Selected Fun Problems of the ACM Programming Contest: Booby Traps

Noah Doersing
noah.doersing@student.uni-tuebingen.de

ABSTRACT

Diese Ausarbeitung beschreibt das *ACM Programming Contest*-Problem „Booby Traps“ und einige algorithmische Lösungsansätze, basierend auf verbreiteten Graphenalgorithmen. Ein vielversprechender Lösungsansatz wird genauer beleuchtet und detailliert beschrieben. Die Implementation dieses Lösungsansatzes in *Python* wird ebenfalls angeschnitten und die Komplexität der Lösung bewertet. Abschließend werden Erweiterungs- und Verbesserungsmöglichkeiten angeschnitten.

CCS Concepts

•**Theory of computation** → **Shortest paths;** *Dynamic graph algorithms;* Design and analysis of algorithms;

Keywords

booby traps; breadth-first search; dynamic graph, shortest path; dijkstra's algorithm; python

1. EINLEITUNG

Die *Association for Computing Machinery* (ACM) ist eine der ältesten wissenschaftlichen Gesellschaften für Informatik. Mit über 100.000 Mitgliedern ist sie außerdem die größte solche Vereinigung [6]. Im Rahmen der ACM findet Unterstützung verschiedenster Forschung im Bereich der Informatik statt. Eine der von der ACM unterstützten Initiativen ist der *ACM International Collegiate Programming Contest*, der jährlich weltweit in mehreren Runden ausgetragen wird [7] und bei dem Teams von Studierenden mehrere Programmierprobleme möglichst vollständig lösen.

Das hier beschriebene und gelöste Problem „Booby Traps“ wurde bei den *Regionals 2006* des *ACM Programming Contest* für die Region *Europe - Southwestern* gestellt [1].

2. PROBLEMBESCHREIBUNG

Das Problem besteht darin, fiktive Grabräuber dabei zu unterstützen, eine neue Art Grab zu durchsuchen. Darin soll die Länge des kürzesten Wegs von einem Start- zu einem Endpunkt ermittelt werden. Diese Gräber (im Folgenden: *Maps*) bestehen aus Feldern, die gitterartig angeordnet sind. Dabei existieren w Felder in x-Richtung und h Felder in y-Richtung, mit $w \cdot h \leq 40\,000$. Der Ursprung $(x, y) = (0, 0)$ liegt in der oberen linken Ecke der Map. Einige der Felder sind begehbar, während andere nicht betreten werden können. Die Grabräuber können sich nur horizontal oder vertikal, nicht aber diagonal bewegen.

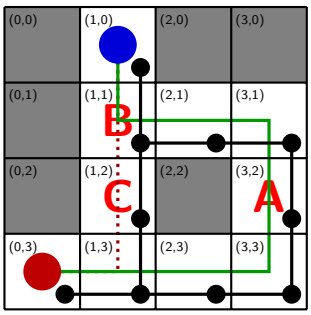


Figure 1: Eine simple Instanz des „Booby Traps“-Problems.

Was den Grabräubern das Leben erschwert, sind die sogenannten *booby traps*, also Fallen, die sich auf begehbaren Feldern der Map befinden. Wird eine Falle vom Typ α ausgelöst, indem das entsprechende Feld betreten wird, werden dabei auch alle Fallen ausgelöst, deren Typ in der mit der Eingabe übergebenen *trap domination order* $\leq \alpha$ ist. Es existieren genau 26 Fallentypen, jedoch können mehrere Fallen eines Typs in einer Map vertreten sein.

Je nachdem, welcher Pfad durch die Map eingeschlagen wird, verändert sich die Menge M der noch begehbaren Felder. Es handelt sich also gewissermaßen um einem dynamischen Graphen. Ein Beispiel für eine Instanz des Problems ist in Figure 1 zu sehen.

3. PROBLEMLÖSUNG

Nun werden die Schritte detailliert, die zur Lösung des Problems nötig sind. Verschiedene Lösungsansätze für den algorithmischen Kern werden erkundet, woraufhin ein vielversprechender Ansatz detaillierter beschrieben und optimiert wird. Abschließend wird dieser Algorithmus im Hinblick auf Komplexität und Laufzeit analysiert.

3.1 Parsing

Die Eingabe ist in der Problemstellung spezifiziert und sieht beispielsweise wie in Figure 2 aus.

Die erste Zeile enthält die *trap domination order*, wobei in Figure 2 `Z` $<$ `Y` $< \cdots <$ `A` gilt. In der zweiten Zeile werden Breite $w =$ `4` und Höhe $h =$ `4` der Map übergeben. Die nächsten h Zeilen enthalten eine ASCII-Darstellung der Map, wobei `o` begehbare Felder, `x` nicht begehbare Felder und ein Zeichen aus der *trap domination order* eine Falle kodiert. Zuletzt werden Start- und Endpunkt übergeben: In der vorletzten Zeile x- und y-Koordinate des Startpunkts

```
ZYXWVUTSRQPONMLKJIHGFEDCBA
4 4
xoxx
xBoo
xCxA
oooo
1 0
0 3
```

Figure 2: Eingabe, welche die Probleminstanz in Figure 1 beschreibt.

und in der letzten Zeile x- und y-Koordinate des Endpunkts.

