Torkelnde Yamyams

Aufgabe 3, Runde 2, 34. Bundeswettbewerb Informatik

Marian Dietz

1 Lösungsidee

1.1 Beschreibung der Welt

Eine *Welt* mit n Zeilen und m Spalten besteht aus $s = n \cdot m$ quadratischen *Hindernissen*, *Ausgängen* und *Feldern*. Sie werden alle *Quadrate* genannt. Ein Quadrat hat die Koordinaten (x, y), ein geordnetes Paar, wobei x die Zeile und y die Spalte (0-basiert) ist. Ein Hindernis blockiert den Yamyam, d. h., er kann sich dort nicht befinden. Befindet sich ein Yamyam auf einem Ausgang, dann bewegt sich der Yamyam nicht weiter. Auf Feldern bewegt sich der Yamyam. Wenn er auf einem Feld steht, dann entscheidet er sich zufällig für eine der maximal vier Himmelsrichtungen, in die er weiterfährt. Er kann nur in eine Himmelsrichtung fahren, wenn das nächste Feld in dieser Richtung überhaupt existiert und es ein Ausgang oder ein Feld ist, denn außerhalb der Welt oder auf Hindernissen darf sich der Yamyam nicht befinden. Fährt ein Yamyam von einem Feld auf ein anderes Feld (kein Ausgang), existiert das nächste Quadrat in derselben Richtung und ist dies kein Hindernis, dann fährt er weiter in diese Richtung auf das nächste Feld bzw. den Ausgang. Kann der Yamyam in keine der vier Himmelsrichtungen fahren, weil alle davon durch Hindernisse bzw. das Ende der Welt blockiert sind, dann bleibt er einfach auf seinem Feld stehen und fährt nicht weiter. Dementsprechend kann er dann auch nie einen Ausgang erreichen.

Ein Beispiel für eine solche Welt, anhand dessen auch die Algorithmen erklärt werden, befindet sich in Abbildung 1.

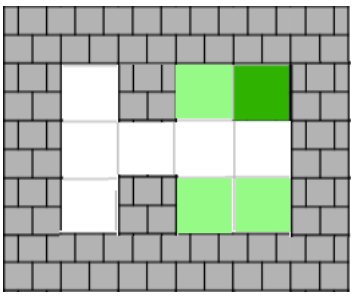


Abbildung 1: Beispielwelt

1.2 Darstellung als Graph

Es gibt zwei mögliche Zustände für den Yamyam, wenn er sich auf einem Feld befindet:

1. Er sucht sich eine zufällige neue Himmelsrichtung aus.
2. Er fährt weiter in die Richtung, in die er zuvor schon gefahren ist.

Dabei kann der Yamyam sich natürlich nicht aussuchen, in welchem Zustand er sich befindet. Fährt er erst einmal in eine Richtung, dann bleibt er auf allen weiteren Feldern im zweiten Zustand, bis er vor einem Hindernis bzw. dem Ende der Welt steht oder sich auf einem Ausgang befindet. Auf dem Feld, auf welchem er ankommt, befindet er sich wieder im ersten Zustand, da er danach wieder in eine zufällige neue Himmelsrichtung fährt. Kommt er hingegen auf einem Ausgang an, dann befindet er sich in keinem der beiden Zuständen, da er sich auf gar keinem Feld befindet.

Die Welt des Yamyams wird als gerichteter und gewichteter Graph $G = (V, E)$ mit der Menge der Knoten V und der Menge der Kanten E dargestellt. Einer der Knoten $u = (x, y)$, $u \in V$, $x, y \in \mathbb{N}$, $0 \leq x < n$, $0 \leq y < m$ ist der erste Zustand des Yamyams, bei dem dieser sich auf dem Feld in der Zeile x und der Spalte y befindet. Wenn der Knoten die Koordinaten eines Ausganges besitzt, dann stellt er einen Ausgang dar und der Yamyam befindet sich in keinem der beiden Zustände. Eine Kante $e \in E$ mit $e = (u, v, w)$ bedeutet, dass der Yamyam von Knoten u zum Knoten v gehen kann, wobei er sich auf dem gesamten Weg (inklusive u und v) auf $w + 1$ Quadraten befindet und dazwischen (ohne u und v) nie eine neue zufällige Himmelsrichtung auswählt, sondern nur über Felder im zweiten Zustand, also immer in dieselbe Richtung fährt. Es sind insgesamt w Schritte, die der Yamyam dabei macht. Da v ebenfalls ein Knoten ist, befindet sich der Yamyam dort wieder im ersten Zustand und wählt eine neue zufällige Richtung aus. Daher muss das nächste Feld hinter v in der Himmelsrichtung von u nach v ein Hindernis bzw. das Ende der Welt sein. Es ist aber auch möglich, dass v ein Ausgang ist. Dann hat der Knoten keine ausgehende Kante, denn von dort aus läuft der Yamyam nicht weiter. Hindernisse sind keine Knoten in G.

Für eine Kante $e \in E$, wobei $e = (u, v, w)$, $u = (x_u, y_u)$, $v = (x_v, y_v)$, gilt folgendes:

1. Wenn $x_u = x_v$ und $y_u < y_v$, dann müssen alle Quadrate mit den Koordinaten (x_u, y), $y_u < y < y_v$ ebenfalls Felder sein, denn ansonsten müsste der Yamyam auf dem Weg irgendwann den ersten Zustand annehmen. Außerdem muss gelten, dass entweder das Quadrat (x_v, y_v) ein Ausgang ist, dass $y_v = m - 1$ oder dass $y_v < m - 1$ und $(x_v, y_v + 1)$ ein Hindernis ist. Würde keine dieser Bedingungen erfüllt sein, dann dürfte der Yamyam nicht auf dem Quadrat (x_v, y_v) eine zufällige Richtung auswählen, sondern müsste weiter in die positive y-Richtung fahren.
2. Analog zu Fall 1 gilt bei $x_u = x_v$ und $y_u > y_v$, dass alle Quadrate mit den Koordinaten (x_u, y), $y_v < y < y_u$ Felder sind. Außerdem muss eine der folgenden Bedingungen erfüllt sein: (x_v, y_v) ist ein Ausgang, $y_v = 0$ oder $y_v > 0$ und $(x_v, y_v - 1)$ ist ein Hindernis.

3. Ist $y_u = y_v$ und $x_u < x_v$, dann sind alle Quadrate mit den Koordinaten (y_u, x), $x_u < x < x_v$ Felder. Eine der folgenden Bedingungen muss erfüllt sein: (x_v, y_v) ist ein Ausgang, $x_v = n - 1$ oder $x_v < n - 1$ und $(x_v + 1, y_v)$ ist ein Hindernis.

4. Ist $y_u = y_v$ und $x_u > x_v$, dann sind alle Quadrate mit den Koordinaten (y_u, x), $x_v < x < x_u$ Felder. Eine der folgenden Bedingungen muss erfüllt sein: (x_v, y_v) ist ein Ausgang, $x_v = 0$ oder $x_v > 0$ und $(x_v - 1, y_v)$ ist ein Hindernis.

Ein Beispiel eines solchen Graphen findet sich von Beispielwelt 1 in Abbildung 2.

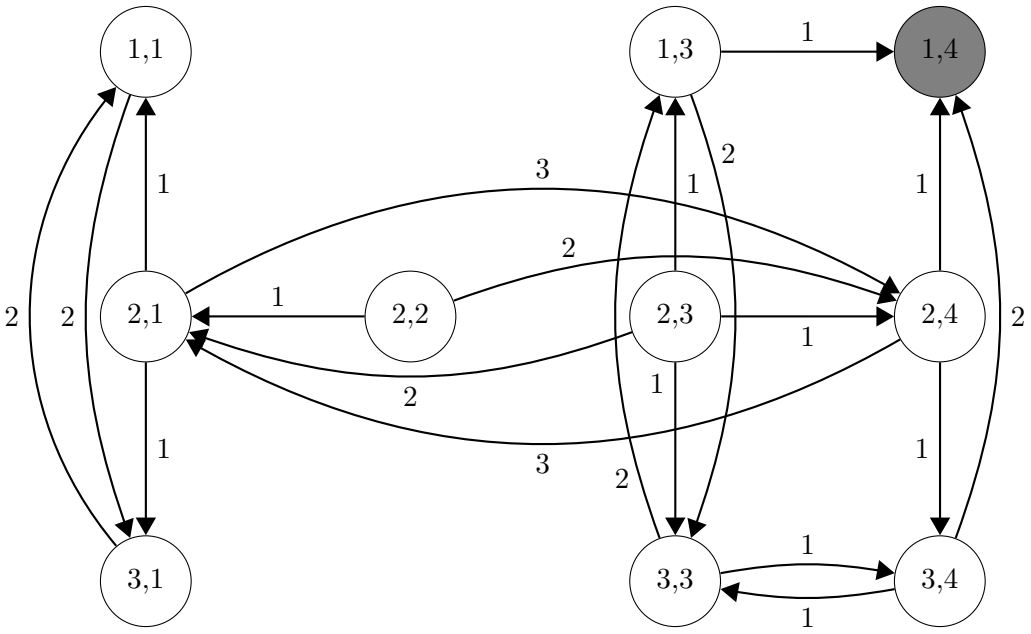


Abbildung 2: Die Beispielwelt aus Abbildung 1 als Graph. Der graue Knoten ist ein Ausgang. In den Knoten werden die x- und die y-Koordinate angezeigt. An jeder Kante steht ihre Länge.

1.3 Erstellen des Graphen

Im ersten Teil muss der Algorithmus diesen Graphen für eine gegebene Welt erstellen.

Zunächst wird für jedes Feld und für jeden Ausgang ein Knoten mit den entsprechenden Koordinaten zum Graphen G hinzugefügt.

Jeder Knoten kann maximal vier ausgehende Kanten haben. Dies liegt daran, dass der Yamyam höchstens in 4 verschiedene Himmelsrichtungen fahren kann. Egal, wann er in eine Richtung fährt, er wird immer auf demselben Quadrat einen Ausgang erreichen bzw. wieder in den ersten Zustand wechseln, da sich die Welt nicht verändert.

Zunächst werden alle Kanten der Knoten in Zeile 0 hinzugefügt, die nach links zeigen. Dafür muss die Zeile nur einmal von links nach rechts durchgegangen werden. Wenn ein Feld besucht wird, dann muss evtl. eine Kante hinzugefügt werden. Das Ziel der

Kante ist der Knoten, der nach den Bedingungen aus Abschnitt 1.2 links vom aktuellen Feld liegen muss. Wenn links vom Knoten ein Hindernis oder das Ende der Welt ist, dann bekommt der Knoten keine Kante nach links. Ansonsten ist das Ziel der Kante der erste Ausgang oder das erste Feld, neben dem links ein Hindernis ist. Daher können alle Kanten nach links in einer Zeile hinzugefügt werden, indem das erste Quadrat, auf dem der Yamyam die Richtung ändern muss bzw. einen Ausgang erreicht, immer gespeichert und laufend aktualisiert wird. Was genau passiert, wenn ein Quadrat abgelaufen wird, wird nach den folgenden Kriterien entschieden:

1. Ist das Quadrat ein Hindernis, dann hat das nächste Quadrat rechts davon keine Kante nach links. Das gespeicherte Quadrat muss zurückgesetzt werden. Da ein Hindernis durch keinen Knoten repräsentiert wird, hat es auch selbst keine ausgehende Kante.

2. Ist das Quadrat ein Ausgang, dann werden, falls vorhanden, die nächsten Felder rechts davon eine Kante nach links haben, wobei dieses Quadrat das Ziel davon ist. Dieser Ausgang wird also das gespeicherte Quadrat. Da ein Ausgang zwar ein Knoten ist, aber keine ausgehenden Kanten besitzt, muss hier keine Kante hinzugefügt werden.

3. Ist das Quadrat ein Feld, dann muss eine Kante mit dem gespeicherten Feld, wenn vorhanden, als Ziel hinzugefügt werden. Dieses Feld kann auch ein Ziel einer Kante nach links von einem Feld, das sich weiter rechts befindet, sein. Dies ist dann der Fall, wenn links vom aktuellen Feld ein Hindernis ist, da der Yamyam davor eine neue Richtung aussuchen muss. Daher muss in diesem Fall das aktuelle Feld zum gespeicherten Quadrat werden.

Nachdem die erste Zeile von links nach rechts abgelaufen wurde, müssen auch in allen anderen Zeilen noch die Kanten nach links erstellt werden. Analog wird das mit allen Zeilen von rechts nach links gemacht, um die Kanten nach rechts zu erstellen. Danach müssen noch die Kanten nach oben und nach unten hinzugefügt werden. Dafür wird jede Spalte einmal von unten nach oben und einmal von oben nach unten abgelaufen.

