32. Bundeswettbewerb Informatik, Runde 2 Ausarbeitungen zu den Aufgaben "Buschfeuer" und "Lebenslinien"

Philip Wellnitz

Vorwort

Armer Korrektor!¹²

Auf den folgenden Seiten sind die vor meiner unendlichen Genialität strotzenden Lösungen der diesjährigen Zweitrundenaufgaben niedergeschrieben und zu bewundern. Die glorreichen Programme zu den einzelnen Aufgaben sollten sich auf einem normalen Rechner³ aus einem Terminal problemlos starten lassen; speziell für Aufgabe 1 eignet sich besonders eines, welches die göttlichen, also mir gleichen, ASCII-Escape-Sequenzen unterstützt.

Weiterhin habe ich (noch) darauf verzichtet, meine überragenden Programmierfähigkeiten in wunderschönen Programmen im gut lesbaren ASM auszudrücken; auch habe ich kein C++ mit inline-ASM verwendet⁴.

Nach diesem mit übermäßiger Bescheidenheit glänzenden Vorwort möchte ich Ihnen nun viel Freude bei der Korrektur meiner Lösungen wünschen... und eine erholsame Zeit danach.

Inhaltsverzeichnis

L	Auf	gabe 1 - Buschfeuer	3
	1.1	Lösungsidee	3
		1.1.1 Korrektheit	6
		1.1.2 Laufzeitanalyse	7
	1.2	Umsetzung	
	1.3	Beispiele	8
		1.3.1 Beispiel 0	8
		1.3.2 Beispiel 1	11
		1.3.3 Beispiel 2	14
		1.3.4 Beispiel 3	18
	1.4	Quelltext	23

 $^{^1\}mathrm{Dieses}$ Vorwort existiert nur, weil es scheinbar zur Struktur gehören muss. Außerdem wollte ich sowas auch mal schreiben...

²Und es ist essenziell, um meine ständige Präsenz in den Perlen der Informatik zu wahren...

³Das sei im Folgenden ein Rechner mit einem Linux-artigen OS

⁴Schade eigentlich.

1 Aufgabe 1 - Buschfeuer

1.1 Lösungsidee

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

BRENNBAR Das Stück Land ist ind 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 seinen diejenigen Felder, welche bei mindestens 2 Beobachtungen brennend waren, als *verkohlt* bezeichnet.

Nach der Feuerausbreiteung 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 Feler 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 umittelbar 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 kleinster 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 kleinster 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 besitmmt ist. Löscht man F, so wird, wie schon gesehen, die minimale Lebenszeit all 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 μ effizient zu bestimmen. Da ein Wald eine rechteckige Gitterform besitzt, ist der kürzeste Abstand zwischen 2 Feldern 1, ganau 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 μ 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
     brennendeFelder := null; //1D-Container für Positionen
4
        brennender Felder
     for (i = 0..Wald.Höhe())
5
       for (j = 0..Wald.Breite())
6
7
          if (Wald[i,j] == BRENNEND)
            brennendeFelder.Add((i;j)); //Gefundene Position
8
               hinzufügen
9
10
     return brennendeFelder; //Alle gefundenen Positionen
        zurückgeben
  }
11
12
  NächsteBeobachtung(aktBrennendeFelder) { //Ermittelt die bei der
13
      nächsten Beonachtung brennenden Felder, aus den Feldern, die
      aktuell brennen
```

