# Missglückte Drohnenlieferung

**Aufgabe 2, Runde 2, 34. Bundeswettbewerb Informatik**

Marian Dietz

## 1 Lösungsidee

### 1.1 Darstellung als Graph

Das quadratische Amacity mit der Seitenlänge $n > 0$ besteht aus $m = n^2$ Häusern. Es ist ein gerichteter und ungewichteter Graph $G = (V, E)$ mit den Knoten $V$ und den Kanten $E$. Ein Knoten $u \in V$ ist ein geordnetes Paar $u = (x, y)$ und stellt das Haus mit den Koordinaten $x$ und $y$ dar, wobei $x$ die Zeile und $y$ die Spalte ist. Für jedes Haus gibt es genau einen Knoten in $V$, es gilt $|V| = m$. Eine Kante $e \in E$, die durch ein geordnetes Paar $(u, v)$ beschrieben wird, beginnt bei dem Knoten $u$ und endet bei dem Knoten $v$. Sie gibt an, dass ein Paket, welches sich auf dem Haus des Knotens $u$ befindet, direkt mit einem Wurf auf das Haus des Knotens $v$ geworfen werden kann. Daher gilt für $u = (x_u, y_u)$ und $v = (x_v, y_v)$ eine der folgenden Bedingungen:

- $x_u - 1 = x_v$ und $y_u = y_v$: Das Paket wird ein Haus weiter nach Norden geworfen.
- $x_u = x_v$ und $y_u + 1 = y_v$: Das Paket wird ein Haus weiter nach Osten geworfen.
- $x_u + 1 = x_v$ und $y_u = y_v$: Das Paket wird ein Haus weiter nach Süden geworfen.
- $x_u = x_v$ und $y_u - 1 = y_v$: Das Paket wird ein Haus weiter nach Westen geworfen.

Andere Fälle sind nicht möglich, da ein Paket nur in eine der vier Himmelsrichtungen und jeweils ein Haus weiter geworfen werden kann. $E$ besteht aus allen Würfen, die diese Bedingungen erfüllen. Von jedem Haus aus kann das Paket in jede der vier Himmelsrichtungen geworfen werden, wenn sich in der Richtung ein weiteres Haus befindet. Dies ist nur dann nicht der Fall, wenn das Haus direkt am Rand der Stadt steht. $|E| = 4m - 4n$, denn es gäbe $4m$ Richtungen, wenn von jedem Haus in jede Richtung geworfen werden könnte, davon müssen jedoch $4n$ abgezogen werden, da es an jeder der 4 Seiten mit $n$ Häusern in eine Richtung nicht möglich ist, Pakete zu werfen. „Eckhäuser“ zählen dabei natürlich zu zwei Seiten, da sie nur in zwei Richtungen werfen können, sofern $n > 1$.

Der Graph $G$ ändert sich nie während des Algorithmus.

## 1.2 Eigenschaften

### 1.2.1 Paketziel

Ein *Zustand* weist jedem Haus ein bestimmtes Paket zu. Ein *Paket* besteht lediglich aus den Koordinaten, an welchen es sich am Ende befinden soll. Das Paket, welches sich am Ende auf dem Haus $q$ befinden soll, wird „Paket $q$“ genannt.

$z(u)$, wobei $u \in V$, ist das Paket $q = z(u)$, welches sich zurzeit (im aktuellen Zustand) auf dem Haus $u$ befindet.

$c(q)$ entspricht den Koordinaten des Hauses, auf welchem sich das Paket $q$ zurzeit befindet.

Daraus ergibt sich: Wenn $z(u) = q$, dann ist $c(q) = u$.

### 1.2.2 Abstand zum Ziel

Die Funktion $d(u, v)$ gibt an, wie oft irgendein Paket, welches sich auf dem Haus $u$ befindet, *mindestens* weitergeworfen werden muss, damit es sich auf dem Haus mit den Koordinaten $v$ befindet. Dies ist unabhängig zum aktuellen Zustand, da es egal ist, welches Ziel das Paket hat. Wenn $u = (x_u, y_u)$ und $v = (x_v, y_v)$, gilt

$$d(u, v) = |x_u - x_v| + |y_u - y_v|,$$

denn das Paket muss noch mindestens $|x_u - x_v|$ Mal nach Norden oder nach Süden geworfen werden, bis es sich in der Zeile $x_v$ befindet und muss noch $|y_u - y_v|$ Mal nach Osten oder nach Westen geworfen werden, bis es sich in der Spalte $y_v$ befindet.

Die Funktion $d(q)$ gibt an, wie oft das Paket $q$ von $c(q)$ aus mindestens weitergeworfen werden muss, damit es sich auf dem Haus mit den Koordinaten $q$ befindet. Dies ist abhängig vom aktuellen Zustand, denn $c(q)$ kann bei jedem Zustand unterschiedlich sein. Es gilt

$$d(q) = d(c(q), q).$$

Durch $d_x(q)$ wird angegeben, wie oft das Paket mit dem Ziel $q = (x, y)$ von $c(q)$ aus mindestens weitergeworfen werden muss, damit es sich in der Zeile $x$ befindet. $d_y(q)$ ist die Mindestanzahl an Würfen, damit das Paket mit dem Ziel $q = (x, y)$ in der Spalte $y$ ist. Es gilt für $q = (x_q, y_q)$ und $c(u) = (x_c, y_c)$:

$$d_x(q) = |x_c - x_q|$$

und

$$d_y(q) = |y_c - y_q|.$$

Daher ist $d(q) = d_x(q) + d_y(q)$.

Die Gesamtdistanz $d_{ges}$ ist die Summe der Distanzen aller Pakete zu ihren Zielen:

$$d_{ges} = \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} d((x, y)).$$

Die Maximaldistanz $d_{max}$ ist das Maximum der Distanzen aller Pakete zu ihren Zielen:

$$d_{max} = \max_{0 \le x < n} \max_{0 \le y < n} d((x, y)).$$

## 1.3 Mindestschrittanzahl

Sieht man sich die Paketverteilung von Amacity an, so erkennt man, dass das Paket $(0, 9)$ auf dem Haus $(6, 0)$, sich mit der Distanz 15 von allen Paketen am weitesten von seinem Ziel entfernt befindet. In jedem Schritt kann es seinem Ziel nur ein Haus näher kommen. Die Mindestschrittanzahl beträgt daher 15. Mit weniger Schritten kann man keine Lösung finden. Dies garantiert aber natürlich nicht, dass es auch auf jeden Fall einen Plan mit 15 Schritten gibt. Mit dem in der Lösungsidee beschriebenen Algorithmus wird ein Plan mit 18 Schritten gefunden. Lässt man ihn mit zufälligen Kriterien (siehe Abschnitt 1.9.3) noch weiter laufen, ist es möglich, dass auch noch Pläne mit weniger als 18 Schritten gefunden werden. Dies kann aber nicht garantiert werden.

## 1.4 Überblick

Das Problem ist eigentlich nur für den vorgegebenen Plan für Amacity gestellt. Der Algorithmus soll daher vor allem in diesem einen Fall die besten Ergebnisse liefern.

Trotzdem könnte es sein, dass das Experiment der Drohnenlieferung, welches laut Aufgabenstellung nur einmalig ist, irgendwann erneut durchgeführt wird. Es könnte z. B. sein, dass die Firma der Drohnen die Mitarbeiter testen möchte, ob diese es vielleicht bei weiteren Versuchen hinbekommen, die Adressenlisten nicht zu vertauschen. Sollten sie dies nicht schaffen oder wurde mit einer weiteren Drohnen-Version ein Bug eingeführt, welcher die Pakete bei falschen Häusern abliefert, dann muss ein weiterer Plan entwickelt werden, mit dem die Pakete durch Werfen an ihren richtigen Orten ankommen. Daher sollen mit dem hier vorgestellten Algorithmus auch bei anderen als der gegebenen Paketverteilung gute Lösungen entstehen. Zumindest soll garantiert werden, dass für jede gegebene Verteilung irgendeine Lösung gefunden wird.

Es gibt verschiedene Einstellungen, die am Algorithmus vorgenommen werden können, um die Suche einzugrenzen oder um länger zu suchen. In den Beispielen steht genau, bei welchen Einstellungen (für die gegebene Paketverteilung in Amacity sowie für andere Beispiele) welche Ergebnisse geliefert werden.

### 1.4.1 Finden von Schritten

Es wird versucht, in möglichst wenig Schritten alle Pakete zu ihren jeweiligen Zielen zu befördern. Dies wird mit einem Greedy-Algorithmus getan, der für jeden Schritt den Zustand auswählt, der „am besten aussieht“. Für diese Bewertung werden verschiedene Kriterien verwendet, siehe Abschnitt 1.8. Dies kann jedoch nicht garantieren, dass das Problem durch ein solches Kriterium in der Mindestschrittanzahl gelöst werden kann, der Greedy-Algorithmus ist hier nicht optimal. Um bei verschiedenen Paketverteilungen möglichst gute Ergebnisse zu erzielen, werden die verschiedenen Kriterien verwenden, welche alle durchprobiert werden können. Die Anzahl an Schritten kann dadurch in einigen Fällen etwas sinken. Außerdem ist es manchmal sinnvoll, Schritte wieder rückgängig zu machen und andere Schritte zu verwenden. Auch durch dieses Verfahren ist es möglich, in weniger Schritten zum Ziel zu kommen. Es wird in Abschnitt 1.9.4 beschrieben.

Sobald sich nach einem Schritt alle Pakete an ihrem Ziel befinden, wurde eine Lösung gefunden.

In Abschnitt 1.11 wird der Algorithmus kurz anhand eines Beispiels veranschaulicht.

## 1.5 Finden eines Schrittes

Um einen Schritt zu machen, werden zunächst alle Pakete $q$ nach ihrer Distanz zum Ziel $d(q)$ sortiert. Das Paket mit der größten Distanz wird als erstes *bearbeitet*, als letztes das mit der kleinsten Distanz. Dadurch werden Knoten, welche von ihrem Ziel weiter entfernt sind als „wichtiger“ bewertet. Von jedem Knoten aus wird maximal eine Kante der höchstens vier *ausgewählt*, die verwendet wird, um nach dem Schritt das Paket auf das entsprechende angrenzende Haus zu werfen.

$G' = (V, E')$ wird in diesem Abschnitt als Graph verwendet, dessen die Knoten identisch zu denen von $G$ ist, jedoch nur die ausgewählten Kanten $E'$ beinhaltet. Alle anderen Kanten werden dabei ignoriert.

### 1.5.1 Gültige Auswahl

Immer, wenn ein Paket fertig bearbeitet wurde, soll es möglich sein, den Schritt so zu vollziehen, dass alle Pakete in ihre zurzeit ausgewählte Richtung geworfen werden und der dadurch entstehende Zustand gültig ist. Dies ist genau dann der Fall, wenn jeder Knoten, von welchem ein Paket losgeworfen wird, auch ein anderes Paket zurück bekommt. Dadurch befindet sich danach auf jedem Haus wieder ein Paket, was zwingend notwendig ist. Der entstehende Zustand ist also gültig, wenn sowohl der Eingangsgrad als auch der Ausgangsgrad von jedem Knoten $u \in V$ in $G'$ beide entweder 1 oder 0 sind. Größer als 1 darf weder der Eingangs- noch der Ausgangsgrad werden, denn ein Haus darf nicht mehrere Pakete werfen oder empfangen.

Betrachtet man die starke Zusammenhangskomponente als Teilgraphen $G''$ von $G'$, in welcher ein Knoten $u$ enthalten ist, dann sind alle Knoten in $G''$ von $u$ aus nur über die ausgewählten Kanten erreichbar und jeder Knoten $v \in G''$ hat, der Bedingung für einen gültigen Schritt zufolge, den Eingangsgrad 1 sowie den Ausgangsgrad 1. Da $G''$ ebenfalls stark zusammenhängend ist, gibt es einen Eulerkreis in $G''$. Da der Eingangsgrad von allen Knoten 1 ist, beinhaltet der Eulerkreis jeden Knoten von $G'$ genau ein Mal. Daher liegt jeder Knoten $v \in V$, welcher eine ausgewählte Kante hat, auf einem Zyklus, der Knoten aus $V$ nur über die ausgewählten Kanten miteinander verbindet. Aus diesem Grund werden beim Bearbeiten eines Paketes Zyklen gesucht.

### 1.5.2 Bearbeiten eines Paketes

Wenn ein Paket $q$ bearbeitet wird, ist es möglich, dass bereits eine ausgehende Kante von $c(q)$ ausgewählt wurde. In diesem Fall wird das Paket übersprungen und das nächste wird bearbeitet. Anderenfalls wird versucht, eine Kante für $c(q)$ zu finden, die ausgewählt werden kann. Damit der Zustand nach dem Bearbeiten von $q$ wieder gültig ist, müssen auch Kanten von anderen Paketen ausgewählt werden. Keine Auswahl einer Kante von zuvor darf wieder rückgängig gemacht werden.

Aufgrund der „Zyklus-Eigenschaft“ aus Abschnitt 1.5.1 werden, wenn der Knoten $u$ bearbeitet wird, Zyklen in $G$ gesucht, welche $u$ beinhalten. Der Zyklus soll so wenig Kanten $e = (v, w)$ wie möglich beinhalten, durch die das Paket $z(v)$ in eine Richtung geworfen wird, sodass sich seine Distanz zum Ziel vergrößert. Solche Kanten werden als „schlecht“ bezeichnet. Eine Kante $e = (v, w)$ ist also schlecht, wenn $d(v, z(v)) < d(w, z(v))$.

Bei der Bearbeitung von $u$ wird daher zunächst versucht, Zyklen mit $u$ zu finden, die keine schlechte Kante beinhalten. Sollte es keinen solchen geben, dann werden Zyklen mit $u$ mit maximal einer schlechten Kante gesucht, danach mit zwei schlechten Kanten usw., bis irgendwann passende Zyklen gefunden wurde. Alle passenden Zyklen, wovon diejenigen rausgesucht wurden, die nach einem bestimmten Kriterium am besten sind (siehe Abschnitt 1.7), werden weiterverwendet. Sollte immer noch kein Zyklus gefunden worden sein, nachdem der Zyklus $m$ schlechte Kanten enthalten durfte, dann kann es gar keinen Zyklus mit $u$ mit der aktuellen Kantenauswahl geben, weshalb die Bearbeitung von $u$ beendet und mit dem nächsten Paket fortgefahren wird. Die Anzahl der schlechten Kanten, die bei der Zyklensuche verwendet werden darf, wird $s$ genannt.

