Bundeswettbewerb Informatik

36. Bundeswettbewerb Informatik, 2. Runde

Lösungshinweise und Bewertungskriterien

Allgemeines

Es ist immer wieder bewundernswert, wie viele Ideen, wie viel Wissen, Fleiß und Durchhaltevermögen in den Einsendungen zur zweiten Runde eines Bundeswettbewerbs Informatik stecken. Um aber die Allerbesten für die Endrunde zu bestimmen, müssen wir die Arbeiten kritisch begutachten und hohe Anforderungen stellen. Von daher sind Punktabzüge die Regel und Bewertungen über die Erwartungen hinaus die Ausnahme. Jedoch sind spannende Erweiterungen der Aufgabenstellung durchaus einige Extrapunkte wert.

Wie auch immer Ihre Einsendung bewertet wurde, lassen Sie sich nicht entmutigen! Allein durch die Arbeit an den Aufgaben und ihren Lösungen hat jede Teilnehmerin und jeder Teilnehmer einiges gelernt; diesen Effekt sollten Sie nicht unterschätzen. Selbst wenn Sie zum Beispiel aus Zeitmangel nur die Lösung zu einer Aufgabe einreichten, so erhielten Sie eine Bewertung Ihrer Einsendung, die Ihnen bei der Anfertigung künftiger Lösungen hilfreich sein kann.

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

Bewertungsbogen

Aus der ersten Runde oder auch aus früheren Wettbewerbsteilnahmen kennen Sie den Bewertungsbogen, der angibt, wie Ihre Einsendung die einzelnen Bewertungskriterien erfüllt hat. Auch in dieser Runde können Sie den Bewertungsbogen im Anmeldesystem PMS einsehen. In der ersten Runde ging die Bewertung noch von 5 Punkten aus, von denen bei Mängeln dann abgezogen werden konnte. In der zweiten Runde geht die Bewertung von 20 Punkten aus; dafür gibt es deutlich mehr Bewertungskriterien, bei denen Punkte abgezogen oder auch hinzuaddiert werden konnten.

Terminlage der zweiten Runde

Für Abiturienten ist der Terminkonflikt zwischen Abiturvorbereitung und zweiten Runde sicher nicht ideal. Doch leider bleibt uns nur die erste Jahreshälfte für die zweite BwInf-Runde: In der zweiten Jahreshälfte läuft nämlich die zweite Runde des Mathematikwettbewerbs, dem wir keine Konkurrenz machen wollen. Aber: die Bearbeitungszeit für die zweite BwInf-Runde beträgt etwa vier Monate. Frühzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, zeitliche Engpässe am Ende der Bearbeitungszeit gerade mit der wichtigen, ausführlichen Dokumentation der Aufgabenlösungen zu vermeiden.

Einige Aufgaben in der zweiten Runde sind oft deutlich schwerer zu lösen, als sie auf den ersten Blick erscheinen mögen. Erst in der konkreten Umsetzung einer Lösungsidee stößt man manchmal auf weitere Besonderheiten bzw. noch zu lösende Schwierigkeiten, was dann zusätzlicher Zeit bedarf. Daher ist es sinnvoll, beide einzureichende Aufgaben nicht nacheinander, sondern relativ gleichzeitig zu bearbeiten, um nicht vom zeitlichen Aufwand der jeweiligen Aufgabe unangenehm kurz vor Ablauf der Bearbeitungszeit überrascht zu werden und keine vollständige Lösung mehr zu schaffen.

Dokumentation

Es ist sehr gut nachvollziehbar, dass Sie Ihre Energie bevorzugt in die Lösung der Aufgaben, die Entwicklung Ihrer Ideen und deren Umsetzung in Software fließen lassen. Doch ohne eine verständliche Beschreibung der Lösungsideen, eine übersichtliche Dokumentation der wichtigsten Komponenten Ihrer Programme, eine geeignete Kommentierung der Quellcodes und eine ausreichende Zahl sinnvoller Beispiele (die die verschiedenen, bei der Lösung des Problems zu berücksichtigenden Fälle abdecken) ist eine Einsendung nur wenig wert.

Bewerterinnen und Bewerter können die Qualität Ihrer Aufgabenlösungen nur anhand dieser Informationen vernünftig einschätzen. Mängel in der Dokumentation der Einsendung können nur selten durch das konkrete Überprüfen der Ergebnisse der Programme durch die Bewerterin oder den Bewerter etwas ausgeglichen werden – wenn diese denn überhaupt ausgeführt werden können: Hier gibt es gelegentlich Probleme, die meist vermieden werden könnten, wenn Lösungsprogramme vor der Einsendung nicht nur auf dem eigenen, sondern auch einmal auf einem fremden Rechner hinsichtlich Lauffähigkeit getestet würden. Insgesamt sollte die Erstellung der Dokumentation die Programmierarbeit begleiten oder ihr teilweise sogar vorangehen: Wer nicht in der Lage ist, Idee und Modell präzise zu formulieren, bekommt auch keine saubere Umsetzung hin, in welcher Programmiersprache auch immer.

Bewertungskriterien

Bei den im Folgenden beschriebenen Lösungsideen handelt es sich um Vorschläge, aber sicher nicht um die einzigen Lösungswege, die wir gelten ließen. Wir akzeptieren in der Regel alle Ansätze, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind. Einige Dinge gibt es allerdings, die – unabhängig vom gewählten Lösungsweg – auf jeden Fall in der Dokumentation erwartet wurden. Zu jeder Aufgabe wird deshalb in einem eigenen Abschnitt erläutert, worauf bei der Bewertung dieser Aufgabe besonders geachtet wurde. Außerdem gibt es aufgabenunabhängig einige Anforderungen an die Dokumentation (klare

Beschreibung der Lösungsidee, genügend aussagekräftige Beispiele und wesentliche Auszüge aus dem Quellcode) einschließlich einer theoretischen Analyse (geeignete Laufzeitüberlegungen und sinnvolle Begründungen) sowie an den Quellcode der implementierten Software (mit übersichtlicher Programmstruktur und verständlicher Kommentierung) und an das lauffähige Programm (keine Implementierungsfehler).

Danksagung

Alle Aufgaben wurden ursprünglich vom Aufgabenausschuss des Bundeswettbewerbs Informatik entwickelt. An der Erstellung der im folgenden skizzierten möglichen Lösungsideen wirkten neben dem Aufgabenausschuss vor allem folgende Einzelpersonen mit: Niccolò Rigi-Luperti und Nikolai Wyderka (Aufgabe 1), Kamal Abdellatif, Thekla Hamm und Robert Vicari (Aufgabe 2) sowie Mario Albrecht und Dominik Meier (Aufgabe 3). Allen Beteiligten sei für Ihre großartige Mitarbeit hiermit ganz herzlich gedankt.

Aufgabe 1: Die Kunst der Fuge

1.1 Grundlegende Überlegungen

Jede Reihe von Ilonas Mauer soll aus den Bausteinen der Längen 1 bis n bestehen. Die Reihen der Mauer sollen derart sein, dass keine zwei Fugen (Zwischenräume zwischen den Steinen) übereinander liegen, auch nicht, wenn mehrere Reihen dazwischen liegen.

Für gegebenes n soll nach diesen Regeln eine möglichst hohe Mauer erzeugt werden.

Zunächst stellt sich die Frage, wie hoch eine Mauer maximal werden kann. Jede Reihe der Mauer besteht aus n Bausteinen und hat eine Gesamtlänge von $L = 1+2+\ldots+n = \frac{n(n+1)}{2}$. Fugen können im Abstand von 1 Längeneinheit auftauchen, insgesamt also an $L-1$ Stellen (genannt Spalten). Jede Reihe wiederum belegt $n-1$ dieser möglichen Fugenplätze.

Im besten Fall können wir eine Mauer konstruieren, bei der für jede Spalte genau eine Reihe existiert, die dort eine Fuge besitzt. Bezeichnen wir die maximale Reihenanzahl mit h, so ergibt sich also

$$h = \frac{L-1}{n-1} \tag{1}$$
$$= \frac{n(n+1)-2}{2(n-1)} \tag{2}$$
$$= \frac{(n-1)(n+2)}{2(n-1)} \tag{3}$$
$$= \frac{n+2}{2} = \frac{n}{2}+1. \tag{4}$$

Für $n = 4$ ergibt sich eine maximale Höhe von $h = 3$, und tatsächlich kann man sich in dem Beispiel des Aufgabenblatts leicht davon überzeugen, dass in der angegebenen Lösung in jeder Spalte auch tatsächlich nur eine Fuge vorkommt.

Ist n ungerade, ist die Maximalhöhe keine ganze Zahl und muss abgerundet werden. In diesem Fall hat eine maximale Lösung allerdings nicht mehr die Eigenschaft, dass jede Spalte eine Fuge besitzt (siehe Abb. 1). Außerdem kann man für gerade n eine Lösung für $n+1$ konstruieren, indem man an der zweiten oder vorletzten Stelle jeder Reihe jeweils einen Stein der Länge $n+1$ einschiebt.

Die Maximalhöhe gibt nur eine theoretische obere Grenze für die mögliche Höhe einer Mauer an. Es ist jedoch nicht garantiert, dass auch tatsächlich eine solche Mauer der maximalen Höhe h für alle n existiert („Maximale Mauer“). Man kann leicht per Hand nachprüfen, dass sie für $n = 1,2,3,4$ existiert. Für größere n wollen wir nun ein Programm schreiben, das versucht, eine solche Mauer algorithmisch aufzubauen. Dabei beschränken wir uns zunächst auf gerade n und nutzen die obere Beobachtung aus.

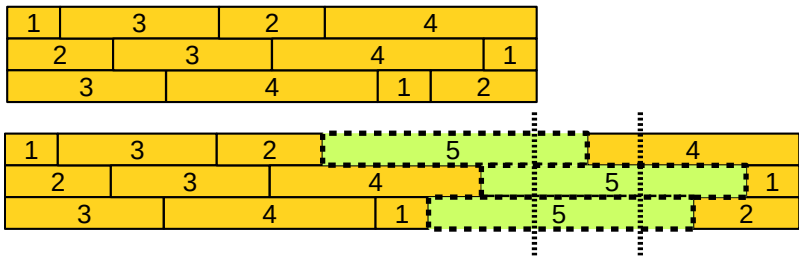


Abbildung 1: Einschieben des Steins der Länge $n+1$ in alle Mauerreihen, wobei hier zwei Spalten fugenfrei bleiben: von geradem $n = 4$ auf ungerades $n = 5$.

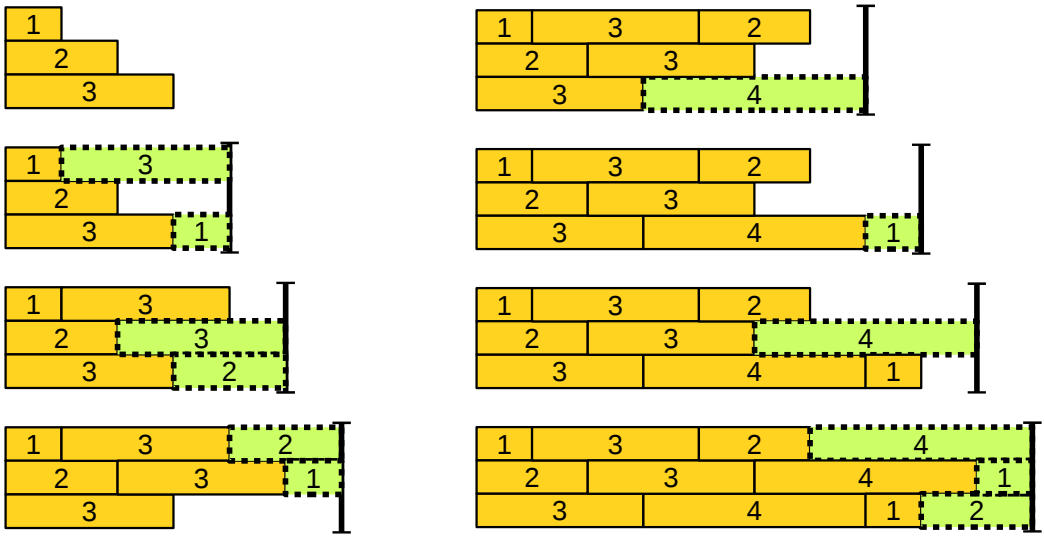


Abbildung 2: Eine Mauer wird für $n = 4$ spaltenweise aufgebaut. Dazu wird an jeder neuen Cursorposition der längste verfügbare Block angelegt. Dieses Beispiel zeigt einen Idealfall, bei dem die Rekursion noch nicht benötigt wird.

1.2 Aufbau der Mauer

Es gibt viele verschiedene Varianten, um eine valide Mauer zu konstruieren. Hier beschreiben wir nur eine mögliche Art und optimieren diese. Das heißt nicht, dass es nicht andere, bessere Verfahren gibt.

Die grundlegende Idee unseres Programms ist, dass wir die Mauer sukzessive spaltenweise von links nach rechts aufbauen. Wir beschränken uns im Folgenden auf den Fall, dass n gerade ist, damit in jeder Spalte genau eine Fuge erzeugt werden muss. Das macht unseren Algorithmus etwas übersichtlicher und soll uns hier genügen, um das Konzept zu erklären. Es sei noch angemerkt, dass die Aufgabenstellung mit dem NP-vollständigen *Exact Cover Problem*[1] verwandt ist.

Wir beginnen also mit $h = \frac{n}{2} + 1$ leeren Reihen. Bevor der eigentliche Algorithmus startet, setzen wir in jede Reihe schon einen ersten Block, aufsteigend von 1 bis h, sodass ein Treppenmuster entsteht (siehe Abb. 2). Das hat sich heuristisch als guter Anfangszustand erwiesen.

[1] Wikipedia, 2018. http://en.wikipedia.org/wiki/Exact_cover

Die Reihenfolge der Reihen ist übrigens nicht wichtig; jede valide Mauer bleibt auch dann valide, wenn man die Reihen umsortiert. Daher können wir ohne Weiteres davon ausgehen, dass die ersten Blöcke der Reihen aufsteigend sortiert sind.

Zentraler Fokus liegt nun immer auf der Spalte, die am weitesten links liegt und noch keine Fuge besitzt. Diese Spalte nennen wir im folgenden „Cursor“ (manchmal auch „Sweep Line“ genannt). Unser Ziel ist es, immer so einen nächsten Block nahtlos an eine Reihe anzulegen, dass am Cursor eine neue Fuge entsteht. Wenn dann unser Cursor z. B. in der 30. Spalte ist, wissen wir, dass in allen vorherigen 29 Spalten schon eine Fuge vorhanden ist. Erreicht der Cursor schließlich das Ende der Mauer, also die L-te Spalte, haben wir eine valide Mauer erzeugt.

Die Schwäche dieses Verfahrens ist, dass es oft vorkommt, dass an einer neuen Cursorposition plötzlich kein einziger Block mehr zum Anlegen verfügbar ist. In diesem Fall ist das Programm in eine Sackgasse geraten. Es wird dann der letzte gesetzte Block zurückgenommen und der nächste verfügbare Block ausprobiert. Scheitert auch der letzte verfügbare Block, wird der Cursor wieder eine Spalte nach links verschoben und dort wiederum die nächste verfügbare Länge ausprobiert, und so weiter. Die Herausforderung besteht also darin, möglichst denjenigen Block zu finden und zu setzen, der mit der kleinsten Wahrscheinlichkeit später in eine Sackgasse führt. In Abschnitt 1.3 stellen wir eine Idee vor, wie man dieses Problem lösen kann.