Wenn bspw. die Kanten für Zeile 3, vorwärts, in der Beispielwelt aus Abbildung 1 hinzugefügt werden sollen, dann wird zunächst ein Hindernis gefunden, weshalb keine Kante hinzugefügt und kein Knoten gespeichert wird. Danach wird ein Feld besucht, welches gespeichert wird, aber keine ausgehende Kante nach Westen bekommt. Das gespeicherte Feld wird durch das nächste Hindernis wieder zurückgesetzt. Dann wird das Feld $(3, 3)$ besucht, welches gespeichert wird, aber auch keine ausgehende Kante nach Westen bekommt. Erst dem Knoten des Feldes $(3, 4)$ wird eine Kante Richtung Westen hinzugefügt, sie endet auf dem gespeicherten Knoten $(3, 3)$.

1.3.1 Laufzeitanalyse

Wenn eine Zeile bzw. eine Spalte abgelaufen wird, wird für jedes Quadrat nur konstante Zeit benötigt, angenommen das Hinzufügen von Kanten kann ebenfalls in $\mathcal{O}(1)$ geschehen. Wenn eine gesamte Zeile abgelaufen wird, wird demnach die Zeit $\mathcal{O}(m)$ benötigt, für

eine Spalte $\mathcal{O}(n)$. Jede Spalte wird zweimal abgelaufen, genauso wie jede Zeile. Dadurch ändert sich nichts an der Zeitkomplexität. Es gibt insgesamt n Zeilen, die abgelaufen werden müssen, für alle Zeilen wird also die Zeit $\mathcal{O}(m \cdot n)$ benötigt, für alle Spalten wird ebenfalls insgesamt die Zeit $\mathcal{O}(m \cdot n)$ benötigt. Für diesen gesamten Algorithmus ist daher die benötigte Zeit $\mathcal{O}(m \cdot n) = \mathcal{O}(s)$.

1.4 Aufgabe

Es sollen alle Knoten, die sicher sind und Felder repräsentieren, gefunden werden. Ein Knoten ist sicher, wenn die Wahrscheinlichkeit, einen Ausgang zu erreichen (egal wie viele Schritte das benötigt), 1 ist. Diese Wahrscheinlichkeit eines Knotens $u \in V$ wird als $P(u)$ bezeichnet. $P((x, y))$ wird mit $P(x, y)$ abgekürzt. E_u ist die Menge aller Knoten, zu denen es eine direkte Kante von u aus gibt. Sie beinhaltet also alle Knoten v aus V, wobei es eine Kante $e \in E$, $e = (u, v, w)$ gibt und w beliebig sein kann.

Die Wahrscheinlichkeit eines Knotens, der einen Ausgang repräsentiert, ist immer 1, denn wenn der Yamyam dort ankommt, erreicht er *immer* einen Ausgang.

Die Wahrscheinlichkeit eines Knotens u, der ein Feld darstellt, wobei $|E_u| = 0$, ist immer 0, denn der Yamyam kann in keine Himmelsrichtung laufen. Dementsprechend gibt es für ihn auch keine einzige Möglichkeit, einen Ausgang zu erreichen, weshalb $P(u) = 0$.

Die Wahrscheinlichkeit $P(u)$ eines Knotens u, welcher ein Feld darstellt, wobei $|E_u| > 0$, wird folgendermaßen berechnet: Die Wahrscheinlichkeit, dass ein Knoten $v \in E_u$ ausgewählt wird, beträgt $\frac{1}{|E_u|}$. Die Wahrscheinlichkeit, dass der Yamyam vom Knoten v aus einen Ausgang erreichen wird, beträgt $P(v)$. Daher ist die Wahrscheinlichkeit, dass der Yamyam die Kante zu v entlangläuft und einen Ausgang erreichen wird, $\frac{1}{|E_u|}P(v)$. Die Wahrscheinlichkeit, dass der Yamyam von u aus einen Ausgang erreichen wird, ist die Summe all dieser Wahrscheinlichkeiten:

$$\begin{aligned} P(u) &= \sum_{v \in E_u} \frac{1}{|E_u|} P(v) \\ &= \frac{1}{|E_u|} \sum_{v \in E_u} P(v) \end{aligned} \tag{1}$$

In der Aufgabenstellung geht es nur um die Frage, für welche Felder u die Aussage $P(u) = 1$ gilt. Damit

$$P(u) = \frac{1}{|E_u|} \sum_{v \in E_u} P(v) = 1,$$

muss

$$\sum_{v \in E_u} P(v) = |E_u|$$

gelten. Daraus folgt wegen $0 \leq P(u) \leq 1$ sofort $P(v) = 1$ für alle $v \in E_u$. Denn wenn für irgendein $v \in E_u$ die Ungleichung $P(v) < 1$ gelten würde, dann wäre $\sum_{v \in E_u} P(v) < |E_u|$.

Weniger mathematisch ausgedrückt ist ein Knoten u sicher, wenn alle anderen Knoten, zu denen es eine Kante von u gibt, ebenfalls sicher sind. Für Knoten, die Felder repräsentieren, jedoch keine ausgehende Kante haben, ist die Wahrscheinlichkeit immer 0. Knoten, die Ausgänge darstellen, besitzen die Wahrscheinlichkeit 1.

Später ist ebenfalls folgende Aussage wichtig: Ist ein Knoten $u \in V$ nicht sicher, dann sind alle Knoten $v \in V$, die einen Weg zu u über die k Knoten $v, v_2, v_3, \cdots, v_{k-1}, u$ besitzen, ebenfalls nicht sicher. Dies kann daraus geschlossen werden, dass zunächst v_{k-1} nicht sicher ist. Daher kann auch v_{k-2} nicht sicher sein usw. v ist dann nicht sicher, weil v_2 nicht sicher ist.

1.5 Das Problem

Die Wahrscheinlichkeiten für Ausgänge sowie Felder mit Knoten ohne ausgehende Kanten können problemlos erfasst werden. Für alle anderen Knoten u müssen zuvor die Wahrscheinlichkeiten aller Knoten $v \in E_u$ ermittelt werden, damit nach Formel 1 $P(u)$ berechnet werden kann. Damit eine passende Reihenfolge gefunden werden kann, in der die Berechnungen getätigt werden können, ist es bspw. möglich, alle Kanten umzudrehen und den entstehenden Graphen topologisch zu sortieren. Da es in G jedoch Zyklen geben kann, ist dies nicht möglich.

Betrachtet man z. B. die Beispielwelt in Abbildung 1, dann ist die Wahrscheinlichkeit $P(2,4) = \frac{1}{3}(P(1,4)+P(2,1)+P(3,4))$. Für das Quadrat $(2,1)$ ist die Wahrscheinlichkeit $P(2,1) = \frac{1}{3}(P(1,1)+P(2,4)+P(3,1))$. Um die Wahrscheinlichkeit vom Quadrat $(2,4)$ zu berechnen, benötigt man die Wahrscheinlichkeit von $(2,1)$. Dasselbe gilt andersherum. Daher ist es nur mit Einsetzen in diese Formel nicht möglich, die Wahrscheinlichkeiten zu berechnen.

1.6 Die Lösung

Der Graph G wird in starke Zusammenhangskomponenten aufgeteilt. Eine starke Zusammenhangskomponente mit der Menge der Knoten K hat die Eigenschaft, dass jeder Knoten $u \in K$ von jedem anderen Knoten $v \in K$ über irgendeinen Weg erreichbar ist.

Ist irgendein Knoten $u \in K$ nicht sicher, dann sind auch alle anderen Knoten $v \in K$, nicht sicher, da es einen Weg von v nach u gibt. Dies bedeutet wiederum, dass entweder *alle* Knoten aus K sicher oder dass sie alle nicht sicher sind.

Seien Z_u die starke Zusammenhangskomponente des Knotens u und E_K die Menge aller Komponenten, die über Kanten direkt von Knoten aus K erreichbar sind. E_K besteht also aus allen Z_v von solchen $v \in V$, für die es eine Kante $e \in E$ gibt mit $e = (u, v, w)$, $u \in K$ und $v \notin K$.

Wenn eine Komponente K sicher ist, dann müssen alle Komponenten $K' \in E_K$ ebenfalls sicher sein. Dies wird folgendermaßen begründet: Jeder Knoten u einer sicheren Komponente K ist ebenfalls sicher. Damit ein Knoten u sicher ist, müssen alle $v \in V$ sicher sein, wenn es eine Kante $e \in E$ mit $e = (u, v, w)$ gibt. Da in diesem Fall v in einer Komponente liegt, die ebenfalls sicher ist und in E_K enthalten ist, müssen alle $K' \in E_K$ sicher sein.

Damit wurde jedoch noch nicht gezeigt, dass K auf jeden Fall sicher ist, wenn alle $K' \in E_K$ sicher sind. Das wird erst dadurch begründet, dass, wenn irgendeine Komponente $K' \in E_K$ nicht sicher ist, auch K nicht sicher sein kann, da jedes $u \in K$ einen Weg zu jedem nicht sicheren $v \in K'$ hat. Daher ist die Komponente K genau dann sicher, wenn alle $K' \in E_K$ sicher sind.

Diese Bedingung dafür, dass eine Komponente sicher ist, entspricht der, dass ein einzelner Knoten sicher ist. Bei der Komponente K ist es davon abhängig, dass alle Knoten aus E_K sicher sind, bei einem einzelnen Knoten u, dass alle Knoten aus E_u sicher sind. Daher kann ein neuer gerichteter und ungewichteter Graph $G' = (V', E')$ erstellt werden. Seine Knotenmenge V' ist die Menge der starken Zusammenhangskomponenten aus G. Daher ist ein Knoten $u \in V'$ eine Menge von Knoten aus V. Die Kantenmenge E' beinhaltet alle Kanten $e' = (u', v')$, wobei es eine Kante $e = (u, v, w)$ zwischen einem Knoten $u \in u'$ und einem Knoten $v \in v'$ mit beliebigem w gibt. Sollte es mehrere Kanten von einer Komponente zu einer anderen geben, dann reicht es, wenn nur eine davon hinzugefügt wird.

Jeder Ausgang $a \in V$ befindet sich in einer Komponente, in der sich keine anderen Knoten befinden. Dies liegt daran, dass a der Definition zufolge keine ausgehende Kante besitzt. Daher wären die anderen Knoten in der Komponente nicht von a aus erreichbar, was jedoch eine Bedingung für eine starke Zusammenhangskomponente ist. Die Komponente von a ist auch sicher, da sie nur aus dem sicheren Knoten a besteht. Wenn also der Yamyam dort ankommt, dann erreicht er auch auf jeden Fall einen Ausgang.

Der Graph, welcher die starken Zusammenhangskomponenten enthält, kann keine Zyklen mehr enthalten, da in einem Zyklus jeder Knoten von jedem anderen Knoten in diesem Zyklus erreicht werden kann. Dies würde bedeuten, dass die Knoten des Zyklus wiederum auf einer starken Zusammenhangskomponente liegen müssten. Aufgrund dieser Zyklenfreiheit ist es nun möglich, den Graph topologisch zu sortieren und die „Sicherheit“ in verkehrter Reihenfolge zu prüfen. Durch eine solche Sortierung können die Knoten, die keine ausgehenden Kanten besitzen, zuerst überprüft werden. Danach werden immer nur Knoten gewählt, deren Ziele der ausgehenden Kanten bereits zuvor getestet wurden.

1.6.1 Beispiel

In Abbildung 3 ist der Graph G' für die Beispielwelt aus Abbildung 1 zu sehen.

