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

Diese Welt ist voller Rätsel und Probleme.
Viele[1] sind lösbar.

Inhaltsverzeichnis

[1] Von 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: Feld

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 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. 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.

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

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 stattfanden, 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 gelöschtes 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 Lösung[2].

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.

[2]Es 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 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 Lösungen 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.

Fall 1: L_2 enthält gelöschte Felder, welche nur von noch nie brennenden Feldern umgeben sind. Existieren derartige Felder in L_2, so sind die Löschungen, die diese Felder löschen überflüssig, d.h., es wäre höchstens besser gewesen bei den entsprechenden Beobachtungen ein anderes Feld zu löschen. Sei daher davon ausgegangen, dieser Fall existierte nicht.
Sei auch weiterhin davon ausgegangen, dass wenn sich das gesamte Feuer nicht weiter ausbreitet, dass dann keine weiteren Felder mehr gelöscht werden; dies wäre Wasserverschwendung. Konsequenterweise seinen diese Einschränkungen auch bei L_1 angewandt.

Fall 2: Jedes gelöschte Feld in L_2 enthält mindestens 1 Feld in seiner Umgebung, welches bei mindestens einer Beobachtung brannte. Die State-Space-Serach findet die Lösung L_1. Da L_2 mehr Löschungen enthält, muss das Feuer über mehr Beobachtungen hin gebrannt haben als bei L_1.
Von einer Beobachtung zur nächsten vergrößert sich das Feuer um mindestens 1 Feld, da es sonst gelöscht ist. Vergrößert es sich jedoch um genau 1 Feld, so kann es bei der nächsten Beobachtung gelöscht werden. Damit bei L_2 insgesamt weniger Felder abbrennen, muss sich das Feuer bei L_2 durchschnittlich weniger ausgebreitet haben als bei den Löschungen von L_1.

Weiterhin kann sich das Feuer nicht beliebig ausbreiten, zum Einen, da der Wald endlich ist, zum Anderen da das gesamte Feuer gelöscht wird.

Daraus ergibt sich trivialerweise, dass auch dieser Fall nicht eintreten kann. Somit ist der Algorithmus korrekt.

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 *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 $F^\star$ 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:

```
Wald    ; //Der Wald; ein 2D-Container

AnfangsBrennendeFelder()        { //Ermittelt die von Anfang brennenden Felder
  brennendeFelder := null; //1D-Container für Positionen brennender Felder
  for (i = 0..Wald.Höhe())
    for (j = 0..Wald.Breite())
      if (Wald[i,j] == BRENNEND)
        brennendeFelder.Add((i;j)); //Gefundene Position hinzufügen

  return brennendeFelder; //Alle gefundenen Positionen zurückgeben
}

NächsteBeobachtung(aktBrennendeFelder) { //Ermittelt die bei der nächsten Beonachtung brennenden Felder, aus den Feldern, die aktuell brennen
  neuBrennendeFelder := null;
  for all((x;y) from aktBrennendeFelder)
    if(Wald[x,y] == GELÖSCHT)
      continue; //Feld kann kein Feuer verteilen

    Wald[x,y] := VERKOHLT; //2 mal brennende Felder sind verkohlt
    for all((x';y') from Umgebung((x;y)))
      if(Wald[x',y'] == BRENNBAR)
        neuBrennendeFelder.Add((x';y')); //Gefundene Position hinzufügen
        Wald[x',y'] := BRENNEND; //Wald beginnt zu brennen

  return neuBrennendeFelder;
}

GetOptBewässerungspunkt(aktBrennendeFelder) { //Ermittelt den besten Bewässerungspunkt
  kleinsterAbstand := null; //Speichert für alle Felder des Waldes den kleinsten Abstand zu jedem Feld aus aktBrennendeFelder

  for(i = 0..kleinsterAbstand.Size())
```

```
        Fülle kleinsterAbstand[i] mithilfe einer Breitensuche

  anzEindeutigKleinstAbstände := null;

  for (i = 0..Wald.Höhe())
    for (j = 0..Wald.Breite())
      if(Es ex. k mit kleinsterAbstand[k][i,j] eindeutiges
         Minimum für alle mögliche k)
        anzEindeutigKleinstAbstände[k]++;

  return aktBrennendeFelder[k, sodass
     anzEindeutigKleinstAbstände[k] maximal];
}

SimuliereFeuer() { //Die eigentliche Berechnung
  aktBrennendeFelder := AnfangsBrennendeFelder(); //Anfangs
     interessante Felder; Kann brennende, von Feuer umschlossene
     Felder beinhalten
  while(!aktBrennendeFelder.Empty()) //Solange es brennende
     Felder gibt
    aktBrennendeFelder := NächsteBeobachtung(
       aktBrennendeFelder) //Ermittle die bei nächster
       Beobachtung brennenden Felder
      if(aktBrennendeFelder.Empty())
        break;          //Keine Felder brennen mehr

  Wald[GetOptBewässerungspunkt(aktBrennendeFelder)] := GELÖSCHT;
     //Lösche das aktuell beste Feld
}
```

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.

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

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

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

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

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

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

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

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

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

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

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

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

Eine bessere Löschung:

```
FO FO FO FO FO FO FO FO FO FO
FO FO FO FO FO FO FO FO FO FO
FO FO FO FO FO FO FO FO FO FO
FO FO FO WA FO WA FO FO FO FO
FO FO FO WA BU WA FO FO FO FO
FO FO WA FO FO FO WA FO FO FO
FO FO WA FO FO FO WA FO FO FO
FO FO WA FO FO FO WA FO FO FO
FO FO FO WA WA WA FO FO FO FO
FO FO FO FO FO FO FO FO FO FO

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

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

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

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

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

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

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

Der Algorithmus ist also nicht optimal, es handelt sich um eine Heuristik. Dabei liefert sie bei vielen Eingaben *ziemlich* gute Ergebnisse[3].

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

[3] Siehe dazu Sektion Beispiele

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 geeigntete Akzessormethoden können diese Attribute vollständig gekapselt werden.

Der eigentliche Algorithmus findet sich in der Datei `Buschfeuer.cpp`; die Ein- und Ausgabe steht in der Datei `Buschfeuer.h`
Das Lesen der Eingabe übernimmt die Prozedur `parseInput`, welche die Daten in eine globale Instanz der Klasse `Woods Forest` einliest.
Ist die Eingabe gelesen, werden aus dieser die zu Beginn brennenden Felder mithilfe der Funktion `getInitialBurningFields` ermittelt und dann gleich an die Prozedur `simulateFire` weitergereicht. Diese Prozedur `simulateFire` simuliert nun das Feuer und ermittelt die zu löschenden Felder unter Zuhilfenahme der Funktion `getOptimalWaterSpot`. Dabei wird nach jedem Löschvorgang eine Ausgabe getätigt, welche die zu löschende Position (oben links mit (0—0) beginnend) ausgibt. Auch wird unter Verwendung von ASCII-Escape-Sequenzen ein Bild in der Konsole angezeigt, welches den Wald darstellt.
Ist das Feuer gelöscht (kann es sich also nicht weiter ausbreiten), wird dem NUtzer eine Meldung ausgegeben, wie viele Felder verbrannten und wie viele Felder verbrannt und gelöscht wurden. (Diese beiden Zahlen beschreiben disjunkte Mengen.) Auch hier wird wieder ein Bild erzeugt und ausgegeben.

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.

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

1.2.1 Eingabeformat

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

Arg. 1 Pfad zu einer Daei 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:

- `*.tex` Produziert TeX-Grafiken
- `*.tex2` Produziert eine TeX-Datei, jedoch ohne Grafiken, dabei werden im Terminal keine ASCII-Sequenz-Bilder angezeigt
- `*.raw` Spiegelt die Ausgabe im Terminal in eine Datei, dabei werden im Terminal keine ASCII-Sequenz-Bilder angezeigt
- `jede andere` Spiegelt die Ausgabe im Terminal in die Datei

Dabei folgt eine mögliche Eingabe immer dem Folgenden Muster:

```
#Spalten #Zeilen
#Zeilen à #Spalten Zeichen,
 die das Feld beschreiben
```

Dabei wird der Anfangszustand einer Zelle wie folgt beschrieben:

```
LEER 0
BRENNBAR 1
BRENNEND 2
GELÖSCHT 4
```

Dabei kann ein Feld auch mehrere Startzustände haben, dann werden einfach die betreffenden Zustände durch ein bitweises Oder verknüpft. Beispieleingaben finden sich auch im Abschnitt *Beispiele*.

1.3 Beispiele

Beispiel 0

Die ist das Beispiel aus der Aufgabenstellung. Umgewandelt für mein Programm sieht diese Eingabe folgendermaßen aus[5]:

```
10 10
1101111101
1001111110
1111111111
1100010001
1111131111
1100111111
```

[5] Diese Eingabe finden Sie auch in der Datei `0.in`

```
1111011011
0111011010
1011011011
1111111111
```

Die Heuristik produziert folgende Ausgabe[6][7]:

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	01	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)`

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)`

[6]Diese 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.

[7]Um die ASCII-Escape-Sequenzen in TEX korrekt darzustellen, habe ich spezielle Ausgabemethoden geschrieben. Diese produzieren anstatt der ASCII-Sequenzen TEX-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	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 Beobachtung[8]:

