

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.

Viele¹ sind lösbar.

Inhaltsverzeichnis

1 Aufgabe 1 - Buschfeuer	3
1.1 Lösungsidee	3
1.1.1 Korrektheit	8
1.1.2 Laufzeitanalyse	10
1.2 Umsetzung	10
1.2.1 Eingabeformat	11
1.3 Beispiele	12
1.4 Quelltext	33
2 Aufgabe 2 - Lebenslinien	47
2.1 Lösungsidee	47
2.1.1 Eigenschaften von Lebensgraphen	47
2.1.2 Algorithmische Erkennung von Lebensgraphen	49
2.2 Laufzeitanalyse	50
2.3 Erweiterungen	50
2.4 Umsetzung	50
2.5 Beispiele	50
2.6 Quelltext	50

¹Von Zeit zu Zeit könnte ich an dieser Stelle auch ein *Wenige* vertreten...

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. Daher ist es nicht sinnvoll, verkohlte Felder zu löschen. Wird im Folgenden daher von brennenden Feldern gesprochen, so werden damit nicht verkohlte Felder gemeint.

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.

Brute-Force

Wie bei allen Problemen, ist natürlich auch bei diesem ein Brute-Force-Ansatz denkbar. Sei zunächst nur der Fall betrachtet, dass nur brennende Felder gelöscht werden dürfen.

Speziell bei diesem Problem würde der Brute-Force-Ansatz jeden möglichen Brandverlauf simulieren und dann aus all diesen Brandverläufen denjenigen ermitteln, bei welchem am Ende insgesamt die wenigsten Felder abgebrannt sind und diesen dann als Lösung ausgeben. Dies kann durch simples Backtracking erfolgen, dabei sollte bei den Löschungen noch der Zeitpunkt gespeichert werden, wann dies stattfand, damit die eigentliche Lösung dann am Ende rekonstruiert werden kann.

Dürfen nun auch andere, brennbare Felder gelöscht werden, so wäre es zunächst denkbar, einfach auch alle brennbaren Felder zu berücksichtigen. Dann würden aber in jedem Schritt schlimmstenfalls alle Felder in Betracht gezogen werden, dieses Verfahren ist wohl schon für kleine Felder nicht mehr praktikabel.

Dieses Brute-Force kann aber noch optimiert werden. Dazu kann man erkennen, dass in der optimalen Löschung kein gelöscht Feld existieren kann, welches nie gebrannt hat und welches vollständig von Feldern umgeben ist, die nie gebrannt haben. Eine solche Löschung wäre *verschwendet*.

Dies kann man sich bei der Berechnung nur zunutze machen. Anstatt bei jeder Beobachtung alle brennenden und alle brennbaren Felder zu berücksichtigen, werden wieder nur alle brennenden Felder berücksichtigt. Zusätzlich wird aber noch eine weitere Möglichkeit eingeführt, nämlich *vorerst* kein Feld zu löschen. Dabei wird die Anzahl der vorerst nicht gelöschten Felder R gespeichert. Erreicht das Backtracking nun einen Zustand, in dem die Anzahl der aktuell brennenden Felder r kleiner oder gleich $R + 1$ ist, so ist der aktuelle Zustand schon eine Lösung. Dabei werden R der r Felder bei einer Beobachtung gelöscht, bei der vorerst kein Feld gelöscht wurde, diese Felder werden also dann gelöscht, wenn sie noch brennbar sind; bei der endgültigen Lösung werden also $\min\{r, R\}$ Felder weniger verbrannt sein, als es eigentlich ermittelt wird, die muss bei der Prüfung auf Optimalität berücksichtigt werden.

Laufzeitanalyse Der Brute-Force-Ansatz probiert alle Möglichkeiten an verschiedenen Löschungen 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 Löschungen der Lösung².

State-Space-Search

Die Laufzeit des Brute-Force ist immer noch in den meisten Fällen inakzeptabel. Eine Möglichkeit der Optimierung ist die Folgende, sei dafür zunächst wieder der Fall betrachtet, dass nur brennende Felder gelöscht werden dürfen.

Aus jeder Beobachtung kann nur eine bestimmte Anzahl an anderen Beobachtungen entstehen. Dabei gibt es eine *Startbeobachtung*, nämlich die erste Beobachtung überhaupt. Auch gibt es letzte Beobachtungen, nach denen sich das Feuer nicht mehr ändert.

Es entsteht ein *Zustandsgraph* $Z = (B, E)$, welcher die Beobachtungen als Knoten hat und bei dem zwischen 2 Knoten eine Kante ist, genau dann, wenn es möglich ist von einer Beobachtung zu einer anderen gelangen kann; da sich das Feuer immer weiter ausbreitet ist der Graph also ein gerichteter, azyklischer Graph.

Ist der kürzeste Pfad von einer letzten Beobachtung zur Startbeobachtung kürzer, als der kürzeste Pfad von einer anderen letzten Beobachtung zur Startbeobachtung, so ist auch die Gesamtanzahl der brennenden Felder der ersten Lösung geringer als die der zweiten.

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

Nur unter den Lösungen, die gleich weit von der Startbeobachtung entfernt sind muss die Güte explizit verglichen werden.

Daraus lässt sich direkt ein Algorithmus ableiten. Von der Startbeobachtung wird eine BFS auf dem Zustandsgraphen gestartet. Dabei wird anstatt der Queue eine Priority Queue verwendet, welche die Elemente zuerst nach Entfernung von der Startbeobachtung und dann nach der Anzahl der brennenden Felder sortiert. Diese Verwendung der Breitensuche wird oft auch *State-Space-Search* genannt.

Wird das Problem auf diese Weise gelöst, so lässt sich auch überprüfen, ob es besser sein kann, nicht brennende Felder zu löschen. Dazu wird zusätzlich zu jeder Beobachtung noch eine weitere Zahl gespeichert. Diese Zahl gibt die Anzahl der Beobachtungen an, bei welchen keine Löschung durchgeführt wurde. Ist bei einer Beobachtung diese Zahl nun größer oder gleich als die verbleibende Anzahl an brennenden Feldern, so kann das gesamte Feuer gelöscht werden. Die Löschungen werden sozusagen "nach hinten verschoben". In der Realität würde dann keine Löschung ausgelassen sondern ein nicht brennendes Feld gelöscht werden. Dies wird also genau so abgehandelt, wie bei dem Brute-Force-Ansatz.