Das Parsing der Eingabe gestaltet sich also recht einfach: die *trap domination order* lässt sich als String repräsentieren, die Map als String-Array und der Rest der Eingabe als Integer. Da die Breite w und Höhe h der Map bekannt sind, sobald die Map gelesen wird, muss hier nicht „geraten“ werden, wo die Map aufhört, bevor die letzten zwei Zeilen eingelesen wurden.

3.2 Adjazenzlistendarstellung

Bevor weitere Schritte unternommen werden können, ist es sinnvoll, die Map zu einem Graphen umzuwandeln. Die einfachste (und *most pythonic*) Repräsentation des Graphen ist eine Adjazenzliste. Dabei wird jedem Feld die Liste der direkten Nachbarn zugeordnet, was sich in der Python-Implementation mit einem `dict` realisieren lässt.

Zum Feld an den Koordinaten $F_1 = (x_1, y_1)$ sind die Felder an den Koordinaten (x_1+1, y_1), (x_1-1, y_1), (x_1, y_1+1) und (x_1, y_1-1) benachbart. Diese werden nur zur Adjazenzliste für F_1 hinzugefügt, wenn sie begehbar oder Fallen sind. Um nicht über die Ränder der Map hinwegzulaufen, müssen entsprechende Spezialfälle eingeführt werden.

3.3 Überlegungen zum algorithmischen Kern

Hier wird der Autor seine Überlegungen zum algorithmischen Kern der Lösung umranden. Zuerst werden dabei bekannte Graphenalgorithmen anhand ihrer Komplexität und Eignung für die Problemstellung beurteilt und zwei daraus folgende Ansätze genauer betrachtet, bevor die finale Lösung detailliert erklärt wird.

3.3.1 Übersicht

In dem Graphen, der im vorigen Schritt erstellt wurde, soll *gesucht* werden. Also bieten sich hier Tiefensuche und Breitensuche (beide in $\mathcal{O}(n+m)$ mit Knotenanzahl n und Kantenanzahl m) an.

Weil eine effizient zur Suche von kürzesten Pfaden verwendbare Tiefensuche mit einer Warteschlange statt eines Stacks implementiert werden würde (was äquivalent zur Breitensuche wäre) oder dabei (fast) alle möglichen Pfade jeder möglichen Länge im Graphen berechnet werden müssten, bevor der kürzeste Pfad mit Sicherheit feststeht, eignet sich Breitensuche besser für dieses Problem als Tiefensuche.

Tatsächlich soll jedoch ein *kürzester Pfad* im Graphen ermittelt werden, was sich mit dem Dijkstra-Algorithmus [2] (in $\mathcal{O}(n^2+m)$ bzw. mit Optimierungen in $\mathcal{O}(n \cdot \log n + m)$ [3]), dem A*-Algorithmus (in $\mathcal{O}(n^2)$ bzw. mit Optimierungen in $\mathcal{O}(n \cdot \log n)$ [4]) oder dem Bellman-Ford-Algorithmus ($\mathcal{O}(n \cdot m)$ [5]) lösen ließe.

Unter diesen drei bekannten Algorithmen ist der Dijkstra-Algorithmus der einfachste und am wenigsten komplexe. Die

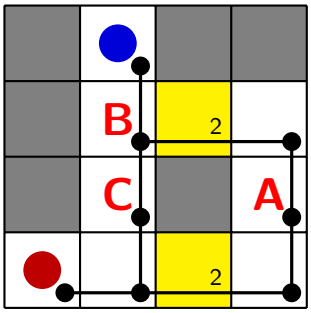


Figure 3: Durch Graph-Optimerung wurden die Knoten in den hervorgehobenen Feldern durch längere Kanten ersetzt.

Einschränkungen dieses Algorithmus, aufgrund derer die zwei anderen Algorithmen häufig verwendet werden, sind hier aufgrund der Gitterstruktur der Map nicht weiter relevant.

Weil hier der kürzeste Pfad in einem regelmäßigen Gitter gesucht werden soll (und dabei gemäß Problemstellung nur die Länge interessiert), könnte man auch mit Hamming-Distanzen ($\mathcal{O}(1)$) arbeiten. Dabei wird die Länge eines Pfades anhand der Unterschiede der x- und y-Werte von Start- und Endpunkt berechnet. Aufgrund der zur Behandlung von nicht begehbaren Feldern und der Fallen nötigen Spezialfälle wäre dieser Ansatz jedoch kaum erfolgsversprechend.

Im Folgenden wird also zuerst der Dijkstra-Algorithmus und dann Breitensuche im Hinblick auf Anwendbarkeit auf das „Booby Traps“-Problem betrachtet.