Im Beispiel wird zunächst die Komponente $K_1 = \{(1, 4)\}$ geprüft. Da sie eine Komponente ist, die einen Ausgang besitzt, ist sie sicher. Die Komponente $K_2 = \{(1, 1), (3, 1)\}$ besitzt ebenfalls keine ausgehende Kante. Daher kann von den Knoten innerhalb der Komponente niemals ein Ausgang erreicht werden und sie ist nicht sicher. Die Komponente $K_3 = \{(1, 3), (3, 3), (3, 4)\}$ hat nur eine ausgehende Kante, dessen Ziel K_1 bereits geprüft wurde und sicher ist. Daher ist auch die aktuelle Komponente sicher. Als nächstes wird $K_4 = \{(2, 1), (2, 4)\}$ ausgewählt, diese Komponente hat drei Kanten, die auf die bereits überprüften Knoten K_1, K_2 und K_3 zeigen. Da K_2 nicht sicher ist, kann auch K_4 nicht sicher sein. Die Komponente $K_5 = \{(2, 2)\}$ hat eine Kante, die auf die unsichere Komponente K_4 zeigt, weshalb K_5 unsicher ist. $K_6 = \{(2, 3)\}$ besitzt zwei Kanten,

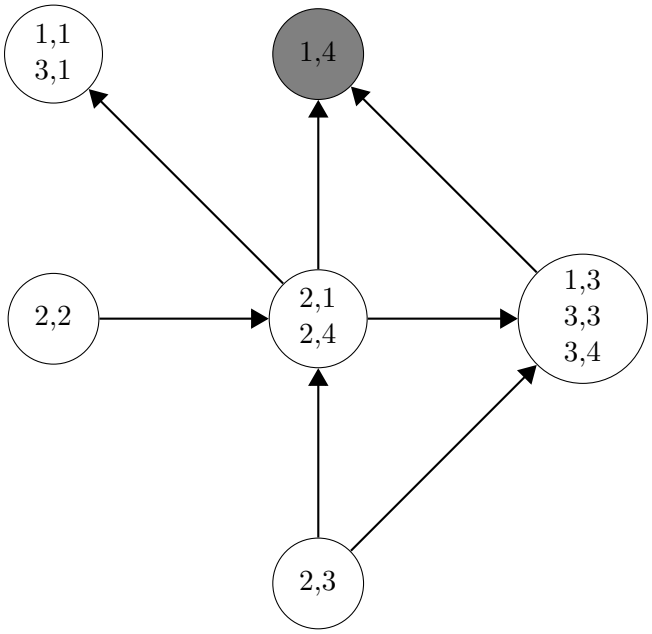


Abbildung 3: Der Graph aus Abbildung 2, wobei starke Zusammenhangskomponenten zu Knoten zusammengefasst wurden. Der graue Knoten ist der Ausgang.

wovon eine ebenfalls auf K_4 zeigt, weshalb auch K_6 nicht sicher ist.

Die einzigen sicheren Komponenten beinhalten also die Knoten $(1,4)$, $(1,3)$, $(3,3)$ und $(3,4)$. Davon sind jedoch nur die letzten drei sichere Felder, da $(1,4)$ ein Ausgang ist und nicht als Feld gewertet wird.

1.6.2 Algorithmus

Die starken Zusammenhangskomponenten werden mit dem Algorithmus von Tarjan bestimmt. Da dieser bekannt ist, reicht es, ihn hier nur kurz zu erläutern, ohne dabei alle Einzelheiten zu nennen oder die Korrektheit zu beweisen.

Von jedem Knoten aus wird eine Tiefensuche gestartet, sofern er noch nicht durch eine vorherige Suche entdeckt wurde. Jede starke Zusammenhangskomponente bekommt eine Wurzel zugewiesen, dies ist derjenige Knoten der Komponente, der sich im Tiefensuchbaum am weitesten oben befindet und daher als erstes von allen Knoten in der Komponente entdeckt wurde. Die Knoten werden in der Reihenfolge ihrer Entdeckung auf einem Stack gespeichert. Wenn eine Komponente vollständig ist und die Rekursion zu ihrer Wurzel zurückkehrt, dann werden alle Knoten bis zur Wurzel der Komponente vom Stack genommen. Alle entfernten Knoten befinden sich in einer Komponente. Wenn der rekursive Aufruf eines Knotens beendet wurde, welcher keine Wurzel einer Komponente ist, dann verbleibt er weiterhin auf dem Stack.

Damit überprüft werden kann, ob ein Knoten u einen Weg zu einem Knoten weiter oben im Baum besitzt, bekommt er zwei Werte: Den Index, der angibt, wann er besucht

wurde, der bei jedem neuen Knoten um eins steigt, sowie einen „Lowlink“, der Index, der dem höchsten Vorfahren von u (der mit dem kleinsten Index) gehört, der von u aus über einen Weg erreichbar ist.

Der Lowlink von u kann geändert werden, nachdem die Tiefensuche von u aus eine Baumkante zu einem zuvor unbesuchten Knoten gefunden hat. Ist sein Lowlink kleiner als der von u, dann ist der Knoten mit dem Index dieses Lowlinks im Baum weiter oben als u und auch von u aus erreichbar. Daher wird der Lowlink von u auf diesen Wert gesetzt wird. Außerdem kann der Lowlink von u geändert werden, wenn von u aus eine Kante zu einem Knoten gefunden wird, der sich auf dem Stack befindet und einen kleineren Index als den Lowlink von u hat. Dieser muss dementsprechend im Baum weiter oben als u sein und es gibt einen Weg von u dorthin. Daher muss der Lowlink von u auf den Index des gefundenen Knotens gesetzt werden.

Wenn der Aufruf eines Knotens u beendet wurde und der Index von u mit dem Lowlink von u übereinstimmt, dann hat u keinen Weg zu einem Knoten, der sich weiter oben im Baum befindet. Daher ist u die Wurzel einer starken Zusammenhangskomponente und es werden wie oben beschrieben die Knoten bis zu u vom Stack entfernt.

Wichtig ist, dass die Komponenten durch den Algorithmus von Tarjan in verkehrter topologischer Reihenfolge gefunden werden. Dies wird dadurch begründet, dass eine Komponente K, bevor der rekursive Aufruf ihrer Wurzel beendet wurde, zunächst alle Komponenten K', zu denen es eine Kante von K aus gibt, ebenfalls rekursiv aufruft. Ihre Wurzeln befinden sich im Tiefensuchbaum unter der Wurzel von K, ansonsten gäbe es Kanten von K aus, die über der Wurzel von K enden würden, wodurch diese Kantenziele ebenfalls Teil von K sein würden. Daher werden die Aufrufe von allen K' vor dem Aufruf von K beendet. Es werden also alle Komponenten K', zu denen es einen Weg von K aus gibt, vor K gefunden.

Aufgrund dieser Reihenfolge ist es gar nicht nötig, die Komponenten noch topologisch zu sortieren, um zu prüfen, ob sie sicher sind. Daher wird, sobald eine Komponente K gefunden wurde, überprüft, ob sie sicher ist, indem die Ziele von allen Kanten, die von K ausgehend in andere Komponenten führen, geprüft werden. Nur wenn mindestens eine solche Kante existiert und wenn alle dieser Kantenziele sicher sind, dann ist die aktuelle Komponente auch sicher.

Wenn die Komponente sicher ist, dann werden alle Knoten, die darin enthalten sind, zu einer Liste mit den sicheren Feldern hinzugefügt. Da auch Ausgänge im Graphen enthalten sind, jedoch nur sichere Felder gesucht werden, wird dies nur getan, wenn die Komponente keinen Ausgang repräsentiert (Ausgänge sind immer einzeln in einer Komponente, da sie keine ausgehenden Kanten haben).

1.6.3 Laufzeitanalyse

Die Laufzeit des Algorithmus von Tarjan benötigt bekanntermaßen $\mathcal{O}(|V|+|E|)$ Zeit, da jeder Knoten genau einmal rekursiv aufgerufen und jede Kante einmal durchlaufen wird. Außerdem wird jeder Knoten höchstens einmal vom Stack genommen. Beim Testen, ob eine Komponente sicher ist, wird jede ausgehende Kante erneut bearbeitet. Dies geschieht jedoch für jede Kante nur einmal und benötigt konstante Zeit. Daher wird die

Zeitkomplexität dadurch auch nicht verändert.

Da $|V|$ höchstens $s = n \cdot m$ Knoten beinhalten kann und jeder Knoten nur vier ausgehende Kanten haben kann, weshalb $|E| \leq 4 \cdot |V| \leq 4 \cdot n \cdot m$, benötigt der Algorithmus im schlechtesten Fall in der Anzahl der Quadrate lineare Zeit, $\mathcal{O}(s)$. Auch der erste Abschnitt, das Erstellen des Graphen, wird in $\mathcal{O}(s)$ Zeit beendet, sodass die Zeitkomplexität des gesamten Programmes bei $\mathcal{O}(s)$ bleibt.

1.7 Wahrscheinlichkeiten

Als Erweiterung werden hier zusätzlich die Wahrscheinlichkeiten für das Erreichen eines Ausgangs und die Erwartungswerte für die Anzahl der Felder, die ein Yamyam von seinem Startfeld aus gehen muss, bis er an einem Ausgang ankommt, berechnet.

1.7.1 Aufstellen eines linearen Gleichungssystems

Zuvor musste nur ermittelt werden, ob die Wahrscheinlichkeit eines Knotens 1 ist. Für eine Komponente K gilt entweder $\forall u \in K : P(u) = 1$ oder $\forall u \in K : P(u) < 1$. Die genauen Wahrscheinlichkeiten können jedoch innerhalb einer Komponente variieren, d. h., es muss nicht gelten: $\forall u \in K : P(u) = c$, wobei $0 \leq c \leq 1$ und c für alle Knoten in K identisch ist.

Zur Berechnung der Wahrscheinlichkeiten aller Knoten einer Komponente K werden die enthaltenen Knoten von 1 bis $|K|$ durchnummeriert. Der i-te Knoten aus K ($1 \leq i \leq |K|$, $i \in \mathbb{N}$) wird als u_i bezeichnet. Die Wahrscheinlichkeiten einer Komponente werden berechnet, direkt nachdem sie durch den Algorithmus von Tarjan gefunden wurde. Zur Abkürzung wird definiert, dass $P(u_i) = p_i$. Es wird nun ein quadratisches lineares Gleichungssystem mit $|K|$ Zeilen aufgestellt. Die i-te Gleichung hat auf der linken Seite p_i zu stehen, auf der rechten Seite eine Summe bestehend aus einer konstanten Zahl z_i sowie aus allen $a_{ij} \cdot p_j$, wobei $1 \leq j \leq |K|$. Es ergibt sich:

$$\begin{aligned} p_i &= \sum_{j=1}^{|K|} a_{ij} p_j + z_i \\ &= a_{i1} p_1 + a_{i2} p_2 + \cdots + a_{i|K|} p_{|K|} + z_i \end{aligned}$$

Es müssen nun alle a_{ij} und z_i für $1 \leq i, j \leq |K|$ für das Gleichungssystem ermittelt werden. Dafür wird die Formel 1 benötigt. $P(u_i)$ kann direkt von den Wahrscheinlichkeiten von zwei Arten von Knoten abhängig sein: Zur ersten Art gehören alle Knoten $v \in E_u$ mit $v \in Z_u$, zur zweiten Art gehören alle Knoten $v \in E_u$ mit $v \notin Z_u$. Durch die Aufteilung in die beiden Arten kann auch die Formel 1 für einen Knoten u_i angepasst werden:

$$P(u_i) = p_i = \left(\sum_{v \in E_u \cap Z_z} \frac{1}{|E_u|} P(v) \right) + \left(\sum_{v \in E_u \setminus Z_u} \frac{1}{|E_u|} P(v) \right)$$

Da aufgrund der verkehrten topologischen Sortierung durch den Algorithmus von Tarjan die Wahrscheinlichkeiten von allen Knoten v, wobei $v \in E_u$ und $v \notin Z_u$, bereits zuvor

berechnet wurden, können die Wahrscheinlichkeiten $P(v)$ solcher Knoten v als konstant und schon feststehend angesehen werden. Der rechte Summand wird daher sofort berechnet, wenn das lineare Gleichungssystem für die Komponente Z_u aufgestellt wird. Er bildet die konstante Zahl z_i.

Aus dem linken Summanden dieser Gleichung können alle Koeffizienten a_{ij} für $1 \leq j \leq |K|$ ermittelt werden. Es gilt auf jeden Fall $u_j \in Z_{u_i}$, da das Gleichungssystem immer nur für eine gesamte Komponente aufgestellt wird.

1. Wenn $u_j \in E_{u_i}$, dann ist $a_{ij} = \frac{1}{|E_{u_i}|}$.
2. Wenn $u_j \notin E_{u_i}$ dann ist $P(u_i)$ nicht direkt von $P(u_j)$ abhängig, weshalb $a_{ij} = 0$.

