# Aufgabe 2
# „Wehret den Wildschweinen!“-
# Dokumentation

### 36. Bundeswettbewerb Informatik 2017/18 - 2. Runde

Lukas Rost

Teilnahme-ID: 44137

9. April 2018

## Inhaltsverzeichnis

# 1 Lösungsidee

## 1.1 Umformung des Problems in ein Min-Cut-Problem

Die Matrix der Planquadrate lässt sich als gerichteter und kantengewichteter Graph $G = \{V, E\}$ darstellen, für den gilt:

- Die Menge der Knoten $V$ entspricht der Menge aller Planquadrate in der Matrix.
- Die Menge der Kanten $E$ enthält für jeden Knoten eine gerichtete Kante zu seinen oben, unten, links und rechts benachbarten Knoten, sofern diese existieren. Das Gewicht einer solchen Kante wird definiert als $1 - \Delta h$, [1] wobei $\Delta h$ den Betrag des Höhenunterschieds zwischen den verbundenen Knoten darstellt. Somit entspricht das Gewicht der Kante dem Höhenunterschied, der zwischen den beiden Knoten noch hinzugewonnen werden müsste, um den Übergang zwischen ihnen unpassierbar zu machen. Hätte eine Kante ein Gewicht $\leq 0$, so wird diese nicht zum Graphen hinzugefügt, da der Übergang zwischen diesen Knoten schon im Ursprungszustand unpassierbar ist.
- Es werden zwei zusätzliche Knoten $s$ (Quelle) und $t$ (Senke) definiert. Die Quelle stellt dabei bildlich gesprochen den Wald dar, aus dem die Wildschweine kommen und die Senke repräsentiert den Acker, den die Wildschweine erreichen wollen.
- Es werden gerichtete Kanten zum Graphen hinzugefügt, die von $s$ zu jedem Knoten in der ersten Reihe[2] der Matrix verlaufen, da die Wildschweine ja immer über diese Reihe das Brachland betreten. Weiterhin werden ebensolche Kanten von jedem Knoten in der letzten Reihe zu $t$ hinzugefügt, da die Wildschweine über diese Reihe das Brachland verlassen. Allen diesen Kanten wird als Gewicht $\infty$ zugewiesen.[3]

Nun stellt der Graph ein Flussnetzwerk dar, welches im nächsten Abschnitt genauer erklärt wird. In diesem Netzwerk entspricht das Problem der kostengünstigsten Erdarbeiten dem Problem des minimalen s-t-Schnitts. Ein s-t-Schnitt ist dabei definiert als eine Aufteilung der Knoten senkrecht zum Netzwerkfluss in zwei disjunkte Teilmengen $S$ und $T$, für die gilt $s \in S$ und $t \in T$. Ein minimaler Schnitt ist ein solcher Schnitt, sodass die Summe der Kantenkapazitäten bzw. Kantengewichte von $S$ nach $T$ minimal wird. Es gilt also, dass

$$c(S,T) = \sum_{u \in S, v \in T \wedge (u,v) \in E} c(u,v) \tag{1}$$

für keinen anderen Schnitt kleiner ist. [8] Die Kanten dieses Schnitts geben dann an, zwischen welchen Feldern Erde verschoben werden muss, um so kostengünstig wie möglich das Brachland unpassierbar zu machen.

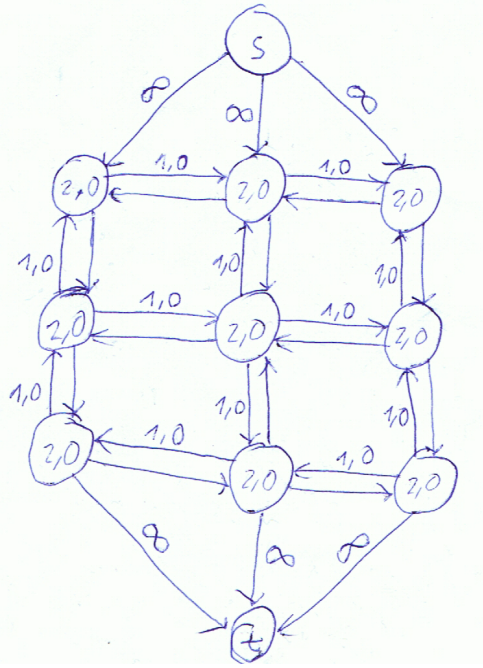
Dadurch, dass die Kanten zu s und zu t ein Gewicht von $\infty$ haben, werden diese keinesfalls Teil des minimalen Schnitts sein. Ebensowenig werden Kanten, die innerhalb der ersten oder letzten Reihe verlaufen, dem minimalen Schnitt angehören, da dann auch

[1] Hier ließe sich anstatt 1 Meter auch jeder beliebige andere Höhenunterschied einsetzen, der dann von den Wildschweinen nicht überwindbar ist.

[2] Reihen verlaufen von links/Westen nach rechts/Osten, Spalten von oben/Norden nach unten/Süden.

[3] Diese Vorgehensweise entspricht derjenigen für das Max-Flow- bzw. Min-Cut-Problem mit mehreren Quellen und Senken, da eigentlich die gesamte erste Reihe die Quellen und die gesamte letzte Reihe die Senken darstellt.[9]

Kanten zu s bzw. t Teil dieses Schnitts wären, was (s.o.) ausgeschlossen ist. Somit gibt der Schnitt in jedem Fall Erdarbeiten an, die, egal von welchem Planquadrat der ersten bzw. letzten Reihe das Brachland betreten bzw. verlassen wird, das Brachland unpassierbar machen. Ein Beispiel eines so gebildeten Netzwerkes mit $n = 3$ ist in der folgenden Abbildung zu sehen.




## 1.2 Grundlegende Informationen zu Flussnetzwerken

Ein Flussnetzwerk ist ein gerichteter Graph $G = \{V, E\}$ mit einer charakteristischen Quelle $s$ und einer Senke $t$ sowie einer Kapazitätsfunktion $e \to c(e)$, die jeder Kante eine nichtnegative reelle Zahl zuweist. In unserem Fall entspricht die Kapazität einer Kante dem Gewicht dieser. Über dieses Netzwerk kann ein Fluss gelegt werden. Dieser muss folgende Bedingungen erfüllen:

- Ein Fluss hat eine Flusswertfunktion $e \to f(e)$, die jeder Kante einen nicht negativen Wert $f(e)$ zuweist. Der Flusswert $f(e)$ ist maximal so groß wie seine Kapazität, d. h. für für jede Kante $e \in V$ gilt $f(e) \leq c(e)$.

- Für jeden Knoten (außer der Quelle $s$ und der Senke $t$) gilt der Flusserhalt: Für alle Kanten, die in einen Knoten gehen, ist die Summe der Werte f(e) genauso groß wie bei den Kanten, die aus einem Knoten gehen. Mathematisch lässt sich dies wie folgt formulieren:

$$\forall v, e \in G \backslash \{s, t\} : \sum_{\text{e führt nach v}} f(e) = \sum_{\text{e führt weg von v}} f(e) \tag{2}$$

- Der Abfluss von der Quelle S ist gleich dem Zufluss zu der Senke t. Es gilt somit ($e \in V$):

$$\forall e \in G : \sum_{\text{e führt weg von s}} f(e) = \sum_{\text{e führt nach t}} f(e) \tag{3}$$

  Der Abfluss von $s$ bzw. der Zufluss zu $t$ wird als Wert des Flusses $|f|$ bezeichnet.

- Ein Fluss ist genau dann maximal, wenn bei keinem anderen Fluss in dem Graphen $|f|$ größer ist.

Mithilfe des maximalen Flusses lässt sich nun ein minimaler s-t-Schnitt in diesem Graphen bestimmen. Nach dem Max-Flow-Min-Cut-Theorem gilt zwischen diesen der Zusammenhang, dass ein maximaler Fluss genau den Wert eines minimalen Schnitts hat. Dies lässt sich folgendermaßen beweisen:

*Beweisskizze.* Der Fluss, der über einen Schnitt fließt, hat den Wert des gesamten Flusses, da ein Schnitt den Graphen in zwei Teilmengen teilt und somit der gesamte Fluss zwischen $s$ und $t$ über diesen Schnitt geflossen sein muss. Daraus folgt, dass der Flusswert maximal so groß sein kann wie die Kapazität dieses Schnittes.
Nun betrachtet man den Fluss $f$. Wenn dieser nicht maximal wäre, gäbe es noch irgendeine Möglichkeit, den Fluss über irgendeinen Pfad zu erhöhen. Wäre $f$ nun maximal, kann diese Eigenschaft folglich nicht mehr auftreten, d. h. über einen Schnitt ist die Kapazität des Schnittes gleich dem darüber fließenden Fluss.
Da der Fluss über dem Schnitt stets gleich ist, folgt daraus, dass es sich hierbei um den minimalen Schnitt handelt. Dadurch ist die Behauptung belegt. □

Zum Verständnis des im nächsten Abschnitt behandelten Algorithmus müssen noch die Begriffe *Residualnetzwerk* und *augmentierender Pfad* definiert werden.