```
10 11
3101111101
1001111110
1111111111
1100010001
1111131111
1100111111
1111011011
0111011010
1011011011
1111113111
1111111111
```

Die Heuristik produziert folgende Ausgabe[9]:

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

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

CO	BU	WA	FO	FO	FO	FO	FO	WA	FO
BU	WA	WA	FO	FO	FO	FO	FO	FO	WA
FO	FO	FO	FO	FO	FO	FO	FO	FO	FO
FO	FO	WA	WA	WA	01	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)
```

[8] Diese Eingabe finden Sie auch in der Datei `1.in`

[9] Diese 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 exisitert ebenfalls.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Die State-Space-Search findet eine bessere Lösung, benötig dafür aber schon einige Sekunden:

```
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]:

```
13 13
1111111111111
1000001000001
1011111111101
1011111111101
1011111111101
1011111111101
1111113111111
1011111111101
1011111111101
1011111111101
1011111111101
1000001000001
1111111111111
```

Mein Programm produziert folgende Ausgabe[11]:

[10] Diese Eingabe finden Sie auch in der Datei `2.in`

[11] Diese 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 exisitert 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	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	01	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	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	01	BU	FO	FO	FO	WA	FO
FO	FO	FO	FO	02	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	WA	WA	WA	WA	WA	FO	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	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

```
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 (¡ 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	WA	WA	WA	WA	WA	FO	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	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	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

```
Explanation:
WA --- EMPTY
FO --- BURNABLE
BU --- BURNED
```

```
 CO  --- COAL (doubly burned)
 ##  --- WATERED at time ##
Fields can have more than 1 state.
```

Beispiel 3

Ein (etwas) größeres Beispiel.[12]:

```
100 100
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
```

[12] Diese Eingabe finden Sie auch in der Datei `3.in`

```
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111131111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111
```

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Mein Programm produziert folgende Ausgabe[13] 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)
```

[13] Diese 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]:

[14] Diese Eingabe finden Sie auch in der Datei `7.in`

```
5 5
11111
11111
11311
11111
11111
```

Mein Programm produziert folgende Ausgabe[15]:

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	01	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	02	01	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
03	02	01	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	04	CO	BU
03	02	01	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
```

[15] Diese 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]:

```
10 10
1111111111
1111111111
1111111111
1110101111
1110301111
1101110111
1101110111
1101110111
1110001111
1111111111
```

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

Mein Programm produziert folgende Ausgabe[17]:

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)`

[17] Diese 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

```cpp
#include <vector>
#include <string>

typedef unsigned short u16;

#define FIELDSTATE      u16
#define EMPTY           0
#define BURNABLE        1
#define BURNED          2
#define WATERED         4
#define COAL            8

class Woods{
  private:
  int Width, Height;
  std::vector<std::vector<FIELDSTATE> > Fields;

public:
  Woods(int width, int height);

  int width() const;
  int height() const;

  FIELDSTATE& operator() (int x, int y);
  FIELDSTATE operator() (int x,  int y) const;

  bool operator==(const Woods& o) const;
```

```
  bool operator<(const Woods& o) const;

  std::string string() const;
  int cnt() const;
  int cnt2() const;
};
```

Woods.cpp

```
#include <vector>
#include <cstdio>

#include "Woods.h"

FIELDSTATE ERROR = -1;

Woods::Woods(int width, int height) : Width(width),
    Height(height) {
  Fields.assign(height, std::vector<FIELDSTATE>(width, 0));
}

int Woods::width() const { return this->Width; }
int Woods::height() const { return this->Height; }

FIELDSTATE& Woods::operator() (int x, int y) {
  if (x < 0 || y < 0 || x >= width() || y >= height()){
    printf("OUT OF BOUNDS 1");
    return ERROR;
  }
  return this->Fields[y][x];
}
FIELDSTATE Woods::operator() (int x,  int y) const {
  if (x < 0 || y < 0 || x >= width() || y >= height()){
    printf("OUT OF BOUNDS 2");
    return ERROR;
  }
  return this->Fields[y][x];
}

bool Woods::operator==(const Woods& o) const{
  if(o.width() != width() || o.height() != height())
    return false;

  for(int i = 0; i< width(); ++i)
    for(int j = 0; j < height(); ++j)
      if(o(i,j) != Fields[i][j])
        return false;
  return true;

  return (string() == o.string());
}

bool Woods::operator <(const Woods& o) const {
```

```
  if(std::min(o.width() , o.height()) < std::min(width(), height()))
    return true;
  if(std::max(o.width() , o.height()) < std::max(width(), height()))
    return true;
  if(std::min(o.width() , o.height()) > std::min(width(), height()))
    return false;
  if(std::max(o.width() , o.height()) > std::max(width(), height()))
    return false;

  for(int i = 0; i< width(); ++i)
    for(int j = 0; j < height(); ++j)
      if(o(i,j) <= Fields[i][j])
        return false;
  return true;
}

extern int dir[4][2];
std::string Woods::string() const {
  std::string ret;

  for(int i = 0; i< width(); ++i)
    for(int j = 0; j < height(); ++j){
      u16 andMask = (BURNABLE | BURNED | COAL);

      for(int k = 0; k < 4; ++k){
        int nx = i + dir[i][0], ny = j + dir[i][1];

        if(nx < 0 || ny < 0 || nx >= width() || ny >= height())
          continue;
        if(!(Fields[nx][ny] & (COAL | BURNED | WATERED))){
          andMask |= (WATERED);//Only watered fields at the border of the fire are of interest
          break;
        }
      }

      ret += (char) (((Fields[i][j] & andMask)) + 'A');
    }

  return ret;
}

int Woods::cnt() const { //counts the burningfields
  int ret = 0;
  for(int i = 0; i< width(); ++i)
    for(int j = 0; j < height(); ++j)
      if(Fields[i][j] & BURNED && !(Fields[i][j] & COAL))
        ret++;
  return ret;
```

```
}

int Woods::cnt2() const { //counts the burningfields
  int ret = 0;
  for(int i = 0; i< width(); ++i)
    for(int j = 0; j < height(); ++j)
      if(Fields[i][j] & BURNED)
        ret++;
  return ret;
}
```