```
14
     neuBrennendeFelder := null;
15
     for all((x;y) from aktBrennendeFelder)
       if (Wald[x,y] == GELÖSCHT)
16
17
         continue; //Feld kann kein Feuer verteilen
18
       Wald[x,y] := VERKOHLT; //2 mal brennende Felder sind verkohlt
19
20
       for all((x';y') from Umgebung((x;y)))
         if(Wald[x',y'] == BRENNBAR)
21
22
           neuBrennendeFelder.Add((x';y')); //Gefundene Position
              hinzufügen
23
           Wald[x',y'] := BRENNEND; //Wald beginnt zu brennen
24
25
     return neuBrennendeFelder;
26 }
27
  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())
           Fülle kleinsterAbstand[i] mithilfe einer Breitensuche
32
33
34
     anzEindeutigKleinstAbstände := null;
35
36
     for (i = 0..Wald.Höhe())
       for (j = 0..Wald.Breite())
37
         if(Es ex. k mit kleinsterAbstand[k][i,j] eindeutiges
38
            Minimum für alle mögliche k)
39
           anzEindeutigKleinstAbstände[k]++;
40
41
     return aktBrennendeFelder[k, sodass
        anzEindeutigKleinstAbstände[k] maximal];
42 }
43
  SimuliereFeuer() { //Die eigentliche Berechnung
44
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())
                             //Keine Felder brennen mehr
49
             break;
50
     Wald[GetOptBewässerungspunkt(aktBrennendeFelder)] := GELÖSCHT;
51
        //Lösche das aktuell beste Feld
52 }
```

1.1.1 Korrektheit

Wie schon beschrieben, wird bei jeder Beobachtung das für diese Beobachtung nach μ 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 μ optimalste Feld gelöscht werden. Verallgemeinernd muss gezeigt werden, dass kein μ' existiert, welches bei midestens 1 Beobachtung 1 anderes Feld als μ vorschlägt und bei der insgesamt weniger Felder abbrennen als bei μ ; dass μ also optimal ist.

Außerdem muss gezeigt werden, dass der Algorithmus terminiert. Da der Algorithmus jedoch nur brennende und nicht verkohlte Felder betrachtet und jedes brennendes Feld nach endlicher Zeit ist den Zustand verkohlt übergeht, gibt es einen Zeitpunkt, ab dem alle einst brennenden Felder vekohlt sind. Dann gibt es jedoch keine Felder, auf denen der Algorithmus operieren kann, der Algorithmus terminiert dann, und somit immer.

Nach der Definition von μ wird dasjenige, beliebige Feld F_i aus allen möglichen Feldern $F_1..F_n$ zum Löschen ausgewählt, welches die minimale Lebenszeit von den meisten Feldern erhöht. Wählte man ein beliebiges Feld $F_i^{<}$ aus $\{F_1,...,F_n\}$, mit $\mu(F_i^{<}) < \mu(F_i)$ so erhöht sich nach Definition der minimalen Lebenszeit diese bei $\mu(F_i) - \mu(F_i^{<}) > 0$ Feldern weniger, als wenn man F_i wählte. Erhalten diese Felder in den nächsten z_o Beobachtungen keine Lebenszeitvelängerung, so brennen sie ab, wobei z_o der kleinste Abstand des Feldes o zu F_i ist.

Es verbleibt also zu zeigen, dass es keine Situation geben kann, bei der die Wahl von $F_i^{<}$ zu einer insgesamt geringeren Anzahl an verbrannten Feldern führt.

Angenommen es gäbe solch eine Situation.