In der Umsetzung nutzen wir Rekursion, da dies ein angenehmer Weg ist, um die früheren Cursorzustände automatisch zwischenzuspeichern. Unser Verfahren ist vom Konzept her eine Tiefensuche mit Rücksetzverfahren („Depth-First-Search with Backtracking“). Je Mauerzustand (Knoten) wird entschieden, welcher nächster Block angelegt werden soll oder ob eine Sackgasse oder eine Lösung erreicht wurde. Der folgende Pseudocode fasst den Lösungsansatz zusammen:

```
1  public void expand(cursorpos) {
2    if(cursorpos==L) {
3      loesungAusgeben();
4      return; //Methode verlassen
5    }
6    else {
7      laengen = listeVerfuegbareLaengen(cursorpos);
8      for int i in laengen {
9        anlegen(i);
10       expand(cursorpos+1);
11       zuruecknehmen(i);
12     }
13   }
14 }
```

Mit diesem Ansatz allein kann man maximale Mauern bis zur Höhe $n = 14$ in annehmbarer Zeit erzeugen (siehe Abb. 3 und 4).

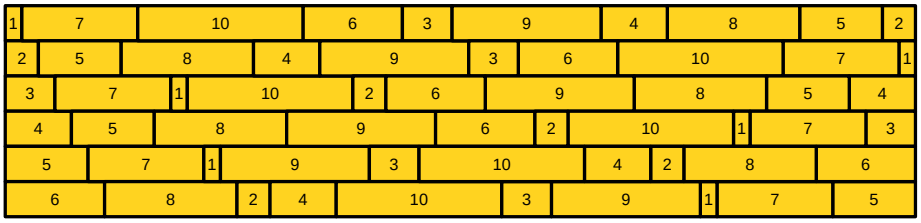


Abbildung 3: Maximal hohe Mauer für $n = 10$.

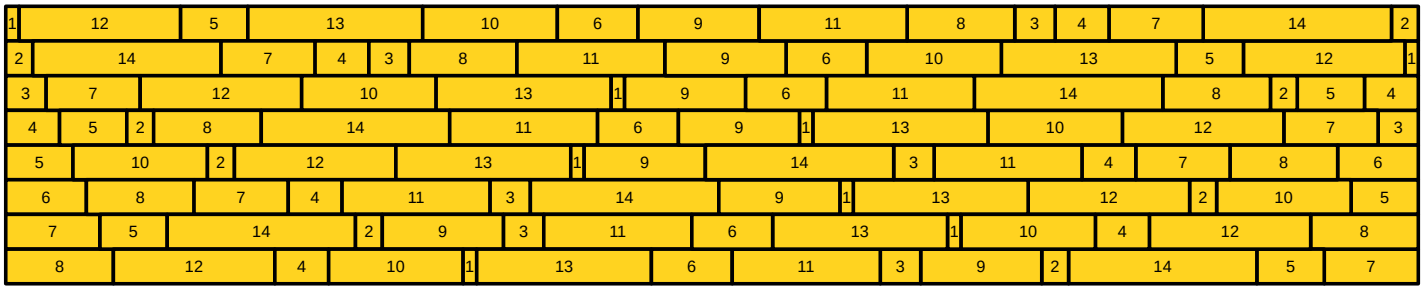


Abbildung 4: Maximal hohe Mauer für $n = 14$.

1.3 Erste Optimierung: Permutationen

Wie bereits erwähnt, hängt die Effizienz unseres Programms stark davon ab, dass wir je Spalte einen möglichst „passenden“ Block auswählen. Im Pseudocode ist dafür Zeile 7 verantwortlich: Dort legt die Funktion *listeVerfuegbareLaengen()* fest, in welcher Reihenfolge die Längen später in der Schleife ausprobiert werden sollen. In einer naiven Implementierung würde man schlicht eine absteigende oder aufsteigende Reihenfolge wählen, sodass die verfügbaren Längen von groß nach klein oder klein nach groß durchprobiert werden. Das ist aber nicht besonders effizient. Tatsächlich ist es sehr viel besser, zufällige Reihenfolgen zu testen („randomized algorithm“). Mithilfe dieser Optimierung kann unser Programm Mauern bis zu einer Höhe von $n = 22$ erzeugen.

Die neuen Reihenfolgen, die Permutationen der Zahlen von 1 bis n sind, stellen wir als Integer-Array dar, mit den einzelnen Blocklängen als Einträgen. Die Blöcke werden dann in dieser vorgegebenen Reihenfolge durchprobiert. Wir nennen diese Reihenfolgen im folgenden Permutationen. Ist beispielsweise $n = 10$, so führen folgende, zu Beginn festgelegte Permutationen zu deutlich unterschiedlichen Laufzeiten. Diese messen wir, indem wir die Anzahl der rekursiven Aufrufe (*Expand*-Aufrufe) mitzählen:[2]

Permutation	rekursive Aufrufe
$[6,7,2,10,3,9,1,4,5,8]$	229
$[5,6,3,2,8,7,10,1,4,9]$	40
$[8,10,5,3,9,7,1,2,6,4]$	26
$[10,9,8,7,6,5,4,3,2,1]$	103

[2]Die hier gezeigten Aufrufzahlen wurden in einer späteren Implementierung gemessen, als noch weitere, unten eingeführte Optimierungen möglich waren. Der relevante Unterschied, dass manche Permutationen um mehrere Größenordnungen schneller fertig sind als andere, bleibt aber weiterhin erhalten.

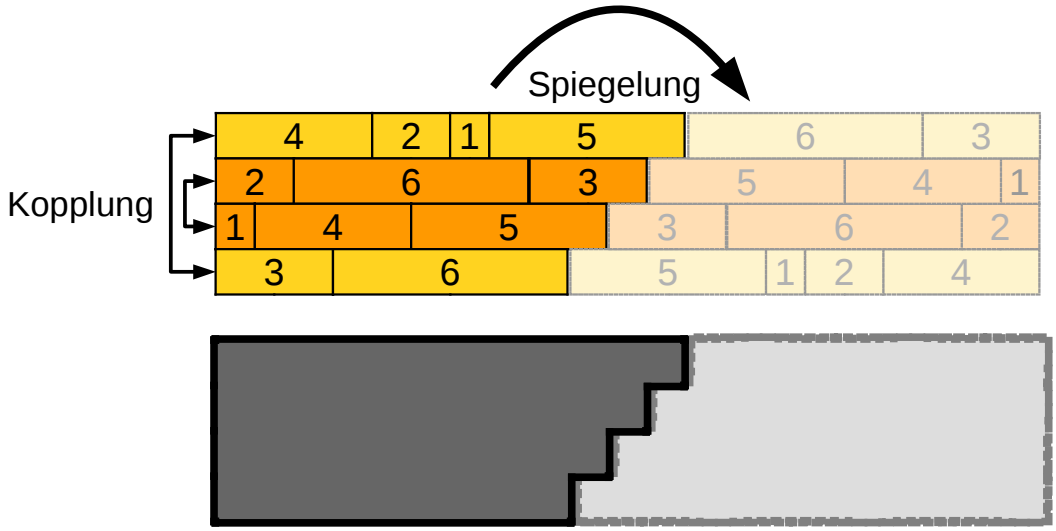


Abbildung 5: *oben:* Beispiel einer $n = 6$-Spiegelung. Jede blass gedruckte Reihe ist eine gespiegelte Version der zugehörigen gekoppelten Reihe.
unten: Die Grundidee der Spiegelung: Der dunkle (berechnete) Block wird kopiert, gespiegelt rechts an die Mauer gesetzt. Das funktioniert mit beliebig vielen Reihen, solange in der Mitte der Mauer weiterhin das Treppenmuster erzeugt wird.

Die Qualität der Permutationen ist sehr unterschiedlich. Bei $n = 10$ liegt der Unterschied in der Laufzeit bei etwa einer Größenordnung. Für größere n wird der Unterschied noch größer; für $n = 90$ benötigt eine durchschnittliche Permutation mehr als 10.000.000 rekursive Aufrufe, die beste gefundene Permutation jedoch nur 5.194.

Trotz intensiver Bemühungen ist leider nicht zu erkennen, was eine gute Permutation ausmacht. Daher erzeugen wir viele unterschiedliche zufällige Permutationen und lassen diese bis zu einer Obergrenze an rekursiven Aufrufen laufen. Dadurch besteht zwar die Gefahr, dass wir einige Permutationen abbrechen, kurz bevor sie eine fertige Mauer geliefert hätten, aber es steigt die Chance, unter den vielen Permutationen eine der seltenen sehr guten zu finden, die uns die Mauer sehr schnell erzeugen. Das Hauptaugenmerk dieser Optimierung liegt darin, wie man die Obergrenze an Aufrufen festlegt. Weiter unten in Abschnitt 1.9 werden wir $\frac{1}{12}n^3$ als brauchbare Obergrenze finden.

1.4 Zweite Optimierung: Kopplung und Spiegelung der halben Mauer

Unter den richtigen Voraussetzungen kann man die zweite Hälfte der Mauer allein durch Spiegelung der ersten Hälfte erhalten. Damit spart man sich das Berechnen der gesamten zweiten Mauerhälfte, womit man im Vergleich zu vorher ungefähr doppelt so große Mauern finden kann. Durch diese Optimierung können wir Mauern statt bis $n = 22$ bis zu einer Höhe von $n = 38$ erzeugen.

Nötig dafür ist, dass es eine gerade Anzahl an Reihen gibt. Dann können nämlich jeweils zwei Reihen miteinander gekoppelt werden. Das heißt, dass sie beide aus einem gemeinsamen Pool

an Blöcken der Längen $1,\ldots,n$ gebaut werden, und dass der letzte Block jeder Reihe so lang sein muss, dass er die letzte Fuge der kürzesten Reihe überdeckt. In dieser Situation kann dann jede Reihe gerade durch die Blöcke aus der Partnerreihe vervollständigt werden, und man spart sich das Berechnen der gesamten zweiten Mauerhälfte. Die Spiegelung geschieht an der Mitte der Mauer, deshalb liegen die neuen Blöcke in umgekehrter Reihenfolge vor.

Fugenkonflikte gibt es beim Einsetzten der gespiegelten Kopie keine, denn durch das Konstruktionsverfahren haben wir die erste Mauerhälfte so erzeugt, dass es dort keine Fugenkonflikte gibt (diese erste Hälfte zu berechnen ist weiterhin der aufwendige Teil). Eine Kopie dieser ersten Hälfte hat somit auch keine inneren Konflikte, woran auch eine zusätzliche Spiegelung nichts ändert. Nur ganz in der Mitte passiert es, dass die überhängenden Reihen unter sich selbst kopiert werden (siehe Abb. 5). Doch genau für diesen Fall haben wir die zusätzliche Bedingung, dass der überhängende Bereich immer nur durch einen einzigen (ausreichend langen) Block gefüllt sein muss. Dadurch sind in diesem mittleren Bereich beim Kopieren nur die schon bestehenden Fugen der Partnerreihe relevant.

Ein Nachteil der Spiegelmethode ist, dass diese (zusätzlich zur bereits erwähnten Bedingung von geradem n) nur für Mauerhöhen n funktioniert, die keine Vielfachen von 4 sind, also für Zahlen der Form $n = 4k+2$ mit ganzzahligem k. Der Grund ist, dass die Anzahl von Reihen $h = \frac{n}{2}+1$ für die Kopplung gerade sein muss.

1.5 Dritte Optimierung: Einschränken des Suchraums

Unser Algorithmus hat momentan noch die Schwäche, dass er naiv alle verfügbaren Längen nacheinander durchprobiert. Das wollen wir jetzt ändern, denn es gibt viele Situationen, bei denen schon vorher erkennbar ist, dass sie unmöglich zu einer validen Mauer führen können. Von diesen Zuständen aus weiter zu expandieren ist Zeitverschwendung. Also wollen wir sie so früh wie möglich erkennen, um sie dann zu überspringen („Branch & Bound“).

Nichtvervollständigbare Mauern sind diejenigen, bei denen sich eine Reihe zu weit vom Cursor entfernt hat, und dadurch kein weiterer Block mehr an sie angelegt werden kann, der sie bis zum aktuellen Cursor auffüllt. Wir demonstrieren das an drei Beispielen (in den folgenden Darstellungen repräsentieren Sterne und Punkte abwechselnd verschiedene Mauerblöcke):

```
n=10
     *...........****   |
     **.....***........ |
     ***.........       |
     ****.....********  |
>    ******...          | dist=11
     ******.......*....|
```

Hier ist die fünfte Reihe der Grund, dass es keine Lösung gibt. Die fünfte Reihe ist bereits elf Einheiten ($dist = 11$) vom Cursor entfernt. Diese Lücke wird nicht mehr geschlossen werden können, da es nur Längen bis $n = 10$ gibt. Also bräuchte man mindestens zwei Blöcke zum Schließen der Lücke, was aber einen Fugenkonflikt erzeugen würde.

```
n=10
     *.........********** |
>    **.....******        | dist=9
```

```
    ***.....*..******     |
    ****.......*********|
    *****.........*....   |
    ******..........**    |
```

Die zweite Reihe ist hier der Indikator. Sie hat zwar nun einen Abstand 9 zum Cursor, aber trotzdem ist die Mauer nicht mehr vervollständigbar. Grund dafür ist die Kopplung der ersten und zweiten Reihe: In der ersten Reihe sind die Längen 10 und 9 schon verbaut. Beide Reihen (insbesondere die zweite) können also maximal nur noch die Länge 8 anlegen.

```
n=10
    *......*********.......  |
    **........*****...****   |
    ***......**.........     |
    ****.......*******.....*|
>   *****........*.......    | dist=5
    *******..*********       | dist=9
```

Eine dritte nicht mehr zum Ziel führende Mauer. Dies kann an der fünften Reihe erkannt werden. Diese ist mit der sechsten Reihe gekoppelt. Zusammen bestehen die beiden Reihen schon aus sieben Blöcken, es sind nur noch die Längen 3, 4 und 10 übrig. Wir sehen aber, dass die Länge 10 der einzige Block ist, der in der 6. Reihe die Lücke von 9 noch füllen kann. Dieser Block steht also für die fünfte Reihe nicht zur Verfügung, es bleiben für diese nur die Längen 3 und 4 übrig. Die sind aber zu kurz, um die dortige Lücke von 5 zu füllen.

Mit dieser zusätzlichen Optimierung können wir maximale Mauern der Höhe $n = 102$ konstruieren.

1.6 Permutationen ergänzen

Bisher haben wir Permutationen komplett zufällig erzeugt. Möchten wir aber höhere Mauern als $n = 102$ erzeugen, können wir einen Trick benutzen. Die Idee: Wir nehmen unsere beste gefundene Permutation für $n = 102$ und ergänzen sie an vier zufälligen Positionen um die Zahlen $103, 104, 105$ und 106. Das ergibt eine neue Permutation für $n = 106$. Nutzt man für den Algorithmus nun allein solche Permutationen, findet man tatsächlich Lösungen für $n = 106$ in annehmbarer Zeit.

Dieses Verfahren kann man dann wiederholen und gelangt zu Lösungen für $n = 110$. Nur für $n = 114$ funktioniert dieser Trick leider nicht mehr. Nach 15 Minuten Laufzeit haben wir die Suche abgebrochen.

Gute Permutationen haben also eine gewisse Struktur, die auch noch erhalten bleibt, wenn man sie durch neue Zahlen ergänzt.