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

```
#include <cstdio>
#include <vector>

#include "Woods.h"

typedef std::pair<int,int> PII;

const int oo = (1 << 29);                                   //
   The infinity
Woods Forest(0, 0);

struct Point {
public:
  int x, y;
  Point(int _x,int _y) : x(_x), y(_y) { }
  };
int dir[4][2] = {{1,0},{0,1},{-1,0},{0,-1}};

std::FILE* OUT;
   // The file to mirror the output to
void (*printSolution)(std::FILE*, bool);

//BEGIN OF INPUT
void parseInput(std::FILE* f) {
  int acFieldWidth, acFieldHeight;
  std::fscanf(f, "%i %i\n",&acFieldWidth, &acFieldHeight);

  Forest = Woods(acFieldWidth, acFieldHeight);

  for(int i = 0; i < acFieldHeight; ++i){
    for(int j= 0; j < acFieldWidth; ++j){
      char c;
      std::fscanf(f, "%c",&c);
      c -= '0';
      Forest(j, i) = (FIELDSTATE) c;
    }
    if(i < acFieldHeight-1)
      std::fscanf(f, "\n");
  }
}
```

```
//END OF INPUT
//BEGIN OF OUTPUT
void printSolution_TEX(std::FILE* f, bool finalOut) {
  std::fprintf(f, "\\\\\n");

  std::fprintf(f, "\\begin{tikzpicture}\n");
  std::fprintf(f, "\\tikzset{square matrix/.style={\n");
  std::fprintf(f, "matrix of nodes,\n");
  std::fprintf(f, "column sep=-\\pgflinewidth, row
     sep=-\\pgflinewidth,\n");
  std::fprintf(f, "nodes={draw,\n");
  std::fprintf(f, "minimum height=#1,\n");
  std::fprintf(f, "anchor=center,\n");
  std::fprintf(f, "text width=#1,\n");
  std::fprintf(f, "align=center,\n");
  std::fprintf(f, "inner sep=0pt\n");
  std::fprintf(f, "},\n");
  std::fprintf(f, "},\n");
  std::fprintf(f, "square matrix/.default=1.2cm\n");
  std::fprintf(f, "}\n");

  std::fprintf(f, "\\matrix[square matrix=1.4em] {\n");
  for(int j= 0; j < Forest.height(); ++j) {
    for(int i= 0; i < Forest.width(); ++i) {
      if(i)
        std::fprintf(f," &");

        FIELDSTATE acField = Forest(i, j);
        if(acField == EMPTY)
          std::fprintf(f, "|[fill=white]|");
        else if(acField & WATERED)
          std::fprintf(f, "|[fill=cyan]|");
        else if(acField & BURNABLE)
          std::fprintf(f, "|[fill=green]|");

        if(acField & COAL)
          std::fprintf(f, "\\color[rgb]{0,0,0}");
        else if(acField & BURNED)
          std::fprintf(f, "\\color[rgb]{1,0,0}");
        else if(acField == EMPTY)
          std::fprintf(f, "\\color[gray]{0.5}");
        else if(acField & BURNABLE)
          std::fprintf(f, "\\color[gray]{0.75}");

        if(acField & WATERED)
          std::fprintf(f, "\\textbf{\\texttt{%02d}}",acField >>
             4);
        else if(acField & COAL)
          std::fprintf(f, "\\textbf{\\texttt{CO}}");
        else if(acField & BURNED)
          std::fprintf(f, "\\textbf{\\texttt{BU}}");
        else if(acField == EMPTY)
          std::fprintf(f, "\\texttt{WA}");
```

```
        else if(acField & BURNABLE)
          std::fprintf(f, "\\texttt{FO}");
        else
          std::fprintf(f, "\\phantom{AA}");
      std::fprintf(f, "%%\n");
    }

    std::fprintf(f, "\\\\\n");
  }

  std::fprintf(f, "};\n\\end{tikzpicture}\\\\\n");

  if(finalOut){
    std::fprintf(f, "\\\\\nExplanation:");
    std::fprintf(f, "\\\\\n\\colorbox{white}{\\color[gray]{0.5}WA}  --- EMPTY");
    std::fprintf(f, "\\\\\n\\colorbox{green}{\\color[gray]{0.5}FO}  --- BURNABLE");
    std::fprintf(f, "\\\\\n\\colorbox{white}{\\color[rgb]{1,0,0}\\textbf{BU}} ---  BURNED");
    std::fprintf(f, "\\\\\n\\colorbox{white}{\\color[rgb]{0,0,0}\\textbf{CO}} ---  COAL (doubly burned)");
    std::fprintf(f, "\\\\\n\\colorbox{cyan}{\\#\\#}  --- WATERED at time \\#\\#");
    std::fprintf(f, "\\\\\nFields can have more than 1 state.");
  }
}

void printSolution_TERMINAL(std::FILE* f, bool finalOut) {
  std::fprintf(f, "\n");
  //The ASCII-magic starts here:
  for(int j= 0; j < Forest.height(); ++j) {
    for(int i= 0; i < Forest.width(); ++i) {
      FIELDSTATE acField = Forest(i, j);
      int waterval = 0;

      std::fprintf(f, "\x1b[s  ");
      if (acField == EMPTY)
        std::fprintf(f, "\x1b[u\x1b[37;47mWA");
      if (acField & BURNABLE)
        std::fprintf(f, "\x1b[u\x1b[32;42mFO");
      if (acField & BURNED)
        std::fprintf(f, "\x1b[u\x1b[1;5;31m/\\");
      if (acField & COAL)
        std::fprintf(f, "\x1b[u\x1b[1;4;5;30m/\\");
      if (acField & WATERED)
        std::fprintf(f, "\x1b[u\x1b[46m%02d", acField >> 4);
      std::fprintf(f, "\x1b[0;39;49m");
    }
```

```
    std::fprintf(f, "\n");
  }

  if (finalOut) {// An Explanation shall be printed
    std::fprintf(f, "\nExplanation:");
    std::fprintf(f, "\n\x1b[37;47mWA\x1b[39;49m  ---  EMPTY");
    std::fprintf(f, "\n\x1b[32;42mFO\x1b[39;49m  ---  BURNABLE");
    std::fprintf(f, "\n\x1b[1;5;31m/\\\x1b[0;39m  ---  BURNED");
    std::fprintf(f, "\n\x1b[1;4;5;30m/\\\x1b[0;39m  ---  COAL (doubly burned)");
    std::fprintf(f, "\n\x1b[46m##\x1b[0;39m  ---  WATERED at time ##");
    std::fprintf(f, "\nFields can have more than 1 state.");
  }
  std::fprintf(f, "\n");
}

void dontPrintSolution(std::FILE* f,bool finalOut) { return; }
//END OF OUTPUT
```

Buschfeuer.cpp Dies ist die Implementierung der Heuristik.

```
#include <cstdio>
#include <vector>
#include <queue>
#include <set>
#include <string>
#include <cstring>
#include <algorithm>

#include "Buschfeuer.h"

Point getOptimalWaterSpot(std::vector<Point>& candidates){
  std::queue<std::pair<PII,Point> > q;                                  // ((distance | color) | Location)
  for(int i= 0; i < candidates.size(); ++i)
    q.push(std::pair<PII,Point>(PII(0,i),candidates[i])); // insert all the candidates as start points for the BFS

  std::vector<std::vector<std::set<int> > > visited(Forest.width(),       // remember all nearest points first
    std::vector<std::set<int> >(Forest.height()));
  std::vector<std::vector<int> > shortDis(Forest.width(),                 // shortest distant to any burning field
    std::vector<int>(Forest.height(),oo));

  //BFS to calculate shortest paths
  while(!q.empty()){
    std::pair<PII,Point> ac = q.front();
    Point acPoint = ac.second;
```

```
    int acDistance = ac.first.first;
    int acColor = ac.first.second;

    q.pop();
    if(visited[acPoint.x][acPoint.y].count(acColor))
      continue;
    visited[acPoint.x][acPoint.y].insert(acColor);
    shortDis[acPoint.x][acPoint.y] = std::min(acDistance,shortDis[acPoint.x][acPoint.y]);

    for(int i= 0; i < 4; ++i){
      int newx = acPoint.x + dir[i][0];
      int newy = acPoint.y + dir[i][1];                    // calculate new field's indexes

      if(newx < 0 || newy < 0 || newy >= Forest.height() || newx >= Forest.width())
        continue;                                          // new field is outside the woods
      if (Forest(newx, newy) != BURNABLE)
        continue;                                          // Field is not of interest

      if(visited[newx][newy].count(acColor) == 0)          // Don't compute things twice
        if(acDistance + 1 <= shortDis[newx][newy]){
          shortDis[newx][newy] = acDistance + 1;
          q.push(std::pair<PII,Point>(PII(acDistance + 1,acColor),Point(newx,newy)));
        }
    }
  }

  //determine the field to be watered
  std::vector<PII> waterval(candidates.size(),PII(0,0));
  std::vector<std::vector<int> > farthDist(candidates.size(), std::vector<int>(2*(Forest.width()+Forest.height()),0));

  //Count the number of fields that have an unique fire spot a.k.a. waterval
  //Reckon the farthest field
  for(int i= 0; i< Forest.width(); ++i)
    for(int j= 0; j < Forest.height(); ++j)
      if(visited[i][j].size() == 1){
        waterval[*visited[i][j].begin()].first++;
        farthDist[*visited[i][j].begin()][shortDis[i][j]]++;
      }
  for(int i = 0; i < waterval.size(); ++i)
    waterval[i].second = i;

  std::sort(waterval.begin(),waterval.end(),std::greater<PII>());

  return candidates[waterval[0].second];
```

```
}

std::vector<Point>& getInitialBurningFields() {
  static std::vector<Point> burnedFields;

  for(int i = 0; i < Forest.height(); ++i)
    for(int j= 0; j < Forest.width(); ++j)
      if(Forest(j, i) & BURNED){
        burnedFields.push_back(Point(j, i));
//      printf("Initially burning: (%i|%i)\n",j, i);
      }
  return burnedFields;
}

void simulateFire(const std::vector<Point>& initiallyBurningFields) {
  std::vector<Point> burnedFields = initiallyBurningFields;
  if(printSolution != dontPrintSolution)
    printSolution_TERMINAL(stdout, false);
  if (OUT != 0)
    printSolution(OUT, false);

  int time = 0;
  while(!burnedFields.empty()) {                             // Simulate as long as there's still fire in the world
    std::vector<Point> newBurnedFields;                      // The burning fields at the next point of time

    //Calculate the new burning fields
    for(size_t i = 0; i < burnedFields.size(); ++i){
      int acx = burnedFields[i].x;
      int acy = burnedFields[i].y;

      if(Forest(acx, acy) & WATERED)
        continue;                                            // The field got watered and does not spread fire
      Forest(acx, acy) |= COAL;                              // Field burned down to coal...

      for(int j = 0; j < 4; ++j) {
        int newx = acx + dir[j][0];
        int newy = acy + dir[j][1];

        if(newx < 0 || newy < 0 || newy >= Forest.height() || newx >= Forest.width())
          continue;                                          // new field is outside the woods
        if(Forest(newx, newy) == BURNABLE){
          Forest(newx, newy) |= BURNED;                      // Field starts burning
          newBurnedFields.push_back(Point(newx,newy));