**Residualnetzwerk:** Es sei $G$ der ursprüngliche Graph und $f$ ein Fluss auf diesem. Des weiteren sei $e$ eine Kante von $v$ zu $w$ in $G$ mit der Kapazität $c(e)$. Dann ist das Residualnetzwerk wie folgt definiert:

- Zunächst wird der ursprüngliche Graph mit allen Knoten und Kanten kopiert.
- Für jede Kante, auf der der Fluss $> 0$ ist, wird eine Rückkante (von $w$ nach $v$) in das Residualnetzwerk eingefügt. Man setze die Kapazität dieser Rückkante auf $f(e)$. Sofern die Rückkante schon existiert, addiere man $f(e)$ zur bisherigen Kapazität.
- Wenn $f(e) < u(e)$, setze man für die Kante $e$ die Kapazität $u(e) - f(e)$

Somit gibt im Residualnetzwerk die Kapazität einer Hinkante an, um wie viel der Fluss auf ihr noch erhöht werden darf, während die Kapazität einer Rückkante angibt, um wie viel der Fluss auf der zugehörigen Hinkante verringert werden darf. Im im letzten Abschnitt beschriebenen, hier benutzten Netzwerk existiert offensichtlich zu jeder Kante schon eine Rückkante, die gemäß der obigen Definition auch im Residualnetzwerk als solche dient.

**augmentierender Pfad:** Unter Augmentieren versteht man grundsätzlich das Erhöhen des Flusses über eine Kante. Ein augmentierender Pfad ist somit ein Pfad zwischen $s$ und $t$, über den der Fluss erhöht werden kann. Das Residualnetzwerk hilft dabei, noch nicht ausgelastete augmentierende Pfade zu finden.

## 1.3 Mögliche Umsetzung durch den Edmonds-Karp-Algorithmus

Grundsätzlich muss zur Bestimmung eines minimalen Schnitts zuerst der maximale Fluss berechnet werden. Aus dem dabei entstandenen Residualnetzwerk lässt sich schließlich der minimale Schnitt ableiten.

Zur Berechnung des maximalen Flusses habe ich den Edmonds-Karp-Algorithmus gewählt.

Dieser ist einfach und leicht verständlich und obwohl alternative Algorithmen mit einer teilweise schnelleren Laufzeit existieren, ist der Edmonds-Karp-Algorithmus für die Bearbeitung dieser Aufgabe völlig ausreichend. Eine Übersicht über alternative Algorithmen findet man unter [6]. Die Laufzeit des Edmonds-Karp-Algoritmus wird in Teilkapitel 1.5 bestimmt.

Grundlegend entspricht der Edmonds-Karp-Algoritmus dem Algorithmus von Ford und Fulkerson, weshalb dieser zuerst besprochen wird. Er arbeitet wie folgt:

1. Initialisiere mit dem Nullfluss, d.h. setze für jede Kante $e \in E$ $f(e) = 0$.
2. Solange es im Residualnetzwerk $G_f$ einen Pfad von $s$ nach $t$ gibt, bestimme einen solchen Pfad $P$ und tue:
   a) Bestimme $\gamma$ als Minimum der Residualkapazitäten auf $P$.
   b) Augmentiere diesen Pfad soweit wie möglich, d.h. setze für alle Kanten auf $P$ $f(e) = f(e) + \gamma$ und für alle Rückkanten zu diesen setze $f(e) = f(e) - \gamma$.

Der Ford-Fulkerson-Algorithmus lässt offen, wie ein solcher augmentierender Pfad bestimmt wird. Der Edmonds-Karp-Algorithmus legt fest, dass stets der kürzeste Pfad (nach der Anzahl der enthaltenen Kanten) genutzt wird. In den meisten Implementierungen geschieht dies über eine Breitensuche oder eine Abwandlung dieser. So wird auch in meiner Implementierung eine bidirektionale Breitensuche genutzt, die gleichzeitig von $s$ und von $t$ aus sucht und genau dann einen Pfad gefunden hat, wenn beide Teile sich an einem Knoten treffen. Damit lässt sich die Laufzeit der Suche weiter reduzieren.

Sobald der Edmonds-Karp-Algorithmus terminiert, kann aus dem Residualgraphen einfach der minimale Schnitt bestimmt werden. Alle Kanten, die im Residualgraphen eine Kapazität von 0 haben, beschreiben den minimalen Schnitt. Auf diesen entspricht im ursprünglichen Graphen der Fluss der Kapazität, sie sind also „gesättigt“. Da durch den minimalen Schnitt (s. o.) der gesamte Fluss fließen muss und, da es sich um einen minimalen Schnitt handelt, diese Kanten auch maximal ausgelastet sein müssen, entspricht der minimale Schnitt genau den im ursprünglichen Graphen gesättigten Kanten.

Wenn man nun diese Kanten aus dem Graphen entfernt, bilden diejenigen Knoten, die von $s$ erreichbar sind, die erste Teilmenge. Alle restlichen Knoten bilden die zweite Teilmenge. Die Erreichbarkeit lässt sich einfach per Breitensuche bestimmen.

## 1.4 Nutzung des minimalen Schnitts zur Lösung des Problems

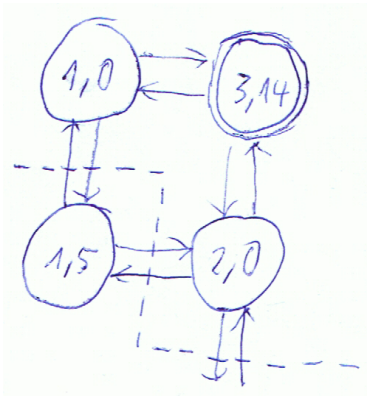
Hat man nun den minimalen Schnitt des Graphen bestimmt, kann man aus diesem die notwendigen Erdarbeiten ableiten. Hier gibt es zunächst einmal den Fall, dass der minimale Schnitt einen Wert von 0 hat. Dann hat der maximale Fluss denselben Wert, was bedeutet, dass kein Fluss zwischen Oberseite ($s$) und Unterseite ($t$) möglich ist. Also ist die Matrix schon unpassierbar und muss nicht mehr verändert werden.

Ist dies nicht der Fall, so sind offensichtlich noch Erdarbeiten nötig. Um zu bestimmen, wo diese stattfinden müssen, benötigt man die Kanten des minimalen Schnitts, also alle Kanten, die von der einen in die andere Teilmenge verlaufen.

Entlang jeder dieser Kanten muss Erde verschoben werden. Dabei wird vom Planquadrat

mit der geringeren Höhe zum höheren Planquadrat genau die Hälfte der oben bestimmten Kantenkapazität $1 - \Delta h$ an Erde verschoben. [4] Dadurch erhöht sich die Höhe des höheren Planquadrats um $\frac{1-\Delta h}{2}$ und die Höhe des niedrigeren Planquadrats sinkt um ebensoviel. Dadurch erhöht sich $\Delta h$ auf 1, und dieser Übergang ist nicht mehr passierbar.

Diese Vorgehensweise führt jedoch in einem Sonderfall nicht zum Erfolg. Wenn der minimale Schnitt wie im folgenden Bild dargestellt, also quasi treppenförmig, verläuft, kann dies unter Umständen dazu führen, dass das Brachland immer noch passierbar bleibt. Dies ist genau dann möglich, wenn in dieser Konstellation in der originalen Matrix drei Planquadrate existieren, sodass sich das höchste und das niedrigste Quadrat auf einer Seite des Schnitts befinden, aber das Quadrat mit mittlerer Höhe auf der anderen Seite liegt. Zusätzlich müssen Kanten, also Übergangsmöglichkeiten vom niedrigsten zum mittleren und vom mittleren zum höchsten Quadrat bestehen.




Dann nämlich wird der oben dargestellte Schritt beispielsweise zwischen dem mittleren und den niedrigsten Quadrat ausgeführt, jedoch erfolgt der gleiche Schritt auch zwischen dem mittleren und dem höchsten Quadrat, was den vorherigen Schritt wieder aufhebt und eine neue Passiermöglichkeit schafft. Im Beispiel verändert sich die Höhe des niedrigsten Quadrats von 1,0 auf 0,75 und die des mittleren Quadrats von 1,5 auf 1,75. Anschließend wird dann jedoch die Höhe des mittleren Quadrats auf 1,375 geändert (und $1,375 - 0,75 < 1$ !) und die des höchsten Quadrats auf 2,375. Um dies zu vermeiden, führt man den ersten Schritt normal aus, erreicht jedoch die benötigte Höhenänderung im zweiten Schritt dadurch, dass man Erde von einem anderem Nachbarquadrat des höchsten Quadrats verschiebt. Dieses Nachbarquadrat sollte sinnvollerweise nicht dem Endknoten einer Kante des Schnitts entsprechen. Im obigen Beispiel könnte dies der doppelt umrandete Knoten sein.

## 1.5 Laufzeitanalyse des Algorithmus

Es ist leicht zu erkennen, dass der Edmonds-Karp-Algorithmus den größten Anteil zur Laufzeit beiträgt. Alle anderen Operationen sind nicht der Rede wert. Zunächst ist leicht zu erkennen, dass die Anzahl der Knoten im Graphen $|V| = n^2 + 2$ ist (abhängig von der Seitenlänge der Matrix $n$), da die Matrix quadratisch ist und somit $n^2$ Planquadrate enthält. Zusätzlich kommen noch $s$ und $t$ dazu, welche jedoch im folgenden vernachlässigbar sind.