1.7.2 Lösen des linearen Gleichungssystems

Das Gleichungssystem wird durch Einsetzen gelöst. Es soll nach jedem Schritt $a_{ii} = 0$ für $1 \leq i \leq |K|$ gelten, damit in der i-ten Gleichung p_i auch nur auf der linken Seite steht und nicht auf der rechten. Der erste Schritt zur Lösung des lineraren Gleichungssystems besteht darin, p_2 in alle anderen Gleichungen einzusetzen. Durch das Einsetzen in die i-te Gleichung verändert sich diese folgendermaßen:

$$p_i = a_{i1}p_1 + a_{i2}(a_{21}p_1 + 0p_2 + \cdots + a_{2|K|}p_{|K|} + z_2) + \cdots + a_{i|K|}p_{|K|} + z_i$$

Durch Umformen ergibt sich dabei:

$$p_i = (a_{i1} + a_{i2}a_{21})p_1 + 0p_2 + \cdots + (a_{i|K|} + a_{i2}a_{2|K|})p_{|K|} + (z_i + a_{i2}z_2)$$

Der Koeffizient a_{i2} wird dadurch in jeder Zeile des Gleichungssystems zu 0. Zu dem Koeffizient a_{ij} mit $1 \leq i, j, \leq |K|$, $i, j \neq 2$, wird $a_{i2} \cdot a_{2j}$ hinzuaddiert. Zu z_i mit $1 \leq i \leq |K|$, $i \neq 2$ wird $a_{i2} \cdot z_2$ hinzuaddiert.

In einer Zeile i ist der Koeffizient a_{ii} jetzt unter Umständen nicht mehr 0. Dies muss korrigiert werden. Die erste Zeile z. B. wird nach dem Einsetzen der zweiten Gleichung zu

$$p_1 = a_{12}a_{21}p_1 + 0p_2 + (a_{13} + a_{12}a_{23})p_3 + \cdots + (a_{1|K|} + a_{12}a_{2|K|})p_{|K|} + (z_1 + a_{12}z_2)$$

Folgendermaßen wird die Gleichung umgeformt:

$$\begin{aligned}(1 - a_{12}a_{21})p_1 &= 0p_2 + (a_{13} + a_{12}a_{23})p_3 + \cdots + (a_{1|K|} + a_{12}a_{2|K|})p_{|K|} + (z_1 + a_{12}z_2) \\ p_1 &= \frac{0p_2 + (a_{13} + a_{12}a_{23})p_3 + \cdots + (a_{1|K|} + a_{12}a_{2|K|})p_{|K|} + (z_1 + a_{12}z_2)}{1 - a_{12}a_{21}}\end{aligned}$$

Es wird also nach dem Einsetzen jeder Koeffizient in der i-ten Gleichung a_{ij}, $1 \leq j \leq |K|$ sowie z_i durch $1 - a_{i2}a_{2i}$ geteilt, damit wieder $a_{ii} = 0$ gilt.

Nach dem Einsetzen von p_2 wird p_3 in alle anderen Gleichungen eingesetzt, um p_3 aus jeder rechten Seite zu eliminieren. Danach wird p_4 eliminiert usw., bis $p_{|K|}$. Danach ist $a_{ij} = 0$ für alle $j \neq 1$. In der ersten Zeile steht auf der rechten Seite nur noch z_1.

Daher ist dann $p_1 = z_1$. Alle anderen p_i können dann auch berechnet werden, da auf der rechten Seite nur noch die a_{i1}, welche jetzt bekannt sind, sowie z_i stehen.

Das Einsetzen ist äquivalentes Umformen, jedoch nur, wenn in der i-ten Zeile beim Einsetzen der j-ten Gleichung $a_{ij}a_{ji} \neq 1$, denn ansonsten dürfte die Division nicht stattfinden. Da jede Wahrscheinlichkeit nicht größer als 1 sein kann, ist auch $a_{ij}, a_{ji} \leq 1$. Es kann demnach nur $a_{i2}a_{2i} = 1$ sein, wenn $a_{i2} = 1$ und $a_{2i} = 1$. Der Koeffizient a_{ij} kann nur dann 1 sein, wenn alle anderen Koeffizienten in Zeile i eliminiert und dabei nie z_i erhöht wurden. Dies ist nur möglich, wenn es keine Kante aus der Komponente raus in einer andere Komponente gibt, da ansonsten $z_i > 0$ in einer Zeile i, was sich durch das Einsetzen in die anderen Gleichungen fortpflanzen würde. Wenn also irgendwann eine Division durch 0 stattfinden soll, dann gibt es keine Kante aus der Komponente hinaus, weshalb alle Wahrscheinlichkeiten $p_i = 0$ mit $1 \leq i \leq |K|$.

1.7.3 Beispiel

Die Vorgehensweise wird jetzt anhand der Komponente $K = \{(2,1),(2,4)\}$ im Graphen G' aus Abbildung 3 verdeutlicht. Der Knoten $(2,1)$ wird mit u_1 abgekürzt, $(2,4)$ mit u_2. Es werden die Wahrscheinlichkeiten von folgenden Knoten benötigt, welche durch den Algorithmus von Tarjan zuvor berechnet wurden:

- $P(1,4) = 1$, denn das Quadrat $(1,4)$ ist ein Ausgang.
- $P(1,1) = P(3,1) = 0$, denn von diesen beiden Quadrat gibt es keine einzige Möglichkeit, einen Ausgang zu erreichen.
- $P(3,4) = 1$, denn die zugehörige Komponente ist sicher.

Es entstehen folgende Wahrscheinlichkeiten:

$$P(u_1) = \frac{1}{3}(P(1,1) + P(3,1) + P(2,4)) = \frac{1}{3}(0 + 0 + P(u_2)) = \frac{1}{3}P(u_2)$$
$$P(u_2) = \frac{1}{3}(P(1,4) + P(2,1) + P(3,4)) = \frac{1}{3}(1 + P(u_1) + 1) = \frac{1}{3}P(u_1) + \frac{2}{3}$$

Daraus wird ein lineares Gleichungssystem mit folgenden beiden Gleichungen, wobei $P(u_1) = p_1$ und $P(u_2) = p_2$:

$$p_1 = \frac{1}{3}p_2 + 0$$
$$p_2 = \frac{1}{3}p_1 + \frac{2}{3}$$

p_2 wird in die erste Gleichung eingesetzt, welche danach umgeformt wird:

$$\begin{aligned} p_1 &= \frac{1}{3}(\frac{1}{3}p_1 + \frac{2}{3}) + 0 \\ p_1 &= \frac{1}{9}p_1 + \frac{2}{9} \\ \Leftrightarrow \frac{8}{9}p_1 &= \frac{2}{9} \\ \Leftrightarrow p_1 &= \frac{2}{9} \cdot \frac{9}{8} = \frac{1}{4} \end{aligned}$$

Das Gleichungssystem sieht danach folgendermaßen aus:

$$\begin{aligned} p_1 &= \frac{1}{4} \\ p_2 &= \frac{1}{3}p_1 + \frac{2}{3} \end{aligned}$$

Da $p_1 = \frac{1}{4}$ jetzt bekannt ist, kann auch p_2 berechnet werden:

$$\begin{aligned} p_1 &= \frac{1}{4} \\ p_2 &= \frac{1}{3} \cdot \frac{1}{4} + \frac{2}{3} = \frac{3}{4} \end{aligned}$$

Damit ist $P(2,1) = \frac{1}{4}$ und $P(2,4) = \frac{3}{4}$.

1.7.4 Laufzeitanalyse

Es wird angenommen, dass alle arithmetische Operationen konstante Zeit benötigen.

Das Aufstellen des Gleichungssystems benötigt für die Komponente K $\mathcal{O}(|K|^2)$ Zeit, da es $|K|^2$ Koeffizienten gibt, die jeweils in konstanter Zeit berechnet werden. Das Berechnen aller konstanten Zahlen benötigt $\mathcal{O}(|K|)$ Zeit, da es $|K|$ dieser Zahlen gibt. Sie werden in konstanter Zeit berechnet, da jeder Knoten maximal vier ausgehende Kanten besitzen kann, durch welche diese Zahlen erhöht werden.

Das Einsetzen einer Gleichung in eine andere Gleichung benötigt $\mathcal{O}(|K|)$ Zeit, da jeder Koeffizient sowie die zusätzliche Zahl in konstanter Zeit neu berechnet werden können. Da $|K| - 1$ Zahlen in jeweils $|K| - 1$ Gleichungen in jeweils $\mathcal{O}(|K|)$ Zeit eingesetzt werden müssen, benötigt das Lösen eines Gleichungssystems der Komponente K insgesamt $\mathcal{O}(|K|^3)$ Zeit. Wenn die gesamte Welt aus einer Komponente besteht (wofür natürlich bei großen Welten auch an einigen Stellen Hindernisse stehen müssen), wird für die gesamte Welt mit $s = n \cdot m$ Feldern $\mathcal{O}(s^3)$ Zeit benötigt (das Erstellen des Graphen und das Finden der Komponenten benötigt Zeit $\mathcal{O}(s)$ und wird daher vernachlässigt).

Die sicheren Felder hätten auch durch das Bestimmen der Wahrscheinlichkeiten gefunden werden können, indem überprüft wird, welche Felder die Wahrscheinlichkeit 1 haben. Da das Bestimmen der Wahrscheinlichkeiten jedoch nicht Teil der ursprünglichen Aufgabenstellung war und man möglicherweise nur die sicheren Felder bestimmen

möchte, wird dieser Algorithmus seperat verwendet. Nur wenn wirklich die Wahrscheinlichkeiten benötigt werden, wird er ausgeführt, damit in allen anderen Fällen nur $\mathcal{O}(s)$ Zeit benötigt wird.

Anmerkung: Wenn die exakten Werte berechnet werden, indem rationale Zahlen als Brüche mit Zähler und Nenner verwendet werden, sollten sie nach jeder Operation (Multiplikation, Addition, etc.) gekürzt werden, damit die Zahlen auch durch bspw. 64 Bit dargestellt werden können. Dadurch können die arithmetischen Operationen jedoch streng genommen nicht in konstanter Zeit durchgeführt werden. Trotzdem geht das Kürzen durch den euklidischen Algorithmus für Zahlen mit maximal 64 Bit sehr schnell, daher wird das hier vernachlässigt.

1.8 Erwartungswerte

Auf dieselbe Art und Weise können auch Erwartungswerte für die Anzahl der Schritte, die ein Yamyam im Mittel gehen muss, bis er an einem Ausgang ankommt, berechnet werden. Der Erwartungswert eines Knotens $u \in V$ ist $E(u)$.

Für Felder $u \in V$, für die $P(u) = 0$ gilt, ist es nicht möglich, einen Erwartungswert zu berechnen, da der Yamyam von dort aus nie an einem Ausgang ankommen kann. Für solche Felder wird daher $E(u)$ nicht definiert. Ist hingegen $P(u) = 1$, dann wird der Yamyam immer an einem Ausgang ankommen. Der Erwartungswert gibt dann an, wie viele Schritte der Yamyam im Mittel fährt, bis er an einem Ausgang ankommt.

Wenn $0 < P(u) < 1$, dann ist es auch möglich, $E(u)$ zu ermitteln. In diesem Fall gibt der Wert an, wie viele Schritte der Yamyam im Mittel fährt, bis er an einem Ausgang ankommt, wenn er nur Kanten $e \in E$, $e = (u, v, w)$ entlang geht, wobei $P(v) > 0$. Er befindet sich dann nur auf Feldern v mit $P(v) > 0$. Dadurch können sich die Wahrscheinlichkeiten ändern, dass ein Yamyam eine Kante entlangfährt. Besitzt z. B. ein Knoten vier ausgehende Kanten, wovon nur genau eine auf ein Feld mit der Wahrscheinlichkeit 0 zeigt, dann ist die Wahrscheinlichkeit, dass er diese Kante entlangfährt, 0. Die Wahrscheinlichkeit, dass er die drei anderen Kanten entlangfährt, erhöht sich von $\frac{1}{4}$ auf $\frac{1}{3}$, da er irgendeine Kante nehmen muss.

Simuliert man einen Yamyam, um einen ungefähren Erwartungswert zu erhalten, muss im Fall $0 < P(u) < 1$ folgendes beachtet werden: Der Yamyam darf wirklich nur Kanten entlangfahren, die auf Quadraten enden, deren Wahrscheinlichkeit größer als 0 ist. Es ist *nicht* möglich, einfach alle beliebigen Kanten zu nehmen und dann nur die Wege in den Erwartungswert einzuberechnen, bei denen der Yamyam irgendwann einen Ausgang gefunden hat. Durch diese Methode würden falsche Erwartungswerte entstehen, da folgendes Szenario entstehen kann: Der Yamyam geht gleich oft auf zwei verschiedene Felder u und v und es ist $P(u) < P(v)$. Die Wege über das Feld v würden öfters in den Erwartungswert einberechnet werden als die über u, obwohl der Yamyam gleich oft beide Felder betritt. Aus diesem Grund würde eine solche Simulation nicht den hier definierten Erwartungswert widerspiegeln.

