

# 32. Bundeswettbewerb Informatik, Runde 2

Ausarbeitungen zu den Aufgaben „Buschfeuer“ und „Lebenslinien“

Philip Wellnitz

# Vorwort

Diese Welt ist voller Rätsel und Probleme.

Zwei von ihnen habe ich auf den folgenden Seiten versucht zu lösen, wobei sich mir durchaus neue Probleme in den Weg stellten.

Keine sollten dagegen beim Starten meiner Programme aus einem Linux-artigen Terminal auftreten.

Es sollte jedoch speziell bei meinem Programm zur Lösung von Aufgabe 1 darauf geachtet werden, dass das Terminal ASCII-Escape-Sequenzen unterstützt, damit die ausgegebenen Kunstwerke aka. Lösungen auch richtig dargestellt werden können.

Diese Welt ist voller Rätsel und Probleme. Viele sind lösbar.

# Inhaltsverzeichnis

<b>1 Aufgabe 1 - Buschfeuer</b>	<b>3</b>
1.1 Lösungsidee . . . . .	3
1.1.1 Korrektheit . . . . .	6
1.1.2 Laufzeitanalyse . . . . .	7
1.2 Umsetzung . . . . .	8
1.2.1 Eingabeformat . . . . .	9
1.3 Beispiele . . . . .	10
1.3.1 Beispiel 0 . . . . .	10
1.3.2 Beispiel 1 . . . . .	12
1.3.3 Beispiel 2 . . . . .	16
1.3.4 Beispiel 3 . . . . .	20
1.4 Quelltext . . . . .	25
<b>2 Aufgabe 2 - Lebenslinien</b>	<b>37</b>
2.1 Lösungsidee . . . . .	37
2.1.1 Eigenschaften von Lebensgraphen . . . . .	37
2.2 Umsetzung . . . . .	39
2.3 Beispiele . . . . .	39
2.4 Quelltext . . . . .	39

# 1 Aufgabe 1 - Buschfeuer

## 1.1 Lösungsidee

Ein *Feld* ist ein quadratisches Stück Land, welches genau einen folgender Zustände inne haben kann:

BRENNBAR Das Stück Land ist in der Lage, zu brennen.

BRENNEND Ein brennendes Stück Land.

GELÖSCHT Ein Stück Land, welches nie wieder brennen wird.

LEER Ein leeres Stück Land.

Alle Felder haben die selbe Fläche.

Ein *Wald* ist nun die rechteckig-gitterförmige Anordnung von  $n \times m$  Feldern. Die *Umgebung*  $U(f)$  eines Feldes  $f$  in einem Wald  $W$  ist dabei die Menge an Feldern, welche in  $W$  eine gemeinsame Kante mit  $f$  haben. Wald Umgebung

Der Wald wird nun diskret beobachtet. Es ist dabei sichergestellt, dass nur sofern ein Feld bei einer Beobachtung brennend ist, dieses und jedes brennbare Feld seiner Umgebung bei der nächsten Beobachtung brennen werden, sofern diese nicht schon brennen. Diese Eigenschaft des Waldes sei mit *Feuerausbreitung* bezeichnet.

Ab der 2. Beobachtung kann pro Beobachtung genau 1 (brennendes) Feld gelöscht werden. Wird ein brennendes Feld gelöscht, so fängt seine Umgebung bis zur nächsten Beobachtung nicht an zu brennen.

Die erste Beobachtung, ab der ein Feld  $f$  brennt, heiße *Entflammung* von  $f$ .

Ziel ist es nun, eine Folge von zu löschenden Feldern anzugeben, sodass bei deren Einhaltung die Anzahl der brennenden Felder minimiert wird.

Im Folgenden seien diejenigen Felder, welche bei mindestens 2 Beobachtungen brennend waren, als *verkohlt* bezeichnet.

Nach der Feuerausbreitung muss jedes Feld der Umgebung eines verkohlten Feldes  $c$  brennend sein oder gewesen sein oder seit der Entflammung von  $c$  nicht brennbar gewesen sein.

Sei nun zunächst der Fall betrachtet, dass nur brennende Felder gelöscht werden können.

Es ist leicht zu erkennen, dass es die Lösung nicht verschlechtert, wenn ab der 2. Beobachtung bei jeder Beobachtung 1 brennendes Feld gelöscht wird. Daher wird im Folgenden davon ausgegangen, dass bei jeder Beobachtung (ab der 2.) 1 brennendes Feld gelöscht wird. Es gilt nun also für jede dieser Beobachtungen dasjenige brennende Feld zu finden, durch dessen Löschung die Anzahl der im Folgenden (nicht unbedingt unmittelbar folgend) zu brennen anfangenden Felder minimiert.

Sei nun eine Beobachtung fixiert.

Nun soll für ein brennendes Feld  $F$  ein Maß  $\mu(F)$  dafür gefunden werden, mit dem bestimmt werden kann, welches Feld zum Löschen in obigem Sinne am Besten ist. Sei  $\mu(F)$  daher die Anzahl der brennbaren Felder, zu denen  $F$  das brennende Feld mit dem *kleinsten Abstand* ist. Dieser kürzeste Abstand ist dabei die minimale Anzahl an Beobachtungen, bis das Feld anfängt zu brennen. (Unter der Annahme, dass keine weiteren Felder gelöscht

werden.)

Löscht man nun  $F$ , so wird der kleinste Abstand aller Felder höchstens größer; bei allen Feldern, bei deren kürzestem Abstand  $F$  jedoch keine Rolle spielte (bei denen der Abstand zu einem anderen brennenden Feld also kleiner oder gleich dem Abstand zu  $F$  ist), tritt keine Veränderung auf.

Für 2 Werte  $\mu(F_1)$  und  $\mu(F_2)$  gilt nun: ist  $\mu(F_1) < \mu(F_2)$ , so erzeugte  $F_2$  bei mehr Feldern eine Vergrößerung des kleinsten Abstands als  $F_1$ .

Die *minimale Lebenszeit* eines Feldes sei nun eben der kleinste Abstand zu einem brennenden Feld. Es ist leicht zu erkennen, dass nach mindesten so vielen Beobachtungen, wie die minimale Lebenszeit eines Feldes ist, das Feld zu brennen beginnt.

$\mu(F)$  gibt also auch die Anzahl der Felder an, deren minimale Lebenszeit allein durch  $F$  bestimmt ist. Löscht man  $F$ , so wird, wie schon gesehen, die minimale Lebenszeit aller dieser Felder höchstens größer, es ist also am Besten, dasjenige Feld  $F^*$  zum Löschen auszuwählen, welches  $\mu(\cdot)$  für alle aktuell brennenden Felder maximiert.

Es gilt nun noch  $\mu$  effizient zu bestimmen. Da ein Wald eine rechteckige Gitterform besitzt, ist der kürzeste Abstand zwischen 2 Feldern 1, genau dann, wenn diese Felder eine gemeinsame Kante haben.

Fasse man das Gitter nun als Graphen auf, wobei die Felder die Knoten sind und zwischen 2 Knoten eine Kante ist, genau dann, wenn zwischen diesen Feldern eine Kante ist. Es nun offensichtlich, dass dieser Graph ungewichtet und ungerichtet ist. Somit ist das Finden von kleinsten Abständen mittels einer *Breitensuche* möglich.

Dabei sind die Startfelder der Breitensuche die brennenden Felder. Dabei muss für jedes dieser brennenden Felder eine eigene Breitensuche gestartet werden; wobei für alle Breitensuchen gemeinsam die ermittelten kleinsten Abstände gespeichert werden müssen. Zusätzlich zu den kleinsten Abständen müssen auch die dazugehörigen brennenden Felder gespeichert werden, von denen pro Feld eventuell mehr als 1 existiert. Weiterhin muss die Breitensuche nur brennbare Felder besuchen.

Sind die kleinsten Abstände gefunden, so kann  $\mu$  ermittelt werden, mithilfe simplem durchiterieren über alle Felder und gleichzeitigem Zählen der Felder, für die nur 1 brennendes Feld gespeichert wurde.

In Pseudocode:

```

1  Wald      ; //Der Wald; ein 2D-Container
2
3  AnfangsBrennendeFelder()      { //Ermittelt die von Anfang
    brennenden Felder
4    brennendeFelder := null; //1D-Container für Positionen
    brennender Felder
5    for (i = 0..Wald.Höhe())
6      for (j = 0..Wald.Breite())
7        if (Wald[i,j] == BRENNEND)
8          brennendeFelder.Add((i,j)); //Gefundene Position
          hinzufügen
9
10   return brennendeFelder; //Alle gefundenen Positionen
      zurückgeben
11 }
12
13 NächsteBeobachtung(aktBrennendeFelder) { //Ermittelt die bei der
    nächsten Beobachtung brennenden Felder, aus den Feldern, die
    aktuell brennen