[4] Diese muss hier neu bestimmt werden, da sich diese aufgrund vorheriger Veränderungen an der Matrix geändert haben könnte.

Weiterhin befinden sich höchstens $|E| = 2 \cdot n + 2 \cdot n \cdot (n-1) = 2 \cdot n + 2 \cdot n^2 - 2 \cdot n = 2 \cdot n^2$ Kanten im Graphen. Im Einzelnen sind dies die jeweils $n$ Kanten, die $s$ als Ausgangspunkt bzw. die $t$ als Endpunkt haben. Weiterhin befinden sich in der Matrix maximal $n \cdot (n-1)$ Kanten von links nach rechts und ebensoviele von oben nach unten. Die doppelt vorhandenen Kanten (in beide Richtungen) werden im Algorithmus als eine Kante und demzufolge hier auch so gezählt.

Der Edmonds-Karp-Algorithmus hat nach Literaturangaben eine Laufzeit in $\mathcal{O}(|V| \cdot |E|^2)$. Setzt man nun die Werte für $|V|$ und $|E|$ ein, so ergibt sich eine Laufzeit von $\mathcal{O}(4 \cdot n^2 \cdot n^4)$. Da konstante Faktoren vernachlässigbar sind, erhält man eine Laufzeit in $\mathcal{O}(n^6)$. Diese Laufzeit ist für die hier zu lösenden Beispiele mit Seitenlängen bis 20 absolut ausreichend und kann selbst größere Graphen in einer annehmbaren Laufzeit lösen. Hier ließe sich natürlich durch Max-Flow-Algorithmen mit geringerer Laufzeit noch eine Verbesserung erreichen.

Die genannte Laufzeit von $\mathcal{O}(|V| \cdot |E|^2)$ lässt sich folgendermaßen nachweisen:

*Nachweis.* Zunächst kann man zeigen, dass die Anzahl der zu augmentierenden Pfade des Edmonds-Karp-Algorithmus höchstens $\frac{|V| \cdot |E|}{2}$ beträgt. Dabei gibt es beim Augmentieren eines Pfades mindestens immer eine Kante, die gänzlich aus dem Residualnetzwerk gelöscht wird, da entweder eine Vorwärtskante vorkommt, die vollkommen gefüllt wird oder auch eine Rückwärtskante, die im Gegenzug geleert wird. Solch eine Kante bezeichne man als kritische Kante. Jedes Mal, wenn solch eine Kante eine kritische Kante ist, muss sich die Länge des durch diese Kante augmentierenden Pfades um 2 erhöhen. Da solch ein augmentierender Pfad höchstens die Länge n hat, können auf jeder Kante maximal $\frac{|V|}{2}$ augmentierende Pfade liegen, sodass die Gesamtzahl der augmentierenden Pfade höchstens $\frac{|V| \cdot |E|}{2}$ beträgt. Da die Breitensuche höchstens $|E|$ Kanten untersucht, folgt, dass der Algorithmus eine Laufzeit von $\mathcal{O}(|V| \cdot |E|^2)$ hat. □

## 1.6 Erweiterung

Nehmen wir einmal an, die Wildschweine hätten eine Art von Intelligenz entwickelt und könnten einen Computer bedienen. Dieses Wissen wollen sie nun nutzen, um das Brachland zu passieren und dabei einen möglichst geringen Gesamthöhenunterschied zu überwinden. Weiterhin könnte für sie eventuell der größte Einzelhöhenunterschied auf diesem Weg von Interesse sein, denn dann könnten sie entsprechend dafür trainieren, das Brachland passieren zu können. Sie könnten dabei eventuell am Zustand vor den Erdarbeiten ebenso wie am Zustand nach den Erdarbeiten interessiert sein.

Zur Lösung dieses Problems kann man die oben dargestellte Formulierung als Graph im Prinzip weiterverwenden. Es müssen dabei jedoch folgende Änderungen vorgenommen werden:

- Kanten zu benachbarten Knoten erhalten nun das Gewicht $\Delta h$, wobei $\Delta h$ weiterhin der Betrag des Höhenunterschieds ist. Auch werden solche Kanten immer hinzugefügt, unabhängig von ihrem Gewicht.
- Kanten, welche vorher das Gewicht $\infty$ hatten, erhalten nun das Gewicht 0.

Nun entspricht das Problem dieser Erweiterung dem Problem des kürzesten Pfades zwischen zwei Knoten in einem Graphen (in diesem Fall $s$ und $t$). Dieses Problem kann bei-

spielsweise durch den Dijkstra-Algorithmus gelöst werden. Dieser arbeitet grundlegend wie folgt:

1. Weise allen Knoten die beiden Eigenschaften „Distanz“ und „Vorgänger“ zu. Initialisiere die Distanz im Startknoten $s$ mit 0 und in allen anderen Knoten mit $\infty$.
2. Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen mit minimaler Distanz aus und
   a) speichere, dass dieser Knoten schon besucht wurde.
   b) Berechne für alle noch unbesuchten Nachbarknoten die Summe des jeweiligen Kantengewichtes und der Distanz im aktuellen Knoten.
   c) Ist dieser Wert für einen Knoten kleiner als die dort gespeicherte Distanz, aktualisiere sie und setze den aktuellen Knoten als Vorgänger.

Wenn hierbei der gesuchte Knoten $t$ im Schritt 2 der aktive ist, kann dann schon abgebrochen werden. Der Dijkstra-Algorithmus hat eine Laufzeit von $\mathcal{O}(|V|^2)$ (bei Implementierung mit einer Vorrangwarteschlange), was gemäß den obigen Berechnungen $\mathcal{O}(n^4)$ entspricht.

Da der Algorithmus den entsprechenden Weg durch das Setzen eines Vorgängers ebenfalls bestimmt, kann auch dieser selbst ausgegeben werden. Das Bestimmen der Einzelkante mit dem größten Gewicht auf diesem Pfad erfolgt dann durch Iterieren über alle Kanten und die Suche des Maximum von deren Gewichten.

Eine weitere mögliche Erweiterung wäre es, die Zäune an der linken und rechten Seite des Brachlands zu „öffnen“und zuzulassen, dass Wildschweine auch von links kommen sowie das Brachland nach rechts verlassen können. Dazu müsste man $s$ mit allen Knoten in der ersten Spalte und alle Knoten in der letzten Spalte mit $t$ verbinden. Die Kantengewichte werden wie beschrieben verteilt, anschließend könnte man den Edmonds-Karp-Algorithmus normal ausführen. Diese Erweiterung wurde nicht implementiert.

# Literatur

[1] Antti Laaksonen: Competitive Programmer's Handbook, Kapitel 20 behandelt Flüsse, `https://cses.fi/book.html`

[2] Steven S. Skiena: The Algorithm Design Manual, ISBN 978-1-84800-069-8, Kapitel 6.5 und 15.8 bzw. 15.9 beschäftigen sich mit Fluss- und Schnittproblemen

[3] Steven Halim und Felix Halim: Competitive Programming, Kapitel 4.8 beschäftigt sich mit dem Edmonds-Karp-Algorithmus, `http://www.comp.nus.edu.sg/~stevenha/myteaching/competitive_programming/cp1.pdf`

[4] Thomas Ottmann und Peter Widmayer: Algorithmen und Datenstrukturen, ISBN 978-3-662-55649-8, Kapitel 9.7 beschäftigt sich mit Flüssen

[5] Wikipedia-Artikel zum Ford und Fulkerson-Algorithmus, `https://de.wikipedia.org/wiki/Algorithmus_von_Ford_und_Fulkerson`

[6] Wikipedia-Artikel zur allgemeinen Übersicht über Flüsse in Netzwerken, `https://de.wikipedia.org/wiki/Fl%C3%BCsse_und_Schnitte_in_Netzwerken`

[7] Wikipedia-Artikel zum Maximum Flow Problem, `https://en.wikipedia.org/wiki/Maximum_flow_problem`

[8] Wikipedia-Artikel zum Max-Flow-Min-Cut-Theorem, `https://de.wikipedia.org/wiki/Max-Flow-Min-Cut-Theorem`

[9] Dokument zum Max-Flow-Problem für mehrere Quellen und Senken, `http://www.ifp.illinois.edu/~angelia/ge330fall09_maxflowl20.pdf`

[10] Wikipedia-Artikel zum Dijkstra-Algorithmus, `https://de.wikipedia.org/wiki/Dijkstra-Algorithmus`

# 2 Umsetzung

## 2.1 Allgemeines zur Bedienung des Programms

Das Programm wurde in Python 3.6.3 implementiert. Zur Verwaltung eines Graphen benutze ich die Python-Bibliothek `NetworkX`, welche über `pip3 install networkx` installiert werden kann. Grundlegend benutzt diese Adjazenzlisten, abstrahiert den Zugriff auf den Graphen jedoch ein wenig, sodass sich viele Operationen leichter durchführen lassen. Alle weiteren verwendeten Bibliotheken sind üblicherweise vorinstalliert. Die Eingabe des Programms erfolgt über eine Abfrage des Eingabedateinamens auf der Konsole, die Ausgabe erfolgt dann automatisch in eine entsprechende Datei mit dem vorgegebenen Dateinamen. Als kleiner Zusatz ist es möglich, die unüberwindbare Steigung zu verändern (entsprechend Unterabschnitt 1.1), was zu Beginn abgefragt wird.

