32. Bundeswettbewerb Informatik, 2. Runde

Lösungshinweise und Bewertungskriterien

Allgemeines

Es ist immer wieder bewundernswert, wie viele Ideen, wie viel Wissen, Fleiß und Durchhaltevermögen in den Einsendungen zur zweiten Runde eines Bundeswettbewerbs Informatik stecken. Um aber die Allerbesten für die Endrunde zu bestimmen, müssen wir die Arbeiten kritisch begutachten und hohe Anforderungen stellen. Von daher sind Punktabzüge die Regel und Bewertungen über die Erwartungen (5 Punkte) hinaus die Ausnahme. Lassen Sie sich davon nicht entmutigen! Wie auch immer Ihre Einsendung bewertet wurde: Allein durch die Arbeit an den Aufgaben und den Einsendungen hat jede Teilnehmerin und jeder Teilnehmer einiges gelernt; den Wert dieses Effektes sollten Sie nicht unterschätzen.

Bevor Sie sich in die Lösungshinweise vertiefen, lesen Sie doch bitte kurz die folgenden Anmerkungen zu Einsendungen und den beiliegenden Unterlagen durch.

Terminprobleme Einige Einsender gestehen ganz offen, dass Ihnen die Zeit zum Einsendeschluss hin knapp geworden ist, worauf wir bei der Bewertung leider keine Rücksicht nehmen können. Abiturienten macht der Terminkonflikt mit der Abiturvorbereitung Probleme. Der ist für eine erfolgreiche Teilnahme sicher nicht ideal. In der zweiten Jahreshälfte läuft aber die zweite Runde des Mathewettbewerbs, dem wir keine Konkurrenz machen wollen. Also bleibt uns nur die erste Jahreshälfte. Aber: Sie hatten etwa vier Monate Bearbeitungszeit für die zweite BwInf-Runde. Rechtzeitig mit der Bearbeitung der Aufgaben zu beginnen war der beste Weg, Konflikte mit dem Abitur zu vermeiden.

Dokumentation Es ist sehr gut nachvollziehbar, dass Sie Ihre Energie bevorzugt in die Lösung der Aufgaben, die Entwicklung Ihrer Ideen und die Umsetzung in Software fließen lassen. Doch ohne eine gute Beschreibung der Lösungsideen, eine übersichtliche Dokumentation der wichtigsten Komponenten Ihrer Programme, eine gute Kommentierung der Quellcodes und eine ausreichende Zahl sinnvoller Beispiele (die die verschiedenen bei der Lösung

des Problems zu berücksichtigenden Fälle abdecken) ist eine Einsendung wenig wert. Bewerterinnen und Bewerter können die Qualität Ihrer Einsendung nur anhand dieser Informationen vernünftig einschätzen. Mängel können nur selten durch gründliches Testen der eingesandten Programme ausgeglichen werden – wenn diese denn überhaupt ausgeführt werden können: Hier gibt es häufig Probleme, die meist vermieden werden könnten, wenn Lösungsprogramme vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner getestet würden. Insgesamt sollte die Erstellung der Dokumentation die Programmierarbeit begleiten oder ihr teilweise sogar vorangehen: Wer nicht in der Lage ist, Idee und Modell präzise zu formulieren, bekommt keine saubere Umsetzung in welche Programmiersprache auch immer hin.

Bewertungsbögen Auf Ihrem Bewertungsbogen sind die Bewertungskriterien als Stichpunkte in den Tabellenzeilen aufgeführt. Keine Markierung in einer Zeile bedeutet, dass das Kriterium den Erwartungen entsprechend erfüllt wurde. Vermerkt wird also in der Regel nur, wenn davon abgewichen wurde – nach oben oder nach unten. Ein Kreuz in der Spalte „+“ bedeutet Zusatzpunkte, ein Kreuz unter „–“ bedeutet Minuspunkte für Fehlendes oder Unzulängliches. Dabei haben die Marken nicht immer den gleichen Einfluss auf die Gesamtbewertung, sondern können je nach Einsendung und Kriterium unterschiedlich gewichtig sein. Die Schattierung eines Feldes bedeutet, dass die entsprechenden Zusatz- bzw. Minuspunkte in der Regel nicht vergeben wurden.

Bewertungskriterien Bei den folgenden Erläuterungen handelt es sich um Vorschläge, nicht um die einzigen Lösungswege, die wir gelten ließen. Wir akzeptieren in der Regel alle Ansätze, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind. Einige Dinge gibt es allerdings, die – unabhängig vom gewählten Lösungsweg – auf jeden Fall diskutiert werden müssen. Zu jeder Aufgabe gibt es deshalb einen Abschnitt, indem gesagt wird, worauf bei der Bewertung letztlich geachtet wurde, zusätzlich zu den grundlegenden Anforderungen an Dokumentation (insbesondere: klare Beschreibung der Lösungsidee, genügend aussagekräftige Beispiele, wesentliche Auszüge aus dem Quellcode enthalten), Quellcode (insbesondere: Strukturierung und Kommentierung, gute Übersicht durch Programm-Dokumentation) und Programm (keine Implementierungsfehler).

Danksagung An der Erstellung der Lösungsideen haben mitgewirkt: Martin Thoma (Aufgabe 1) und Philip Wellnitz (Aufgabe 2). Die Aufgaben wurden vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt, und zwar aus Vorschlägen von Rainer Gemulla (Aufgabe 1), Torben Hagerup (Aufgabe 2) und Jens Gallenbacher (Aufgabe 3).

Aufgabe 1: Buschfeuer

1.1 Grundlegendes

In dieser Aufgabe geht es darum, die Feuerwehr (also die Mountain Rangers) bei der Planung eines Löscheinsatzes zu unterstützen. Der Einsatz findet in einem Waldgebiet statt, das durch ein Raster modelliert werden kann.

Bei der Beschreibung der Lösungsideen werden wir mit folgenden Begriffen arbeiten:

Waldgebiet Das gesamte Gebiet, modelliert durch ein Raster.

Karte Raster-Plan des Waldgebiets, aufgeteilt in:

Felder Jedes Feld des Rasters modelliert ein *Waldstück* (Begriff aus der Aufgabenstellung). Es gibt drei Arten von Feldern auf der Karte: *Wald-Felder* (intakter Wald), *Feuer-Felder* (in Brand geratener Wald) und *Schneisen-Felder* (Teil einer Brandschneise). Jedes Feld kann außerdem *gelöscht* werden, wobei es sinnlos wäre, Schneisen-Felder zu „löschen". Gelöschte Felder wie auch die Schneisen-Felder können nicht brennen, also nicht (wieder) zu Feuer-Feldern werden.

Die Ausbreitung des Feuers und der Einsatz der Feuerwehr kann man sich als ein Spiel mit zwei Gegnern vorstellen, die abwechseln ziehen. Ist das Feuer am Zug, breitet es sich um ein Feld in alle Richtungen aus; ist die Feuerwehr am Zug, löscht sie ein Feld. Der Einsatz der Feuerwehr hat also folgenden „Spiel"-Ablauf:

1. Das Spielfeld wird initialisiert: Die Karte mit Wald, Brandschneisen und Feuer wird bekanntgegeben.
2. **Feuer-Zug**: Alle Wald-Felder, die links, rechts, oben oder unten an Feuer-Felder angrenzen, werden zu Feuer-Feldern.
3. Gibt es noch Wald-Felder, die an Feuer-Felder angrenzen? Falls nein, ist das Spiel zu Ende.
4. **Lösch-Zug**: Die Feuerwehr darf nun ein Feuer-Feld (oder ein Wald-Feld, siehe Zusatzfrage) löschen.
5. Gehe zu Schritt 2.

Natürlich will die Feuerwehr so viel Wald (genauer: so viele Wald-Felder) wie möglich retten. Aber das kann sie vielleicht auf unterschiedliche Art und Weise erreichen – und z. B. unterschiedlich viel Wasser verbrauchen. Deshalb wollen wir eine Spielsituation (und insbesondere die Situation am Ende des Spiels) primär danach bewerten, wie viele (intakte) Wald-Felder es noch gibt. Gibt es gleich viele Wald-Felder, wollen wir, dass möglichst wenige Felder gelöscht wurden.

Wir betrachten also für eine Spielsituation das Zahlenpaar

$$(\text{Anzahl Wald-Felder}, \text{Anzahl gelöschte Felder})$$

oder kurz: (W, G). Situation $A = (W_A, G_A)$ ist besser als Situation $B = (W_B, G_B)$, wenn:

$$(W_A > W_B) \text{ oder } (W_A = W_B \text{ und } G_A < G_B)$$

Die Situationen sind gleich gut, wenn:

$$W_A = W_B \text{ und } G_A = G_B$$

1.2 Beobachtungen

Wie immer, wenn man ein kompliziertes Problem bekommt, sollte man überlegen, ob die einfachsten Techniken zur Lösung des Problems funktionieren. Eine solche einfache Technik ist die *Brute-Force-Methode*, bei der man alle Kombinationen ausprobiert. Wie man sich schnell an dem 10×10 Beispiel klar macht, ist es hier nicht sinnvoll alle Möglichkeiten zu probieren. Es gibt in jedem Lösch-Zug $10 \cdot 10 = 100$ Felder, die gelöscht werden können – die wollen wir als *Kandidaten* bezeichnen. Da das Feuer sich in jedem Zug mindestens eine Zelle weiter bewegt, die Feuerwehr aber auch in jedem Zug eine Zelle löscht, könnte es bis zu $\frac{10 \cdot 10}{2} = 50$ Züge dauern, bis ein Szenario durchgespielt ist. Wenn man nur berücksichtigt, dass man nichts löscht, was schon mal gelöscht wurde, sind das immerhin $\frac{100!}{50!} = 100 \cdot 99 \cdot (\ldots) \cdot 50 \approx 3 \cdot 10^{93}$ Szenarien. Das ist zu viel.