```

```

14     neuBrennendeFelder := null;
15     for all((x;y) from aktBrennendeFelder)
16         if(Wald[x,y] == GELÖSCHT)
17             continue; //Feld kann kein Feuer verteilen
18
19     Wald[x,y] := VERKOHLT; //2 mal brennende Felder sind verkohlt
20     for all((x';y') from Umgebung((x;y)))
21         if(Wald[x',y'] == BRENNBAR)
22             neuBrennendeFelder.Add((x';y')); //Gefundene Position
                hinzufügen
23             Wald[x',y'] := BRENNEND; //Wald beginnt zu brennen
24
25     return neuBrennendeFelder;
26 }
27
28 GetOptBewässerungspunkt(aktBrennendeFelder) { //Ermittelt den
    besten Bewässerungspunkt
29     kleinsterAbstand := null; //Speichert für alle Felder des
        Waldes den kleinsten Abstand zu jedem Feld aus
        aktBrennendeFelder
30
31     for(i = 0..kleinsterAbstand.Size())
32         Fülle kleinsterAbstand[i] mithilfe einer Breitensuche
33
34     anzEindeutigKleinstAbstände := null;
35
36     for (i = 0..Wald.Höhe())
37         for (j = 0..Wald.Breite())
38             if(Es ex. k mit kleinsterAbstand[k][i,j] eindeutiges
                Minimum für alle mögliche k)
39                 anzEindeutigKleinstAbstände[k]++;
40
41     return aktBrennendeFelder[k, sodass
        anzEindeutigKleinstAbstände[k] maximal];
42 }
43
44 SimuliereFeuer() { //Die eigentliche Berechnung
45     aktBrennendeFelder := AnfangsBrennendeFelder(); //Anfangs
        interessante Felder; Kann brennende, von Feuer umschlossene
        Felder beinhalten
46     while(!aktBrennendeFelder.Empty()) //Solange es brennende
        Felder gibt
47         aktBrennendeFelder := NächsteBeobachtung(
            aktBrennendeFelder) //Ermittle die bei nächster
            Beobachtung brennenden Felder
48         if(aktBrennendeFelder.Empty())
49             break; //Keine Felder brennen mehr
50
51     Wald[GetOptBewässerungspunkt(aktBrennendeFelder)] := GELÖSCHT;
        //Lösche das aktuell beste Feld
52 }

```

### 1.1.1 Korrektheit

Wie schon beschrieben, wird bei jeder Beobachtung das für diese Beobachtung nach  $\mu$  beste Feld zum Löschen ausgewählt.

Es gilt also zu zeigen, dass insgesamt nicht weniger Felder abbrennen, sollte bei einer Beobachtung nicht das für diese Beobachtung nach  $\mu$  optimalste Feld gelöscht werden.

Es lässt sich jedoch ein einfaches Beispiel konstruieren, indem eben dies der Fall ist; eine bessere Lösung also gefunden werden kann, wird nicht das nach  $\mu$  optimalste Feld gelöscht:

Die Löschung nach dem Algorithmus:

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	WA	FO	WA	FO	FO	FO	FO
FO	FO	FO	WA	BU	WA	FO	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	FO	WA	WA	WA	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 1: Water spot (4|3)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	WA	01	WA	FO	FO	FO	FO
FO	FO	FO	WA	CO	WA	FO	FO	FO	FO
FO	FO	WA	FO	BU	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	FO	WA	WA	WA	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 2: Water spot (4|6)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	WA	01	WA	FO	FO	FO	FO
FO	FO	FO	WA	CO	WA	FO	FO	FO	FO
FO	FO	WA	BU	CO	BU	WA	FO	FO	FO
FO	FO	WA	FO	02	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	FO	WA	WA	WA	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 3: Water spot (3|6)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	WA	01	WA	FO	FO	FO	FO
FO	FO	FO	WA	CO	WA	FO	FO	FO	FO
FO	FO	WA	CO	CO	CO	WA	FO	FO	FO
FO	FO	WA	03	02	BU	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	FO	WA	WA	WA	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 4: Water spot (5|7)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	WA	01	WA	FO	FO	FO	FO
FO	FO	FO	WA	CO	WA	FO	FO	FO	FO
FO	FO	WA	CO	CO	CO	WA	FO	FO	FO
FO	FO	WA	03	02	CO	WA	FO	FO	FO
FO	FO	WA	FO	FO	04	WA	FO	FO	FO
FO	FO	FO	WA	WA	WA	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- And you'll find 5 pieces of coal  
and 4 pieces of watered coal

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	WA	01	WA	FO	FO	FO	FO
FO	FO	FO	WA	CO	WA	FO	FO	FO	FO
FO	FO	WA	CO	CO	CO	WA	FO	FO	FO
FO	FO	WA	03	02	CO	WA	FO	FO	FO
FO	FO	WA	FO	FO	04	WA	FO	FO	FO
FO	FO	FO	WA	WA	WA	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

Explanation:

WA --- EMPTY  
 FO --- BURNABLE  
 BU --- BURNED  
 CO --- COAL (doubly burned)  
 ## --- WATERED at time ##  
 Fields can have more than 1 state.

Eine bessere Löschung:

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	WA	FO	WA	FO	FO	FO	FO
FO	FO	FO	WA	BU	WA	FO	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	FO	WA	WA	WA	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 1: Water spot (4|5)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	WA	BU	WA	FO	FO	FO	FO
FO	FO	FO	WA	CO	WA	FO	FO	FO	FO
FO	FO	WA	FO	01	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	FO	WA	WA	WA	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 2: Water spot (4|2)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	02	FO	FO	FO	FO	FO
FO	FO	FO	WA	CO	WA	FO	FO	FO	FO
FO	FO	FO	WA	CO	WA	FO	FO	FO	FO
FO	FO	WA	FO	01	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	FO	WA	WA	WA	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- And you'll find 2 pieces of coal  
and 2 pieces of watered coal

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	02	FO	FO	FO	FO	FO
FO	FO	FO	WA	CO	WA	FO	FO	FO	FO
FO	FO	FO	WA	CO	WA	FO	FO	FO	FO
FO	FO	WA	FO	01	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	WA	FO	FO	FO	WA	FO	FO	FO
FO	FO	FO	WA	WA	WA	FO	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

Explanation:

WA --- EMPTY  
 FO --- BURNABLE  
 BU --- BURNED  
 CO --- COAL (doubly burned)  
 ## --- WATERED at time ##

Fields can have more than 1 state.

Der Algorithmus ist also nicht optimal, es handelt sich um eine Heuristik. Dabei liefert sie bei vielen Eingaben *ziemlich* gute Ergebnisse<sup>1</sup>. Zum Vergleich habe ich den Brute-Force-Ansatz implementiert, der garantiert optimale Lösungen liefert.

### 1.1.2 Laufzeitanalyse

Der Brute-Force-Ansatz probiert alle Möglichkeiten an verschiedenen Lösungen und wählt die optimalste. Grob überschlagen gibt es für jede Löschung 4 Möglichkeiten, somit ergibt sich eine grobe obere Schranke für den Worst-Case von  $\mathcal{O}(4^b)$ , mit  $b$  der Anzahl der Lösungen der Lösung<sup>2</sup>. Dieser Ansatz hat also eine exponentielle Laufzeit, im Gegensatz zu der Heuristik wie im Folgenden gezeigt wird.

<sup>1</sup>Siehe dazu Sektion Beispiele

<sup>2</sup>Es gibt wohl Pfade im Suchbaum, die länger als  $b$  sind; durch geschicktes Pruning ist diese Schranke jedoch einhaltbar

Eine Breitensuche hat eine Laufzeit von  $\mathcal{O}(V + E)$  in einem Graphen mit  $E$  Kanten und  $V$  Knoten. Speziell hat der Graph bei dieser Aufgabe  $n \cdot m$  Knoten und  $(n - 1) \cdot (m - 1)$  Kanten.

Eine Breitensuche wird nach obigem Algorithmus bei jeder der insgesamt  $b$  Beobachtungen  $f(b_i)$ -mal benötigt, wobei  $f(b_i)$  die Anzahl der zu betrachtenden brennenden Felder bei Beobachtung  $b_i$  sei.

Eine Breitensuche besucht nach obigem Algorithmus höchstens  $n \cdot m - f(b_i)$  Felder; die Breitensuchen haben also eine Laufzeit von  $\mathcal{O}(f(b_i) \cdot (2 \cdot n \cdot m - f(b_i)))$ . Es ist leicht zu erkennen, dass die Funktion  $F(x) = x(a - x)$  das Maximum an der Stelle  $x_{\max} = \frac{a}{2}$  hat.

Somit gilt  $\mathcal{O}(f(b_i) \cdot (2 \cdot n \cdot m - f(b_i))) = \mathcal{O}(\frac{nm}{2}(2nm - \frac{nm}{2})) = \mathcal{O}(\frac{3n^2m^2}{4}) = \mathcal{O}(n^2m^2)$ . Es ergibt sich eine Gesamtlaufzeit von  $\mathcal{O}(n^2 \cdot m^2 \cdot b)$ . Mit  $b = \mathcal{O}(n \cdot m)$  ergibt sich eine (wohl sehr grobe) obere Schranke der Laufzeit von  $\mathcal{O}(n^3 \cdot m^3)$ .

Mit diesem Algorithmus lassen sich also Lösungen für Wälder gut berechnen, deren Dimensionen 200 nicht überschreiten, bei denen also  $\max n, m \leq 200$ .

## 1.2 Umsetzung

Für die Umsetzung habe ich die Sprache C++ verwendet. Dabei habe ich sowohl den Brute-Force-Ansatz als auch die Heuristik implementiert.

Zunächst habe ich mir für Wälder eine Klasse `Woods` geschrieben. Deren Deklaration findet sich in der Datei `Woods.h`, die Implementierung in der Datei `Woods.cpp`. Jeder Wald hat dabei eine Breite (`Width`) und eine Höhe (`Height`).

Dabei benutzen Wälder für die Representierung eines Feldes einen `FIELDSTATE`, welcher als `char` definiert ist.<sup>3</sup> Dabei kann ein `FIELDSTATE` einen oder mehrere, ebenfalls definierter, Zustände annehmen. Dabei handelt es sich um die in der Lösungsidee beschriebenen Zustände eines Feldes, `EMPTY`, `BURNABLE`, `BURNED`, `WATERED` und `COAL`.