```

```
//        printf("  From now on burning: (%i|%i)\n",newx,newy); // log the happenings
        }
      }
    }
    if(newBurnedFields.empty())                            // Nothing to water, all plants happy...
      break;

    burnedFields = newBurnedFields;

    Point toWater = getOptimalWaterSpot(newBurnedFields); // Determine the field to water
    Forest(toWater.x, toWater.y) |= (WATERED | (time << 4));           // ... and water it


    //Output / mirror the partial solution

    std::printf("---\nAt time %i: Water spot (%i|%i)\n",++time,toWater.x,toWater.y);
    if(printSolution != dontPrintSolution)
      printSolution_TERMINAL(stdout, false);

    if (OUT) {
      std::fprintf(OUT, "---\nAt time %i: Water spot (%i|%i)\n",time,toWater.x,toWater.y);
      printSolution(OUT, false);
    }

  }
//    printf("Fire died.\n");

  //Count the total number of burned or coaled
  int wcnt = 0, ccnt = 0;
  for(int i= 0; i < Forest.width(); ++i)
    for(int j= 0; j < Forest.height(); ++j)
      if(Forest(i, j) & WATERED)
        wcnt++;
      else if(Forest(i, j) & COAL)
        ccnt++;

  //Output / Mirror the solution
  std::printf("---\nAnd you'll find %i pieces of coal and %i pieces of watered coal\n",ccnt,wcnt);
  if(printSolution != dontPrintSolution)
    printSolution_TERMINAL(stdout, true);
  if (OUT) {
    std::fprintf(OUT, "---\nAnd you'll find %i pieces of coal and %i pieces of watered coal\n",ccnt,wcnt);
    printSolution(OUT, true);
  }
}
```

```

int main(int argc, char** argv){
  if (argc > 1) {
    std::freopen(argv[1],"r",stdin);
    std::printf("Using %s as input.\n", argv[1]);
  }
  if (argc > 2){
    printf("Mirroring output to %s.\n", argv[2]);
    if (strstr(argv[2], ".tex2")) {
      std::printf("I reckon you want me to produce some
         graphicless TeX stuff...\n");
      printSolution = dontPrintSolution;
    }
    else if (strstr(argv[2], ".tex")) {
      std::printf("I reckon you want me to produce some TeX
         stuff...\n");
      printSolution = printSolution_TEX;
    }
    else if (strstr(argv[2], ".raw")) {
      std::printf("I reckon you want me to surpress
         graphics...\n");
      printSolution = dontPrintSolution;
    }
    else
      printSolution = printSolution_TERMINAL;
    OUT = std::fopen(argv[2],"w");
  }
  else{
    OUT = 0;
    printSolution = printSolution_TERMINAL;
  }

  parseInput(stdin);
  simulateFire(getInitialBurningFields());
}
```

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

```
#include <cstdio>
#include <vector>
#include <queue>
#include <set>
#include <string>
#include <cstring>
#include <algorithm>
#include <map>

#include "Buschfeuer.h"

typedef std::tuple<int,int,Woods> sssType; //State-Space-Search:
   distance from source; #of skipped waterings, acForest
typedef std::pair<int,int> pii;

```

```
bool operator <(const sssType &a, const sssType &b){

  int aBurning = std::get<2>(a).cnt();
  int bBurning = std::get<2>(b).cnt();

  int aBurned = std::get<2>(a).cnt2();
  int bBurned = std::get<2>(b).cnt2();

  auto aSk = std::get<1>(a);
  auto bSk = std::get<1>(b);

  return (aBurned > bBurned || (aBurned == bBurned && (aBurning
     - aSk > bBurning - bSk)));
}

std::pair<Woods,std::vector<Point>> getNextState(const Woods& w){
     //Calculate the next state's fire paired with the positions
     //of the new fire spots
  Woods ret(w.width(),w.height());
  std::vector<Point> pnts;

  for(int i = 0; i < w.width(); ++i)
    for(int j = 0; j < w.height(); ++j){
      ret(i,j) = w(i,j);
      if(w(i,j) == BURNABLE){
        bool startsBurning = false;
        for(int k = 0; k < 4; ++k){
          int x = i + dir[k][0];
          int y = j + dir[k][1];
          if(x < 0 || y < 0 || x >= w.width() || y >= w.height())
            continue;
          if(w(x,y) & BURNED && !(w(x,y) & WATERED))
            startsBurning = true;
        }
        if(startsBurning){
          ret(i,j) |= BURNED;
          pnts.push_back(Point(i,j));
        }
      }
      else if(w(i,j) & BURNED && !(w(i,j) & WATERED))
        ret(i,j) |= COAL;
    }
  return std::pair<Woods,std::vector<Point>>(ret, pnts);
}

int main(int argc, char ** argv){
  if (argc > 1) {
    std::freopen(argv[1],"r",stdin);
    std::printf("Using %s as input.\n", argv[1]);
  }
  if (argc > 2){
    std::printf("Mirroring output to %s.\n", argv[2]);
    if (strstr(argv[2], ".tex2")) {
```

```
      std::printf("I reckon you want me to produce some graphicless TeX stuff...\n");
      printSolution = dontPrintSolution;
    }
    else if (strstr(argv[2], ".tex")) {
      std::printf("I reckon you want me to produce some TeX stuff...\n");
      printSolution = printSolution_TEX;
    }
    else if (strstr(argv[2], ".raw")) {
      std::printf("I reckon you want me to surpress graphics...\n");
      printSolution = dontPrintSolution;
    }
    else
      printSolution = printSolution_TERMINAL;
    OUT = std::fopen(argv[2],"w");
  }
  else{
    OUT = 0;
    printSolution = printSolution_TERMINAL;
  }

  parseInput(stdin);

  std::priority_queue<sssType> q; //The SSS-Queue
  q.push(sssType(1,0,Forest)); //Initial Node, let time start at 1

  while(!q.empty()) {
    sssType ac = q.top(); q.pop();

    auto acForest = std::get<2>(ac);
    int acDis = std::get<0>(ac);
    int acSkipped = std::get<1>(ac);

    auto next = getNextState(acForest);


    if(acForest.cnt() <= acSkipped)
    { //Fire can be dead by this time
      //Reconstruct and output Solution

      Forest = acForest;

      std::set<int> remTimes; //check at which points in time a watering was required
                              //so that those fields that are still burning can get one of the
                              //remaining times to be watered

      for(int i = 1; i<= acDis; ++i)
        remTimes.insert(i);
```

```


      for(int i = 0; i< Forest.width(); ++i)
        for(int j = 0; j < Forest.height(); ++j)
          if((Forest(i,j) & BURNED) && ( Forest(i,j) & WATERED ) )
            remTimes.erase(remTimes.find(Forest(i,j) >> 4));

      int ccnt = 0;  //No of coaled tiles

      for(int i = 0; i< Forest.width(); ++i)
        for(int j = 0; j < Forest.height(); ++j)
          if((Forest(i,j) & BURNED) && !( Forest(i,j) & (WATERED | COAL )) ){
            Forest(i,j) |= (WATERED | ((*remTimes.begin()) << 4));
            remTimes.erase(remTimes.begin());
          }
          else if ((Forest(i,j) & COAL) && !( Forest(i,j) & WATERED) )
            ccnt++;

      std::printf("Water always optimally (to save water)...\nand you'll find %i pieces of coal and %i pieces of watered coal\n",ccnt,acDis-acSkipped -1);

      if(printSolution != dontPrintSolution)
        printSolution_TERMINAL(stdout, true);
      if (OUT) {
        std::fprintf(OUT, "Water always optimally (to save water)...\nAnd you'll find %i pieces of coal and %i pieces of watered coal\n",ccnt,acDis- acSkipped-1);
        printSolution(OUT, true);
      }
      continue;
//        break;
    }

    q.push(sssType(acDis + 1,acSkipped + 1, next.first));
    for(auto i : next.second){
      next.first(i.x,i.y) |= (WATERED | (acDis << 4)); // Save time whe field got watered
      q.push(sssType(acDis+ 1,acSkipped, next.first));
      next.first(i.x,i.y) &= ~(WATERED | (acDis << 4));
    }
  }
  printf("Finished\n");
}
```

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. Lebenszeit

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. Lebensgraph

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 Teilgraph von G ausgegeben werden, für welchen allein es kein solches f geben kann.

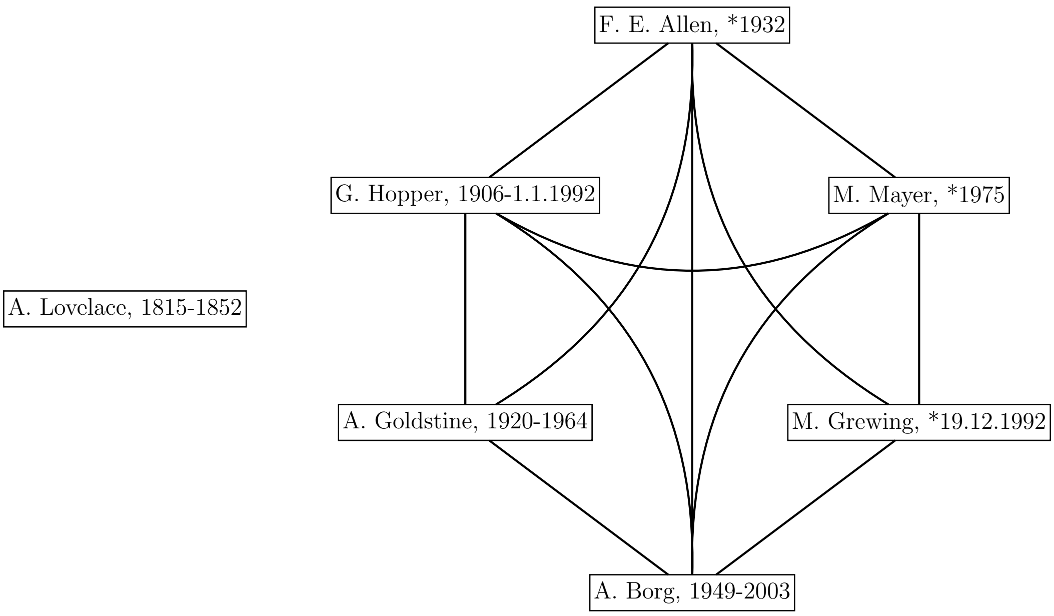

Abbildung 1: Der Lebensgraph aus der Aufgabenstellung

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 $\mathbb{R}$ (Es sei der Einfachheit der Schreibweise wegen angenommen, dass eine solche Potenzmenge existiert.)