Dies heißt jedoch, dass es eine oder mehrere Löschungen von Feldern gibt, welche insgesamt dazu führen, dass die Lebenszeit von $\mu(F_i) - \mu(F_i^<) + 1$ Feldern verlängert wird. Außerdem dürfen diese Löschungen nicht möglich sein, wenn F_i anstatt $F_i^<$ gelöscht wird. Dies im Speziellen heißt jedoch, dass Felder gelöscht werden, welche sonst durch die Löschung von F_i eine Lebenszeitverlängerung erhielten. Somit wäre es aber besser gewesen, F_i zu löschen, da bei den Beobachtungen danach auch andere Felder gelöscht werden könnten und die insgesamte Anzahl an verbrannten Feldern so insgesamt gesunken wäre. Es ist also optimal, ein Feld mit maximalem $\mu(F_i)$ auszuwählen. Bleibt zu zeigen, dass die Wahl eines speziellen F_i mit maximalem $\mu(F_i)$ an der insgesamten Anzahl an verbrannten Feldern nichts ändert.

Sei $F_i^=$ ein beliebiges Feld aus $\{F_1, ..., F_n\}$, mit $\mu(F_i^=) = \mu(F_i)$ und $F_i^= \neq F_i$. Es genügt zu zeigen, dass das Wählen von $\mu(F_i^=)$ keine Verringerung der am Ende insgesamt brennenden Felder gegenüber $\mu(F_i)$ darstellt, da $\mu(F_i)$ und $\mu(F_i^=)$ beliebig gewählt sind.

Angenommen dies sei der Fall.

Der Algorithmus ist also korrekt und optimal.

 $^{^5}$ Es ist theoretisch möglich, dass die Wahl zwischen $F_i^<$ und F_i keinen Unterschied macht, beispielsweise, wenn alle Felder innerhalb der nächsten o Beobachtungen verkohlen oder gelöscht werden. Dabei sei o der maximale Abstand, der in $\mu(F_i^<)$ Berücksichtigung fand. Dies stellt jedoch keinen Widerspruch zur Behauptung dar.

⁶Sei vorrausgestzt, dass ein solches $F_i^=$ existiert. Andernfalls existiert dieser Fall nicht, der Beweis ist dann hier beendet.

1.1.2 Laufzeitanalyse

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 $\max n, m \le 200$.

1.2 Umsetzung

Für die Umsetzung habe ich die Sprache C++ verwendet. Zunächst habe ich FIELDSTATE definiert⁷

 $^{^7\}mathrm{Das}$ Wort "definiert" ist durchaus ernst zu nehmen, da es hier beschreiben soll, dass etwas mit # defines "gelöst" wurde.

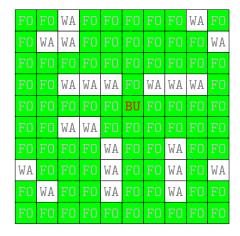
1.3 Beispiele

1.3.1 Beispiel 0

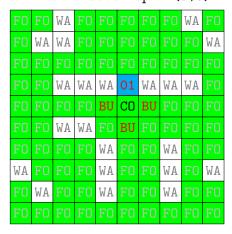
Die ist das Beispiel aus der Aufgabenstellung. Umgewandelt für mein Programm sieht diese Eingabe folgendermaßen aus⁸:

- 1 10 10
- 2 1101111101
- 3 1001111110
- 4 1111111111
- 5 1100010001
- 6 1111131111
- 7 1100111111
- 8 1111011011 9 0111011010
- 10 1011011011
- 11 111111111

Mein Programm produziert folgende Ausgabe⁹¹⁰:



At time 1: Water spot (5|3)



⁸Diese Eingabe finden Sie auch in der Datei O.in

⁹Diese Ausgabe finden Sie auch in der Datei 0.out.tex; Eine Datei 0.out mit den ASCII-Escape-Sequenzen exisitert ebenfalls.

¹⁰Um die ASCII-Escape-Sequenzen in T_EX korrekt darzustellen, habe ich spezielle Ausgabemethoden geschrieben. Diese produzieren anstatt der ASCII-Sequenzen T_EX-Befehle, welche optisch zu ähnlichen Ergebnissen führen.

At time 2: Water spot (3|4)

FO	FO	WA	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	FO	FO	FO	FO	FO	FO	WA
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									

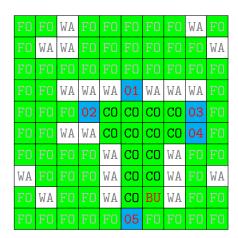
At time 3: Water spot (8|4)

FO	FO	WA	FO	FO	FO	FO	FO	WA	FO
FO	WA	WA	FO	FO	FO	FO	FO	FO	WA
FO									
FO	FO	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									

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	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									

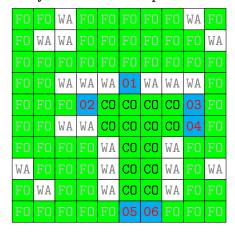
At time 5: Water spot (5|9)



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	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



Explanation:

WA --- EMPTY

FO --- BURNABLE

BU --- BURNED

CO --- COAL (doubly burned)

--- WATERED at time