1.7 Übersicht über die Optimierungen und Laufzeitanalyse

Abschließend noch eine Auflistung, inwieweit die einzelnen Optimierungen zur erreichbaren Mauerhöhe n beitragen:

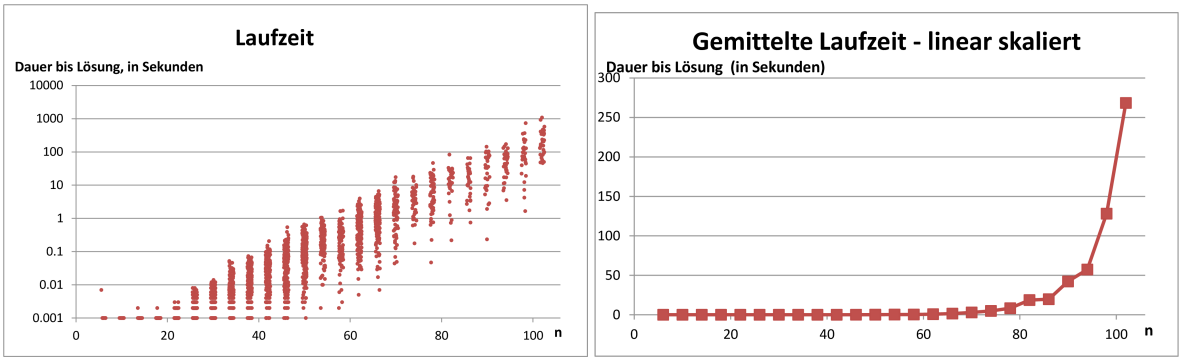


Abbildung 6: *links:* Laufzeiten für den Average Case, logarithmisch skaliert.
rechts: Gemittelte Laufzeiten für verschiedene n (linear skaliert).

Optimierung	n
Tiefensuche	14
mit Permutationen	22
und mit Spiegelung	38
und mit Suchraumbeschränkung	102
und mit Permutationsergänzung	110

Schließlich wollen wir noch die Laufzeit unseres Programmes abschätzen.

Im schlimmsten Fall („Worst Case“) hat unser Algorithmus eine überexponentielle Laufzeit. Eine grobe obere Grenze an möglichen Aufbauten der Mauer ist durch $(n!)^h = (n!)^{(\frac{n}{2}+1)}$ gegeben, was der Anzahl aller möglichen Anordnungen der Mauerblöcke entspricht, aber auch inklusive aller verbotenen Zustände, bei denen Fugen übereinander liegen. Die tatsächliche obere Grenze liegt also niedriger, aber vermutlich nicht wesentlich.

Für den durchschnittlichen Fall („Average Case“) messen wir die Laufzeiten für alle verfügbaren n mehrmals und tragen die Ergebnisse in Abb. 6 logarithmisch gegen n auf. Zur besseren Ansicht wird je Messwert noch eine kleine künstliche Streuung um n eingefügt, damit sich die Punkte nicht so sehr überlappen. Man sieht, dass die eingetragenen Punkte eine gerade aufsteigende Linie bilden, was exponentiellem Wachstum entspricht (da die y-Achse logarithmisch skaliert ist). Im „Average Case“ hat unser Algorithmus also eine exponentielle Laufzeit.

Zuletzt wollen wir noch eine Funktion $f(n)$ bestimmen, die die Laufzeit des Programms in Sekunden vorhersagt. Mit dem Ansatz $f(n) = a \cdot 2^{bn}$ (a, b freie Parameter) erhalten wir nach Einsetzen unserer Messwerte $a = 0,0001\,\mathrm{s}$ und $b = 0,2$. Die Laufzeit unseres Programms ist somit näherungsweise gegeben durch

$$f(n) = 0,0001\,\mathrm{s} \cdot 2^{0,2 \cdot n}.$$

$n = 62$ ist die größte Mauer, die unser Programm zuverlässig in unter einer Sekunde schafft. $n = 80$ ist die größte Mauer, die noch in unter zehn Sekunden gefunden wird. Ab $n = 80$ zieht dann aber die Laufzeit stark an, die folgenden höheren Mauern werden nicht mehr in annehmbarer Zeit gefunden. Diesen plötzlichen Anstieg bei ca. $n = 80$ kann man in Abb. 6 gut erkennen.

1.8 Beispiele

Im Folgenden zeigen wir ausgewählte Beispiele für gültige maximale Mauern für verschiedene n. Die Zahlen stehen für die Längen der einzelnen Bausteine.

$n = 10$

01 07 10 06 03 09 04 08 05 02 __sum=55
02 05 08 04 09 03 06 10 07 01 __sum=55
03 07 01 10 02 06 09 08 05 04 __sum=55
04 05 08 09 06 02 10 01 07 03 __sum=55
05 07 01 09 03 10 04 02 08 06 __sum=55
06 08 02 04 10 03 09 01 07 05 __sum=55

29 *Expand*-Aufrufe

Permutation: 7 5 1 8 6 10 9 4 2 3

$n = 14$

01 12 05 13 10 06 09 11 08 03 04 07 14 02 __sum=105
02 14 07 04 03 08 11 09 06 10 13 05 12 01 __sum=105
03 07 12 10 13 01 09 06 11 14 08 02 05 04 __sum=105
04 05 02 08 14 11 06 09 01 13 10 12 07 03 __sum=105
05 10 02 12 13 01 09 14 03 11 04 07 08 06 __sum=105
06 08 07 04 11 03 14 09 01 13 12 02 10 05 __sum=105
07 05 14 02 09 03 11 06 13 01 10 04 12 08 __sum=105
08 12 04 10 01 13 06 11 03 09 02 14 05 07 __sum=105

203 *Expand*-Aufrufe

Permutation: 14 5 12 7 8 2 10 4 9 13 3 11 6 1

$n = 22$

01 21 05 11 20 07 17 12 18 03 16 19 10 13 08 04 14 06 15 09 22 02 __sum=253
02 22 09 15 06 14 04 08 13 10 19 16 03 18 12 17 07 20 11 05 21 01 __sum=253
03 20 02 05 06 11 15 17 12 13 10 18 19 07 21 01 16 08 14 09 22 04 __sum=253
04 22 09 14 08 16 01 21 07 19 18 10 13 12 17 15 11 06 05 02 20 03 __sum=253
05 14 21 16 19 17 07 18 13 03 10 12 08 04 15 20 01 22 11 02 09 06 __sum=253
06 09 02 11 22 01 20 15 04 08 12 10 03 13 18 07 17 19 16 21 14 05 __sum=253
07 22 02 06 09 15 17 19 10 18 12 03 13 04 20 16 01 14 11 21 05 08 __sum=253
08 05 21 11 14 01 16 20 04 13 03 12 18 10 19 17 15 09 06 02 22 07 __sum=253
09 05 06 21 01 02 11 12 16 04 19 03 18 08 13 17 07 15 14 20 22 10 __sum=253
10 22 20 14 15 07 17 13 08 18 03 19 04 16 12 11 02 01 21 06 05 09 __sum=253
11 05 02 21 14 16 08 07 17 18 10 13 03 19 04 15 06 01 20 22 09 12 __sum=253
12 09 22 20 01 06 15 04 19 03 13 10 18 17 07 08 16 14 21 02 05 11 __sum=253

1421 *Expand*-Aufrufe

Permutation: 5 9 2 11 14 21 6 15 16 1 17 8 7 4 20 22 12 10 18 13 3 19

$n = 38$

01 26 07 09 30 08 14 06 22 18 16 03 23 15 11 32 34 04 13 33 17 12 20 31 28 05 10 24 25 19 29 27 21 35 36 37 38 02 __sum=741
02 38 37 36 35 21 27 29 19 25 24 10 05 28 31 20 12 17 33 13 04 34 32 11 15 23 03 16 18 22 06 14 08 30 09 07 26 01 __sum=741
03 26 07 09 08 36 02 06 35 18 23 15 27 25 33 01 13 10 24 05 31 12 17 20 28 30 32 19 29 11 21 22 34 16 14 37 38 04 __sum=741
04 38 37 14 16 34 22 21 11 29 19 32 30 28 20 17 12 31 05 24 10 13 01 33 25 27 15 23 18 35 06 02 36 08 09 07 26 03 __sum=741
05 26 07 09 37 08 22 16 03 21 23 15 32 19 25 13 01 24 17 28 20 12 31 10 33 34 11 04 35 29 27 18 14 30 02 36 38 06 __sum=741
06 38 36 02 30 14 18 27 29 35 04 11 34 33 10 31 12 20 28 17 24 01 13 25 19 32 15 23 21 03 16 22 08 37 09 07 26 05 __sum=741
07 26 30 09 36 02 06 35 15 21 03 32 11 25 04 24 34 10 20 28 17 31 05 12 13 33 01 19 29 23 27 18 16 14 22 37 38 08 __sum=741
08 38 37 22 14 16 18 27 23 29 19 01 33 13 12 05 31 17 28 20 10 34 24 04 25 11 32 03 21 15 35 06 02 36 09 30 26 07 __sum=741
09 26 06 08 37 14 22 18 27 15 29 01 11 36 04 13 33 28 31 20 12 17 05 25 24 34 19 32 23 03 21 35 16 02 30 07 38 10 __sum=741
10 38 07 30 02 16 35 21 03 23 32 19 34 24 25 05 17 12 20 31 28 33 13 04 36 11 01 29 15 27 18 22 14 37 08 06 26 09 __sum=741
11 26 37 30 14 18 16 23 32 21 25 01 13 34 10 28 17 05 31 20 33 24 35 19 04 15 36 29 06 03 27 22 09 08 02 07 38 12 __sum=741
12 38 07 02 08 09 22 27 03 06 29 36 15 04 19 35 24 33 20 31 05 17 28 10 34 13 01 25 21 32 23 16 18 14 30 37 26 11 __sum=741
13 26 30 06 36 18 16 29 27 01 11 25 19 34 04 10 28 12 20 17 31 24 33 05 32 15 35 23 03 21 22 37 08 38 09 02 07 14 __sum=741
14 07 02 09 38 08 37 22 21 03 23 35 15 32 05 33 24 31 17 20 12 28 10 04 34 19 25 11 01 27 29 16 18 36 06 30 26 13 __sum=741
15 07 08 30 06 02 38 14 27 23 21 03 35 32 04 13 24 10 28 20 17 12 05 31 33 34 19 25 01 11 29 22 18 36 37 26 09 16 __sum=741
16 09 26 37 36 18 22 29 11 01 25 19 34 33 31 05 12 17 20 28 10 24 13 04 32 35 03 21 23 27 14 38 02 06 30 08 07 15 __sum=741
17 07 30 08 02 26 37 22 27 03 29 19 21 32 33 05 20 28 31 12 10 34 24 04 13 01 25 15 11 23 16 35 14 36 06 09 38 18 __sum=741
18 38 09 06 36 14 35 16 23 11 15 25 01 13 04 24 34 10 12 31 28 20 05 33 32 21 19 29 03 27 22 37 26 02 08 30 07 17 __sum=741
19 07 26 09 38 18 14 15 22 21 27 04 11 25 33 10 01 34 28 31 05 12 17 24 35 13 32 29 03 23 16 37 06 02 36 30 08 20 __sum=741
20 08 30 36 02 06 37 16 23 03 29 32 13 35 24 17 12 05 31 28 34 01 10 33 25 11 04 27 21 22 15 14 18 38 09 26 07 19 __sum=741

17501 *Expand*-Aufrufe

Permutation: 26 7 9 8 30 2 36 6 14 18 22 16 29 27 23 32 37 21 3 15 11 25 19 38 1 4 13 24 33 10 5 35 28 34 17 20 31 12

$n = 110$

Diese Lösung ist so groß, dass wir sie nur online unter https://bwinf.de/bundeswettbewerb/der-aktuelle-wettbewerb/2-runde/material-362/ zur Verfügung stellen können.

1.9 Anhang: Optimale Obergrenze

Wir suchen eine Formel, die uns für gegebenes n einen vernünftigen Wert für die Obergrenze an rekursiven Aufrufen liefert, bevor zur nächsten Permutation gewechselt wird. Wird die Obergrenze zu niedrig gewählt, werden zu viele Permutationen abgebrochen, bevor sie eine

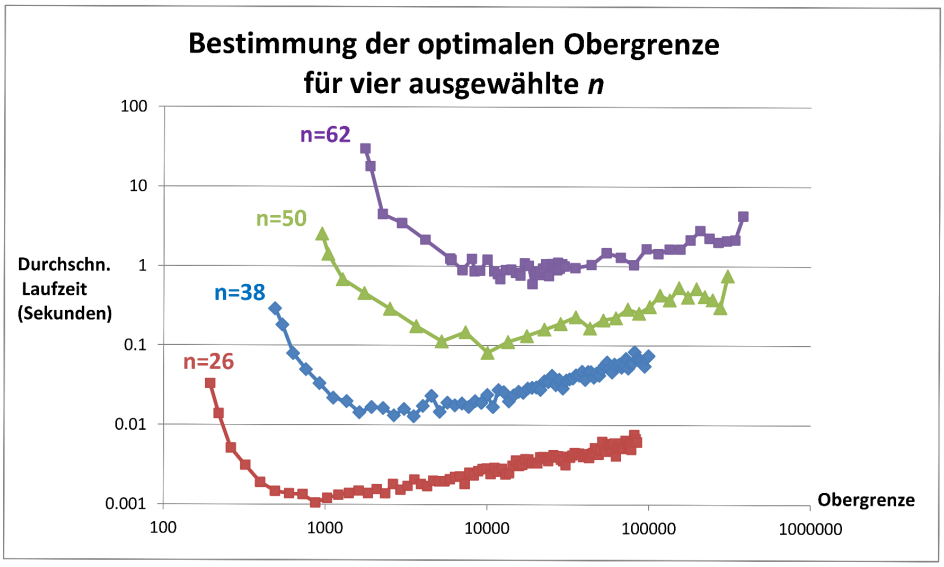


Abbildung 7: Suche nach der optimalen Obergrenze (x-Achse), die den Minima der Kurven entspricht. Je niedriger die benötigte Zeit pro Lösung (y-Achse), desto besser.

Lösung liefern. Ist die Obergrenze zu hoch, wird viel Zeit in schlechte Permutationen verschwendet. Wir wollen deshalb einen Kompromiss finden.

Dazu haben wir für verschiedene n und verschiedene Obergrenzen die durchschnittliche Zeit gemessen, die das Programm zum Finden einer Lösung benötigt. Die Ergebnisse sind exemplarisch für vier verschiedene n in Abb. 7 gezeigt. Die x-Achse (Obergrenze) ist logarithmisch skaliert, um die Details bei kleinen Werten besser aufzulösen.

Aus den Diagrammen können wir jeweils die ungefähre optimale Obergrenze T_{opt} ablesen. Diese sind in Tabelle 1 aufgeführt. Diesen Wert vergleichen wir mit der minimalen Anzahl an rekursiven Aufrufen, die jeweils nötig wären, um überhaupt eine Lösung zu generieren. Diese ist gegeben durch $T_{\text{min}} = \frac{L}{2} = \frac{n^2}{4}$. Das Verhältnis dieser beiden Zahlen zueinander, $T_{\text{opt}}/T_{\text{min}}$, kann dann nach einem Zusammenhang mit n untersucht werden. Es zeigt sich, dass $T_{\text{opt}}/T_{\text{min}} \approx \frac{n}{3}$ eine brauchbare Näherung ist. Daraus ergibt sich

$$T_{\text{opt}} \approx T_{\text{min}} \frac{n}{3} = \frac{n^3}{12}. \tag{5}$$

Dies ist natürlich nur eine grobe Abschätzung des Zusammenhangs, aber für unsere Zwecke ausreichend, um für alle n eine Abbruchgrenze berechnen zu können.