### 1.5.3 Finden von Zyklen

Ein Zyklus, der den Knoten $u$ und $s$ schlechte Kanten enthält, wird folgendermaßen gefunden:

Rekursiv werden die Knoten aufgerufen, zunächst der Knoten $u$, da dieser auf jeden Fall im Zyklus enthalten sein muss. Der gerade aufgerufene Knoten wird $v$ genannt. Es werden alle ausgehenden Kanten $e \in E$, $e = (v, w)$ aus $v$ gesucht. Diese Kanten werden nach der Kantenpriorität sortiert, siehe Abschnitt 1.6. Die Priorität $p(e)$ der Kante $e$ besagt, wie wichtig es ist, dass die Kante $e$ im Zyklus enthalten ist und ausgewählt wird. Die Kanten mit der höchsten Priorität erscheinen in der Sortierung vorne, die mit der kleinsten Priorität hinten. Schlechte Kanten haben immer die kleinste Priorität, denn sie sollen möglichst nur verwendet werden, es nicht anders geht. Dieser Sortierung nach wird jetzt die erste Kante $e = (v, w)$ mit der höchsten Priorität genommen. Sie wird als ausgewählte Kante von $v$ markiert. Rekursiv wird dann der Knoten $w$ aufgerufen. Für ihn gilt derselbe Ablauf. Wenn der abgeschlossen ist, wird die Auswahl der Kante $e$ wieder entfernt. Danach wird dasselbe für alle anderen ausgehenden Kanten aus $v$ getan. Dann ist der Aufruf von $v$ beendet. Eine Kante darf nicht genommen werden, wenn sie schlecht ist und bereits $s$ schlechte Kanten im aktuellen Zyklus enthalten sind. Es muss daher gespeichert werden, wie viele schlechte Kante in der Rekursion bereits genommen wurden. Wichtig ist zudem, dass keine schlechte Kante $e = (v, w)$ genommen werden darf, wenn $d(c(v)) = d_{max}$ oder wenn $d(c(v)) = d_{max} - 1$. Diese Bedingung wird strikt eingehalten, damit die Distanz keines Paketes auf die Maximaldistanz oder gar höher erhöht wird und damit die Pakete, die besonders weit von ihrem Ziel entfernt sind, bevorzugt behandelt werden.

Wenn ein Knoten $v$ rekursiv aufgerufen wird, er jedoch bereits eine ausgewählte ausgehende Kante $e = (v, w)$ besitzt (egal ob diese durch die Bearbeitung eines vorherigen Knotens oder durch die aktuelle Rekursion verursacht wurde), dann wird er übersprun-

gen, denn er kann keine zweite ausgewählte Kante besitzen.

Wenn der erste Knoten $u$, von welchem die Rekursion gestartet wurde, erneut aufgerufen wird, dann wurde ein Zyklus gefunden. Wenn der Zyklus einem bestimmten Kriterium zufolge, siehe Abschnitt 1.7, besser ist als die bereits gespeicherten Zyklen, dann werden sie alle wieder entfernt und nur der gerade gefundene Zyklus wird hinzugefügt. Dasselbe passiert, wenn zuvor noch gar kein Zyklus gefunden wurde, denn dann kann er mit gar keinem anderen Zyklus verglichen werden. Ist der gefundene Zyklus genauso gut wie die gespeicherten Zyklen, dann wird er zu ihnen hinzugefügt. Ist er schlechter als die gespeicherten Zyklen, dann passiert nichts weiter. Es werden auch hier leicht unterschiedliche Kriterien verwendet, um in möglichst vielen Fällen gute Ergebnisse zu liefern, sie können alle durchprobiert werden.

Um die Laufzeit zu verringern, wird immer, wenn ein Knoten $v$ rekursiv aufgerufen wird, überprüft, ob der Startknoten $u$ überhaupt noch erreicht werden kann, oder ob durch die aktuelle Auswahl der Kanten jeder Weg zurück zum Start „blockiert“ wird. Dafür wird ein *Flood Fill*-Algorithmus ausgeführt, welcher nur Kanten entlanggeht, welche auch von der Rekursion genommen werden dürfen. Dazu gehören keine schlechten Kanten $e = (v, w)$, wenn $d(c(v)) \geq d_{max} - 1$. Außerdem werden Knoten, die bereits eine ausgewählte Kante besitzen, nicht aufgerufen, denn sie können keine weitere ausgewählte Kante bekommen.

Durch die Sortierung der Kanten nach ihrer Prioritäten wird als erstes entlang der „besseren“ Kanten gegangen. Dadurch erscheinen Zyklen, welche ausgehend von $u$ die besseren Kanten entlanggehen, weiter vorne in den gespeicherten Zyklen, wenn es mehrere Zyklen gibt, die nach dem Kriterium als gleich gut bewertet wurden. Die ausgewählte Kante des ersten Knotens $u$ hat dabei den größten Einfluss, an welcher Stelle der Zyklus auftaucht, denn sie wird als erstes von der Rekursion abgelaufen.

### 1.5.4 Reihenfolge der Bearbeitung von Paketen

Durch die Bearbeitung eines Paketes $u$ wurden alle einem bestimmten Kriterium zufolge besten Zyklen gespeichert. Da sie alle als gleich gut bewertet wurden, wird der Reihenfolge nach jeder von ihnen einmal genommen und seine Kanten werden erneut ausgewählt. Diese Auswahl von Kanten wird verwendet, um das nächste Paket $v$ zu bearbeiten. Dabei werden Zyklen mit $v$ gefunden und gespeichert, für jeden davon wird wieder das nächste Pakete bearbeitet usw. Wenn bei der Bearbeitung eines Paketes $q$ keine Zyklen gefunden werden können, z. B. weil alle angrenzenden Häuser von $c(q)$ bereits ausgewählte Kanten haben oder weil $c(q)$ bereits eine ausgewählte Kante hat, dann werden keine weiteren Kanten ausgewählt und es wird einfach das nächste Paket bearbeitet.

Nachdem alle Pakete bearbeitet wurden, werden alle ausgewählten Kanten durchgeführt, d. h., für jede ausgewählte Kante $e = (u, v)$ wird das Paket $z(u)$ auf das Haus $v$ geworfen, wodurch sich die Koordinaten der Pakete ändern. Dadurch entsteht ein neuer Zustand. Es ist möglich, dass mehrere Zustände entstehen, da nach jedem bearbeiteten Paket jeder beste gefundene Zyklus einmal verwendet wird, um die nächsten Pakete zu bearbeiten. Von diesen Zustände werden auch die besten gespeichert. Sie werden anhand der Kriterien in Abschnitt 1.8 verglichen. Wird ein besserer Zustand als die zuvor

gespeicherten gefunden, dann werden diese entfernt, bevor der neue Zustand eingefügt wird.

### 1.5.5 Zeitabschätzung

Eine (grobe) Zeitabschätzung darf natürlich nicht vernachlässigt werden, auch wenn es bei diesem Problem mehr darum geht, möglichst wenig Schritte für eine Lösung zu benötigen, als darum, einen möglichst schnell laufenden Algorithmus zu entwickeln. Sofern das Programm noch innerhalb einer erträglichen Zeitspanne fertig wird, ist es wichtiger, einen möglichst kurzen Plan zu finden. Außerdem ist es eher unwichtig, wenn nur für ein konstruiertes Beispiel die benötigte Zeit sehr lang ist. Es geht vor allem um die gegebene Paketverteilung in Amacity. Somit soll eher sehr grob ein *Average Case* oder „Normal Case“, anstelle eines *Worst Cases* betrachtet werden, auch basierend auf den Ergebnissen, die in der Praxis vom Algorithmus geliefert werden.

Wenn ein Zyklus gesucht wird, dann gibt es vom Startknoten aus maximal vier weitere Knoten, die er rekursiv aufruft. Alle anderen aufgerufen Knoten $v$ rufen zwar auch wiederum maximal vier weitere Knoten auf, jedoch gibt es mindestens einen davon, dessen Aufruf in konstanter Zeit wieder beendet wird. Dies ist der Knoten aus der Richtung, aus der $v$ aufgerufen wurde. Außerdem muss der Flood Fill-Algorithmus betrachtet werden, welcher bei dem Aufruf eines Knotens durchgeführt wird. Er benötigt im schlechtesten Fall die Zeit $\mathcal{O}(|V| + |E|)$. Da $|E| = 4m - 4n$ und $|V| = m$, entspricht dies $\mathcal{O}(m)$. Somit kann zum Suchen eines Zyklus mit einem festen $s$ von einem Knoten aus die Zeit $\mathcal{O}(3^m \cdot m)$ nicht überschritten werden. Sucht man von einem Knoten aus für jedes mögliche $s$ einen Zyklus, dann wird die Zeit auf $\mathcal{O}(3^m \cdot m^2)$ erhöht. Diese Laufzeit wird normalerweise jedoch sehr stark eingeschränkt und liegt sehr weit unter dieser oberen Schranke. Dies hat einige Gründe. Zum einen wird mit $s = 0$ begonnen, so ist die Rekursionstiefe meistens sehr gering und erreicht nie wirklich $m$, während gleichzeitig von jedem Knoten aus *jede* ausgehende Kante abgelaufen wird. Außerdem werden durch bereits ausgewählte Kanten viele Knoten „gesperrt“, sie können keine weiteren Kanten auswählen und dementsprechend auch keine anderen Knoten rekursiv aufrufen. Zudem sorgt der Aufruf des Flood Fill-Algorithmus, welcher in die obere Schranke zusätzlich den Faktor $m$ hineinbringt, in den meisten Fällen für eine Verbesserung der benötigten Zeit, da durch ihn verhindert wird, unnötig Gebiete abzusuchen. Lässt man das Programm laufen, so erkennt man, dass bei verschiedenen Paketverteilung auch die benötigte Zeit sehr stark variieren kann, siehe Beispiele. Keine getestete Verteilung sorgte jedoch dafür, dass die angegebene Laufzeit irgendwie annäherungsweise erreicht wird. Sie hängt hauptsächlich von der eigentlichen Paketverteilung ab.

Jeder beste gefundene Zyklus (für die Kriterien siehe Abschnitt 1.7) wird einmal verwendet, um das nächste Paket zu bearbeiten. Möchte man allerdings nur den allerersten möglichen Zustand finden, dann reicht es, bei der Bearbeitung eines Paketes immer nur den ersten Zyklus zu nehmen und damit fortzufahren. Somit würde bei der obigen Laufzeit noch ein weiteres Mal der Faktor $m$ dazu kommen, wenn man die gesamte Laufzeit für das Finden eines Zustandes bekommen möchte. Anderenfalls muss jeder beste gefundene Zyklus ausprobiert werden, um das nächste Paket einmal zu bearbeiten und

auch dort wieder Zyklen zu finden usw. Das Testen des Algorithmus hat gezeigt, dass die Anzahl der besten gefundenen Zyklen immer sehr gering ist. So wird oft gar kein Zyklus gefunden, z. B. wenn ein Knoten bereits eine ausgewählte Kante hat, die durch die vorherigen Zyklen gefunden wurde. Ansonsten ist die Zahl sehr oft nur 1, manchmal auch 2 oder 3, nur selten mehr. Dafür sorgen die Kriterien aus Abschnitt 1.7.

## 1.6 Kantenpriorität

Die Funktion $p$ ordnet jeder Kante eine Priorität zu. $p(e)$, wobei $e \in E$ mit $e = (u, v)$, besagt, wie „wichtig“ es ist, das Paket auf dem Haus des Knotens $u$ auf das Haus des Knotens $v$ weiterzuwerfen. Je höher der Wert, desto wichtiger ist dies. Kanten mit höherer Priorität werden bei der Rekursion als erstes entlanggelaufen. Es gibt zwei mögliche Definitionen der Priorität.

### 1.6.1 Abstand zum Ziel bei guten Kanten

Wenn durch den Wurf von $u$ nach $v$ sich der Abstand $d(z(u))$ vergrößern würde, ist $p(e) = 0$, da ein solcher Wurf „schlecht“ ist. Wenn sich der Abstand des Paketes auf $u$ hingegen verkleinern würde, wird der Wurf „gut“ genannt.

Ändert sich die Zeile von $z(u)$ durch den Wurf von $u$ nach $v$ und würde sich der Abstand verkleinern, ist $p(e) = d_x(z(u))$. Ändert sich hingegen die Spalte von $z(u)$ durch den Wurf, während sich der Abstand verkleinert, ist $p(e) = d_y(z(u))$. In diesen Fällen gibt die Priorität also an, wie viele Zeilen bzw. Spalten das Paket noch in die entsprechende Richtung geworfen werden muss.

Die mathematische Definition der Priorität lautet demnach folgendermaßen, wobei $e = (u, v)$, $q = z(u)$, $u = (x_u, y_u)$ und $v = (x_v, y_v)$:

$$p(e) = \begin{cases} 0 & \text{wenn } d(q) < d(v, q) \\ d_x(q) & \text{wenn } d(q) > d(v, q) \text{ und } x_u \neq x_v \\ d_y(q) & \text{wenn } d(q) > d(v, q) \text{ und } y_u \neq y_v \end{cases}$$

Muss ein Paket bspw. eine Spalte Richtung Osten, aber 8 Zeilen Richtung Norden, so wäre die Priorität für die Kante, welche angibt, dass das Paket nach Norden geworfen wird, 8, die Priorität der Kante, bei der das Paket nach Osten geworfen wird, lediglich 1. Es ist also wichtiger, das Paket erst einmal nach Norden zu werfen. Das Werfen nach Osten in die richtige Spalte benötigt wahrscheinlich weniger Schritte und kann später auch noch gemacht werden. Außerdem kann, wenn es nach Osten geworfen wird, sodass er nur noch 8 Zeilen Richtung Norden geworfen werden muss, um am Ziel anzukommen, es schneller passieren, dass es in eine falsche Richtung geworfen werden muss, denn es hat dann drei schlechte Kanten. Es ist einfacher, das Paket entlang einer guten Kante zu werfen, wenn es mehrere davon gibt.

Mit dieser Definition liefert der Algorithmus für die gegebene Paketverteilung von Amacity den kürzesten Plan.