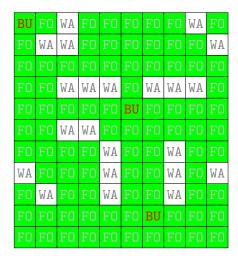
Fields can have more than 1 state.

1.3.2 Beispiel 1

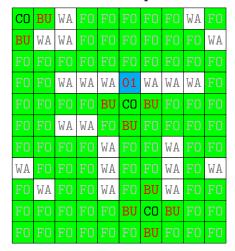
Eine Situation mit mehr als einem Feuer bei der ersten Beobachtung¹¹:

- 1 10 11
- 2 3101111101
- 3 1001111110
- 4 111111111
- 5 1100010001
- 6 1111131111
- 7 1100111111
- 8 1111011011
- 9 0111011010
- 10 1011011011
- 11 1111113111
- 12 111111111

Mein Programm produziert folgende Ausgabe¹²:



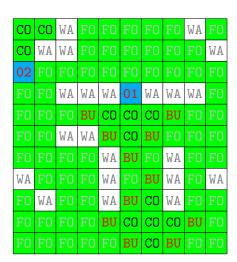
At time 1: Water spot (5|3)



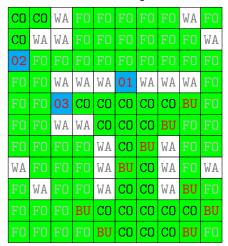
At time 2: Water spot (0|2)

¹¹Diese Eingabe finden Sie auch in der Datei 1.in

¹²Diese Ausgabe finden Sie auch in der Datei 1.out.tex; Eine Datei 1.out mit den ASCII-Escape-Sequenzen exisitert ebenfalls.



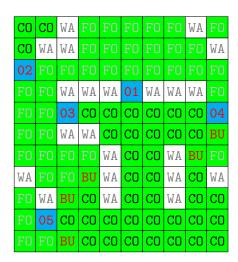
At time 3: Water spot (2|4)



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	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						
FO	FO	FO	BU	CO	CO	CO	CO	CO	BU

At time 5: Water spot (1|9)



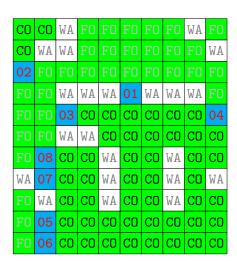
At time 6: Water spot (1|10)

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

At time 7: Water spot (1|7)

CO	CO	WA	FO	FO	FO	FO	FO	WA	FO
CO	WA	WA	FO	FO	FO	FO	FO	FO	WA
02	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	BU	CO	WA	CO	CO	WA	CO	CO
WA	07	CO	CO	WA	CO	CO	WA	CO	WA
FO	WA	CO	CO	WA	CO	CO	WA	CO	CO
FO	05	CO							
FO	06	CO							

At time 8: Water spot (1|6)



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	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	80	CO	CO	WA	CO	CO	WA	CO	CO
WA	07	CO	CO	WA	CO	CO	WA	CO	WA
FO	WA	CO	CO	WA	CO	CO	WA	CO	CO
FO	05	CO							
FO	06	CO							

Explanation:

WA --- EMPTY

--- BURNABLE

BU --- BURNED

CO --- COAL (doubly burned)

--- WATERED at time

Fields can have more than 1 state.

1.3.3 Beispiel 2

13.

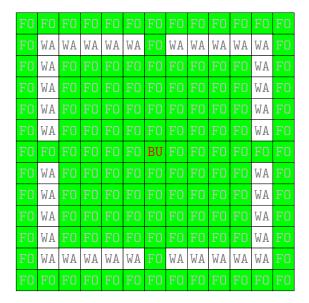
- 1 13 13
- 2 11111111111111
- 3 1000001000001
- 4 1011111111101
- 5 1011111111101
- 6 10111111111101
- 7 1011111111101

^{8 1111113111111}

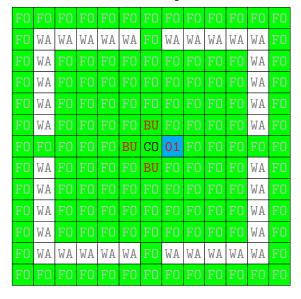
¹³Diese Eingabe finden Sie auch in der Datei 2.in

- 9 1011111111101
- 10 1011111111101
- 11 1011111111101
- 12 1011111111101
- 13 1000001000001
- 14 111111111111

Mein Programm produziert folgende Ausgabe¹⁴:



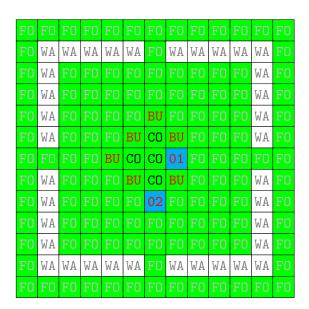
--- At time 1: Water spot (7|6)



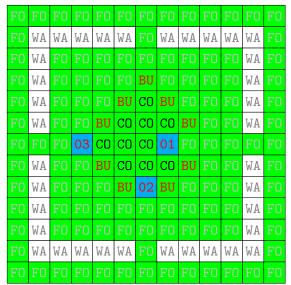
--- At time 2: Water spot (6|8)

1

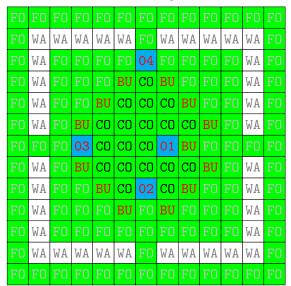
¹⁴Diese Ausgabe finden Sie auch in der Datei 2.out.tex; Eine Datei 2.out mit den ASCII-Escape-Sequenzen exisitert ebenfalls.



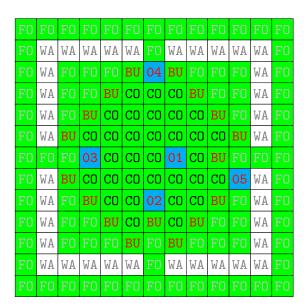
--- At time 3: Water spot (3|6)



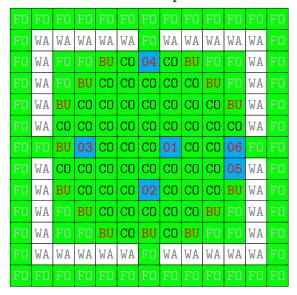
--- At time 4: Water spot (6|2)



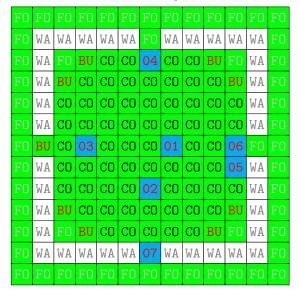
--- At time 5: Water spot (10|7)



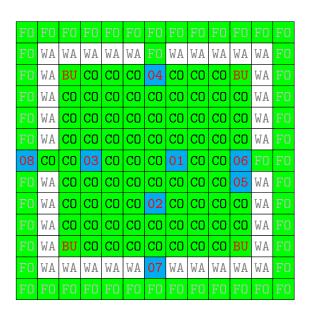
--- At time 6: Water spot (10|6)



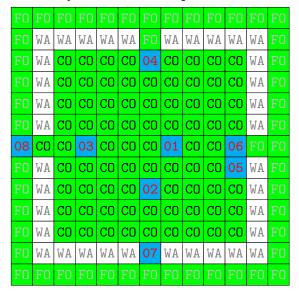
--- At time 7: Water spot (6|11)



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



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



Explanation:

WA --- EMPTY

FO --- BURNABLE

BU --- BURNED

CO --- COAL (doubly burned)

--- WATERED at time

Fields can have more than 1 state.

1.3.4 Beispiel 3

Ein (etwas) größeres Beispiel. 15:

1 100 100

¹⁵Diese Eingabe finden Sie auch in der Datei 3.in