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

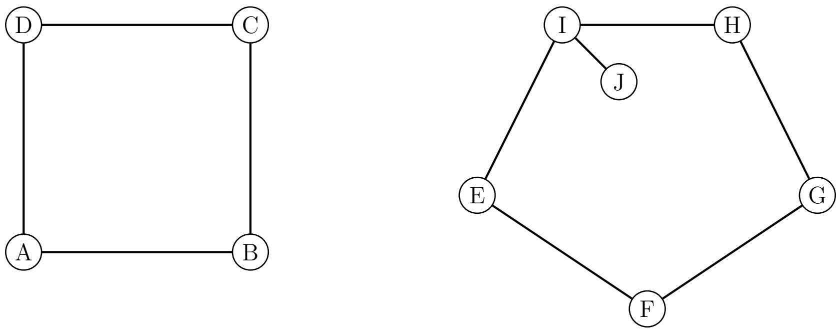

Abbildung 2: Graphen mit Löchern können niemals Lebensgraph sein *(im 2. Graphen ist J **nicht** Teil des Loches)*

Der Grund hierfür ist offensichtlich: Sei $Z = (v_i, v_k, ..., v_i)$ 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 Konten 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

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

[20] Der 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:

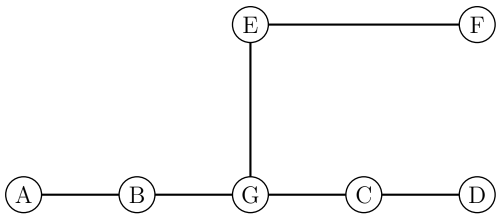

Abbildung 3: Chordalgraph, der **kein** Lebensgraph ist

Es gilt nun also ein *hinreichendes Kriterium* dafür zu finden, ob ein Graph G ein Lebensgraph ist.

Seien dazu die *maximalen Cliquen* in G betrachtet. Eine Clique C eines Graphen $G = (V, E)$ heißt dabei maximal, wenn es keinen Knoten $v \in V \setminus C$ gibt, sodass $C \cup \{v\}$ eine Clique in G bildet; wenn also das Hinzufügen eines beliebigen weiteren Knotens des Graphen zu der Clique C bewirkt, dass C keine Clique mehr ist.

maximalen Cliquen

Es kann gezeigt werden, dass Chordalgraphen $G_C = (V, E)$ genau diejenigen Graphen sind, bei denen sich eben diese maximalen Cliquen so in einem *Cliquenbaum* T anordnen lassen, so dass für jeden Knoten $vSub$ aus V gilt, dass die Cliquen, in denen v enthalten ist einen zusammenhängenden Teilbaum von T bilden.

Cliquenbaum

Speziell bei Lebensgraphen vereinfacht sich dieser Baum jedoch zu einem Pfad, man kann die maximalen Cliquen also so anordnen, dass alle Cliquen, die einen Knoten v enthalten in dieser Anordnung aufeinanderfolgend sind. Diese Anordnung sein im Folgenden mit *Cliquenkette* beschrieben. Aus einer Cliquenkette lassen sich nun leicht Intervalle für Knoten ablesen, und umgekehrt, da zwei Konten nur in der selben Clique sind, wenn sie durch eine Kante verbunden sind, und ihnen so zurecht ein gemeinsamer Intervallabschnitt zugeordnet worden ist.

Cliquenkette

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 ein Cliquenbaum erzeugt, überprüft, ob dieser eine Cliquenkette ist und zuletzt wird überprüft, ob jeder Knoten nur in aufeinanderfolgenden Cliquen vorkommt. Die hierzu notwendigen Algorithmen[21] werden nun im Folgenden vorgestellt.

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

```
//Lex-BFS
//Eingabe: Graph G = (V,E), Knoten seien durchnummeriert 0..|V|-1
//Ausgabe: Reihenfolge der Knoten

```

[21] nach Habib, M., McConnell, R., Paul, C. und Viennot , L.: *"Lex-BFS and partition reÿnement, with applications to transitive orientation, interval graph recognition and consecutive ones testing"*, erschienen in `Theoretical Computer Science 234 (2000)`, Seiten 59–84

```
begin
  int[] ausgabe := int[|V|];

  Liste<int> L  := V; //Initiale Anordnung der Knoten (L[i] = i)

  Liste<int>[] S := {L};//Klassen

  int cnt = |V| - 1;  //Zähler für Ausgabe
  while S != { } do begin
    int x := letztes Element der letzten Klasse in S;
    entferne x aus der letzten Klasse in S,
       wird diese Klasse dadurch leer, entferne diese aus S;

    ausgabe[x] := cnt; cnt := cnt - 1;

    //Klassen werden in 2 Teilklassen aufgespalten:
    //diejenigen Knoten, die Nachbar von x sind,
    //und die, die es nicht sind

    foreach Liste<int> i in S do begin
      nachbarn := { Knoten in i, die benachbart zu x };
      nicht_nachbarn := i \ nachbarn;

      //Ordne Nachbarn vor Nicht-Nachbarn in S
      ersetzte { i } durch { nachbarn , nicht_nachbarn } in S;
            //Ignoriere leere Mengen
    end;
  end;
  return ausgabe;
end.
```

perfekte Eliminationsordnung

Eine *perfekte Eliminationsordnung eines Graphen* $G = (V, E)$ heißt eine Anordnung A der Knoten V, sodass für jeden Konten $v \in V$ gilt:
v und die Nachbarn von v, die nach v in A auftreten, bilden eine *Clique* in G.

Ein Satz über Chordalgraphen besagt, dass ein Graph G genau dann ein Chordalgraph ist, wenn G eine perfekte Eliminationsordnung besitzt.
Auch kann bewiesen werden, dass die Lex-BFS bei einem Chordalgraphen G eine perfekte Eliminationsordnung von G erzeugt.
Speziell für die Überprüfung von Graphen auf Chordalität muss also nur noch die von der Lex-BFS erzeugte Abfolge PI der Knoten darauf hin überprüft werden, ob diese eine perfekte Eliminationsordnung ist.

Dies kann beispielsweise mit folgendem Algorithmus geschehen, dabei seien mit $RN(v)$ die in der Eliminationsornung rechts gelegenen Nachbarknoten von v bezeichnet und mit $P(v)$ der in der Eliminationsornung am weitesten links liegende Knoten von $RN(v)$.

```
//Überprüfung einer Ordnung der Knoten V darauf, ob diese
//eine perfekte Eliminationsordnung ist
//Eingabe: Graph G = (V,E), Reihenfolge PI der Knoten V
//Ausgabe: true, falls PI perfekte Eliminationsordnung, false sonst
begin
  Ermittle RN(v) und P(v) für jeden Knoten v;

  foreach v in V do begin
```

```
    if ( RN(v) \ P(v) ist keine
      Teilmenge von RN(P(v)))
        then return false;
  end;
  return true;
end.
```

Obiger Algorithmus nutzt aus, dass bei einer perfekten Eliminationsordnung für jeden Knoten v gilt, dass $v \cup RN(v)$ eine Clique bildet, somit muss auch $RN(v)$ eine Clique bilden.
Deshalb muss $RN(v) \setminus P(v) \subseteq RN(P(v))$ für jeden Knoten v gelten.
Sollte es sich bei PI nicht um eine perfekte Eliminationsordnung, so gibt es ein v, für dass $v \cup RN(v)$ keine Clique ist. Dies heißt aber im speziellen, dass $RN(v)$ keine Clique bilden und somit $P(v)$ nicht mit allen Knoten aus $RN(v) \setminus P(v)$ durch eine Kante verbunden ist oder $RN(v) \setminus P(v)$ selbst keine Clique bildet. Der zweite Fall kann dann rekursiv weiter behandelt werden, bis einmal Fall 1 auftritt, dieser muss auftreten, da der betrachtete Graph endlich ist.
Speziell bei Fall 1 enthält dann $RN(P(v))$ jedoch mindestens einen Knoten nicht, der in $RN(v) \setminus P(v)$ enthalten ist.
Somit ist obiger Algorithmus also korrekt.