3.3.2 Dijkstra-Algorithmus

Der offensichtliche Nachteil des Dijkstra-Algorithmus im Vergleich zur Breitensuche ist die höhere Komplexität: der Dijkstra-Algorithmus liegt mit Optimierung bestenfalls in $\mathcal{O}(n \cdot \log n + m)$, Breitensuche jedoch in $\mathcal{O}(n+m)$. Darüberhinaus ist Breitensuche wesentlich simpler und entsprechend leichter erweiterbar.

Was dem Dijkstra-Algorithmus jedoch einen entscheidenden Vorteil verschafft, ist seine Fähigkeit, mit unterschiedlichen Kantengewichten umzugehen. Dies ermöglicht eine potenzielle Performance-Steigerung, wenn nach der Umwandlung der Map in einen Graphen noch ein weiterer Preprocessing-Schritt einführt wird.

Graph-Optimierung. Hierbei werden Knoten v, die keinen Fallen entsprechen und nur zwei Kanten e_1 (zu einem Knoten v_1) und e_2 (zu v_2) haben (mit Gewicht w_1, w_2), also lediglich eine Verbindung zwischen den zwei Knoten v_1, v_2 darstellen und bei denen keine Verzweigung möglich ist, durch eine einzelne Kante e zwischen v_1 und v_2 mit Gewicht $w = w_1 + w_2$ ersetzt. Dies wird für alle Knoten so lange wiederholt, bis sich keine Änderungen am Graphen mehr ergeben. Dies ist beispielhaft in Figure 3 gezeigt und kann prinzipiell auch auf zwei und mehr Dimensionen verallgemeinert werden.

Anpassbarkeit. Was sich jedoch als größeres Problem darstellt, ist die Anpassbarkeit des Dijkstra-Algorithmus an das „Booby Traps“-Problem. Vor allem ist dem Autor dabei keine gute (d.h. performante oder elegante) Möglichkeit eingefallen, die *trap domination order* zu berücksichtigen.

Zu Beispiel könnte der Dijkstra-Algorithmus „geforkt“ werden, wenn eine Falle bearbeitet wird. Dabei wird sozusagen eine neue Instanz des Algorithmus gestartet, die von

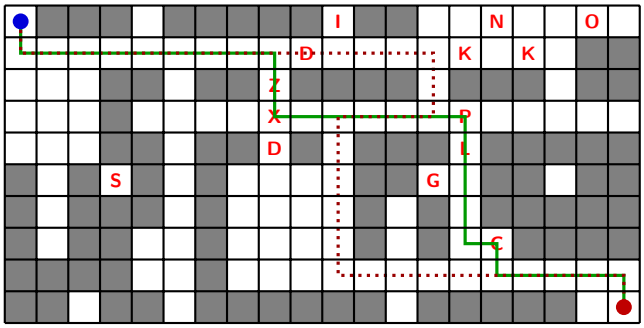


Figure 4: Inkorrektes Ergebnis einer frühen, Dijkstra-Algorithmus-basierten Version der Problemlösung.

jedem Nachbarn der Falle aus einen Pfad zum Endpunkt sucht, wobei die Distanzen zu allen nun ausgelösten Fallen auf ∞ gesetzt werden. Hier ist vor allem die Laufzeit ein Problem: Wenn $|T|$ die Anzahl der Fallen in der Map beschreibt, liegt die Laufzeit dieses modifizierten Dijkstra-Algorithmus in $\mathcal{O}((n \cdot \log n + m) \cdot 3^{|T|})$. Der Faktor $3^{|T|}$ kommt daher, dass jede Falle bis zu drei unbesuchte Nachbarn haben kann.

Zu Beginn der Arbeit an der Problemlösung wurde erfolglos verucht, eine etwas weniger optimierte Variante des Dijkstra-Algorithmus zu implementieren. Diese verwendete Backtracking, sobald ein nicht begehbares Feld erreicht wurde. Sie fand zwar immer einen Pfad vom Start- zum Endpunkt (wenn ein solcher Pfad existierte), jedoch war dieser Pfad nicht immer optimal. Ein Beispiel ist in Figure 4 zu sehen, dabei ist der rot gepunktete Pfad das Ergebnis der inkorrekten Dijkstra-Implementation und der grüne Pfad eine korrekte Lösung.

3.3.3 *Breitensuche*

Die Breitensuche ist wesentlich simpler als der Dijkstra-Algorithmus: Beginnend beim Startfeld werden dabei die unbesuchten Nachbarn des aktuellen Felds in eine Warteschlage eingefügt, als besucht markiert und nach und nach auf die gleiche Weise verarbeitet. Sobald der Endpunkt erreicht ist, wird „True“ zurückgegeben; wenn dies jedoch nie der Fall ist und die Warteschlange leer ist, gibt der Algorithmus „False“ zurück. In Pseudocode ausgedrückt lässt sich Breitensuche wie in Figure 5 definieren (der Aufruf `adj(c)` liefert die Menge der Nachbarn von `c`).