```
7
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
33
34
35
36
37
38
39
40
41
42
43
45
46
47
48
49
50
51
52
53
55
```

```
59
60
63
64
65
66
67
69
70
74
76
77
79
80
81
83
88
89
90
92
93
94
95
99
100
101
```

Mein Programm produziert folgende Ausgabe¹⁶ 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)

¹⁶Diese Ausgabe finden Sie auch in der Datei 3.out.tex2;

- 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)
- 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

```
1 #include <cstdio>
2 #include <vector>
3 #include <queue>
4 #include <set>
5 #include <string>
6 #include <cstring>
7
  using namespace std;
9
  typedef pair<int,int> PII;
10
11 #define FIELDSTATE
                            char
12 #define EMPTY
13 #define BURNABLE
14 #define BURNED
                            2
15 #define WATERED
16 #define COAL
17
  const int oo = (1 << 29);</pre>
                                                                  //
      The infinity
```

```
19
20 \quad {\tt class} \ {\tt Woods} \{
21 private:
22
     int Width, Height;
     vector < vector < FIELDSTATE > > Fields;
23
24
25
  public:
26
     Woods(int width, int height) : Width(width), Height(height) {
27
       Fields.assign(height, vector < FIELDSTATE > (width, 0));
28
29
30
     int width() const { return Width; }
31
     int height() const { return Height; }
32
33
     FIELDSTATE& operator() (int x, int y) {
        if (x < 0 | | y < 0 | | x >= width() | | y >= height())
34
35
          printf("OUT OF BOUNDS 1");
36
       return Fields[y][x];
37
38
     FIELDSTATE operator() (int x, int y) const {
39
        if (x < 0 || y < 0 || x >= width() || y >= height())
40
          printf("OUT OF BOUNDS 2");
       return Fields[y][x];
41
42
     }
43
   } Forest(0, 0);
44
45
  struct Point {
46 public:
47
     int x, y;
48
     Point(int _x,int _y) : x(_x), y(_y) { }
49
  int dir [4] [2] = \{\{1,0\},\{0,1\},\{-1,0\},\{0,-1\}\};
50
51
52 vector < Point > Solution;
53 FILE* OUT;
                                                                     11
      The file to mirror the output to
  void (*printSolution)(FILE*, bool);
54
55
56 Point getOptimalWaterSpot(vector < Point > & candidates) {
     queue < pair < PII, Point > > q;
                                                                     //
57
         ((distance | color) | Location)
     for(int i= 0; i < candidates.size(); ++i)</pre>
58
59
        q.push(pair < PII, Point > (PII(0,i), candidates[i]));
                                                                     //
           insert all the candidates as start points for the BFS
60
61
     vector < vector < set < int > > visited(Forest.width(),
                                                                     //
        remember all nearest points first
62
       vector<set<int> >(Forest.height()));
63
     vector < vector < int > > shortDis(Forest.width(),
                                                                     //
        shortest distant to any burning field
64
       vector < int > (Forest.height(),oo));
65
     //BFS to calculate shortest paths
66
```