## 2.2 Umsetzung des Edmonds-Karp-Algorithmus

Im Modul `boarflow.py` sind folgende relevanten Funktionen zur Berechnung des minimalen Schnitts enthalten.

| Funktion | Beschreibung |
|---|---|
| breadth_first_search() | bestimmt mithilfe einer Breitensuche die von einem bestimmten Knoten aus erreichbaren Knoten. Dabei werden schon gesehene Knoten in einem Dictionary gespeichert, ebenso wie Knoten auf der aktuellen und der nächsten Ebene. Der Algorithmus besucht dann für jede Ebene alle noch nicht gesehenen Knoten und fügt deren Nachbarn als Besuchskandidaten für die nächste Ebene hinzu. Die Funktion wird dazu genutzt, die von der Senke aus erreichbare Teilmenge der Knoten zu bestimmen. |
| build_residual_network() | erstellt die Anfangsform des Residualnetzwerks. Dazu werden alle Knoten kopiert sowie alle Kanten mit den entsprechenden Kapazitäten ebenfalls. Zu allen Kanten werden dann Rückkanten mit einer Kapazität von 0 bzw. falls diese Rückkanten im ursprünglichen Graphen schon existieren, der entsprechenden Kapazität, eingefügt. Falls für eine Kante keine Kapazität angegeben ist, wird dafür $\infty$ angenommen. Für alle Kanten wird mit dem Nullfluss initialisiert. |

| Funktion | Beschreibung |
|---|---|
| edmonds_karp() | enthält den eigentlichen Edmonds-Karp-Algorithmus. Das hier genutzte Residualnetzwerk arbeitet dabei nicht mit Residualkapazitäten im klassischen Sinne, sondern kommt mit der Speicherung von normaler Kapazität und dem Fluss aus. In einer Schleife wird immer wieder eine bidirektionale Breitensuche nach einem augmentierenden Pfad gestartet. Solange diese Ergebnisse liefert, werden beide Teile des Pfades zu einem Pfad zusammengefügt. Dieser wird anschließend augmentiert. |
| augment() | ist eine Unterfunktion von edmonds_karp(). Sie wird dazu genutzt, einen Pfad zu augmentieren. Dazu wird zuerst das Minimum der Residualkapazitäten bestimmt, welche hierfür als $c(e) - f(e)$ bestimmt werden. Dann wird der Fluss auf den genutzten Hinkanten entsprechend erhöht und auf den Rückkanten dazu verringert. |
| bidirectional_bfs() | ist eine Unterfunktion von edmonds_karp(). Diese bidirektionale Breitensuche sucht nach einem augmentierenden Pfad. Die Beschränkung auf augmentierende Pfade wird dabei dadurch erreicht, dass nur diejenigen Kanten benutzt werden dürfen, auf denen der Fluss kleiner als die Kapazität ist. Der Grund für die Benutzung einer solchen Suche in diesem Programm ist, dass dadurch die Laufzeit des Algorithmus verbessert werden kann. Es muss dann im Prinzip nur zweimal die Hälfte des Graphen durchsucht werden, was schneller möglich ist, als einmal den ganzen Graphen zu durchsuchen. |
| minimum_cut() | In dieser Funktion wird der minimaler Schnitt des Graphen bestimmt. Dazu werden alle gesättigten Kanten entfernt ($f(e) = c(e)$) und dann per Breitensuche die Erreichbarkeit von der Quelle aus bestimmt. Diese Information wird genutzt, um den Graphen in zwei Teilmengen/Partitionen zu teilen. |

## 2.3 Implementierung des Hauptprogramms

Im Hauptprogramm `wildschweine.py` befinden sich bis auf einige Ausnahmen nahezu alle Anweisungen außerhalb von Funktionen, werden also einfach hintereinander ausgeführt. Zunächst wird die Eingabedatei eingelesen (dazu wird die Funktion `read()` benutzt) und die zurückgegebene Matrix wird kopiert, da sie im Programm verändert wird, jedoch später in ihrer ursprünglichen Fassung noch gebraucht wird. Nun wird die Zeitmessung gestartet.

Daraufhin wird der Graph wie in Unterabschnitt 1.1 beschrieben erstellt. Dabei werden die Knoten im Format „$x|y$“benannt, um sie später wiederfinden zu können. Dabei ist $x$ die Reihe in der Matrix, beginnend bei 0 und $y$ die Spalte in der Matrix, ebenfalls beginnend bei 0. $s$ und $t$ sind entsprechend den Konventionen Quelle und Senke. Bei Kanten, deren Kapazität $\infty$ ist, wird diese, wie bereits beschrieben, nicht angegeben. Zu beachten ist hierbei auch, dass Kanten nur dann eingefügt werden dürfen, wenn sie auch wirklich passierbar wären (Kapazität $> 0$). Dies wird aufgrund der Ungenauigkeit von Gleitpunktzahlen durch Vergleich mit einem $\varepsilon$, hier $10^{-4}$ realisiert.

Anschließend wird der minimale Schnitt berechnet. Wenn dieser 0 ist, ist die Matrix schon unpassierbar und es kann abgebrochen werden. Sonst werden durch Iteration über alle Kanten diejenigen Kanten bestimmt, die von der ersten Teilmenge $S$ in die zweite Teilmenge $T$ führen. Diese sind die Kanten des minimalen Schnitts, die im folgenden benötigt werden.

Nun wird gemäß der generellen Regel zum kostengünstigsten Verschieben (Unterabschnitt 1.4) Erde verschoben. Dabei wird gleichzeitig gezählt, wie viel Erde verschoben wurde. Sollte zufällig über eine Kante weniger als keine Erde verschoben werden müssen, wurde diese Kante schon durch Erdarbeiten über eine andere Kante unpassierbar gemacht. Deshalb kann eine solche Kante ignoriert werden. Weiterhin werden zwei Listen geführt, die für jede Kante den jeweils höheren und den tieferen Knoten protokollieren.

Nun muss noch der Sonderfall berücksichtigt werden. Dazu geht man alle Kanten durch und untersucht, ob der hier tiefere Knoten schon einmal ein höherer Knoten war. Dann entnimmt man der Liste der tieferen Knoten den zu diesem höheren Knoten gehörenden tieferen Knoten. Diese drei Knoten (hier höherer Knoten, hier tieferer Knoten und der Knoten aus der Liste) entsprechen den in Unterabschnitt 1.4 genannten drei Planquadraten. Bei diesen stellt man nun die ursprüngliche Höhe wieder her und sortiert sie anschließend nach dieser. Dann wird entsprechend die Verschiebung zwischen tiefstem und mittlerem Planquadrat normal ausgeführt. Für die nötige Verschiebung zwischen mittlerem und höchstem Planquadrat wird mithilfe der Funktion `auswahl()` ein Nachbarfeld des höchsten Felds herangezogen, welches nicht am Schnitt und damit an Erdarbeiten beteiligt ist.

Nun hat man eine unpassierbare Lösungsmatrix, welche ausgegeben wird. Außerdem wird die verschobene Erde (in cm), welchen den Kosten entspricht, sowie die benötigte Zeit ausgegeben.

## 2.4 Implementierung der Erweiterung

In der Datei zur Erweiterung `dijkstra.py` befindet sich die Funktion `shortest_path()`. Diese erhält als Eingabe eine beliebige Matrix mit den Höhen. Anschließend wird gemäß den bereits genannten Regeln der Graph aufgebaut. Der Dijkstra-Algorithmus wird dann ausgeführt und gibt den kürzesten Weg und dessen Länge zurück. Nun wird durch Iterieren über alle Kanten des Weges diejenige mit minimalem Gewicht (entsprechend der minimalen Steigung) bestimmt. Anschließend werden minimale Steigung, Gesamtsteigung und der Weg selbst ausgegeben. Die Funktion wird für jede Eingabedatei sowohl für die Eingabematrix als auch für die Ausgabematrix aufgerufen.

Der Dijkstra-Algorithmus selbst befindet sich in der Funktion `dijkstra_st()`. Er arbeitet nach dem oben bereits beschriebenen Prinzip. Zur effizienten Implementierung wird hier eine Vorrangwarteschlange bzw. Priority Queue aus der Bibliothek `heapq` verwendet.