1.10 Bewertungskriterien

- Die grundsätzliche Erwartungshaltung an das Programm ist (ohne Plus- oder Minuspunkte), dass es Mauern mit einer Maximalhöhe für $n = 15$ bis etwa $n = 19$ in annehmbarer Zeit erzeugt. Das Programm sollte (nicht zwingend maximale) Lösungen in der ungefähren Größenordnung $h = \lfloor \frac{n}{2} \rfloor + 1$ erzeugen. Für größere n etwa ab $n = 20$ ist es

n	$T_{\min}$	T_{opt}	$T_{\text{opt}}/T_{\min}$ (gerundet)	$\frac{n}{3}$ (gerundet)
26	169	900	5	8
38	361	5000	14	13
50	625	10000	16	17
62	961	19000	20	20

Tabelle 1: Suche nach einem Zusammenhang zwischen n und $T_{\text{opt}}/T_{\min}$.

in Ordnung, wenn gegebenenfalls nur noch Spezialfälle gelöst werden können (so wie unser Spiegeltrick nur für eine gerade Anzahl an Reihen funktioniert).

- Bis mindestens $n = 10$ muss jedoch eine Mauer der Maximalhöhe $h = \lfloor \frac{n}{2} \rfloor + 1$ erzeugt werden, sonst gibt es einige Minuspunkte. Findet das Programm sogar für größere n Mauern der maximalen Höhe in akzeptabler Zeit, gibt es Pluspunkte ab $n = 20$ sowie noch mehr ab $n = 30$ bzw. $n = 40$, ansonsten wenige Minuspunkte unter $n = 15$ bis $n = 10$.
- Die erzeugten Mauern müssen stets korrekt sein, also den Vorgaben der Aufgabenstellung entsprechen.
- Für besonders große n darf die Berechnungszeit des Programms im Stundenbereich liegen, ansonsten eher nur im Sekunden- oder Minutenbereich.
- Das Verfahren bzw. seine Implementierung darf nicht auf unnötige Weise aufwändig sein und damit die Konstruktion höherer Mauern verhindern.
- Die Erstellung von Lösungen für ungerade n sollte nicht vernachlässigt werden, sondern geeignet diskutiert werden.
- Es sollen Überlegungen und Aussagen über den Zusammenhang zwischen n und der maximalen Mauerhöhe $h = \lfloor \frac{n}{2} \rfloor + 1$ vorhanden sein, am besten analytischer Natur.
- Einige sinnvolle Gedanken zur Laufzeit des Programms werden erwartet und müssen verständlich sein.
- Der gewählte Algorithmus (inklusive aller Optimierungen) zum Erzeugen einer Mauer soll nachvollziehbar beschrieben und durch ausreichend viele aussagekräftige Beispiele für verschiedene n verdeutlicht werden.
- Es muss für mindestens drei verschiedene n einschließlich des konkreten Falls $n = 10$ eine konstruierte Lösung angegeben werden. Wenn die erfolgreiche Erzeugung von Mauern mit großem n durch das Programm behauptet wird, wird hierfür die Angabe von berechneten Beispiellösungen erwartet.
- Die Ausgabe des Programms muss korrekt und übersichtlich sein. Insbesondere soll sie nicht nur für kleine n derart sein, dass die Korrektheit der Ausgabe einfach überprüfbar ist.
- Mögliche Erweiterungen der Aufgabenstellung wären zum Beispiel Rundmauern wie bei einem Turm oder die Zulassung einer vorgegebenen maximalen Anzahl von übereinanderliegenden Fugen.

Aufgabe 2: Wehret den Wildschweinen!

2 Erstellen eines kritischen Pfads

Das Ziel der Erdarbeiten ist es, dass jeder Weg für ein Wildschwein vom Nordrand zum Südrand mindestens einmal einen Höhenunterschied von mehr als einem Meter beinhaltet. Der *Weg* eines Wildschweins sei dabei die Abfolge von besuchten Planquadraten. Geht ein Wildschwein von einem Planquadrat zu einem benachbarten Planquadrat, überschreitet es dabei die *Linie* zwischen beiden Quadraten. Kann eine Linie von einem Wildschwein nicht mehr bewältigt werden, da der Betrag der Höhendifferenz größer oder gleich einem Meter ist, handelt es sich um eine *kritische* Linie.

Das Ziel ist genau dann erreicht, wenn mindestens ein zusammenhängender Pfad aus kritischen Linien vom linken zum rechten Feldrand gefunden wurde. Ein solcher Pfad sei ein *kritischer* Pfad. Existiert dagegen kein kritischer Pfad, so gibt es mindestens einen möglichen Weg für ein Wildschwein von Norden nach Süden.

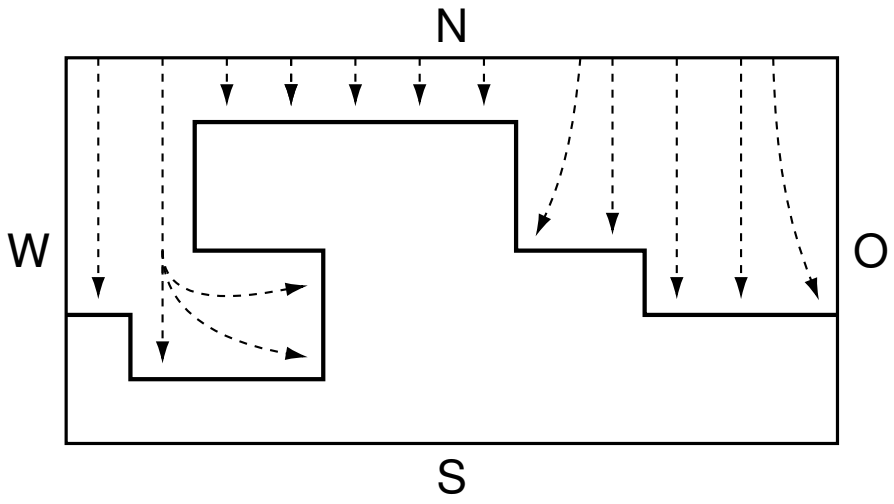


Abbildung 8: Kritischer Pfad.

Das Problem wird gelöst, indem ein kritischer Pfad mit möglichst wenigen Erdarbeiten konstruiert wird. Ist ein bestimmter Pfad noch nicht kritisch, so sind noch von Wildschweinen passierbare Linien enthalten. Um eine Lösung zu finden, werden diese Linien durch Erdarbeiten in kritische Linien umgewandelt.

2.1 Umbau von Linien

Wir betrachten also zwei benachbarte Planquadrate der Höhen a und b, deren gemeinsame Linie nach der Bearbeitung kritisch sein soll. Die zu erfüllende Bedingung an den Betrag der Höhendifferenz lautet: $\Delta = |b - a| \geq 1$.[3] Es sollen dabei nur Erdarbeiten bezüglich jeweils

[3] 1m kostet laut Aufgabenstellung 100€.

beider betrachteten Planquadrate unternommen werden. Um die Differenz zu erhöhen, ist es immer günstiger, den schon bestehenden Höhenunterschied zu vergrößern.

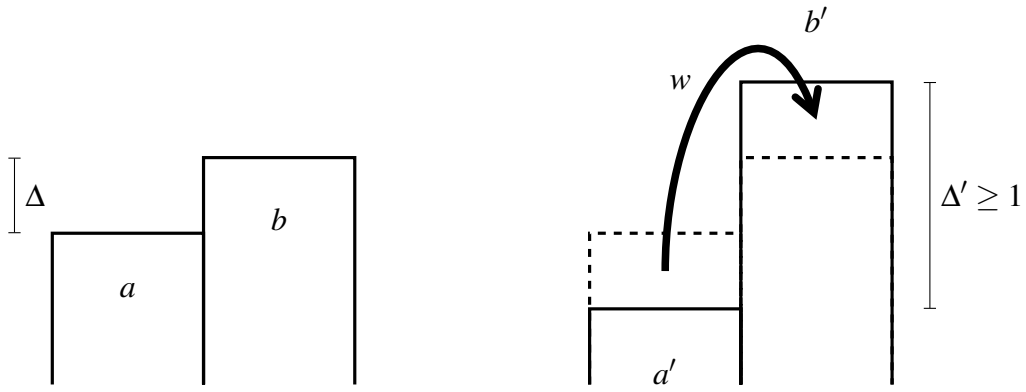


Abbildung 9: Umschaufeln zur kritischen Linie.

Dazu wird Erde vom niedrigeren Quadrat zum höheren Quadrat geschaufelt. Wurden w Erdarbeiten auf diese Weise verrichtet, gilt für die jeweiligen Höhen a' und b' nach verrichteter Arbeit:

$$a' = a - w \qquad b' = b + w$$

Um die genannte Bedingung zu erfüllen, muss unter der Annahme $b' > a'$ für die neue Differenz Δ' gelten:

$$\begin{aligned} 1 \leq \Delta' = b' - a' = (b+w) - (a-w) = (b-a) + 2w = \Delta + 2w \\ 1 - 2w \leq \Delta \\ -2w \leq \Delta - 1 \end{aligned}$$

Aus dieser Gleichung ergibt sich die geringste Menge an Erdarbeiten w zum Umbauen einer Linie mit Höhenunterschied Δ zu einer kritischen Linie:

$$w \geq \frac{1}{2}(1 - \Delta)$$

2.2 Heuristik für die Pfadfindung

Zur Minimierung der Kosten wird der Pfad gewählt, der die wenigsten Erdarbeiten insgesamt erfordert, um kritisch zu werden. Es bietet sich an, die gesamten Erdarbeiten als Summe der einzelnen benötigten Arbeiten aller Linien zu errechnen. Allerdings gilt dies nur unter der Annahme, dass einzelne Erdarbeiten an verschiedenen Linien unabhängig voneinander sind. Leider ist dies jedoch nicht der Fall. Liegen zwei Linien am selben Planquadrat an, tritt einer von zwei Fällen auf:

(1) Die jeweiligen Erdarbeiten beider Linien entnehmen beide dem betroffenen Planquadrat Erde oder fügen ihm beide Erde hinzu.

(2) Die Erdarbeiten der einen Linie entnehmen Erde vom betroffenen Planquadrat, während die Erdarbeiten der anderen Linie Erde hinzufügen.

Im Fall (1) wird überflüssige Arbeit verrichtet: Nach dem Umschaufeln durch Erdarbeiten an der einen Linie erreicht das betroffene Planquadrat bereits eine bessere Höhe, sodass durch Erdarbeiten an der zweiten Linie sowie auch insgesamt weniger Erde umgeschaufelt werden muss. Im Fall (2) kommt es zu einem Konflikt, durch welchen nach Tätigung der Erdarbeiten an beiden Linien nicht garantiert werden kann, dass nun beide Linien kritisch sind. Es muss zusätzliche Erdarbeit verrichtet werden, da geschaufelte Erde für eine Linie mehr Arbeit für die andere Linie bedeutet.

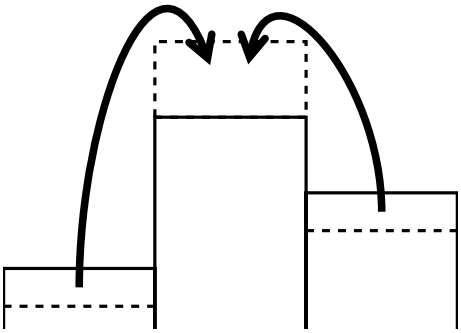


Abbildung 10: Fall (1).

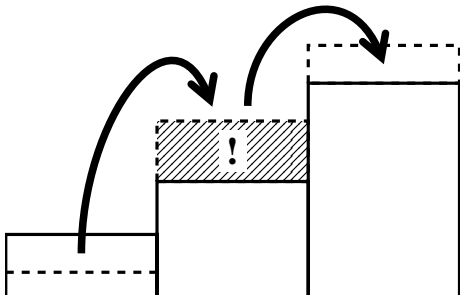


Abbildung 11: Fall (2).

Die Summe der einzelnen Erdarbeiten kann somit nicht als exaktes Maß für die benötigten Erdarbeiten zur Umsetzung eines Pfades gewählt werden. Stattdessen wird diese Summe nur als heuristisches Maß verwendet, um verschiedene Pfade miteinander zu vergleichen. Die Heuristik wählt den Pfad von Westrand zu Ostrand, welcher die geringste Summe an Erdarbeiten für die einzelnen Linien aufweist. Dieser Pfad sei der *geringste* Pfad. Die Heuristik erzeugt nicht notwendigerweise stets die beste Lösung, läuft aber dafür in akzeptabler Laufzeit.

Das Problem des geringsten Pfads wird in einen Graphen $G = (V, E)$ eingebettet. Jede Kante aus E stellt eine Linie auf dem Brachland dar. Die Knoten aus V liegen auf den Eckpunkten der Planquadrate. Da die Ränder abgezäunt sind, werden die Linien vernachlässigt, die direkt am West- oder Ostrand anliegen. Der Nord- und Südrand des Brachlandes wird für die Wildschweine immer als passierbar angenommen: Jedes Wildschwein startet auf einem Planquadrat am Nordrand und läuft zu einem Planquadrat am Südrand. Somit fallen auch Linien weg, welche inzident zu mindestens einem Knoten am Rand sind.

Die Gewichtung $w(e)$ für jede Kante $e \in E$ entspricht den benötigten Erdarbeiten für den Umbau zu einer kritischen Linie. Die *Länge* eines Pfads innerhalb des Graphen G entspricht demnach der Summe der einzelnen Erdarbeiten. Der geringste Pfad vom Westrand zum Ostrand ist der kürzeste gewichtete Pfad von einem Knoten am Westrand zu einem Knoten am Ostrand im Graphen.

Es werden zwei virtuelle Knoten s und t mit jeweils einer Kante an alle Knoten des Westrandes bzw. des Ostrandes angefügt. Die Gewichtung dieser Kanten sei 0. Auf diese Weise wird das Problem darauf reduziert, den kürzesten Pfad vom Knoten s zum Knoten t zu finden.

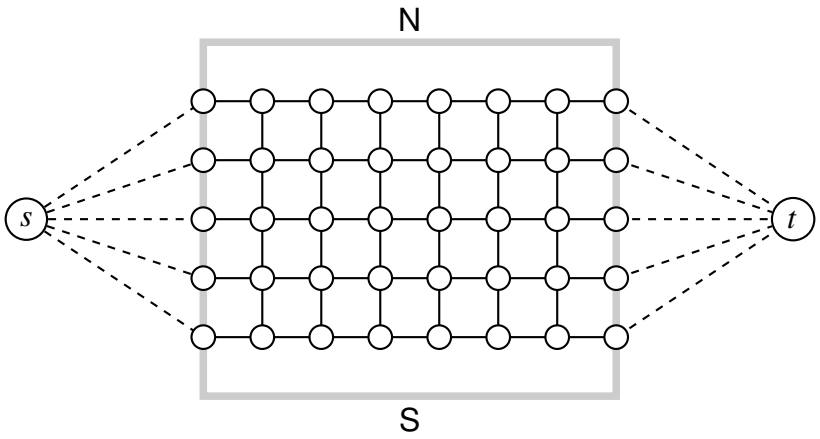


Abbildung 12: Einbettung in einen Graphen.

Um den kürzesten Pfad zu erhalten, wird der Dijkstra-Algorithmus[4] im ungerichteten, gewichteten Graphen G angewendet. Die berechneten geringsten Pfade zu den vorgegebenen Beispielen sind im Abschnitt 2enthalten.