Die Anzahl der Kandidaten muss also unbedingt verringert werden. Mit den folgenden Kriterien können Kandidaten ausgeschlossen werden, die zu löschen nicht lohnen würde. Offensichtlich gilt:

Trivial-Kriterium Gelöscht wird nur Feuer oder Wald. Also keine gelöschten Felder und auch keine Schneisen-Felder.

Feuerlinien-Kriterium Falls Wald-Felder gelöscht werden, sollten sie direkt an ein Feuer-Feld angrenzen oder durch Wald mit einem brennenden Feld verbunden sein. Wald-Felder, die durch Schneisen-Felder oder gelöschte Felder bereits vom Feuer abgeschnitten sind, können wir getrost ignorieren.

Inferno-Kriterium : Es lohnt sich nur ein Feld zu löschen, das entweder selbst ein Wald-Feld ist oder direkt an ein Wald-Feld angrenzt. Also sollte man insbesondere kein Feld löschen, das nur von Feuer umgeben ist.

Endzug-Kriterium Wenn man mit einem Lösch-Zug das Spiel beenden kann, sollte man das tun. Kann man also durch Löschen eines Feldes alle Wald-Felder vom Feuer abschneiden, wird dieses Feld im Lösch-Zug ausgewählt und alle anderen ignoriert.

Felder, die nach Anwendung all dieser Kriterien zum Löschen ausgewählt wurden, nennen wir auch *sinnvolle Kandidaten*.

Nun könnte man die Hoffnung haben, dass es sich nur lohnt Wald zu löschen, der direkt bedroht wird oder gerade erst verbrannt wurde. Ob dem so ist oder nicht, wird im nächsten Abschnitt erörtert.

Wald gießen

Bringt es etwas, Wald zu löschen, der noch nicht gebrannt hat? Kurzum: Ja, folgende Aussage ist wahr:

Es existieren Situationen, im dem ein Zug, der ein Waldstück löscht, das noch nicht brennt, am Ende mehr Wald rettet als jeder Zug, der ein brennendes Waldstück löscht.

Um das einzusehen, sollte man die Situation aus Abbildung 1 betrachten. Die roten (dunklen) Felder brennen, die grünen (hellen) Felder sind intakter Wald. Wenn man zu Beginn nahe der Mitte „löscht“ (also intakte Wald-Felder begießt), kann man einige Wald-Felder retten. Löscht man jedoch nur Feuer-Felder, kann man kein einziges Wald-Feld retten.

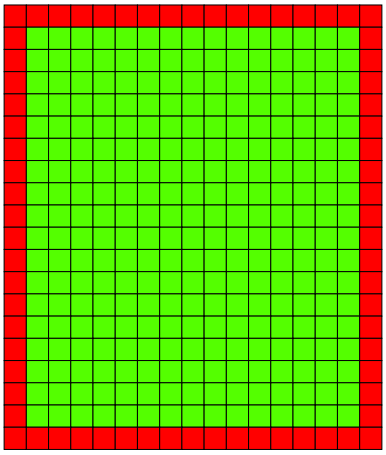


Abbildung 1: Hier macht es Sinn, zuerst intakten Wald zu löschen

Modellierung als Graph

Das Waldgebiet bzw. die Karte kann auch mithilfe eines ungerichteten Graphen modelliert werden. Dabei ist jedes Feld ein Knoten; Kanten werden zwischen benachbarten Feldern (bzw. den ihnen entsprechenden Knoten) gezogen. Die Knoten von Schneisen-Feldern und alle zugehörigen Kanten werden aus dem Graph entfernt. Auch das Löschen eines Feldes entspricht dem Entfernen seines Knotens und aller zugehörigen Kanten.

Feuer ist in einem solchen Graphen dann ein binäres Attribut jedes Knotens. Ein Knoten kann also mit Feuer markiert sein (Feuerknoten; entspricht einem Feuer-Feld) oder nicht (Waldknoten; entspricht einem Wald-Feld).

Wenn wir also zuvor ein $n \times m$-Raster hatten, haben wir nun einen Graphen $G = (V, E)$ mit $|V| \leq n \cdot m$ Knoten. Außerdem hat dieser Graph $|E| \leq 4nm$ Kanten.

Nun kann man, ausgehend von den Feuerknoten, mittels Tiefensuche alle verbundenen Knoten finden und somit das Feuerlinien-Kriterium überprüfen. Da die Tiefensuche eine Laufzeit von $\mathbf{O}(|V| + |E|)$ hat, ist die Laufzeit in diesem Graphen in $\mathbf{O}(nm + 4nm) = \mathbf{O}(nm)$, also linear in der Anzahl der Zellen.

Das Trivial-Kriterium und das Inferno-Kriterium können beide in konstanter Laufzeit, also in $\mathbf{O}(1)$, überprüft werden.

Das Endzug-Kriterium bedeutet im Graph, dass es nur einen Waldknoten gibt, der mit einem Feuerknoten direkt verbunden ist. Dies kann offensichtlich auch in $\mathbf{O}(nm)$ überprüft werden.

Wenn wir jedoch speichern, welche Knoten noch benachbarte Waldknoten haben, können wir den Feuer-Zug und die Überprüfung des Endzug-Kriteriums deutlich schneller durchführen. Diese Feuerknoten-Datenstruktur muss selbstverständlich nach jedem Feuer-Zug aktualisiert werden.

Zusammenfassen kann man sagen, dass sich die sinnvollen Löschkandidaten in $\mathbf{O}(nm)$ ermitteln lassen.

1.3 Lösungsideen

Branch-and-Bound

Den Spielablauf kann man als Baum darstellen. Jeder Knoten des Baumes stellt eine Situation dar. Die Wurzel ist die Startsituation. Da der Feuer-Zug durch die Spielregeln vorgegeben ist, werden mögliche Folgesituationen nur durch die Auswahl des zu löschenden Feldes bestimmt. Die Kinder eines Knotens sind also alle Situationen, die sich aus dem Löschen eines Feldes im Lösch-Zug und dem anschließenden Feuer-Zug ergeben. Mit jeder Löschentscheidung steigt man im Baum eine Ebene tiefer hinab. Jedes Blatt stellt eine Situation dar, in der sich das Feuer nicht weiter ausbreiten kann. Abbildung 2 zeigt den Anfang eines solchen Entscheidungsbaums für ein 3×3 Raster.

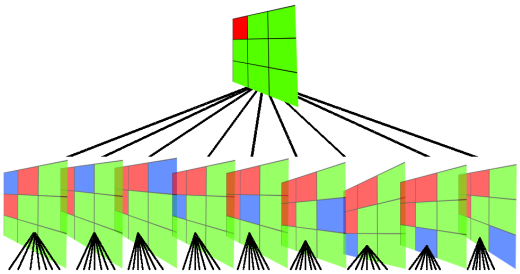


Abbildung 2: Entscheidungsbaum

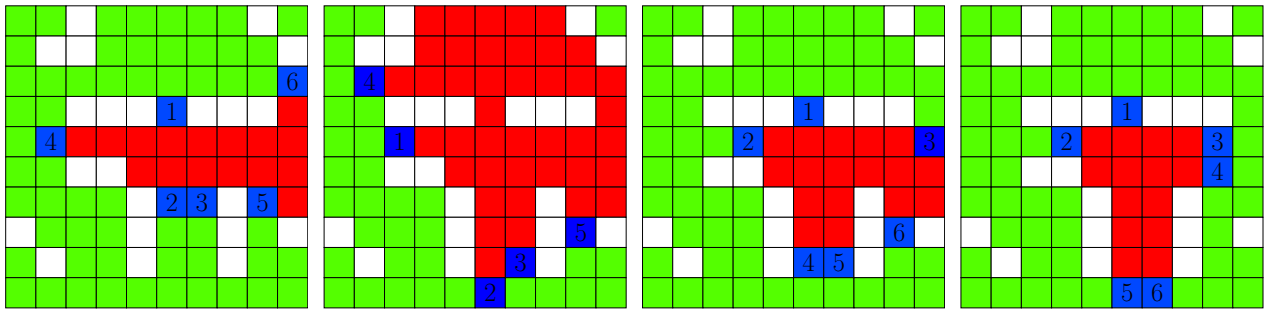
Dieser Entscheidungsbaum ist sehr regelmäßig: Jeder Elternknoten hat so viele Kindknoten, wie es Löschmöglichkeiten gibt. Also hat jeder Elternknoten bei einem $n \times m$-Feld genau $n \cdot m$ Kindknoten. Wie schon gesagt, lässt sich die Größe des Baumes durch Beschränkung auf sinnvolle Löschkandidaten verringern.

Jedoch haben wir bereits eingesehen, dass der komplette Entscheidungsbaum zu groß ist. Allerdings sind wir auch eigentlich nicht am Entscheidungsbaum, sondern an dem Pfad, also den optimalen Entscheidungen interessiert. Es liegt also nahe, Situationen, die auf keinen Fall besser werden können also bereits gefundene Lösungen, nicht weiter zu betrachten. Hierbei nutzt man aus, das die Bewertung einer Spielsituation immer schlechter wird: Solange das

Spiel nicht zu Ende ist, wird immer mindestens ein Wald-Feld zu einem Feuer-Feld werden. Die Spielsituation eines Elternknotens ist also immer echt besser als alle Situationen seiner Kindknoten.

Wenn wir also eine Situation vorliegen haben, die unabhängig von unseren folgenden Entscheidungen immer schlechter sein wird als eine bereits gefundene Lösung, so verwerfen wir diese Situation und betrachten insbesondere ihre Kindknoten nicht.