Ist der eingegebene Graph nun ein Chordalgraph, so wird nun der Cliquenbaum erzeugt. Dazu kann man, das schon für den vorangegangenen Algorithmus für jeden Knoten v definierte, $P(v)$ nutzen. Es ist leicht zu sehen, dass dieses $P(v)$ schon einen Baum impliziert. Es werden nun also einfach die maximalen Cliquen ermittelt und dann entsprechend dieses Baumes geordnet:

```
//Ermittlung des Cliquenbaumes
//Eingabe: Graph G = (V,E), Reihenfolge PI der Knoten V
//Ausgabe: Ein Cliquenbaum B
begin
  Ermittle RN(v) und P(v) für jeden Knoten v;

  Sei T der durch P(v) implizierte Baum;
  Sei r die Wurzel von T;

  Sei Clique ein Array, das Knoten eine Clique zuordnet;

  foreach v in T, v != r in preorder do begin
    if( RN(v) \ {P(v)} != RN(P(v)) ) then begin
      Sei c := {v} eine neue Clique;
      Clique[v] := c;
      PAR(c) := Clique[P(v)];
    end else begin
      Clique[P(v)] += { v };
      Clique[v] := Clique[P(v)];
    end;
  end;
  Sei B der durch PAR(C) implizierte Baum;
  return B;
end.
```

Anschließend wird nun versucht, die maximalen Cliquen zu ordnen, sodass eine Cliquenkette entsteht. Der naive Ansatz, alle möglichen Anordnungen durchzuprobieren ist allerdings zu langsam.
Eine solche Anordnung kann auch mit folgendem, der Lex-BFS sehr ähnlichen, Algorithmus von M. Habib, R. McConnell, C. Paul und L. Viennot geschehen, der diesen Test bei richtiger Implementierung in Linearzeit durchführen kann:

```
//Lebensgraphen-Test
//Ermittlung einer Cliquenkette
//Eingabe: Graph G = (V,E), Reihenfolge PI der Knoten V
//Ausgabe: Eine Cliquenkette L
begin
  Sei B=(X,F) ein Cliquenbaum der mit dem vorangegangenen
     Algorithmus gefunden wurde;
  Sei X die Menge der maximalen Cliquen, X = {C1, C2, ... Cn};
  Sei L eine Liste von Mengen, L := ( X );
  Sei PIVOTS ein leerer Stack;
  Sei USED ein Array;

  while L enthält eine Menge Xc mit |Xc| > 1 do begin
    Sei b eine Menge;
    if PIVOTS == { } then begin
      Sei Cl die Clique in Xc mit der grö ß ten Nummer;
      Ersetze Xc durch Xc\{Cl}, {Cl} in L;
      b := {Cl};
    end else begin
      while USED[PIVOTS.top()] == TRUE do
        PIVOTS.pop();
      x := PIVOTS.top();
      USED[x] := TRUE;
      b := { W aus X | x in W };
      Seien Xa und Xb die erste bzw letzte Menge in L,
         die eine Klasse enthält die auch in b vorhanden ist;
      Ersetze Xa durch Xa\b, (Xa geschnitten b)
        und Xb durch (Xb geschnitten b), Xb\b;
    end;
    foreach (Ci,Cj) in F mit Ci in b und Cj nicht in b do begin
      PIVOTS += (Ci geschnitten Cj);
      entferne (Ci,Cj) aus F;
    end;
  end;

  foreach v in V do begin
    if Cliquen, in denen v vorkommt
       sind nicht aufeinanderfolgend in L then
       return "G ist kein Lebensgraph";
  end;
  //L ist nun die Cliquenkette
  return "G ist ein Lebensgraph";
end.
```

Ist diese Anordnung möglich, so handelt es sich um einen Lebensgraphen, andernfalls nicht. Zuallerletzt sollten nun noch die eigentlichen Intervalle für die Knoten ausgegeben werden. Dabei ist es nicht von Bedeutung, ob und wie nun Daten als Begrenzung für die Intervalle angegeben werden, oder ob einfach Zahlen ausgegeben werden. Dabei ist das eigentliche Finden der Intervalle aus einer Cliquenkette trivial realisierbar, da eine Cliquenkette schon Intervalle impliziert.

2.1.3 Erkennung des kleinsten Teilgraphen, der kein Lebensgraph ist

Zunächst ist es einfach zu erkennen, dass es nach obigem Algorithmus 2 verschiedene Arten von Graphen gibt, die keine Lebensgraphen sind. Es gibt diejenigen Graphen, die keine Chordalgraphen sind und diejenigen Graphen, die zwar Chordalgraphen sind, die jedoch trotzdem keine Lebensgraphen sind.

Weiterhin ist es leicht zu sehen, dass jeder Graph mit 3 oder weniger Knoten ein Lebensgraph ist.

Ein naheliegender Ansatz zur Findung des kleinsten Teilgraphen ist es daher alle möglichen Teilgraphen der Größe nach zu überprüfen, ob diese kein Lebensgraph sind. Dabei können schon von vorne herein gewisse Teilmengen ausgeschlossen werden. So ist es nicht sinnvoll, Cliquen zu überprüfen, auch sollte der Teilgraph zusammenhängend sein.

Laufzeitanalyse Bei geschickter Implementierung kann erreicht werden, dass jeder der obigen Algorithmen eine Laufzeit von $\mathcal{O}(|V|+|E|)$ besitzt. Diese Schranke erfüllt auch die geforderte Effizienz des Verfahrens.

Das Ausprobieren aller Teilmengen einer Menge V benötigt $\mathcal{O}(2^{|V|})$, diese Schranke ergibt sich also speziell auch für die naheliegende Variante des Findens des minimalen Teilgraphen, der kein Lebensgraph ist.

2.2 Umsetzung

Zur Implementierung der Lösungsidee habe ich die Sprache `C++` gewählt.

Dabei finden sich alle Methoden in der Datei `Lebenslinien.cpp`. Dabei ist der Graph global als Adjazenzmatrix `G` gespeichert.

Das Lesen der Eingabe übernimmt die Methode `readInput`. Ist die Eingabe gelesen, so wird eine Ordnung der Knoten mithilfe einer Lex-BFS ermittelt, diese ist in der Methode `LexBFSOrder` implementiert. Die Überprüfung dieser Ordnung darauf, eine perfekte Eliminationsordnung zu sein geschieht in der Methode `isChordal`. Sofern die Ordnung dieser Überprüfung standhält wird ein Cliquenbaum ermittelt, implementiert in der Methode `getCliqueTree`. Abschließend wird noch überprüft, ob sich aus dem Cliquen des Cliquenbaumes auch eine Cliquenkette machen lässt. Dies ist in der Methode `isIntervalGraph` wiederzufinden. Die Ermittlung und Ausgabe der eigentlichen Intervalle übernimmt die Methode `main`.
Dabei kann aufgrund von naiver Implementierung nur eine Schranke von $\mathcal{O}((|V|+|E|)^2)$ eingehalten werden.

Handelt es sich nicht um einen Lebensgraphen, so übernimmt die Methode `smallestFailingSubgraph` das Finden des minimalen Teilgraphen, der kein Lebensgraph ist. Auch in diesem Fall übernimmt die `main`-Methode die Ausgabe.

2.2.1 Eingabeformat

Die Eingabe in mein Programm kann über die Standardeingabe erfolgen. Dabei wird zunächst die Anzahl N an Knoten in dem Folgenden Graphen angegeben. Darauf folgen N^2 Zahlen, entweder 0 oder 1 und durch Leerzeichen getrennt; der Graph als Adjazenzmatrix. (Dabei steht eine 1 dafür, dass eine Kante zwischen den beiden entsprechenden Knoten existiert.)

2.3 Beispiele

Beispiel 0

Das Beispiel aus der Aufgabenstellung.[22]

```
7
0 1 1 0 1 1 0
1 0 1 1 1 1 0
1 1 0 1 1 0 0
0 1 1 0 1 0 0
1 1 1 1 0 1 0
1 1 0 0 1 0 0
0 0 0 0 0 0 0
```

Eine mögliche Belegung mit Intervallen[23]:

```
The graphsize.
7 lines with 7 space seperated integers;
the graph as adjancy matrix.
The graph is chordal!
In clique 3: 1 2 3 4
In clique 4: 0 1 2 4
In clique 5: 0 1 4 5
In clique 6: 6
The graph is an interval graph:
Node 0: 25.09.1904 - 16.10.1907
Node 1: 25.09.1900 - 16.10.1907
Node 2: 25.09.1900 - 16.10.1905
Node 3: 25.09.1900 - 16.10.1903
Node 4: 25.09.1900 - 16.10.1907
Node 5: 25.09.1906 - 16.10.1907
Node 6: 25.09.1908 - 16.10.1909
```

Mein Programm benötigt für die Berechnung dieses Beispiels weniger als eine Sekunde.

Eine Visualisierung:

Beispiel 1

Ein Graph der kein Chordalgraph ist.[24]

```
4
0 1 0 1
1 0 1 0
0 1 0 1
1 0 1 0
```

Die Ausgabe meines Programms[25]:

[22] Dieses Beispiel lässt sich auch in der Datei 0.in wiederfinden.
[23] Diese Ausgabe lässt sich auch in der Datei 0.out wiederfinden.
[24] Dieses Beispiel lässt sich auch in der Datei 1.in wiederfinden.
[25] Diese Ausgabe lässt sich auch in der Datei 1.out wiederfinden.