Mit dieser Änderung wird der Zustandsgraph etwas größer, der eigentliche Algorithmus funktioniert jedoch weiterhin.

Dass es sich überhaupt lohnen kann, brennbare Felder zu Löschen, wird mit Beispiel 2 gezeigt.

Korrektheit Im Folgenden wird davon ausgegangen, dass bei jeder Beobachtung ein Feld gelöscht wird.

Es genügt zu zeigen, dass es keine zwei Lösungen geben kann derart, dass die eine Lösung das Feuer nach k_1 Löschungen und die andere nach $k_2 > k_1$ Löschungen komplett löscht, und dass die Anzahl der insgesamt verbrannten Felder bei der 2. Lösung geringer ist als bei der ersten.

Angenommen L_1 und L_2 seien 2 Löschungen dieser Art, d.h. für die Anzahl der insgesamt verbrannten Felder A gilt $A(L_1) > A(L_2)$ und bei L_2 wurden insgesamt mehr Felder gelöscht.

Laufzeitanalyse Die Berechnung der Nachbarknoten einer Beobachtung im Zustandsgraphen benötigt schlimmstenfalls $\mathcal{O}(nm)$. Die State-Space-Search besucht wie eine normale Breitensuche schlimmstenfalls jeden Knoten im Zustandsgraphen 1 mal, bricht jedoch nach der ersten gefundenen Lösung ab. Somit werden maximal $\mathcal{O}((nm)^k)$ Berechnungen durchgeführt, wenn k die Anzahl der Löschungen in der Lösung ist. Somit ist dieser Algorithmus im Worst-Case-Szenario nicht besser als ein Brute-Force-Algorithmus; allerdings wird die Lösungssuche in der Regel stark geprunt.

Auch benötigt dieser Algorithmus schlimmsten exponentiell viel Speicherplatz.

Heuristik

Die State-Space-Search ist also schlimmstenfalls nicht besser als ein naiver Brute-Force-Ansatz. Gibt man sich jedoch auch mit einer vielleicht nicht optimalen Lösung zufrieden, so ist auch die folgende Heuristik denkbar.

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

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 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, 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 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
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 }

```

Bei dieser Heuristik ist es auch offensichtlich, dass es nicht sinnvoll ist, brennbare Felder für die Löschung in Betracht zu ziehen. Brennbare Felder werden trivialerweise immer ein geringeres oder gleiches μ besitzen als ein direkt benachbartes brennendes Feld, da für brennbare Felder nicht diejenigen Felder berücksichtigt werden dürfen, welche vor dem betrachteten Feld abbrennen, da dies sonst mit der inhaltlichen Erklärung im Widerspruch steht und das μ sonst eine total willkürliche Zahl wäre und sich aus dieser keine Informationen herauslesen lassen würden.

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.

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 μ 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 Lösung:

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	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 Ergebnisse³.

1.1.2 Laufzeitanalyse

Wie schon gesehen, haben der Brute-Force-Ansatz und die State-Space-Search schlimmstenfalls exponentielle Laufzeit, im Gegensatz zur Heuristik, wie im Folgenden gezeigt wird.

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 die State-Space-Search 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.⁴ 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

³Siehe dazu Sektion Beispiele

⁴Das Wort „definiert“ ist durchaus ernst zu nehmen, da es hier beschreiben soll, dass etwas mittels `#define` „gelöst“ wurde.

`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.

Die Implementierung der State-Space-Search kann in der Datei `Buschfeuer2.cpp` nachgelesen werden, dabei wird der Zustandsgraph nicht komplett vorberechnet, sonder erst just-in-time berechnet. Die Ein- und Ausgabe ist dabei die selbe wie bei dem anderen Algorithmus, auch wenn sich die Ausgabe auf die Endgültige Lösung beschränkt.

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 Dateiendung dieser Datei das Verhalten meines Programmes an:

1.3 Beispiele

Beispiel 0

Die ist das Beispiel aus der Aufgabenstellung. Umgewandelt für mein Programm sieht diese Eingabe folgendermaßen aus⁵:

```

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 Ausgabe⁶⁷:

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)

⁵Diese Eingabe finden Sie auch in der Datei 0.in

⁶Diese Ausgabe finden Sie auch in der Datei 0.out.tex bzw. 0-2.out.tex für die der State-Space-Search; Eine Datei 0.out mit den ASCII-Escape-Sequenzen existiert ebenfalls.

⁷Um 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.

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.

Die State-Space-Search findet eine andere, wenn auch genau so gute Lösung:

Water always optimally (to save water)... 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	02	FO	FO	FO	FO
FO	FO	WA	WA	WA	CO	WA	WA	WA	FO
FO	04	CO	CO	CO	CO	01	FO	FO	FO
FO	FO	WA	WA	CO	CO	CO	03	FO	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.

Beispiel 1

Eine Situation mit mehr als einem Feuer bei der ersten Beobachtung⁸:

```

1  10  11
2  3101111101
3  1001111110
4  1111111111
5  1100010001
6  1111131111
7  1100111111
8  1111011011
9  0111011010
10 1011011011
11 1111113111
12 1111111111

```

Die Heuristik produziert folgende Ausgabe⁹:

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)

⁸Diese Eingabe finden Sie auch in der Datei 1.in

⁹Diese Ausgabe finden Sie auch in der Datei 1.out.tex bzw. 1-2.out.tex für die der State-Space-Search; Eine Datei 1.out mit den ASCII-Escape-Sequenzen existiert ebenfalls.

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	FO	BU	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	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	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)

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	O3	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.

Die State-Space-Search findet eine bessere Lösung, benötigt dafür aber schon einige Se-

kunden:

Water always optimally (to save water)... And you'll find 42 pieces of coal and 6 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	06
FO	FO	WA	WA	WA	01	WA	WA	WA	CO
FO	04	CO	CO	CO	CO	CO	CO	CO	CO
FO	FO	WA	WA	CO	CO	CO	CO	CO	CO
FO	FO	FO	FO	WA	CO	CO	WA	CO	CO
WA	FO	FO	FO	WA	CO	CO	WA	CO	WA
FO	WA	FO	FO	WA	CO	CO	WA	CO	CO
FO	FO	FO	03	CO	CO	CO	CO	CO	CO
FO	FO	05	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.