Für die Erwartungswerte werden die Kantengewichte des Graphen G benötigt. E_u eines Knotens u wurde zuvor als Menge aller Knoten v definiert, für die es eine Kante von u zu v gibt. Da jetzt die Kantenlängen wichtig sind, wird F_u eines Knotens $u \in V$ als

Menge aller geordneten Paare (v, w) definiert, wobei es eine Kante $e \in E$, $e = (u, v, w)$, $P(v) \neq 0$ gibt. Das erste Element eines geordneten Paares p wird als $p^{(1)}$ bezeichnet, das zweite Element ist $p^{(2)}$.

Der Erwartungswert eines Feldes $u \in V$ lässt sich durch folgende Formel berechnen:

$$\begin{aligned} E(u) &= \sum_{g \in F_u} \frac{1}{|F_u|}(E(g^{(1)}) + g^{(2)}) \\ &= \frac{1}{|F_u|} \sum_{g \in F_u} (E(g^{(1)}) + g^{(2)}) \end{aligned}$$

Die Wahrscheinlichkeit, dass der Yamyam eine Kante von u aus entlanggeht, beträgt $\frac{1}{|F_u|}$. Die Anzahl der Schritte, die der Yamyam im Mittel fahren muss, wenn er die Kante nimmt, die durch $g \in F_u$ dargestellt wird, ist $E(g^{(1)}) + g^{(2)}$. Dies ist die Länge des Weges bis zu dem nächsten Knoten, sowie die mittlere Anzahl der Schritte, nachdem der Yamyam dann an diesem Knoten angekommen ist.

Die Erwartungswerte werden, wie die Wahrscheinlichkeiten, berechnet, nachdem die zugehörige Komponente gefunden wurde. Die im Gleichungssystem verwendete Formel lautet:

$$e_i = a_{i1}e_1 + a_{i2}e_2 + \cdots + a_{i|K|}e_{|K|} + z_i$$

Dabei ist e_i der Erwartungswert des Knotens in der aktuellen Komponente K mit der Nummer i. Die Knoten in K werden auch hier von 1 bis $|K|$ durchnummeriert. Jeder Knoten in der Komponente K bekommt eine Gleichung. Sie wird aus der Formel für den Erwartungswert ermittelt.

Für den Knoten u mit der Nummer i hat der Koeffizient a_{ij} zu Beginn den Wert $\frac{1}{|F_u|}$, wenn es eine Kante von u zu dem Knoten mit der Nummer j gibt. Gibt es keine solche Kante, dann ist $a_{ij} = 0$. z_i ist hier jedoch nicht wie bei der Berechnung der Wahrscheinlichkeiten nur die Summe der bereits berechneten Erwartungswerte von den Knoten in F_u mit den zugehörigen Weglängen, sondern zusätzlich noch die Summe aller Kantenlängen in F_u von den Knoten, die in K liegen, da diese als konstante Zahlen bereits zu z_i hinzuaddiert werden können (F'_u ist hier die Teilmenge von F_u, deren Knoten nicht in K liegen):

$$z_i = \frac{1}{|F'_u|} \sum_{g \in F'_u} (E(g^{(1)}) + g^{(2)}) + \frac{1}{|F_u \setminus F'_u|} \sum_{g \in F_u \setminus F'_u} g^{(2)}.$$

Das entstehende Gleichungssystem wird dann genau wie das Gleichungssystem für die Wahrscheinlichkeiten gelöst.

1.8.1 Beispiel

Die Vorgehensweise zur Aufstellung des linearen Gleichungssystems soll auch hier anhand der Komponente $K = \{(2, 1), (2, 4)\}$ im Graphen G' aus Abbildung 3 verdeutlicht werden. Der Knoten $(2, 1)$ wird mit u_1 abgekürzt, $(2, 4)$ mit u_2. Es werden folgende zuvor berechneten Erwartungswerte benötigt:

- $E(1,4) = 0$, denn das Quadrat $(1,4)$ ist ein Ausgang.
- $E(3,4) = \frac{9}{2}$.

Es entstehen folgende Erwartungswerte:

$$\begin{aligned}
E(u_1) &= 1(E((2,4)) + 3) \\
&= 1E(u_2) + 3 \\
E(u_2) &= \frac{1}{3}(E((2,1)) + 3) + \frac{1}{3}(E((1,4)) + 1) + \frac{1}{3}(E((3,4)) + 1) \\
&= \frac{1}{3}(E((2,1)) + 3) + \frac{1}{3}(0 + 1) + \frac{1}{3}(\frac{9}{2} + 1) \\
&= \frac{1}{3}E(u_1) + \frac{19}{6}
\end{aligned}$$

Es ergibt sich das folgende Gleichungssystem, wobei $E(u_1) = e_1$ und $E(u_2) = e_2$:

$$\begin{aligned}
e_1 &= 1e_2 + 3 \\
e_2 &= \frac{1}{3}e_1 + \frac{19}{6}
\end{aligned}$$

Durch Einsetzen der zweiten Gleichung in die erste ergibt sich $e_1 = \frac{37}{4}$ und danach $e_2 = \frac{25}{4}$.

1.8.2 Laufzeitanalyse

Die Laufzeitanalyse ist ähnlich zu der für die Wahrscheinlichkeiten. Es wird auch hier angenommen, dass arithmetische Operationen konstante Zeit benötigen.

Das Aufstellen des Gleichungssystems für die Komponente K benötigt $\mathcal{O}(|K|^2)$ Zeit, da es $|K|^2$ Koeffizienten gibt, die alle in konstanter Zeit berechnet werden. Die zusätzlichen konstanten Zahlen, von welchen $|K|$ berechnet werden müssen, benötigen jeweils konstante Zeit, da F_u nur maximal vier Elemente beinhalten kann. Das Lösen des Gleichungssystem benötigt auch hier $\mathcal{O}(|K|^3)$ Zeit.

2 Umsetzung

Das Programm wurde in der Programmiersprache Swift 2 geschrieben und ist lauffähig auf OS X 10.9 Mavericks. Möglicherweise muss bspw. `chmod` verwendet werden, damit es ausgeführt werden kann. Es kann dann im Terminal gestartet werden. Um die Lösung einer Aufgabe zu erhalten, muss der Pfad zur Datei mit der Eingabe eingegeben werden. Möchte man zusätzlich die Wahrscheinlichkeiten und Erwartungswerte der Felder berechnet haben, muss zusätzlich der Parameter `-p` angehängt werden.

2.1 Programmstruktur

- *Rational.swift* - Implementiert rationale Zahlen.
- *Task.swift* - Löst eine Aufgabe.
- *TaskReader.swift* - Liest eine Aufgabe ein.
- *main.swift* - Startet den `TaskReader` sowie `Task` und gibt am Ende die Lösung aus.

Der für die Aufgabe relevante Teil wird also in *Task.swift* ausgeführt. Darin sind folgende Komponenten enthalten:

- `VertexType` - Dies ist ein `enum` und besteht aus den Werten `Empty` und `Exit`. Dadurch wird angegeben, ob ein Knoten ein normales Feld oder einen Ausgang darstellt.
- `Edge` - Stellt eine Kante als einfachen `struct` dar. Sie besteht nur aus dem Kantenziel (einem Knoten) sowie aus der Kantenlänge (einem Integer), welche für das Berechnen der Erwartungswerte benötigt wird.
- `Vertex` - Stellt einen Knoten als `class` dar.
- `VertexStack` - Dieser `struct` implementiert einen Stack für Knoten, welcher für den Algorithmus von Tarjan benötigt wird.
- `Task` - Diese `class` löst eine Aufgabe. Sie erstellt zunächst den Graphen und führt dann den Algorithmus von Tarjan aus.

In der Datei *Rational.swift* gibt es einen `struct`, dies ist `Rational`, welcher eine rationale Zahl implementiert. Er speichert dafür den Zähler sowie den Nenner des Bruches. Außerdem werden verschiedene Operatoren wie Addition etc. zur Verfügung gestellt. Nachdem eine solche Aktion durchgeführt wurde, wird er mithilfe des euklidischen Algorithmus gekürzt. Wenn Integer Overflows durch die Operatoren entstehen, speichert ein `Rational` anstelle des Zählers und des Nenners eine ungefähre Zahl als `Double`, mit welcher dann weitergerechnet werden kann.

2.2 `Vertex`

Wird ein Knoten erstellt, dann werden im Konstruktor direkt die drei Werte `row`, `column` und `type` übergeben, welche die Koordinaten sowie die Art des Knotens als `VertexType` angeben. Dies sind die einzigen drei Variablen, welche später nicht mehr geändert werden können. Ein Knoten beinhaltet außerdem eine Adjazenzliste namens `edges`, welche alle ausgehenden Kanten repräsentiert. Diese Kanten werden erst später durch `Task` hinzugefügt. Für den Algorithmus von Tarjan werden die beiden Integer-Werte `index` und `lowlink` sowie die boolesche Variable `isOnStack` benötigt, wobei `index` (zu Beginn -1)

auch später noch als eindeutige Identifizierung des Knotens verwendet wird. Dies wird bei der Berechnung der Wahrscheinlichkeiten und Erwartungswerte benötigt. Wenn eine Komponente gefunden wurde, dann werden die Variablen `isInCurrentComponent` von jedem darin enthaltenen Knoten auf `true` gesetzt. Dieser Wert wird benötigt, um zu überprüfen, ob die ausgehenden Kanten der Knoten in derselben Komponente enden oder ob sie in eine andere Komponente führen. Nach der Bearbeitung der Komponente werden sie entsprechend wieder auf `false` gesetzt. Die Variable `isSecure` eines Knotens gibt an, ob er sicher ist. Anfangs ist er `false` und wird nur auf `true` gesetzt, wenn die zugehörige Komponente gefunden und als sicher identifiziert wurde.

2.3 `VertexStack`

Diese Klasse speichert einen Stack von `Vertex`, welcher als einfaches Array implementiert ist. Es stellt nur die beiden vom Algorithmus von Tarjan benötigten Methoden `push()` und `pop()` zur Verfügung.

2.4 `Task`

Diese Klasse löst die Aufgabe und prüft, welche Felder sicher sind. Wenn gefordert, werden auch die Wahrscheinlichkeiten und Erwartungswerte berechnet.

Es gibt vier unveränderliche Werte, mit welchen die `Task` erstellt wird. Sie stellen allein die Aufgabenstellung dar. Dies sind zunächst `width` und `height`, sie sind die Breite und Höhe der Welt als Integer. Als drittes wird die eigentliche Welt `world` vom Typ `[[Vertex?]]` übergeben. Sie wird als zweidimensionales Array dargestellt. Der erste Index gibt die Zeile, der zweite Index die Spalte an. Es gibt drei Möglichkeiten für jedes darin enthaltene Quadrat: Ist es `nil`, dann befindet sich an dieser Stelle ein Hindernis. Aus diesem Grund wird der optionale Typ `Vertex?` verwendet. Für ein Hindernis gibt es keinen Knoten. Ist es nicht `nil`, dann wird zwischen einem Feld und einem Ausgang durch die Variable `type` des Knotens unterschieden. Als letztes wird noch die boolesche Variable `checkProbabilities` übergeben, welche angibt, ob zusätzlich zu den sicheren Feldern die Wahrscheinlichkeiten und Erwartungswerte berechnet werden sollen.

Um die Aufgabe zu lösen, muss die Methode `run()` ausgeführt werden. Sie ruft zunächst `findVertices()` und dann `findEdges()` auf, um den Graphen zu erstellen. Danach durchläuft sie alle Knoten des Graphen, und führt, sofern sie noch nicht durch die Tiefensuche entdeckt wurden, den Algorithmus von Tarjan durch die Methode `tarjan()` mit ihnen als Startpunkt aus. Wenn `index` eines Knotens -1 entspricht, dann wurde er noch nicht entdeckt.

2.4.1 `findVertices()`

Diese Methode hat die Aufgabe, alle Knoten des Graphen zu dem eindimensionalen Array `vertices` hinzuzufügen. Dafür durchläuft sie einfach jede Zeile in `world`, dann jeden Knoten in der Zeile und fügt ihn, sofern er nicht `nil` ist, zu `vertices` hinzu.

2.4.2 `findEdges()`

Wenn diese Methode ohne Parameter aufgerufen wird, fügt sie alle Kanten in der gesamten Welt hinzu. Sie kann auch mit einem Parameter, einem Array des Typs `[Vertex?]`, ausgeführt werden. In diesem Fall muss das Array eine Zeile oder eine Spalte der Welt sein, entweder vorwärts oder rückwärts. Damit durch sie alle maximal vier Kanten von jedem Knoten aus in jede Richtung hinzugefügt werden, werden von der ersten `findEdges()`-Methode zunächst die Zeilen und dann die Spalten durchlaufen, um sie jeweils normaler und in verkehrter Richtung der zweiten Methode zu übergeben.