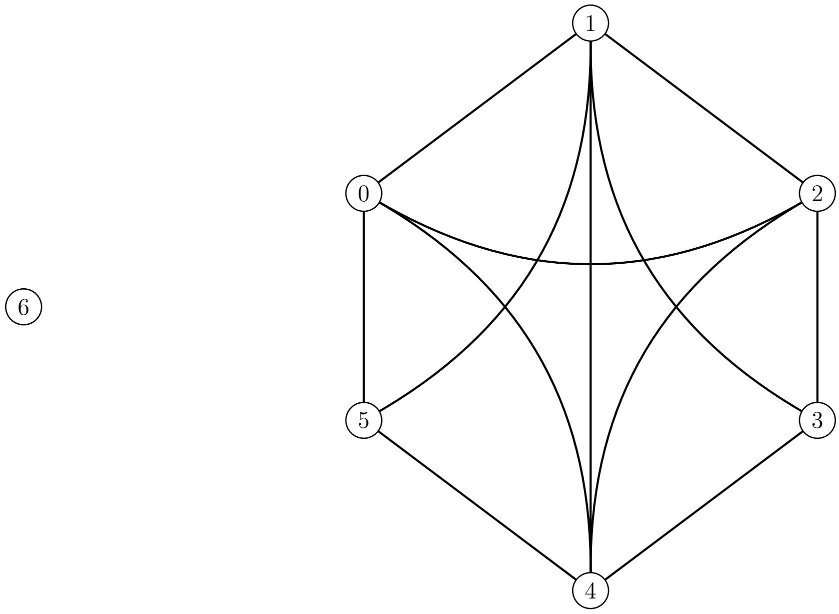

Abbildung 4: Der Graph aus Beispiel 0.

```
The graphsize.
4 lines with 4 space seperated integers;
the graph as adjancy matrix.
The graph is not chordal.
Smallest subgraph that is not chordal:
Node 0
Node 1
Node 2
Node 3
```

Mein Programm benötigt für die Berechnung dieses Beispiels weniger als eine Sekunde.

Eine Visualisierung:

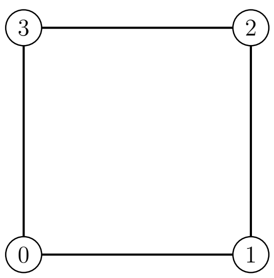

Abbildung 5: Der Graph aus Beispiel 1.

Beispiel 2

Ein weiterer Graph, der nicht chordal ist[26]

```
6
0 1 0 0 1 0
```

[26] Dieses Beispiel lässt sich auch in der Datei 2.in wiederfinden.

```
1 0 1 0 0 0
0 1 0 1 0 0
0 0 1 0 1 0
1 0 0 1 0 1
0 0 0 0 1 0
```

Die Ausgabe meines Programms[27]:

```
The graphsize.
6 lines with 6 space seperated integers;
the graph as adjancy matrix.
The graph is not chordal.
Smallest subgraph that is not chordal:
Node 0
Node 1
Node 2
Node 3
Node 4
```

Mein Programm benötigt für die Berechnung dieses Beispiels weniger als eine Sekunde.

Eine Visualisierung:

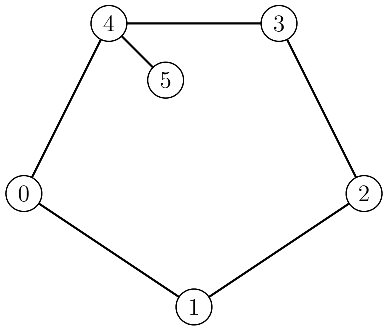

Abbildung 6: Der Graph aus Beispiel 2.

Beispiel 3

Ein Dreieck, das offenkundig ein Lebensgraph ist[28]

```
3
0 1 1
1 0 1
1 1 0
```

Eine gefundene Belegung mit Intervallen[29]:

```
The graphsize.
3 lines with 3 space seperated integers;
the graph as adjancy matrix.
The graph is chordal!
```

[27]Diese Ausgabe lässt sich auch in der Datei 2.out wiederfinden.
[28]Dieses Beispiel lässt sich auch in der Datei a.in wiederfinden.
[29]Diese Ausgabe lässt sich auch in der Datei a.out wiederfinden.

```
In clique 2: 0 1 2
The graph is an interval graph:
Node 0: 25.09.1900 - 16.10.1901
Node 1: 25.09.1900 - 16.10.1901
Node 2: 25.09.1900 - 16.10.1901
```

Mein Programm benötigt für die Berechnung dieses Beispiels weniger als eine Sekunde.

Eine Visualisierung:

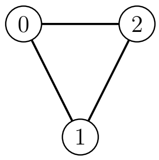

Abbildung 7: Der Graph aus Beispiel 3.

Beispiel 4

Ein weiteres Beisiel.[30]

```
7
0 1 0 0 0 1 0
1 0 1 0 0 0 0
0 1 0 1 1 1 1
0 0 1 0 0 0 0
0 0 1 0 0 0 0
1 0 1 0 0 0 0
0 0 1 0 0 0 0
```

Auch dieser Graph enthält ein Loch.[31]:

```
The graphsize.
7 lines with 7 space seperated integers;
the graph as adjancy matrix.
The graph is not chordal.
Smallest subgraph that is not chordal:
Node 0
Node 1
Node 2
Node 5
```

Mein Programm benötigt für die Berechnung dieses Beispiels weniger als eine Sekunde.

Beispiel 5

Ein weiteres Beisiel für einen Lebensgraphen.[32]

[30] Dieses Beispiel lässt sich auch in der Datei 6.in wiederfinden.
[31] Diese Ausgabe lässt sich auch in der Datei 6.out wiederfinden.
[32] Dieses Beispiel lässt sich auch in der Datei 7.in wiederfinden.

```
7
0 1 1 0 0 0 0
1 0 1 0 0 0 0
1 1 0 1 0 0 0
0 0 1 0 1 0 0
0 0 0 1 0 1 1
0 0 0 0 1 0 1
0 0 0 0 1 1 0
```

Die gefundene Belegung mit den Intervallen[33]:

```
The graphsize.
7 lines with 7 space seperated integers;
the graph as adjancy matrix.
The graph is chordal!
In clique 2: 0 1 2
In clique 3: 2 3
In clique 4: 3 4
In clique 6: 4 5 6
The graph is an interval graph:
Node 0: 25.09.1900 - 16.10.1903
Node 1: 25.09.1900 - 16.10.1903
Node 2: 25.09.1900 - 16.10.1905
Node 3: 25.09.1904 - 16.10.1907
Node 4: 25.09.1906 - 16.10.1909
Node 5: 25.09.1908 - 16.10.1909
Node 6: 25.09.1908 - 16.10.1909
```

Mein Programm benötigt für die Berechnung dieses Beispiels weniger als eine Sekunde.

Beispiel 6

Ein weiteres Beisiel für einen Lebensgraphen.[34]

```
4
0 1 1 1
1 0 0 1
1 0 0 1
1 1 1 0
```

Die gefundene Belegung mit den Intervallen[35]:

```
The graphsize.
4 lines with 4 space seperated integers;
the graph as adjancy matrix.
The graph is chordal!
In clique 2: 0 2 3
In clique 3: 0 1 3
The graph is an interval graph:
Node 0: 25.09.1900 - 16.10.1903
Node 1: 25.09.1902 - 16.10.1903
Node 2: 25.09.1900 - 16.10.1901
Node 3: 25.09.1900 - 16.10.1903
```

[33] Diese Ausgabe lässt sich auch in der Datei 7.out wiederfinden.
[34] Dieses Beispiel lässt sich auch in der Datei 3.in wiederfinden.
[35] Diese Ausgabe lässt sich auch in der Datei 3.out wiederfinden.

Mein Programm benötigt für die Berechnung dieses Beispiels weniger als eine Sekunde.

2.4 Quelltext

```
#include <cstdio>
#include <vector>
#include <list>
#include <stack>
#include <deque>
#include <algorithm>
#include <iterator>
#include <set>
#include <queue>

std::vector<std::vector<int> > G; //The Graph
int N; //Graphsize

void readInput(){
  std::printf("The graphsize.");
  scanf("%i",&N);
  G.assign(N,std::vector<int>(N,0));

  std::printf("%d lines with %d space seperated integers;\nthe
     graph as adjancy matrix.\n",N,N);
  //Read Graph stored in an Adjancy Matrix
  for(int i = 0; i < N; ++i)
    for(int j= 0; j < N; ++j)
      scanf("%i",&G[i][j]);
}


//BEGIN lexicographic BFS

std::vector<int> LexBFSOrder(){
  std::list<int> L;
  for(int i = 0; i < (int)G.size(); ++i)
    L.push_back(i);
  std::deque<std::list<int>> classes;
  classes.push_back(L);
  std::deque<int> ret;

  while(!classes.empty()){
    int ac = *(classes.begin()->begin());
    classes.begin()->pop_front();
    if(classes.begin()->empty())
      classes.pop_front();

    ret.push_front(ac);

    std::deque<std::list<int>> new_classes;
    for(auto i : classes){
      std::list<int> tmp_in, tmp_out;