2 Lösung mittels Dualgraph

Ist durch die Heuristik ein geringster Pfad ermittelt worden, gilt es diesen noch umzubauen, so dass ein kritischer Pfad entsteht. Die dazu unternommenen Erdarbeiten sollen dabei folgende Ziele verfolgen (siehe Abschnitt 2.2):

(1) Ausnutzen begünstigender Umstände.

(2) Auflösen möglicher Konflikte.

Um die genannten Arten zu erkennen und umzusetzen, wird ein Dualgraph erzeugt. In diesem Graphen werden die Beziehungen zwischen voneinander abhängigen Erdarbeiten dargestellt, welche von der Heuristik vernachlässigt werden. Alle Erdarbeiten werden zur Vereinfachung nur an Linien des gewählten Pfades getätigt. Demnach finden Veränderungen nur in den am Pfad direkt anliegenden Planquadraten statt.[5]

Alle am Pfad anliegenden Planquadrate sind in einem Knoten im Dualgraphen repräsentiert. Liegt die Linie zwischen zwei benachbarten Quadraten auf dem Pfad, so wird eine gerichtete Kante zwischen den jeweiligen Knoten im Dualgraph gezogen. Die Richtung der Kante zeigt dabei immer vom niedrigeren Quadrat zum höheren Quadrat.

[4] Wikipedia, 2018. http://de.wikipedia.org/wiki/Dijkstra-Algorithmus

[5] Der Einbeziehung weiter außen liegender Quadrate erwiese sich in manchen Fällen zwar als vorteilhaft, würde aber eine deutlich erhöhte algorithmische Komplexität bedeuten.

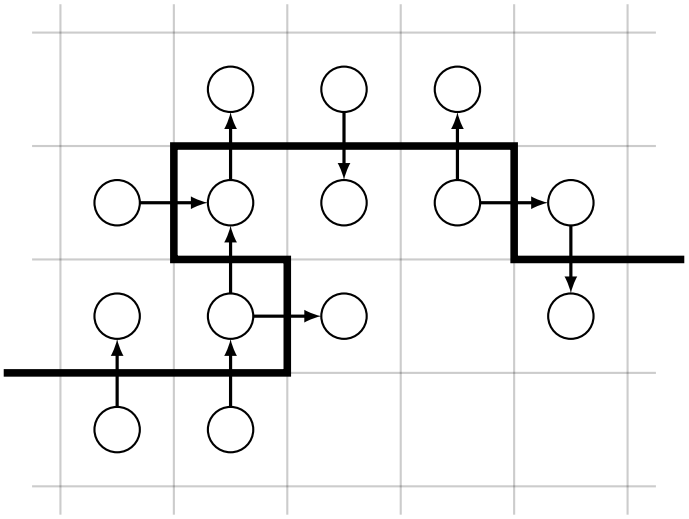


Abbildung 13: Kritischer Pfad mit entsprechendem Dualgraph.

Der Dualgraph ist meist nicht zusammenhängend. Unverbundene Teile des Graphen haben keinerlei Bezug zueinander, sodass sie unabhängig voneinander betrachtet werden können.

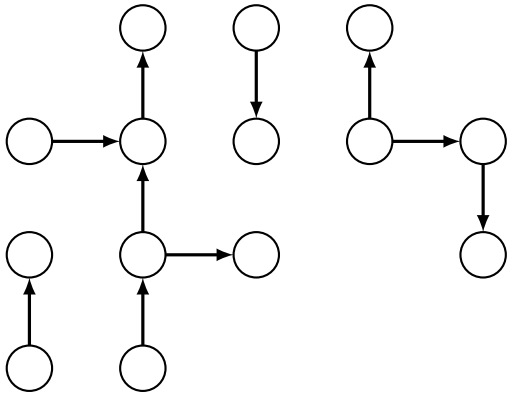


Abbildung 14: Unabhängige Teilgraphen.

Es gilt im Folgenden, die einzelnen zusammenhängenden Teilgraphen von Konflikten zu befreien.

Es bleibt anzumerken, dass ein Dualgraph immer zyklenfrei ist. Wäre ein Zyklus enthalten, würde das zu einem Widerspruch führen, da jeder Nachfolger im Zyklus größer sein muss als der bisherige Knoten. So müsste ein beliebiger Knoten im Zyklus größer als sein Vorgänger sein. Diese Situation ist in Abbildung (15) dargestellt.

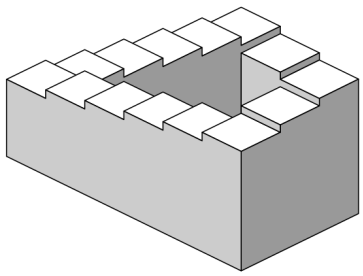


Abbildung 15: Unmöglicher Höhenverlauf im Fall eines Zyklus im Dualgraphen.

2.1 Konfliktlösung

Die naiven Erdarbeiten, von welchen in der Heuristik ausgegangen wird, sind im Dualgraph wiedererkennbar: Da die Arbeit an einer Linie immer in Richtung des schon bestehenden Gefälles stattfindet, zeigen die Kanten direkt in Richtung der Erdarbeiten, die an jeder Linie geplant waren. Für Knoten bedeuten eingehende Kanten zugeschaufelte, ausgehende Kanten abgetragene Erde.

Knoten, welche sowohl eingehende als auch ausgehende Kanten haben, bergen Konflikte. Knoten hingegen, die inzident ausschließlich zu entweder ein- oder ausgehenden Kanten sind, ermöglichen Einsparungen bei beiden Erdarbeiten.

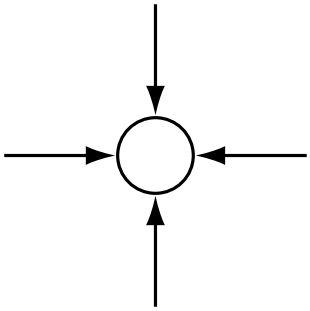


Abbildung 16: Konfliktfreier Knoten.

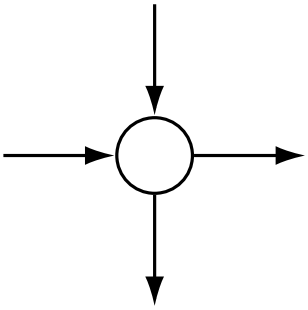


Abbildung 17: Knoten mit konfliktären Kanten.

Eigenschaften eines konfliktfreien Graphen

Ein vollständig konfliktfreier Graph besteht nur aus konfliktfreien Knoten: Jeder Knoten besitzt entweder nur eingehende oder nur ausgehende Kanten. Die Ausrichtung der Kanten sei dabei die *Ausrichtung* des Knoten. Zwei adjazente Knoten können durch eine Kante nicht gleichzeitig eingehend oder ausgehend sein, d.h. adjazente Knoten besitzen stets verschiedene Ausrichtungen. Der Graph ist demnach 2-knotenfärbbar (siehe Abb. 18).

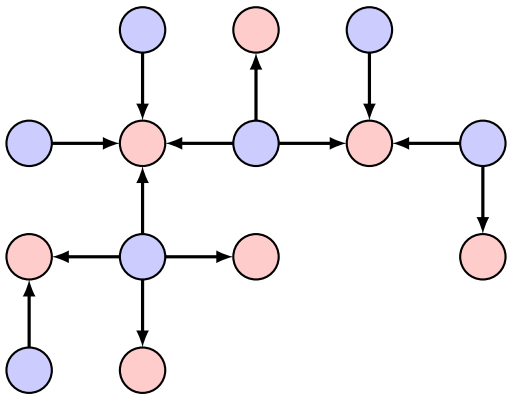


Abbildung 18: 2-Färbung eines konfliktfreien Graphen.

Kantendrehungen im konfliktären Graphen

Um einen konfliktfreien Graphen ausgehend von einem konfliktären Graphen zu erhalten, müssen Kanten in ihrer Ausrichtung verändert werden. Kehrt sich die Richtung einer Kante (a,b) zur entgegengesetzten Kante (b,a) um, so sei die Kante *gedreht*.

Die Drehung einer Kante erfordert Erdarbeiten. Da das Gefälle umgekehrt wird, muss zusätzliche Arbeit verrichtet werden, um das vorherige Gefälle auszugleichen, bevor das entgegengesetzte Gefälle geschaufelt wird. Das Ausgleichen der Differenz Δ kostet dabei $\frac{1}{2}\Delta$, und das Aufschaufeln der neuen Differenz 1 kostet $\frac{1}{2}$. Somit gilt für die zusätzliche Erdarbeit δ_w neben der Erdarbeit $w \geq \frac{1}{2}(1-\Delta)$ (vgl. Abschnitt (2.1)):

$$\begin{aligned} w + \delta_w &= \frac{1}{2}\Delta + \frac{1}{2} \\ \frac{1}{2}(1-\Delta) + \delta_w &\geq \frac{1}{2}(1+\Delta) \\ \delta_w &\geq \frac{1}{2}(1+\Delta) - \frac{1}{2}(1-\Delta) = \Delta \end{aligned}$$

Die zusätzlich benötigte Erdarbeit δ_w für das Drehen einer Kante ist also mindestens so groß wie die anfänglich vorhandene Differenz.

Es sind immer zwei verschiedene Färbungen möglich, die komplementär zueinander sind. Je nach Färbung werden unterschiedliche Kanten gedreht, welche verschiedene Mengen an Erdarbeit hinzufügen. Es wird letztlich stets die Färbung gewählt, die die geringere Summe an zusätzlichen Arbeiten benötigt.

2.2 Bearbeitung des Brachlandes

Nachdem die 2-Färbung mit geringsten zusätzlichen Arbeiten unternommen wurde, sind alle notwendigen Erdarbeiten bekannt: Jede Kante im konfliktfreien Dualgraph gibt eine Erdarbeit

zwischen den Planquadraten der inzidenten Knoten in eine bestimmte Richtung an. Die Menge an zu bewegender Erde ist abhängig davon, ob die Kante gedreht wurde oder nicht:

$$w_{end} = \begin{cases} w & \\ w + \delta w & \end{cases} = \frac{1}{2}(1 \pm \Delta)$$

Die einzelnen Arbeiten werden sortiert nach abnehmender Erdmenge ausgeführt. Hierbei werden mögliche Begünstigungen ausgenutzt: Durch die 2-Färbung ist jeder Knoten eingehend oder ausgehend. So steigt die Höhe eines eingehenden Knotens mit jeder Erdarbeit stetig an, während die Höhe eines ausgehenden Knotens fällt. Damit ist garantiert, dass ein eingehender Knoten, nachdem er unter eine bestimmte Höhe während den Erdarbeiten fällt, darunter bleiben wird. Gleiches gilt umgekehrt für ausgehende Knoten. Diese Eigenschaft des konfliktfreien Dualgraphen ist seine *Invariante*.

Ist durch vorherige Erdarbeiten die Differenz bereits vorhanden, die mit einer Erdarbeit geschaffen werden sollte, kann diese ausgelassen werden. Durch die Invariante des Graphen ist garantiert, dass nachfolgende Arbeiten die Differenz erhalten oder nur vergrößern, sodass die entsprechende Linie kritisch bleibt: Erdarbeiten an Linien, welche durch vorherige Arbeit schon kritisch sind, können unter Erhaltung der Invariante des Dualgraphen ausgelassen werden. Die ausgelassenen Erdarbeiten bedeuten Einsparungen in den Gesamtkosten. Sind alle Erdarbeiten getätigt worden bzw. alle verbliebenen Erdarbeiten redundant, ist jede Linie kritisch, und der geringste Pfad ist somit ein kritischer Pfad.

2 Bemerkungen zur Laufzeit

In einer $\ell \times \ell$-Matrix gibt es ℓ^2 Planquadrate. In den im Abschnitt 2 beschriebenen Graphen sind alle $(\ell+1)^2$ Eckpunkte bis auf die $2(\ell+1)$ Eckpunkte am Nord- und Südrand als Knoten enthalten:

$$n = (\ell+1)^2 - 2(\ell+1) = (\ell+1)(\ell-1)$$

Nach Eulers Charakteristik für planare Graphen[6] verhält sich die Anzahl m von Kanten proportional zu der Anzahl n von Knoten, da sich die Zahl der Planquadrate asymptotisch der Zahl der Knoten annähert. Für die Konstruktion des Graphen muss jeder Knoten und jede Kante einmal bearbeitet werden, was $\mathcal{O}(n+m) = \mathcal{O}(n)$ Schritte erfordert.

Der kürzeste Pfad enthält im ungünstigsten Fall alle n Knoten. Die 2-Färbung betrachtet jeden Knoten genau einmal, was $\mathcal{O}(n)$ bedeutet. Die Konfliktlösung betrachtet die Drehung jeder Kante genau einmal, da eine Kante entweder in einer Färbung oder in ihrem Komplement gedreht wird. Dafür wird $\mathcal{O}(m)$ benötigt. Dies trifft auch auf die Bearbeitung zu, in der pro Kante eine Erdarbeit getätigt werden muss.

Da in allen Teilschritten das Finden des kürzesten Pfades überwiegt, ergibt sich asymptotisch eine Gesamtlaufzeit von $\mathcal{O}(n^2)$, die für den Dijkstra-Algorithmus bekannte Laufzeit.[7]

[6] Wolfram MathWorld, 2018. http://mathworld.wolfram.com/EulerCharacteristic.html
[7] Wikipedia, 2018. http://de.wikipedia.org/wiki/Dijkstra-Algorithmus

2 Diskussion der Ergebnisse

Die Ergebnisse im Abschnitt 2 für die bereitgestellten Beispiele zeigen, dass die beschriebene Heuristik für die Aufgabenlösung meist sehr hilfreich ist. Für das Beispiel `wildschwein2.txt` ist sogar die optimale Lösung generiert worden: In jeder Spalte von Planquadraten von Nord nach Süd muss mindestens einmal eine kritische Linie vorkommen, da sonst eine Passage für die Wildschweine vorhanden ist. In diesem Beispiel sind alle Höhen gleich groß, sodass das Bauen einer kritischen Linie überall $\frac{1}{2}$ kostet. Der kürzeste Pfad in einem gleich gewichteten Kantengraphen von Ost nach West ist eine Gerade. Da jede Linie im Pfad genau eine Spalte abdeckt, ist der Pfad optimal. Die gesamten Erdarbeiten ergeben sich aus der Anzahl von Spalten ℓ multipliziert mit den Erdarbeiten pro Linie $\frac{1}{2}$.

Diese Überlegung ergibt auch eine allgemeine Obergrenze für Erdarbeiten. Jede Linie benötigt maximal $\frac{1}{2}$ Erdarbeiten, um kritisch zu werden. Wird ein Pfad als gerade Linie von West nach Ost gezogen, so beinhaltet dieser ℓ Linien. So ist es immer möglich, einen kritischen Pfad mit maximal $\frac{1}{2}\ell$ Erdarbeiten zu konstruieren.

Je mehr Umwege und Wendungen der Pfad nimmt, desto größer werden Abhängigkeiten zwischen Erdarbeiten. Komplexere Abhängigkeiten werden durch die Konfliktlösung nur suboptimal gelöst, da Kantendrehung höhere Kosten erzeugt. Allerdings kommt dies selten vor. Der Grund dafür ist, dass größere Umwege pro Linie verhältnismäßig weniger Kosten aufweisen. Da Erdarbeiten pro Linie mit $\frac{1}{2}$ nach oben begrenzt sind, ist es in den meisten Fällen günstiger, die kürzere Route zu wählen. Da Umwege mehr Linien enthalten, benötigen diese weniger Erdarbeiten, um insgesamt kostengünstiger zu sein. Demnach beinhalten Umwege weniger Kosten, was enthaltene Konflikte weniger ins Gewicht fallen lässt.

