n", "1;32");
		if ((loeschen != null) && isPossible(RasterUtils.clone(teilr), raster.getSize(), loeschen))
		{
			// Koordinaten "übersetzen"
			for (int i = 0; i < super.loeschen.size(); i++)
			{
				if (loeschen.get(i) != null)
				{
					loeschen.get(i).x += teilr.minx;
```

```
                        loeschen.get(i).y += teilr.miny;
                    }
                }
                // Fertig
                break;
            }
            bestSolution = 0;
            printBashText("[ERGEBNIS] nicht möglich\n", "31");
        }


        // Raster speichern
        File file = new File(System.getProperty("java.io.tmpdir")+File.separator+"buschfeuer",
                "raster_teilraster-bruteforce.bfr");
        try { RasterUtils.save(raster, loeschen, file); }
        catch (IOException ioe) { ioe.printStackTrace(); }
}

/**
* Überprüft, ob diese Lösung möglich ist oder ob das Raster vergrößert werden muss.
*/
protected boolean isPossible (Teilraster raster, Dimension size, List<Point> solution)
{
        // Brand simulieren
        for (Point p : solution)
        {
            RasterUtils.burn(raster);
            if (p != null)
                raster.set(p.x, p.y, raster.get(p.x, p.y)+GELOESCHT);
        }

        // oberer Rand
        if (raster.miny != 0)
        {
            for (int i = 0; i < raster.getSize().width; i++)
                if (raster.get(i, 0) == BRAND)
                    return false;
        }

        // rechter Rand
        if (raster.minx != 0)
        {
            for (int j = 0; j < raster.getSize().height; j++)
                if (raster.get(0, j) == BRAND)
                    return false;
        }

        // unterer Rand
        if (raster.maxy != size.height)
        {
            for (int i = 0; i < raster.getSize().width; i++)
                if (raster.get(i, raster.getSize().height-1) == BRAND)
                    return false;
        }

        // linker Rand
        if (raster.maxx != size.width)
        {
            for (int j = 0; j < raster.getSize().height; j++)
```

```
                        if (raster.get(raster.getSize().width-1, j) == BRAND)
                              return false;
                  }

                  return true;
            }
}
```

6.5 Die Klasse `TreeBuildingFeuerloescher`

```
public class TreeBuildingFeuerloescher extends Feuerloescher
{
      protected List<Point> solution;
      protected int bestSolution = 0;
      protected int bestTree=0, treeIgnorance=20;

      public void init (Raster raster)
      {
            itSeek(raster);
            if (solution == null)
            {
                  System.out.print("[ERGEBNIS] ");
                  printBashText("failed (no solution)\n", "31");
                  solution = new LinkedList<>();
                  return;
            }
            System.out.println("[ERGEBNIS] "+solution);
            System.out.println("[ERGEBNIS] "+bestSolution);

            File file = new File(System.getProperty("java.io.tmpdir")+File.separator+"buschfeuer",
                        "raster_treebuilding.bfr");
            try { RasterUtils.save(raster, solution, file); }
            catch (IOException ioe) { ioe.printStackTrace(); }
      }

      /**
      * Diese Unterklasse von Point hat eine statische Liste aller bis jetzt gebrauchten
      * Punkte. Es werden somit nur Punkte erzeugt, die noch nicht existieren.
      * @author Dominic S. Meiser
      */
      protected static class MemorySavePoint extends Point
      {
            private static final long serialVersionUID = -5208906917530817216L;

            /** Alle Punkte. */
            private static final List<MemorySavePoint> all = new LinkedList<>();

            /**
            * Sucht den Punkt (x|y). Ist er noch nicht vorhanden wird er hinzugefügt.
            * Anschließend wird er zurückgegeben.
            */
            public static MemorySavePoint newPoint (int x, int y)
            {
                  // Punkt suchen
                  for (MemorySavePoint p : all)
                        if ((p.x == x) && (p.y == y))
                              return p;
```

```
            // Punkt erstellen
            MemorySavePoint p = new MemorySavePoint(x, y);
            all.add(p);
            return p;
      }