Beispiel 2

¹⁰:

```

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 Ausgabe¹¹:

¹⁰Diese Eingabe finden Sie auch in der Datei 2.in

¹¹Diese Ausgabe finden Sie auch in der Datei 2.out.tex bzw. 2-2.out.tex für die der State-Space-Search; 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 (6|5)

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	O1	FO	FO	FO	FO	WA	FO
FO	FO	FO	FO	FO	BU	CO	BU	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 (4|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	BU	O1	BU	FO	FO	FO	WA	FO
FO	FO	FO	FO	O2	CO	CO	CO	BU	FO	FO	FO	FO
FO	WA	FO	FO	FO	BU	CO	BU	FO	FO	FO	WA	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	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 (6|9)

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	BU	FO	BU	FO	FO	FO	WA	FO
FO	WA	FO	FO	BU	CO	01	CO	BU	FO	FO	WA	FO
FO	FO	FO	FO	02	CO	CO	CO	CO	BU	FO	FO	FO
FO	WA	FO	FO	BU	CO	CO	CO	BU	FO	FO	WA	FO
FO	WA	FO	FO	FO	BU	CO	BU	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	FO	03	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 (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	FO	FO	FO	FO	FO	FO	FO	WA	FO
FO	WA	FO	FO	FO	BU	FO	BU	FO	FO	FO	WA	FO
FO	WA	FO	FO	BU	CO	BU	CO	BU	FO	FO	WA	FO
FO	WA	FO	BU	CO	CO	01	CO	CO	BU	FO	WA	FO
FO	FO	FO	FO	02	CO	CO	CO	CO	CO	04	FO	FO
FO	WA	FO	BU	CO	CO	CO	CO	CO	BU	FO	WA	FO
FO	WA	FO	FO	BU	CO	CO	CO	BU	FO	FO	WA	FO
FO	WA	FO	FO	FO	BU	03	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 (5|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	05	FO	BU	FO	FO	FO	WA	FO
FO	WA	FO	FO	BU	CO	BU	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	01	CO	CO	CO	BU	WA	FO
FO	FO	FO	BU	02	CO	CO	CO	CO	CO	04	FO	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	FO	FO	BU	CO	03	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 (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	BU	05	06	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	01	CO	CO	CO	CO	WA	FO
FO	FO	BU	CO	02	CO	CO	CO	CO	CO	04	FO	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	FO	BU	CO	CO	03	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 (1|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	BU	CO	05	06	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	01	CO	CO	CO	CO	WA	FO
FO	07	CO	CO	02	CO	CO	CO	CO	CO	04	FO	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	BU	CO	CO	CO	03	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	BU	WA	WA	WA	WA	WA	FO
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO	FO

--- At time 8: Water spot (6|12)

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

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

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

Die State-Space-Search versucht sich an diesem Beispiel, scheitert aber an dem bei mir zu wenig vorhandenen RAM (j 2 GiB). Deshalb wird im Folgenden die eigentlich erwartete Lösung aufgezeigt.

Dies ist auch ein Beispiel dafür, dass es besser sein kann, brennbare Felder zu löschen.

Water always optimally (to save water)... And you'll find 73 pieces of coal and 3 pieces of watered coal

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

Explanation:

WA --- EMPTY

FO --- BURNABLE

BU --- BURNED

[illegible]

[illegible]

Mein Programm produziert folgende Ausgabe¹³. 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)
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)

¹³Diese Ausgabe finden Sie auch in der Datei `3.out.tex2`;

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
(Ich warte noch heute auf das Ergebnis der State-Space-Search...)

Beispiel 3b

Beispiel 3, diesmal jedoch (etwas) kleiner.¹⁴:

¹⁴Diese Eingabe finden Sie auch in der Datei 7.in

```

1 5 5
2 11111
3 11111
4 11311
5 11111
6 11111

```

Mein Programm produziert folgende Ausgabe¹⁵:

FO	FO	FO	FO	FO
FO	FO	FO	FO	FO
FO	FO	BU	FO	FO
FO	FO	FO	FO	FO
FO	FO	FO	FO	FO

--- At time 1: Water spot (2|1)

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

--- At time 2: Water spot (1|1)

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

--- At time 3: Water spot (0|1)

FO	FO	FO	BU	FO
O3	O2	O1	CO	BU
CO	CO	CO	CO	CO
BU	CO	CO	CO	BU
FO	BU	CO	BU	FO

--- At time 4: Water spot (2|0)

FO	FO	O4	CO	BU
O3	O2	O1	CO	CO
CO	CO	CO	CO	CO
CO	CO	CO	CO	CO
BU	CO	CO	CO	BU

--- And you'll find 19 pieces of coal and 4 pieces of watered coal

¹⁵Diese Ausgabe finden Sie auch in der Datei 7.out.tex bzw. 7-2.out.tex für die der State-Space-Search

FO	FO	04	CO	CO
03	02	01	CO	CO
CO	CO	CO	CO	CO
CO	CO	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.

Die State-Space-Search:

Water always optimally (to save water)... And you'll find 19 pieces of coal and 4 pieces of watered coal

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

Explanation:

WA --- EMPTY

FO --- BURNABLE

BU --- BURNED

CO --- COAL (doubly burned)

--- WATERED at time

Fields can have more than 1 state.

Beispiel X

Ein die Heuristik zerstörendes Beispiel.¹⁶:

```

1  10 10
2  1111111111
3  1111111111
4  1111111111
5  1110101111
6  1110301111
7  1101110111
8  1101110111
9  1101110111
10 1110001111
11 1111111111

```

¹⁶Diese Eingabe finden Sie auch in der Datei `x.in`

Mein Programm produziert folgende Ausgabe¹⁷:

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|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)

¹⁷Diese Ausgabe finden Sie auch in der Datei `x.out.tex` bzw. `x-2.out.tex` für die der State-Space-Search

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.

Die State-Space-Search:

Water always optimally (to save water)... 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.

1.4 Quelltext

Woods.h

```

1 #include <vector>
2 #include <string>
3
4 typedef unsigned short u16;
5
6 #define FIELDSTATE      u16
7 #define EMPTY          0
8 #define BURNABLE       1
9 #define BURNED         2
10 #define WATERED        4
11 #define COAL           8
12
13 class Woods{
14     private:
15         int Width, Height;
16         std::vector<std::vector<FIELDSTATE> > Fields;
17
18     public:
19         Woods(int width, int height);
20
21         int width() const;
22         int height() const;
23
24         FIELDSTATE& operator() (int x, int y);
25         FIELDSTATE operator() (int x, int y) const;
26
27         bool operator==(const Woods& o) const;

```