Algorithm 1: Breitensuche

```
1  q ← Queue(start)
2  v ← {start}
3  while not q.empty() do
4  |  c ← q.pop()
5  |  foreach n ∈ adj(c) do
6  |  |  if c ∉ v then
7  |  |  |  if c = end then
8  |  |  |  |  return True
9  |  |  |  q.push(n)
10 |  |  |  v ← v ∪ {n}
11 return False
```

Figure 5: Pseudocode-Algorithmus für Breitensuche.

Es gibt dabei verschiedene Möglichkeiten, die besuchten Felder zu repräsentieren: zum einen könnte ein `dict` verwendet werden, das Felder auf boolesche Werte abbildet, andererseits lässt sich stattdessen eine Menge von besuchten Feldern verwalten.

Ein klarer Vorteil der Breitensuche ist die Komplexität von $\mathcal{O}(n+m)$. Weil ein Feld aufgrund der Gitterstruktur der Map maximal vier Nachbarn besitzt, gilt $1 < m \leq 4n$. Auf einer Instanz des „Booby Traps“-Problems liegt die Komplexität folglich in $\mathcal{O}(n + 4n) = \mathcal{O}(n)$, sie ist also linear im Bezug auf die Anzahl der Felder in der Map.

3.4 Modifizierte Breitensuche

Aufgrund der geringeren Komplexität der Breitensuche wurde diese als Basis der Problemlösung gewählt. Um die Fallen sowie die *trap domination order* zu berücksichtigen, müssen jedoch einige Modifikationen durchgeführt werden.

3.4.1 *Warteschlange*

Die Problemstellung fordert die Ausgabe der Länge des kürzesten Pfades. Diese muss also für jedes Feld in der Warteschlage gespeichert werden, damit sie zurückgegeben werden kann, falls nach dem Entnehmen des Felds aus der Warteschlange während der Verarbeitung dessen Nachbarn das Ende erreicht wird. Tatsächlich wird in der Implementation der gesamte bisherige Pfad gespeichert, um eine Visualisierung des Ergebnisses des Algorithmus zu ermöglichen.

Darüberhinaus macht es Sinn, die maximale bisher ausgelöste Falle α ebenfalls in der Warteschlage zu speichern, um schnell prüfen zu können, ob ein Feld, das eine Falle α' enthält, besucht werden kann. Ein Beispiel für die auf diese Art modifizierte Warteschlagen ist in Figure 6 zu sehen.

Bevor ein Feld zur Warteschlange hinzugefügt werden kann (siehe Zeile 9 des Pseudocode-Algorithmus), muss geprüft werden, ob es überhaupt noch besucht werden kann. Sei α die maximale bisher ausgelöste Falle. Dann kann für jedes Feld einer von vier Fällen auftreten:

1. Das Feld wurde bereits besucht. Dann muss es nicht nochmals besucht werden, wird also nicht zur Warteschlange hinzugefügt.

2. Das Feld wurde nicht nicht besucht und es enthält keine Falle. Dann kann es unabhängig von α besucht werden, wird also zur Warteschlange hinzugefügt und als besucht markiert.

3. Das Feld wurde noch nicht besucht und enthält eine Falle α', die in der *trap domination order* $\leq \alpha$ ist. Das heißt, dass α' bereits ausgelöst wurde, das Feld kann also nicht besucht werden und wird folglich nicht zur Warteschlange hinzugefügt.

4. Das Feld wurde noch nicht besucht und enthält eine Falle α', die in der *trap domination order* $> \alpha$ ist.

$$
\begin{aligned}
q = [&((1,1),[\ldots,(1,2),(1,1)],A),\\
&((6,5),[\ldots,(5,5),(6,5)],B),\\
&((1,6),[\ldots,(1,5),(1,6)],A),\ldots]
\end{aligned}
$$

Figure 6: Beispiel: Warteschlange mit zu besuchendem Feld, bisherigem Pfad und maximaler ausgelöster Falle.

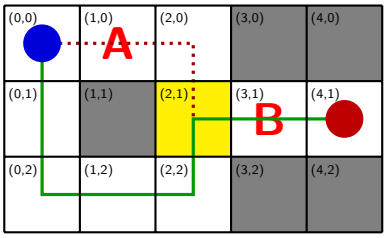


Figure 7: Zwei verschiedene Pfade durch $(2, 1)$ machen es unmöglich, eine einzelne Menge v für die Speicherung der besuchten Felder zu nutzen.

Also wurde α' bisher noch nicht ausgelöst, das Feld kann also als besucht markiert werden und mit der neuen maximalen bisher ausgelösten Falle α' zur Warteschlange hinzugefügt werden.

3.4.2 *Repräsentation der besuchten Felder*

Aufgrund der Fallen ist es nicht möglich, wie im oben angegebenen Pseudocode eine einzelne Menge v zu verwenden, um zu speichern, welche Felder bereits besucht wurden.