### 1.6.2 Abstand zum Ziel

Bei dieser Definition können die Prioritäten auch negativ sein. Hier wird angenommen dass es „schlechter“ ist, wenn ein Paket in eine falsche Richtung geworfen wird, wenn das Ziel weit in der entgegengesetzten Richtung liegt, als wenn das Ziel in der entgegengesetzten Richtung näher ist. Es ergibt sich die folgende Definition für die Kante $e = (u, v)$ mit $q = z(u)$, $u = (x_u, y_u)$ und $v = (x_v, y_v)$:

$$p(e) = \begin{cases} -d_x(q) & \text{wenn } d(q) < d(v,q) \text{ und } x_u \neq x_v \\ -d_y(q) & \text{wenn } d(q) < d(v,q) \text{ und } y_u \neq y_v \\ d_x(q) & \text{wenn } d(q) > d(v,q) \text{ und } x_u \neq x_v \\ d_y(q) & \text{wenn } d(q) > d(v,q) \text{ und } y_u \neq y_v \end{cases}$$

## 1.7 Auswahl der Zyklen

Bei der Zyklensuche muss eine Auswahl der gefundenen Zyklen getroffen werden, denn jeder gespeicherte Zyklus wird einmal verwendet, um das nächste Paket zu bearbeiten, wobei wieder mehrere Zyklen gefunden werden können usw. Würde man jeden einzelnen gefundenen Zyklus durchprobieren, könnte es passieren, dass das Programm viel zu viel Zeit benötigt. Ein Zyklus $Z$ wird hier als Menge der Kanten des Zyklus verstanden.

### 1.7.1 Länge des Zyklus

Die erste Möglichkeit, die Zyklen auszuwählen, ist, nur die längsten Zyklen, also die mit den meisten Knoten bzw. Kanten zu verwenden. Dies ist sinnvoll, da nach einer Zyklensuche jeder der gefundenen Zyklen dieselbe Anzahl an schlechten Kanten $s$ besitzt. Dadurch, dass die längsten Zyklen verwendet werden, gibt es bereits so viele gute Kanten wie möglich.

Nach diesem Kriterium ist ein Zyklus $Z_1$ besser als der Zyklus $Z_2$, wenn gilt:

$$|Z_1| > |Z_2|$$

Bei den anderen möglichen Kriterien kann es passieren, dass der Zyklus weniger gute Kanten beinhaltet und beim Bearbeiten von späteren Paketen mehr schlechte Kanten ausgewählt werden müssen.

### 1.7.2 Summe der Prioritäten aller Kanten

Es ist auch möglich, bei der Rekursion die Prioritäten von allen Kanten, die ausgewählt werden, zusammenzuaddieren. Es werden dann die Zyklen gespeichert, die die höchste Summe besitzen.

Wird die erste Definition der Priorität verwendet, dann hat auch die Zykluslänge einen Einfluss auf dieses Kriterium. Beispielsweise kann hier ein langer Zyklus nur mit 1er-Prioritäten besser sein als ein Zyklus mit zwei Knoten, wobei die Kanten die Prioritäten 3 besitzen. Wird die zweite Definition verwendet, dann haben vor allem schlechte Kanten einen hohen negativen Einfluss auf die Bewertung eines Zyklus.

Der Zyklus $Z_1$ ist diesem Kriterium zufolge besser als der Zyklus $Z_2$, wenn gilt:

$$\sum_{e \in Z_1} p(e) > \sum_{e \in Z_2} p(e)$$

### 1.7.3 Durchschnitt der Prioritäten aller Kanten

Bei diesem Kriterium wird das arithmetische Mittel der Prioritäten aller Kanten verwendet. Dadurch hängt die Bewertung eines Zyklus gar nicht mehr von der Länge des Zyklus ab.

Nach diesem Kriterium ist als der Zyklus $Z_1$ besser als der Zyklus $Z_2$, wenn gilt:

$$\frac{1}{|Z_1|} \sum_{e \in Z_1} p(e) > \frac{1}{|Z_2|} \sum_{e \in Z_2} p(e)$$

## 1.8 Auswahl der Zustände

Außerdem muss es eine Möglichkeit geben, die durch einen Schritt entstandenen Zustände zu vergleichen, damit mit dem „besten“ Zustand fortgefahren werden kann. Wie bereits beschrieben wird versucht, einen möglichst guten Zustand auszuwählen. Da jedoch nicht jede Paketverteilung (siehe Beispiele) durch dieselbe Auswahl des Zustandes in möglichst wenigen Schritten gelöst werden kann, gibt es zwei verschiedene mögliche Entscheidungen.

Für alle Strategien gilt, dass sie natürlich nicht immer zum besten Ergebnis führen müssen, da es möglich ist, dass durch den ausgewählten Zustand die Pakete so verteilt werden, dass es noch mehr Schritte benötigt, alle Pakete an ihre Ziele zu befördern, als wenn man einen Zustand genommen hätte, welcher dem Kriterium zufolge schlechter ist.

### 1.8.1 Den ersten Zustand verwenden

Es ist möglich, einfach den ersten Zustand, der gefunden wird, zu verwenden, denn während der Rekursion wurden immer die Kanten mit der höchsten Priorität als erstes aufgerufen. Außerdem wurden zunächst Zyklen von den Paketen mit der größten Distanz aus zuerst gesucht. Deshalb befinden sich in den möglichen Zuständen immer ganz vorne diejenigen, bei denen die wichtigen Pakete (mit größerer Distanz) entlang ihren wichtigen Richtungen (mit größerer Priorität) geworfen wurden. Es sind also die „wichtigen“ Zustände im Hinblick auf die Priorität der Richtungen und auf die Pakete mit größerer Distanz ganz vorne und werden daher durch dieses Auswahlkriterium ausgewählt.

Diese Auswahl hat einen wichtigen Vorteil in Hinblick auf die Laufzeit: Sobald bei der Suche ein einziger Zustand gefunden wurde, kann sie abgebrochen werden. Es müssen dann nicht alle weiteren gespeicherten Zyklen verwendet werden, um die nächsten Pakete zu bearbeiten. Es reicht also, immer den ersten besten gefundenen Zyklus zu nehmen, um das nächste Paket zu bearbeiten. Wenn bei dem Bearbeiten vieler Pakete mehrere beste Zyklen gefunden werden, kann dieses Kriterium daher deutlich schneller sein.

Diese Zustandsauswahl ist für die gegebene Paketverteilung für Amacity die, bei der am wenigsten Schritte benötigt werden.

### 1.8.2 Kleinste Maximal- und Gesamtdistanz

Die zweite Variante besteht darin, zunächst den Zustand auszuwählen, welcher die Maximaldistanz verkleinert. Wenn es keinen gibt oder wenn mehrere Zustände die Maximaldistanz verkleinern, dann wird davon der Zustand gewählt, welcher die kleinste Gesamtdistanz hat. Erst wenn auch hier nicht genau entschieden werden kann, wird der Zustand genommen, der sich am weitesten vorn befindet.

Der Sinn dieser Auswahl ist die Beobachtung, dass wenn in jedem Schritt die Maximaldistanz verkleinert wird, auf jeden Fall eine Lösung mit der Mindestschrittanzahl erstellt werden kann. Bei mehreren Zuständen wird dann derjenige ausgewählt, welcher die Gesamtdistanz möglichst stark verkleinert, weil dann angenommen wird, dass dadurch ebenfalls weniger Schritte benötigt werden. Leider ist auch dieses Kriterium natürlich keine Garantie dafür, dass der Plan in der kleinsten möglichen Schrittanzahl erstellt wird. Für die gegebene Paketverteilung für Amacity z. B. sorgt das erste Kriterium, bei dem der erste Zustand verwendet wird, für eine geringere Schrittanzahl.

## 1.9 Strategien

Es gibt drei verschiedene Strategien, die verwendet werden können, um einen Plan zu generieren. Zusätzlich kann noch ausgewählt werden, welche der Definitionen zur Kantenpriorität, wie die Zyklen und wie die Zustände ausgewählt werden.

### 1.9.1 Jede Kombination verwenden

Bei dieser Strategie werden alle Kombinationen der Definition der Kantenpriorität, der Auswahl der Zyklen und der Auswahl der Zustände einmal ausprobiert. Für jede dieser maximal $2 \cdot 3 \cdot 2 = 12$ Kombinationen wird ein Plan erstellt. Der Algorithmus liefert dann den kürzesten Plan, der generiert werden konnte.

### 1.9.2 In jedem Schritt die beste Kombination nehmen

Eine weitere Strategie ist, für jeden Schritt einzeln jede Kombination der Definition der Kantenpriorität und der Auswahl der Zyklen auszuprobieren. Nachdem alle dadurch entstandenen Zustände gefunden wurden, wird der beste nach dem Kriterium für die Auswahl des Zustandes genommen.

Das Verwenden der ersten Zustandsauswahl wäre hier sinnlos, da immer der erste Zustand ausgewählt werden würde. Dies wäre immer einer, der durch die erste Kombination entstanden ist. Somit könnte auch einfach nach der ersten Strategie mit der ersten Auswahl des Zustandes ein Plan erstellt werden.

### 1.9.3 Zufällige Auswahl

Es ist auch möglich, dass für jeden Schritt eine neue zufällige Kombination erstellt wird. Bei den anderen Strategien wird bei jedem Programmdurchlauf immer dieselbe Lösung generiert, hier ist es möglich, dass irgendwann eine bessere Lösung generiert wird, indem bei jedem Schritt unterschiedliche, zufällige Kombinationen verwendet werden. Es

kann aber natürlich auch sein, dass auch nach einer langen Zeit keine besseren Lösungen entstehen.

### 1.9.4 Schritte rückgängig machen

Dieser Abschnitt ist keine weitere Strategie, er stellt nur eine weitere mögliche Einstellung dar, die für alle anderen Strategien getroffen werden kann, um durch eine größere Suche möglicherweise kürzere Pläne generieren zu können:

Es kann vorkommen, dass nach Beenden eines Schrittes keine gute Auswahl des Zustandes getroffen wird, auch wenn er nach dem verwendeten Kriterium als der beste angesehen wurde. Eine Schrittfolge war möglicherweise nicht optimal, wenn durch irgendeinen Schritt die aktuelle Maximaldistanz nicht verringert wird. Dies hätte unter Umständen anders sein können, wenn bei diesem oder bei weiteren Schritten zuvor die „nächstbesseren“ oder „gleich guten“ Zustände, den Kriterien in Abschnitt 1.8 zufolge, ausgewählt wurden.

Um dies zu beachten, werden bei der Auswahl des Zustandes gleich die zwei besten ermittelt, diese Zahl ist auch veränderbar, um eine noch größere Suche zu ermöglichen. Für die erste mögliche Auswahl des Zustandes wäre dies einfach der nächste gespeicherte Zustand hinter dem ersten. Bei der zweiten Auswahl würden nur Zustände ausgewählt werden, die die gleiche Maximal- und Gesamtdistanz haben und hinter dem ersten gespeicherten Zustand stehen.

Danach werden zunächst alle folgenden Schritte vom ersten Zustand aus fortgeführt. Wenn durch diesen ersten Zustand die Maximaldistanz nicht verkleinert wurde, dann werden erneut die nächsten Schritte ausgeführt, diesmal jedoch ausgehend vom zweiten Zustand, sofern es einen gibt. Sollte eingestellt sein, dass noch mehr beste Zustände ermittelt werden, dann werden für jeden davon die nachfolgenden Schritte ausgeführt. Es wird dann die Schrittfolge als Ergebnis genommen, bei der insgesamt am wenigsten Schritte benötigt werden.

Natürlich kann es sein, dass auch durch die Verwendung eines anderen Zustandes die Anzahl an benötigten Schritten nicht sinkt. Vielleicht ändert sich dies jedoch, wenn man noch mehr Schritte zurückgeht und die nächstbesseren Zustände dort verwendet, um die weiteren Schritte zu ermitteln. Aus diesem Grund merkt sich der Algorithmus nach jedem verwendeten Zustand, wie viele Schritte er noch zurückgehen muss. Wenn die Maximaldistanz direkt nach irgendeinem Zustand nicht verkleinert wird, dann ist diese Zahl irgendein festgelegter Wert. Bei jedem zurückgegangenen Schritt wird die Anzahl der zurückzugehenden Schritte um eins verringert. Sollte sie größer als 0 sein, nachdem der erste Zustand verwendet wurde, dann muss noch der nächste Zustand verwendet werden, um die weitere Schrittfolge erneut zu berechnen. Wenn mehr als zwei beste Zustände ermittelt werden sollen, können danach natürlich noch weitere davon verwendet werden. Dabei wird dann wieder ermittelt, wie viele Schritte der Algorithmus noch zurückgehen muss. Ist diese Zahl nach der Verwendung irgendeines Zustandes geringer als bei einem anderen, dann wird die geringere Zahl verwendet.

Der festgelegte Wert, der angibt, wie viele Schritte der Algorithmus zurückgehen muss, wenn sich die Maximaldistanz nicht verkleinert hat, kann z. B. 3 sein. Er sollte, wie die

folgende Laufzeitanalyse zeigt, nicht viel höher sein.

## 1.10 Zeitabschätzung

Werden keine Schritte rückgängig gemacht, dann muss der Algorithmus für das Generieren eines Plans durch Strategie 1 und 3 für eine Kombination mit insgesamt $k$ Schritten $k$ Mal einen Zustand finden, eine Zeitabschätzung davon wird in Abschnitt 1.5.5 durchgeführt. Bei Strategie 2 ändert sich die obere Schranke auch nicht, weil die Anzahl der möglichen Kombinationen konstant ist.

Werden die Schritte zusätzlich noch rückgängig gemacht, dann steigt die Laufzeit noch einmal deutlich. Sie ist dann vor allem davon abhängig, wie oft die Maximaldistanz nicht verringert wird. Diese Zahl wird als $k_m$ bezeichnet. Die Anzahl der Schritte, die der Algorithmus zurückgehen muss, wird $k_z$ genannt, die Anzahl der maximal verwendeten Zustände, um weitere Schritte zu generieren, ist $k_s$. Dann müssen insgesamt höchstens $k_s^{k_m \cdot k_z}$ Zustände generiert werden. Daran ist erkennbar, dass $k_s$ und $k_z$ nur sehr kleine Zahlen sein sollten, damit das Programm in akzeptabler Zeit terminieren kann.