# 3 Beispiele

wildschwein1-lsg.txt

```
20
2.317 2.366 2.421 2.478 2.530 2.568 2.585 2.576 2.544 2.499 2.448 2.399 2.354 2.315 2.282 2.253 2.228 2.207 2.188 2.172
2.367 2.436 2.517 2.607 2.696 2.766 2.795 2.775 2.714 2.634 2.551 2.476 2.413 2.360 2.317 2.282 2.252 2.227 2.206 2.187
2.422 2.518 2.640 2.788 2.949 3.085 3.143 3.096 2.971 2.821 2.682 2.568 2.479 2.410 2.357 2.314 2.279 2.249 2.224 2.203
2.480 2.609 2.789 3.033 3.338 3.633 3.769 3.648 3.366 3.075 2.842 2.673 2.552 2.464 2.398 2.348 2.307 2.273 2.244 2.220
2.533 2.700 2.951 3.339 3.916 4.606 4.973 4.625 3.952 3.393 3.020 2.781 2.625 2.518 2.442 2.384 2.337 2.299 2.267 2.239
2.574 2.771 3.089 3.637 4.609 6.120 7.132 6.144 4.656 3.707 3.179 2.877 2.692 2.572 2.486 2.422 2.371 2.328 2.291 2.259
2.592 2.802 3.150 3.775 4.978 7.135 8.817 7.167 5.041 3.867 3.268 2.942 2.748 2.623 2.534 2.465 2.409 2.361 2.320 2.283
2.585 2.784 3.106 3.658 4.635 6.153 7.172 6.194 4.717 3.779 3.263 2.971 2.794 2.676 2.588 2.517 2.455 2.401 2.353 2.310
3.034 3.188 3.412 3.739 4.193 4.702 5.055 4.766 4.321 3.930 3.197 2.982 2.842 2.742 2.659 2.584 2.515 2.450 2.392 2.341
2.034 2.188 2.412 2.739 3.193 3.702 3.894 3.766 3.321 2.530 3.530 3.006 2.917 2.841 2.763 2.680 2.595 2.514 2.441 2.377
2.465 2.571 2.706 2.871 3.053 3.217 3.312 3.313 3.247 3.641 2.641 3.080 3.053 3.008 2.931 2.825 2.709 2.597 2.500 2.419
2.419 2.500 2.597 2.709 2.825 2.931 3.008 3.053 3.080 2.641 3.641 3.247 3.313 3.312 3.217 3.053 2.871 2.706 2.571 2.465
2.377 2.441 2.514 2.595 2.680 2.763 2.841 2.917 3.006 2.723 4.134 3.321 3.766 3.894 3.702 3.193 2.739 2.412 2.188 2.034
2.341 2.392 2.450 2.515 2.584 2.659 2.742 2.842 2.982 3.197 3.134 4.321 4.766 5.055 4.702 4.193 3.739 3.412 3.188 3.034
2.310 2.353 2.401 2.455 2.517 2.588 2.676 2.794 2.971 3.263 3.779 4.717 6.194 7.172 6.153 4.635 3.658 3.106 2.784 2.585
2.283 2.320 2.361 2.409 2.465 2.534 2.623 2.748 2.942 3.268 3.867 5.041 7.167 8.817 7.135 4.978 3.775 3.150 2.802 2.592
2.259 2.291 2.328 2.371 2.422 2.486 2.572 2.692 2.877 3.179 3.707 4.656 6.144 7.132 6.120 4.609 3.637 3.089 2.771 2.574
2.239 2.267 2.299 2.337 2.384 2.442 2.518 2.625 2.781 3.020 3.393 3.952 4.625 4.973 4.606 3.916 3.339 2.951 2.700 2.533
2.220 2.244 2.273 2.307 2.348 2.398 2.464 2.552 2.673 2.842 3.075 3.366 3.648 3.769 3.633 3.338 3.033 2.789 2.609 2.480
2.203 2.224 2.249 2.279 2.314 2.357 2.410 2.479 2.568 2.682 2.821 2.971 3.096 3.143 3.085 2.949 2.788 2.640 2.518 2.422
```

wildschwein2-lsg.txt

```
12
2.500 2.500 2.500 2.500 2.500 2.500 2.500 2.500 2.500 2.500 2.500 2.500
1.500 1.500 1.500 1.500 1.500 1.500 1.500 1.500 1.500 1.500 1.500 1.500
2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000
2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000
2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000
2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000
2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000
```

```
2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000
2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000
2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000
2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000
2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000 2.000
```

wildschwein3-lsg.txt

```
10
3.042 2.500 3.042 3.042 3.042 2.500 3.042 3.042 3.042 2.500
3.065 1.500 3.065 3.065 3.065 1.500 3.065 3.065 3.065 1.500
3.110 2.000 3.110 3.110 3.110 2.000 3.110 3.110 3.110 2.000
3.220 2.000 3.220 3.220 3.220 2.000 3.220 3.220 3.220 2.000
3.325 2.000 3.325 3.325 3.325 2.000 3.325 3.325 3.325 2.000
4.325 2.000 4.325 4.325 4.325 2.000 4.325 4.325 4.325 2.000
3.550 2.000 3.550 3.550 3.550 2.000 3.550 3.550 3.550 2.000
3.220 2.000 3.220 3.220 3.220 2.000 3.220 3.220 3.220 2.000
3.110 2.000 3.110 3.110 3.110 2.000 3.110 3.110 3.110 2.000
3.065 2.000 3.065 3.065 3.065 2.000 3.065 3.065 3.065 2.000
```

wildschwein4-lsg.txt

```
15
3.731 3.410 3.208 3.333 3.968 3.006 3.964 3.940 3.947 3.937 3.397 3.348 3.294 3.506
↪ 3.116
3.771 3.660 3.157 3.378 3.140 3.695 3.805 3.005 3.523 3.744 3.142 3.482 3.545 3.577
↪ 3.205
3.623 3.185 3.011 3.161 3.178 3.540 3.974 3.245 3.395 3.218 3.432 3.233 3.890 3.038
↪ 3.592
3.655 3.120 3.652 3.984 3.207 3.375 3.463 3.334 3.443 3.504 3.999 3.630 3.910 3.508
↪ 3.491
3.206 3.056 3.276 4.062 2.923 3.173 3.549 3.555 3.588 3.789 3.699 3.205 3.255 3.778
↪ 3.225
4.206 4.056 4.277 3.062 3.966 2.792 3.829 2.900 3.976 3.890 3.723 3.509 3.987 3.153
↪ 3.711
3.824 3.124 3.617 3.482 3.100 3.792 2.829 3.900 2.976 2.785 4.122 3.071 3.253 3.956
↪ 3.059
3.599 3.292 3.876 3.744 3.423 3.376 3.271 3.730 3.887 3.044 3.122 4.071 4.253 2.956
↪ 4.059
3.428 3.999 3.062 3.586 3.567 3.662 3.907 3.246 3.248 3.707 3.338 3.607 3.315 3.024
↪ 3.451
3.369 3.464 3.177 3.676 3.716 3.743 3.929 3.590 3.644 3.110 3.775 3.306 3.881 3.630
↪ 3.929
3.197 3.755 3.158 3.042 3.379 3.787 3.713 3.110 3.762 3.030 3.317 3.667 3.426 3.401
↪ 3.744
3.365 3.764 3.323 3.513 3.479 3.131 3.032 3.444 3.692 3.719 3.778 3.288 3.781 3.966
↪ 3.997
3.239 3.445 3.130 3.234 3.656 3.572 3.556 3.250 3.920 3.724 3.777 3.242 3.176 3.445
↪ 3.830
3.602 3.638 3.303 3.065 3.938 3.266 3.229 3.304 3.372 3.857 3.807 3.389 3.265 3.032
↪ 3.670
3.074 3.609 3.179 3.030 3.708 3.278 3.116 3.691 3.349 3.228 3.168 3.595 3.810 3.515
↪ 3.349
```