Umso schneller wir also gute Lösungen finden, desto weniger unnötige Situationen müssen wir betrachten. Eine gute Heuristik beschleunigt dieses Verfahren also sehr. Mögliche Heuristiken werden im Folgenden vorgestellt.

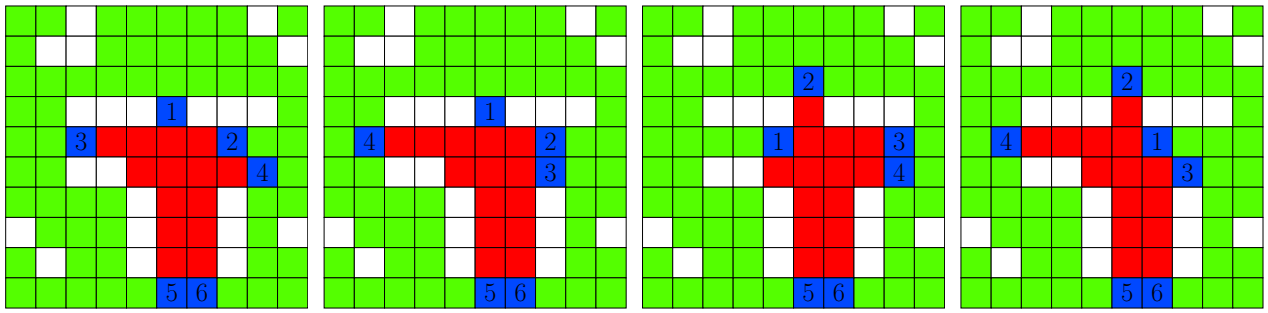


(a) Lösung vom Aufgabenblatt: (56, 6)

(b) Zufälliges Löschen mit Simulation: (32, 5)

(c) Gefahr reduzieren (55, 6)

(d) Branch-and-Bound 1 (58, 6)



(e) Branch-and-Bound 2 (58, 6)

(f) Branch-and-Bound 3 (58, 6)

(g) Branch-and-Bound 4 (58, 6)

(h) Branch-and-Bound 5 (58, 6)

Abbildung 3: Beispiele für verschiedene Endsituationen und ihre Bewertung

Abbildung 3 zeigt für das Beispiel des Aufgabenblattes Ergebnisse für die verschiedenen Heuristiken. Abbildung 3d bis Abbildung 3h zeigen alle fünf optimalen Ergebnisse.

Zufälliges Löschen mit Spielsimulation

Eine sehr einfache und auch schnell zu implementierende Heuristik, die mittelmäßige Bewertungen liefert, besteht im zufälligen Löschen. Dabei wird das Spiel mehrfach durchsimuliert. In der Simulation wird in jedem Lösch-Zug aus der Liste der sinnvollen Löschkandidaten zufällig einer gewählt. Man merkt sich die Ergebnisse der verschiedenen Simulationsläufe. Dann nimmt man die Pfade aus den Simulationen mit den besten Ergebnissen, löscht das erste Feld

so, wie es bei den meisten Pfaden mit bestem Ergebnis gemacht wurde, und simuliert von da aus den Rest des Spiels wiederum mehrfach mit zufälligen Losch-Zügen.

Abbildung 3b zeigt, wie sich der Algorithmus bei einer Ausführung für das Beispiel des Aufgabenblattes verhalten hat. Da dieser Algorithmus nicht deterministisch ist, sind hier jedoch viele Endsituationen möglich.

Akkumulierte Gefahr reduzieren

Eine erstaunlich gute Heuristik funktioniert wie folgt:

1. Lösche parallel alle Kandidaten.
2. Für jede der so entstehenden Situationen lässt man nun alles abbrennen und merkt sich für jedes Feld, nach wie vielen Schritten dies abgebrannt ist.
3. Für jeden Kandidaten erhält man so ein $m \times n$ Zahlenfeld aus natürlichen Zahlen. Die Stellen, die nahe am Feuer waren, haben niedrige Werte und die Stellen, die weit weg waren, haben hohe Werte. Die Felder, die bereits brannten, gelöscht wurden oder Schneisen haben, haben die 0 als Wert. Diese Zahlen werden nun aufsummiert und dem Löschkandidaten zugeordnet.
4. Wähle zum Löschen den Kandidaten mit der größten Summe.

Bei dieser Heuristik ist es essentiell zu überprüfen, ob man in einem Schritt die Simulation beenden könnte. Wenn man die Simulation beenden kann, wird ein ganzer Abschnitt einfach 0 und daher wird die Heuristik am Ende alles tun um das Feuer *nicht* zu löschen!

Die Intuition hinter dieser Heuristik ist, dass man bei besonders langen Löscheinsätzen das Feuer in eine Schneise zwängt.

Die Laufzeit dieser Lösung ist $\mathbf{O}(nm)$, da man den Vorgang für alle Kandidaten durchführen muss.

Abbildung 3c zeigt die Lösung für die Situation aus dem Aufgabenblatt.

Greedy-Strategie

Eine weitere Standard-Technik besteht darin, in jedem Schritt, also sozusagen Zeit-lokal, den "besten" Löschkandidaten zu nutzen. Algorithmen, die zu einem festen Zeitpunkt den Folgezustand wählen, der das beste Ergebnis verspricht (bzw. der gemäß unserer Bewertung besten Spielsituation entspricht), nennt man *gierige* oder *Greedy-Algorithmen*.

In unserem Fall heißt das, bevor wir uns also zum Löschen eines Feldes entscheiden, schauen wir uns an wie viel Wald das Feuer im folgenden Schritt verbrennen würde. Dann nehmen wir eine der Löschkandidaten, bei denen eine minimale Anzahl an Wald-Feldern im folgenden Schritt verbrennen würde.

Offensichtlich ist eine solche Greedy-Strategie nicht (für alle Eingaben) optimal, da es von Vorteil sein kann ein Feld zu löschen, das nicht gebrannt hat (vgl. Abschnitt 1.2).

Wenn wir ein $n \times m$-Feld haben können wir, wie in Abschnitt 1.2 gezeigt, die sinnvollen Löschkandidaten in $\mathbf{O}(nm)$ bestimmen. Für jeden dieser Löschkandidaten muss dann ein Feuer-Zug durchgeführt werden, und für das Resultat müssen die Feuer-Felder gezählt werden. Da die Greedy-Strategie nur Feuer-Felder löschen wird, kommt ihr die Feuerknoten-Datenstruktur zugute. Da ein Feuer-Feld im extremen Fall höchstens drei Wald-Felder in Brand stecken kann, kann man auf der Suche nach der besten Lösung aufhören, sobald man ein Feuer-Feld gefunden hat, das an drei Wald-Felder angrenzt. Wobei zu beachten ist, dass diese drei Wald-Felder nicht an andere Feuer-Felder angrenzen dürfen.

Anstatt also den kompletten Feuer-Zug für jeden Löschkandidaten durchzuführen, was in einer Laufzeit von $\mathbf{O}(n^2m^2)$ resultieren würde, kann man also pro Löschkandidat bis zu maximal 4 direkte Nachbarn und jeweils höchstens 4 indirekte Nachbarn, also 16 Zellen anschauen. Das führt zu einer Laufzeit von $\mathbf{O}(nm)$.

1.4 Erweiterungen

Bei einem Problem, das im Wesentlichen als Spiel zu beschreiben ist, lassen sich die Spielregeln natürlich ändern. Hier kann man kreativ werden; einige sinnvolle Ansatzpunkte sind:

- die Art der Ausbreitung des Feuers: z.B. auch in diagonal benachbarte Felder.
- die Ausbreitungsgeschwindigkeit des Feuers: diese könnte pro Feld variabel sein. zusätzlich zu den Ausbreitungsgeschwindigkeiten 1 für Wald-Felder und 0 für Schneisen und gelöschte Felder könnte es auch 2 für Felder mit besonders leicht brennbarem Bewuchs geben. Auch könnte die Ausbreitungsgeschwindigkeit von der Richtung abhängig sein, womit man den Einfluss von Wind auf Feuer berücksichtigen würde. Wenn dabei die Richtung während des Spiels wechselt, würde das drehenden Winden entsprechen.
- die Auswirkungen des Lösch-Zuges: es könnten auch mehrere Felder pro Zug gelöscht werden – hierfür muss eine ordentlich gemachte Lösung allerdings nur geringfügig angepasst werden.

1.5 Bewertungskriterien

- Die „Spielregeln“ müssen korrekt erkannt und umgesetzt sein: Es muss erkannt worden sein, dass sich zuerst das Feuer ausbreitet und dann gelöscht wird. Das Ausbreiten des Feuers muss korrekt implementiert sein; insbesondere müssen Schneisen und gelöschte Felder berücksichtigt sein.
- Die genannten Kriterien zur Auswahl von Löschkandidaten (das Trivial-, das Feuerlinien-, das Inferno- und das Endzug-Kriterium) müssen erkannt worden sein, wenn sie für die gewählte Strategie relevant sind.
- Das gewählte Verfahren sollte auch nicht aus anderen Gründen als den nicht erkannten Kriterien besonders ineffizient sein; auch größere Waldstücke sollten bearbeitet werden können.