```
28     bool operator< (const Woods& o) const;
29
30     std::string string() const;
31     int cnt() const;
32     int cnt2() const;
33 };
```

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
40     return (string() == o.string());
41 }
42
43 bool Woods::operator <(const Woods& o) const {
```

```
44     if(std::min(o.width() , o.height()) < std::min(width(),
45         height()))
46         return true;
47     if(std::max(o.width() , o.height()) < std::max(width(),
48         height()))
49         return true;
50     if(std::min(o.width() , o.height()) > std::min(width(),
51         height()))
52         return false;
53     if(std::max(o.width() , o.height()) > std::max(width(),
54         height()))
55         return false;
56     for(int i = 0; i < width(); ++i)
57         for(int j = 0; j < height(); ++j)
58             if(o(i,j) <= Fields[i][j])
59                 return false;
60     return true;
61 }
62
63 extern int dir[4][2];
64 std::string Woods::string() const {
65     std::string ret;
66
67     for(int i = 0; i < width(); ++i)
68         for(int j = 0; j < height(); ++j){
69             u16 andMask = (BURNABLE | BURNED | COAL);
70
71             for(int k = 0; k < 4; ++k){
72                 int nx = i + dir[i][0], ny = j + dir[i][1];
73
74                 if(nx < 0 || ny < 0 || nx >= width() || ny >= height())
75                     continue;
76                 if(!(Fields[nx][ny] & (COAL | BURNED | WATERED))){
77                     andMask |= (WATERED); //Only watered fields at the
78                                             border of the fire are of interest
79                     break;
80                 }
81             }
82             ret += (char) (((Fields[i][j] & andMask)) + 'A');
83         }
84     return ret;
85 }
86
87 int Woods::cnt() const { //counts the burningfields
88     int ret = 0;
89     for(int i = 0; i < width(); ++i)
90         for(int j = 0; j < height(); ++j)
91             if(Fields[i][j] & BURNED && !(Fields[i][j] & COAL))
92                 ret++;
93     return ret;
94 }
```

```
92 }
93
94 int Woods::cnt2() const { //counts the burningfields
95     int ret = 0;
96     for(int i = 0; i < width(); ++i)
97         for(int j = 0; j < height(); ++j)
98             if(Fields[i][j] & BURNED)
99                 ret++;
100     return ret;
101 }
```

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); //
9     The infinity
10 Woods Forest(0, 0);
11
12 struct Point {
13 public:
14     int x, y;
15     Point(int _x,int _y) : x(_x), y(_y) { }
16 };
17
18 int dir[4][2] = {{1,0},{0,1},{-1,0},{0,-1}};
19
20 std::FILE* OUT;
21     // The file to mirror the output to
22 void (*printSolution)(std::FILE*, bool);
23
24 //BEGIN OF INPUT
25 void parseInput(std::FILE* f) {
26     int acFieldWidth, acFieldHeight;
27     std::fscanf(f, "%i %i\n",&acFieldWidth, &acFieldHeight);
28
29     Forest = Woods(acFieldWidth, acFieldHeight);
30
31     for(int i = 0; i < acFieldHeight; ++i){
32         for(int j = 0; j < acFieldWidth; ++j){
33             char c;
34             std::fscanf(f, "%c",&c);
35             c -= '0';
36             Forest(j, i) = (FIELDSTATE) c;
37         }
38         if(i < acFieldHeight-1)
39             std::fscanf(f, "\n");
40     }
41 }
```

```

39 //END OF INPUT
40 //BEGIN OF OUTPUT
41 void printSolution_TEX(std::FILE* f, bool finalOut) {
42     std::fprintf(f, "\\\\n");
43
44     std::fprintf(f, "\\begin{tikzpicture}\n");
45     std::fprintf(f, "\\tikzset{square matrix/.style={\n");
46     std::fprintf(f, "matrix of nodes,\n");
47     std::fprintf(f, "column sep=-\\pgflinewidth, row
        sep=-\\pgflinewidth,\n");
48     std::fprintf(f, "nodes={draw,\n");
49     std::fprintf(f, "minimum height=#1,\n");
50     std::fprintf(f, "anchor=center,\n");
51     std::fprintf(f, "text width=#1,\n");
52     std::fprintf(f, "align=center,\n");
53     std::fprintf(f, "inner sep=0pt\n");
54     std::fprintf(f, "},\n");
55     std::fprintf(f, "},\n");
56     std::fprintf(f, "square matrix/.default=1.2cm\n");
57     std::fprintf(f, "}\n");
58
59     std::fprintf(f, "\\matrix[square matrix=1.4em] {\n");
60     for(int j= 0; j < Forest.height(); ++j) {
61         for(int i= 0; i < Forest.width(); ++i) {
62             if(i)
63                 std::fprintf(f, " &");
64
65             FIELDSTATE acField = Forest(i, j);
66             if(acField == EMPTY)
67                 std::fprintf(f, "|[fill=white]|");
68             else if(acField & WATERED)
69                 std::fprintf(f, "|[fill=cyan]|");
70             else if(acField & BURNABLE)
71                 std::fprintf(f, "|[fill=green]|");
72
73             if(acField & COAL)
74                 std::fprintf(f, "\\color[rgb]{0,0,0}");
75             else if(acField & BURNED)
76                 std::fprintf(f, "\\color[rgb]{1,0,0}");
77             else if(acField == EMPTY)
78                 std::fprintf(f, "\\color[gray]{0.5}");
79             else if(acField & BURNABLE)
80                 std::fprintf(f, "\\color[gray]{0.75}");
81
82             if(acField & WATERED)
83                 std::fprintf(f, "\\textbf{\\texttt{%02d}}", acField >>
                    4);
84             else if(acField & COAL)
85                 std::fprintf(f, "\\textbf{\\texttt{C0}}");
86             else if(acField & BURNED)
87                 std::fprintf(f, "\\textbf{\\texttt{BU}}");
88             else if(acField == EMPTY)
89                 std::fprintf(f, "\\texttt{WA}");

```