wildschwein5-lsg.txt

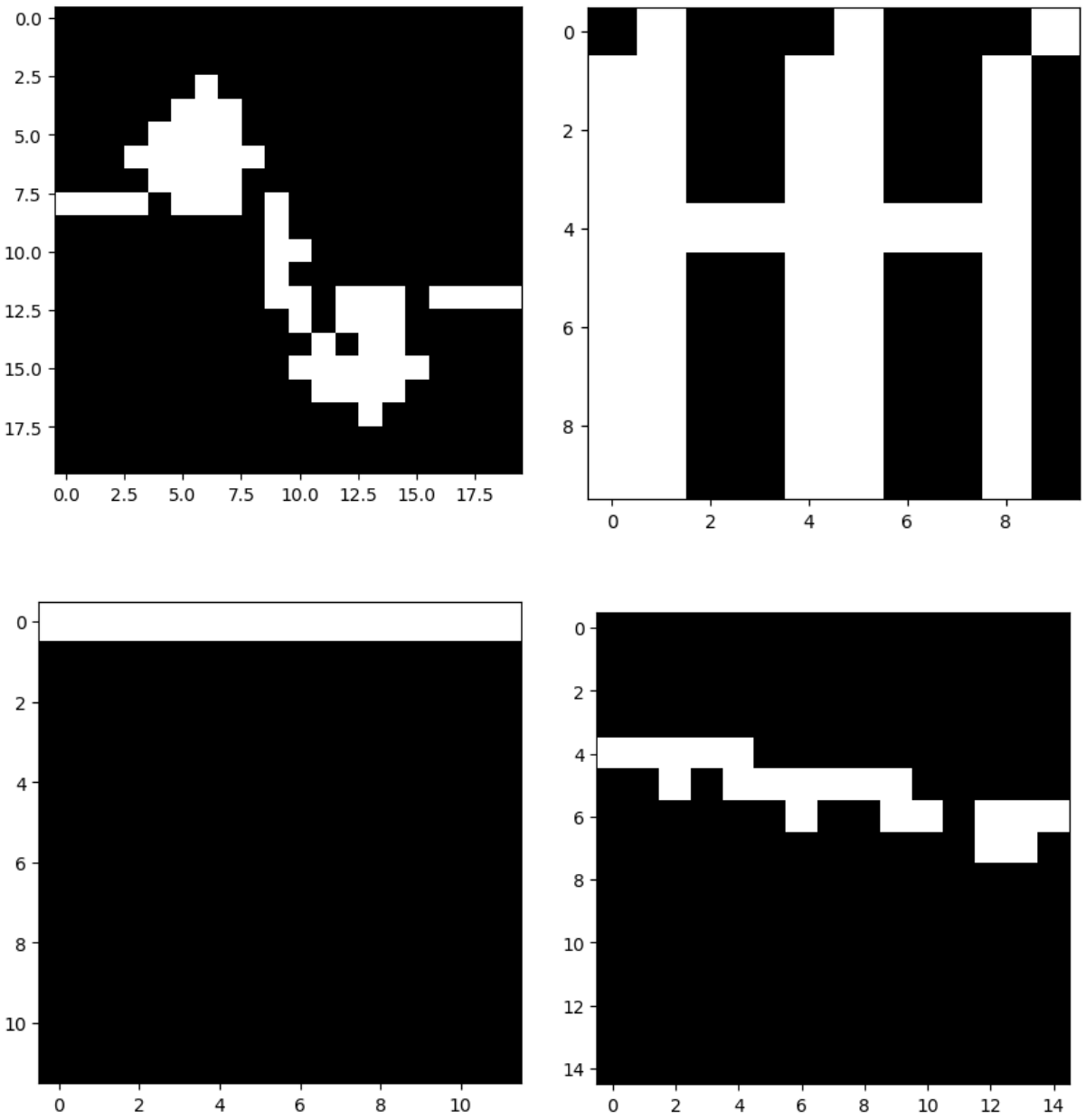
```
20
4.000 4.613 4.863 4.595 3.875 2.965 2.199 1.839 1.953 2.389 2.856 3.064 2.865 2.325 1.697 1.300 1.367 1.936 2.823 3.707
4.448 5.001 5.197 4.883 4.130 3.199 2.430 2.081 2.220 2.695 3.212 3.478 3.340 2.862 2.292 1.946 2.054 2.650 3.551 4.433
4.719 5.217 5.367 5.020 4.247 3.312 2.554 2.231 2.409 2.934 3.508 3.836 3.760 3.340 2.821 2.516 2.651 3.261 4.159 5.023
4.729 5.182 5.298 4.931 4.154 3.231 2.498 2.214 2.442 3.024 3.660 4.049 4.032 3.663 3.185 2.907 3.056 3.663 4.543 5.376
4.450 4.869 4.965 4.594 3.828 2.930 2.237 2.003 2.288 2.932 3.630 4.077 4.110 3.782 3.332 3.067 3.214 3.803 4.652 5.440
3.914 4.313 4.406 4.046 3.305 2.447 1.803 1.626 1.973 2.679 3.435 3.933 4.007 3.707 3.270 3.002 3.131 3.689 4.493 5.228
3.212 3.607 3.710 3.376 2.675 1.866 1.280 1.165 1.573 2.337 3.144 3.683 3.785 3.498 3.058 2.774 2.871 3.384 4.135 4.810
2.471 2.877 3.006 2.711 2.060 1.308 0.784 0.730 1.197 2.012 2.860 3.427 3.541 3.252 2.795 2.478 2.531 2.991 3.682 4.295
1.827 2.259 2.428 2.182 1.588 0.898 0.435 0.440 0.958 1.814 2.689 3.269 3.381 3.074 2.585 2.225 2.224 2.625 3.253 3.806
1.394 1.866 2.084 1.896 1.363 0.735 0.330 0.386 0.945 1.828 2.717 3.294 3.389 3.050 2.517 2.103 2.043 2.381 2.949 3.446
1.240 1.761 2.037 1.910 1.440 0.870 0.516 0.613 1.199 2.095 2.982 3.541 3.604 3.221 2.635 2.161 2.038 2.316 2.829 3.279
1.370 1.949 2.286 2.221 1.809 1.290 0.977 1.101 1.701 2.595 3.464 3.991 4.010 3.574 2.927 2.391 2.208 2.430 2.896 3.311
1.476 2.147 2.575 2.594 2.253 1.789 1.509 1.646 2.235 3.095 3.909 4.367 4.302 3.775 3.331 2.734 2.495 2.670 3.101 3.494
2.476 3.147 3.575 3.594 3.253 2.789 2.510 2.646 3.235 4.095 4.909 5.367 5.302 4.775 3.742 3.089 2.804 2.944 3.353 3.739
2.721 3.481 3.989 4.074 3.781 3.344 3.069 3.187 3.734 4.535 5.274 5.645 5.488 4.957 3.956 3.345 3.024 3.143 3.545 3.941
3.117 3.929 4.477 4.590 4.311 3.871 3.579 3.664 4.168 4.915 5.594 5.903 5.686 5.075 4.075 3.405 3.063 3.174 3.586 4.006
3.336 4.187 4.764 4.890 4.608 4.151 3.826 3.868 4.318 5.006 5.623 5.871 5.598 4.923 3.923 3.213 2.865 2.985 3.421 3.878
3.352 4.232 4.822 4.945 4.646 4.156 3.788 3.776 4.167 4.792 5.348 5.541 5.221 4.500 3.500 2.773 2.434 2.578 3.051 3.557
3.201 4.094 4.681 4.787 4.456 3.923 3.501 3.429 3.757 4.322 4.823 4.968 4.613 4.037 3.037 2.295 1.998 2.204 2.755 3.347
2.962 3.853 4.422 4.497 4.121 3.534 3.053 2.919 3.187 3.696 4.150 4.260 3.883 2.896 2.037 1.295 0.998 1.204 1.755 2.347
```

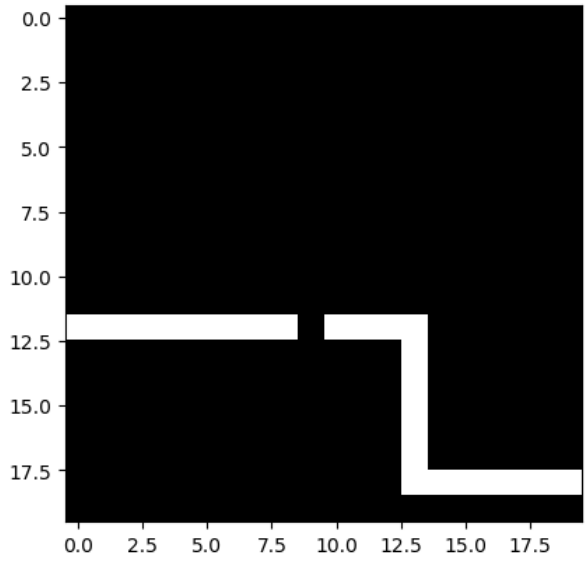
| Beispiel | benötigte Kosten | Laufzeit (ca.) |
|---|---|---|
| wildschwein1.txt | 706.60 Euro | 316 Millisekunden |
| wildschwein2.txt | 600.00 Euro | 19 Millisekunden |
| wildschwein3.txt | 307.50 Euro | 11 Millisekunden |
| wildschwein4.txt | 370.90 Euro | 139 Millisekunden |
| wildschwein5.txt | 405.20 Euro | 261 Millisekunden |
| eigenes Beispiel (50 x 50) test1_50.txt | 1031.00 Euro | 11788 Millisekunden |
| eigenes Beispiel (100 x 100) test2_100.txt | 4564.30 Euro | 156630 Millisekunden |

Somit löst das Programm sämtliche vorgegebenen Beispiele in unter einer Sekunde. Diese

Lösungen sind auch nachweisbar richtig, denn gibt man die Lösungsdateien wieder in das Programm ein, so wird anhand des minimalen Schnitts, welcher dann einen Wert von 0 hat, die Unpassierbarkeit der Lösungsmatrizen festgestellt. Die eigenen Beispiele sind der Implementierung zusammen mit deren Lösungsdateien beigelegt.

Weiterhin wurden mit dem beigelegten Skript `heatmap.py` Heatmaps für die Lösungen aller Beispiele erstellt, die einen Überblick über die Höhen der einzelnen Felder geben. Hierfür wurden `numpy` und `matplotlib` eingesetzt. Außerdem befinden sich im Ordner `extras` Bilder, die für jede Lösung zeigen, wo ungefähr die Barriere für die Wildschweine verläuft (s.u.). Teilweise scheint diese Barriere nicht ganz geschlossen zu sein, was jedoch damit zusammenhängt, dass dafür Daten über die Passierbarkeit von oben nach unten und von links nach rechts zusammengefügt wurden, welche nicht vollständig zueinander passen.
Theoretisch ist es möglich, dass sich in den Ausgabedateien Werte durch Rundungsfehler im Millimeterbereich von den optimalen Werten unterscheiden. Diese Fehler sind jedoch völlig unerheblich.