Dies lässt sich am Beispiel in Figure 7 veranschaulichen: Im rot gepunkteten Pfad wird im ersten Schritt die Falle A ausgelöst (dabei gilt $B < A$) und in zwei weiteren Schritten das Feld mit (x, y)-Koordinaten $(2, 1)$ besucht. Der einzige Nachbar, der zum Ziel führen könnte, enthält jedoch die nun bereits ausgelöste Falle B. Dieser Pfad wird also in eine Sackgasse führen.

Möglich ist jedoch der grüne Pfad, wobei jedoch fünf Schritte benötigt werden, um das Feld $(2, 1)$ zu erreichen – würde hier ein v wie im obigen Pseudocode verwendet, wäre dieses bereits als besucht markiert worden und der Weg zum Endpunkt somit abgeschnitten.

Naive Lösung. Eine unkomplizierte Art, die besuchten Felder zu repräsentieren, ist, lediglich die Felder im Pfad zum jeweilig aktuellen Feld als besucht zu betrachten. Der Pfad wird ohnehin mit dem Feld in der Warteschlange gespeichert, man benötigt außer dieser also keine weiteren Datenstrukturen – v fällt hier weg. Diese Variante hat also eine gewisse Eleganz. Der entscheidende Nachteil ist jedoch die Laufzeit, die sich in Benchmarks lediglich für Pfadlängen von weniger als 25 Feldern als akzeptabel erwiesen hat. Der Grund ist, dass hier alle Felder, die mit einem Pfad der minimalen Pfadlänge (vom Start- zum Endpunkt) vom Startpunkt aus erreichbar sind, in jeder vom Startpunkt aus möglichen Reihenfolge besucht werden. Theoretisch ergibt sich also eine Komplexität in $\mathcal{O}(3^n)$[1].

Mehrere Mengen. Wesentlich effizienter ist es, für jeden Fallentyp $\in T = \{A, B, \ldots, Z\}$ eine Menge $v_A, v_B, \ldots, v_Z$ zu verwenden und beim Besuchen von Feldern genau wie bei der Standard-Breitensuche mit nur einer dieser Mengen zu arbeiten, nämlich dem v_α der maximalen bisher ausgelösten Falle α. Dabei muss zunächst eine Menge v_0 genutzt werden, solange auf dem jeweiligen Pfad noch keine Falle ausgelöst wurde. Mit diesem Ansatz verringert sich die Anzahl der Besuche eines Felds von „Anzahl aller möglichen Pfade vom Start aus durch dieses Feld“ auf $|T| + 1 = 26 + 1 = 27$ (oder niedriger, wenn nicht jedes $t \in T$ in der Map vorkommt oder das Ende früh erreicht wird). Die hierbei resultierenden

[1]Nicht, wie fälschlicherweise im Vortrag erwähnt, in $\mathcal{O}(n!)$.

$$
\begin{aligned}
v_0 &= \{(1, 0)\} \\
v_A &= \{(3, 2)\} \\
v_B &= \{(1, 1), (2, 1), (1, 0), (3, 1)\} \\
v_C &= \varnothing \\
q &= [(\ldots, [\ldots], \alpha]
\end{aligned}
$$

Figure 8: Beispiel: Zustand der Datenstrukturen an einem Punkt der Ausführung der finalen Implementation.

Datenstrukturen sind examplarisch in Figure 8 gezeigt.

Diese Lösung funktioniert, da hier für jeden Fallentyp (und nicht wie in der Standard-Breitensuche ganz allgemein, was zu *false positives* führt) gespeichert wird, welche Felder bereits besucht wurden. Jedoch treten hier *false negatives* auf: Felder, die im bisherigen Pfad liegen, gelten nicht mehr als besucht markiert, sobald eine neue Falle ausgelöst und damit zur maximalen bisher ausgelösten Falle α wird, in deren v_α zukünftig zu arbeiten ist. Tatsächlich gilt hier direkt nach dem erstmaligen Besuch einer neuen Falle die gesamte Map als unbesucht[2].

Redundante Besuche vermeiden. Es wurde eine weitere Optimierungsmöglichkeit identifiziert, die die Laufzeit der Implementation auf großen Maps mit hoher Fallenanzahl um einen Faktor > 2 verringern kann. Dabei wird auf der vorherigen Lösung aufgebaut und lediglich auf eine andere Weise geprüft, ob ein Feld besucht werden kann. Anstatt nur in v_α zu prüfen, wird zunächst eine Menge T' aller Fallen erzeugt, die auf dem bisherigen Pfad liegen. Geprüft wird nun in $\bigcup_{t \in T'} v_t$[3].

Das löst das *false negatives*-Problem der vorherigen Lösung. Die Korrektheit dieser Erweiterung ist gegeben, da kürzeste Pfade, die durch Fallen führen, die sich auf dem Pfad zum aktuellen Feld befinden, sich immer höchstens so schnell „ausbreiten“ wie der Pfad durch das aktuelle Feld. Das bedeutet, dass im aktuellen Feld nicht „zurückgeschaut“ werden muss, weil mögliche Pfade zum Ziel, die durch die bereits besuchten Felder führen, bereits gesucht werden.