```

90         else if(acField & BURNABLE)
91             std::fprintf(f, "\\texttt{FO}");
92         else
93             std::fprintf(f, "\\phantom{AA}");
94         std::fprintf(f, "%%\n");
95     }
96
97     std::fprintf(f, "\\n");
98 }
99
100 std::fprintf(f, "};\n\\end{tikzpicture}\\n");
101
102 if(finalOut){
103     std::fprintf(f, "\\nExplanation:");
104     std::fprintf(f,
105         "\\n\\colorbox{white}{\\color[gray]{0.5}WA} ---
106         EMPTY");
107     std::fprintf(f,
108         "\\n\\colorbox{green}{\\color[gray]{0.5}FO} ---
109         BURNABLE");
110     std::fprintf(f,
111         "\\n\\colorbox{white}{\\color[rgb]{1,0,0}\\textbf{BU}}
112         --- BURNED");
113     std::fprintf(f,
114         "\\n\\colorbox{white}{\\color[rgb]{0,0,0}\\textbf{CO}}
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     std::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             std::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                 std::fprintf(f, "\x1b[u\x1b[46m%02d", acField >> 4);
141             std::fprintf(f, "\x1b[0;39;49m");
142         }
143     }

```

```

134
135     std::fprintf(f, "\n");
136 }
137
138 if (finalOut) { // An Explanation shall be printed
139     std::fprintf(f, "\nExplanation:");
140     std::fprintf(f, "\n\x1b[37;47mWA\x1b[39;49m --- EMPTY");
141     std::fprintf(f, "\n\x1b[32;42mFO\x1b[39;49m --- BURNABLE");
142     std::fprintf(f, "\n\x1b[1;5;31m/\x1b[0;39m --- BURNED");
143     std::fprintf(f, "\n\x1b[1;4;5;30m/\x1b[0;39m --- COAL
        (doubly burned)");
144     std::fprintf(f, "\n\x1b[46m#\x1b[0;39m --- WATERED at
        time ##");
145     std::fprintf(f, "\nFields can have more than 1 state.");
146 }
147 std::fprintf(f, "\n");
148 }
149
150 void dontPrintSolution(std::FILE* f, bool finalOut) { return; }
151 //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         );
23     std::vector<std::set<int> >(Forest.height());
24     std::vector<std::vector<int> > shortDis(Forest.width(),
25         // shortest distant to any burning field
26     );
27     std::vector<int>(Forest.height(),oo);
28
29     //BFS to calculate shortest paths
30     while(!q.empty()){
31         std::pair<PII,Point> ac = q.front();
32         Point acPoint = ac.second;

```

```
25     int acDistance = ac.first.first;
26     int acColor = ac.first.second;
27
28     q.pop();
29     if(visited[acPoint.x][acPoint.y].count(acColor))
30         continue;
31     visited[acPoint.x][acPoint.y].insert(acColor);
32     shortDis[acPoint.x][acPoint.y] =
        std::min(acDistance, shortDis[acPoint.x][acPoint.y]);
33
34     for(int i= 0; i < 4; ++i){
35         int newx = acPoint.x + dir[i][0];
36         int newy = acPoint.y + dir[i][1];           //
            calculate new field's indexes
37
38         if(newx < 0 || newy < 0 || newy >= Forest.height() || newx
            >= Forest.width())
39             continue;                               //
                new field is outside the woods
40         if (Forest(newx, newy) != BURNABLE)
41             continue;                               //
                Field is not of interest
42
43         if(visited[newx][newy].count(acColor) == 0) //
            Don't compute things twice
44         if(acDistance + 1 <= shortDis[newx][newy]){
45             shortDis[newx][newy] = acDistance + 1;
46             q.push(std::pair<PII, Point>(PII(acDistance +
                1, acColor), Point(newx, newy)));
47         }
48     }
49 }
50
51 //determine the field to be watered
52 std::vector<PII> waterval(candidates.size(), PII(0,0));
53 std::vector<std::vector<int>> > farthDist(candidates.size(),
    std::vector<int>(2*(Forest.width()+Forest.height()),0));
54
55 //Count the number of fields that have an unique fire spot
    a.k.a. waterval
56 //Reckon the farthest field
57 for(int i= 0; i< Forest.width(); ++i)
58     for(int j= 0; j < Forest.height(); ++j)
59         if(visited[i][j].size() == 1){
60             waterval[*visited[i][j].begin()].first++;
61             farthDist[*visited[i][j].begin()][shortDis[i][j]]++;
62         }
63 for(int i = 0; i < waterval.size(); ++i)
64     waterval[i].second = i;
65
66 std::sort(waterval.begin(), waterval.end(), std::greater<PII>());
67
68 return candidates[waterval[0].second];
```



```
69 }
70
71 std::vector<Point>& getInitialBurningFields() {
72     static std::vector<Point> burnedFields;
73
74     for(int i = 0; i < Forest.height(); ++i)
75         for(int j = 0; j < Forest.width(); ++j)
76             if(Forest(j, i) & BURNED){
77                 burnedFields.push_back(Point(j, i));
78 //             printf("Initially burning: (%i|%i)\n",j, i);
79             }
80     return burnedFields;
81 }
82
83 void simulateFire(const std::vector<Point>&
84     initiallyBurningFields) {
85     std::vector<Point> burnedFields = initiallyBurningFields;
86     if(printSolution != dontPrintSolution)
87         printSolution_TERMINAL(stdout, false);
88     if (OUT != 0)
89         printSolution(OUT, false);
90
91     int time = 0;
92     while(!burnedFields.empty()) {
93         // Simulate as long as there's still fire in the world
94         std::vector<Point> newBurnedFields;
95         // The burning fields at the next point of time
96
97         //Calculate the new burning fields
98         for(size_t i = 0; i < burnedFields.size(); ++i){
99             int acx = burnedFields[i].x;
100             int acy = burnedFields[i].y;
101
102             if(Forest(acx, acy) & WATERED)
103                 continue;
104             // The field got watered and does not spread fire
105             Forest(acx, acy) |= COAL;
106             // Field burned down to coal...
107
108             for(int j = 0; j < 4; ++j) {
109                 int newx = acx + dir[j][0];
110                 int newy = acy + dir[j][1];
111
112                 if(newx < 0 || newy < 0 || newy >= Forest.height() ||
113                     newx >= Forest.width())
114                     continue;
115                 // new field is outside the woods
116                 if(Forest(newx, newy) == BURNABLE){
117                     Forest(newx, newy) |= BURNED;
118                     // Field starts burning
119                     newBurnedFields.push_back(Point(newx, newy));
120                 }
121             }
122         }
123     }
124 }
```