## 1.11 Beispiel

Der Algorithmus wird jetzt anhand des $3 \cdot 3$-Beispiels aus der Aufgabenstellung erläutert. Es wird die erste Strategie verwendet, wobei die erste Definiton der Kantenpriorität, die erste Möglichkeit zur Auswahl der Zyklen und die erste Möglichkeit zur Auswahl des Zustandes verwendet werden. Es werden keine Schritte rückgängig gemacht. Der Graph $G$ davon und der Zustand zu Beginn ist in Abbildung 1 zu sehen.

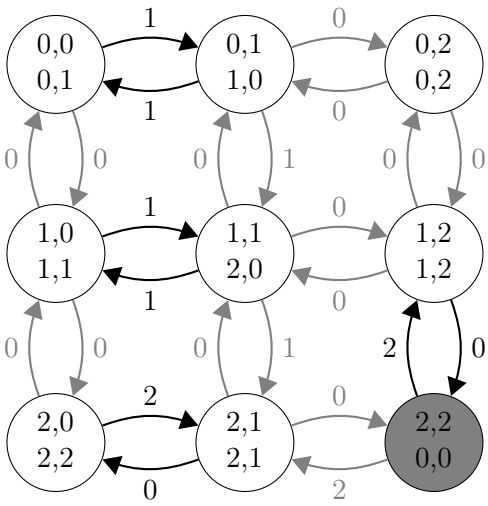


Abbildung 1: Der Graph und der Zustand zu Beginn für das 3·3-Beispiel. Die ersten Zeile in einem Knoten gibt an, an welchen Koordinaten sich das Haus befindet. In der zweiten Zeile steht, welches Paket sich auf dem zugehörigen Haus befindet. An den Kanten steht die zugewiesene Kantenpriorität. Der grau markierte Knoten befindet sich am weitesten von seinem Ziel entfernt. Die dunkleren Kanten zeigen die ausgewählten Kanten, nachdem alle Zyklen gefunden wurden.

Das Paket $(0, 0)$ auf Haus $(2, 2)$ befindet sich am weitesten von seinem Ziel entfernt, von ihm aus wird als erstes Zyklus mit keiner schlechten Kante gesucht. Einen solchen Zyklus gibt es jedoch nicht. Erst bei einer schlechten Kante findet die Rekursion einen Zyklus, indem sie den Weg nach Norden und dann nach Süden nimmt. Dies ist der erste und einer der längsten möglichen Zyklen. Dann wird der nächste Zyklus für das Paket $(1, 0)$ auf Haus $(0, 1)$ mit keiner schlechten Kante gesucht. Ein solcher Zyklus existiert. Als nächstes wird vom Paket $(2, 0)$ auf Haus $(1, 1)$ ein Zyklus gesucht, auch hier gibt es einen, der keine schlechte Kante entlanggeht. Als letztes wird für $(2, 2)$ auf Haus $(2, 0)$ ein Zyklus gesucht. Ein solcher Zyklus ist nur mit einer schlechten Kante möglich. Alle anderen Pakete werden übersprungen, da sie entweder schon in einem Zyklus enthalten sind oder weil kein Zyklus mehr gefunden werden kann. Da immer der erste Zustand genommen wird, ist die Zyklensuche beendet und der Zustand wird verwendet, um die Pakete entlang der ausgewählten Kanten zu werfen. Es werden bei diesem kleinen Beispiel nur paarweise Würfe zwischen je zwei Häusern verwendet.

In Abbildung 2 wird der Graph mit dem durch den ersten Schritt entstandenen Zustand gezeigt.

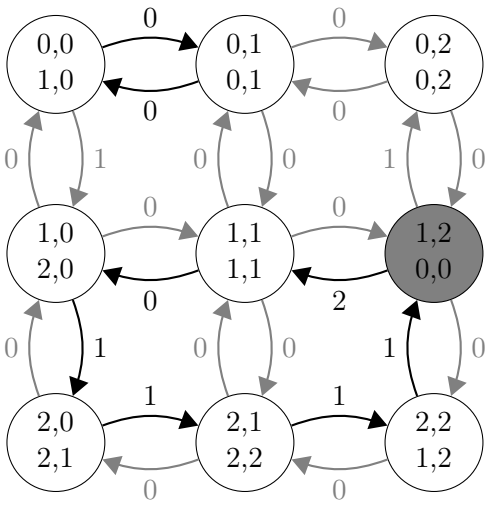


Abbildung 2: Der Graph und der Zustand nach dem ersten Schritt für das $3 \cdot 3$-Beispiel.

Es wird wieder als erstes ein Zyklus vom Paket $(0, 0)$ auf Haus $(1, 2)$ aus gesucht. Mit keiner schlechten Kante ist dies nicht möglich. Mit einer schlechten Kante ist der längste gefundene Zyklus 6 Knoten lang, er wird weiterverwendet. Als nächstes wird vom Paket $(1, 0)$ auf Haus $(0, 0)$ aus ein Zyklus gesucht. Es wird erst einer mit zwei schlechten Kanten gefunden, dabei wird von keinem einzigen Knoten aus eine gute Kante ausgewählt. Solche Fälle könnten zwar einfach im Algorithmus verhindert werden, jedoch wird dadurch die Schrittanzahl für die gegebene Paketverteilung von Amacity erhöht, weshalb sie trotzdem durchgeführt werden.

Der entstandene Zustand ist in Abbildung 3 zu sehen.

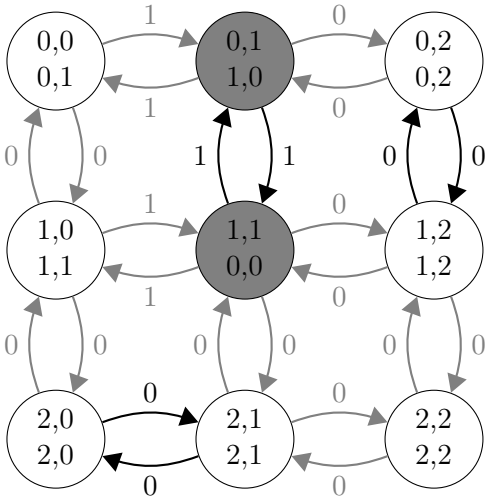


Abbildung 3: Der Graph und der Zustand nach dem zweiten Schritt für das $3 \cdot 3$-Beispiel.

Die Pakete mit dem größten Abstand zum Ziel sind hier $(1,0)$ und $(0,0)$ auf den Häusern $(0,1)$ und $(1,1)$. Von $(1,0)$ aus wird als erstes ein Zyklus gesucht, denn es befindet sich auf einem Haus, dessen Zeile einen kleineren Index hat. Wäre die Zeile dieselbe, dann würde zuerst das Paket auf dem Haus mit dem kleineren Spaltenindex genommen werden. Der gefundene Zyklus beinhaltet keine schlechte Kante, $(0,0)$ ist ebenfalls enthalten. Der nächste Zyklus wird von $(0,1)$ auf Haus $(0,0)$ aus gesucht. Von diesem Haus aus kann jedoch keine schlechte Kante genommen werden, da $d((0,1)) = d_{max} - 1$. Dasselbe gilt für das Paket $(1,1)$ auf Haus $(1,0)$. Sie bleiben beide an ihren Plätzen. Von den Paketen $(0,2)$ und $(2,0)$ auf den Häusern $(0,2)$ und $(2,0)$ aus werden wie im letzten Schritt „schlechte“ Zyklen gefunden, welche keine guten Kanten beinhalten.

Der durch diesen Schritt entstandenen Zustand wird in Abbildung 4 gezeigt.

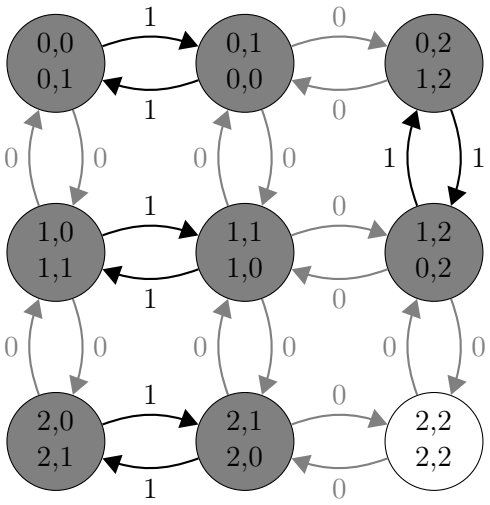


Abbildung 4: Der Graph und der Zustand nach dem dritten Schritt für das $3 \cdot 3$-Beispiel.

Alle Pakete außer $(2,2)$ haben jetzt die Distanz 1 zu ihren Zielen. Zuerst wird ein Zyklus vom Paket $(0,1)$ aus gesucht, dann $(1,2)$, $(1,1)$ und zum Schluss $(2,0)$. Durch paarweise Tausche zwischen je 2 Häusern stehen dann alle Pakete auf den Häusern, wo sie hin sollen. Diesen Zustand zeigt Abbildung 5.

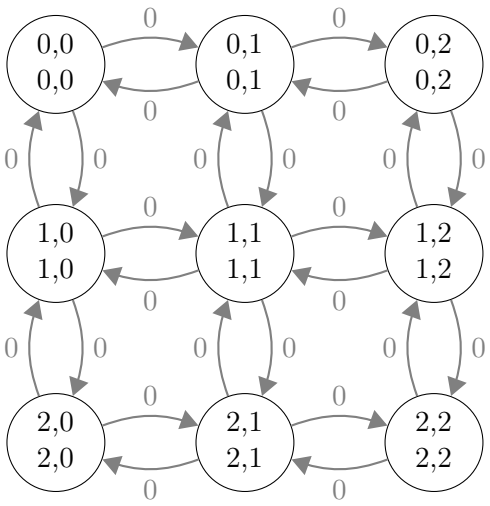


Abbildung 5: Der Graph und der Zustand nach dem vierten und letzten Schritt für das $3 \cdot 3$-Beispiel.

Bevor der erste Schritt angefangen wurde, war die Maximaldistanz $d_{max} = 4$. Die minimal mögliche Schrittanzahl für diese Paketverteilung ist daher 4, denn Pakete können in jedem Schritt ihre Distanz nur um maximal 1 verkleinern. Es wurde daher eine optimale Lösung mit minimaler Schrittzahl gefunden.

## 2 Umsetzung

Das Programm wurde in der Programmiersprache Swift 2 geschrieben und ist lauffähig auf OS X 10.9 Mavericks und neuer. 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. Bevor das Programm damit beginnt, eine Lösung zu suchen, können die „Suchparameter“, die Einstellungen, verändert werden. Dazu gehören die auszuprobierenden Strategien, die zu verwendenden Kantenprioritäten, Kriterien zur Auswahl der Zyklen und Kriterien zur Auswahl der Zustände. Zusätzlich ist es möglich, einzustellen, wie viele Schritte das Programm maximal zurück gehen kann und wie viele Zustände maximal für jeden Schritt ausprobiert werden können (dies wirkt sich nur aus, wenn das Programm mindestens einen Schritt zurück gehen kann). Während das Programm nach Lösungen sucht, können Befehle eingegeben werden, die das Programm beenden, die aktuell beste Schrittanzahl ausgeben oder die aktuell beste Lösung in einer Datei im vom Terminal ausgewählten Ordner speichern.

Außerdem wurde ein Programm geschrieben, welches Eingabe- und Ausgabedateien verwendet, um eine Lösung grafisch darzustellen. Es befindet sich mit seinem Quelltext auch im Ordner dieser Abgabe.

## 2.1 Programmstruktur

- *Task.swift* - Sucht nach Lösungen.
- *TaskReader.swift* - Liest eine Paketverteilung und die Einstellungen ein.
- *main.swift* - Startet den `TaskReader` sowie `Task` und fragt Befehle ab, die während der Ausführung eingegeben werden können. Dies geschieht in einem anderen Thread als das Finden von Lösungen.

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

- `MoveDirection` - Dies ist ein `enum` und kann die Werte `North`, `East`, `South` und `West` annehmen. Dadurch wird angegeben, in welche Richtung ein Paket geworfen wird.
- `Coordinates` - Dieser `struct` stellt Koordinaten dar und beinhaltet die Zeile und die Spalte als Integer.
- `PriorityDefinition` - Dieses `enum` beinhaltet die Werte `DistanceGoodEdges` und `DistanceAllEdges`, welche den beiden verschiedenen Definitionen der Kantenpriorität entsprechen.
- `CycleSelection` - Die Werte dieses `enums` `CycleLength`, `PrioritySum` und `PriorityAverage` stellen die drei möglichen Kriterien zur Auswahl der Zyklen dar.
- `StateSelection` - Dieses `enum` mit seinen Werten `FirstState` und `SmallestMaximumTotal` repräsentiert die möglichen Kriterien zur Auswahl des Zustandes.
- `Package` - Dieser `struct` stellt ein Paket dar.
- `Path` - Durch diesen `struct` wird ein Weg implementiert.
- `State` - Dieser `struct` repräsentiert einen Zustand.
- `Task` - Diese `class` führt die eigentliche Suche aus.

Der Graph wird nicht als ein Graph mit Knoten und Kanten implementiert, sondern seine Knoten sind als Array in `State` enthalten. Ein Knoten wird durch Koordinaten dargestellt. Eine Kante wird immer nur durch `MoveDirection` und dem Knoten, von dem die Kante ausgeht, repräsentiert.

## 2.2 Package

Ein Paket beinhaltet vier wichtige Variablen. Dies sind zunächst sein Ziel (`destination` vom Typ `Coordinates`) und seine Distanz zum Ziel (`distance` als Int).

Außerdem beinhaltet es die Richtung, in die es geworfen werden soll, `moveDirection` vom Typ `MoveDirection?`. Diese Variable entspricht der „ausgewählten Kante“ aus der

Lösungsidee. Der Unterschied dabei ist, dass diese Kante im Paket selbst statt in einem Knoten gespeichert wird. Wenn die Variable `nil` ist, bedeutet dies, dass keine Kante ausgewählt ist.

Als letztes beinhaltet ein Paket alle möglichen Kanten, die es entlanggeworfen werden kann. Zusätzlich zu den Richtungen werden noch die Prioritäten der entsprechenden Kanten gespeichert. Die Variable `possibleMoves` ist daher vom Typ `[(direction: MoveDirection, priority: Int)]`.

## 2.3 Path

Ein Weg wird später bei der Rekursion benötigt, um zu speichern, welche Knoten und Kanten den Zyklus bilden.