Links oben: wildschwein1-lsg.txt
Links unten: wildschwein2-lsg.txt
Rechts oben: wildschwein3-lsg.txt
Rechts unten: wildschwein4-lsg.txt
auf dieser Seite: wildschwein5-lsg.txt

## 3.1 Ergebnisse der Erweiterung

Der genaue Weg wurde ausgelassen, da er hier uninteressant ist.

| Beispiel | Gesamt-Höhenunterschied | größte Einzelsteigung |
|---|---|---|
| wildschwein1.txt | 0,59 | 0,058 |
| wildschwein1-lsg.txt | 2,35 | 1,0 |
| wildschwein2.txt | 0,0 | 0,0 |
| wildschwein2-lsg.txt | 1,5 | 1,0 |
| wildschwein3.txt | 0,0 | 0,0 |
| wildschwein3-lsg.txt | 1,178 | 1,11 |
| wildschwein4.txt | 2,508 | 0,554 |
| wildschwein4-lsg.txt | 3,381 | 1,0 |
| wildschwein5.txt | 5,71 | 0,646 |
| wildschwein5-lsg.txt | 5,883 | 1,0 |
| test1_50.txt | 2,012 | 0,624 |
| test1_50-lsg.txt | 2,482 | 1,0 |
| test2_100.txt | 0,0 | 0,0 |
| test2_100-lsg.txt | 2,0 | 1,0 |

# 4 Quellcode

```
1  # Funktion zur Auswahl des Feldes, von dem in Sonderfall (s.u.) Erde genommen wird
2  def auswahl(size, u, bisher_ob, bisher_unt):
3      moeglich_liste = []
4      # direkt oben, unten, links, rechts vom Feld
5      if (int(u[1])-1) >= 0:
6          links = u[0] + "|" + str(int(u[1])-1)
7          moeglich_liste.append(links)
8      if (int(u[1])+1) < size:
9          rechts = u[0] + "|" + str(int(u[1])+1)
10         moeglich_liste.append(rechts)
11     if (int(u[0])+1) < size:
12         unten = str(int(u[0])+1) + "|" + u[1]
```

```
        moeglich_liste.append(unten)
    if (int(u[0])-1) >= 0:
        oben = str(int(u[0])-1) + "|" + u[1]
        moeglich_liste.append(oben)
    for moeglich in moeglich_liste:
        # Feld bisher nicht in Höhe verändert?
        if moeglich in bisher_ob or moeglich in bisher_unt:
            moeglich_liste.remove(moeglich)
    return moeglich_liste[len(moeglich_liste)-1]


filename = input("Bitte Namen der Eingabedatei eingeben: ")
if not os.path.isfile(filename):
    sys.exit(0)

vt = read(filename)
size = vt[0]
vertex_matrix = vt[1]

impossible_height = 1
height_inp = input("Falls gewünscht, unüberwindbare Steigung angeben (Enter=Default/1):")
if height_inp != "":
    impossible_height = int(height_inp)
# "Sicherheitskopie" der Matrix
old_vertex_matrix = copy.deepcopy(vertex_matrix)
time_start = TimestampMillisec64()

graph = nx.DiGraph()
# Quelle und Senke
graph.add_node("s")
graph.add_node("t")

# Knoten für jedes Feld der Matrix
# Benennung der Knoten: ReiheInMatrix|SpalteInMatrix
for i in range(0, size):
    for k in range(0, size):
        graph.add_node(str(i)+"|"+str(k))

for i in range(0, size):
    # Kanten von Quelle in erste Reihe
    graph.add_edge("s", "0|"+str(i))
    # Kanten von letzter Reihe zu Senke
    graph.add_edge(str(size-1)+"|"+str(i), "t")
    # jeweils Kante zu Knoten direkt rechts und unten, ...
    for k in range(0, size):
        if k+1 < size:
            capacity = impossible_height - abs(vertex_matrix[i][k+1] - vertex_matrix[i][k])
            # ... aber nur wenn nötige Höhenänderung größer als 10^-4
            if capacity > epsilon:
                graph.add_edge(str(i)+"|"+str(k), str(i)+"|"+str(k+1), capacity=capacity)
                graph.add_edge(str(i) + "|" + str(k+1), str(i) + "|" + str(k), capacity=capacity)
        if i+1 < size:
            capacity = impossible_height - abs(vertex_matrix[i+1][k] - vertex_matrix[i][k])
            if capacity > epsilon:
```

```
                graph.add_edge(str(i) + "|" + str(k), str(i+1) + "|" + str(k),
                ↪  capacity=capacity)
                graph.add_edge(str(i+1) + "|" + str(k), str(i) + "|" + str(k),
                ↪  capacity=capacity)

# Berechnen des minimalen Schnitts
mincut = boarflow.minimum_cut(graph, "s", "t")
cutedges = []

# wenn Mincut Wert von 0 hat, existiert kein Weg von s nach t
if mincut[0] == 0:
    print("Bereits unpassierbar!")
    sys.exit(0)

# Finden der Mincut-Kanten mithilfe der beiden Partitionen des Graphen
for (u, v) in graph.edges:
    if u in mincut[1][0] and v in mincut[1][1]:
        cutedges.append([u, v])

# Zählvariable für verschobene Erde => Kosten
verschobene_erde = 0.0

bisher_obere_knoten = []
bisher_untere_knoten = []

# für jede Mincut-Kante
for edge in cutedges:
    u = edge[0].split("|")
    v = edge[1].split("|")
    if vertex_matrix[int(u[0])][int(u[1])] <= vertex_matrix[int(v[0])][int(v[1])]:
        # benötigte Höhenänderung
        delta = impossible_height - (vertex_matrix[int(v[0])][int(v[1])] -
        ↪  vertex_matrix[int(u[0])][int(u[1])])
        # Hälfte davon wird von tieferem auf höheres Feld gebracht
        delta = delta / 2
        if delta < 0:
            continue
        verschobene_erde += delta
        vertex_matrix[int(u[0])][int(u[1])] -= delta
        vertex_matrix[int(v[0])][int(v[1])] += delta
        bisher_obere_knoten.append(edge[1])
        bisher_untere_knoten.append(edge[0])
    elif vertex_matrix[int(u[0])][int(u[1])] > vertex_matrix[int(v[0])][int(v[1])]:
        # gleiches Prinzip
        delta = impossible_height - (vertex_matrix[int(u[0])][int(u[1])] -
        ↪  vertex_matrix[int(v[0])][int(v[1])])
        delta = delta / 2
        if delta < 0:
            continue
        verschobene_erde += delta
        vertex_matrix[int(u[0])][int(u[1])] += delta
        vertex_matrix[int(v[0])][int(v[1])] -= delta
        bisher_obere_knoten.append(edge[0])
        bisher_untere_knoten.append(edge[1])

# Behandlung des Sonderfalls (treppenartiger Mincut) => siehe Doku
for edge in cutedges:
    u = edge[0].split("|")
    v = edge[1].split("|")
```

```
    do_tausch = False
    if vertex_matrix[int(u[0])][int(u[1])] <= vertex_matrix[int(v[0])][int(v[1])]:
        # hier tieferes Feld war schon mal höheres Feld -> Änderung nötig
        if edge[0] in bisher_obere_knoten:
            index = bisher_obere_knoten.index(edge[0])
            unterer_knoten = bisher_untere_knoten[index]
            do_tausch = True
    elif vertex_matrix[int(u[0])][int(u[1])] > vertex_matrix[int(v[0])][int(v[1])]:
        if edge[1] in bisher_obere_knoten:
            index = bisher_obere_knoten.index(edge[1])
            unterer_knoten = bisher_untere_knoten[index]
            do_tausch = True
    # Änderung durchführen
    if do_tausch:
        # Sortieren der drei Felder nach ursprünglicher Höhe
        list_knoten = [edge[0], edge[1], unterer_knoten]
        list_knoten = sorted(list_knoten, key=lambda knoten:
        ↪  old_vertex_matrix[int(knoten.split("|")[0])][int(knoten.split("|")[1])])
        knoten_unten = list_knoten[0].split("|")
        knoten_mitte = list_knoten[1].split("|")
        knoten_oben = list_knoten[2].split("|")
        # Wiederherstellen der ursprünglichen Höhe
        vertex_matrix[int(knoten_unten[0])][int(knoten_unten[1])] =
        ↪  old_vertex_matrix[int(knoten_unten[0])][int(knoten_unten[1])]
        vertex_matrix[int(knoten_mitte[0])][int(knoten_mitte[1])] =
        ↪  old_vertex_matrix[int(knoten_mitte[0])][int(knoten_mitte[1])]
        vertex_matrix[int(knoten_oben[0])][int(knoten_oben[1])] =
        ↪  old_vertex_matrix[int(knoten_oben[0])][int(knoten_oben[1])]
        # Verschiebung zwischen tiefstem und mittlerem Feld normal durchführen
        delta = impossible_height -
        ↪  (vertex_matrix[int(knoten_mitte[0])][int(knoten_mitte[1])] -
        ↪  vertex_matrix[int(knoten_unten[0])][int(knoten_unten[1])])
        delta = delta / 2
        vertex_matrix[int(knoten_unten[0])][int(knoten_unten[1])] -= delta
        vertex_matrix[int(knoten_mitte[0])][int(knoten_mitte[1])] += delta
        # für höchstes Feld wird noch benötigte Höhenänderung ...
        # ... mithilfe eines anderen Nachbarfelds erreicht
        knoten_to_take_from = auswahl(size, knoten_oben, bisher_obere_knoten,
        ↪  bisher_untere_knoten).split("|")
        delta1 = impossible_height -
        ↪  (vertex_matrix[int(knoten_oben[0])][int(knoten_oben[1])] -
        ↪  vertex_matrix[int(knoten_mitte[0])][int(knoten_mitte[1])])
        delta2 = delta1 / 2
        verschobene_erde -= delta2
        verschobene_erde += delta1
        vertex_matrix[int(knoten_oben[0])][int(knoten_oben[1])] += delta1
        vertex_matrix[int(knoten_to_take_from[0])][int(knoten_to_take_from[1])] -=
        ↪  delta1
```