      /** Erstellt einen normalen Punkt. */
      private MemorySavePoint (int x, int y) { super(x, y); }

      /** Gibt eine (sehr kurze) Beschreiung des Punkts zurück. */
      public String toString ()
      {
            return "("+x+"|"+y+")";
      }
}

/**
* Diese Klasse speichert eine Möglichkeit zum nächsten Schritt in der
* Brandlöschung.
* @author Dominic S. Meiser
*/
protected static class Possibility
{
      /** Die auf diese Möglichkeit folgenden Möglichkeiten. */
      public List<Possibility> nextPossibilities;
      /** Diese Möglichkeit. */
      public Point thisPossibility;

      /**
      * Erzeugt eine neue Möglichkeit mit der angegebenen Position. Die
      * Parameter werden in einen <code>MemorySavePoint</code> umgewandelt.
      */
      public Possibility (int x, int y)
      {
            this(MemorySavePoint.newPoint(x, y));
      }
      /**
      * Erzeugt eine neue Möglichkeit mit der angegebenen Position.
      */
      public Possibility (MemorySavePoint p)
      {
            thisPossibility = p;
            nextPossibilities = new LinkedList<>();
      }
      /**
      * Erzeugt eine neue Möglichkeit mit der angegebenen Position. Der
      * Parameter wird in einen <code>MemorySavePoint</code> umgewandelt.
      */
      public Possibility (Point p)
      {
            this(p.x, p.y);
      }
      /**
      * Erzeugt eine neue Möglichkeit, bei der keine Position gespeichert
      * wird.
      */
      public Possibility ()
      {
            thisPossibility = null;
```

```
            nextPossibilities = new LinkedList<>();
      }
}

/**
* Dieser Frame zeigt dem Benutzer Informationen über den aktuell laufenden
* Prozess an.
* @author Dominic S. Meiser
*/
protected class ProcessFrame extends JFrame
{
      private static final long serialVersionUID = -3450340340719622555L;

      private JLabel ebene, best, time, store;
      private JButton stop;
      private long startTime;

      public ProcessFrame ()
      {
            super("TreeBuildingFeuerlöscher - Prozess Informationen");

            JPanel cp = new JPanel ();
            SpringLayout layout = new SpringLayout();
            cp.setLayout(layout);

            ebene = new JLabel("Ebene: <unknown>");
            cp.add(ebene);
            layout.putConstraint(NORTH, ebene, 10, NORTH, cp);
            layout.putConstraint(WEST, ebene, 10, NORTH, cp);
            layout.putConstraint(EAST, ebene, -10, EAST, cp);

            best = new JLabel("Beste Lösung: <unknown>");
            cp.add(best);
            layout.putConstraint(NORTH, best, 10, SOUTH, ebene);
            layout.putConstraint(WEST, best, 10, NORTH, cp);
            layout.putConstraint(EAST, best, -10, EAST, cp);

            time = new JLabel("Verbrauchte Zeit: <unknown>");
            cp.add(time);
            layout.putConstraint(NORTH, time, 10, SOUTH, best);
            layout.putConstraint(WEST, time, 10, NORTH, cp);
            layout.putConstraint(EAST, time, -10, EAST, cp);

            store = new JLabel("Verbrauchter Arbeitsspeicher: <unknown>");
            cp.add(store);
            layout.putConstraint(NORTH, store, 10, SOUTH, time);
            layout.putConstraint(WEST, store, 10, NORTH, cp);
            layout.putConstraint(EAST, store, -10, EAST, cp);

            stop = new JButton("Aktuelle Lösung übernehmen");
            stop.addActionListener((ActionEvent e) -> poss.clear());
            cp.add(stop);
            layout.putConstraint(HORIZONTAL_CENTER, stop, 0, HORIZONTAL_CENTER, cp);
            layout.putConstraint(SOUTH, stop, -10, SOUTH, cp);

            setContentPane(cp);
            setSize(350,200);
            setLocationRelativeTo(null);
            setDefaultCloseOperation(DISPOSE_ON_CLOSE);
```