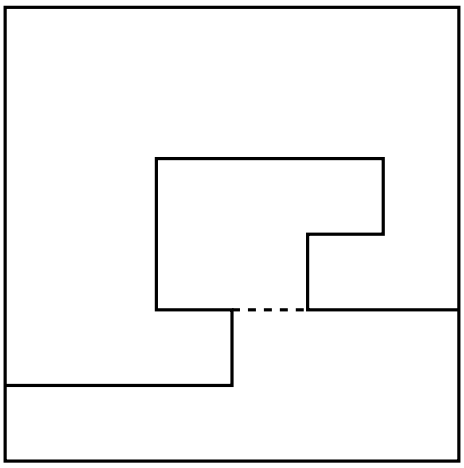


Abbildung 19: Abkürzung statt Umwegen.

In Brachland mit stark variierenden Höhen der einzelnen Planquadrate schneidet die Heuristik sehr gut ab, siehe hierzu die zufällig generierten Beispiele im Abschnitt 2.1. Der geringste Pfad nutzt die vorhandenen großen Gefälle aus, sodass nur wenige zusätzliche Erdarbeiten anfallen.

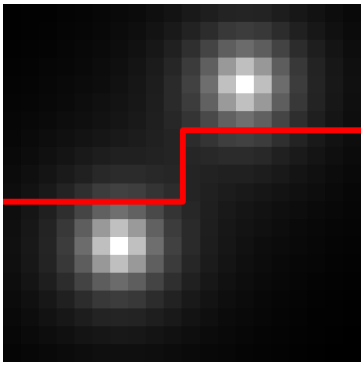
In Fällen, welche weder stark variierendes noch uniformes Brachland darstellen, treten die Nachteile der Konfliktlösung zu Tage. Betrachtet man beispielsweise den geringsten Pfad zu

`wildschwein3.txt`, so ist ein Knick kurz vor dem Ostrand zu sehen. Der Pfad hätte auch gerade bleiben können, da die Linie in Richtung Norden schon kritisch ist und 0 Erdarbeiten erfordert. Dass dieser umständliche Weg genommen wurde, liegt allein an der Reihenfolge der Kanten während der Konstruktion des Graphen. Der direkte Weg hätte diegleichen Kosten.

2 Ergebnisse

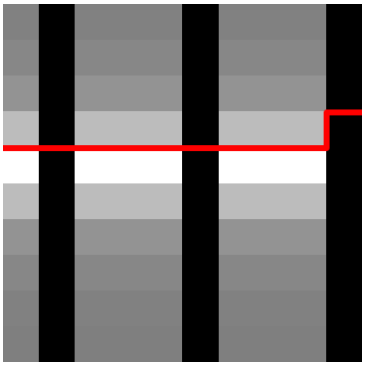
2.1 Vorgegebene Beispiele

Die Höhen der einzelnen Planquadrate sind in den folgenden Abbildungen als sogenannte *Heatmaps*[8] dargestellt, wobei hellere Farbtöne höhere Erhebungen darstellen. Die rote Linie zeigt die berechnete Lösung für das jeweilige Beispiel, der Nordrand befindet sich jeweils oben, der Südrand unten.

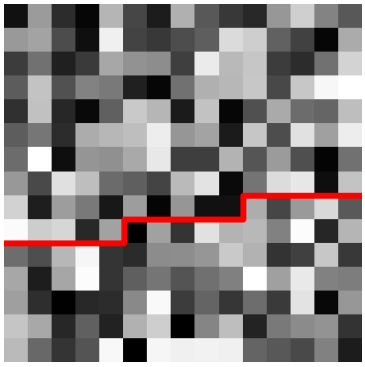


`wildschwein1.txt`

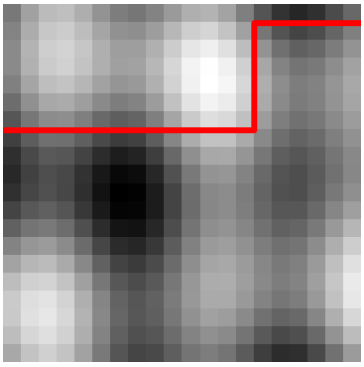
`wildschwein2.txt`



`wildschwein3.txt`



`wildschwein4.txt`

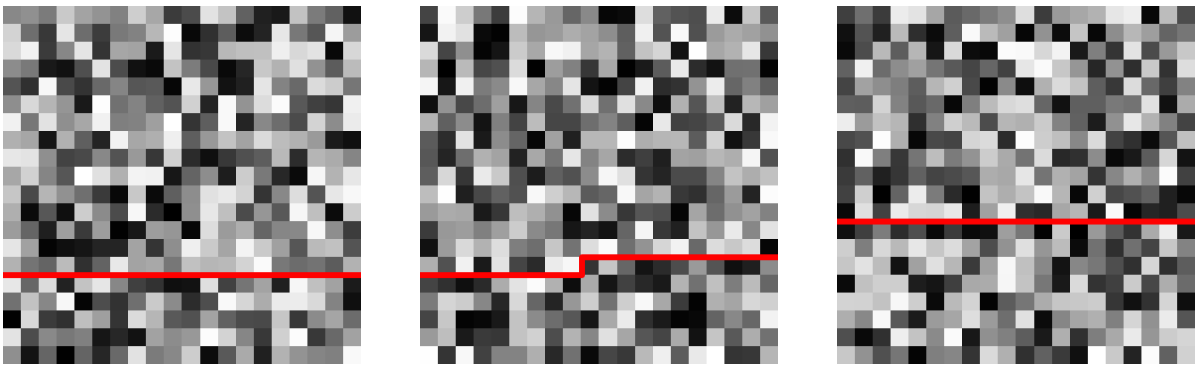


`wildschwein5.txt`

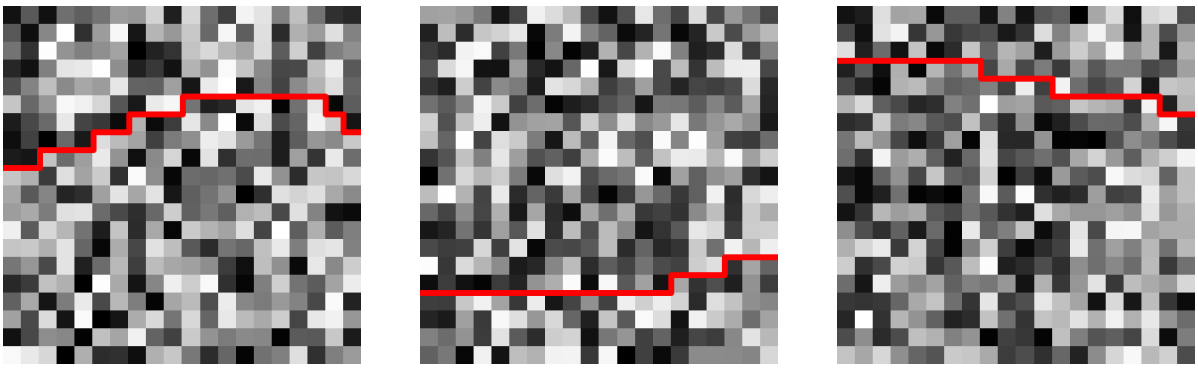
2.2 Zufällig generierte Beispiele

Es wurden zusätzlich zufällige Beispiele mit stark variierenden Höhen der einzelnen Planquadrate generiert. Dafür wurden innerhalb von 20×20-Matrizen gleichverteilte, zufällige Höhen aus dem Intervall $[0; r]$ verwendet und verschiedene Parameter r angewandt.

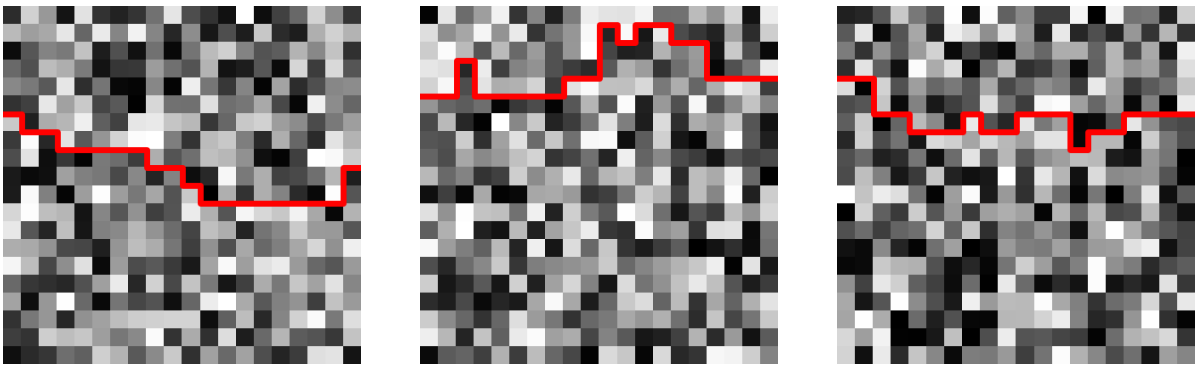
[8] Wikipedia, 2018. http://de.wikipedia.org/wiki/Heatmap



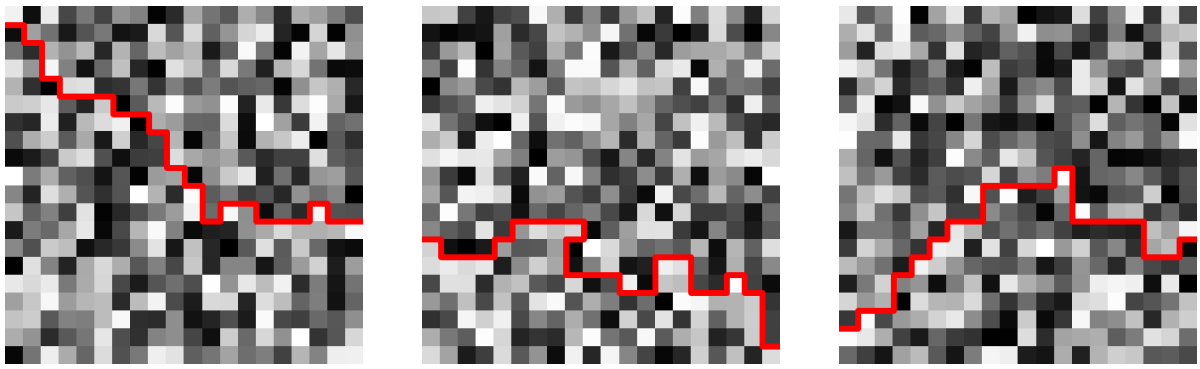
$r = 0.5$



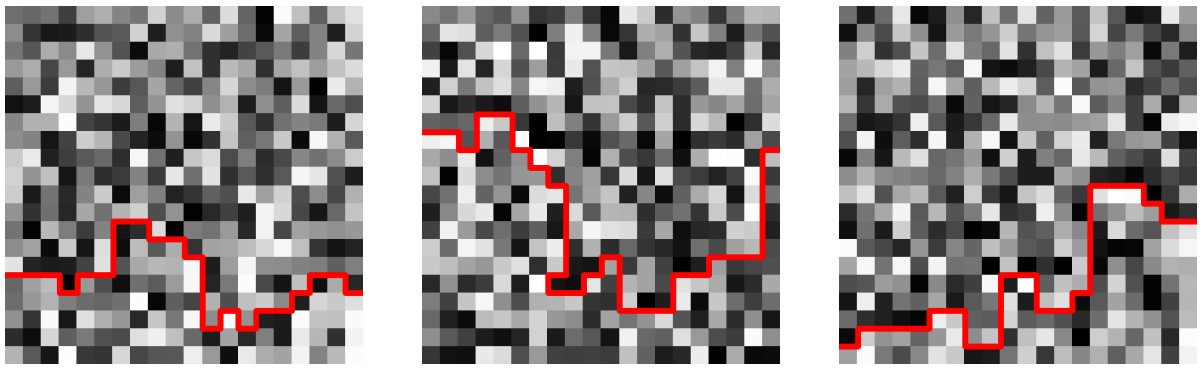
$r = 1.0$



$r = 1.5$



$r = 2.0$



$r = 2.5$

2 Vergleich mit alternativen Ergebnissen

Ein alternativer Ansatz zur Lösung der Aufgabe ist lineare Programmierung (LP).[9] Hierbei wird die Aufgabestellung als Optimierungsproblem mittels eines gemischt-ganzzahligen linearen Programms, ein sogenanntes *Mixed Integer Linear Program* (MILP), formuliert.[10] Ein MILP beinhaltet sowohl Variablen für ganze Zahlen als auch für reelle Zahlen sowie eine lineare Zielfunktion und lineare Ungleichungen, d.h. Nebenbedingungen mit den Variablen. Die Zielfunktion wird unter Einhaltung der Nebenbedingungen optimiert. Zum Erstellen und Lösen linearer Programme gibt es seit vielen Jahren zahlreiche professionelle Softwarewerkzeuge wie z.B. CPLEX[11].

Da wir eine Lösung mittels linearer Programmierung von den Teilnehmerinnen und Teilnehmern nicht erwarteten, wird an dieser Stelle keine Alternativlösung mittels MILP näher erläutert. Wir wollen nur die alternativen Ergebnisse, die mit einem derartigen MILP-Ansatz unter Verwendung von CPLEX erreichbar sind (siehe Abb. 20 und Abb. 21), in der Tabelle 2 mit denen der obigen, graphenbasierten Lösung vergleichen. Die dritte Lösung in der Tabelle 2 nützt

[9] Wikipedia, 2018. http://de.wikipedia.org/wiki/Lineare_Optimierung
[10] Wikipedia, 2018. http://de.wikipedia.org/wiki/Ganzzahlige_lineare_Optimierung
[11] IBM ILOG CPLEX Optimization Studio, 2018. http://www.ibm.com/products/ilog-cplex-optimization-studio

Tabelle 2: Ergebnisse zu den fünf vorgegebenen Beispielen basierend auf drei verschiedenen Lösungsansätzen.

Beispiel	MILP	Graph	Parallel-Graph
`wildschwein1.txt`	6.064	6.931	6.287
`wildschwein2.txt`	6.000	6.000	6.000
`wildschwein3.txt`	3.075	3.625	3.075
`wildschwein4.txt`	3.451	3.635	3.517
`wildschwein5.txt`	4.065	4.645	4.108

die Nebenläufigkeit der angewandten Algorithmen zu einer graphenbasierten Lösung durch entsprechende Parallelisierung und spart hierdurch weitere Zeit bei der Lösungssuche.

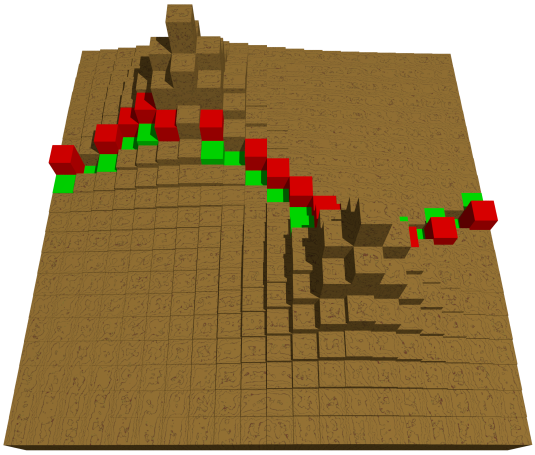


Abbildung 20: CPLEX-Lösung für das Beispiel `wildschwein1.txt`. Beachte, dass sich der Nordrand aufgrund der perspektivischen Darstellung unten befindet, der Südrand oben.