Die zweite Methode speichert immer das Ende der nächsten Kanten sowie die aktuelle Kantenlänge zwischen. Es wird dann das übergebene Array durchlaufen und es werden zu den Feldern Kanten hinzugefügt, die die gespeicherte Kantenlänge besitzen und bei dem gespeicherten Knoten enden. Ausgänge haben, wie in Abschnitt 1 beschrieben, keine ausgehenden Kanten.

2.4.3 `tarjan()`

Wird die Tarjan-Tiefensuche auf einen Knoten ausgeführt, dann werden zunächst die Variablen `index` und `lowlink` des Knotens gesetzt. Der Tiefensuchzähler `index` der Klasse `Task`, welcher am Anfang bei 0 beginnt, wird dann inkrementiert.

Wenn der Knoten ein Ausgang ist, dann ist bereits bekannt, dass er keine ausgehende Kante besitzt und in seiner eigenen Komponente ist. Daher kann in diesem Fall schon gleich seine Variable `isSecure` auf `true` gesetzt und die Methode vorzeitig beendet werden.

Ansonsten wird der aktuelle Knoten auf den Stack gelegt und es werden rekursiv alle Knoten aufgerufen, zu denen es vom aktuellen Knoten eine direkte Kante aus gibt. Dabei kann auch `lowlink` des aktuellen Knotens entsprechend angepasst werden.

Anschließend wird überprüft, ob der Knoten die Wurzel einer starken Zusammenhangskomponente ist. Dies ist der Fall, wenn sein `index` und `lowlink` identisch sind. Alle Knoten bis zur Wurzel werden vom Stack genommen und alle ausgehenden Kanten aus den Knoten der gefundenen Komponente werden überprüft und gezählt. Wenn es mindestens eine davon gibt und wenn alle Knoten, auf die diese Kanten zeigen, ebenfalls sicher sind (ihre Komponenten wurden in jedem Fall bereits vorher von der Tiefensuche gefunden), sind alle Knoten in der aktuellen Komponente ebenfalls sicher, weshalb `isSecure` von ihnen auf `true` gesetzt wird.

2.5 Wahrscheinlichkeiten und Erwartungswerte

Um die Wahrscheinlichkeiten und Erwartungswerte zu berechnen, muss das Gleichungssystem aufgestellt werden. Auf der linken Seite steht die Wahrscheinlichkeit eines Knotens, auf der rechten Seite stehen eine konstante Zahl sowie die Koeffizienten der Wahrscheinlichkeiten von allen anderen Knoten in der entsprechenden Komponente. Aus diesem Grund wird einfach in jedem Knoten der Komponente die rechte Seite der Gleichung gespeichert, bei der auf der linken Seite die Wahrscheinlichkeit des eigenen Knotens steht. Daher enthält jeder Knoten vier weitere Variablen. Dies sind zunächst `probability` und

`expectedValue` vom Typ `Rational`. Sie stellen jeweils die konstante Zahl auf der rechten Seite des Gleichungssystems dar. Wurde das Gleichungssystem fertig gelöst, dann stellt diese Zahl die fertige Wahrscheinlichkeit bzw. den Erwartungswert dar, denn dann ist jeder Koeffizient 0. Zudem beinhaltet jeder Knoten `probabilityCoefficients` und `expectedValueCoefficients` vom Typ `[Int: Rational]` (in anderen Programmiersprachen auch „hash map“ genannt). Sie stellen die Koeffizienten dar. Der Schlüssel darin entspricht dem `index` eines Knotens, der zugehörige Wert ist dann der Koeffizient, der vor der Wahrscheinlichkeit des Knotens aus dem Schlüssel steht.

Die Wahrscheinlichkeiten und Erwartungswerte werden, sofern gefordert, berechnet, nachdem alle ausgehenden Kanten einer Komponente geprüft wurde. Dafür werden zunächst `calculateProbabilityCoefficients()` und `calculateExpectedValueCoefficients()` auf alle Knoten in der Komponente aufgerufen. Anschließend werden `replaceProbabilityCoefficientOfVertex()` und `replaceExpectedValueCoefficientOfVertex()` verwendet, um Gleichungen in andere Gleichungen einzusetzen. Der Wert für den ersten Knoten steht danach bereits fest. Die Werte der anderen Knoten werden aus dem des ersten Knotens und aus ihren konstanten Werten berechnet.

Es wird `Rational` verwendet, damit die beiden Werte exakt berechnet werden können und nicht unter der Ungenauigkeit von Gleitkommazahlen leiden.

2.5.1 `calculateProbabilityCoefficients()` und `calculateExpectedValueCoefficients()`

Diese Methoden der Klasse `Vertex` berechnen die Koeffizienten und die Konstante auf der rechten Seite der Gleichung des Knotens. Sie durchlaufen dafür alle Kanten. Wenn das Ziel einer Kante in der eigenen Komponente liegt (was durch `isInCurrentComponent` überprüft wird), dann wird ein Koeffizient hinzugefügt. Ansonsten wird die entsprechende Zahl zur Konstante addiert. Bei den Koeffizienten der Erwartungswerte müssen zuvor die Kanten herausgefiltert werden, die auf Knoten zeigen, deren Erwartungswerte nicht berechnet werden können. Zudem müssen die Kantenlängen noch zur konstanten Zahl hinzugefügt werden.

Ist der Erwartungswert für einen Knoten nicht definiert, dann wird das durch den Wert -1 dargestellt. Hat ein Knoten keine ausgehende Kante, bekommt er den Erwartungswert -1.

2.5.2 `replaceCoefficientOfVertex()`

Diese Methode der Klasse `Vertex` wird von `replaceProbabilityCoefficientOfVertex()` und `replaceExpectedValueCoefficientOfVertex()` aufgerufen. Sie ersetzt den Koeffizienten eines Knotens in der eigenen Gleichung der Wahrscheinlichkeit oder des Erwartungswertes. Die Variable mit den Koeffizienten sowie den konstanten Wert des eigenen Knotens bekommt die Methode als `inout`-Parameter zur Verfügung gestellt. Der konstante Wert und die Koeffizienten der Wahrscheinlichkeit des zu ersetzenden Knotens werden als normale Argumente übergeben, da sie nicht abgeändert werden müssen. Durch diese Parameter muss der gesamte Inhalt der Methode nicht zweimal für

die Wahrscheinlichkeit sowie den Erwartungswert kopiert werden. Die entsprechenden Parameter werden einfach von den beiden Methoden `replaceProbabilityCoefficientOfVertex()` und `replaceExpectedValueCoefficientOfVertex()` aus dem aktuellen Knoten sowie aus dem Knoten, dessen Wahrscheinlichkeit ersetzt wird, ermittelt.

Zusätzlich wird ein weiterer Standardwert als Parameter benötigt. Er wird im Falle, dass auf der rechten Seite der Gleichung nur noch der eigene Wert mit einer eins als Koeffizient steht, als Wahrscheinlichkeit bzw. Erwartungswert eingesetzt, da ansonsten eine Division durch null stattfinden müsste. Dies ist nur dann der Fall, wenn die Komponente keine ausgehende Kante, endend in einer anderen Komponente, besitzt. Bei Erwartungswerten müssen diese Komponenten auch noch Wahrscheinlichkeiten größer null haben. Wie bereits beschrieben ist der Standardwert in diesem Fall -1. Wird die Wahrscheinlichkeit berechnet, dann ist der Standardwert 0, da die Wahrscheinlichkeit einer Komponente ohne ausgehender Kante 0 ist.

3 Beispiele

Es werden zunächst die Beispiele gezeigt, die auf der Website vom Bundeswettbewerb Informatik zu finden waren. In Beispiel 7 wird die Lösung der Beispielwelt aus Abbildung 1 gezeigt, in Beispiel 8 wird die Lösung eines größeren Beispiels berechnet.

Zur Ausgabe des Programmes gehört zunächst die Zeitangabe, wie lange das Programm nur zum Lösen der Aufgabe lief. Es folgt die Welt des Yamyams, wobei sichere Felder durch ein „o“ gekennzeichnet werden. Anschließend wird eine Liste der Koordinaten aller sicheren Felder ausgegeben, welche in der Dokumentation jedoch nicht gezeigt werden. Wenn angegeben wurde, dass die Wahrscheinlichkeiten und Erwartungswerte berechnet werden sollen, folgt danach eine Liste der Koordinaten von allen Feldern mit ihren Wahrscheinlichkeiten und, sofern definiert, ihren Erwartungswerten. Sie werden hier nur bei kleinen Beispielen gezeigt, da sie ansonsten über mehrere Seiten gehen müssten. In Ordner der Abgabe finden sich jedoch die Ausgaben von allen Beispielen, wobei auch überall die Wahrscheinlichkeiten und Erwartungswerte sowie die Liste mit den sicheren Feldern angezeigt werden.

Unter allen Programmausgaben steht in der Dokumentation, wie viele Millisekunden das mehrfache Ausführen auf einem gewöhnlichen Computer durchschnittlich benötigte. Der erste Wert steht für das Ausführen des Programmes, wobei nur die sicheren Felder gesucht werden, der zweite Wert für Durchläufe, bei denen zusätzlich die Wahrscheinlichkeiten und Erwartungswerte berechnet wurden.

3.0 Beispiel 0

```
############
# #ooooooo# #
#    ##      #
#    #oooE# #
#         ## #
#            #
# #E  oooE###
############
```

Die Wahrscheinlichkeiten und Erwartungswerte für das Erreichen eines Zieles:

```
1|1   -   0 = 0.00%
1|3   -   1 = 100.00%   -   9 = 9.00
1|4   -   1 = 100.00%   -   11 = 11.00
1|5   -   1 = 100.00%   -   11 = 11.00
1|6   -   1 = 100.00%   -   91/8 ≈ 11.38
1|7   -   1 = 100.00%   -   91/8 ≈ 11.38
1|8   -   1 = 100.00%   -   8 = 8.00
1|10   -   0 = 0.00%
2|1   -   1/4 = 25.00%   -   11 = 11.00
2|2   -   3/8 = 37.50%   -   14 = 14.00
2|3   -   3/4 = 75.00%   -   9 = 9.00
2|6   -   3/4 = 75.00%   -   63/4 = 15.75
2|7   -   3/4 = 75.00%   -   63/4 = 15.75
2|8   -   3/4 = 75.00%   -   99/8 ≈ 12.38
2|9   -   1/2 = 50.00%   -   79/4 = 19.75
2|10   -   1/4 = 25.00%   -   79/4 = 19.75
3|1   -   1/4 = 25.00%   -   11 = 11.00
3|2   -   3/8 = 37.50%   -   14 = 14.00
3|3   -   3/4 = 75.00%   -   9 = 9.00
3|5   -   1 = 100.00%   -   26/5 = 5.20
3|6   -   1 = 100.00%   -   317/40 ≈ 7.92
3|7   -   1 = 100.00%   -   317/40 ≈ 7.92
3|10   -   0 = 0.00%
4|1   -   1/4 = 25.00%   -   95/4 = 23.75
4|2   -   3/8 = 37.50%   -   163/8 ≈ 20.38
4|3   -   3/4 = 75.00%   -   123/8 ≈ 15.38
4|4   -   5/8 = 62.50%   -   311/16 ≈ 19.44
4|5   -   3/4 = 75.00%   -   601/40 ≈ 15.03
4|6   -   3/4 = 75.00%   -   71/4 = 17.75
4|7   -   3/4 = 75.00%   -   71/4 = 17.75
4|10   -   0 = 0.00%
5|1   -   0 = 0.00%
5|2   -   1/8 = 12.50%   -   17 = 17.00
5|3   -   1/2 = 50.00%   -   7 = 7.00
5|4   -   3/8 = 37.50%   -   121/8 ≈ 15.12
5|5   -   1/2 = 50.00%   -   63/10 = 6.30
5|6   -   1/2 = 50.00%   -   47/4 = 11.75
```

```
5|7   -   1/2 =  50.00%   -   47/4 =  11.75
5|8   -   1/3 ≈  33.33%   -   1 =  1.00
5|9   -   0 =  0.00%
5|10   -   0 =  0.00%
6|1   -   0 =  0.00%
6|4   -   7/8 =  87.50%   -   141/16 ≈  8.81
6|5   -   1 =  100.00%   -   22/5 =  4.40
6|6   -   1 =  100.00%   -   57/8 ≈  7.12
6|7   -   1 =  100.00%   -   57/8 ≈  7.12
```