Ein Wald hält sich nun ein 2-dimensional, variabel großes Feld von `FIELDSTATEs`, der eigentliche Wald.

Durch geschickte Operatorenüberladung und geeignete Akzessormethoden können diese Attribute vollständig gekapselt werden.

Der eigentliche Algorithmus findet sich in der Datei `Buschfeuer.cpp`; die Ein- und Ausgabe steht in der Datei `Buschfeuer.h`.

Das Lesen der Eingabe übernimmt die Prozedur `parseInput`, welche die Daten in eine globale Instanz der Klasse `Woods Forest` einliest.

Ist die Eingabe gelesen, werden aus dieser die zu Beginn brennenden Felder mithilfe der Funktion `getInitialBurningFields` ermittelt und dann gleich an die Prozedur `simulateFire` weitergereicht. Diese Prozedur `simulateFire` simuliert nun das Feuer und ermittelt die zu löschenden Felder unter Zuhilfenahme der Funktion `getOptimalWaterSpot`. Dabei wird nach jedem Löschvorgang eine Ausgabe getätigt, welche die zu löschende Position (oben links mit (0—0) beginnend) ausgibt. Auch wird unter Verwendung von ASCII-Escape-Sequenzen ein Bild in der Konsole angezeigt, welches den Wald darstellt.

Ist das Feuer gelöscht (kann es sich also nicht weiter ausbreiten), wird dem Nutzer eine Meldung ausgegeben, wie viele Felder verbrannten und wie viele Felder verbrannt und gelöscht wurden. (Diese beiden Zahlen beschreiben disjunkte Mengen.) Auch hier wird wieder ein Bild erzeugt und ausgegeben.

<sup>3</sup>Das Wort „definiert“ ist durchaus ernst zu nehmen, da es hier beschreiben soll, dass etwas mittels `#define` „gelöst“ wurde.



### 1.2.1 Eingabeformat

Wird mein Programm über ein Terminal gestartet, so können ihm bis zu 2 Kommandozeilenparameter übergeben werden:

Arg. 1 Pfad zu einer Datei mit einer Eingabe

Arg. 2 Pfad zu einer Datei für eine Ausgabe; existierende Dateien werden überschrieben.  
Dabei gibt die Dateierweiterung dieser Datei das Verhalten meines Programmes an:

## 1.3 Beispiele

### 1.3.1 Beispiel 0

Die ist das Beispiel aus der Aufgabenstellung. Umgewandelt für mein Programm sieht diese Eingabe folgendermaßen aus<sup>4</sup>:

```

1 10 10
2 1101111101
3 1001111110
4 1111111111
5 1100010001
6 1111131111
7 1100111111
8 1111011011
9 0111011010
10 1011011011
11 1111111111

```

Die Heuristik produziert folgende Ausgabe<sup>56</sup>:

FO	FO	WA	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	FO	FO	FO	FO	FO	FO	WA
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	FO	WA	WA	WA	FO
FO	FO	FO	FO	FO	BU	FO	FO	FO	FO
FO	FO	WA	WA	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	WA	FO	FO	WA	FO	FO
WA	FO	FO	FO	WA	FO	FO	WA	FO	WA
FO	WA	FO	FO	WA	FO	FO	WA	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 1: Water spot (5|3)

FO	FO	WA	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	FO	FO	FO	FO	FO	FO	WA
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	O1	WA	WA	WA	FO
FO	FO	FO	FO	BU	CO	BU	FO	FO	FO
FO	FO	WA	WA	FO	BU	FO	FO	FO	FO
FO	FO	FO	FO	WA	FO	FO	WA	FO	FO
WA	FO	FO	FO	WA	FO	FO	WA	FO	WA
FO	WA	FO	FO	WA	FO	FO	WA	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 2: Water spot (3|4)

<sup>4</sup>Diese Eingabe finden Sie auch in der Datei 0.in

<sup>5</sup>Diese Ausgabe finden Sie auch in der Datei 0.out.tex; Eine Datei 0.out mit den ASCII-Escape-Sequenzen existiert ebenfalls.

<sup>6</sup>Um die ASCII-Escape-Sequenzen in T<sub>E</sub>X korrekt darzustellen, habe ich spezielle Ausgabemethoden geschrieben. Diese produzieren anstatt der ASCII-Sequenzen T<sub>E</sub>X-Befehle, welche optisch zu ähnlichen Ergebnissen führen.

FO	FO	WA	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	FO	FO	FO	FO	FO	FO	WA
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	FO	02	CO	CO	CO	BU	FO	FO
FO	FO	WA	WA	BU	CO	BU	FO	FO	FO
FO	FO	FO	FO	WA	BU	FO	WA	FO	FO
WA	FO	FO	FO	WA	FO	FO	WA	FO	WA
FO	WA	FO	FO	WA	FO	FO	WA	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 3: Water spot (8|4)

FO	FO	WA	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	FO	FO	FO	FO	FO	FO	WA
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	FO	02	CO	CO	CO	CO	03	FO
FO	FO	WA	WA	CO	CO	CO	BU	FO	FO
FO	FO	FO	FO	WA	CO	BU	WA	FO	FO
WA	FO	FO	FO	WA	BU	FO	WA	FO	WA
FO	WA	FO	FO	WA	FO	FO	WA	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 4: Water spot (8|5)

FO	FO	WA	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	FO	FO	FO	FO	FO	FO	WA
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	FO	02	CO	CO	CO	CO	03	FO
FO	FO	WA	WA	CO	CO	CO	CO	04	FO
FO	FO	FO	FO	WA	CO	CO	WA	FO	FO
WA	FO	FO	FO	WA	CO	BU	WA	FO	WA
FO	WA	FO	FO	WA	BU	FO	WA	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 5: Water spot (5|9)

FO	FO	WA	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	FO	FO	FO	FO	FO	FO	WA
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	FO	02	CO	CO	CO	CO	03	FO
FO	FO	WA	WA	CO	CO	CO	CO	04	FO
FO	FO	FO	FO	WA	CO	CO	WA	FO	FO
WA	FO	FO	FO	WA	CO	CO	WA	FO	WA
FO	WA	FO	FO	WA	CO	BU	WA	FO	FO
FO	FO	FO	FO	FO	05	FO	FO	FO	FO

--- At time 6: Water spot (6|9)

FO	FO	WA	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	FO	FO	FO	FO	FO	FO	WA
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	FO	02	CO	CO	CO	CO	03	FO
FO	FO	WA	WA	CO	CO	CO	CO	04	FO
FO	FO	FO	FO	WA	CO	CO	WA	FO	FO
WA	FO	FO	FO	WA	CO	CO	WA	FO	WA
FO	WA	FO	FO	WA	CO	CO	WA	FO	FO
FO	FO	FO	FO	FO	05	06	FO	FO	FO

--- And you'll find 14 pieces of coal and 6 pieces of watered coal

FO	FO	WA	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	FO	FO	FO	FO	FO	FO	WA
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	FO	02	CO	CO	CO	CO	03	FO
FO	FO	WA	WA	CO	CO	CO	CO	04	FO
FO	FO	FO	FO	WA	CO	CO	WA	FO	FO
WA	FO	FO	FO	WA	CO	CO	WA	FO	WA
FO	WA	FO	FO	WA	CO	CO	WA	FO	FO
FO	FO	FO	FO	FO	05	06	FO	FO	FO

Explanation:

WA --- EMPTY

FO --- BURNABLE

BU --- BURNED

CO --- COAL (doubly burned)

## --- WATERED at time ##

Fields can have more than 1 state.

Diese Ausgabe deckt sich auch mit der des Brute-Force-Ansatzes, weshalb ich dessen Ausgabe hier weglassen.

### 1.3.2 Beispiel 1

Eine Situation mit mehr als einem Feuer bei der ersten Beobachtung<sup>7</sup>:

```

1 10 11
2 3101111101
3 1001111110
4 1111111111
5 1100010001
6 1111131111
7 1100111111
8 1111011011

```

<sup>7</sup>Diese Eingabe finden Sie auch in der Datei 1.in

```

9  0111011010
10 1011011011
11 1111113111
12 1111111111

```

Mein Programm produziert folgende Ausgabe<sup>8</sup>:

BU	FO	WA	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	FO	FO	FO	FO	FO	FO	WA
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	FO	WA	WA	WA	FO
FO	FO	FO	FO	FO	BU	FO	FO	FO	FO
FO	FO	WA	WA	FO	FO	FO	FO	FO	FO
FO	FO	FO	FO	WA	FO	FO	WA	FO	FO
WA	FO	FO	FO	WA	FO	FO	WA	FO	WA
FO	WA	FO	FO	WA	FO	FO	WA	FO	FO
FO	FO	FO	FO	FO	FO	BU	FO	FO	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 1: Water spot (5|3)

CO	BU	WA	FO	FO	FO	FO	FO	WA	FO
BU	WA	WA	FO	FO	FO	FO	FO	FO	WA
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	O1	WA	WA	WA	FO
FO	FO	FO	FO	BU	CO	BU	FO	FO	FO
FO	FO	WA	WA	FO	BU	FO	FO	FO	FO
FO	FO	FO	FO	WA	FO	FO	WA	FO	FO
WA	FO	FO	FO	WA	FO	FO	WA	FO	WA
FO	WA	FO	FO	WA	FO	BU	WA	FO	FO
FO	FO	FO	FO	FO	BU	CO	BU	FO	FO
FO	FO	FO	FO	FO	BU	FO	FO	FO	FO

--- At time 2: Water spot (0|2)

CO	CO	WA	FO	FO	FO	FO	FO	WA	FO
CO	WA	WA	FO	FO	FO	FO	FO	FO	WA
O2	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	O1	WA	WA	WA	FO
FO	FO	FO	BU	CO	CO	BU	FO	FO	FO
FO	FO	WA	WA	BU	CO	BU	FO	FO	FO
FO	FO	FO	FO	WA	BU	FO	WA	FO	FO
WA	FO	FO	FO	WA	FO	BU	WA	FO	WA
FO	WA	FO	FO	WA	BU	CO	WA	FO	FO
FO	FO	FO	FO	BU	CO	CO	CO	BU	FO
FO	FO	FO	FO	FO	BU	CO	BU	FO	FO

--- At time 3: Water spot (2|4)

<sup>8</sup>Diese Ausgabe finden Sie auch in der Datei 1.out.tex; Eine Datei 1.out mit den ASCII-Escape-Sequenzen existiert ebenfalls.