Zu den fünf vorgegebenen Beispieleingaben lässt sich Folgendes noch festhalten (siehe Abb. 20 und Abb. 21); die Planquadrate sind rot gefärbt, wenn dort Erde hinzugefügt wurde, und grün, wenn Erde entfernt wurde:

- `wildschwein1.txt` ist zusammen mit der Lösung in der Abb. 20 gezeigt und ist ein Brachland mit zwei spitzen Hügeln.
- `wildschwein2.txt` ist vollkommen flach (Abb. 21), daher ist natürlich die beste Lösung einfach zu sehen, nämlich ein Gerade von links nach rechts.
- `wildschwein3.txt` ist ebenfalls ein recht künstliches Brachland, bei welchem die beste Lösung einfach zu sehen ist (Abb. 21).
- `wildschwein4.txt` ist relativ flach mit zufälligen lokalen Erhöhungen und Vertiefungen (Abb. 21).
- `wildschwein5.txt` ist eine wellenförmige Landschaft (Abb. 21).

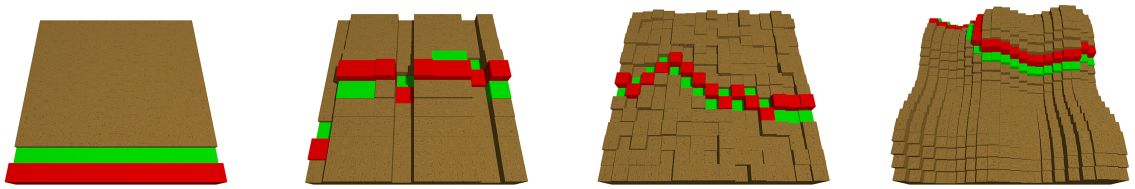


Abbildung 21: CPLEX-Lösungen für die Beispiele `wildschwein2.txt` bis `wildschwein5.txt`. Beachte, dass sich der Nordrand aufgrund der perspektivischen Darstellung unten befindet, der Südrand oben.

Die Beispiele `wildschwein2.txt` und `wildschwein3.txt` wurden mittels MILP sogar exakt gelöst. CPLEX kam bei den anderen Beispielen jedoch oft zu keinem Ende und gab jeweils die beste, bisher dahin gefundene Lösung nach Abbruch der stundenlangen Suche aus.

2 Bewertungskriterien

- In der Einsendung sollte das Problem sinnvoll und richtig definiert und modelliert werden, so dass ersichtlich wird, dass es in allen Einzelheiten verstanden wurde.
- Die Laufzeit des Verfahrens sollte in einem angemessenen Rahmen liegen, und die vorgegebenen Beispiele sollten alle schnell verarbeitet werden.
- Die angewandten Erdarbeiten müssen immer zu einer korrekten Lösung führen; das Brachland soll nach der Ausführung aller Erdarbeiten für die Wildschweine unpassierbar sein.
- Die Optimalität einer gefundenen Lösung wird nicht erwartet. Falls ein garantiert optimales Verfahren entwickelt und realisiert wurde, gibt es dafür Pluspunkte.
- Die Ergebnisse des Verfahrens sollten in den meisten Fällen gut sein und etwa die im Abschnitt 2 aufgeführten Werte erreichen; dementsprechend sollten die Kosten angewandter Erdarbeiten nicht durch offensichtliche Änderungen verbessert werden können.
- Man konnte sich detailliert Gedanken zu lokal guten Erdarbeiten machen, indem man nach der Umgebung und den möglichen Höhenunterschieden Fälle unterscheidet. Da dies sehr aufwändig und schwierig ist, wird dies bei der Bewertung berücksichtigt.
- Falls Optimalität der umgesetzten Methode behauptet wird, muss dies gut begründet werden. Ansonsten muss hervorgehoben werden, dass das Problem nicht optimal oder nur langsam gelöst wird und die Grenzen des Verfahrens aufgezeigt werden; dafür ist die Darstellung an eigenen Beispielen sinnvoll.
- Das implementierte Verfahren soll nachvollziehbar beschrieben und durch Beispiele veranschaulicht sein. Die Funktionsweise und Richtigkeit verwendeter bekannter Algorithmen muss gut dargestellt werden, so dass erkennbar ist, dass diese komplett ver-

standen wurden. Bei der Verwendung von MILP-Softwarewerkzeugen gilt das nur eingeschränkt; hier genügt ein passender Verweis.

- Alle fünf vorgegebenen Beispiele müssen bearbeitet werden und die jeweiligen Ergebnisse in der Dokumentation enthalten sein. Das vorgegebene Eingabeformat muss korrekt verarbeitet werden.
- Ein sinnvolles Format für die Ausgabe war selbst zu bestimmen. In der Aufgabenstellung wird nach den Erdarbeiten gefragt. Daher genügt es nicht, nur die modifizierten Höhen der Planquadrate auszugeben. Die Kosten für die errechneten Erdarbeiten sollten leicht ablesbar sein.
- Eine mögliche Erweiterung der Aufgabenstellung wäre zum Beispiel, die Menge an Erde, die von jedem Planquadrat abgetragen werden darf, beispielsweise durch seine Höhe zu beschränken. Denkbar ist auch, kompliziertere Kostenfunktionen für die Erdarbeiten oder eine weitere Methode zur Blockade der Wildschweine einzuführen, die sich anders auf umgebende Planquadrate auswirkt. Eine einfache Möglichkeit hierfür wären zum Beispiel Zäune, die dann keinerlei Einfluss auf die Höhe der anliegenden Planquadrate hätten.

Aufgabe 3: Quo vadis, Quax?

Diese Aufgabe ermöglicht viele methodische Herangehensweisen und Erweiterungen der Aufgabenstellung. Auf den ersten Blick in die Algorithmen-Literatur fällt auf, dass die Aufgabenstellung dem Problem des sogenannten *Motion Planning*[12] in der Robotik ähnelt, das auch als Teilgebiet der *Computational Geometry* (Algorithmischen Geometrie) bekannt ist.[13] Hierbei geht es um die Bewegung eines Roboters (statt eines Menschen) durch einen Raum oder eine Landschaft von einem Startpunkt S zu einem Zielort T der Stadt, ohne mit Hindernissen zu kollidieren. Zur Aufklärung einer unbekannten Landschaft für die autonome Navigation eines oder mehrerer Roboter (allg. *unmanned ground vehicle*s, UGVs) werden tatsächlich seit einigen Jahren auch Quadkopter (allg. *unmanned aerial vehicles*, UAVs) mit entsprechenden Sensoren eingesetzt.[14]

Zur Lösung des Problems mittels des Robotik-Ansatzes kann man ein Gitter mit einer bestimmten Rasterauflösung verwenden, in dem sich der Roboter an jedem Gitterpunkt zu benachbarten Gitterpunkten bewegen darf. Die Robotergestalt ist oft entweder nur durch einen Punkt oder durch ein Polygon im zweidimensionalen Raum repräsentiert, die Hindernisse sind meist durch Polygone umgrenzt (ggf. wird ein sogenannter Sichtbarkeitsgraph der Hindernisse verwendet[15]). Dieser gitterbasierte Ansatz erfordert eine bestimmte Rasterauflösung, wobei natürlich die Suche mit gröberen Gittern schneller ist. Bei einer nur groben Rasterauflösung besteht aber auch die Gefahr, dass dieser Ansatz keine Wege durch enge Stellen findet. Zur Bestimmung der Hindernisse könnte man zudem auf die Idee kommen, das sogenannte *Connected-Component Labeling*[16] auf die binäre Landschaft mit Planquadraten entweder für Land oder Wasser anzuwenden.

Insgesamt würde es jedoch den Umfang dieser Dokumentation sprengen, all die unterschiedlichen möglichen Ansätze aus der Robotik im Detail anzusprechen. Deshalb werden wir uns im Folgenden auf eine einzelne exemplarische Lösung beschränken, die zudem – wie die meisten Wettbewerbseinsendungen – einen anderen Lösungsansatz mittels eines Quaternärbaums ohne Anlehnung an die Robotik wählt. Ein wichtiger Gedanke zur Lösung der Aufgabe ist hierbei, dass nicht wie beim *Motion Planning* üblich der kürzeste Weg gesucht ist, sondern es im Prinzip für eine akzeptable Lösung genügt, mindestens einen Weg beliebiger Länge vom Startstandort S von Quax zur Stadt T zu finden.

3.1 Lösungsidee

Ein Quaternärbaum („Quadtree“) ist in der Informatik eine spezielle Baum-Struktur, in der jeder innere Knoten bis zu vier Kinder haben kann[17]. Das Wort „Quadtree“ leitet sich von der Zahl der Kinder eines inneren Knoten ab (quad (vier) + tree (Baum) = Quadtree). Ein

[12] Wikipedia, 2018. http://en.wikipedia.org/wiki/Motion_planning

[13] Steven S. Skiena: The Algorithm Design Manual, 1998. http://www3.cs.stonybrook.edu/~algorith/files/motion-planning.shtml

[14] Beispielartikel: M. Garzón, J. Valente, D. Zapata und A. Barrientos: An aerial–ground robotic system for navigation and obstacle mapping in large outdoor areas, *Sensors*, 2013. http://doi.org/10.3390/s130101247

[15] Wikipedia, 2018. http://en.wikipedia.org/wiki/Visibility_graph

[16] Wikipedia, 2018. http://en.wikipedia.org/wiki/Connected-component_labeling

[17] Wikipedia, 2018. http://de.wikipedia.org/wiki/Quadtree

Binärbaum ist eng verwandt mit einem Quaternärbaum, jedoch hat jeder innere Knoten eines Binärbaums nur bis zu zwei Kinder [18].

Der Quaternärbaum wird hauptsächlich zur Organisation zweidimensionaler Daten im Bereich der Computergrafik eingesetzt. Die Wurzel des Baums repräsentiert dabei eine quadratische Fläche. Diese wird rekursiv in je vier gleich große Quadranten zerlegt, bis die gewünschte Auflösung erreicht ist und die Rekursion in einem Blatt endet. Durch rekursive Anwendung dieser Zerteilung kann die vom Wurzelknoten repräsentierte Fläche beliebig fein aufgelöst werden.

Da ein Blatt unter Umständen eine verhältnismäßig große Fläche abdecken kann, ist die Datenstruktur relativ speichersparend[19] sowie besonders schnell nach einem Knoten durchsuchbar, der einen bestimmten Punkt der Landschaft beinhaltet.

3.2 Teilaufgabe (a): Wegfindung durch die Landschaft

Ein beliebiger Weg in der Landschaft wird als eine Kette direkt benachbarter Planquadrate der kleinsten Seitenlänge 10 m definiert, so dass jeweils von einem Planquadrat auf eines der maximal vier direkt benachbarten Planquadrate (im Norden, Osten, Süden und Westen) gewechselt wird. Jedes Planquadrat der Seitenlänge 20 m kann laut Aufgabenstellung einen von drei möglichen Geländewerten als Ergebnis eines Flugs annehmen: „Wasser“ (W), „Land“ (L) oder „gemischt“ (G). Zu Beginn sind alle Planquadrate, mit Ausnahme der beiden Land-Planquadrate der Seitenlänge 10 m an den Standorten S und T von Quax bzw. der Stadt, noch „unerkundet“ (U).

Als Sonderfall ist bei der Ermittlung eines tatsächlich gangbaren Weges zwischen S und T die mögliche mittige Lage zweier benachbarter Wasser-Planquadrate der Seitenlänge 10 m zu berücksichtigen, die sich aus zwei benachbarten Planquadrate der Seitenlänge 20 mit jeweils gemischtem Gelände ergeben kann (siehe Abb. 22). Zwei aneinander grenzende Wasser-Planquadrate der Gesamtlänge 20 m können von Quax nicht durchschwommen werden. Jeder gangbare Weg aus Planquadraten der Seitenlänge 10 muss daher innerhalb einer Kette überlappender Planquadrate der Seitenlänge 20 m mit dem Geländewert „L“ oder „G“ verlaufen, um auch den Sonderfall zu berücksichtigen.

Das Ergebnis einer Wegsuche zwischen S und T ist also letztlich nicht ein Pfad der Breite 10 m, sondern der Breite 20 m aus Planquadraten der Seitenlänge 20 m. Die Weglänge ist im Folgenden die Anzahl der benachbarten Planquadrate der Seitenlänge 20 m entlang des Pfades, die nirgends überlappen.

Zur Wegbestimmung von S nach T im von der Landschaft abgeleiteten Gitter von Planquadraten, der Landschaftsmatrix, wird eine verallgemeinerte Variante des Dijkstra-Algorithmus[20] für kürzeste Wege in der Ebene als heuristischer A*-Suchalgorithmus[21] verwendet, um einen

[18] Wikipedia, 2018. http://de.wikipedia.org/wiki/Binärbaum
[19] Demo zur Bildkompression mittels eines Quaternärbaums: http://www.michaelfogleman.com/static/quads/
[20] Wikipedia, 2018. http://de.wikipedia.org/wiki/Dijkstra-Algorithmus
[21] Wikipedia, 2018. http://de.wikipedia.org/wiki/A*-Algorithmus

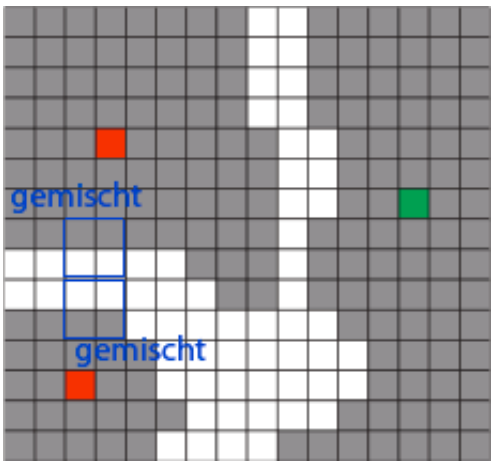


Abbildung 22: Sonderfall der mittigen Lage einer Wasserfläche, die von Quax aufgrund der Breite von 20 m nicht überquert werden kann.

(möglichst kurzen) Weg zu finden. [22] Beachte jedoch, dass es zur Erfüllung der Aufgabenstellung genügen würde, einen Weg beliebiger Länge (und nicht einen möglichst kurzen Weg aller möglichen Wege) zu finden. Beim A*-Suchalgorithmus wird die Suche durch eine Funktion gesteuert, welche die Entfernung von der aktuellen Situation zu einem Ziel abschätzt.[23]

Sie liefert ein Prioritätskriterium für die Prioritätswarteschlange („Priority Queue“) der demnächst zu besuchenden Zustände (hier: die zu besuchenden Wegpunkte). In dieser Lösung ist dieses Kriterium die Summe aus der bisherigen Wegstrecke zum jeweiligen Punkt P und der minimalen Entfernung des Wegpunktes P zum Ziel T, d.h. die L^1-Norm-Distanz (auch als „Manhattan-Distanz“ bekannt) zwischen P und T.

Die verwendete Landschaftsmatrix enthält an den jeweiligen x- und y-Koordinaten jedoch nicht den Geländewert des entsprechenden Planquadrats der kleinsten Seitenlänge 10 m, sondern des Planquadrats der Seitenlänge 20 m. Jedes Planquadrat der Seitenlänge 20 m besteht weiter aus vier Unterplanquadraten („NW“, „NO“, „SO“, „SW“) der kleinsten Seitenlänge 10 m, so dass die Koordinaten des jeweiligen NW-Unterplanquadrats das dazu gehörige Planquadrat bestimmen. In dieser Landschaftsmatrix läuft die A*-Wegsuche von S aus ab, nachdem die Matrix mit den Ergebnissen bereits durchgeführter Flüge gefüllt wurde und noch unerkundete Planquadrate (also mit noch unbekannten Geländewerten) als undurchquerbar angenommen werden.