- Der vorgestellte Algorithmus sollte eine optimale Lösung finden. Andernfalls sollte die diesbezügliche Schwäche des Verfahrens erkannt worden sein.
- Der Algorithmus muss mit mehreren Brandherden funktionieren.
- Die Zusatzfrage muss beantwortet sein. Ein Beispiel, bei dem das „Löschen“ intakten Waldes vorteilhaft ist, genügt zwar, um die Frage mit „Ja“ zu beantworten. Erwartet werden aber verallgemeinernde Überlegungen, die nicht nur spekulativ, sondern durch systematische Überlegungen oder weitere Beispiele untermauert sein sollten.
- Da eine naive Lösung dieser Aufgabe offensichtlich zu Laufzeitproblemen führt, werden vernünftige Überlegungen zur Laufzeit erwartet. Für besonders gute Analysen kann es Pluspunkte geben.
- Der Algorithmus muss anhand des Beispiels auf dem Aufgabenblatt (auch wenn dies nicht explizit gefordert wurde) und mindestens zwei weiterer Beispiele vorgestellt werden. Es muss ersichtlich sein, in welcher Reihenfolge wo gelöscht wird. Insbesondere genügt es also nicht, nur die Endsituation darzustellen, ohne die Löschstellen und -folge anzugeben.
- Die Eingabe von Waldstücken und den verschiedenen Feldarten sollte unproblematisch möglich sein, etwa mittels eines gut nachvollziehbaren Dateiformats oder durch eine brauchbare GUI.

Aufgabe 2: Lebenslinien

2.1 Lösungsidee

Die *Lebenszeit* eines Menschen ist in der Aufgabenstellung nicht präzise definiert, aber natürlicherweise handelt es sich um eine kontinuierliche Phase zwischen zwei Zeitpunkten. Wenn wir davon ausgehen, dass sich jedem Zeitpunkt eineindeutig eine reelle Zahl zuordnen lässt, lässt sich die Lebenszeit auch als abgeschlossenes Intervall $L = [a,b] \subset \mathbb{R}$ angeben. Im Folgenden werden wir also Lebenszeiten als solche Intervalle auffassen. In der Graphentheorie werden (potenzielle) Lebensgraphen deshalb auch *Intervallgraphen* genannt.

Ein ungerichteter Graph $G = (V,E)$ ist durch seine Knotenmenge V und Kantenmenge E gegeben, wobei die Kanten ungeordnete Paare $\{u,v\}$ zweier Knoten sind ($u,v \in V$). Ein *Lebensgraph* ist dann ein ungerichteter Graph, für den eine Funktion $f : V \mapsto Pot(\mathbb{R})$[1] definiert ist, welche jedem Knoten eine Lebenszeit eines Menschen, also ein Intervall reeller Zahlen so zuordnet, dass $\forall u,v \in V : \{u,v\} \in E \Leftrightarrow f(u) \cap f(v) \neq \emptyset$ 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, zu dem beide Menschen gelebt haben.

Unsere Aufgabe ist es, für einen gegebenen ungerichteten Graphen $G = (V,E)$ zu prüfen, ob es (mindestens) eine Funktion $f : V \mapsto Pot(\mathbb{R})$ gibt, so dass G Lebensgraph wird; genau dann spricht die Aufgabenstellung von einem potenziellen Lebensgraphen (PLG). Falls es ein solches f gibt, soll $f(v)$ für alle Knoten $v \in V$ ausgegeben werden; andernfalls soll der minimale Teilgraph von G ausgegeben werden, für den allein es kein solches f geben kann.

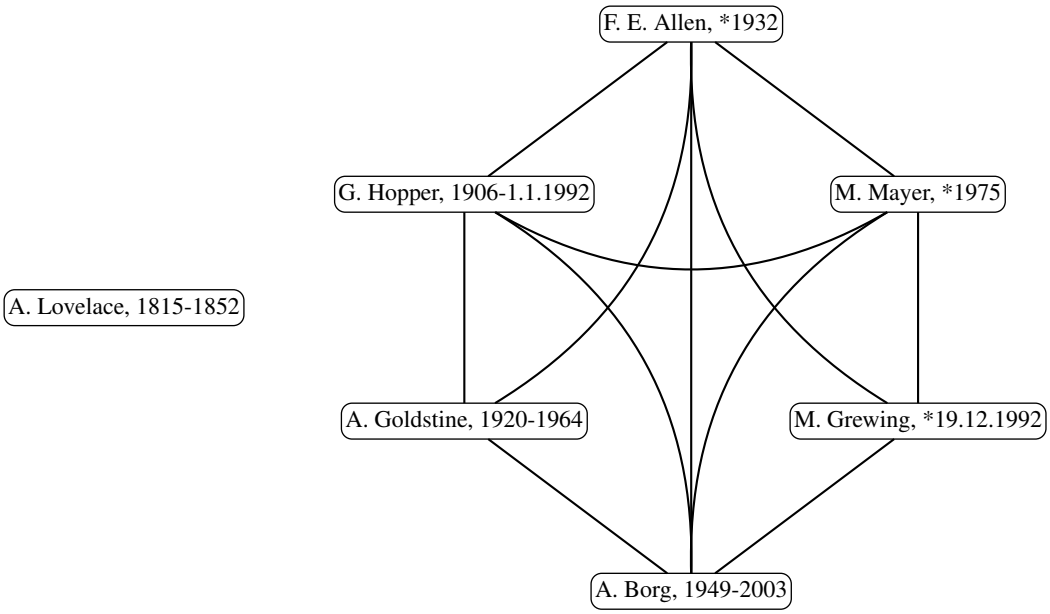


Abbildung 4: Der Lebensgraph aus der Aufgabenstellung

[1] $Pot(\mathbb{R})$ bezeichnet die Potenzmenge von $\mathbb{R}$, also die Menge aller Teilmengen von $\mathbb{R}$.

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

Eigenschaften von Lebensgraphen

Ein so genannter „naiver“ Algorithmus zur Prüfung eines Graphen $G = (V,E)$ auf PLG-Eigenschaft könnte im Wesentlichen daraus bestehen, alle möglichen zeitlichen Anordnungen der Knoten zueinander durchzuprobieren. Dies wird schon bei etwas größeren Graphen aber nicht zum Ziel führen, da solch ein Verfahren mit einer grob approximierten Laufzeit von $\mathbf{O}(|V|!)$ wohl zu langsam ist.

Es wäre daher hilfreich, bestimmte Eigenschaften von Graphen zu kennen, aus denen sich die PLG-Eigenschaft entweder ableiten oder ausschließen lässt. Wir wollen deshalb Lebensgraphen bzw. PLGs etwas genauer betrachten. Eine Kante in einem PLG steht ja für die nichtleere Schnittmenge der durch ihre Knoten gegebenen Intervalle. Kann es spezielle Konstellationen von Knoten und Kanten geben, die gegen diese „Schnittbedingung“ für Kanten grundsätzlich verstoßen?

Es fällt zunächst auf, dass ein Graph, in dem ein *Loch*[2] auftritt, niemals Lebensgraph sein kann:

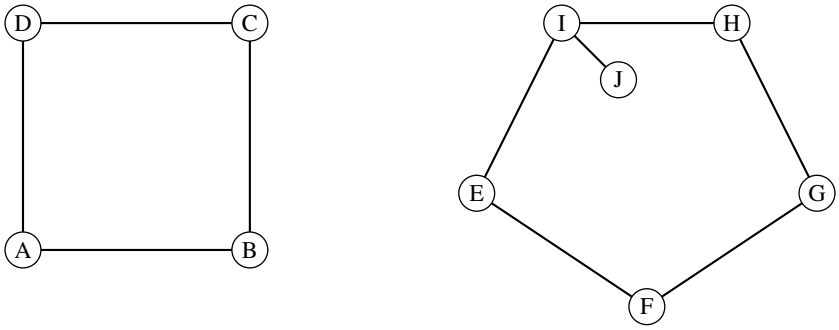


Abbildung 5: Graphen mit Löchern können niemals Lebensgraph sein (im 2. Graphen ist J *nicht* Teil des Loches).

Der Grund hierfür ist offensichtlich: Sei $Z = (v_0, v_1, ..., v_k = v_0)$ ein Zyklus der Länge $k > 3$ in einem Graphen $G = (V,E)$, und gelte für G: zwischen 2 Knoten aus Z existiert genau dann eine Kante in G, sofern diese beiden Knoten im Zyklus benachbart sind; bei Z handelt es sich also um ein Loch von G.

Nun weise die „Lebensgraph-Funktion“ f dem Knoten $v_0 \in Z$ nun ein Intervall $f(v_0) = [a_0, b_0]$ zu. Nun muss dem Nachfolger v_{i_1} von v_i im Zyklus Z ein Intervall $f(v_1) = [a_1, b_1]$ 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

[2]Ein Loch ist ein Zyklus mit einer Länge größer 3, zwischen dessen Knoten genau dann eine Kante existiert, wenn die Knoten im Zyklus benachbart sind.

entschieden, so muss man sich bei der Zuweisung von Intervallen zu den nächsten Knoten in Z immer für den gleichen 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_0 und seinem Vorgänger v_{k-1} im Zyklus Z. Wenn nämlich $f(v_{k-1}) \cap f(v_1) \neq \emptyset$, dann hat $f(v_{k-1})$ auch einen nicht-leeren Schnitt mit allen anderen den Zyklusknoten zugewiesenen Intervallen $f(v_i)$. Das aber steht im Widerspruch zur Annahme, dass der Zyklus auch ein Loch ist. Somit lässt sich für ein Loch keine Lebensgraph-Funktion finden, und andersherum kann ein Lebensgraph kein Loch haben.

Graphen ohne Löcher werden in der Literatur *Chordalgraph* oder *Triangulierter Graph*[3] genannt; es gibt effiziente Algorithmen zur Erkennung solcher Graphen.

Es sei an dieser Stelle angemerkt, dass ein Lebensgraph sehr wohl *Dreiecke*, also Zyklen der Länge 3 haben kann. 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.