3.5 Komplexität

In dieser Sektion wird gezeigt, dass die oben beschriebene modifizierte Breitensuche keine höhere asymptotische Komplexität besitzt als die „normale“ Breitensuche.

Auf einem Graphen $G = (V, E)$ mit Knotenanzahl $n = |V|$ und Kantenanzahl $m = |E|$ hat die unmodifizierte Breitensuche eine Komplexität von $\mathcal{O}(n + m)$.

Die Laufzeit der modifizierten Breitensuche liegt jedoch in $\mathcal{O}((n + m) \cdot (1 + |T''|))$, weil Felder aufgrund des Vorhandenseins von Fallen mehrfach besucht werden können. Dabei beschreibt $|T''|$ die Anzahl der paarweise verschiedenen Fallen bzw. die Anzahl der Fallentypen in der Map. Zu diesem Wert wird 1 hinzugefügt, da es in aller Regel auch

[2]Ein Beispiel hierfür hat in dieser Ausarbeitung nicht genügend Platz, findet sich jedoch auf den Folien 31-43 des Vortrags und kann darüberhinaus für beliebige Maps erzeugt werden, wenn in `boobytraps-latex.py` Zeile 332 aus- und Zeile 333 einkommentiert wird.

[3]Ein Beispiel hierfür findet sich auf den Folien 45-54 des Vortrags und kann darüberhinaus mit `boobytraps-latex.py` für beliebige Maps erzeugt werden.

möglich ist, vom Startpunkt aus einige Felder zu besuchen, ohne vorher eine Falle auszulösen.

Nun lassen sich die möglichen Wertebereiche von $|T''|$ und m bestimmen: Es gilt $0 \leq |T''| \leq |T| = 26$ (da es in einer beliebigen Map maximal 26 Fallen von verschiedenen Typen gibt, jedoch auch Maps möglich sind, die keine Fallen enthalten) und $0 \leq m \leq 4n$ (da ein Knoten aufgrund der Gitterstruktur der Map maximal 4 Kanten hat).

Wenn man nun diese Obergrenzen in die Komplexität einsetzt, ergibt sich

$$\begin{aligned} & \mathcal{O}((n+m) \cdot (1+|T''|)) \\ &= \mathcal{O}((n+4n) \cdot (1+26)) \\ &= \mathcal{O}(5n \cdot 27) \\ &= \mathcal{O}(135n) \\ &= \mathcal{O}(n) = \mathcal{O}(w \cdot h), \end{aligned}$$

wobei w und h die Breite bzw. Höhe der Map sind. Auf quadratischen Maps (was der häufigste *use case* sein könnte) mit $w = h = l$ ergibt sich also eine Komplexität in $\mathcal{O}(l^2)$, d.h. quadratisch bezüglich der Seitenlänge der Map.

4. IMPLEMENTATION

4.1 Programmiersprache

Im Folgenden wird die Wahl von Python als Programmiersprache für die Implementation begründet.

Ein entscheidender Vorteil ist, dass die Syntax im allgemeinen simpler und intuitiver ist als bei vergleichbaren (d.h. imperativen) Programmiersprachen. Die ermöglicht ein schnelles Entwerfen von Funktionen und Klassen und geht Hand in Hand mit der Tatsache, dass Python schwach und dynamisch typisiert ist, was die schnelle Entwicklung von Entwürfen zusätzlich begünstigt (jedoch auf Dauer zu weniger stabiler Software führen kann).

Was ebenfalls hilfreich war, ist die von Haus aus integrierte Unterstützung für Listen, Mengen (`set`) und vor allem Dictionaries (`dict`). Diese Datenstrukturen sind in der Python-Standardbibliothek auch recht performant implementiert. Darüberhinaus empfindet der Autor die Möglichkeit, eine Funktion mehrere verschiedene Werte zurückgeben zu lassen, ohne sie vorher in ein Tupel oder eine Liste einpacken zu müssen, als sehr angenehm.

Ein großer Nachteil im Vergleich zu kompilierenden Sprachen wie C oder Haskell ist natürlich die Performance: eine interpretierte Sprache wie Python kann nicht schneller sein als eine kompilierte.

Ich habe für die Implementation aus zwei Gründen eine imperative und keine funktionale Sprache gewählt: Zum Einen war mir zu Beginn des Semesters keine funktionale Sprache besonders vertraut[4]. Zum anderen profitieren die von der Breitensuche verwendeten Datenstrukturen (Warteschlangen sowie Mengen, auf die über den gesamten Programmablauf hinweg schreibend und lesen zugegriffen wird) von Nebenläufigkeit. Dieses Konzept ist in imperativen Sprachen eher zu Hause als in funktionalen Sprachen.