```
113 //      printf("  From now on burning: (%i|%i)\n",newx,newy);
      // log the happenings
114     }
115 }
116 }
117 if(newBurnedFields.empty()) //
    Nothing to water, all plants happy...
118     break;
119
120 burnedFields = newBurnedFields;
121
122 Point toWater = getOptimalWaterSpot(newBurnedFields); //
    Determine the field to water
123 Forest(toWater.x, toWater.y) |= (WATERED | (time << 4));
    // ... and water it
124
125
126 //Output / mirror the partial solution
127
128 std::printf("---\nAt time %i: Water spot
    (%i|%i)\n",++time,toWater.x,toWater.y);
129 if(printSolution != dontPrintSolution)
130     printSolution_TERMINAL(stdout, false);
131
132 if (OUT) {
133     std::fprintf(OUT, "---\nAt time %i: Water spot
        (%i|%i)\n",time,toWater.x,toWater.y);
134     printSolution(OUT, false);
135 }
136
137 }
138 // printf("Fire died.\n");
139
140 //Count the total number of burned or coaled
141 int wcnt = 0, ccnt = 0;
142 for(int i = 0; i < Forest.width(); ++i)
143     for(int j = 0; j < Forest.height(); ++j)
144         if(Forest(i, j) & WATERED)
145             wcnt++;
146         else if(Forest(i, j) & COAL)
147             ccnt++;
148
149 //Output / Mirror the solution
150 std::printf("---\nAnd you'll find %i pieces of coal and %i
    pieces of watered coal\n",ccnt,wcnt);
151 if(printSolution != dontPrintSolution)
152     printSolution_TERMINAL(stdout, true);
153 if (OUT) {
154     std::fprintf(OUT, "---\nAnd you'll find %i pieces of coal
        and %i pieces of watered coal\n",ccnt,wcnt);
155     printSolution(OUT, true);
156 }
157 }
```

```

158
159 int main(int argc, char** argv){
160     if (argc > 1) {
161         std::freopen(argv[1], "r", stdin);
162         std::printf("Using %s as input.\n", argv[1]);
163     }
164     if (argc > 2){
165         printf("Mirroring output to %s.\n", argv[2]);
166         if (strstr(argv[2], ".tex2")) {
167             std::printf("I reckon you want me to produce some
168                 graphicless TeX stuff...\n");
169             printSolution = dontPrintSolution;
170         }
171         else if (strstr(argv[2], ".tex")) {
172             std::printf("I reckon you want me to produce some TeX
173                 stuff...\n");
174             printSolution = printSolution_TEX;
175         }
176         else if (strstr(argv[2], ".raw")) {
177             std::printf("I reckon you want me to surpress
178                 graphics...\n");
179             printSolution = dontPrintSolution;
180         }
181         else
182             printSolution = printSolution_TERMINAL;
183         OUT = std::fopen(argv[2], "w");
184     }
185     else{
186         OUT = 0;
187         printSolution = printSolution_TERMINAL;
188     }
189     parseInput(stdin);
190     simulateFire(getInitialBurningFields());
191 }

```

Buschfeuer2.cpp Dies ist die Implementierung der State-Space-Search.

```

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 typedef std::pair<int,int> pii;
15

```

```
15 bool operator <(const sssType &a, const sssType &b){
16
17     int aBurning = std::get<2>(a).cnt();
18     int bBurning = std::get<2>(b).cnt();
19
20     int aBurned = std::get<2>(a).cnt2();
21     int bBurned = std::get<2>(b).cnt2();
22
23     auto aSk = std::get<1>(a);
24     auto bSk = std::get<1>(b);
25
26     return (aBurned > bBurned || (aBurned == bBurned && (aBurning
        - aSk > bBurning - bSk)));
27 }
28
29 std::pair<Woods, std::vector<Point>> getNextState(const Woods& w){
30     //Calculate the next state's fire paired with the positions
31     //of the new fire spots
32     Woods ret(w.width(), w.height());
33     std::vector<Point> pnts;
34
35     for(int i = 0; i < w.width(); ++i)
36         for(int j = 0; j < w.height(); ++j){
37             ret(i, j) = w(i, j);
38             if(w(i, j) == BURNABLE){
39                 bool startsBurning = false;
40                 for(int k = 0; k < 4; ++k){
41                     int x = i + dir[k][0];
42                     int y = j + dir[k][1];
43                     if(x < 0 || y < 0 || x >= w.width() || y >= w.height())
44                         continue;
45                     if(w(x, y) & BURNED && !(w(x, y) & WATERED))
46                         startsBurning = true;
47                 }
48                 if(startsBurning)
49                     ret(i, j) |= BURNED;
50             }
51             else if(w(i, j) & BURNED && !(w(i, j) & WATERED))
52                 ret(i, j) |= COAL;
53         }
54     return std::pair<Woods, std::vector<Point>>(ret, pnts);
55 }
56
57 int main(int argc, char ** argv){
58     if (argc > 1) {
59         std::freopen(argv[1], "r", stdin);
60         std::printf("Using %s as input.\n", argv[1]);
61     }
62     if (argc > 2){
63         std::printf("Mirroring output to %s.\n", argv[2]);
64         if (strstr(argv[2], ".tex2")) {
65             std::printf("I reckon you want me to produce some
                graphicless TeX stuff...\n");
66         }
67     }
68 }
```

```
66     printSolution = dontPrintSolution;
67 }
68 else if (strstr(argv[2], ".tex")) {
69     std::printf("I reckon you want me to produce some TeX
70                 stuff...\n");
71     printSolution = printSolution_TEX;
72 }
73 else if (strstr(argv[2], ".raw")) {
74     std::printf("I reckon you want me to surpress
75                 graphics...\n");
76     printSolution = dontPrintSolution;
77 }
78 else
79     printSolution = printSolution_TERMINAL;
80 OUT = std::fopen(argv[2], "w");
81 }
82 else{
83     OUT = 0;
84     printSolution = printSolution_TERMINAL;
85 }
86
87 parseInput(stdin);
88
89 std::priority_queue<sssType> q; //The SSS-Queue
90 q.push(sssType(1,0,Forest)); //Initial Node, let time start at
91 1
92
93 while(!q.empty()) {
94     sssType ac = q.top(); q.pop();
95
96     auto acForest = std::get<2>(ac);
97     int acDis = std::get<0>(ac);
98     int acSkipped = std::get<1>(ac);
99
100     auto next = getNextState(acForest);
101
102     if(acForest.cnt() <= acSkipped){ //Fire can be dead by this
103         time
104         //Reconstruct and output Solution
105
106         Forest = acForest;
107
108         std::set<int> remTimes; //check at which points in time a
109             watering was required
110             //so that those fields that are still
111             burning can get one of the
112             //remaining times to be watered
113
114         for(int i = 1; i<= acDis; ++i)
115             remTimes.insert(i);
116
117         for(int i = 0; i< Forest.width(); ++i)
```