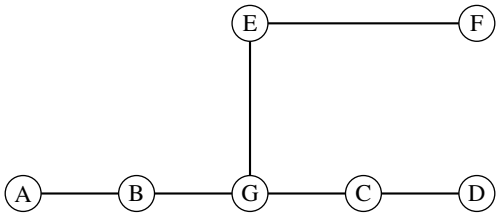


Abbildung 6: Chordalgraph, der *kein* Lebensgraph ist

Weiterhin ist es für einen Lebensgraphen nur *notwendig* Chordalgraph zu sein; also: Jeder Lebensgraph ist auch Chordalgraph. Andersherum gilt das nicht: Der Graph in Abbildung 6 ist Chordalgraph, kann jedoch nicht Lebensgraph sein.

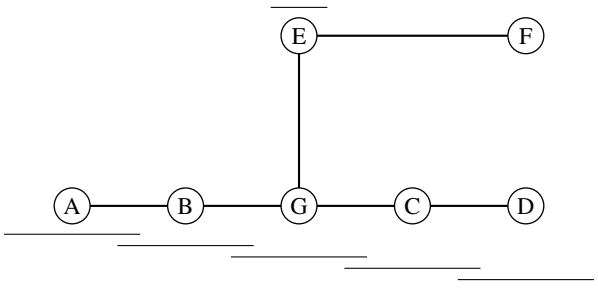


Abbildung 7: Versuch einer Intervallbelegung

Offensichtlich werden die Möglichkeiten, Knoten mit Intervallen zu belegen, durch „Linien“ eingeschränkt, also Pfade, bei denen ausschließlich im Pfad aufeinanderfolgende Knoten durch Kanten verbunden sind. Eine Linie von mindestens drei Knoten lässt sich nur mit einer

[3] Der englischsprachige Wikipedia-Artikel ist in diesem Fall (mal wieder) deutlich informativer: `https://en.wikipedia.org/wiki/Chordal_graph`

Folge von Intervallen belegen, deren obere Grenzen wachsen und in der nur benachbarte Intervalle einen nicht-leeren Schnitt haben: eine „Intervall-Treppe“ wie in Abb. 7 unter der Linie $A-B-G-C-D$ zu sehen. Deshalb ließ sich schon ein Loch nicht passend mit Intervallen belegen. Eine Abzweigung von einer Linie, wie die von Knoten G zu Knoten E, ist mit Länge 1 noch möglich; für E lässt sich ein Intervall finden, das ein echter Teil des Intervalls von G ist. Nur fortführen lässt sich die Abzweigung nicht; jedes Intervall, dass sich mit dem von E schneidet, muss sich auch mit dem von G schneiden.

Es genügt also nicht, in einem Graph das Vorhandensein von Löchern auszuschließen, um ihn als PLG zu erkennen. Es gilt nun also ein *hinreichendes Kriterium* dafür zu finden, dass ein Graph G ein PLG ist.

Seien dazu die *maximalen Cliquen* in G betrachtet. Eine Clique C ist eine Knotenteilmenge $C \subseteq V$ eines Graphen $G = (V,E)$, in der jeder Knoten mit jedem anderen durch eine Kante verbunden ist. Eine solche Clique heißt 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.

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.
Speziell bei PLGs 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 aufeinander folgen. Diese Anordnung sei im Folgenden mit *Cliquenkette* bezeichnet. Aus einer Cliquenkette lassen sich nun leicht Intervalle für Knoten ablesen – und umgekehrt, da zwei Knoten nur in derselben Clique sind, wenn sie durch eine Kante verbunden sind, und ihnen so zurecht ein gemeinsamer Intervallabschnitt zugeordnet worden ist.

Algorithmische Erkennung von Lebensgraphen

Der vorangegangene Abschnitt liefert nun einen direkten Algorithmus zur Überprüfung, ob ein Graph ein PLG ist. Zunächst wird überprüft, ob der gegebene Graph ein Chordalgraph ist. Dann wird ein Cliquenbaum erzeugt und überprüft, ob dieser eine Cliquenkette ist. Zuletzt wird noch geprüft, ob jeder Knoten nur in aufeinander folgenden Cliquen vorkommt. Die für die einzelnen Schritte notwendigen Algorithmen[4] 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.

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

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

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
      ersetze { i } durch { nachbarn , nicht_nachbarn } in S;
            //Ignoriere leere Mengen
    end;
  end;
  return ausgabe;
end.
```

Eine *perfekte Eliminationsanordnung* 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 Eliminationsanordnung besitzt. Auch kann bewiesen werden, dass die Lex-BFS bei einem Chordalgraphen G eine perfekte Eliminationsanordnung 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 Eliminationsanordnung ist.

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

```
//Überprüfung einer Ordnung der Knoten V darauf, ob diese
//eine perfekte Eliminationsanordnung ist
//Eingabe: Graph G = (V,E), Reihenfolge PI der Knoten V
//Ausgabe: true, falls PI perfekte Eliminationsanordnung, 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 Eliminationsanordnung 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 Eliminationsanordnung, 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 aber 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 ∩ b
        und Xb durch Xb ∩ b, Xb\b;
    end;
    foreach (Ci,Cj) in F mit Ci in b und Cj nicht in b do begin
      PIVOTS ∪= Ci ∩ 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 PLG, 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.

Erkennung des kleinsten Teilgraphen, der kein PLG ist

Zunächst ist es einfach zu erkennen, dass es nach obigem Algorithmus verschiedene Arten von Graphen gibt, die keine PLGs sind. Es gibt diejenigen Graphen, die keine Chordalgraphen sind (also ein Loch aufweisen), und diejenigen Graphen, die zwar Chordalgraphen, aber trotzdem keine PLGs sind.

Weiterhin ist es leicht zu sehen, dass jeder (zusammenhängende) Graph mit drei oder weniger Knoten ein PLG ist.

Ein nahe liegender Ansatz zur Findung des kleinsten Teilgraphen ist es, alle möglichen Teilgraphen ab Größe 4 der Größe nach zu überprüfen, ob diese kein PLG sind. Dabei können schon von vorne herein gewisse Teilmengen ausgeschlossen werden. So ist es nicht sinnvoll, Cliquen zu überprüfen; außerdem sollte der Teilgraph zusammenhängend sein.

Laufzeitanalyse

Bei geschickter Implementierung kann erreicht werden, dass jeder der obigen Algorithmen bei einem Graphen $G = (V,E)$ eine Laufzeit von $\mathbf{O}(|V| + |E|)$ besitzt.

So wird bei einer Lex-BFS nur jeder der Knoten einmal dazu benutzt, um die restlichen Knoten zu partitionieren. Dabei werden notwendigerweise auch alle Kanten betrachtet, da bei einer Partitionierung Nachbarschaften zwischen betrachtet werden müssen. Somit ergibt sich eine Laufzeit von $\mathbf{O}(|V| + |E|)$

Das Ermitteln von $RN(v)$ für alle Knoten $v \in V$ geht ebenfalls in $\mathbf{O}(|V| + |E|)$. Das Überprüfen, ob eine Menge Teilmenge einer anderen ist, geht bei sortierten Mengen ebenfalls in Linearzeit, und um die Knoten in Linearzeit zu sortieren bietet sich *Radixsort* an. Somit ist auch das Finden des Cliquenbaumes einfach in Linearzeit möglich.

Der Algorithmus zur Findung der Cliquenkette benutzt jeden Knoten höchstens 1 mal als Pivot-Element. Weiterhin existieren in G trivialerweise $\mathbf{O}(|V|)$ maximale Cliquen, alle diese Cliquen zusammen sind asymptotisch gleich groß zu dem Graphen, sind also $\mathbf{O}(|V| + |E|)$

groß. Weiterhin kann man zeigen, dass jede Clique höchstens einmal für jeden in ihr enthaltenen Knoten betrachtet wird. Somit ergibt sich hierfür eine Laufzeit von $\mathbf{O}(|V|+|E|)$.

Weiterhin hat jede Clique in dem Cliquenbaum höchstens einen Elternknoten. Dies führt, zusammen mit der Größenbeschränkung aller Cliquen zusammen ($\mathbf{O}(|V|+|E|)$), dazu, dass nur $\mathbf{O}(|V|+|E|)$ mal ein Knoten zu den Pivot-Elementen hinzugefügt wird. Für die restlichen Operationen ist diese Laufzeitschranke nun einfach zu zeigen.

Diese Schranke erfüllt auch die geforderte Effizienz des Verfahrens.

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

2.2 Weitere Lösungsideen

Neben der Erkennung mithilfe einer Lex-BFS sind auch andere Methoden denkbar. So kann man auch mithilfe eines PQ-Baumes diese Erkennung vornehmen.[5] Auch können für die Erkennung weitere Eigenschaften von Lebensgraphen ausgenutzt werden, so z.B., dass Lebensgraphen auch genau diejenigen Chordalgraphen sind, welche kein *asteroidales Tripel*[6] enthalten. Es sind noch weitere Verfahren leicht im Internet zu finden. Alle diese Verfahren eint, dass sie eine polynomielle Laufzeit für die Erkennung von Lebensgraphen besitzen.

Für das Finden des kleinsten Teilgraphen, der kein Lebensgraph ist, wären auch Heuristiken denkbar. Auch können die speziellen Eigenschaften von Chordalgraphen und Lebensgraphen ausgenutzt werden, um eine Suche stärker zu beschneiden. So kann bei nicht-chordalen Graphen ausgenutzt werden, dass Chordalgraphen keine Löcher besitzen, und somit nach dem kleinsten Loch gesucht werden.