4.2 Weiterer Code

Zusätzlich zur Implementation der hier beschriebenen erweitereten Breitensuche wurden im Verlauf des Seminarprojekts auch weitere Programme geschrieben, welche hautsächlich zur Visualisierung und einfacheren Testbarkeit dienen.

4.2.1 Map Generator

Um die Performace der Implementation auf großen Maps testen zu können, wurde ein simpler Map Generator geschrieben. Dieser unterstützt zwei Modi: Wenn im Aufruf `-mode random` übergeben wird, werden auf einer zunächst komplett nicht begehbaren Map zufällig begehbare Felder und Fallen gesetzt. Dahingegen werden im `dungeon`-Modus begehbare Linien, Rechtecken und Fallen „gezeichnet“, was zu unterschiedlichen Eigenschaften führt (siehe Benchmarks in Figure 9).

Wie viele begehbare Felder und Fallen erzeugt werden, lässt sich mit der `-complexity`-Flag steuern. Eine vollständige Liste der möglichen Optionen lässt sich mittels `gravedigger.py -h` anzeigen. Einige interessante Beispiele wurden in `README.md` zusammengetragen.

4.2.2 Pretty Printing

Wird `boobytraps.py` mit der `-v`-Flag aufgerufen, so wird zusätzlich zur Länge des kürzesten Pfades auch eine ASCII-Darstellung der Map ausgegeben. Dabei werden Start- und Endpunkt, (nicht-)begehbare Felder, Fallen und der Pfad farblich hervorgehoben. Bei Verwendung der `-v2`-Flag werden zusätzlich alle besuchten Felder eingefärbt.

4.2.3 LaTeXCode Generator

Um Beispiele sowohl in den Folien als auch in dieser Ausarbeitung zu erzeugen, wurde ein Python-Skript mit einer modifizierten Variante des Algorithmus geschrieben, die in jedem Schritt den LaTeX-Quellcode für eine *Beamer*-Folie mit einer Visualisierung der Map, des Pfades und des Zustands der Datenstrukturen ausgibt.

Dieses Skript unterstützt darüberhinaus die Ausgabe von LaTeX-Code für eine Visualisierung der Map, des dazugehörigen Graphen und des kürzesten Pfades.

4.3 Benchmarks

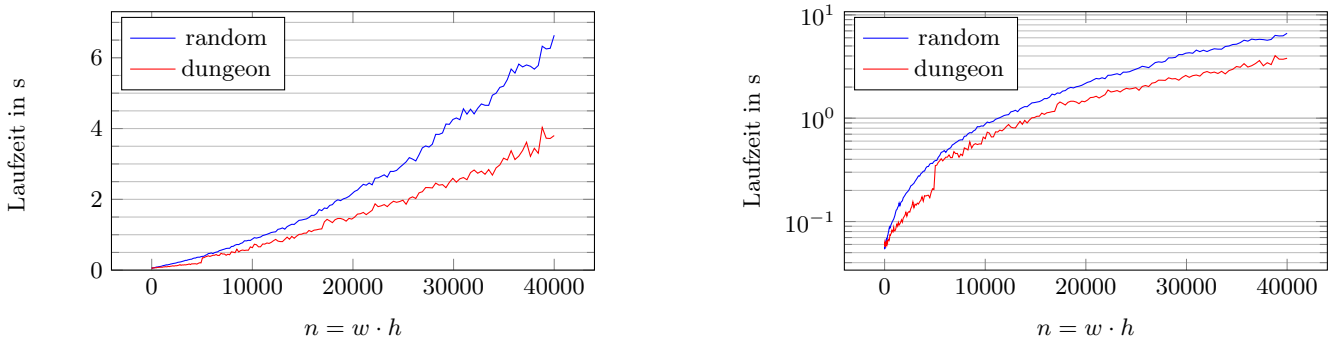
Um die tatsächliche Laufzeit der Implementation zu bestimmen, wurde ein Bash-Script geschrieben, das für jedes $l \in \{1, \ldots, 200\}$ (da laut Problemstellung $l^2 \leq 40\,000 = 200^2$ gilt) eine zufällige Map generiert und darauf die modifizierte Breitensuche ausführt. Dabei werden je 1000 Samples verwendet. Das Ergebnis dieses einfachen Benchmarks ist in Figure 9 zu sehen und bestätigt die Annahme, dass die Laufzeit grob linear im Bezug auf die Anzahl der Felder n ist.

Die dabei sichtbare Nichtlinearität (d.h. das leichte Wachstum der Steigung) führt der Autor auf die in der Python-Implementation verwendeten Datenstrukturen zurück, deren Laufzeiten teilweise in $\mathcal{O}(n \log n)$ liegen, und darauf, dass die in der Komplexitätsbetrachtung eingesetzten Obergrenzen bei kleineren Map-Größen im Durchschnitt nicht erreicht werden.