CO	CO	WA	FO	FO	FO	FO	FO	WA	FO
CO	WA	WA	FO	FO	FO	FO	FO	FO	WA
02	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	03	CO	CO	CO	CO	CO	BU	FO
FO	FO	WA	WA	CO	CO	CO	BU	FO	FO
FO	FO	FO	FO	WA	CO	BU	WA	FO	FO
WA	FO	FO	FO	WA	BU	CO	WA	FO	WA
FO	WA	FO	FO	WA	CO	CO	WA	BU	FO
FO	FO	FO	BU	CO	CO	CO	CO	CO	BU
FO	FO	FO	FO	BU	CO	CO	CO	BU	FO

--- At time 4: Water spot (9|4)

CO	CO	WA	FO	FO	FO	FO	FO	WA	FO
CO	WA	WA	FO	FO	FO	FO	FO	FO	WA
02	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	03	CO	CO	CO	CO	CO	CO	04
FO	FO	WA	WA	CO	CO	CO	CO	BU	FO
FO	FO	FO	FO	WA	CO	CO	WA	FO	FO
WA	FO	FO	FO	WA	CO	CO	WA	BU	WA
FO	WA	FO	BU	WA	CO	CO	WA	CO	BU
FO	FO	BU	CO	CO	CO	CO	CO	CO	CO
FO	FO	FO	BU	CO	CO	CO	CO	CO	BU

--- At time 5: Water spot (1|9)

CO	CO	WA	FO	FO	FO	FO	FO	WA	FO
CO	WA	WA	FO	FO	FO	FO	FO	FO	WA
02	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	03	CO	CO	CO	CO	CO	CO	04
FO	FO	WA	WA	CO	CO	CO	CO	CO	BU
FO	FO	FO	FO	WA	CO	CO	WA	BU	FO
WA	FO	FO	BU	WA	CO	CO	WA	CO	WA
FO	WA	BU	CO	WA	CO	CO	WA	CO	CO
FO	05	CO	CO	CO	CO	CO	CO	CO	CO
FO	FO	BU	CO	CO	CO	CO	CO	CO	CO

--- At time 6: Water spot (1|10)

CO	CO	WA	FO	FO	FO	FO	FO	WA	FO
CO	WA	WA	FO	FO	FO	FO	FO	FO	WA
02	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	03	CO	CO	CO	CO	CO	CO	04
FO	FO	WA	WA	CO	CO	CO	CO	CO	CO
FO	FO	FO	BU	WA	CO	CO	WA	CO	BU
WA	FO	BU	CO	WA	CO	CO	WA	CO	WA
FO	WA	CO	CO	WA	CO	CO	WA	CO	CO
FO	05	CO	CO	CO	CO	CO	CO	CO	CO
FO	06	CO	CO	CO	CO	CO	CO	CO	CO

--- At time 7: Water spot (2|6)

CO	CO	WA	FO	FO	FO	FO	FO	WA	FO
CO	WA	WA	FO	FO	FO	FO	FO	FO	WA
02	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	03	CO	CO	CO	CO	CO	CO	04
FO	FO	WA	WA	CO	CO	CO	CO	CO	CO
FO	FO	07	CO	WA	CO	CO	WA	CO	CO
WA	BU	CO	CO	WA	CO	CO	WA	CO	WA
FO	WA	CO	CO	WA	CO	CO	WA	CO	CO
FO	05	CO	CO	CO	CO	CO	CO	CO	CO
FO	06	CO	CO	CO	CO	CO	CO	CO	CO

--- At time 8: Water spot (1|6)

CO	CO	WA	FO	FO	FO	FO	FO	WA	FO
CO	WA	WA	FO	FO	FO	FO	FO	FO	WA
02	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	03	CO	CO	CO	CO	CO	CO	04
FO	FO	WA	WA	CO	CO	CO	CO	CO	CO
FO	08	07	CO	WA	CO	CO	WA	CO	CO
WA	CO	CO	CO	WA	CO	CO	WA	CO	WA
FO	WA	CO	CO	WA	CO	CO	WA	CO	CO
FO	05	CO	CO	CO	CO	CO	CO	CO	CO
FO	06	CO	CO	CO	CO	CO	CO	CO	CO

--- And you'll find 48 pieces of coal and 8 pieces of watered coal

CO	CO	WA	FO	FO	FO	FO	FO	WA	FO
CO	WA	WA	FO	FO	FO	FO	FO	FO	WA
02	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	WA	WA	WA	FO
FO	FO	03	CO	CO	CO	CO	CO	CO	04
FO	FO	WA	WA	CO	CO	CO	CO	CO	CO
FO	08	07	CO	WA	CO	CO	WA	CO	CO
WA	CO	CO	CO	WA	CO	CO	WA	CO	WA
FO	WA	CO	CO	WA	CO	CO	WA	CO	CO
FO	05	CO	CO	CO	CO	CO	CO	CO	CO
FO	06	CO	CO	CO	CO	CO	CO	CO	CO

Explanation:

WA --- EMPTY

FO --- BURNABLE

BU --- BURNED

CO --- COAL (doubly burned)

## --- WATERED at time ##

Fields can have more than 1 state.

### 1.3.3 Beispiel 2

9:

```

1  13 13
2  11111111111111
3  1000001000001
4  10111111111101
5  10111111111101
6  10111111111101
7  10111111111101
8  11111131111111
9  10111111111101
10 10111111111101
11 10111111111101
12 10111111111101
13 1000001000001
14 11111111111111

```

Mein Programm produziert folgende Ausgabe<sup>10</sup>:

<sup>9</sup>Diese Eingabe finden Sie auch in der Datei 2.in

<sup>10</sup>Diese Ausgabe finden Sie auch in der Datei 2.out.tex; Eine Datei 2.out mit den ASCII-Escape-Sequenzen existiert ebenfalls.



FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	FO	FO	FO	FO	FO	BU	FO	FO	FO	FO	FO	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 1: Water spot (7|6)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	BU	FO	FO	FO	FO	WA	FO
FO	FO	FO	FO	FO	BU	CO	O1	FO	FO	FO	FO	FO
FO	WA	FO	FO	FO	FO	BU	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 2: Water spot (6|8)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	BU	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	BU	CO	BU	FO	FO	FO	WA	FO
FO	FO	FO	FO	BU	CO	CO	O1	FO	FO	FO	FO	FO
FO	WA	FO	FO	FO	BU	CO	BU	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	O2	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 3: Water spot (3|6)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	BU	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	BU	CO	BU	FO	FO	FO	WA	FO
FO	WA	FO	FO	BU	CO	CO	CO	BU	FO	FO	WA	FO
FO	FO	FO	03	CO	CO	CO	01	FO	FO	FO	FO	FO
FO	WA	FO	FO	BU	CO	CO	CO	BU	FO	FO	WA	FO
FO	WA	FO	FO	FO	BU	02	BU	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 4: Water spot (6|2)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	WA	FO	FO	FO	FO	04	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	BU	CO	BU	FO	FO	FO	WA	FO
FO	WA	FO	FO	BU	CO	CO	CO	BU	FO	FO	WA	FO
FO	WA	FO	BU	CO	CO	CO	CO	CO	BU	FO	WA	FO
FO	FO	FO	03	CO	CO	CO	01	BU	FO	FO	FO	FO
FO	WA	FO	BU	CO	CO	CO	CO	BU	FO	WA	FO	FO
FO	WA	FO	FO	BU	CO	02	CO	BU	FO	FO	WA	FO
FO	WA	FO	FO	FO	BU	FO	BU	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 5: Water spot (10|7)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	WA	FO	FO	FO	BU	04	BU	FO	FO	FO	WA	FO
FO	WA	FO	FO	BU	CO	CO	CO	BU	FO	FO	WA	FO
FO	WA	FO	BU	CO	CO	CO	CO	CO	BU	FO	WA	FO
FO	WA	BU	CO	CO	CO	CO	CO	CO	CO	BU	WA	FO
FO	FO	FO	03	CO	CO	CO	01	CO	BU	FO	FO	FO
FO	WA	BU	CO	CO	CO	CO	CO	CO	05	WA	FO	FO
FO	WA	FO	BU	CO	CO	02	CO	CO	BU	FO	WA	FO
FO	WA	FO	FO	BU	CO	BU	CO	BU	FO	FO	WA	FO
FO	WA	FO	FO	FO	BU	FO	BU	FO	FO	FO	WA	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 6: Water spot (10|6)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	WA	FO	FO	BU	CO	04	CO	BU	FO	FO	WA	FO
FO	WA	FO	BU	CO	CO	CO	CO	CO	BU	FO	WA	FO
FO	WA	BU	CO	CO	CO	CO	CO	CO	CO	BU	WA	FO
FO	WA	CO	CO	CO	CO	CO	CO	CO	CO	CO	WA	FO
FO	FO	BU	03	CO	CO	CO	01	CO	CO	06	FO	FO
FO	WA	CO	CO	CO	CO	CO	CO	CO	CO	05	WA	FO
FO	WA	BU	CO	CO	CO	02	CO	CO	CO	BU	WA	FO
FO	WA	FO	BU	CO	CO	CO	CO	CO	BU	FO	WA	FO
FO	WA	FO	FO	BU	CO	BU	CO	BU	FO	FO	WA	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 7: Water spot (6|11)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	WA	FO	BU	CO	CO	04	CO	CO	BU	FO	WA	FO
FO	WA	BU	CO	CO	CO	CO	CO	CO	CO	BU	WA	FO
FO	WA	CO	CO	CO	CO	CO	CO	CO	CO	CO	WA	FO
FO	WA	CO	CO	CO	CO	CO	CO	CO	CO	CO	WA	FO
FO	BU	CO	03	CO	CO	CO	01	CO	CO	06	FO	FO
FO	WA	CO	CO	CO	CO	CO	CO	CO	CO	05	WA	FO
FO	WA	CO	CO	CO	CO	02	CO	CO	CO	CO	WA	FO
FO	WA	BU	CO	CO	CO	CO	CO	CO	CO	BU	WA	FO
FO	WA	FO	BU	CO	CO	CO	CO	CO	BU	FO	WA	FO
FO	WA	WA	WA	WA	WA	07	WA	WA	WA	WA	WA	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 8: Water spot (0|6)

FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	WA	WA	WA	WA	WA	FO	WA	WA	WA	WA	WA	FO
FO	WA	BU	CO	CO	CO	04	CO	CO	CO	BU	WA	FO
FO	WA	CO	CO	CO	CO	CO	CO	CO	CO	CO	WA	FO
FO	WA	CO	CO	CO	CO	CO	CO	CO	CO	CO	WA	FO
FO	WA	CO	CO	CO	CO	CO	CO	CO	CO	CO	WA	FO
08	CO	CO	03	CO	CO	CO	01	CO	CO	06	FO	FO
FO	WA	CO	CO	CO	CO	CO	CO	CO	CO	05	WA	FO
FO	WA	CO	CO	CO	CO	02	CO	CO	CO	CO	WA	FO
FO	WA	CO	CO	CO	CO	CO	CO	CO	CO	CO	WA	FO
FO	WA	BU	CO	CO	CO	CO	CO	CO	CO	BU	WA	FO
FO	WA	WA	WA	WA	WA	07	WA	WA	WA	WA	WA	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- And you'll find 76 pieces of coal and 8 pieces of watered coal



[illegible]

[illegible]

Mein Programm produziert folgende Ausgabe<sup>12</sup> Dabei hat die Berechnung wenige Sekunden in Anspruch genommen, sofern nicht die Ausgabe der ASCII-Escape-Sequenzen gefordert wird. Dies erhöhte die Laufzeit auf ca. 30s.:

At time 1: Water spot (45|51)  
At time 2: Water spot (45|52)  
At time 3: Water spot (46|50)  
At time 4: Water spot (46|53)  
At time 5: Water spot (47|49)  
At time 6: Water spot (47|54)  
At time 7: Water spot (48|48)  
At time 8: Water spot (48|55)  
At time 9: Water spot (49|47)  
At time 10: Water spot (49|56)  
At time 11: Water spot (50|46)  
At time 12: Water spot (50|57)  
At time 13: Water spot (51|45)  
At time 14: Water spot (51|58)  
At time 15: Water spot (52|44)

---

<sup>12</sup>Diese Ausgabe finden Sie auch in der Datei `3.out.tex2`:

At time 16: Water spot (52|59)  
At time 17: Water spot (53|43)  
At time 18: Water spot (53|60)  
At time 19: Water spot (54|42)  
At time 20: Water spot (54|61)  
At time 21: Water spot (55|41)  
At time 22: Water spot (55|62)  
At time 23: Water spot (56|40)  
At time 24: Water spot (56|63)  
At time 25: Water spot (57|39)  
At time 26: Water spot (57|64)  
At time 27: Water spot (58|38)  
At time 28: Water spot (58|65)  
At time 29: Water spot (59|37)  
At time 30: Water spot (59|66)  
At time 31: Water spot (60|36)  
At time 32: Water spot (60|67)  
At time 33: Water spot (61|35)  
At time 34: Water spot (61|68)  
At time 35: Water spot (62|34)  
At time 36: Water spot (62|69)  
At time 37: Water spot (63|33)  
At time 38: Water spot (63|70)  
At time 39: Water spot (64|32)  
At time 40: Water spot (64|71)  
At time 41: Water spot (65|31)  
At time 42: Water spot (65|72)  
At time 43: Water spot (66|30)  
At time 44: Water spot (66|73)  
At time 45: Water spot (67|29)  
At time 46: Water spot (67|74)  
At time 47: Water spot (68|28)  
At time 48: Water spot (68|75)  
At time 49: Water spot (69|27)  
At time 50: Water spot (69|76)  
At time 51: Water spot (70|26)  
At time 52: Water spot (70|77)  
At time 53: Water spot (71|25)

At time 54: Water spot (71|78)  
At time 55: Water spot (72|24)  
At time 56: Water spot (72|79)  
At time 57: Water spot (73|23)  
At time 58: Water spot (73|80)  
At time 59: Water spot (74|22)  
At time 60: Water spot (74|81)  
At time 61: Water spot (75|21)  
At time 62: Water spot (75|82)  
At time 63: Water spot (76|20)  
At time 64: Water spot (76|83)  
At time 65: Water spot (77|19)  
At time 66: Water spot (77|84)  
At time 67: Water spot (78|18)  
At time 68: Water spot (78|85)  
At time 69: Water spot (79|17)  
At time 70: Water spot (79|86)  
At time 71: Water spot (80|16)  
At time 72: Water spot (80|87)  
At time 73: Water spot (81|15)  
At time 74: Water spot (81|88)  
At time 75: Water spot (82|14)  
At time 76: Water spot (82|89)  
At time 77: Water spot (83|13)  
At time 78: Water spot (83|90)  
At time 79: Water spot (84|12)  
At time 80: Water spot (84|91)  
At time 81: Water spot (85|11)  
At time 82: Water spot (85|92)  
At time 83: Water spot (86|10)  
At time 84: Water spot (86|93)  
At time 85: Water spot (87|9)  
At time 86: Water spot (87|94)  
At time 87: Water spot (88|8)  
At time 88: Water spot (88|95)  
At time 89: Water spot (89|7)  
At time 90: Water spot (89|96)  
At time 91: Water spot (90|6)



At time 92: Water spot (90|97)

At time 93: Water spot (91|5)

At time 94: Water spot (91|98)

At time 95: Water spot (92|4)

At time 96: Water spot (92|99)

At time 97: Water spot (93|3)

At time 98: Water spot (93|2)

At time 99: Water spot (93|1)

At time 100: Water spot (93|0)

And you'll find 6948 pieces of coal and 100 pieces of watered coal

## 1.4 Quelltext

### Woods.h

```
1 #include <vector>
2
3 #define FIELDSTATE      char
4 #define EMPTY          0
5 #define BURNABLE        1
6 #define BURNED          2
7 #define WATERED         4
8 #define COAL            8
9
10 class Woods{
11     private:
12         int Width, Height;
13         std::vector<std::vector<FIELDSTATE> > Fields;
14
15     public:
16         Woods(int width, int height);
17
18         int width() const;
19         int height() const;
20
21         FIELDSTATE& operator() (int x, int y);
22         FIELDSTATE operator() (int x, int y) const;
23
24         bool operator==(const Woods& o) const;
25 };
```

### Woods.cpp

```
1 #include <vector>
2 #include <cstdio>
3
4 #include "Woods.h"
5
6 FIELDSTATE ERROR = -1;
```

```

7
8 Woods::Woods(int width, int height) : Width(width),
    Height(height) {
9     Fields.assign(height, std::vector<FIELDSTATE>(width, 0));
10 }
11
12 int Woods::width() const { return this->Width; }
13 int Woods::height() const { return this->Height; }
14
15 FIELDSTATE& Woods::operator() (int x, int y) {
16     if (x < 0 || y < 0 || x >= width() || y >= height()){
17         printf("OUT OF BOUNDS 1");
18         return ERROR;
19     }
20     return this->Fields[y][x];
21 }
22 FIELDSTATE Woods::operator() (int x, int y) const {
23     if (x < 0 || y < 0 || x >= width() || y >= height()){
24         printf("OUT OF BOUNDS 2");
25         return ERROR;
26     }
27     return this->Fields[y][x];
28 }
29
30 bool Woods::operator==(const Woods& o) const{
31     if(o.width() != width() || o.height() != height()){
32         return false;
33     }
34     for(int i = 0; i < width(); ++i)
35         for(int j = 0; j < height(); ++j)
36             if(o(i,j) != Fields[i][j])
37                 return false;
38     return true;
39 }

```

**Buschfeuer.h** Dies ist die Ein- und Ausgabe; sowie einige Definitionen.

```

1 #include <cstdio>
2 #include <vector>
3
4 #include "Woods.h"
5
6 typedef std::pair<int,int> PII;
7
8 const int oo = (1 << 29); //
    The infinity
9 Woods Forest(0, 0);
10
11 struct Point {
12 public:
13     int x, y;
14     Point(int _x,int _y) : x(_x), y(_y) { }
15 };