```
67
      while(!q.empty()){
68
        pair < PII , Point > ac = q.front();
69
        Point acPoint = ac.second;
70
        int acDistance = ac.first.first;
71
        int acColor = ac.first.second;
72
73
        q.pop();
        if(visited[acPoint.x][acPoint.y].count(acColor))
74
75
           continue;
76
        visited[acPoint.x][acPoint.y].insert(acColor);
77
78
        for(int i= 0; i < 4; ++i){</pre>
79
           int newx = acPoint.x + dir[i][0];
80
           int newy = acPoint.y + dir[i][1];
                                                                      //
              calculate new field's indexes
81
82
          if(newx < 0 || newy < 0 || newy >= Forest.height() || newx
              >= Forest.width())
83
             continue:
                                                                      //
                new field is outside the woods
          if (Forest(newx, newy) != BURNABLE)
84
85
             continue;
                                                                      //
                Field is not of interest
86
87
          if(visited[newx][newy].count(acColor) == 0)
                                                                      //
              Don't compute things twice
88
             if (acDistance + 1 <= shortDis[newx][newy]){</pre>
               shortDis[newx][newy] = acDistance + 1;
89
               q.push(pair < PII, Point > (PII (acDistance +
90
                  1,acColor),Point(newx,newy)));
91
            }
92
        }
      }
93
94
95
      //determine the field to be watered
96
      vector < int > waterval(candidates.size(),0);
97
98
      //Count the number of fields that have an unique fire spot
         a.k.a. waterval
99
      for(int i= 0; i < Forest.width(); ++i)</pre>
100
        for(int j= 0; j < Forest.height(); ++j)</pre>
           if(visited[i][j].size() == 1)
101
102
             waterval[*visited[i][j].begin()]++;
103
104
      //determine the field of the candidates which has the highest
         waterval
      int maxv = waterval[0];
                                                                     //
105
         maximal value
106
      int maxi = 0;
                                                                     //
         index of maximal value
107
      for(int i= 1; i < candidates.size(); ++i)</pre>
108
        if (waterval[i] > maxv){
109
```

```
110
          maxv = waterval[i];
111
          maxi = i;
112
        }
113
114
      return candidates[maxi];
    }
115
116
117
   //BEGIN OF INPUT
118
   void parseInput(FILE* f) {
119
      int acFieldWidth, acFieldHeight;
      fscanf(f, "%i %i\n",&acFieldWidth, &acFieldHeight);
120
121
122
      Forest = Woods(acFieldWidth, acFieldHeight);
123
124
      for(int i = 0; i < acFieldHeight; ++i){</pre>
125
        for(int j= 0; j < acFieldWidth; ++j){</pre>
126
           char c;
           fscanf(f, "%c",&c);
127
128
           c = 0;
129
           Forest(j, i) = (FIELDSTATE) c;
        }
130
131
        if(i < acFieldHeight-1)</pre>
           fscanf(f, "\n");
132
133
      }
134
   }
   //END OF INPUT
135
    //BEGIN OF OUTPUT
136
    void printSolution_TEX(FILE* f, bool finalOut) {
137
      fprintf(f, "\\\\n");
138
139
140
      fprintf(f, "\\begin{tikzpicture}\n");
      fprintf(f, "\\tikzset{square matrix/.style={\n");
141
      fprintf(f, "matrix of nodes,\n");
142
      fprintf(f, "column sep=-\\pgflinewidth, row
143
         sep=-\\pgflinewidth,\n");
      fprintf(f, "nodes={draw,\n");
144
145
      fprintf(f, "minimum height=#1,\n");
      fprintf(f, "anchor=center,\n");
146
147
      fprintf(f, "text width=#1,\n");
      fprintf(f, "align=center,\n");
148
      fprintf(f, "inner sep=0pt\n");
149
150
      fprintf(f, "},\n");
      fprintf(f, "},\n");
151
      fprintf(f, "square matrix/.default=1.2cm\n");
152
      fprintf(f, "}\n");
153
154
      fprintf(f, "\\matrix[square matrix=1.4em] {\n");
155
156
      for(int j= 0; j < Forest.height(); ++j) {</pre>
        for(int i= 0; i < Forest.width(); ++i) {</pre>
157
158
           if(i)
159
             fprintf(f," &");
160
161
             FIELDSTATE acField = Forest(i, j);
```