Quellcode 1: Auszug aus dem Hauptprogramm (*wildschweine.py*)

```
import networkx as nx

# Breitensuche
def breadth_first_search(adj, firstlevel):
    seen = {}                   # Dictionary mit Entfernung vom Start
    level = 0                   # aktuelle Ebene
    nextlevel = firstlevel      # Knoten auf der nächsten Ebene
```

```
8
9       while nextlevel:
10          thislevel = nextlevel  # neue Ebene
11          nextlevel = {}         # Dictionary für nächste Ebene
12          for v in thislevel:
13              if v not in seen:
14                  seen[v] = level  # Entfernung für Knoten setzen
15                  nextlevel.update(adj[v])  # Nachbarn für nächste Ebene hinzufügen
16                  yield (v, level)
17          level += 1
18      del seen
19
20  # Residualnetzwerk erstellen
21  def build_residual_network(G):
22      # neues Netzwerk mit gleichen Knoten
23      R = nx.DiGraph()
24      R.add_nodes_from(G)
25
26      inf = float('inf')
27
28      # Kanten-Liste holen
29      edge_list = [(u, v, attr) for u, v, attr in G.edges(data=True)]
30
31      for u, v, attr in edge_list:
32          r = min(attr.get('capacity', inf), inf)
33          if not R.has_edge(u, v):
34              # Hinkante hat entsprechende Kapazität
35              R.add_edge(u, v, capacity=r)
36              # Rückkante mit Kapazität null
37              R.add_edge(v, u, capacity=0)
38          else:
39              # Kapazität für Rückkante doch definiert
40              # wird entsprechend gesetzt
41              R[u][v]['capacity'] = r
42
43      # Fluss auf allen Kanten null
44      for u in R:
45          for e in R[u].values():
46              e['flow'] = 0
47
48      return R
49
50
51  # Edmonds-Karp-Algorithmus
52  def edmonds_karp(G, s, t):
53      R = build_residual_network(G)
54      R_pred = R.pred
55      R_succ = R.succ
56
57      # Fluss entlang eines Pfades augmentieren
58      def augment(path):
59          flow = float('inf')
60          # minimale Residualkapazität entlang des Pfades bestimmen
61          it = iter(path)
62          u = next(it)
63          for v in it:
64              attr = R_succ[u][v]
65              flow = min(flow, attr['capacity'] - attr['flow'])
66              u = v
```

```
        # augmentieren
        it = iter(path)
        u = next(it)
        for v in it:
            # Fluss auf Hinkante erhöhen
            R_succ[u][v]['flow'] += flow
            # Fluss auf Rückkante verringern
            R_succ[v][u]['flow'] -= flow
            u = v
        return flow

    # bidirektionale Breitensuche nach augmentierendem Pfad
    def bidirectional_bfs():
        pred = {s: None}
        q_s = [s]
        succ = {t: None}
        q_t = [t]
        while True:
            q = []
            # von s aus suchen
            if len(q_s) <= len(q_t):
                for u in q_s:
                    for v, attr in R_succ[u].items():
                        # nur wenn nächster Knoten noch nicht im Pfad
                        # ... und entsprechende Kante noch nicht gesättigt
                        if v not in pred and attr['flow'] < attr['capacity']:
                            pred[v] = u
                            # Pfade treffen sich
                            if v in succ:
                                return v, pred, succ
                            q.append(v)
                # kein augmentierender Pfad mehr vorhanden
                if not q:
                    return None, None, None
                q_s = q
            # von t aus suchen -> gleiches Prinzip
            else:
                for u in q_t:
                    for v, attr in R_pred[u].items():
                        if v not in succ and attr['flow'] < attr['capacity']:
                            succ[v] = u
                            if v in pred:
                                return v, pred, succ
                            q.append(v)
                if not q:
                    return None, None, None
                q_t = q

    # maximalen Fluss bestimmen
    flow_value = 0
    while flow_value < float('inf'):
        v, pred, succ = bidirectional_bfs()
        # kein augmentierender Pfad mehr vorhanden -> max. Fluss
        if pred is None:
            break
        path = [v]
        # Pfad von s bis zum Treffpunkt
        u = v
        while u != s:
```

```
126             u = pred[u]
127             path.append(u)
128         path.reverse()
129         # weiterer Pfad bis zu t
130         u = v
131         while u != t:
132             u = succ[u]
133             path.append(u)
134         # Fluss augmentieren
135         flow_value += augment(path)
136 
137     R.graph['flow_value'] = flow_value
138     return R
139 
140 # minimaler Schnitt
141 def minimum_cut(flowgraph, s, t):
142 
143     R = edmonds_karp(flowgraph, s, t)
144     # gesättigte Kanten aus Residualnetzwerk entfernen
145     cutset = [(u, v, d) for u, v, d in R.edges(data=True)
146               if d['flow'] == d['capacity']]
147     R.remove_edges_from(cutset)
148 
149     # per Breitensuche von s aus ...
150     # ... werden von s erreichbare Knoten gesammelt
151     nextlevel = {s: 1}
152     non_reachable = set(dict(dict(breadth_first_search(R.adj, nextlevel))))
153     # nicht erreichbar sind alle anderen Knoten
154     # => in 2 Partitionen geteilt
155     partition = (set(flowgraph) - non_reachable, non_reachable)
156 
157     return R.graph['flow_value'], partition
```

Quellcode 2: Das Modul zur Berechnung des minimalen Schnitts (*boarflow.py*)

## 4.1 Quellcode der Erweiterung

```
1  import networkx as nx
2  import heapq as hq
3 
4 
5  def dijkstra_st(G, source, target):
6      G_succ = G._succ
7      paths = {source: [source]}  # Pfad source->target
8      dist = {}  # Distanzliste
9      seen = {}  # schon besuchte Knoten
10     queue = []  # Priorityqueue
11 
12     seen[source] = 0  # source hinzufügen
13     hq.heappush(queue, (0, source))
14     while queue:
15         (d, v) = hq.heappop(queue)
16         if v in dist:
17             continue  # schon besucht
18         dist[v] = d
19         if v == target:  # gefunden
20             break
21         for u, e in G_succ[v].items():
```

```
            cost = G.edges[v, u]["weight"]
            vu_dist = dist[v] + cost  # Distanz berechnen
            if u not in dist and (u not in seen or vu_dist < seen[u]):  # Distanz und
            ↪ Pfad setzen, wenn notwendig
                seen[u] = vu_dist
                hq.heappush(queue, (vu_dist, u))
                paths[u] = paths[v] + [u]

    return dist[target], paths[target]


def shortest_path(matrix):
    size = len(matrix)
    graph = nx.DiGraph()
    graph.add_node("s")
    graph.add_node("t")
    for i in range(0, size):
        for k in range(0, size):
            graph.add_node(str(i) + "|" + str(k))

    for i in range(0, size):
        # Kanten von Quelle in erste Reihe
        graph.add_edge("s", "0|" + str(i), weight=0.0)
        # Kanten von letzter Reihe zu Senke
        graph.add_edge(str(size - 1) + "|" + str(i), "t", weight=0.0)
        # jeweils Kante zu Knoten direkt rechts und unten
        for k in range(0, size):
            if k + 1 < size:
                weight = abs(matrix[i][k + 1] - matrix[i][k])
                graph.add_edge(str(i) + "|" + str(k), str(i) + "|" + str(k + 1),
                ↪ weight=weight)
                graph.add_edge(str(i) + "|" + str(k + 1), str(i) + "|" + str(k),
                ↪ weight=weight)
            if i + 1 < size:
                weight = abs(matrix[i + 1][k] - matrix[i][k])
                graph.add_edge(str(i) + "|" + str(k), str(i + 1) + "|" + str(k),
                ↪ weight=weight)
                graph.add_edge(str(i + 1) + "|" + str(k), str(i) + "|" + str(k),
                ↪ weight=weight)

    dijkstr = dijkstra_st(graph, "s", "t")
    path = dijkstr[1]
    length = dijkstr[0]
    max_steigung = 0.0
    for i in range(0, len(path)-1):
        v1 = path[i]
        v2 = path[i+1]
        data = graph.get_edge_data(v1, v2)
        w = data["weight"]
        if w > max_steigung:
            max_steigung = w

    print("Zu überwindende Höhe: " + str(round(length,3)))
    print("Über den Pfad: " + str(path))
    print("Größte Einzelsteigung: " + str(round(max_steigung,3)))
```

Quellcode 3: Die Erweiterung (*dijkstra.py*)