```
113         for(int j = 0; j < Forest.height(); ++j)
114             if((Forest(i,j) & BURNED) && ( Forest(i,j) & WATERED )
115                 )
116                 remTimes.erase(remTimes.find(Forest(i,j) >> 4));
117
118     int ccnt = 0;    //No of coaled tiles
119
120     for(int i = 0; i< Forest.width(); ++i)
121         for(int j = 0; j < Forest.height(); ++j)
122             if((Forest(i,j) & BURNED) && !( Forest(i,j) & (WATERED
123                 | COAL )) ){
124                 Forest(i,j) |= (WATERED | ((*remTimes.begin()) <<
125                     4));
126                 remTimes.erase(remTimes.begin());
127             }
128             else if ((Forest(i,j) & COAL) && !( Forest(i,j) &
129                 WATERED) )
130                 ccnt++;
131
132     std::printf("Water always optimally (to save
133         water)...\nand you'll find %i pieces of coal and %i
134         pieces of watered coal\n",ccnt,acDis-acSkipped -1);
135
136     if(printSolution != dontPrintSolution)
137         printSolution_TERMINAL(stdout, true);
138     if (OUT) {
139         std::fprintf(OUT, "Water always optimally (to save
140             water)...\nAnd you'll find %i pieces of coal and %i
141             pieces of watered coal\n",ccnt,acDis- acSkipped-1);
142         printSolution(OUT, true);
143     }
144     break;
145 }
146
147 q.push(sssType(acDis + 1,acSkipped + 1, next.first));
148 for(auto i : next.second){
149     next.first(i.x,i.y) |= (WATERED | (acDis << 4)); // Save
150         time whe field got watered
151     q.push(sssType(acDis+ 1,acSkipped, next.first));
152     next.first(i.x,i.y) &= ~(WATERED | (acDis << 4));
153 }
154 }
155 printf("Finished\n");
156 }
```

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 $f : V \mapsto P(\mathbb{R})$ ¹⁸ 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.

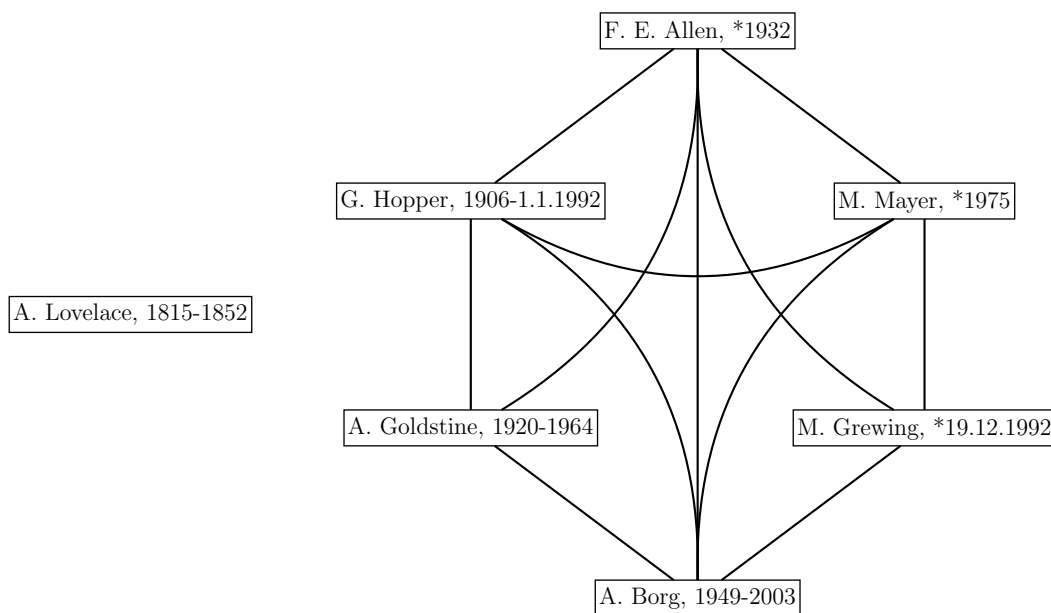


Abbildung 1: Der Lebensgraph aus der Aufgabenstellung

Im Folgenden wird nur von zusammenhängenden Graphen ausgegangen. Für aus mehreren Zusammenhangskomponenten bestehende Graphen lässt sich die Berechnung für jede dieser einzeln durchführen, eventuell muss allen einer Komponente zugewiesenen Intervalle eine reelle Konstante addiert werden, dies ändert jedoch nichts an der eigentlichen Lösung.

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

¹⁸ $P(\mathbb{R})$ beschreibt die Potenzmenge von \mathbb{R} , also die Menge aller Teilmengen von \mathbb{R}

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 ein *Loch*¹⁹ auftritt niemals Lebensgraph sein kann:

Loch

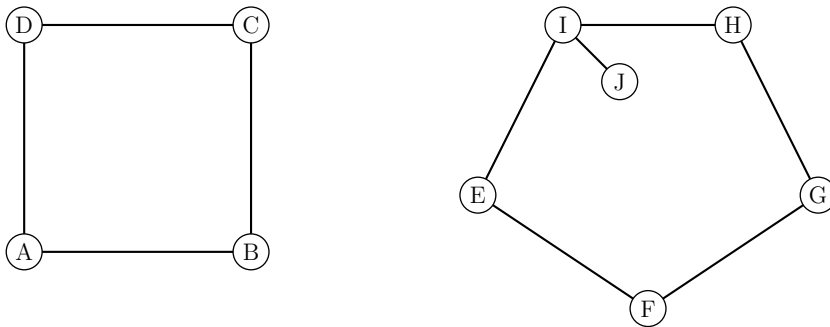


Abbildung 2: Graphen mit Löchern können niemals Lebensgraph sein (*im 2. Graphen ist J nicht Teil des Loches*)

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*²⁰ genannt, es gibt effiziente Algorithmen zur Erkennung solcher Graphen.

Chordalgraph

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.

Dreiecke
Clique

¹⁹Ein Loch ist dabei ein Zyklus mit einer Länge größer 3, zwischen dessen einzelnen Knoten nur eine Kante existiert, wenn diese auch im Zyklus existiert.

²⁰Der englischsprachige Wikipediaartikel ist in diesem Fall (mal wieder) deutlich informativer: https://en.wikipedia.org/wiki/Chordal_graph

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

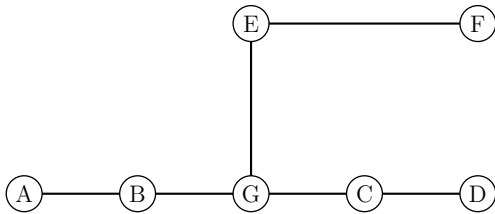


Abbildung 3: Chordalgraph, der **kein** Lebensgraph ist

Es gilt nun also ein *hinreichendes Kriterium* für Lebensgraphen zu finden.

Sei dazu der *Komplementärgraph* \overline{G} ²¹ eines Lebensgraphen G betrachtet. Ist zwischen 2 Knoten v_i, v_j in \overline{G} nun eine Kante, so bedeutet dies für den Schnitt der diesen Knoten zugeordneten Lebenszeiten I_{v_i}, I_{v_j} , dass dieser leer ist, $I_{v_i} \cap I_{v_j} = \emptyset$. Somit muss sich also eines dieser beiden Lebenszeiten I_{v_i}, I_{v_j} auf der reellen Zahlengeraden vor dem anderen befinden. Dabei sei durch $I_{v_i} <_I I_{v_j}$ beschrieben, dass sich das Intervall I_{v_j} nach dem Intervall I_{v_i} auf der reellen Zahlengeraden befindet.

Für den Graphen $\overline{G} = (V, \overline{E})$ ist also eine Halbordnung $<_I$ definiert, mit $\forall v_i, v_j \in V : (v_i, v_j) \in \overline{E} \Leftrightarrow (I_{v_i} <_I I_{v_j} \vee I_{v_j} <_I I_{v_i})$.

Bei dem Graphen \overline{G} zusammen mit der Halbordnung $<_I$ spricht man in der Literatur²² von *Vergleichbarkeitsgraph*.

Vergleichbar-
keitsgraph

Es ist nun möglich zu zeigen, dass ein Lebensgraph genau derjenige Graph ist, der ein Chordalgraph ist, und dessen Komplementärgraph ein Vergleichbarkeitsgraph mit obiger Halbordnung ist. Auf den Beweis sei an dieser Stelle verzichtet, dieser kann in der Literatur²³ nachgelesen werden.

2.1.2 Algorithmische Erkennung von Lebensgraphen

Der vorangegangene Abschnitt liefert nun einen direkten Algorithmus zur Überprüfung, ob ein Graph ein Lebensgraph ist. Zunächst wird überprüft, ob der gegebene Graph ein Chordalgraph ist, dann wird überprüft, ob der Komplementärgraph ein Vergleichbarkeitsgraph ist. Idealerweise sollte bei der Überprüfung gleich eine mögliche Zuordnung von Lebenszeiten bzw. Intervallen zu Knoten abfallen, auch wenn dies noch nicht direkt ersichtlich ist.

Die Überprüfung, ob ein gegebener Graph ein Chordalgraph ist, kann mithilfe einer *lexikografischen Breitensuche* (im Folgenden Lex-BFS) geschehen. Dabei ist eine Lex-BFS ähnlich einer normalen Breitensuche. Anstatt einer Warteschlange (Queue) verwendet die Lex-BFS jedoch eine geordnete Folge von Knotenmengen. Die Lex-BFS wird speziell dazu benutzt, eine spezielle *Abfolge* der Knoten zu erhalten, mit welcher im Folgenden dann weiter operiert werden kann.

Lex-
BFS

²¹ \overline{G} besitzt die selben Knoten wie G , \overline{G} hat jedoch nur genau dann eine Kante zwischen zwei Knoten, wenn zwischen diesen Knoten in G keine Kante ist.

²²<https://de.wikipedia.org/wiki/Vergleichbarkeitsgraph>

²³Gilmore, P. C.; Hoffman, A. J. (1964), "A characterization of comparability graphs and of interval graphs", Canadian Journal of Mathematics 16: 539–548, <http://cms.math.ca/cjm/v16/cjm1964v16.0539-0548.pdf>. (Der Beweis ist eine direkte Schlussfolgerung aus Theorem 2, zusammen mit Theorem 1 und der definierenden Eigenschaft von Chordalgraphen; Lebensgraphen heißen dort *Intervallgraphen*)