```
162
            if(acField == EMPTY)
163
               fprintf(f, "|[fill=white]|");
164
            else if(acField & WATERED)
165
               fprintf(f, "|[fill=cyan]|");
166
            else if(acField & BURNABLE)
167
               fprintf(f, "|[fill=green]|");
168
169
            if(acField & COAL)
170
               fprintf(f, "\\color[rgb]{0,0,0}");
171
            else if(acField & BURNED)
               fprintf(f, "\\color[rgb]{1,0,0}");
172
173
            else if(acField == EMPTY)
174
               fprintf(f, "\\color[gray]{0.5}");
175
            else if(acField & BURNABLE)
176
               fprintf(f, "\\color[gray]{0.75}");
177
178
            if(acField & WATERED){
               for (int t = 0; t < Solution.size(); ++t)</pre>
179
                 if (Solution[t].x == i && Solution[t].y == j) {
180
181
                   fprintf(f, "\\textbf{%02d}",t+1);
                   break;
182
183
                 }
            }
184
185
            else if(acField & COAL)
186
               fprintf(f, "\\textbf{CO}");
187
            else if(acField & BURNED)
               fprintf(f, "\\textbf{BU}\");
188
189
            else if(acField == EMPTY)
               fprintf(f, "WA");
190
191
            else if(acField & BURNABLE)
               fprintf(f, " FO");
192
193
            else
194
               fprintf(f, "\\phantom{AA}");
195
          fprintf(f, "%%\n");
196
197
        fprintf(f, "\\\\n");
198
199
200
201
      fprintf(f, "};\n\\end{tikzpicture}\\\\n");
202
203
      if(finalOut){
204
        fprintf(f, "\\\nExplanation:");
        fprintf(f, "\\\\n\\colorbox{white}{\\color[gray]{0.5}WA}
205
                EMPTY");
206
        fprintf(f, "\\\\n\\colorbox{green}{\\color[gray]{0.5}F0}
                BURNABLE");
           ---
207
        fprintf(f,
           "\\\\n\\colorbox{white}{\\color[rgb]{1,0,0}\\textbf{BU}}
           --- BURNED");
208
        fprintf(f,
           "\\\n\\colorbox{white}{\\color[rgb]{0,0,0}\\textbf{CO}}
                COAL (doubly burned)");
```

```
209
        fprintf(f, "\\\\n\\colorbox{cyan}{\\#\\#} ---
                                                             WATERED at
           210
        fprintf(f, "\\\\nFields can have more than 1 state.");
211
212 }
213
214
   void printSolution_TERMINAL(FILE* f, bool finalOut) {
215
      fprintf(f, "\n");
216
      //The ASCII-magic starts here:
217
      for(int j= 0; j < Forest.height(); ++j) {</pre>
        for(int i= 0; i < Forest.width(); ++i) {</pre>
218
219
          FIELDSTATE acField = Forest(i, j);
220
          int waterval = 0;
221
222
          fprintf(f, "\x1b[s ");
223
          if (acField == EMPTY)
224
            fprintf(f, \sqrt{x1b[u \times 1b[37;47mWA");}
225
          if (acField & BURNABLE)
226
            fprintf(f, "\x1b[u\x1b[32;42mF0");
227
          if (acField & BURNED)
228
            fprintf(f, \sqrt{x1b[u \times 1b[1;5;31m/\])};
229
          if (acField & COAL)
230
            fprintf(f, "\x1b[u\x1b[1;4;5;30m/\\");
          if (acField & WATERED)
231
232
            for (int t = 0; t < Solution.size(); ++t)</pre>
233
               if (Solution[t].x == i && Solution[t].y == j) {
234
                 fprintf(f, "x1b[u\\x1b[46m%02d", t+1);
235
                 break;
236
               }
237
          fprintf(f, "\x1b[0;39;49m");
238
        }
239
240
        fprintf(f, "\n");
      }
241
242
243
      if (finalOut) {// An Explanation shall be printed
244
        fprintf(f, "\nExplanation:");
        fprintf(f, "\n\x1b[37;47mWA\x1b[39;49m
245
                                                  --- EMPTY");
        fprintf(f, "\n\x1b[32;42mF0\x1b[39;49m --- BURNABLE");
246
        fprintf(f, "\n\x1b[1;5;31m/\\x1b[0;39m --- BURNED");
247
        fprintf(f, \sqrt{x1b[1;4;5;30m/\sqrt{x1b[0;39m}]} --- COAL (doubly
248
           burned)");
249
        fprintf(f, "\n\x1b[46m##\x1b[0;39m --- WATERED at time])
           ##");
250
        fprintf(f, "\nFields can have more than 1 state.");
251
252
      fprintf(f, "\n");
253 }
254
255
   void dontPrintSolution(FILE* f, bool finalOut) { return; }
256
   //END OF OUTPUT
257
258
   vector < Point > & getInitialBurningFields() {
```