2.3 Erweiterungen

Diese Aufgabe scheint zunächst in sich abgeschlossen, einige kleinere Erweiterungen sind dennoch denkbar. Zum einen wäre es denkbar, Graphen die keine Lebensgraphen sind so zu erweitern (zu „reparieren“), dass sie Lebensgraphen werden. Dabei wäre allerdings nur die minimale Erweiterung interessant, jeder Graph kann schließlich zu einem vollständigen Graphen erweitert werden. Zum anderen können Menschen (in der Regel) nicht unbegrenzt leben. Man könnte also auch eine maximale Lebenszeit festlegen, diese Festlegung benötigt jedoch noch weitere Einschränkungen, um sinnvoll zu sein. Schließlich könnten die Grenzen aller Intervalle einfach mit einer reellen Konstanten multipliziert werden, um jedwede Schranke zu unterschreiten. Daher könnte man Lebenszeiten nur auf ganzen Zahlen definieren, um dies zu unterbinden.

[5] Booth, K., Lueker, S.: Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms. In: Journal of Computer and System Sciences 13, 335–379 (1976)

[6] Drei Knoten in einem Graphen G formen ein *asteroidales Tripel* genau dann, wenn jeweils 2 dieser Knoten durch einen Pfad in G verbunden sind, welcher keinen Nachbarknoten des dritten enthält. Der Graph in Abbildung 6 enthält ein asteroidales Tripel, nämlich die Knoten A, D und F; er ist auch ein minimaler Graph mit dieser Eigenschaft.

2.4 Beispiele

In diesem Abschnitt werden Ergebnisse gezeigt, die eine Implementierung des oben vorgestellten Lösungsansatzes liefert. Für die Berechnung der einzelnen Beispiel wird jeweils weniger als eine Sekunde benötigt.

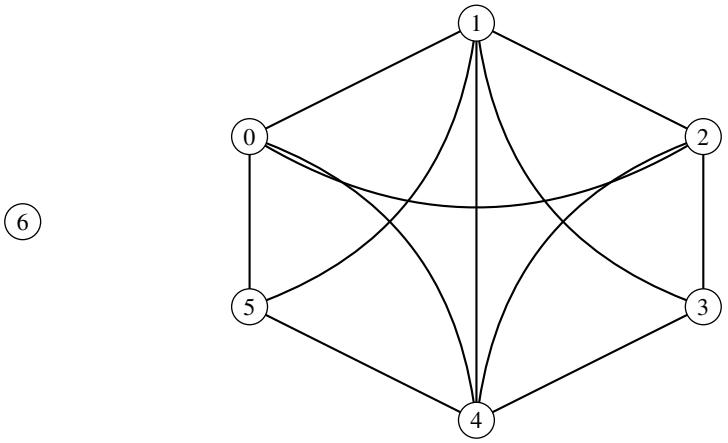


Abbildung 8: Der Graph aus Beispiel 0.

Beispiel 0

Das Beispiel aus der Aufgabenstellung.

```
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:

```
The graph size.
7 lines with 7 space separated integers;
the graph as adjacency 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
```

Beispiel 1

Ein Graph, der kein Chordalgraph ist.

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

Die Ausgabe:

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

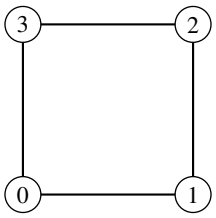


Abbildung 9: Der Graph aus Beispiel 1.

Beispiel 2

Ein weiterer Graph, der nicht chordal ist.

```
6
0 1 0 0 1 0
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:

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

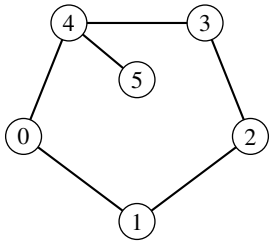


Abbildung 10: Der Graph aus Beispiel 2.

Beispiel 3

Ein Dreieck, das offenkundig ein PLG ist.

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

Eine gefundene Belegung mit Intervallen:

```
The graph size.
3 lines with 3 space separated integers;
the graph as adjacency matrix.
The graph is chordal!
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
```

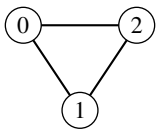


Abbildung 11: Der Graph aus Beispiel 3.

Beispiel 4

Ein weiteres Beispiel.

```
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.

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

Beispiel 5

Ein weiteres Beispiel für einen PLG.

```
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 Intervallen:

```
The graph size.
7 lines with 7 space separated integers;
the graph as adjacency 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
```

2.5 Bewertungskriterien

- Eine erste entscheidende Hürde bei dieser Aufgabe ist das Verständnis der Problemstellung. Die Definition des *potenziellen* Lebensgraphen ist kompliziert. Ein mögliches Missverständnis kann dazu führen, dass statt der PLG-Erkennung die wesentlich einfachere Prüfung konkreter Lebensgraphen (also mit gegebenen Zuweisungen von Intervallen zu Knoten) auf Korrektheit realisiert wurde.
- Die Lösung soll die Anforderungen der Aufgabe erfüllen:
  - Korrekte Identifikation von PLGs und Nicht-PLGs.
  - Für PLGs korrekte Berechnung passender Lebensspannen oder Intervalle.
  - Korrekte Angabe minimaler Teilgraphen, wenn es sich nicht um einen PLG handelt.

  Einschränkungen sind an dieser Stelle akzeptabel, wenn die Probleme begründet auf die Ineffizienz des Verfahrens zurückzuführen sind. Bei Lösungen, die nicht auf Literatur zurückgreifen, sind Schwächen akzeptabel, wenn sie erkannt sind.
- Es gibt effiziente Verfahren zur Lösung der Aufgabe, die als Forschungsergebnisse bekannt sind. Für Verfahren, die auf eigenen Überlegungen beruhen, sind schwächere Laufzeiten akzeptabel. Verfahren mit exponentiellen Laufzeiten im worst-case (z.B. eine vollständige Suche über Zuordnungen von Lebenszeit-Relationen zu den Kanten des Graphen) sind akzeptabel, wenn versucht wurde, sie geeignet zu optimieren. Inakzeptabel sind Verfahren mit Laufzeitverhalten in der Größenordnung von $n!$, wobei Optimierungen anerkannt werden.
- Das Laufzeitverhalten sollte betrachtet und (nicht notwendigerweise formal; das kann Pluspunkte geben) begründet werden.
- Für die Erkennung von Potenziellen Lebensgraphen bzw. *Intervallgraphen* lassen sich leicht Algorithmen im Internet finden. Es ist akzeptabel, wenn die Ansätze zur Lösung dieser Aufgabe aus der Fachliteratur übernommen wurden. Es ist schon eine Leistung, die Literatur soweit zu verstehen, dass man die Lösungen übernehmen und implementieren kann. Aber: Eine „Literatur-Lösung“ wird gegenüber eigenständig erarbeiteten Ideen nicht zu hoch bewertet. Deshalb soll bei Literatur-Lösungen die Beschreibung der Lösungsideen nachvollziehbar vermitteln, dass der Einsender verstanden hat, warum die in der Literatur angegebenen Verfahren funktionieren. Andererseits: Wenn nicht auf Literatur verwiesen wird, sollte ebenfalls gut begründet sein, wie und warum die Verfahren funktionieren. Selbstverständlich sollen Plagiate nicht belohnt werden.

- Auch ohne Rückgriff auf Literatur sind einige zentrale Ideen erreichbar. Mit etwas Nachdenken kann man die wesentlichen Ausschlusskriterien für PLGs, nämlich das Vorhandensein eines Loches bzw. der Grundform eines asteroidalen Tripels („Stern“) durchaus erkennen. Es wird belohnt, wenn diese Kriterien eigenständig gefunden wurden.
- Bei der Verwendung von Fachliteratur – ob elektronisch zugänglich oder nicht – wird eine angemessene Referenzierung erwartet.
- Die Funktionalität des beschriebenen Verfahrens sollte anhand von aussagekräftigen Beispielen dokumentiert sein. Konkrete Laufzeiten für die Beispiele sind hilfreich (aber nicht gefordert), speziell bei Beispielen, bei denen die Eingabe kein PLG ist.
- Eine grafische Ausgabe einer Zuweisung von Lebenszeiten zu Knoten bzw. der minimalen Teilgraphen ist zwar schön, aber nicht gefordert; eine nicht-grafische Ausgabe sollte aber leicht verständlich sein. Hingegen ist es sehr sinnvoll, den in einem Beispiel untersuchten Graphen auch abzubilden.
- Die Eingabe der Graphen sollte unproblematisch möglich sein, etwa mittels eines gut nachvollziehbaren Dateiformats oder durch eine brauchbare GUI.

Aufgabe 3: Vortänzer Challenge

Schon in der Aufgabe „Vortänzer“ der ersten Runde war ein Wettbewerb zwischen den Tanzrobotern des Typs „Dancemaster 2000“ beschrieben worden. Es ging darum, dass ein Roboter den Tanz bzw. das Tanzprogramm eines anderen durch ein eigenes Tanzprogramm (das *Imitat*) „nachmachen“ sollte, und zwar unter Vermeidung von – nach bestimmten Regeln vergebenen – Strafpunkten. Da lag es nahe, diesen Wettbewerb in ähnlicher Weise zum Thema einer Turnier-Aufgabe der zweiten Runde zu machen. Dabei waren im Wesentlichen zwei Teilaufgaben zu bearbeiten:

Vortanzen Bestimme Tanzprogramme, für die es schwierig ist, ein mit möglichst wenigen Strafpunkten zu bewertendes Imitat zu finden.

Nachtanzen Finde für ein Tanzprogramm ohne Wiederholungsbefehle (kurz: Schleifen) ein mit möglichst wenigen Strafpunkten zu bewertendes Imitat.