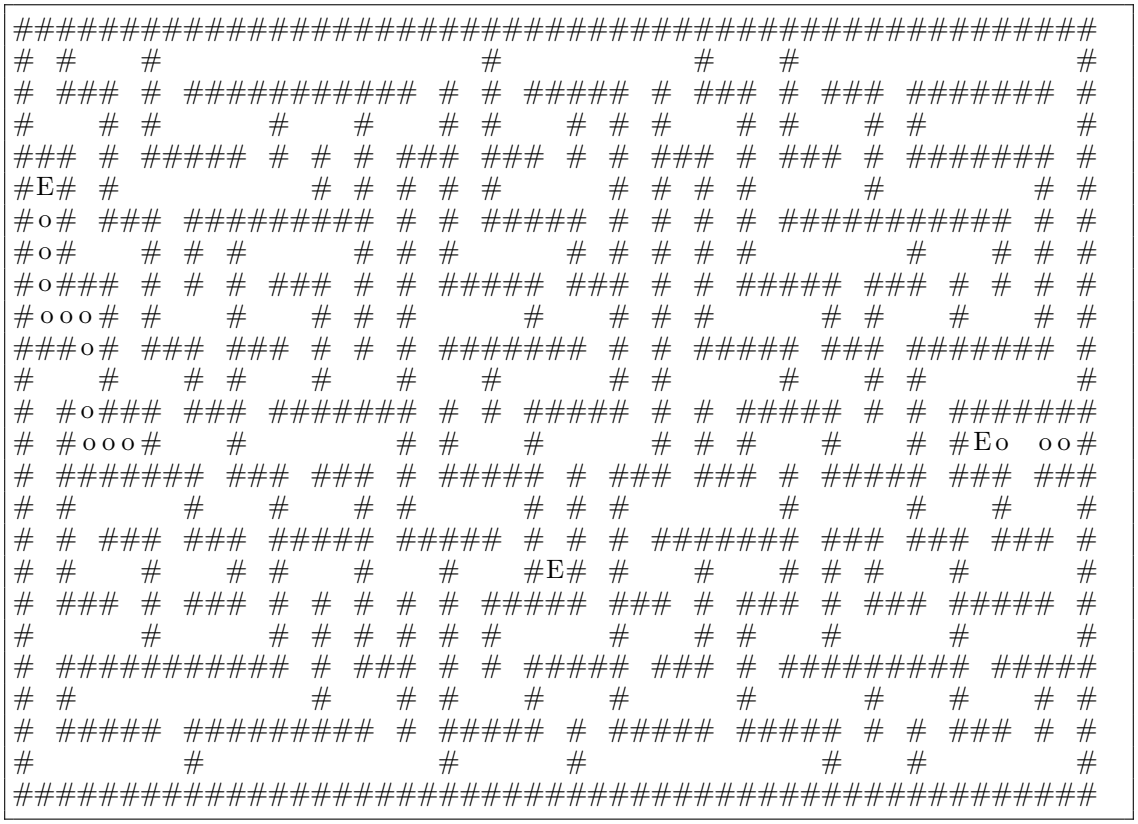
Dies ist das Beispiel, welches in der Aufgabenstellung verwendet wird. Zum Finden der sicheren Felder benötigte das Programm auf einem gewöhnlichen Computer durchschnittlich ca. 0,3 Millisekunden. Wenn zusätzlich die Wahrscheinlichkeiten und Erwartungswerte berechnet wurden, benötigte das Programm ca. 0,5 Millisekunden.

3.1 Beispiel 1

```
##########################
#oooo#      #      #oooooooo#
#oooo#      #      #oooooooo#
#           #           Eoo##
#oooo                 oooooooo#
#                     #oooooooo#
#oooo##   ##         #oooooooo#
#oooo#o    o#         #ooo#ooo#
#oooo#o  Eo#         #oooooooo#
#ooo E######         #oooooooo#
#oooo#      #      #oooooooo#
#oooo#      #      #oooooooo#
##########################
```

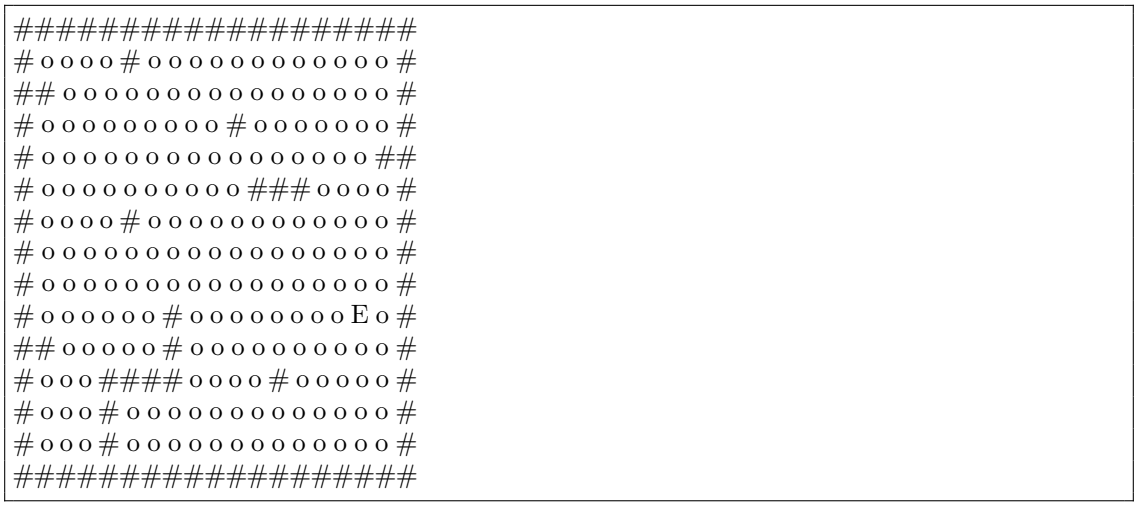
Für dieses Beispiel benötigte der Computer durchschnittlich 0,6 bzw. 1 Millisekunde.

3.2 Beispiel 2



Für dieses Beispiel benötigte der Computer durchschnittlich 2 bzw. 6 Millisekunden.

3.3 Beispiel 3



Für dieses Beispiel benötigte der Computer durchschnittlich 0,6 bzw. 2 Millisekunden.

3.4 Beispiel 4

```
####################
#      #         #      #
##     #                #
#              ##        #
#      #                 #
#      #       ###       #
#   E  #####             #
#         #               #
#         #               #
#         #####           #
##        #               #
#o  o####      #######    #
#o  o#                    #
#o  o#      #             #
####################
```

Für dieses Beispiel benötigte der Computer durchschnittlich 0,5 bzw. 2 Millisekunden.

3.5 Beispiel 5

```
##########################
#oooooooooooooo#          #
#ooooooooooooooo# ###     #
#ooooooooooooooo###       #
#ooooooo###o##             #
#oooo###oooo#              #
#oooo#oooo#o#      #       #
#oooo#oo###o#      #       #
####o#ooooooo#    ###      #
#Eooo  ooo####      #       #
####   #ooo   ooo# #   #   #
#       #       ###E     #  #
#       #ooo    o#      ### #
#       ###o    o#          #
#    E      o   oo  o###     #
#               ##     #     #
#####    #o    o#      #     #
#    #    ##    #             #
#    #         #oo  o#        #
#    # ####  ###   #          #
#           #        ###      #
#                   E         #
#                             #
##########################
```

Für dieses Beispiel benötigte der Computer durchschnittlich 1 bzw. 3 Millisekunden.

3.6 Beispiel 6

```
######
#E##o#
#ooooo#
##ooo#
#oo#o#
#o#oo#
#ooooo#
##ooo#
#oo#o#
#o#oo#
#ooooo#
##ooo#
#oo#o#
#o#oo#
#ooooo#
######
```

Für dieses Beispiel benötigte der Computer durchschnittlich 0,4 bzw. 1 Millisekunde.

3.7 Beispiel 7

```
######
#  #oE#
#     #
#  #oo#
######
```

Die Wahrscheinlichkeiten und Erwartungswerte für das Erreichen eines Zieles:

```
1|1   -   0 = 0.00%
1|3   -   1 = 100.00%   -   9/2 = 4.50
2|1   -   1/4 = 25.00%   -   37/4 = 9.25
2|2   -   1/2 = 50.00%   -   37/4 = 9.25
2|3   -   3/4 = 75.00%   -   31/4 = 7.75
2|4   -   3/4 = 75.00%   -   25/4 = 6.25
3|1   -   0 = 0.00%
3|3   -   1 = 100.00%   -   6 = 6.00
3|4   -   1 = 100.00%   -   9/2 = 4.50
```

Für dieses Beispiel benötigte der Computer durchschnittlich 0,2 bzw. 0,3 Millisekunde.

3.8 Beispiel 8

Dieses Beispiel passt aufgrund seiner Größe mit $1000 \cdot 1000 = 1000000$ Quadraten nicht in die Dokumentation, es ist jedoch im Ordner mit den Beispielen dieser Aufgabe in der Einsendung zu finden.

Die Welt wurde mithilfe eines Zufallszahlengenerators erstellt. Am Rand befinden sich, wie in allen anderen Beispielen, Hindernisse. Die Arten der anderen Quadrate wurden

zufällig generiert. Die erste Art ist der Ausgang, sie wurden in einem von zwanzig Fällen in verwendet. In weiteren neun der zwanzig Fällen wurden Hindernisse platziert, die restlichen Quadrate wurden freigelassen.

Für dieses Beispiel benötigte der Computer durchschnittlich gut 1 bzw. 4 Sekunden. Dabei muss jedoch beachtet werden, dass die Zeit nur während des Algorithmus (Erstellen des Graphen, Algorithmus von Tarjan und evtl. Berechnung der Wahrscheinlichkeiten und Erwartungswerte) gemessen wird. Das Einlesen der Datei und die Ausgabe zählen nicht dazu. Es wurden ca. 375 Megabyte Speicher benötigt.

4 Quelltext

Listing 1: main.swift

```
let task = TaskReader().read()
task.run()

// Ausgabe der Lösung...
```