```
            }

            public void open ()
            {
                  startTime = System.currentTimeMillis();
                  setAlwaysOnTop(true);
                  setVisible(true);
                  new Thread((() -> autoUpdate()), "ProcessInfo").start();
            }

            private void autoUpdate ()
            {
                  while (isVisible())
                  {
                        best.setText("Beste Lösung: "+bestSolution);

                        long d = (long) ((System.currentTimeMillis() - startTime) / 1000.0 / 60 / 60 / 24);
                        long h = (long) ((System.currentTimeMillis() - startTime) / 1000.0 / 60 / 60 - d * 24);
                        long m = (long) ((System.currentTimeMillis() - startTime) / 1000.0 / 60 - d * 24 * 60 - h * 60);
                        long s = (long) ((System.currentTimeMillis() - startTime) / 1000.0 - d * 24 * 60 * 60 - h * 60 * 60
- m * 60);
                        time.setText("Verbrauchte Zeit: "+(d<10 ? "0"+d : d)+":"+(h<10 ? "0"+h : h)+":"+(m<10 ? "0"+m : m)+":"+(s<10
? "0"+s : s));

                        if ((Runtime.getRuntime().totalMemory() >
                                    Integer.parseInt(System.getProperty("treebuilding.maxstore"))*1024*1024)
                                          && (treeIgnorance > 5))
                              treeIgnorance-;
                        store.setText("Verbrauchter Arbeitsspeicher: "+(Runtime.getRuntime().totalMemory() / 1024.0 / 1024)+"
MiB");

                        try { Thread.sleep(500); } catch (Exception e) {}
                  }
            }

            public void update (int depth)
            {
                  ebene.setText("Ebene: "+depth);
            }
      }

      /** Das ProcessPane das den aktuellen Prozess visulaisiert. */
      protected final ProcessFrame processFrame = new ProcessFrame();

      /** Eine Liste, in der für jede Position verschiedene Punkte gespeichert werden.
      * Hierbei handelt es sich um den Baum, der gebildet wird. */
      List<Possibility> poss = new LinkedList<>();
      /**
      * Diese Methode sucht iterativ im Raster r nach Möglichkeiten. Dabei wird
      * ein immer weiter wachsender Baum aufgestellt. Die beste Möglichkeit wird
      * anschließend in solution gespeichert.. Zwischendurch wird die beste Lösung
      * auch schon in solution gespeichert.
      */
      protected void itSeek (Raster r)
      {
            processFrame.open();

            // Für verschiedene Tiefen im Baum durchloopen und Lösungen finden.
            for (int depth = 1; (depth!=1 ? poss.size()!=0 : true) &&
```

```
                (depth < r.getSize().width*r.getSize().height); depth++)
        {
                processFrame.update(depth);

                bestTree = -1;

                // Das Raster vorbereiten
                Raster raster = RasterUtils.clone(r);
                RasterUtils.burn(raster);

                // Beim 1. Mal müssen alle Möglichkeiten erstmal bereitgestellt werden
                if (depth == 1)
                {
                        List<RasterValue> braende = getBraende(raster);
                        for (RasterValue rv : braende)
                                poss.add(new Possibility(rv.pos));

                        continue;
                }

                // Sonst für jede Möglichkeit weitere Möglichkeiten erstellen und schauen ob
                // bei einer Möglichkeit der Brand gelöscht wurde. Wenn Möglichkeiten dabei
                // entfernt werden sollen dies tun.
                List<Possibility> remove = new LinkedList<>();
                for (Possibility p : poss)
                {
                        Raster clone = RasterUtils.clone(raster);
                        clone.set(p.thisPossibility.x, p.thisPossibility.y, BRAND+GELOESCHT);
                        LinkedList<Point> added = new LinkedList<>();
                        added.add(p.thisPossibility);
                        if (!itSeek0(clone, p, added))
                                remove.add(p);
                }
                for (Possibility p : remove)
                        poss.remove(p);
        }