Ein Weg beinhaltet zunächst eine Variable namens `vertices` vom Typ `[(vertex: Coordinates, direction: MoveDirection)]`. Jedes Element im Array stellt einen Knoten und die von ihm aus ausgewählte Kante dar. Die Koordinaten entsprechen denen des Knotens bzw. Hauses. Sie stellen *nicht* das Ziel des Paketes des Knotens dar. Es muss jeder Knoten im Weg auch eine ausgehende Kante haben, bei der dann der Weg fortgeführt wird, damit ein Weg später als Zyklus verwendet werden kann.

Zudem beinhaltet der Weg eine Variable namens `prioritySum`, einen Integer. Immer, wenn ein Element zum Weg hinzugefügt bzw. entfernt wird, wird diese Summe der Prioritäten aller Kanten im Weg inkrementiert bzw. dekrementiert.

Hinzugefügt zum bzw. entfernt werden aus den Weg können Knoten und Kanten mithilfe der Methoden `append()` und `removeLast()`.

## 2.4 State

Ein Zustand beinhaltet in der Variablen `packages` als zweidimensionales Array (`[[Package]]`) alle Pakete. Dieses Array stellt die gesamte Stadt und die sich auf den Häusern befindlichen Pakete dar. Der erste Index gibt an, in welcher Zeile das Haus bzw. Paket ist, der zweite Index entspricht der Spalte.

Zusätzlich werden im Zustand die von jedem Haus bereits getätigten Würfe als `moves` gespeichert, welche verwendet wurden, um diesen Zustand zu erreichen. Die Variable ist vom Typ `[[String]]`, um die Wurfe als `String` im die Ausgabedatei notwendigen Format anzugeben. Auch hier gibt der erste Index die Zeile und der zweite Index die Spalte an.

Um Zeit zu sparen, werden vor jedem Schritt direkt die Maximal- und die Gesamtdistanz durch die Methode `calculateDistances()` ausgerechnet und in `maxDist` und `totalDist` gespeichert, damit dies nicht jedes Mal geschehen muss, wenn sie abgefragt werden. Zudem berechnet die Methode die Distanzen von allen Paketen zu ihren Zielen.

`addCurrentMovesToSolution()` fügt die ausgewählten Kanten zu `moves` hinzu und `move()` bewegt alle Pakete entlang ihren ausgewählten Kanten und setzt `moveDirection` von allen Paketen auf `nil` zurück. Die Methode `createEdges()` füllt die Arrays `possibleMoves` aller Pakete und sortiert sie nach Kantenpriorität. Dafür wird die übergebene Definition der Kantenpriorität verwendet.

### 2.4.1 `floodFillReachability()` und `isReachable()`

Diese beiden Methoden prüfen durch einen Flood Fill-Algorithmus die Erreichbarkeit eines Knotens. Dafür muss `isReachable()` mit zwei Koordinaten aufgerufen werden. Die ersten Koordinaten geben die Position des aktuellen Knotens an, die zweiten Koordinaten die des Knotens, dessen Erreichbarkeit geprüft werden soll. Die Methode erstellt ein `[[Bool]]`-Array, welches die Stadt repräsentiert. Ein Wert ist später `false`, wenn der zugehörige Knoten noch nicht erreicht wurde und `true`, wenn er erreicht werden kann.

Dann wird die rekursive Methode `floodFillReachability()` aufgerufen. Sie bekommt das Array als weiteres `inout`-Argument zu den beiden anderen Parametern des zu besuchenden Knotens und des Zielknotens. Für jeden besuchten Knoten wird der Wert im Array auf `true` gesetzt. Dadurch wird später sichergestellt, dass kein Knoten mehrmals besucht wird. Wenn irgendwann der Zielknoten erreicht wird, gibt die Methode `true` zurück. Wurden alle möglichen Kanten abgelaufen, der Knoten jedoch nicht gefunden, wird `false` zurückgegeben.

### 2.4.2 searchCycle()

Diese Methode sucht die Zyklen innerhalb eines Graphen. Als ersten Parameter bekommt sie die Koordinaten des Knotens, den die Rekursion besuchen soll. Zudem wird die Anzahl der verbleibenden schlechten Kanten übergeben, die noch genommen werden dürfen. Der dritte Parameter ist als `Path` der aktuelle „Zyklus“, welcher aber noch kein ganzer Zyklus sein muss. Als viertes bekommt die Methode ein Array mit allen gespeicherten Zyklen als `inout`-Parameter übergeben. Sie sind dem aktuellen Kriterium zur Zyklenauswahl zufolge die besten gefundenen Zyklen. Der fünfte und letzte Parameter ist ein Wert des `enum`s `CycleSelection` und gibt an, welches Kriterium zur Auswahl der Zyklen verwendet werden soll.

Zunächst wird überprüft, ob ein Zyklus gefunden wurde. Dies ist der Fall, wenn der erste Knoten im Zyklus derselbe wie der aufgerufene Zyklus ist. Es wird in diesem Fall überprüft, ob er besser oder gleich gut wie die anderen gefundenen Zyklen ist. Alle dafür benötigten Werte wurden in `Path` gespeichert. Ist der gefundene Zyklus besser als die bisherigen, dann wird die Liste geleert und der Zyklus wird hinzugefügt. Ist er gleich gut oder existiert noch gar kein gespeicherter Zyklus, dann wird er einfach an das Array angehängt. Da die Werte für alle gespeicherten Zyklen gleich sind, wird einfach der erste im Array zum Vergleich verwendet. Die Rekursion wird an dieser Stelle nicht weiter fortgesetzt.

Danach werden, wenn der aufgerufene Knoten noch keine ausgewählte Kante hat und der Anfang des Zyklus laut Flood Fill noch erreicht werden kann, alle ausgehenden Kanten aus `possibleMoves` des Knotens bzw. Paketes durchlaufen. Sie werden, wenn sie genommen werden dürfen, zum Zyklus hinzugefügt und die Rekursion wird beim nächsten Knoten fortgesetzt. Anschließend wird immer der letzte Knoten aus dem Zyklus wieder entfernt.

### 2.4.3 roundCoordinates()

Diese Methode startet die Zyklensuche von einem gegebenen Knoten aus und ruft sich für jeden gefundenen Zyklen rekursiv selbst auf, sodass der nächste Knoten in der sortierten Reihenfolge bearbeitet wird. Dafür werden zwei Parameter benötigt, dies sind ein Array mit den Koordinaten aller Knoten nach der Distanz sortiert sowie der Index des zu bearbeitenden Knotens in dieser Liste. Da die Zyklensuche ein Auswahlkriterium benötigt, muss dieses auch in `roundCoordinates()` stets als Argument übergeben werden.

Ein weiterer als `inout` markierter Parameter ist ein Array mit allen besten gefundenen Zuständen. Wenn rekursiv alle Knoten in `roundCoordinates()` bearbeitet wurden, wird nach dem Kriterium zur Zustandsauswahl, welches als weiteres Argument übergeben wird, entschieden, ob der gefundene Zustand besser ist als die zuvor gespeicherten. Für diese Überprüfung wird zunächst der Zustand, in dem diese Methode ausgeführt wird, kopiert (was bei `structs` immer automatisch geschieht) und anschließend `addCurrentMovesToSolution()`, `move()` und `calculateDistances()` auf die Kopie ausgeführt. Dadurch wird der Schritt „durchgeführt“. Ähnlich wie bei der Zyklensuche kann der gefundene Zustand hinzugefügt werden, wenn er besser oder genauso gut ist wie die zuvor gespeicherten. Außerdem wird evtl. das gesamte Array geleert.

Der letzte Parameter vom Typ `Int?` wird benötigt, um kürzere Laufzeiten zu haben. Ist er nicht `nil`, dann wird abgebrochen, sobald so viele Zustände gefunden wurden, dass sie dieser Zahl entsprechen. Der Parameter wird verwendet, wenn das erste Kriterium zur Auswahl der Zustände verwendet wird (immer den ersten gefundenen Zustand auswählen), damit nur eine begrenzte Anzahl an Zuständen gefunden wird.

## 2.5 Task

Eine `Task` bekommt direkt im Konstruktor den eingelesenen „Startzustand“ sowie die zu verwendenden Einstellungen übergeben. Zudem speichert sie die aktuell beste Lösung, die gefunden wurde. Eine Lösung ist ein einfacher Zustand, bei dem alle durchgeführten Schritte in `moves` gespeichert wurden.

### 2.5.1 run()

Durch diese Methode wird das Suchen von Lösungen gestartet. Dafür werden alle drei Strategien nacheinander durchgegangen, sofern diese den Einstellungen zufolge verwendet werden sollen. Es wird mit den entsprechenden Kriterien etc. die Methode `solve()` aufgerufen.

### 2.5.2 solve()

`solve()` führt einen Schritt aus und ruft sich rekursiv auf, um die nächsten Schritte bis zum Ende durchzuführen. Dafür wird als Argument den zu verwendenden Zustand übergeben. Nach einem weiteren Parameter für die benötigte Anzahl der Züge folgen drei Arrays, welche `PriorityDefinition`, `CycleSelection` und `StateSelection` enthalten.

Wenn der übergebene Zustand beim Aufruf bereits „fertig“ ist, also alle Pakete an ihrem Ziel sind, was die Variable `isFinished` des Zustandes angibt, wird, wenn weniger Schritte als die zuvor beste Lösung benötigt wurden, diese aktualisiert.

Ansonsten werden zunächst die zu verwendenden Kriterien und die Kantenpriorität-Definition ermittelt. Wird die dritte Strategie verwendet, was der boolesche Paramter `random` angibt, dann werden aus den drei Arrays zufällige Werte ausgewählt. Wird die erste Strategie verwendet, dann ist in jedem Array sowieso nur ein Wert enthalten. Bei der zweiten Strategie wird jede Kombination von Kantenpriorität-Definitionen und Kriterien zur Zyklenauswahl aus den Arrays verwendet, um fortzufahren.

Nachdem mit `createEdges()` die Kanten des aktuellen Zustandes erstellt wurden und `roundCoordinates()` auf ihn aufgerufen wurde, um die besten daraus entstehenden Zustände abzufragen, wird `solve()` auf den ersten der Zustände ausgeführt. Da `solve()` die Anzahl der Schritte, die noch zurückgegangen werden soll, zurückgibt, kann diese Zahl verwendet werden, um weitere Zustände zu verwenden, indem `solve()` auf sie aufgerufen wird, wodurch möglicherweise bessere Ergebnisse zustande kommen. Die dafür benötigten Einstellungen werden in der Klasse `Task` gespeichert.

# 3 Beispiele

In diesem Abschnitt werden verschiedene Beispiele gezeigt. In Abschnitt 3.1 wird ausführlich das Ergebnis von der gegebenen Paketverteilung für Amacity beschrieben. Die Beispiele werden in die drei Strategien aufgeteilt, um zu sehen, welche Strategien für welche Paketverteilungen am besten geeignet sind und welche am wenigsten Zeit benötigen. Um die Zeit messen und vergleichen zu können, wurden alle Beispiele auf demselben Computer ausgeführt.

Für die erste Strategie werden Tabellen verwendet, um die benötigte Schrittzahlen und Zeiten bei allen Kombinationen zu zeigen. In der linken Spalte steht, welche Definition der Kantenpriorität (Definition 1 ist „Gute Kanten“, Definition 2 ist „Alle Kanten“) und welches Kriterium zur Zustandsauswahl verwendet wird (Kriterium 1 ist „Erster Zustand“, Kriterium 2 ist „Kleinste Distanz“). In der oberen Zeile steht, welches Kriterium zur Zyklenauswahl verwendet wird (Kriterium 1 ist „Zyklenlänge“, Kriterium 2 ist „Summe“, Kriterium 3 ist „Durchschnitt“). Diese Werte innerhalb der Tabellen wurden aus den Programmausgaben ermittelt.

Alle vom Programm generierten Pläne sind im Ordner dieser Aufgabe in der Einsendung enthalten. Dabei wurden immer der erste generierte Plan mit der kleinsten Schrittanzahl für jede Strategie gespeichert. Im Dateinamen steht die Nummer des Beispiels, die Nummer der Strategie, und, wenn es in der Dokumentation mehrere Durchläufe für eine Strategie gibt, eine weitere Nummer zur Angabe des Prorgammdurchlaufs in der Reihenfolge, wie sie auch in der Dokumentation verwendet wird. Zusätzlich liegen die gesamten Programmausgaben von den Beispielen für Strategie 3 im Ordner, denn in der Dokumentation stehen dafür nur ausgewählte Werte bzw. Durchschnittswerte.

Der Plan für die gegebene Paketverteilung von Amacity mit 18 Schritten, und somit die dafür beste vom Programm gefundene Lösung, befindet sich in der Datei `droh-`

`nen1_plan_1_2.txt`.

## 3.0 Beispiel 0

Dies ist das $3 \cdot 3$-Beispiel, welches auch in Lösungsidee gezeigt wurde und in der Aufgabenstellung zu finden ist.

### 3.0.1 Strategie 1

| | **Zyklenlänge** | **Summe** | **Durchschnitt** |
|---|---|---|---|
| **Gute Kanten** | 4 | 4 | 5 |
| **Erster Zustand** | 0,9ms | 0,5ms | 0,7ms |
| **Gute Kanten** | 4 | 4 | 4 |
| **Kleinste Distanz** | 0,5ms | 0,4ms | 0,8ms |
| **Alle Kanten** | 4 | 4 | 5 |
| **Erster Zustand** | 0,6ms | 0,4ms | 1,5ms |
| **Alle Kanten** | 4 | 4 | 4 |
| **Kleinste Distanz** | 0,4ms | 0,4ms | 0,7ms |

Tabelle 1: Strategie 1 für Beispiel 0, ohne Schritte zurückzugehen

Nach knapp 1 Millisekunde wurde die erste Lösung gefunden, gleich mit der Mindestschrittanzahl von 4. Die meisten Kombinationen erreichen eine Schrittanzahl von 4, zwei davon stellen jedoch eine Ausnahme dar.