```

```
      for(auto j : i)
        if(G[j][ac]) //partion depending on whether neighbor of
            ac or not
          tmp_in.push_back(j);
        else
          tmp_out.push_back(j);

      if(!tmp_in.empty())
        new_classes.push_back(tmp_in);
      if(!tmp_out.empty())
        new_classes.push_back(tmp_out);
    }
    classes = new_classes;
  }
  return {ret.begin(), ret.end()};
}

//END

//BEGIN Chordality test

 //checks if ordering is a perfect elimination ordering
bool isChordal(std::vector<int> ordering){

  std::vector<std::vector<int> > rightNeighs(N, std::vector<int>
     ()); //Neghbours to the right

  std::vector<bool> used(N, false);
  for(int x = 0; x < N; ++x){
    int acnode = ordering[x];

    used[acnode] = true;

    for(int i = 0; i < N; ++i)
      if(G[i][acnode] && used[i])
        rightNeighs[i].push_back(acnode);
  }

  for(int x = 0; x < N; ++x){
    if(rightNeighs[x].size() <= 1)
      continue;

   // naive implementation
    for(int i = 1; i < rightNeighs[x].size(); ++i){
      bool existant = false;
      for(auto j : rightNeighs[rightNeighs[x][0]])
        if(j == rightNeighs[x][i]){
          existant = true;
          break;
        }
      if(!existant)
        return false;
    }
```

```
  }

  return true;
}

std::vector<std::pair<std::vector<int>, int > >
getCliqueTree(std::vector<int> ordering, bool printCliques =
    true) {
  std::vector<std::vector<int> > rightNeighs(N, std::vector<int>
      ()); //Neghbours to the right
  std::vector<std::vector<int> > T(N,std::vector<int>());

  std::vector<bool> used(N, false);
  for(int x = 0; x < N; ++x){
    int acnode = ordering[x];

    used[acnode] = true;
    for(int i = 0; i < N; ++i)
      if(G[i][acnode] && used[i]){
        if(rightNeighs[i].empty())
          T[acnode].push_back(i);
        rightNeighs[i].push_back(acnode);
      }
  }

  std::stack<int> s;
  std::vector<std::set<int> > clique(N,std::set<int>());
  std::set<int> cliques;

  for(int i = 0; i< N; ++i)
    if(rightNeighs[i].empty()){
      s.push(i); //push the tree roots
      if(T[i].empty()) { //Is simple node
        clique[i].insert(i);
        cliques.insert(i);
      }
    }
  while(!s.empty()){
    auto ac = s.top(); s.pop();

    for(auto child : T[ac]) {
      for(auto i : rightNeighs[child])
        clique[child].insert(i);
      clique[child].insert(child);

      auto acParent = rightNeighs[child][0];

      if(std::includes(clique[child].begin(),clique[child].end(),
          clique[acParent].begin(), clique[acParent].end()))
        if(cliques.count(acParent))
          clique[acParent].erase(clique[acParent].find(acParent));

      cliques.insert(child);
```

```
      if(!T[child].empty())
        s.push(child);
    }
  }

  std::vector<std::pair<std::vector<int>, int > > t;
  std::vector<int> num(N, -1);
  int cnt = 0;
  for(int i = 0; i < N; ++i)
    if(cliques.count(i) && clique[i].count(i))
      num[i] = cnt++;
  for(int i = 0; i < N; ++i)
    if(cliques.count(i) && clique[i].count(i)){

      if(printCliques){
        std::printf("In clique %d:", i);
        for(auto k : clique[i])
          std::printf(" %d", k);
        std::printf("\n");
      }
      int par = (rightNeighs[i].empty() ? -1 :
          num[rightNeighs[i][0]]);
      t.push_back(std::pair<std::vector<int>, int
          >({clique[i].begin(),clique[i].end()}, par));
    }
  return t;
}

//END

//BEGIN Interval Graph Reckon'ing

std::deque<std::vector<int>> L; //clique chain

bool isIntervalGraph(std::vector<int> ordering, bool printOut =
    true){
  auto T = getCliqueTree(ordering, printOut);

  std::list<std::pair<std::vector<int>,int> > l;
  std::list<std::pair<int,int> > treeEdges;
  for(size_t o = 0; o < T.size(); ++o){
    auto i = T[o];
    l.push_back(std::pair<std::vector<int>,int>(i.first,o));
    if(i.second >= 0)
      treeEdges.push_back(std::pair<int,int>(o,i.second));
  }
  std::list<std::list<std::pair<std::vector<int>, int> > >
      classes;
  classes.push_back(l);

  std::vector<int> pivots;

```

```
std::vector<bool> used(N, false);

L.clear();

while(!classes.empty()){
  std::set<int> C;
  while(!pivots.empty() && used[pivots.back()])
    pivots.pop_back();

  if(pivots.empty()){

    std::list<std::pair<std::vector<int>, int> >::iterator Xc
       = classes.rbegin()->begin();

    for(auto I = classes.rbegin()->begin(); I !=
       classes.rbegin()->end(); ++I)
      if(I->second > Xc->second)
        (Xc) = (I);

    C.insert(Xc->second);
    L.push_front(Xc->first);

    classes.rbegin()->erase(Xc);

    if(classes.rbegin()->empty())
      classes.pop_back();
  }
  else {
    int x = pivots.back(); pivots.pop_back();
    used[x] = true;

    auto Xa = classes.begin(), Xb = classes.begin();
    bool Xaset = false;

    for(auto i = classes.begin(); i != classes.end(); ++i)
      for(auto j : *i) {
        bool contains = false;
        for(auto k: j.first)
          if(k == x){
            contains = true;
            break;
          }
        if(contains) {
          C.insert(j.second);

          Xb = i;
          if(!Xaset){
            Xa = i;
            Xaset = true;
          }
        }
      }

```

```
        //Partion Xa und Xb
        std::list<std::pair<std::vector<int>, int>> tmp_inA,
           tmp_outA, tmp_inB, tmp_outB;

        for(auto i : (*Xa))
          if(C.count(i.second))
            tmp_inA.push_back(i);
          else
            tmp_outA.push_back(i);
        for(auto i : (*Xb))
          if(C.count(i.second))
            tmp_inB.push_back(i);
          else
            tmp_outB.push_back(i);

        if(tmp_outA.empty())
          std::swap(tmp_inA,tmp_outA);

        (*Xa) = tmp_outA;
        if(!tmp_inA.empty())
          classes.insert(++Xa, tmp_inA);

        if(Xa != ++Xb) {
          if(tmp_inB.empty())
            std::swap(tmp_inB,tmp_outB);
          (*Xb) = tmp_inB;
          if(!tmp_outB.empty())
            classes.insert(Xb, tmp_outB);
        }
      }

      //Update Pivots
      for(auto i = treeEdges.begin(); i != treeEdges.end(); ++i){
        if(C.count(i->first) != C.count(i->second)){
          std::set<int> Ci;
          for(auto j : T[i->first].first)
            Ci.insert(j);
          for(auto j : T[i->second].first)
            if(Ci.count(j))
              pivots.push_back(j);

          treeEdges.erase(i);
          --i;
        }
      }
    }

    std::vector<bool> alive(N,false), ended (N,false);

    for(auto i : L){
      std::vector<bool> seen (N, false);
      for(auto j : i)
        seen[j] = true;
```

```

    for(int j = 0; j < N; ++j)
      if(ended[j] && seen[j])
        return false;
      else if(seen[j])
        alive[j] = true;
      else if(!seen[j] && alive[j]){
        ended[j] = true;
        alive[j] = false;
      }
  }
  return true;
}

//END

//determines the number of connected components
std::vector<bool> visited;
void dfs(int u){
  visited[u] = true;
  for(int i = 0; i < N; ++i)
    if(G[i][u] && !visited[i])
      dfs(i);
}

//determines smallest subgraph that is not an interval graph
std::vector<int> smallestFailingSubgraph(bool chordal){
  auto G2 = G;
  int N2 = N;
  for(N = 4; N <= N2; ++N){ //try all size i subsets of the nodes
    std::vector<bool> btm(N2,false);
    for(int i = N2-1; i >= N2 - N; --i)
      btm[i] = true;
    do {
      G.assign(N,std::vector<int>(N,0));
      int i = 0, j = 0;

      for(int k = 0; k < N2; ++k)
        if(btm[k]){
          j = 0;
          for(int l = 0; l < N2; ++l)
            if(btm[l])
              G[i][j++] = G2[k][l];
          ++i;
        }

      visited.assign(N, false);
      dfs(0);
      bool fail = false;
      for(int p = 0; p< N; ++p)
        if(!visited[p]) {
          fail = true;
          break;
```

```
        }
      if(fail)
        continue;
      auto LO = LexBFSOrder();
      if(!isChordal(LO) || (!chordal && !isIntervalGraph(LO, false))) { //Found smallest subgraph, that is not chordal

        std::vector<int> ret;

        for(int k = 0; k < N2; ++k)
          if(btm[k])
            ret.push_back(k);
        return ret;
      }
    }while(std::next_permutation(btm.begin(),btm.end()));
  }

  return std::vector<int>();
}

int main(){
  readInput();
  auto LO = LexBFSOrder();
  if(isChordal(LO))
    std::printf("The graph is chordal!\n");
  else{
    std::printf("The graph is not chordal.\n"); //Calculation stops here

    std::printf("Smallest subgraph that is not chordal:\n");
    for(auto i : smallestFailingSubgraph(true))
      std::printf("Node %d\n",i);

    return 0;
  }

  if(isIntervalGraph(LO)){

    std::vector<int> start(N, -1), end(N, -1);
    for(int i = 0; i < L.size(); ++i){
      auto ac = L[i];
      for(auto j : ac){
        end[j] = i;
        if(start[j] == -1)
          start[j] = i;
      }
    }

    std::printf("The graph is an interval graph:\n");
    for(int i = 0; i < N; ++i)
      std::printf("Node %d: 25.09.%4d - 16.10.%4d\n", i, 1900 + 2 * start[i], 1901 + 2 * end[i]);
  }
```

```
  else {
    std::printf("The graph is not an interval graph.\n");

    std::printf("Smallest subgraph that is not an interval
       graph:\n");
    for(auto i : smallestFailingSubgraph(false))
      std::printf("Node %d\n",i);

    return 0;
  }
}
```