```

```
16 int dir[4][2] = {{1,0},{0,1},{-1,0},{0,-1}};
17
18 std::vector<Point> Solution;
19 std::FILE* OUT;
    // The file to mirror the output to
20 void (*printSolution)(std::FILE*, bool);
21
22 //BEGIN OF INPUT
23 void parseInput(std::FILE* f) {
24     int acFieldWidth, acFieldHeight;
25     std::fscanf(f, "%i %i\n",&acFieldWidth, &acFieldHeight);
26
27     Forest = Woods(acFieldWidth, acFieldHeight);
28
29     for(int i = 0; i < acFieldHeight; ++i){
30         for(int j= 0; j < acFieldWidth; ++j){
31             char c;
32             std::fscanf(f, "%c",&c);
33             c -= '0';
34             Forest(j, i) = (FIELDSTATE) c;
35         }
36         if(i < acFieldHeight-1)
37             std::fscanf(f, "\n");
38     }
39 }
40 //END OF INPUT
41 //BEGIN OF OUTPUT
42 void printSolution_TEX(std::FILE* f, bool finalOut) {
43     std::fprintf(f, "\\\n");
44
45     std::fprintf(f, "\\begin{tikzpicture}\n");
46     std::fprintf(f, "\\tikzset{square matrix/.style={\n");
47     std::fprintf(f, "matrix of nodes,\n");
48     std::fprintf(f, "column sep=-\\pgflinewidth, row\n");
49     std::fprintf(f, "sep=-\\pgflinewidth,\n");
50     std::fprintf(f, "nodes={draw,\n");
51     std::fprintf(f, "minimum height=#1,\n");
52     std::fprintf(f, "anchor=center,\n");
53     std::fprintf(f, "text width=#1,\n");
54     std::fprintf(f, "align=center,\n");
55     std::fprintf(f, "inner sep=0pt\n");
56     std::fprintf(f, "},\n");
57     std::fprintf(f, "square matrix/.default=1.2cm\n");
58     std::fprintf(f, "}\n");
59
60     std::fprintf(f, "\\matrix[square matrix=1.4em] {\n");
61     for(int j= 0; j < Forest.height(); ++j) {
62         for(int i= 0; i < Forest.width(); ++i) {
63             if(i)
64                 std::fprintf(f, " &");
65
66             FIELDSTATE acField = Forest(i, j);
```

```

67         if(acField == EMPTY)
68             std::fprintf(f, "|[fill=white]|");
69         else if(acField & WATERED)
70             std::fprintf(f, "|[fill=cyan]|");
71         else if(acField & BURNABLE)
72             std::fprintf(f, "|[fill=green]|");
73
74         if(acField & COAL)
75             std::fprintf(f, "\\color{rgb}{0,0,0}");
76         else if(acField & BURNED)
77             std::fprintf(f, "\\color{rgb}{1,0,0}");
78         else if(acField == EMPTY)
79             std::fprintf(f, "\\color{gray}{0.5}");
80         else if(acField & BURNABLE)
81             std::fprintf(f, "\\color{gray}{0.75}");
82
83         if(acField & WATERED){
84             for (int t = 0; t < Solution.size(); ++t)
85                 if (Solution[t].x == i && Solution[t].y == j) {
86                     std::fprintf(f, "\\textbf{\\texttt{%02d}}", t+1);
87                     break;
88                 }
89         }
90         else if(acField & COAL)
91             std::fprintf(f, "\\textbf{\\texttt{CO}}");
92         else if(acField & BURNED)
93             std::fprintf(f, "\\textbf{\\texttt{BU}}");
94         else if(acField == EMPTY)
95             std::fprintf(f, "\\texttt{WA}");
96         else if(acField & BURNABLE)
97             std::fprintf(f, "\\texttt{FO}");
98         else
99             std::fprintf(f, "\\phantom{AA}");
100         std::fprintf(f, "%%\n");
101     }
102
103     std::fprintf(f, "\\n");
104 }
105
106 std::fprintf(f, "};\n\\end{tikzpicture}\\n");
107
108 if(finalOut){
109     std::fprintf(f, "\\n\\nExplanation:");
110     std::fprintf(f,
111         "\\n\\colorbox{white}{\\color{gray}{0.5}WA} ---
112         EMPTY");
111     std::fprintf(f,
112         "\\n\\colorbox{green}{\\color{gray}{0.5}FO} ---
113         BURNABLE");
112     std::fprintf(f,
114         "\\n\\colorbox{white}{\\color{rgb}{1,0,0}\\textbf{BU}}
115         --- BURNED");

```

```

113     std::fprintf(f,
114         "\\\n\\colorbox{white}{\\color{rgb}{0,0,0}\\textbf{C0}}
115         --- COAL (doubly burned)");
116     std::fprintf(f, "\\\n\\colorbox{cyan}{\\#\\#} ---
117     WATERED at time \\#\\#");
118     std::fprintf(f, "\\\nFields can have more than 1 state.");
119 }
120 }
121
122 void printSolution_TERMINAL(std::FILE* f, bool finalOut) {
123     fprintf(f, "\n");
124     //The ASCII-magic starts here:
125     for(int j= 0; j < Forest.height(); ++j) {
126         for(int i= 0; i < Forest.width(); ++i) {
127             FIELDSTATE acField = Forest(i, j);
128             int waterval = 0;
129
130             fprintf(f, "\x1b[s ");
131             if (acField == EMPTY)
132                 std::fprintf(f, "\x1b[u\x1b[37;47mWA");
133             if (acField & BURNABLE)
134                 std::fprintf(f, "\x1b[u\x1b[32;42mFO");
135             if (acField & BURNED)
136                 std::fprintf(f, "\x1b[u\x1b[1;5;31m/\\");
137             if (acField & COAL)
138                 std::fprintf(f, "\x1b[u\x1b[1;4;5;30m/\\");
139             if (acField & WATERED)
140                 for (int t = 0; t < Solution.size(); ++t)
141                     if (Solution[t].x == i && Solution[t].y == j) {
142                         std::fprintf(f, "\x1b[u\x1b[46m%02d", t+1);
143                         break;
144                     }
145             std::fprintf(f, "\x1b[0;39;49m");
146         }
147
148         std::fprintf(f, "\n");
149     }
150
151     if (finalOut) { // An Explanation shall be printed
152         std::fprintf(f, "\nExplanation:");
153         std::fprintf(f, "\n\x1b[37;47mWA\x1b[39;49m --- EMPTY");
154         std::fprintf(f, "\n\x1b[32;42mFO\x1b[39;49m --- BURNABLE");
155         std::fprintf(f, "\n\x1b[1;5;31m/\\ \x1b[0;39m --- BURNED");
156         std::fprintf(f, "\n\x1b[1;4;5;30m/\\ \x1b[0;39m --- COAL
157         (doubly burned)");
158         std::fprintf(f, "\n\x1b[46m##\x1b[0;39m --- WATERED at
159         time ##");
160         std::fprintf(f, "\nFields can have more than 1 state.");
161     }
162     std::fprintf(f, "\n");
163 }
164
165 void dontPrintSolution(std::FILE* f, bool finalOut) { return; }

```

161 //END OF OUTPUT

**Buschfeuer.cpp** Dies ist die Implementierung der Heuristik.

```

1  #include <cstdio>
2  #include <vector>
3  #include <queue>
4  #include <set>
5  #include <string>
6  #include <cstring>
7  #include <algorithm>
8
9  #include "Buschfeuer.h"
10
11 Point getOptimalWaterSpot(std::vector<Point>& candidates){
12     std::queue<std::pair<PII,Point> > q;
13     // ((distance | color) |
14     // Location)
15     for(int i= 0; i < candidates.size(); ++i)
16         q.push(std::pair<PII,Point>(PII(0,i),candidates[i]));
17     // insert all the candidates as start points for the BFS
18
19     std::vector<std::vector<std::set<int> > >
20         visited(Forest.width(), // remember all nearest points
21             first
22             std::vector<std::set<int> >(Forest.height()));
23     std::vector<std::vector<int> > shortDis(Forest.width(),
24         // shortest distant to any burning field
25         std::vector<int>(Forest.height(),oo));
26
27     //BFS to calculate shortest paths
28     while(!q.empty()){
29         std::pair<PII,Point> ac = q.front();
30         Point acPoint = ac.second;
31         int acDistance = ac.first.first;
32         int acColor = ac.first.second;
33
34         q.pop();
35         if(visited[acPoint.x][acPoint.y].count(acColor))
36             continue;
37         visited[acPoint.x][acPoint.y].insert(acColor);
38         shortDis[acPoint.x][acPoint.y] =
39             std::min(acDistance,shortDis[acPoint.x][acPoint.y]);
40
41         for(int i= 0; i < 4; ++i){
42             int newx = acPoint.x + dir[i][0];
43             int newy = acPoint.y + dir[i][1]; //
44             calculate new field's indexes
45
46             if(newx < 0 || newy < 0 || newy >= Forest.height() || newx
47                 >= Forest.width())
48                 continue; //
49             new field is outside the woods