Wenn man die maximale Anzahl an verwendeten Zuständen pro Schritt auf 2 belässt, ist es egal, wie viele Schritte der Algorithmus zurück gehen darf. Die beiden Lösungen mit 5 Schritten bleiben erhalten. Erst wenn die maximale Anzahl an verwendeten Zuständen pro Schritt auf 3 gesetzt und der Algorithmus 2 Schritte zurückgehen darf, entstehen bei allen Kombinationen Lösungen mit 4 Schritten:

| | **Zyklenlänge** | **Summe** | **Durchschnitt** |
|---|---|---|---|
| **Gute Kanten** | 4 | 4 | 4 |
| **Erster Zustand** | 2,2ms | 0,5ms | 0,9ms |
| **Gute Kanten** | 4 | 4 | 4 |
| **Kleinste Distanz** | 0,6ms | 0,5ms | 0,7ms |
| **Alle Kanten** | 4 | 4 | 4 |
| **Erster Zustand** | 0,5ms | 0,5ms | 0,9ms |
| **Alle Kanten** | 4 | 4 | 4 |
| **Kleinste Distanz** | 0,5ms | 0,5ms | 0,4ms |

Tabelle 2: Strategie 1 für Beispiel 0, wobei maximal 2 Schritte zurück gegangen und maximal 3 Zustände pro Schritt verwendet werden dürfen

#### 3.0.2 Strategie 2

Bei der zweiten Strategie (ohne Schritte zurückzugehen) werden nur Pläne mit 4 Schritten generiert. Für die Zustandsauswahl durch Nehmen des ersten Zustands hat der Computer ca. 3,9 Millisekunden benötigt, beim zweiten Kriterium für die Auswahl des Zustandes ca. 2,5 Millisekunden. Die Ausgabedatei beinhaltet den gleichen Plan, wie der, der von Strategie 1 produziert wird.

#### 3.0.3 Strategie 3

Bei der Zufalls-Strategie (ohne Schritte zurückzugehen) benötigte 1 von 20 generierten Plänen 5 Schritte, die anderen 4. Die Durchschnittszeit, die pro Plan benötigt wurde, beträgt ca. 0,6 Millisekunden.

### 3.1 Beispiel 1

Dieses Beispiel verwendet die gegebene Paketverteilung für Amacity als Eingabe.

#### 3.1.1 Strategie 1

Lässt man das Programm laufen, ohne, dass der Algorithmus Schritte zurück gehen darf, dann beträgt die kleinste gefundene Schrittanzahl 19:

| | **Zyklenlänge** | **Summe** | **Durchschnitt** |
|---|---|---|---|
| **Gute Kanten** | 19 | 21 | 20 |
| **Erster Zustand** | 1s | 1s | 3s |
| **Gute Kanten** | 23 | 21 | 21 |
| **Kleinste Distanz** | 6s | 2s | 2s |
| **Alle Kanten** | 25 | 22 | 20 |
| **Erster Zustand** | 1s | 1s | 1s |
| **Alle Kanten** | 21 | 20 | 19 |
| **Kleinste Distanz** | 6s | 1s | 1s |

Tabelle 3: Strategie 1 für Beispiel 1, ohne Schritte zurückzugehen

Die Lösung mit den 19 Schritten wurde nach knapp 1 Sekunde vom Programm gefunden. Bei diesem Beispiel kann man schon deutlich sehen, dass in den meisten Fällen mehr Zeit benötigt wird, wenn der Zustand mit der kleinsten Maximal- bzw. Gesamtdistanz statt der erste gefundene Zustand verwendet wird.

Die Schrittanzahl kann noch auf 18 reduziert werden, wenn der Algorithmus 1 Schritt zurück gehen und maximal 2 Zustände pro Schritt verwenden darf:

| | **Zyklenlänge** | **Summe** | **Durchschnitt** |
|---|---|---|---|
| **Gute Kanten** | 18 | 21 | 19 |
| **Erster Zustand** | 1s | 2s | 3s |
| **Gute Kanten** | 22 | 21 | 21 |
| **Kleinste Distanz** | 23s | 27s | 3s |
| **Alle Kanten** | 21 | 22 | 18 |
| **Erster Zustand** | 4s | 1s | 1s |
| **Alle Kanten** | 20 | 20 | 19 |
| **Kleinste Distanz** | 275s | 1s | 2s |

Tabelle 4: Strategie 1 für Beispiel 1, wobei maximal 1 Schritt zurück gegangen und maximal 2 Zustände pro Schritt verwendet werden dürfen

Hierbei wurde die erste Lösung mit 19 Schritten nach knapp 0,7 Sekunden gefunden, die zweite Lösung mit 18 Schritte 0,02 Sekunden später. Hier wird noch deutlicher sichtbar, dass meistens mehr Zeit benötigt wird, wenn der Zustand mit der kleinsten Maximal- bzw. Gesamtdistanz statt der erste gefundene Zustand verwendet wird.

### 3.1.2 Strategie 2

Bei der zweiten Strategie (ohne Schritte zurückzugehen) wurde mit dem 1. Kriterium für die Zustandsauswahl ein Plan in 4 Sekunden gefunden, der 19 Schritte benötigt sowie mit dem 2. Kriterium ein Plan in 18 Sekunden, bei dem 23 Schritte benötigt werden.

### 3.1.3 Strategie 3

Es wurde 20 Mal Strategie 3 angewandt. Der Plan mit der kleinsten Schrittanzahl benötigte 18 Schritte, der mit der höchsten Schrittanzahl benötigte 22 Schritte. Das arithmetische Mittel der zehn Werte beträgt 20. Der Durchschnitt der benötigten Zeiten beträgt ca. 3 Sekunden.

## 3.2 Beispiel 2

Bei diesem $10 \cdot 10$-Beispiel ist die Paketverteilung „verkehrt herum sortiert“. Auf Abbildung 6 kann man sehen, was mit einer verkehrten Sortierung gemeint ist. Offensichtlich gibt es eine Lösung mit der Mindestschrittanzahl von 18. Dafür muss einfach jeder „Ring“ so lange gedreht werden, bis sich alle Pakete an ihren Zielen befinden. Es gibt insgesamt 5 solcher Ringe. Ein Ring besteht an jedem Rand aus allen sich dort befindlichen Paketen. Der äußerste Ring beinhaltet alle Pakete am Stadtrand, der zweite Ring dann alle Pakete am Rand, wenn alle Pakete des äußersten Ringes abgezogen wurden usw.

Für dieses Beispiel wird nur das erste Kriterium zur Zustandsauswahl verwendet, bei dem immer der erste Zustand genommen wird, da in diesem Fall immer, wenn ein Paket bearbeitet wurde, nur ein Zyklus verwendet werden muss, um mit diesem die nächsten Pakete zu bearbeiten. Bei der verkehrten Sortierung gibt es sehr viele Pakete, die in der Ausgangssituation gleich zwei beste Zyklen (nach Strategie 1 die längsten Zyklen) haben, dies sind zum Beispiel die vier Pakete in den Ecken und noch weitere. Da diese Zahl an Paketen sehr hoch ist und bei dem zweiten Kriterium zur Zustandsauswahl immer jeder Zyklus einmal verwendet werden muss, um das nächste Paket zu bearbeiten, würde diese Zustandsauswahl sehr viel Zeit benötigen, weshalb sie in diesem Beispiel nicht angewendet wird.

### 3.2.1 Strategie 1

| | **Zyklenlänge** | **Summe** | **Durchschnitt** |
|---|---|---|---|
| **Gute Kanten** | 34 | 37 | 35 |
| **Erster Zustand** | 4s | 7s | 5s |
| **Alle Kanten** | 34 | 28 | 27 |
| **Erster Zustand** | 4s | 2s | 4s |

Tabelle 5: Strategie 1 für Beispiel 2, ohne Schritte zurückzugehen

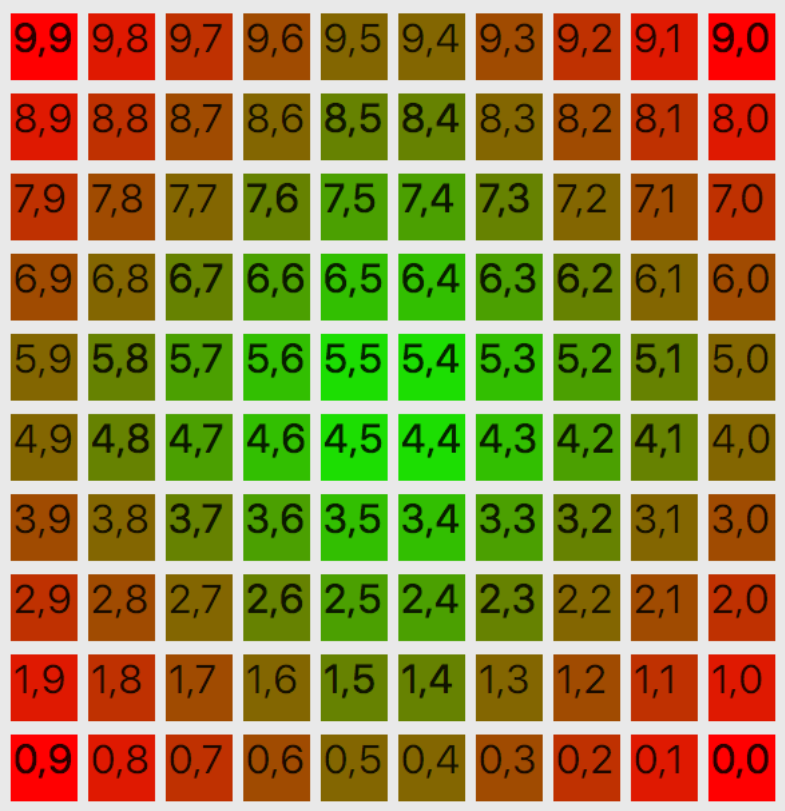


Abbildung 6: Die verkehrt herum sortierte Paketverteilung. Dies ist ein Screenshot aus dem Programm, welches die Schritte graphisch darstellt.

Der erste Plan mit 34 Schritten wurde nach 4 Sekunden gefunden. Es folgten bei späteren Kombinationen verbesserte Pläne mit 28 und 27 Schritten. Diese Zahlen sind trotzdem sehr weit von der Minimalschrittanzahl von 18 entfernt. Der Algorithmus ist für ein solches Beispiel also nicht sehr gut geeignet. Mit dem Zurückgehen von Schritten benötigte das Programm zu viel Zeit, weshalb die Ergebnisse davon hier nicht gezeigt werden. Der Grund dafür ist, dass Die Maximaldistanz aufgrund der hohen Schrittzahlen sehr oft nicht verkleinert werden.

### 3.2.2 Strategie 2

Die 2. Strategie wurde an dieser Stelle nicht angewendet, da das Suchen nach einer Lösung mit dem ersten Kriterium zur Zustandsauswahl identisch ist zum Suchen nach einer Lösung mit der ersten Strategie und der ersten Kombination von Kantenpriorität und Zyklenauswahl.

### 3.2.3 Strategie 3

Die dritte Strategie wurde 20 Mal angewendet. Der Plan mit der kleinsten Schrittanzahl benötigt 30 Schritte, der mit der höchsten Schrittanzahl benötigt 37 Schritte, durchschnittlich wurden Pläne mit 34 Schritten generiert. Die vom Programm benötigte Zeit betrug im Durchschnitt ca. 10 Sekunden, sie variierte jedoch zwischen 3 und 42 Sekunden.

## 3.3 Beispiel 3

Dieses $10 \cdot 10$-Beispiel wurde so konstruiert, dass es möglich ist, mit mehreren Zyklen alle Pakete in einem Schritt an ihr Ziel zu befördern. Es ist in Abbildung 7 mit der einschrittigen Lösung zu sehen.

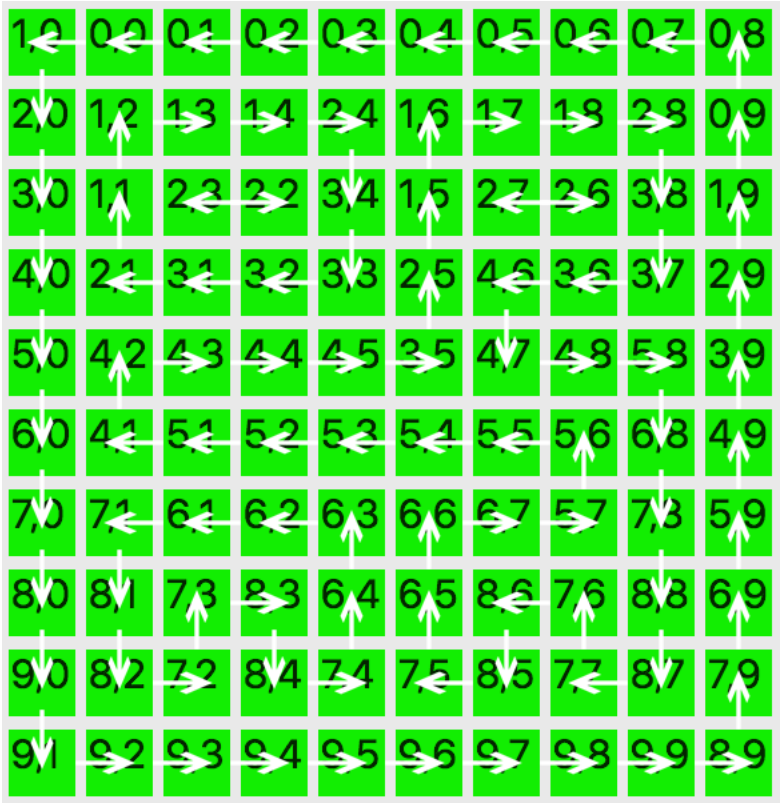

Abbildung 7: Eine Paketverteilung, die in einem Schritt gelöst werden kann. Die Lösung wird durch die Pfeile dargestellt. Dies ist ein Screenshot aus dem Programm, welches die Schritte graphisch darstellt.

Hier ist es nicht nötig, in die drei Strategien zu unterteilen. Das Programm findet immer die Lösung mit einem Schritt. Der Computer benötigte durchschnittlich für jede Lösung ca. 2 Millisekunden. Es werden immer dieselben Zyklen gefunden, da zunächst bei jedem Zyklus versucht wird, keine schlechte Kante zu nehmen. Dies ist nur möglich, indem die in Abbildung 7 gezeigten Zyklen genommen werden.

### 3.4 Beispiel 4

Für dieses Beispiel wurden die Pakete einfach zufällig in einer $10 \cdot 10$-Stadt verteilt. Die Paketverteilung ist in Abbildung 8 zu sehen.