Listing 2: Task.swift

```
// Die Art eines Knotens in Graphen (Feld / Ausgang). Hindernisse gehören nicht dazu.
enum VertexType {
    case Empty
    case Exit
}

// Eine Kante von einem Knoten zu einem anderen. Die besitzt das Ziel der Kante sowie die
    Kantenlänge.
struct Edge {
    let to: Vertex
    let length: Int
}

// Ein Knoten. Er besteht aus seinen Koordinaten sowie einer Knotenart.
// Die Kanten werden nach der Erstellung hinzugefügt. Alle anderen Werte werden für
// die Suche der Zusammenhangskomponenten oder für die Berechnung der Wahrscheinlichkeiten
// bzw. Erwartungsewrte benötigt.
class Vertex {

    let row: Int
    let column: Int
    let type: VertexType

    var edges = [Edge]()

    // Gibt an, als wievielter Knoten er besucht wurde
    var index = -1
    // Lowlink für den Algorithmus von Tarjan
```

```
    var lowlink = -1
    // Gibt an, ob sich der Knoten auf dem Stack befindet
    var isOnStack = false
    // Gibt an, ob sich der Knoten in der Komponente befindet, die gerade bearbeitet wird
    var isInCurrentComponent = false
    // Gibt an, ob der Knoten sicher ist
    var isSecure = false

    // Wahrscheinlichkeit und Erwartungswert sowie die dafür verwendeten Koeffizienten
    // für das Gleichungssystem. Die konstanten Zahlen in der Gleichung wird in
    // der Wahrscheinlichkeit bzw. dem Erwartungswert gespeichert.
    var probability = Rational(number: 0)
    var expectedValue = Rational(number: 0)
    var probabilityCoefficients = [Int: Rational]()
    var expectedValueCoefficients = [Int: Rational]()

    init(row: Int, column: Int, type: VertexType) {
        self.row = row
        self.column = column
        self.type = type
    }

    // Berechnet die Koeffizienten sowie die konstante Zahl für die Gleichung der
        Wahrscheinlichkeit.
    func calculateProbabilityCoefficients() {
        // Die Koeffizienten sind für alle Wahrscheinlichkeiten, die einen Koeffizient
            besitzen, am Anfang gleich:
        let coefficient = Rational(counter: 1, denominator: edges.count)

        for edge in edges {
            if edge.to.isInCurrentComponent {
                probabilityCoefficients[edge.to.index] = coefficient
            } else {
                // Nicht in der Komponente, daher wird die Wahrscheinlichkeit zur
                    konstanten Zahl hinzuaddiert.
                probability += coefficient * edge.to.probability
            }
        }
    }

    // Berechnet die Koeffizienten sowie die konstante Zahl für die Gleichung des
        Erwartungswertes.
    func calculateExpectedValueCoefficients() {
        // Nur Kanten zu Knoten nehmen, deren Erwartungswert berechnet werden kann:
        let usedEdges = edges.filter { $0.to.expectedValue != -1 }
        let coefficient = Rational(counter: 1, denominator: usedEdges.count)

        for edge in usedEdges {
            if edge.to.isInCurrentComponent {
```

```
            expectedValueCoefficients[edge.to.index] = coefficient
            // Die konstante Zahl erhöht sich auch durch die Weglängen der Kanten:
            expectedValue += coefficient * edge.length
        } else {
            // Nicht in der Komponente, daher Wert zur konstanten Zahl hinzuaddieren.
            expectedValue += coefficient * (edge.to.expectedValue + edge.length)
        }
    }

    if usedEdges.count == 0 {
        // Es gibt keine ausgehende Kante, die auf einen Knoten zeigt, dessen
            Erwartungswert berechnet werden kann. Daher kann auch dieser Erwartungswert
            nicht berechnet werden.
        expectedValue = Rational(number: -1)
    }
}

// Setzt die Gleichung des angegebenen Knotens in die Gleichung dieses Knotens ein.
// Zu additionValue werden alle entstehenden konstanten Zahlen hinzuaddiert.
// otherAdditionalValue ist die konstante Zahl der einzusetztenden Gleichung.
// coefficients sind die Koeffizienten der eigenen Gleichung, otherCoefficients die
// der eingesetzten. defaultValue wird als entstehender Wert der Gleichung verwendet,
// wenn die Gleichung nicht eindeutig lösbar ist. Dies ist bei Erwartungswerten der
// Fall, wenn es keine ausgehende Kante aus der Komponente gibt, die zu einem Knoten
// führt, dessen Erwartungswert eindeutig bestimmt werden konnte.
func replaceCoefficientOfVertex(vertex: Vertex, inout additionalValue: Rational,
    otherAdditionalValue: Rational, inout coefficients: [Int: Rational],
    otherCoefficients: [Int: Rational], defaultValue: Rational) {

    if let coefficient = coefficients[vertex.index] {
        coefficients.removeValueForKey(vertex.index) // aus Gleichung entfernen

        // Erhöhen des zusätzlichen Wertes:
        additionalValue += coefficient * otherAdditionalValue

        var selfCoefficient: Rational? // Koeffizient der eigenen Variablen
        for (index, value) in otherCoefficients {
            if index == self.index {
                // Koeffizient der eigenen Variablen gefunden
                selfCoefficient = coefficient * value
            } else if let previousCoefficient = coefficients[index] {
                // Koeffizient gibt es bereits in der eigenen Gleichung
                coefficients[index] = previousCoefficient + (coefficient * value)
            } else {
                // Koeffizient war in der eigenen Gleichung zuvor 0
                coefficients[index] = coefficient * value
            }
        }
```

```
            if let selfCoefficient = selfCoefficient {
                if selfCoefficient == 1 {
                    // Gleichung ist nicht eindeutig lösbar, da auf beiden Seiten nur
                    // die eigene Variable mit einer 1 als Koeffizient steht
                    additionalValue = defaultValue
                } else {
                    // Ansonsten alle Koeffizienten sowie den zusätzlichen Wert
                        multiplizieren, sodass auf der linken Seite wieder eine 1 als
                        Koeffizient der eigenen Variablen steht
                    let multiplier = 1 / (1 - selfCoefficient)
                    additionalValue *= multiplier
                    for (index, value) in coefficients {
                        coefficients[index] = value * multiplier
                    }
                }
            }
        }
    }

    // Setzt die Gleichung des gegebenen Knotens in die eigene ein, um die
        Wahrscheinlichkeiten zu berechnen
    func replaceProbabilityCoefficientOfVertex(vertex: Vertex) {
        replaceCoefficientOfVertex(vertex, additionalValue: &self.probability,
            otherAdditionalValue: vertex.probability, coefficients:
            &probabilityCoefficients, otherCoefficients: vertex.probabilityCoefficients,
            defaultValue: Rational(number: 0))
    }

    // Setzt die Gleichung des gegebenen Knotens in die eigene ein, um die Erwartungswerte
        zu berechnen
    func replaceExpectedValueCoefficientOfVertex(vertex: Vertex) {
        replaceCoefficientOfVertex(vertex, additionalValue: &self.expectedValue,
            otherAdditionalValue: vertex.expectedValue, coefficients:
            &expectedValueCoefficients, otherCoefficients:
            vertex.expectedValueCoefficients, defaultValue: Rational(number: -1))
    }

}

// VertexStack wird als Stack für Knoten verwendet. Er ist als Array implementiert.
struct VertexStack {

    var items = [Vertex]()

    mutating func push(item: Vertex) {
        items.append(item)
    }

    mutating func pop() -> Vertex {
```

```
        return items.removeLast()
    }

}

// Task findet alle sicheren Felder einer Welt und, wenn gefordert, die Wahrscheinlichkeiten
// und Erwartungswerte aller Felder.
class Task {

    // Die Breite und Höhe der Welt
    let width: Int
    let height: Int
    // Alle Quadrate als zweidimensionales Array. Hindernisse entsprechen nil.
    let world: [[Vertex?]]
    // Wenn true, werden zusätzlich Wahrscheinlichkeiten und Erwartungswerte berechnet
    let checkProbabilities: Bool

    // Beinhaltet alle Knoten
    var vertices = [Vertex]()
    // Index für den nächsten entdeckten Knoten
    var index = 0
    // Stack vom Algorithmus von Tarjan
    var stack = VertexStack()
    // Liste mit den Koordinaten aller sicheren Knoten
    var result = [(row: Int, column: Int)]()

    init(width: Int, height: Int, world: [[Vertex?]], checkProbabilities: Bool) {
        self.width = width
        self.height = height
        self.world = world
        self.checkProbabilities = checkProbabilities
    }

    // Startet die Berechnung. Es werden zunächst die Knoten und Kanten erstellt,
    // danach wird der Algorithmus von Tarjan gestartet.
    func run() {
        findVertices()
        findEdges()

        for vertex in vertices where vertex.index == -1 {
            // Noch nicht entdeckt, also Algorithmus von Tarjan mit diesem Knoten starten
            tarjan(vertex)
        }
    }

    // Fügt alle Knoten der Welt in das Array ein.
    func findVertices() {
        for row in world {
            for vertex in row {
```

```
            if let vertex = vertex {
                vertices.append(vertex)
            }
        }
    }
}

// Sucht alle Kanten in allen Zeilen und Spalten, vorwärts und rückwärts.
func findEdges() {
    for row in world {
        // Jede Zeile überprüfen
        findEdges(row)
        findEdges(row.reverse())
    }
    for columnIndex in 0..<width {
        // Jede Spalte überprüfen, indem aus jeder Zeile das entsprechende
        // Quadrat genommen und zur Spalte hinzugefügt wird
        let column = world.map{ $0[columnIndex] }
        findEdges(column)
        findEdges(column.reverse())
    }
}

// Sucht alle Kanten der gegebenen Zeile bzw. Spalte, in der Reihenfolge
// des Arrays.
func findEdges(vertices: [Vertex?]) {
    // Knoten, auf den die nächsten Kanten zeigen, ausgehend von Knoten weiter
    // hinten im Array
    var edge: Vertex? = nil
    // Die Länge der Kante bis zum gespeicherten Knoten
    var edgeLength = 0

    for vertex in vertices {
        if let vertex = vertex {
            if vertex.type == .Exit {
                // Ausgang gefunden, die nächsten Felder müssen eine Kante dorthin
                // besitzen, wenn kein Hindernis dazwischen ist
                edge = vertex
                edgeLength = 0
            } else {
                // Es wurde ein freies Feld gefunden
                if let edge = edge {
                    // Es wurde ein Knoten gespeichert, daher muss eine
                    // Kante dorthin hinzugefügt werden mit der entsprechenden Länge
                    edgeLength++
                    vertex.edges.append(Edge(to: edge, length: edgeLength))
                } else {
                    // Kein gespeichertes Feld, daher bekommen die nächsten Felder
                    // eine Kante hierhin, wenn kein Hindernis dazwischen ist
```

```
                    edge = vertex
                    edgeLength = 0
                }
            }
        } else {
            // Hindernis gefunden, also gespeicherten Knoten zurücksetzen
            edge = nil
            edgeLength = 0
        }
    }
}

// Die Tiefensuche durch den Algorithmus von Tarjan.
func tarjan(vertex: Vertex) {
    vertex.index = index
    vertex.lowlink = index
    index++

    if vertex.type == .Exit {
        // Der Knoten ist ein Ausgang. Er besitzt keine ausgehende
        // Kante und ist in einer eigenen, sicheren Komponente
        // mit Wahrscheinlichkeit 1 und Erwartungswert 0.
        vertex.isSecure = true
        vertex.probability = Rational(number: 1)
        vertex.expectedValue = Rational(number: 0)
        return
    }

    // Knoten auf den Stack legen
    stack.push(vertex)
    vertex.isOnStack = true

    // Alle Kanten prüfen, rekursiv Knoten aufrufen und Lowlink anpassen.
    for edge in vertex.edges {
        if edge.to.index == -1 {
            tarjan(edge.to)
            vertex.lowlink = min(vertex.lowlink, edge.to.lowlink)
        } else if edge.to.isOnStack {
            vertex.lowlink = min(vertex.lowlink, edge.to.index)
        }
    }

    // Der Knoten ist Wurzel einer starken Zusammenhangskomponente
    if vertex.lowlink == vertex.index {
        var componentVertices = [Vertex]()

        // Alle Knoten, die zur Komponente gehören, vom Stack entfernen,
        // die entsprechenden Variablen anpassen und zu componentVertices hinzufügen:
        var current: Vertex
```

```
        repeat {
            current = stack.pop()
            current.isOnStack = false
            current.isInCurrentComponent = true
            componentVertices.append(current)
        } while vertex !== current

        // Die Komponente ist sicher, wenn sie mindestens eine ausgehende Kante in eine
        // andere Komponente besitzt und alle Komponenten, zu denen sie eine Kante hat,
        // ebenfalls sicher sind:
        var hasOutgoingEdge = false
        var secure = true

        for vertex in componentVertices {
            for edge in vertex.edges where edge.to.isInCurrentComponent == false {
                hasOutgoingEdge = true
                if !edge.to.isSecure {
                    secure = false
                }
            }

            if checkProbabilities {
                // Bei der Gelegenheit gleich die Koeffizienten für die Berechnung
                // der Wahrscheinlichkeiten und Erwartungswerte initialisieren:
                vertex.calculateProbabilityCoefficients()
                vertex.calculateExpectedValueCoefficients()
            }
        }

        if checkProbabilities {
            // Wahrscheinlichkeiten und Erwartungswerte berechnen

            // Alle Gleichungen, außer die erste, in alle anderen Gleichungen einsetzen:
            for replace in componentVertices.dropFirst() {
                for replaceIn in componentVertices {
                    if replace !== replaceIn {
                        replaceIn.replaceProbabilityCoefficientOfVertex(replace)
                        replaceIn.replaceExpectedValueCoefficientOfVertex(replace)
                    }
                }
            }

            // Für den ersten Knoten wurden die Werte bereits fertig berechnet. Alle
            // anderen müssen noch mithilfe der ersten Gleichung berechnet werden:
            for vertex in componentVertices.dropFirst() {
                vertex.probability +=
                    vertex.probabilityCoefficients[componentVertices[0].index]! *
                    componentVertices[0].probability
```

```
                    if componentVertices[0].expectedValue == -1 {
                        // Erwartungswert konnte nicht berechnet werden
                        vertex.expectedValue = Rational(number: -1)
                    } else {
                        vertex.expectedValue +=
                            vertex.expectedValueCoefficients[componentVertices[0].index]! *
                            componentVertices[0].expectedValue
                    }
                }
            }

            // isInCurrentComponent zurücksetzen und gegebenenfalls isSecure auf true
            // setzen. Außerdem Knoten zur Liste mit den sicheren Knoten hinzufügen,
            // wenn die Komponente sicher ist.
            for vertex in componentVertices {
                vertex.isInCurrentComponent = false
                if hasOutgoingEdge && secure {
                    vertex.isSecure = true
                    result.append((row: vertex.row, column: vertex.column))
                }
            }
        }
    }

}
```