```

```
40     if (Forest(newx, newy) != BURNABLE)
41         continue; //
42         Field is not of interest
43     if(visited[newx][newy].count(acColor) == 0) //
44         Don't compute things twice
45         if(acDistance + 1 <= shortDis[newx][newy]){
46             shortDis[newx][newy] = acDistance + 1;
47             q.push(std::pair<PII,Point>(PII(acDistance +
48                 1,acColor),Point(newx,newy)));
49         }
50     }
51 }
52 //determine the field to be watered
53 std::vector<PII> waterval(candidates.size(),PII(0,0));
54 std::vector<std::vector<int> > farthDist(candidates.size(),
55     std::vector<int>(2*(Forest.width()+Forest.height()),0));
56 //Count the number of fields that have an unique fire spot
57     a.k.a. waterval
58 //Reckon the farthest field
59 for(int i = 0; i < Forest.width(); ++i)
60     for(int j = 0; j < Forest.height(); ++j)
61         if(visited[i][j].size() == 1){
62             waterval[*visited[i][j].begin()].first++;
63             farthDist[*visited[i][j].begin()][shortDis[i][j]]++;
64         }
65 for(int i = 0; i < waterval.size(); ++i)
66     waterval[i].second = i;
67
68 std::sort(waterval.begin(),waterval.end(),std::greater<PII>());
69
70 //BEGIN HOTFIX
71 for(int i = 0; i < waterval.size(); ++i)
72     for(int j = 1; j < candidates.size(); ++j)
73         if(farthDist[waterval[i].second][j] > 1)
74             return candidates[waterval[i].second];
75 //END HOTFIX
76
77 return candidates[waterval[0].second];
78 }
79
80 std::vector<Point>& getInitialBurningFields() {
81     static std::vector<Point> burnedFields;
82
83     for(int i = 0; i < Forest.height(); ++i)
84         for(int j = 0; j < Forest.width(); ++j)
85             if(Forest(j, i) & BURNED){
86                 burnedFields.push_back(Point(j, i));
87             }
88     printf("Initially burning: (%i|%i)\n",j, i);
89 }
90
91 return burnedFields;
```

```
88 }
89
90 void simulateFire(const std::vector<Point>&
    initiallyBurningFields) {
91     std::vector<Point> burnedFields = initiallyBurningFields;
92     if(printSolution != dontPrintSolution)
93         printSolution_TERMINAL(stdout, false);
94     if (OUT != 0)
95         printSolution(OUT, false);
96
97     int time = 0;
98     while(!burnedFields.empty()) {                                     //
99         Simulate as long as there's still fire in the world
100         std::vector<Point> newBurnedFields;
101         // The burning fields at the next point of time
102
103         //Calculate the new burning fields
104         for(size_t i = 0; i < burnedFields.size(); ++i){
105             int acx = burnedFields[i].x;
106             int acy = burnedFields[i].y;
107
108             if(Forest(acx, acy) & WATERED)
109                 continue;                                             //
110             The field got watered and does not spread fire
111             Forest(acx, acy) |= COAL;                                  //
112             Field burned down to coal...
113
114             for(int j = 0; j < 4; ++j) {
115                 int newx = acx + dir[j][0];
116                 int newy = acy + dir[j][1];
117
118                 if(newx < 0 || newy < 0 || newy >= Forest.height() ||
119                    newx >= Forest.width())
120                     continue;                                         //
121                 new field is outside the woods
122                 if(Forest(newx, newy) == BURNABLE){
123                     Forest(newx, newy) |= BURNED;                    //
124                     Field starts burning
125                     newBurnedFields.push_back(Point(newx,newy));
126
127                 //      printf("  From now on burning: (%i|%i)\n",newx,newy);
128                 // log the happenings
129             }
130         }
131     }
132     if(newBurnedFields.empty())                                       //
133         Nothing to water, all plants happy...
134         break;
135
136     burnedFields = newBurnedFields;
137
138     Point toWater = getOptimalWaterSpot(newBurnedFields); //
139     Determine the field to water
```



```
130     Forest(toWater.x, toWater.y) |= WATERED;                                // ...
        and water it
131     Solution.push_back(toWater);
132
133
134     //Output / mirror the partial solution
135
136     std::printf("---\nAt time %i: Water spot
        (%i|%i)\n", ++time, toWater.x, toWater.y);
137     if(printSolution != dontPrintSolution)
138         printSolution_TERMINAL(stdout, false);
139
140     if (OUT) {
141         std::fprintf(OUT, "---\nAt time %i: Water spot
        (%i|%i)\n", time, toWater.x, toWater.y);
142         printSolution(OUT, false);
143     }
144
145 }
146 //     printf("Fire died.\n");
147
148     //Count the total number of burned or coaled
149     int wcnt = 0, ccnt = 0;
150     for(int i = 0; i < Forest.width(); ++i)
151         for(int j = 0; j < Forest.height(); ++j)
152             if(Forest(i, j) & WATERED)
153                 wcnt++;
154             else if(Forest(i, j) & COAL)
155                 ccnt++;
156
157     //Output / Mirror the solution
158     std::printf("---\nAnd you'll find %i pieces of coal and %i
        pieces of watered coal\n", ccnt, wcnt);
159     if(printSolution != dontPrintSolution)
160         printSolution_TERMINAL(stdout, true);
161     if (OUT) {
162         std::fprintf(OUT, "---\nAnd you'll find %i pieces of coal
        and %i pieces of watered coal\n", ccnt, wcnt);
163         printSolution(OUT, true);
164     }
165 }
166
167 int main(int argc, char** argv){
168     if (argc > 1) {
169         std::freopen(argv[1], "r", stdin);
170         std::printf("Using %s as input.\n", argv[1]);
171     }
172     if (argc > 2){
173         printf("Mirroring output to %s.\n", argv[2]);
174         if (strstr(argv[2], ".tex2")) {
175             std::printf("I reckon you want me to produce some
        graphicless TeX stuff...\n");
176             printSolution = dontPrintSolution;
```

```

177     }
178     else if (strstr(argv[2], ".tex")) {
179         std::printf("I reckon you want me to produce some TeX
180                     stuff...\n");
181         printSolution = printSolution_TEX;
182     }
183     else if (strstr(argv[2], ".raw")) {
184         std::printf("I reckon you want me to surpress
185                     graphics...\n");
186         printSolution = dontPrintSolution;
187     }
188     else
189         printSolution = printSolution_TERMINAL;
190     OUT = std::fopen(argv[2], "w");
191 }
192 else{
193     OUT = 0;
194     printSolution = printSolution_TERMINAL;
195 }
196
197 parseInput(stdin);
198 simulateFire(getInitialBurningFields());
199 }

```

**Buschfeuer2.cpp** Dies ist die Implementierung des Brute-Force-Ansatzes.

```

1  #include <cstdio>
2  #include <vector>
3  #include <queue>
4  #include <set>
5  #include <string>
6  #include <cstring>
7  #include <algorithm>
8  #include <map>
9
10 #include "Buschfeuer.h"
11
12 typedef std::tuple<int,int,Woods> sssType; //State-Space-Search:
13     distance from source; #of skipped waterings, acForest
14
15 pair<Woods,vector<Point>> getNextState(const Woods& w){
16     Woods ret(w.width(),w.height());
17     vector<Point> pnts;
18
19     for(int i = 0; i < w.width; ++i)
20         for(int j = 0; j < w.height; ++j){
21             ret(i,j) = w(i,j);
22             if(w(i,j) == BURNABLE){
23                 bool startsBurning = false;
24                 for(int k = 0; k < 4; ++k){
25                     int x = i + dir[k][0];
26                     int y = j + dir[k][1];
27                     if(x < 0 || y < 0 || x >= w.width() || y >= w.height())

```

```
27         continue;
28         if(w(x,y) & BURNED)
29             startsBurning = true;
30     }
31     if(startsBurning){
32         ret(i,j) |= BURNED;
33         pnts.push_back(Point(i,j));
34     }
35 }
36 }
37 return pair<Woods,vector<Point>>(ret, pnts);
38 }
39
40 int main(){
41     if (argc > 1) {
42         std::freopen(argv[1], "r", stdin);
43         std::printf("Using %s as input.\n", argv[1]);
44     }
45     if (argc > 2){
46         printf("Mirroring output to %s.\n", argv[2]);
47         if (strstr(argv[2], ".tex2")) {
48             std::printf("I reckon you want me to produce some
49                 graphicless TeX stuff...\n");
50             printSolution = dontPrintSolution;
51         }
52         else if (strstr(argv[2], ".tex")) {
53             std::printf("I reckon you want me to produce some TeX
54                 stuff...\n");
55             printSolution = printSolution_TEX;
56         }
57         else if (strstr(argv[2], ".raw")) {
58             std::printf("I reckon you want me to surpress
59                 graphics...\n");
60             printSolution = dontPrintSolution;
61         }
62         else
63             printSolution = printSolution_TERMINAL;
64         OUT = std::fopen(argv[2], "w");
65     }
66     else{
67         OUT = 0;
68         printSolution = printSolution_TERMINAL;
69     }
70     parseInput(stdin);
71
72     std::queue<sssType> q;
73     q.push(sssType(0,0,Forest));
74     set<Woods> visited;
75
76     while(!q.empty()){
77         sssType ac = q.front(); q.pop();
78         if(visited.count(get<2>(ac))) //case already calculated
```

```
77     continue;
78     visited.insert(get<2>(ac));
79
80     auto next = getNextState(get<2>(ac));
81     if(next.second.size() <= get<1>(ac) + 1){ //Fire can be dead
82         by this state
83         //Reconstruct Solution Here
84         Forest = get<2>(ac);
85         break;
86     }
87     q.push(sssType(get<0>(ac) + 1, get<1>(ac) + 1, next.first));
88     for(auto i : next.second){
89         next.first(i.first, i.second) |= WATERED;
90         q.push(sssType(get<0>(ac) + 1, get<1>(ac), next.first));
91         next.first(i.first, i.second) &= ~WATERED;
92     }
93 }
94
95 }
```

## 2 Aufgabe 2 - Lebenslinien

### 2.1 Lösungsidee

Die *Lebenszeit* eines Menschen ist ein abgeschlossenes Intervall  $L = [a, b]$  zwischen 2 Zeitpunkten  $a, b$ . Da es eine Bijektion  $J$  gibt, welche jeder Zeit eine reelle Zahl zuordnet, lässt sich die Lebenszeit eines Menschen auch als Intervall  $L' = [J(a), J(b)]$  von reellen Zahlen auffassen. Dies wird im Folgenden getan.