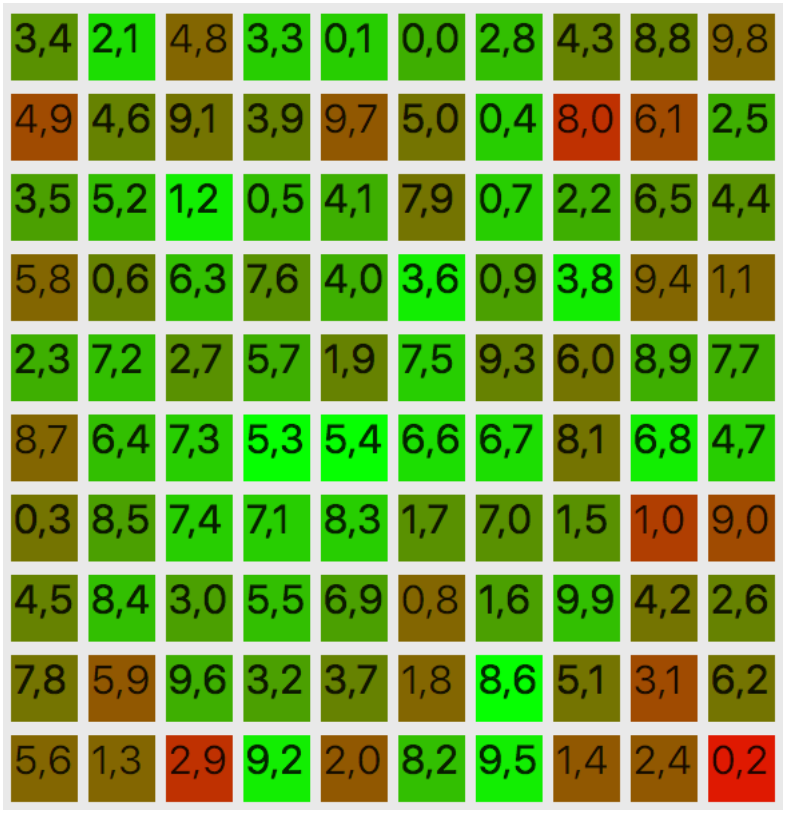


Abbildung 8: Eine zufällige Paketverteilung. Dies ist ein Screenshot aus dem Programm, welches die Schritte graphisch darstellt.

### 3.4.1 Strategie 1

| | Zyklenlänge | Summe | Durchschnitt |
|---|---|---|---|
| **Gute Kanten** | 21 | 23 | 19 |
| **Erster Zustand** | 0,3s | 0,6s | 0,3s |
| **Gute Kanten** | 22 | 21 | 22 |
| **Kleinste Distanz** | 29s | 0,7s | 1,3s |
| **Alle Kanten** | 23 | 22 | 21 |
| **Erster Zustand** | 0,2s | 0,6s | 0,4s |
| **Alle Kanten** | 21 | 19 | 20 |
| **Kleinste Distanz** | 40s | 1,0s | 1,5s |

Tabelle 6: Strategie 1 für Beispiel 4, ohne Schritte zurück gehen

Der erste Plan wurde nach 0,3 Sekunden gefunden, er benötigt 21 Schritte. Erst bei einer späteren Kombination, welche auch 0,3 Sekunden zur Berechnung benötigte, wird ein Plan mit 19 Schritten gefunden.

### 3.4.2 Strategie 2

Die zweite Strategie generiert Pläne mit 21 und 22 Schritten für das erste und das zweite Kriterium zur Zustandsauswahl, wofür das Programm 2 und 67 Sekunden Zeit benötigte. Es wurden alle Kombinationen der Definition der Kantenpriorität und der Zyklenauswahl verwendet.

### 3.4.3 Strategie 3

Die dritte Strategie generierte in 20 Durchläufen Pläne mit einem Durchschnitt der Schrittanzahl von $20,6$, wobei das Minimum bei 19 und das Maximum bei 24 lag. Die durchschnittliche vom Programm benötigte Zeit betrug ca. 6 Sekunden.

## 4 Quelltext

Listing 1: main.swift

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

// ... Asynchron Befehle abfragen ...

task.run()

// Endlosschleife, damit weiterhin asynchron Befehle abgefragt werden können:
while true {}
```

Listing 2: Task.swift

```swift
enum MoveDirection {
    case North, East, South, West
}

struct Coordinates: Equatable {

    let row: Int
    let column: Int

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

    // Diese vier Methoden geben die Koordinaten ein Haus weiter
    // in der entsprechenden Himmelsrichtung zurück:
    func east() -> Coordinates {
        return Coordinates(row: row, column: column + 1)
    }
    func west() -> Coordinates {
        return Coordinates(row: row, column: column - 1)
    }
    func north() -> Coordinates {
        return Coordinates(row: row - 1, column: column)
    }
    func south() -> Coordinates {
        return Coordinates(row: row + 1, column: column)
    }

    // Gibt die Koordinaten in der angegebenen Himmelsrichtung zurück
    func inDirection(direction: MoveDirection) -> Coordinates {
        switch direction {
        case .North: return north()
        case .East: return east()
        case .South: return south()
        case .West: return west()
        }
    }

}

func ==(left: Coordinates, right: Coordinates) -> Bool {
    return left.row == right.row && left.column == right.column
}

// Mögliche Definitionen der Kantenpriorität
enum PriorityDefinition {
    case DistanceGoodEdges
```

```
    case DistanceAllEdges
}

// Mögliche Kriterien zur Auswahl von Zyklen
enum CycleSelection {
    case CycleLength
    case PrioritySum
    case PriorityAverage
}

// Mögliche Kriterien zur Auswahl von Zuständen
enum StateSelection {
    case FirstState
    case SmallestMaximumTotal
}

// Repräsentiert ein Paket
struct Package {

    let destination: Coordinates // Paketziel
    var distance = 0 // Distanz zu seinem Ziel
    var moveDirection: MoveDirection? = nil // Wurfrichtung
    var possibleMoves = [(direction: MoveDirection, priority: Int)]() // mögliche
        Wurfrichtungen

    init(destination: Coordinates) {
        self.destination = destination
    }

}

// Repräsentiert einen Weg im Graphen, wobei jeder enthaltene Knoten eine ausgehende Kante
// hat, damit später ein Zyklus daraus wird.
struct Path {

    // Verwendete Knoten und Kanten
    var vertices = [(vertex: Coordinates, direction: MoveDirection)]()
    var prioritySum = 0

    var length: Int {
        return vertices.count
    }
    var start: Coordinates {
        return vertices[0].0
    }
    var priorityAverage: Double {
        return Double(prioritySum) / Double(length)
    }
```

```
    // Fügt den Knoten mit seiner Kante zum Weg hinzu
    mutating func append(coordinates: Coordinates, moveDirection: MoveDirection, priority:
        Int) {
        vertices.append((coordinates, moveDirection))
        prioritySum += priority
    }

    // Entfernt den letzten Knoten und die Kantenpriorität aus dem Weg
    mutating func removeLast(priority priority: Int) {
        vertices.removeLast()
        prioritySum -= priority
    }

}

// Stellt einen Zustand dar.
struct State {

    var packages: [[Package]] // Pakete auf den Häusern
    var moves: [[String]] // Wurfrichtungen der Häuser

    var maxDist = 0 // Maximaldistanz
    var totalDist = 0 // Gesamtdistanz

    // Ermöglicht den Zugriff auf das Paket eines Hauses durch Koordinaten
    subscript(coordinates: Coordinates) -> Package {
        get {
            return packages[coordinates.row][coordinates.column]
        }
        set {
            packages[coordinates.row][coordinates.column] = newValue
        }
    }

    // true, wenn sich alle Pakete an ihren Zielen befinden, ansonsten false
    var isFinished: Bool {
        for row in 0..<packages.count {
            for column in 0..<packages.count {
                if packages[row][column].destination != Coordinates(row: row, column:
                    column) {
                    return false
                }
            }
        }
        return true
    }

    // Die Lösung im Format der Ausgabedatei
    var solutionString: String {
```

```
    // Lösung als String generieren ...
}

init(width: Int) {
    // Arrays zunächst mit Standardwerten initialisieren
    packages = [[Package]](count: width, repeatedValue: [Package](count: width,
        repeatedValue: Package(destination: Coordinates(row: 0, column: 0))))
    moves = [[String]](count: width, repeatedValue: [String](count: width,
        repeatedValue: ""))
}

// Berechnet die Maximaldistanz, die Gesamtdistanz und die Distanzen von allen Paketen
    zu ihren Zielen
mutating func calculateDistances() {
    maxDist = 0
    totalDist = 0
    for row in 0..<packages.count {
        for column in 0..<packages.count {
            packages[row][column].distance = abs(packages[row][column].destination.row
                - row) + abs(packages[row][column].destination.column - column)
            maxDist = max(maxDist, packages[row][column].distance)
            totalDist += packages[row][column].distance
        }
    }
}

// Fügt alle Kanten in die entsprechenden Arrays der Pakete ein und sortiert sie
mutating func createEdges(priorityDefinition: PriorityDefinition) {
    for row in 0..<packages.count {
        for column in 0..<packages.count {

            // Array zunächst von vorherigen Werten leeren
            packages[row][column].possibleMoves = [(direction: MoveDirection, priority:
                Int)]()

            switch priorityDefinition {
            case .DistanceGoodEdges:
                // Wenn das Paket in die richtige Richtung geworfen wird, den Abstand
                // zum Ziel in dieser Richtung verwenden, ansonsten 0
                if row != 0 {
                    packages[row][column].possibleMoves.append((direction: .North,
                        priority: packages[row][column].destination.row < row ? (row -
                        packages[row][column].destination.row) : 0))
                }
                if column != packages.count - 1 {
                    packages[row][column].possibleMoves.append((direction: .East,
                        priority: packages[row][column].destination.column > column ?
                        (packages[row][column].destination.column - column) : 0))
                }
```

```
                if row != packages.count - 1 {
                    packages[row][column].possibleMoves.append((direction: .South,
                        priority: packages[row][column].destination.row > row ?
                        (packages[row][column].destination.row - row) : 0))
                }
                if column != 0 {
                    packages[row][column].possibleMoves.append((direction: .West,
                        priority: packages[row][column].destination.column < column ?
                        (column - packages[row][column].destination.column) : 0))
                }
            case .DistanceAllEdges:
                // Den Abstand zum Ziel in die Wurfrichtung verwenden
                if row != 0 {
                    packages[row][column].possibleMoves.append((direction: .North,
                        priority: row - packages[row][column].destination.row))
                }
                if column != packages.count - 1 {
                    packages[row][column].possibleMoves.append((direction: .East,
                        priority: packages[row][column].destination.column - column))
                }
                if row != packages.count - 1 {
                    packages[row][column].possibleMoves.append((direction: .South,
                        priority: packages[row][column].destination.row - row))
                }
                if column != 0 {
                    packages[row][column].possibleMoves.append((direction: .West,
                        priority: column - packages[row][column].destination.column))
                }
            }

            packages[row][column].possibleMoves.sortInPlace{ $0.1 > $1.1 }
        }
    }
}

// Fügt die ausgewählten Kanten zu moves hinzu, was den Plan um einen Schritt erweitert
mutating func addCurrentMovesToSolution() {
    // N, O, S, W oder _ wird aus moveDirection ermittelt und zu moves hinzugefügt ...
}

// Bewegt alle Pakete entlang ihrer Wurfrichtungen
mutating func move() {
    // Alten Zustand kopieren:
    let old = self

    for row in 0..<packages.count {
        for column in 0..<packages.count {
            // Wurfrichtungen aus dem alten Zustand verwenden:
            if old.packages[row][column].moveDirection != nil {
```

```
                let newPosition = old.coordinatesInMoveDirection(Coordinates(row: row,
                    column: column))
                self[newPosition] = old.packages[row][column]
                self[newPosition].moveDirection = nil
            }
        }
    }
}

// Gibt die Koordinaten in der Wurfrichtung des Paketes an den gegebenen Koordinaten
    zurück
func coordinatesInMoveDirection(coordinates: Coordinates) -> Coordinates {
    switch self[coordinates].moveDirection! {
    case .North: return coordinates.north()
    case .East: return coordinates.east()
    case .South: return coordinates.south()
    case .West: return coordinates.west()
    }
}

// Erstellt ein Array mit allen Koordinaten und gibt es sortiert zurück
func sortedCoordinatesArray() -> [Coordinates] {
    // Array erstellen:
    var array = [Coordinates]()
    for row in 0..<packages.count {
        for column in 0..<packages.count {
            array.append(Coordinates(row: row, column: column))
        }
    }

    // Array sortieren:
    return array.sort { (a, b) -> Bool in
        if self[a].distance == self[b].distance {
            // Bei gleicher Distanz nach Zeile/Spalte sortieren
            if a.row == b.row { return a.column < b.column }
            else { return a.row < b.row }
        } else {
            return self[a].distance > self[b].distance
        }
    }
}

// Prüft die Erreichbarkeit von to von coordinates aus. Alle Knoten mit true in visited
// werden nicht erneut besucht.
func floodFillReachability(coordinates: Coordinates, to: Coordinates, inout visited:
    [[Bool]]) -> Bool {
    if coordinates == to {
        // Ziel erreicht
        return true
```

```
        }
        if visited[coordinates.row][coordinates.column] || self[coordinates].moveDirection
            != nil {
            // Wurde schon besucht oder kann keine weitere Kante auswählen
            return false
        }

        // Knoten wird jetzt besucht:
        visited[coordinates.row][coordinates.column] = true

        // Alle erlaubten Kanten durchgehen und Flood Fill auf deren Ziele ausführen:
        for (direction, priority) in self[coordinates].possibleMoves {
            guard priority > 0 || self[coordinates].distance + 1 < maxDist else { continue }
            if floodFillReachability(coordinates.inDirection(direction), to: to, visited:
                &visited) {
                // Flood Fill hat Zielknoten erreicht
                return true
            }
        }

        // Von hier aus nicht gefunden:
        return false
    }

    // Prüft die Erreichbarkeit von coordinates von from aus.
    func isReachable(coordinates: Coordinates, from: Coordinates) -> Bool {
        // Alle Werte zunächst mit false initialisieren:
        var reachable = [[Bool]](count: packages.count, repeatedValue: [Bool](count:
            packages.count, repeatedValue: false))
        return floodFillReachability(from, to: coordinates, visited: &reachable)
    }