        // fertig
        processFrame.dispose();
    }
    /**
    * Diese Methode sucht rekursiv die Enden von p und schaut welche weiteren Lösungen zur
    * Verfügung stehen.
    * @param raster Das Raster. Es wird in dieser Methode verändert, es sollte also ein
    * geklontes Raster übergeben werden.
    * @return Ob p weiterhin in der Liste enthalten sein soll.
    */
    protected boolean itSeek0 (Raster raster, Possibility p, LinkedList<Point> added)
    {
        // schauen, ob diese Möglichkeit noch benötigt wird
        int wald = raster.count(WALD);
//          System.out.println("[TREE] "+wald+": "+added);
        if (wald <= bestSolution)
                return false;

        if (p.thisPossibility != null)
                raster.set(p.thisPossibility.x, p.thisPossibility.y, BRAND+GELOESCHT);

        // Raster weiterbrennen lassen
```

```
		RasterUtils.burn(raster);

		// Wenn hier keine weitere Möglichkeit mehr ist, diese Möglichkeit begutachten
		if (p.nextPossibilities.size() == 0)
		{
			// Zuerst diese Möglichkeit in relation zur bisher besten bewerten
			if (wald < bestTree-treeIgnorance)
				return false;
			if (wald > bestTree)
				bestTree = wald;


			// Wenn jetzt zu wenig Wald noch da ist, diese Möglichkeit verwerfen
			wald = raster.count(WALD);
			if (wald <= bestSolution) return false;

			// Weitere Möglichkeiten suchen
			List<RasterValue> braende = getBraende(raster);
			for (RasterValue rv : braende)
			{
				raster.set(rv.pos.x, rv.pos.y, BRAND+GELOESCHT);

				if (RasterUtils.istGeloescht(raster))
				{
					bestSolution = wald;
					added.add(rv.pos);
					solution = (List<Point>) added.clone();
					System.out.println("[GELÖSCHT] "+wald+": "+added);
					added.removeLast();
				}
				else p.nextPossibilities.add(new Possibility(rv.pos));

				raster.set(rv.pos.x, rv.pos.y, BRAND);
			}
		}

		// Ansonsten rekursiv weiter nach Enden suchen
		else
		{
			List<Possibility> remove = new LinkedList<>();

			for (Possibility p0 : p.nextPossibilities)
			{
				added.add(p0.thisPossibility);
				if (!itSeek0(RasterUtils.clone(raster), p0, added))
					remove.add(p0);
				added.removeLast();
			}

			// Die nicht mehr gebrauchten Möglichkeiten löschen
			for (Possibility p0 : remove)
				p.nextPossibilities.remove(p0);
			if (p.nextPossibilities.size() == 0)
				return false;
		}

		return true;
	}
	/**
```

```
	* Diese Methode sucht alle Brandfelder mit mindestens einem Waldstück in der Umgebung und
	* gibt eine Liste derer zurück.
	*/
	protected List<RasterValue> getBraende (Raster raster)
	{
		List<RasterValue> braende = new LinkedList<>();

		// Jedes Rasterfeld durchsuuchen
		for (int i = 0; i < raster.getSize().width; i++)
		{
			for (int j = 0; j < raster.getSize().height; j++)
			{
				// Brennt das Waldstück überhaupt?
				if (raster.get(i, j) == BRAND)
				{
					// nicht brenndenden Wald in Umgebung suchen
					int wald_in_umgebung = 0;
					if (i > 0)
						if (raster.get(i-1, j) == WALD)
							wald_in_umgebung++;
					if (j > 0)
						if (raster.get(i, j-1) == WALD)
							wald_in_umgebung++;
					if (i < raster.getSize().width-1)
						if (raster.get(i+1, j) == WALD)
							wald_in_umgebung++;
					if (j < raster.getSize().height-1)
						if (raster.get(i, j+1) == WALD)
							wald_in_umgebung++;
					// Brandstück hinzufügen, wenn mindestens 1 Waldstück in der
					// Umgebung liegt
					if (wald_in_umgebung != 0)
						braende.add(new RasterValue(i, j, BRAND, wald_in_umgebung));
				}
			}
		}

		return braende;
	}


	int i = -1;
	public Point delete (Raster raster)
	{
		if (++i >= solution.size()) return null;
		return solution.get(i);
	}
}
```