```
1 //Lex-BFS
2 //Eingabe: Graph G = (V,E), Knoten seien durchnummeriert 0..|V|-1
3 //Ausgabe: Reihenfolge der Knoten
4
5 begin
6   int[] ausgabe := int[|V|];
7
8   Liste<int> L := V; //Initiale Anordnung der Knoten (L[i] = i)
9
10  Liste<int>[] S := {L}; //Klassen
11
12  int cnt = |V| - 1; //Zähler für Ausgabe
13  while S != { } do begin
14    int x := letztes Element der letzten Klasse in S;
15    entferne x aus der letzten Klasse in S,
16    wird diese Klasse dadurch leer, entferne diese aus S;
17
18    ausgabe[x] := cnt; cnt := cnt - 1;
19
20    //Klassen werden in 2 Teilklassen aufgespalten:
21    //diejenigen Knoten, die Nachbar von x sind,
22    //und die, die es nicht sind
23
24    foreach Liste<int> i in S do begin
25      nachbarn := { Knoten in i, die benachbart zu x };
26      nicht_nachbarn := i \ nachbarn;
27
28      //Ordne Nachbarn vor Nicht-Nachbarn in S
29      ersetze { i } durch { nachbarn , nicht_nachbarn } in S;
30      //Ignoriere leere Mengen
31    end;
32  end;
33  return ausgabe;
34 end.
```

2.2 Laufzeitanalyse

2.3 Erweiterungen

2.4 Umsetzung

2.5 Beispiele

2.6 Quelltext