    // Sucht einen Zyklus von coordinates aus, wobei maximal remainingWrong weitere
    // schlechte Kanten benutzt werden. Alle durch die Rekursion aufgerufenen Knoten und
    // Kanten werden in currentPath gespeichert, bestCycles beinhaltet die besten bisher
    // gefundenen Zyklen. cycleSelection ist das Kriterium zur Zyklenauswahl.
    mutating func searchCycle(coordinates coordinates: Coordinates, remainingWrong: Int,
        inout currentPath: Path, inout bestCycles: [Path], cycleSelection: CycleSelection) {

        if currentPath.length > 0 && currentPath.start == coordinates {
            // Zyklus gefunden

            if bestCycles.isEmpty {
                // Noch kein Zyklus in bestCycles, also einfach hinzufügen
                bestCycles.append(currentPath)
                return
            }

            switch cycleSelection {
```

```
            case .CycleLength:
                // Die längsten Zyklen werden verwendet
                if currentPath.length > bestCycles[0].length {
                    bestCycles.removeAll()
                    bestCycles.append(currentPath)
                } else if currentPath.length == bestCycles[0].length {
                    bestCycles.append(currentPath)
                }
            case .PrioritySum:
                // Die Zyklen mit der höchsten Summe aller Prioritäten werden verwendet
                if currentPath.prioritySum > bestCycles[0].prioritySum {
                    bestCycles.removeAll()
                    bestCycles.append(currentPath)
                } else if currentPath.prioritySum == bestCycles[0].prioritySum {
                    bestCycles.append(currentPath)
                }
            case .PriorityAverage:
                // Die Zyklen mit dem höchsten Durchschnitt aller Prioritätet werden
                    verwendet
                if currentPath.priorityAverage > bestCycles[0].priorityAverage {
                    bestCycles.removeAll()
                    bestCycles.append(currentPath)
                } else if currentPath.priorityAverage == bestCycles[0].priorityAverage {
                    bestCycles.append(currentPath)
                }
            }
            return
        }

        // Hat bereits eine ausgehende Kante:
        guard self[coordinates].moveDirection == nil else { return }
        // Startknoten ist nicht erreichbar:
        guard currentPath.length == 0 || isReachable(currentPath.start, from: coordinates)
            else { return }

        for (direction, priority) in self[coordinates].possibleMoves {
            // Überprüfen, ob Kante gewählt werden darf:
            guard priority > 0 || (self[coordinates].distance + 1 < maxDist &&
                remainingWrong > 0) else { continue }

            self[coordinates].moveDirection = direction

            // Knoten und Kante zum Weg hinzufügen und rekursiv den nächsten Knoten aufrufen
            currentPath.append(coordinates, moveDirection: direction, priority: priority)
            searchCycle(coordinates: coordinatesInMoveDirection(coordinates),
                remainingWrong: priority <= 0 ? remainingWrong - 1 : remainingWrong,
                currentPath: &currentPath, bestCycles: &bestCycles, cycleSelection:
                cycleSelection)
            currentPath.removeLast(priority: priority)
```

```
        }

        // Ausgewählte Kante des Knotens wieder entfernen
        self[coordinates].moveDirection = nil
    }

    // Sucht die besten Zyklen für den Knoten mit dem Index coordinatesIndex im Array
    // sortedCoordinates. Für jeden dieser Zyklen wird die Methoden rekursiv aufgerufen,
    // wobei der nächste Knoten in sortedCoordinates bearbeitet wird.
    mutating func roundCoordinates(inout bestStates bestStates: [State], inout
        sortedCoordinates: [Coordinates], coordinatesIndex: Int, cycleSelection:
        CycleSelection, stateSelection: StateSelection, maxStates: Int?) {

        if coordinatesIndex == sortedCoordinates.count {
            // Alle Pakete fertig bearbeitet

            // Zustand kopieren und Schritte durchführen
            var resultState = self
            resultState.addCurrentMovesToSolution()
            resultState.move()
            resultState.calculateDistances()

            if bestStates.isEmpty {
                // Vorher noch keinen Zustand gefunden, daher hinzufügen:
                bestStates.append(resultState)
                return
            }

            switch stateSelection {
            case .FirstState:
                // Einfach den ersten Zustand (und evtl. die dahinter) nehmen, daher
                    einfach hinzufügen:
                bestStates.append(resultState)
            case .SmallestMaximumTotal:
                // Den Zustand mit der kleinsten Maximal- und Gesamtdistanz als besten
                    Zustand werten:
                if resultState.maxDist < bestStates[0].maxDist || (resultState.maxDist ==
                    bestStates[0].maxDist && resultState.totalDist <
                    bestStates[0].totalDist) {
                    bestStates.removeAll()
                    bestStates.append(resultState)
                } else if resultState.maxDist == bestStates[0].maxDist &&
                    resultState.totalDist == bestStates[0].totalDist {
                    bestStates.append(resultState)
                }
            }

            return
        }
```

```
        let coordinates = sortedCoordinates[coordinatesIndex]

        // Als erstes keine schlechte Kante verwenden, dann eine, zwei usw.
        for (var remainingWrong = 0; remainingWrong <= packages.count * packages.count;
            remainingWrong++) {

            // Zyklen mit remainingWrong schlechten Kanten suchen:
            var currentPath = Path()
            var bestCycles = [Path]()
            searchCycle(coordinates: coordinates, remainingWrong: remainingWrong,
                currentPath: &currentPath, bestCycles: &bestCycles, cycleSelection:
                cycleSelection)

            if !bestCycles.isEmpty {
                // Es wurden Zyklen gefunden, alle durchlaufen:
                for cycle in bestCycles {
                    // Alle Kanten im Zyklus auswählen:
                    for (cycleCoords, direction) in cycle.vertices {
                        self[cycleCoords].moveDirection = direction
                    }

                    // Zyklen für das nächste Paket finden:
                    roundCoordinates(bestStates: &bestStates, sortedCoordinates:
                        &sortedCoordinates, coordinatesIndex: coordinatesIndex + 1,
                        cycleSelection: cycleSelection, stateSelection: stateSelection,
                        maxStates: maxStates)

                    // Die Auswahl der Kanten im Zyklus wieder rückgängig machen:
                    for (cycleCoords, _) in cycle.vertices {
                        self[cycleCoords].moveDirection = nil
                    }

                    if let maxStates = maxStates where bestStates.count >= maxStates {
                        // Es wurden genügend Zustände gefunden, daher kann abgebrochen
                            werden
                        return
                    }
                }
                return
            }
        }

        // Es war nicht möglich, einen Zyklus zu finden, daher ohne fortfahren:
        roundCoordinates(bestStates: &bestStates, sortedCoordinates: &sortedCoordinates,
            coordinatesIndex: coordinatesIndex + 1, cycleSelection: cycleSelection,
            stateSelection: stateSelection, maxStates: maxStates)
    }
```

```
}

// Task findet Lösungen zu einer gegebenen Paketverteilung:
class Task {

    // Startstatus und Einstellungen:
    var startState: State
    let priorityDefinitions: [PriorityDefinition]
    let cycleSelections: [CycleSelection]
    let stateSelections: [StateSelection]
    let maxChoosenStates: Int
    let goBackOnNotDecreasingMaxDistance: Int
    let tryCombinations: Bool
    let tryCombinationsForEachStateSelection: Bool
    let tryRandom: Bool

    // Beste gefundene Lösung:
    var bestSolution: State? = nil

    // Kleinste Schrittanzahl für die aktuelle Strategie und die aktuelle Kombination
    // von Auswahlkriterien etc.:
    var currentBest = 0

    // Zeitpunkt, an dem angefangen wurde, mit der aktuellen Strategie und der
    // aktuellen Kombination von Auswahlkriterien etc. zu suchen:
    var startDate: NSDate?

    // Eine Aufgabe mit dem Startstatus und den Einstellungen initialisieren:
    init(startState: State, priorityDefinitions: [PriorityDefinition], cycleSelections:
        [CycleSelection], stateSelections: [StateSelection], maxChoosenStates: Int,
        goBackOnNotDecreasingMaxDistance: Int, tryCombinations: Bool,
        tryCombinationsForEachStateSelection: Bool, tryRandom: Bool) {
        self.startState = startState
        self.priorityDefinitions = priorityDefinitions
        self.cycleSelections = cycleSelections
        self.stateSelections = stateSelections
        self.maxChoosenStates = maxChoosenStates
        self.goBackOnNotDecreasingMaxDistance = goBackOnNotDecreasingMaxDistance
        self.tryCombinations = tryCombinations
        self.tryCombinationsForEachStateSelection = tryCombinationsForEachStateSelection
        self.tryRandom = tryRandom
    }

    // Führt einen Schritt für state aus. Die Methode wird rekursiv aufgerufen, sodass auch
        alle
    // weiteren Schritte ausgeführt werden.
    func solve(var state: State, moves: Int, random: Bool, priorityDefinitions:
        [PriorityDefinition], cycleSelections: [CycleSelection], stateSelections:
        [StateSelection]) -> Int {
```

```
        if state.isFinished {
            // Es wurde eine Lösung gefunden. Wenn sie besser oder die erste ist, dann muss
            // das bekanntgegeben werden.
            if let bestSolution = bestSolution {
                if moves < bestSolution.moves[0][0].characters.count {
                    self.bestSolution = state
                    print("Es wurde eine bessere Lösung mit \(moves) Schritten nach
                        \(NSDate().timeIntervalSinceDate(startDate!)) Sekunden gefunden.")
                }
            } else {
                bestSolution = state
                print("Es wurde eine Lösung mit \(moves) Schritten nach
                    \(NSDate().timeIntervalSinceDate(startDate!)) Sekunden gefunden.")
            }

            // Außerdem currentBest aktualisieren:
            currentBest = min(currentBest, moves)

            return 0
        }

        // Alle für diesen Schritt verwendeten Definitionen und Auswahlkriterien:
        let usedPriorityDefinitions: [PriorityDefinition]
        let usedCycleSelections: [CycleSelection]
        let usedStateSelection: StateSelection

        if random {
            // Strategie 3, also eine zufällige Kombination auswählen:
            usedPriorityDefinitions =
                [priorityDefinitions[Int(arc4random_uniform(UInt32(priorityDefinitions.count)))]]
            usedCycleSelections =
                [cycleSelections[Int(arc4random_uniform(UInt32(cycleSelections.count)))]]
            usedStateSelection =
                stateSelections[Int(arc4random_uniform(UInt32(stateSelections.count)))]
        } else {
            // Strategie 1 oder 2, daher die gegebene(n) Kombination(en) verwenden.
            // stateSelections kann in diesem Fall nur einen Wert besitzen.
            usedPriorityDefinitions = priorityDefinitions
            usedCycleSelections = cycleSelections
            usedStateSelection = stateSelections[0]
        }

        var bestStates = [State]()
        var sortedCoordinates = state.sortedCoordinatesArray()

        // Alle Kombinationen durchlaufen (nur bei Strategie 2 gibt es mehr als eine
            Kombination):
        for priorityDefinition in usedPriorityDefinitions {
```

```
        for cycleSelection in usedCycleSelections {
            // Kanten mit der gewählten Definition erstellen:
            state.createEdges(priorityDefinition)

            // Die besten Zustände generieren:
            state.roundCoordinates(bestStates: &bestStates, sortedCoordinates:
                &sortedCoordinates, coordinatesIndex: 0, cycleSelection:
                cycleSelection, stateSelection: usedStateSelection, maxStates:
                usedStateSelection == .FirstState ? maxChoosenStates : nil)
        }
    }

    var i = 0 // Index des nächsten Zustandes in bestStates
    var remainingUp: Int // Anzahl der zurückzugehenden Schritte

    if bestStates[i].maxDist == state.maxDist {
        // Maximaldistanz wurde nicht verkleinert
        remainingUp = goBackOnNotDecreasingMaxDistance
        solve(bestStates[i++], moves: moves + 1, random: random, priorityDefinitions:
            priorityDefinitions, cycleSelections: cycleSelections, stateSelections:
            stateSelections)
    } else {
        remainingUp = solve(bestStates[i++], moves: moves + 1, random: random,
            priorityDefinitions: priorityDefinitions, cycleSelections: cycleSelections,
            stateSelections: stateSelections) - 1
    }

    // Wenn notwendig und so eingestellt noch weitere Zustände ausprobieren, bis
    // einer gefunden wird, wo keine Schritte mehr zurückgegangen werden müssen
    // oder die Maximalanzahl an verwendeten Zuständen erreicht ist
    var done = 1
    while done < maxChoosenStates && i < bestStates.count && remainingUp > 0 {
        if bestStates[i].maxDist == state.maxDist {
            // Maximaldistanz nicht verkleinert, daher
            // kann remainingUp sich auch nicht ändern
            solve(bestStates[i++], moves: moves + 1, random: random,
                priorityDefinitions: priorityDefinitions, cycleSelections:
                cycleSelections, stateSelections: stateSelections)
        } else {
            // remainingUp ist das Minimum von seinem Wert davor
            // und dem jetzt berechneten
            remainingUp = min(remainingUp, solve(bestStates[i++], moves: moves + 1,
                random: random, priorityDefinitions: priorityDefinitions,
                cycleSelections: cycleSelections, stateSelections: stateSelections) - 1)
        }
        done++
    }

    return remainingUp
```

```
    }

    func run() {
        // Zeitmessung und Ausgaben werden weggelassen...
        startState.calculateDistances()
        if tryCombinations {
            for priorityDefinition in priorityDefinitions {
                for cycleSelection in cycleSelections {
                    for stateSelection in stateSelections {
                        currentBest = Int.max
                        solve(startState, moves: 0, random: false, priorityDefinitions:
                            [priorityDefinition], cycleSelections: [cycleSelection],
                            stateSelections: [stateSelection])
                    }
                }
            }
        }
        if tryCombinationsForEachStateSelection {
            for stateSelection in stateSelections {
                currentBest = Int.max
                solve(startState, moves: 0, random: false, priorityDefinitions:
                    priorityDefinitions, cycleSelections: cycleSelections, stateSelections:
                    [stateSelection])
            }
        }
        if tryRandom {
            while true {
                currentBest = Int.max
                solve(startState, moves: 0, random: true, priorityDefinitions:
                    priorityDefinitions, cycleSelections: cycleSelections, stateSelections:
                    stateSelections)
            }
        }
    }

}
```