Ein *Lebensgraph* ist ein ungerichteter Graph  $G = (V, E)$ , auf dem eine Funktion  $f : V \mapsto P(\mathbb{R})$ <sup>13</sup> definiert ist, welche jedem Knoten eine Lebenszeit eines Menschen, also ein Intervall reeller Zahlen zuordnet und zusätzlich  $\forall u, v \in V : (u, v) \in E \Leftrightarrow f(u) \cap f(v) \neq \emptyset$  gilt. Es gibt also genau dann eine Kante zwischen 2 Knoten, wenn der Schnitt der beiden Lebenszeiten der Knoten nicht leer ist, es also einen Zeitpunkt gibt, zudem beide Menschen gelebt haben.

Aufgabe ist es nun, für einen gegebenen ungerichteten Graphen  $G = (V, E)$  zu prüfen, ob es eine Funktion  $f : V \mapsto P(\mathbb{R})$  gibt, sodass  $G$  Lebensgraph wird.

Dabei soll, sofern es ein solches  $f$  gibt,  $f(v)$  für alle Knoten  $v \in V$  ausgegeben werden, andernfalls soll der minimale Subgraph von  $G$  ausgegeben werden, für welchen allein es kein solches  $f$  geben kann.

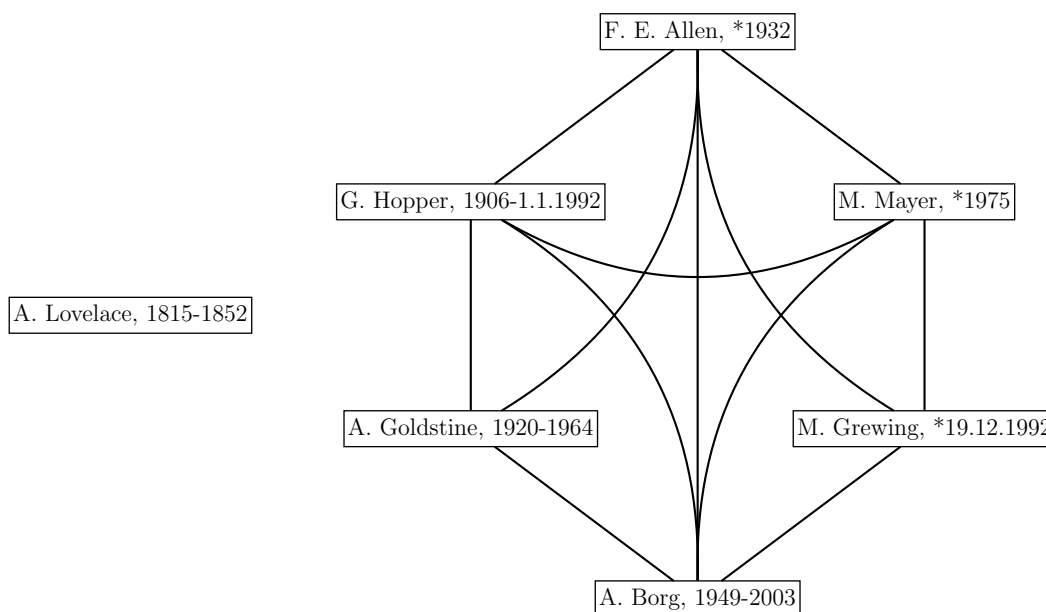


Abbildung 1: Der Lebensgraph aus der Aufgabenstellung

#### 2.1.1 Eigenschaften von Lebensgraphen

Es ist leicht ersichtlich, dass ein naiver Algorithmus zur Prüfung eines Graphen auf Lebensgrapheneigenschaft, also ein Algorithmus der alle möglichen zeitlichen Anordnungen der Knoten zueinander durchprobiert, nicht zum Ziel führt, da dieser mit einer grob approximierten Laufzeit von  $\mathcal{O}(\mathcal{V}!)$  wohl zu langsam ist.

Zur Überprüfung eines Graphen, ob dieser ein Lebensgraphen ist, ist es daher zunächst hilfreich sich Lebensgraphen etwas genauer zu betrachten. Es fällt zunächst auf, dass ein Graph, in dem

<sup>13</sup> $P(\mathbb{R})$  beschreibt die Potenzmenge von  $\mathbb{R}$ , also die Menge aller Teilmengen von  $\mathbb{R}$

ein *Loch*<sup>14</sup> auftritt niemals Lebensgraph sein kann:

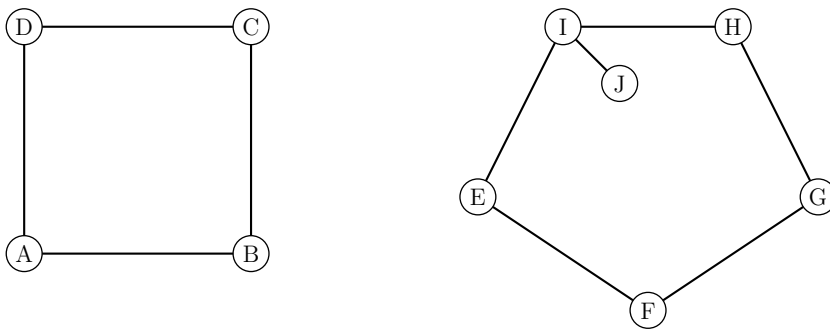


Abbildung 2: Graphen mit Löchern können niemals Lebensgraph sein (*im 2. Graphen ist J nicht Teil des Loches*)

Der Grund hierfür ist offensichtlich: Sei  $Z = (v_i, v_k, \dots, v_i)$  ein Zyklus der Länge größer 3 eines Graphen  $G = (V, E)$ , und gelte für  $G$ : zwischen 2 Knoten aus  $Z$  existiert nur genau dann eine Kante in  $G$ , sofern diese beiden Knoten im Zyklus nacheinander durchlaufen werden; bei  $Z$  handelt es sich also um ein Loch von  $G$ .

Weise man einem Knoten  $v_i \in Z$  nun ein Intervall  $I_0 = [a_0, b_0]$  zu. Nun muss dem Nachfolger  $v_{i_1}$  von  $v_i$  im Zyklus  $Z$  ein Intervall  $I_1 = [a_1, b_1]$  zugewiesen werden, wobei entweder  $a_0 < a_1 \leq b_0 < b_1$  oder  $a_1 < a_0 \leq b_1 < b_0$  gelten muss, da in  $G$  zwischen  $v_i$  und  $v_{i_1}$  eine Kante existiert. Hat man sich jedoch für einen dieser beiden Fälle entschieden, so muss man sich bei der Zuweisung von Intervallen zu den nächsten Knoten in  $Z$  immer für diesen Fall entscheiden. Sonst würde man Intervalle erhalten, welche einen nichtleeren Schnitt besitzen, deren Knoten in  $G$  jedoch nicht durch eine Kante verbunden sind. Dies wäre ein Widerspruch zur Definition eines Lebensgraphen.

Setzt man diese Zuweisungen jedoch bis zum Ende des Zyklus fort, so erhält man zwangsläufig ein Problem mit der Kante zwischen dem Knoten  $v_i$  und seinem Vorgänger im Zyklus  $Z$ . In jedem Fall muss der Schnitt der diesen beiden Knoten zugewiesenen Intervalle nach Konstruktion leer sein, da man ansonsten bei einem vorangegangenen Schritt einen Widerspruch zur Definition eines Lebensgraphen erhalten hatte. Dies an sich stellt jedoch auch einen Widerspruch dar, da diese beiden Knoten in  $G$  mit einer Kante verbunden sind.

Somit hat ein Lebensgraph kein Loch.

Graphen ohne Löcher werden in der Literatur *Chordalgraph* oder *Triangulierter Graph*<sup>15</sup> genannt, es gibt effiziente Algorithmen zur Erkennung solcher Graphen.

Es sei an dieser Stelle angemerkt, dass ein Lebensgraph sehr wohl *Dreiecke*, also Zyklen der Länge 3 haben darf. Dies liegt insbesondere daran, dass ein Dreieck eine *Clique* der Größe 3 bildet, jeder der 3 Knoten also mit jedem anderen der 3 Knoten verbunden ist. Speziell bei Dreiecken muss es also einen Zeitpunkt geben, an dem alle 3 entsprechenden Menschen gelebt haben.

Weiterhin ist es für einen Lebensgraphen nur *notwendig* Chordalgraph zu sein. Betrachte man folgenden Graphen, der Chordalgraph ist, jedoch nicht Lebensgraph sein kann:

<sup>14</sup>Ein Loch ist dabei ein Zyklus mit einer Länge größer 3, zwischen dessen einzelnen Knoten nur eine Kante existiert, wenn diese auch im Zyklus existiert.

<sup>15</sup>Der englischsprachige Wikipediaartikel ist in diesem Fall (mal wieder) deutlich informativer: [https://en.wikipedia.org/wiki/Chordal\\_graph](https://en.wikipedia.org/wiki/Chordal_graph)

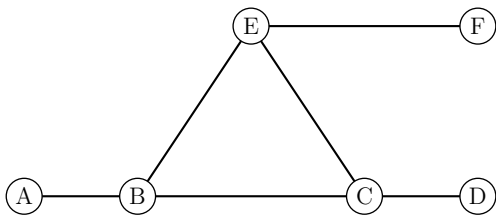


Abbildung 3: Chordalgraph, der **kein** Lebensgraph ist

## 2.2 Umsetzung

## 2.3 Beispiele

## 2.4 Quelltext