```
259
      static vector < Point > burnedFields;
260
      for(int i = 0; i < Forest.height(); ++i)</pre>
261
262
        for(int j= 0; j < Forest.width(); ++j)</pre>
          if(Forest(j, i) & BURNED){
263
             burnedFields.push_back(Point(j, i));
264
             printf("Initially burning: (%i|%i)\n",j, i);
265
266
          }
267
      return burnedFields;
268
269
270
   void simulateFire(const vector < Point > & initially BurningFields) {
271
      vector < Point > burnedFields = initiallyBurningFields;
272
      if(printSolution != dontPrintSolution)
273
        printSolution_TERMINAL(stdout, false);
274
      if (OUT != 0)
275
        printSolution(OUT, false);
276
277
      int time = 0;
278
      while(!burnedFields.empty()) {
                                                                    //
         Simulate as long as there's still fire in the world
279
        vector < Point > newBurnedFields;
                                                                    //
            The burning fields at the next point of time
280
281
        //Calculate the new burning fields
        for(size_t i = 0; i < burnedFields.size(); ++i){</pre>
282
283
          int acx = burnedFields[i].x;
          int acy = burnedFields[i].y;
284
285
286
          if(Forest(acx, acy) & WATERED)
             continue:
                                                                    //
287
                The field got watered and does not spread fire
288
          Forest(acx, acy) |= COAL;
                                                                    //
              Field burned down to coal...
289
290
          for(int j = 0; j < 4; ++j) {
291
             int newx = acx + dir[j][0];
292
             int newy = acy + dir[j][1];
293
             if(newx < 0 || newy < 0 || newy >= Forest.height() ||
294
                newx >= Forest.width())
                                                                    //
295
               continue;
                  new field is outside the woods
             if (Forest(newx, newy) == BURNABLE){
296
297
               Forest(newx, newy) |= BURNED;
                                                                    //
                  Field starts burning
298
               newBurnedFields.push_back(Point(newx,newy));
299
300
   //
               printf(" From now on burning: (%i|%i)\n",newx,newy);
            // log the happenings
301
302
          }
303
        }
```

```
304
        if (newBurnedFields.empty())
                                                                    //
            Nothing to water, all plants happy...
305
             break:
306
307
        burnedFields = newBurnedFields;
308
309
        Point toWater = getOptimalWaterSpot(newBurnedFields); //
            Determine the field to water
310
        Forest(toWater.x, toWater.y) |= WATERED;
                                                                    // ...
            and water it
        Solution.push_back(toWater);
311
312
313
314
        //Output / mirror the partial solution
315
        printf("---\nAt time %i: Water spot
316
            (\%i \mid \%i) \setminus n'', ++time, toWater.x, toWater.y);
        if(printSolution != dontPrintSolution)
317
          printSolution_TERMINAL(stdout, false);
318
319
        if (OUT) {
320
321
           fprintf(OUT, "---\nAt time %i: Water spot
              (%i|%i)\n",time,toWater.x,toWater.y);
322
          printSolution(OUT, false);
323
        }
324
325
      }
326
         printf("Fire died.\n");
327
328
      //Count the total number of burned or coaled
329
      int wcnt = 0, ccnt = 0;
      for(int i= 0; i < Forest.width(); ++i)</pre>
330
331
        for(int j= 0; j < Forest.height(); ++j)</pre>
332
           if(Forest(i, j) & WATERED)
333
             wcnt++;
           else if(Forest(i, j) & COAL)
334
335
             ccnt++;
336
337
      //Output / Mirror the solution
      printf("---\nAnd you'll find %i pieces of coal and %i pieces
338
         of watered coal\n", ccnt, wcnt);
      if(printSolution != dontPrintSolution)
339
340
        printSolution_TERMINAL(stdout, true);
341
      if (OUT) {
        fprintf(OUT, "---\nAnd you'll find %i pieces of coal and %i
342
            pieces of watered coal\n",ccnt,wcnt);
        printSolution(OUT, true);
343
344
      }
345
    }
346
    int main(int argc, char** argv){
347
      if (argc > 1) {
348
349
        freopen(argv[1], "r", stdin);
```

```
350
        printf("Using %s as input.\n", argv[1]);
351
      }
      if (argc > 2){
352
353
        printf("Mirroring output to %s.\n", argv[2]);
        if (strstr(argv[2], ".tex2")) {
354
          printf("I reckon you want me to produce some graphicless
355
             TeX stuff...\n");
          printSolution = dontPrintSolution;
356
357
358
        else if (strstr(argv[2], ".tex")) {
          printf("I reckon you want me to produce some TeX
359
             stuff...\n");
360
          printSolution = printSolution_TEX;
361
362
        else if (strstr(argv[2], ".raw")) {
363
          printf("I reckon you want me to surpress graphics...\n");
364
          printSolution = dontPrintSolution;
        }
365
366
        else
367
          printSolution = printSolution_TERMINAL;
368
        OUT = fopen(argv[2], "w");
369
      }
      else{
370
371
        OUT = 0;
372
        printSolution = printSolution_TERMINAL;
373
      }
374
375
      parseInput(stdin);
376
      simulateFire(getInitialBurningFields());
377 }
```