Im Folgenden werden Ansätze zur Bearbeitung dieser Teilaufgaben beschrieben. Vorher sollen aber einige dazu nützliche Begriffe in ihrer Bedeutung festgelegt werden.

3.1 Begriffe

Wir wollen einige Begriffe einführen, um besser über Tanzprogramme reden zu können. Ein Tanzprogramm besteht aus Bewegungsbefehlen und Schleifen. Die Ausführung des Tanzprogramms (kurz: *Tanz*) kann wiederum als Folge von Bewegungsbefehlen beschrieben werden – also als schleifenfreies Tanzprogramm. Die Eingabe für das Nachtanzen ist ein auf 255 Zeichen abgeschnittener Tanz. Das *Imitat I* eines Tanzprogramms T ist selbst ein Tanzprogramm. Die *Bewertung* eines Imitats bezüglich eines Tanzprogramms ist durch die Strafpunkteregelung fürs Nachtanzen (aus der ersten Runde) gegeben. Wir können auch von der Bewertung eines Tanzprogramms (bezüglich sich selbst) sprechen; hat das Tanzprogramm die Länge l, ist diese Bewertung immer $3 \cdot l$ (da Strafpunkte nur für die verwendeten Zeichen vergeben werden).

Ein *perfektes Imitat* eines Tanzprogramms generiert exakt den gleichen Tanz wie das Ausgangsprogramm. Ein *verbesserndes Imitat* eines Tanzprogramms T hat eine bessere Bewertung (bzgl. T) als T selbst. Ein Tanzprogramm T ist *minimal*, wenn es kein kürzeres perfektes Imitat von T gibt als T selbst; andernfalls ist T *komprimierbar*. Ein komprimierbares Tanzprogramm enthält Teilprogramme mit Wiederholungen, die in einem kürzeren perfekten Imitat mit Hilfe des Wiederholungsbefehls ausgedrückt werden können. Beispiel: `4rrFFFF.` hat das kürzere perfekte Imitat `4rr4F..` (nicht aber `42r.4F..` – ein Teilprogramm mit Wiederholungen muss mindestens Länge 3 haben, um mit dem Wiederholungsbefehl kürzer formulierbar zu sein). Die *minimale Kompression* eines Tanzprogramms T ist ein minimales perfektes Imitat von T.

3.2 Nachtanzen

Beim Nachtanzen kann man zunächst davon ausgehen, dass die Bewegungsfolge der Tanz eines minimalen Tanzprogramms ist, zu dem es auch kein verbesserndes Imitat gibt. Damit ist die beste Möglichkeit, aus dem Tanz das originale Tanzprogramm zu rekonstruieren. Wenn das perfekt gelingt, gibt es keine Strafpunkte für Abweichungen, und die Bewertung ergibt sich allein aus der Länge des gefundenen Programms. Rekonstruktion bedeutet hier, für die im Tanz enthaltenen Wiederholungen die passenden Wiederholungsbefehle (Schleifen) zu finden. Dies lässt sich auf verschiedenen Wegen versuchen.

Greedy

Es wird versucht, rekursiv (Schleifen können verschachtelt sein) den vorliegenden Tanz durch Einführung von Schleifen zu verkürzen. Auf jeder Rekursionsebene wird diejenige Ersetzung eines Teilprogramms mit Wiederholung durch einen Wiederholungsbefehl gewählt, welche die Bewertung am meisten verbessert. Beispiel: Das Tanzprogramm `rrrrFrrrrFrrrrF` wird im ersten Schritt zu `3rrrrF.` und im zweiten Schritt zu `34r.F.` umgeschrieben.

Fortgeschrittene Schleifenerkennung

Ein solch scheinbar einfaches Verfahren zur Schleifenerkennung kann sehr erfolgreich sein, wenn die folgenden kritischen Punkte geeignet behandelt werden.

Überlappende Schleifen Wenn sich komprimierbare Teilstücke des Tanzes überlappen, muss darauf geachtet werden, die beste Möglichkeit zu wählen. Im Tanz `FrrFrrrBrrrB` überlappen sich `FrrFrr` und `rrrBrrrB`; letzteres Stück sollte komprimiert werden.

Offene Schleifen Wenn der Tanz volle 255 Zeichen lang ist, kann es sein, dass das originale Tanzprogramm einen Tanz von mehr als 255 Zeichen produziert hat, dieser aber auf die Maximallänge beschnitten wurde. Es kann sich deshalb lohnen, die letzten Befehle des Tanzes mit in eine Schleife zu packen, auch wenn diese Befehle nur den Anfang des Schleifenkörpers bilden. Beispiel: `FrrrFrrrFr` (Zeichen 246 bis 255 des Tanzes) lässt sich zu `2Frrr.Fr` komprimieren, besser aber zu `9Frrr.` (wobei in solchen Fällen als Wiederholungszahl immer die 9 genommen werden kann – das schadet nicht).

Schleifen versetzen Die Wahl der auf einer Rekursionsebene gefundenen „beste“ Schleife stellt evtl. nur lokal die beste Lösung dar, verhindert also möglicherweise die insgesamt beste Kompression. Beispiel: Von links ausgehend, findet man in `FrrFrrrBrrrBrrrBrr` als scheinbar bestes Wiederholungsstück `rrrBrrrBrrrB`. Besser wäre aber, die Auswahl um eins nach rechts zu versetzen: dann lässt sich immer noch eine 3-fache Wiederholung finden, nämlich `rBrrrBrrrBrr`, aber zusätzlich ist dann auch der Anfang des Tanzes eine Wiederholung: `FrrFrr`.

Schleifenzähler bestimmen Die Wiederholungszahl einer Schleife liegt sinnvoll zwischen 2 und 9 (einschließlich). Begegnet man im Tanz einer 63-fachen Wiederholung der Teilfolge t, kann dies zu $79t..$ komprimiert werden. Eine 65-fache Wiederholung hingegen lässt sich nicht so einfach behandeln: die Primfaktoren von 65 sind 5 und 13. Hier müssen Wiederholungsbefehle und einzelne Befehle ideal miteinander kombiniert werden, etwa so: $88t..t$.

Endabweichung Ein paar Strafpunkte lassen sich evtl. sparen, falls die beste Kompression nicht mit einem . endet. Einzelne Bewegungsbefehle am Ende können einfach weggelassen werden, solange die Anzahl der damit eingehandelten Abweichungs-Strafpunkte kleiner ist als die der eingesparten Befehlszeichen-Strafpunkte. Hier kann also Nichtstun besser sein als Bewegung.

Zusammensetzung aus Teillösungen

Es lässt sich zeigen, dass sich die minimale Kompression eines Tanzprogramms aus minimalen Kompressionen von Teilstücken zusammensetzen lässt, die sich wieder aus minimalen Kompressionen ihrer Teilstücke zusammensetzen lassen usw. Deshalb kann man die minimale Kompression mit eine Verfahren nach dem Prinzip der Dynamischen Programmierung ermitteln; dieses bestimmt optimale Lösungen durch Zusammensetzung aus optimalen Lösungen von Teilproblemen. Wenn man davon ausgeht, dass das originale Tanzprogramm maximal 10 Zeichen lang ist, können nach der gleichen Idee alle möglichen Schemata für die Schleifenstruktur von Tanzprogrammen gebildet und mit dem zu komprimierenden Tanz abgeglichen werden.

Abweichungen behandeln

Weiter unten werden wir sehen, dass es für den Vortänzer sinnvoll ist, ein Programm T zu wählen, für das es kein verbesserndes Imitat gibt, also kein anderes Programm I, das bzgl. T besser bewertet wird als T selbst. Damit kann sich der Nachtänzer wiederum darauf beschränken, ein perfektes Imitat ohne Abweichungen von der Vortänzerbewegung zu bestimmen – und dazu genügt eben ein Verfahren zur Rekonstruktion der Schleifen. Wenn wir aber die zu Beginn formulierte Aufgabe des Nachtanzens ernst nehmen, dann sollten auch Bewegungsabweichungen berücksichtigt und folglich auch verbessernde Imitate gefunden werden können. Dazu genügt eine reine Schleifenrekonstruktion bzw. Kompression nicht. Eine mögliche Lösung ist, eine gute Kompression durch eine vollständige Generierung aller möglichen Tanzprogramme zu ergänzen – und dabei das Ergebnis der Kompression als Einschränkung dieser vollständigen Suche mitzuführen. Aus Zeitgründen wird aber eine solche Brute-Force-Ergänzung nicht in allen Fällen vollständig laufen können, so dass die Bestimmung eines optimalen Imitats nicht für alle Fälle garantiert ist.

Backtracking

Auch ein geschicktes Backtracking kann erfolgreich sein, ohne den Laufzeitrahmen des Turnierservers zu sprengen. Darin lässt sich sogar „Fehlertoleranz“ einbauen, um auch Abweichungen berücksichtigen zu können.

3.3 Vortanzen

Ein zum Vortanzen gewähltes Programm sollte minimal sein, damit der Nachtänzer keine Chance hat, ein kürzeres perfektes Imitat zu finden. Außerdem sollte es kein verbesserndes Imitat eines Vortanzprogramms geben. Das kann durch möglichst viel Bewegung (also durch die Befehle F und B) erreicht werden.