5. FUTURE WORK

Im Verlauf der Arbeit an der Implementation, des Seminarvortrags und dieser Ausarbeitung sind einige Punkte offensichtlich geworden, die (wenn auch nicht strikt notwendig) zur Verbesserung der Lösung beitragen könnten bzw. interessante Fragestellungen darstellen. Einige davon sind als „TODO“s für eine verbesserte Variante zu verstehen, andere lediglich als interessante Fragestellungen.

[4]Diesen Bug hat die Vorlesung „Funktionale Programmierung“ jedoch schnell gefixt.



(a) Lineare Skala

(b) $\log_{10}$-Skala

Figure 9: Benchmarks (für $1 \leq \sqrt{n} = w = h \leq 200$ jeweils 1000 Samples).

- Teile der modifizierten Breitensuche können parallel ausgeführt werden, was die Laufzeit auf Multicore-Prozessoren oder Clustern verringern würde. Intitiv sieht der Autor dabei jedoch keine signifikanten Performance-Steigerungen.

- Anstelle einer Standard-Warteschlange könnte man eine *Priority Queue* verwenden, wobei die Position eines Elements von der euklidischen Distanz vom Start und/oder zum Ende abhängt.

- Das Problem und die hier präsentierte Lösung lässt sich fast trivial auf mehr Dimensionen generalisieren, jedoch würde dabei recht schnell eine Performace-Schranke erreicht werden. Bezüglich der Felder wäre die Laufzeit jedoch immer noch linear und bezüglich der (Seiten-)Länge l einer Linie (1D), eines Quadrats (2D), eines Würfels (3D), eines *Hypercube* (4D) immer noch polynomiell (nämlich $l^{\text{Dimension}}$).

- Das *domination ordering* der Fallen legt nahe, ein *domination ordering* auf v_i zu definieren. Eine dazu polymorphe Variante wurde implementiert, dies lässt sich aber sicher eleganter (und vor allem performanter) lösen.

- Die angegebene Implementation lässt sich noch optimieren, z.B. durch Verwendung einer nicht-interpretierten Programmiersprache.

- Wenn diagonale Bewegungen in der Map auch zugelassen wären, könnte man keine einfache Breitensuche mehr verwenden. Wie sähe dann eine elegante und performante Lösung aus?

- Die Benchmarks sind sehr simpel gehalten. Benchmarks auf einer größeren Vielfalt von Map-Typen und nichtquadratischen Maps wären sicher auch hilfreich (und würden sich auch auf erweiterte Lösungen des Problems anwenden lassen). Außerdem: Benchmarks für $n > 200$ wären interessant, um die weitere Entwicklung der Laufzeit zu betrachten. Dabei müsste jedoch auch der Map Generator entsprechend angepasst werden.

- Sobald der Endpunkt erreicht ist, liegt es nahe, die aktuelle „Runde“, d.h. alle Pfade mit der aktuellen Pfadlänge, fertig zu berechnen und – falls für die gleiche Pfadlänge mehrere Pfade zum Endpunkt gefunden wurden – den „besten“ auszuwählen. Dies könnte z.B. der Pfad mit den wenigsten Fallen, den meisten Fallen, oder der geringsten/höchsten Anzahl von Richtungswechseln sein. Den damit verbundenen Rechenaufwand schätzt der Autor als gering ein.

- Aktuell wird eine gültige Eingabe vorausgesetzt. Wenn eine ungültige Eingabe erfolgt, ergeben sich meist kryptischen Python-Fehlermeldungen. Das ließe sich durch Input-Validition während des Parsing-Schritts vermeiden.

6. REFERENCES

[1] ACM Programming Contest. Problem G: Booby Traps. http://db.inf.uni-tuebingen.de/staticfiles/teaching/ws0809/fun-problems/Booby-Traps.pdf, 2006 (accessed February 14, 2016).

[2] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[3] Wikipedia-Autoren. Dijkstra-Algorithmus. https://de.wikipedia.org/wiki/Dijkstra-Algorithmus, 2002-2016 (accessed February 27, 2016).

[4] Wikipedia-Autoren. A*-Algorithmus. https://de.wikipedia.org/wiki/A*-Algorithmus, 2004-2016 (accessed February 27, 2016).

[5] Wikipedia-Autoren. Bellman-Ford-Algorithmus. https://de.wikipedia.org/wiki/Bellman-Ford-Algorithmus, 2004-2016 (accessed February 27, 2016).

[6] Wikipedia editors. Association for Computing Machinery. https://en.wikipedia.org/wiki/Association_for_Computing_Machinery, 2001-2016 (accessed February 27, 2016).

[7] Wikipedia editors. ACM International Collegiate Programming Contest. https://en.wikipedia.org/wiki/ACM_International_Collegiate_Programming_Contest, 2004-2016 (accessed February 27, 2016).

7. BEMERKUNG

Der Quellcode der Implementation steht unter der MIT-Lizenz und ist auf GitHub verfügbar: https://github.com/doersino/acm-boobytraps.