Vorgetanzte minimale Tanzprogramme sollten die Länge 10 haben, damit ein maximaler Bewertungsabstand zwischen vorgetanztem und nachgetanztem Programm erreicht werden kann: Vorausgesetzt, dass ein Gegner es schafft, ein (gleich langes) perfektes Imitat eines vorgetanzten minimalen Tanzprogramms zu finden, erhält dieses bei Länge 10 die meisten Strafpunkte (30), während der Vortänzer keine Strafpunkte erhält. Bei weniger als 10 Zeichen bleibt der Vortänzer ebenfalls ohne Strafpunkte, aber das Imitat wird für jedes Zeichen weniger mit 3 Strafpunkten weniger bewertet; der Bewertungsabstand sinkt. Auch längere Tänze lassen den Bewertungsabstand sinken, weil ab Zeichen 11 der Vortänzer mehr Strafpunkte für jedes Zeichen bekommt als der Nachtänzer. Deshalb kann eine KI, die nicht immer Tanzprogramme der Länge 10 vortanzt, gegen perfekte Gegner nicht gewinnen.

Es genügt, drei Vortanzprogramme fest zu wählen und sie in den drei Matches einer Runde nacheinander zu präsentieren. So können die Gegner einer Runde nicht „lernen“. Der Fall, dass man in einem Turnier mehrfach den gleichen Gegner hat und diesem dann mehrfach das gleiche Programm vortanzt, ist zwar unwahrscheinlich. Begegnen kann man ihm dadurch, dass man die Vortanzprogramme variiert, z.B. durch Austausch von B und F bzw. l und r. Die Qualität der Vortanzprogramme wird dadurch nicht beeinflusst.

Hier einige Beispiele für gute Vortanzprogramme:

`299FF..Fr.`	Der Tanz hat „Überlänge“, das zweite `r` kommt im übergebenen String nicht vor.
`99rB.r9B..`	Im Tanz kommt `rB` 10mal hintereinander vor, so dass ein Greedy-Verfahren das mit einem Zeichen zu viel zusammenfassen könnte (`952rB..8B..`).
`FF988F..l.`	Alle drei Schleifen sind am Ende des Tanzes offen; der Tanz beginnt mit 66 `F` Befehlen, von denen aber nur 64 durch eine Schleife zusammengefasst werden sollten.

3.4 Erweiterungen

... der Spielregeln des Turniers sind denkbar, könnten aber nur mit einem eigenen Turnierserver probiert werden. Deshalb ist die Realisierung eines eigenen Turnierservers zwar keine direkte Erweiterung der Aufgabenstellung, eröffnet aber interessante Möglichkeiten.

3.5 Bewertungskriterien

- Das für das Nachtanzen realisierte Kompressionsverfahren (Schleifenerkennung) sollte optimale Ergebnisse liefern. Auch eine nach eher einfachen Regeln funktionierende, rekursive Schleifenerkennung kann durchaus erfolgreich sein. Dabei müssen allerdings die genannten kritischen Punkte beachtet werden (überlappende und offene Schleifen, versetzte Schleifen, Bestimmung von Schleifenzählern).
- Ein reiner Greedy-Ansatz, der auf einer Rekursionsebene die erstbeste Kompression wählt, wird z.B. bei versetzten Schleifen scheitern. Die Mängel eines Vorgehens nach einem reinen Greedy-Prinzip sollten, genau so wie Optimalitätsmängel anderer Verfahren, erkannt worden sein.
- Die Lösung sollte auch in der Lage sein, verbessernde Imitate (also solche mit Abweichung) zu finden – idealerweise besonders gute oder gar optimale verbessernde Imitate.
- Die Vortänze sollten geeignet bestimmt bzw. manuell gewählt sein. Wichtig ist, dass sie erkennbar Eigenschaften haben, die das Nachtanzen erschweren.
- Das Abschneiden im Turnier wird in die Bewertung mit einbezogen. Für besonders gute Platzierungen gibt es Pluspunkte, für besonders schwache gibt es Minuspunkte.
- Da die Rechenzeit einer KI auf dem Turnierserver beschränkt ist, sollte das Zeitverhalten der Lösung diskutiert werden. Dabei sind bei einer Turnieraufgabe auch Betrachtungen absoluter Zeiten sinnvoll und akzeptabel. Formale Analysen können mit Pluspunkten belohnt werden.
- Eine Beschränkung des Nachtanz-Verfahrens auf Kompression bzw. Schleifenrekonstruktion ist akzeptabel, muss aber begründet sein. Ebenso muss begründet sein, wenn sich die Lösung auf die Behandlung von Vortänzen mit maximal oder sogar exakt 10 Zeichen beschränkt.
- Zum Vortanzen sind aufwändige Verfahren zur Suche nach besonders guten Vortänzen zwar interessant, aber nicht nötig. Ob automatisch bestimmt oder fest gewählt: Vortänze sollten Eigenschaften haben, die das Nachtanzen erschweren (s.o.). Es sollte erläutert sein, welche solcher Eigenschaften die gewählten bzw. automatisch generierten Vortänze haben.
- Die Dokumentation soll Beispiele enthalten. Schön ist, wenn eine Turnierrunde mit drei Spielern einmal vollständig dokumentiert wird: also mit allen sechs Eingaben, die die eigene KI erhält, und den drei selbst vorgetanzten Programmen. Genau so viel wert sind natürlich Einzelbeispiele, die möglichst aus dem Turnier stammen, insbesondere bzgl. des Nachtanzens. Aber auch selbst überlegte Beispiele können genügen, wenn sie die Funktionsweise insbesondere des Nachtanzens verständlich machen.
- Die Darstellung der Beispiele sollte gut nachvollziehbar sein.

Perlen der Informatik – aus den Einsendungen

Teilweise mit Kommentaren von der Bewertung

Allgemeines

Worte des Wettbewerbs: Algorythmus, Algrouthmis *Insbesondere die zweite Variante ist echt neu!*

Ich hätte nicht gedacht, dass die allgemeinen Hinweise der 2. Runde wirklich sich als wahr beinhalten.

Ich habe mein Programm in C geschrieben, weil ein Brute-Force in C schneller ist als in allen anderen Programmiersprachen, die ich beherrsche.

Der nicht deterministische Baum ...

... und das Integer Array wird an die Datei zurückgegeben

Da diese Konstanten schnell sehr groß werden ...

Here comes Brutus - The Force!

Es gibt allerdings nicht nur logische Optimierungen, sondern auch welche, die durch falsche Implementierung entstehen.

In der Einsendung aufgrund der Größe zuerst nicht enthalten, sobald jedoch die Audiodatei `"HansZimmer-SuitefromBackdraft.wav"` im Classpath unter dem Ordner `"res"` liegt, so wird optional eine supertolle Hintergrundmusik abgespielt.

Aufgabe 1: Buschfeuer

Funktion: `getRandBrand()`

Learning by Burning

Je quadratischer, desto höher die Laufzeit.

Als einfachen und immer perfekten Algorithmus habe ich Bruteforce implementiert.

Die Bäume werden aufgebaut, indem der Wald abgebrannt wird und die Randfelder hinzugefügt werden.

Sinnvoll ist es dagegen, wichtigere Felder als Wald, nämlich z.B. Städte, vor Brand zu schützen.

Weiterhin fiel mir nach einer Recherche über reelle Waldbrände auf, dass oftmals durch großflächige Brände Einrichtungen von Menschen bedroht werden.

Ohne Optimierung sind es mehr als 10 Minuten (der höchst geduldige Programmierer, der die Zeit messen sollte, ist nach besagten 10 Minuten eingeschlafen).

Hat man es allerdings eilig – zum Beispiel weil der Akku fast leer ist, der Einsendeschluss näher rückt, die Zimmerpartnerin schlafen möchte oder irgendwo ein Wald abfackelt – gibt man sich gerne auch mal mit einer „nur“ sehr guten Lösung zufrieden

Ich habe das ungute Gefühl, dass es sich hier um ein NP-schweres Problem handelt. Wie bei der berüchtigten Hydra aus den griechischen Erzählungen schienen mit jeder Lösung für ein Problem neue Probleme überall aufzutauchen.

Eine weitere Erweiterungsmöglichkeit, die ich privat noch nutzen werde ... *Wenn der eigene Garten brennt?*

Aufgabe 2: Lebenslinien

eine ausgegebene Lebensspanne: keine Zeit mehr

Diese Bedingung ist insofern logisch, da ein Mensch nur lückenlos leben kann.

Die doppelte Überlappung ergibt sich aus der doppelten Gleichzeitigkeit.

Diese Intervalle erinnern nun nicht gerade an Lebensspannen (höchstens in Tagen für Haustiere).

Die Laufzeit ist bei dieser Umsetzung sehr gering. Sie beträgt im schlimmsten Fall $\mathbf{O}(N^2!)$.

Mit diesem Problem haben sich bereits diverse Informatiker auseinandergesetzt. Allerdings ist hinzuzufügen, dass die Dokumentation dieser auf (nicht sonderlich gutem) Englisch erfolgt ist.

Und jetzt zur Lebenspanne ... *Damit ist hoffentlich nicht die Teilnahme am BwInf gemeint!*

Aufgabe 3: Vortänzer Challenge

Das Vortanzprogramm hat vorgegebene Vortanz-Programme für den Roboter.

Nachdem meine KI alle drei Logiken durchlaufen hat, *sollte* mein Danceroboter nun genau den Tanz tanzen, der die geringsten Strafpunkte erhält. *Drei Logiken sind eine ganze Menge – die meisten Leute sind schon mit einer überfordert.*

Wenn der Server nicht bereit ist, mir ausreichend Rechenkapazität zur Verfügung zu stellen, bin ich auch nicht bereit, meinen Speicherbedarf zu optimieren.

Die Datei wird wie folgt vereinfacht: Finden des Bestandteils, Vereinfachung des Bestandteils, Vereinfachung des am Anfang fertiggestellten Bestandteils, Vereinfachung des Anfangs des Bestandteils, Vereinfachung des Bestandteils des Bestandteils, Vereinfachung des Endes der Bestandteils, Vereinfachung des am Anfang festgestellten Endes.