Hierbei kann entweder mindestens ein gangbarer Weg beliebiger Länge von S nach T gefunden werden oder es ist im Weiteren erst noch zu klären, ob es tatsächlich keinen Weg zwischen S und T gibt – oder die Landschaft erst noch weiter durch zusätzliche Flüge erkundet werden muss, um eine Entscheidung über die Existenz eines möglichen Weges treffen zu können. Zur Klärung der letzteren Frage wird dieselbe, zuvor erfolglose A*-Wegsuche von S aus nochmals auf die bereits vorhandene Landschaftsmatrix angewandt, dieses Mal jedoch unter der Hypothese, dass alle noch unerkundeten Planquadrate eine Überquerung durch Quax erlauben, obgleich ihre konkreten Geländewerte noch unbekannt sind. Hierfür wird allen noch

[22] Wikipedia, 2018. GIF-Animation des A*-Suchalgorithmus: http://de.wikipedia.org/wiki/A*-Algorithmus#/media/File:Astar_progress_animation.gif

[23] Visuelle A*-Wegsuche mit Kostenfunktion: http://www.geosimulation.de/methoden/a_stern_algorithmus.htm

unerkundeten Planquadraten der Geländewert „L“ spekulativ in der Landschaftsmatrix zugewiesen. Nur wenn diese A*-Wegsuche in der hypothetischen Landschaft erfolgreich einen möglichen Weg zwischen S und T findet, kann sich eine weitere Erkundung der Landschaft mit Flügen lohnen. Ansonsten ist definitiv klar geworden, dass es keinen Weg zwischen S und T geben kann und sich zusätzliche Flüge nicht lohnen werden. (Natürlich könnten beide A*-Wegsuchen auch miteinander integriert werden, so dass letztendlich nicht zwei Durchläufe, sondern nur ein Durchlauf benötigt wird. Allerdings ist die programmiertechnische Umsetzung dann komplizierter.)

Im Übrigen könnte man die Wegsuche sowohl vom Startpunkt als auch vom Endpunkt beginnend gleichzeitig durchführen (sogenannte birektionale Suche[24]); sobald sich die Wege kreuzen, hätte man eine Lösung des Problems gefunden. Auch wenn dies die Komplexität der Programmentwicklung erhöhen würde, wäre die beidseitige Suche für die üblichen Computerprozessoren (CPUs) mit mehreren Kernen (Cores) mittels Nebenläufigkeit (Threads) durchaus umsetzbar.

3.3 Teilaufgabe (b): Strategie zur Minimierung der Fluganzahl

Diese Teilaufgabe besteht vor allem im Finden einer effizienten Strategie, den beschädigten Quadcopter möglichst wenige Flüge zur Erkundung der Landschaft machen zu lassen. Im Folgenden konzentrieren wir uns auf den (Bereichs-)quaternärbaum, der eine Aufteilung der zweidimensionalen Landschaft darstellt und sie rekursiv jeweils in vier gleich große Unterquadrate (Bereiche) zerlegt. Jeder Knoten in dem Baum hat entweder genau vier Kinder oder gar keine (Blattknoten). (Der Bereichsquaternärbaum ist somit auch eine Art von Trie.[25])

Ein Bereichsquaternärbaum mit einer Tiefe von n kann benutzt werden, um eine Landschaft aus $2n \times 2n$ Planquadraten darzustellen, wobei jedes Planquadrat der Seitenlänge 20 m die drei möglichen Geländewerte als Ergebnis eines Flugs annehmen kann: „Wasser“ (W), „Land“ (L) oder „Gemischt“ (G). Der Wurzelknoten repräsentiert die gesamte Landschaft. Sofern in dem einem Knoten zugeordneten Unterquadrat nicht die komplette Landschaft einheitlich mit Wasser oder Land bedeckt ist, wird dieser Bereich weiter rekursiv unterteilt (siehe Abb. 23). Jeder Knoten des Bereichsquaternärbaum repräsentiert somit das Sensorergebnis eines Fluges, wobei er die x- und y-Koordinaten der jeweils oberen linken Ecke des Fluges bzw. des betreffenden Quadrats enthält. Der positive x-Achsenabschnitt verläuft von Westen nach Osten (von links nach rechts) und der positive y-Achsenabschnitt von Norden nach Süden (von oben nach unten).

Die entsprechend wichtigste Methode zur Lösung dieser Teilaufgabe nimmt die ganzzahligen x- und y-Koordinaten eines auf dem Weg zum Ziel anvisierten Punktes der Landschaft als Eingabe entgegen und unternimmt solange Flüge dorthin, bis der Geländewert des entsprechenden Planquadrats ermittelt ist. Als Ausgabe liefert die Methode die Ergebnisse der durchgeführten Flüge in einer Liste, die wiederum zur jeweiligen Aktualisierung der zweidimensionalen Landschaftsmatrix dient.

[24] Wikipedia, 2018. http://en.wikipedia.org/wiki/Bidirectional_search
[25] Wikipedia, 2018. http://de.wikipedia.org/wiki/Trie

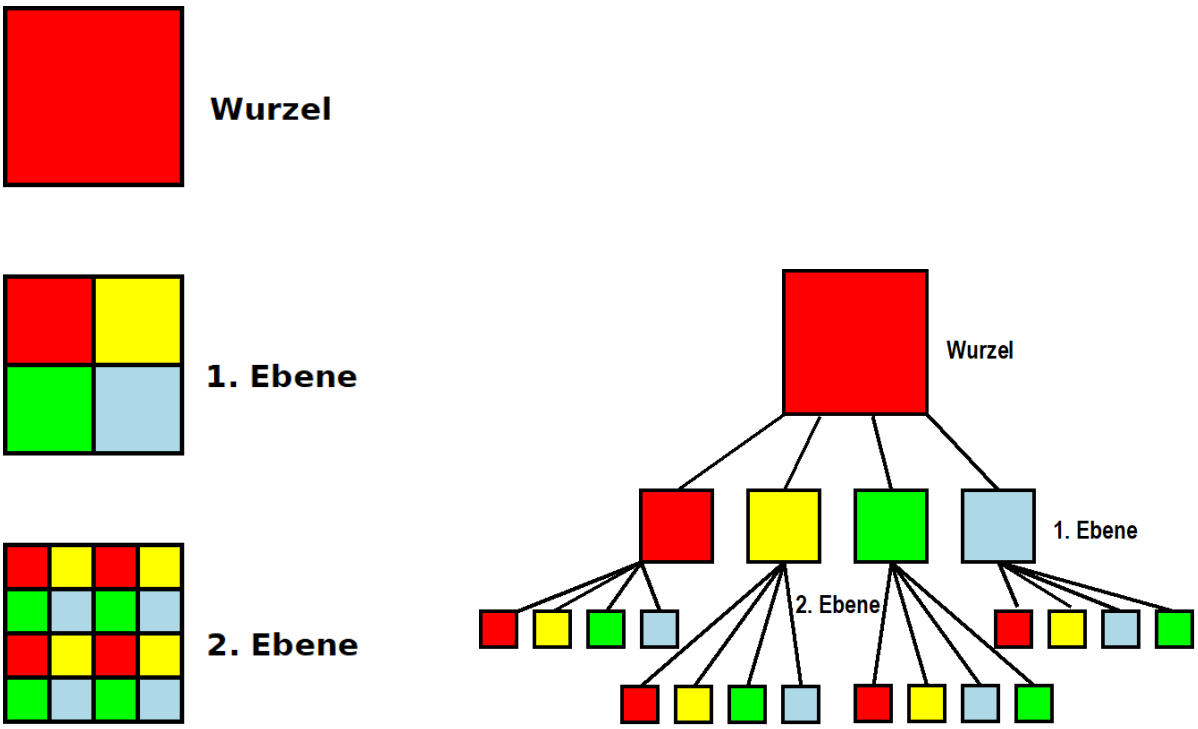


Abbildung 23: Aufteilung der Quadrate des Quaternärbaums in jeweils vier Unterquadrate.

In der Methode wird wie folgt mittels des Bereichsquaternärbaums vorgegangen: Falls der gewünschte Punkt in dem zu einem Knoten gehörigen Gelände liegt, ist man fertig und braucht keinen Flug durchführen. Andernfalls sucht man rekursiv mit der Wurzel des Baums beginnend den nächsten Knoten auf, in dessen Quadrat der Punkt liegt[26]. Die Rekursion ist beendet, wenn der Geländewert entweder eindeutig ermittelt ist („W“ oder „L“) oder das dem Knoten entsprechende Quadrat nur mehr eine Seitenlänge von 20 m hat, d.h. ein Blattknoten erreicht wurde. Die Anzahl der notwendigen Flüge ergibt sich somit aus der Rekursionstiefe (Anzahl von Rekursionen).

Die A*-Wegsuche mittels einer Prioritätswarteschlange hat bekanntlich eine asymptotische Laufzeitkomplexität von $\mathcal{O}(n^2 \log n)$ (unter der Annahme einer relativ quadratischen Landschaftsgröße mit $n \approx m$).[27] Hinzu kommt die Verwendung des Quaternärbaums mit $\mathcal{O}(n^2)$, was jedoch an der durch den A*-Algorithmus bestimmten Gesamtkomplexität theoretisch nichts ändert.

3.4 Teilaufgabe (c): Implementierung der Strategie

Die in Teilaufgabe (b) beschriebene Strategie wurde so umgesetzt und ein Programm entwickelt, das die Landschaft als Pixelbild einliest. Rechteckige Landschaftsformen, die nicht quadratisch sind, werden ganz zu Beginn entsprechend durch Wasserflächen am Rand zu quadratischen Landschaften ergänzt. Die dann quadratischen Landschaften werden gegebenen-

[26] Interaktive Demos zur Visualisierung eines Quadtree-Search sind http://jimkang.com/quadtreevis/ und http://bl.ocks.org/mbostock/4343214

[27] Wikipedia, 2018. http://de.wikipedia.org/wiki/A*-Algorithmus

Beispiel	Ausgabenummer	Start	Ziel	Fluganzahl	Pfadlänge
quax1	1	(510; 914)	(226; 148)	9	524
quax1	2	(1454; 948)	(226; 148)	1850	1177
quax1	3	(1582; 328)	(226; 148)	281	0
quax2	1	(614; 926)	(580; 888)	13057	2404
quax2	2	(844; 66)	(580; 888)	1183	866
quax2	3	(844; 66)	(580; 888)	1183	866
quax2	4	(844; 66)	(580; 888)	1183	866
quax2	5	(844; 66)	(580; 888)	1183	866
quax2	6	(1384; 726)	(580; 888)	12541	2578
quax2	7	(1514; 582)	(580; 888)	159	0
quax3	1	(16; 58)	(1018; 808)	2091	1051
quax3	2	(22; 566)	(1018; 808)	9392	892
quax3	3	(390; 1124)	(1018; 808)	10266	1309
quax3	4	(726; 120)	(1018; 808)	390	525
quax3	5	(1070; 828)	(1018; 808)	230	0
quax3	6	(1584; 1098)	(1018; 808)	1640	665
quax3	7	(1802; 74)	(1018; 808)	2212	1078
quax3	8	(1898; 590)	(1018; 808)	1952	834

Tabelle 3: Statistik zu den drei für die Aufgabe vorgegebenen Beispielen.

falls nochmals mittels zusätzlicher Wasserflächen am Rand erweitert, so dass die Seitenlänge der gesamten Landschaft eine Zweierpotenz ist; so lässt sich die Größe der jeweiligen Unterquadrate einfach durch Halbierung der Seitenlänge erreichen. Außerdem könnte man eine Flugverbotszone für die ergänzten Wasserflächen festlegen; falls der anvisierte Punkt darin liegen würde, würde in diesen Wasserflächen nicht unnütze Zeit mit Landsuche vergeudet werden.

Die für die Wege gefundenen Lösungen werden als animierte GIF-Bilder mit transparenten Quadraten für die Flüge abgespeichert sowie Statistikdaten über die Flüge und Weglängen als Text ausgegeben.[28]

In der Tabelle 3 sind die Anzahl der Flüge und die jeweilige Weglänge unserer Lösungen für die drei Beispiel-Landschaften mit verschiedenen vorgegebenen Startpunkten enthalten.

3.5 Teilaufgabe (d): Diskussion der Wasserverteilungen

Bei einer eher unnatürlichen Landschaften mit zahlreichen, zufällig verteilten, kleinen Wasserlöchern (bzw. wie bei einem weit verzweigten Irrgarten/Labyrinth[29]) statt weniger, großflächiger Flüsse und Seen werden viele Flüge benötigt werden, um die Landschaft aufgrund der zufälligen Verteilung der vielen Wasserlöcher im Detail zu erkunden. Hierbei wird ein

[28] Siehe https://bwinf.de/bundeswettbewerb/der-aktuelle-wettbewerb/2-runde/material-362/

[29] Beispiel eines verzweigten Irrgartens: http://upload.wikimedia.org/wikipedia/commons/d/df/ZweiWegeIrrgarten.png

relativ umfangreicher Quaternärbaum erzeugt. Auch bei einer schachbrettartigen Verteilung von zahlreichen Wasserlöchern und Land oder bei sich abwechselnden langen, engen Streifen von Wasser und Land mit feinen Strukturen sind relativ viele Flüge zu erwarten, da die Landschaft für die Wegsuche mit hoher Auflösung aufgeklärt werden muss. Allgemein ist festzuhalten, dass das vorgestellte Verfahren vor allem dann nur wenige Flüge benötigt, wenn es große Land- und Wasserflächen gibt und nur wenige, einfach strukturierte Hindernisse für die Wegsuche; dann besteht der Quaternärbaum schließlich nur aus wenigen Knoten.

3.6 Anhang: Generierung einer zufälligen Landschaft

Eine zufällige Landschaft lässt sich leicht um ein beliebiges Startquadrat der Seitenlänge 20 m generieren mit initial zufälligen Geländewerte für jedes der vier Planquadrate der Seitenlänge 10 m. Hierfür werden spiralförmig im (oder gegen) den Uhrzeigersinn die nächstbenachbarten Quadrate um das Startquadrat herum ergänzt, zunächst sind dies acht zu ergänzende Randquadrate. Hierbei wird dem jeweiligen neuen Randquadrat der Seitenlänge 10 m mit einer gewissen Wahrscheinlichkeit w derselbe Geländewert Land (bzw. $1-w$ für Wasser) zugewiesen wie dem jeweils innen liegenden Quadrat, um größere zusammenhängende Landschaftsflächen mit demselben Geländewert zu erreichen.

Dem verbleibenden vier neuen äußeren Quadrate, die jeweils diagonal in einer der vier Ecken liegen, wird wie folgt ein Geländewert zugewiesen: Wenn die beiden benachbarten neuen Quadrate, die beide nicht in Ecken liegen, Land oder Wasser beinhalten, dann ist das jeweilige neue Eckquadrat mit einer Wahrscheinlichkeit w wieder Land (bzw. $1-w$ für Wasser). Wenn die beiden benachbarten Quadrate nicht denselben Geländewert haben, dann ist das Eckquadrat mit der Wahrscheinlichkeit p Land (bzw. $1-p$ Wasser).

3.7 Bewertungshinweise

- In der Einsendung sollte das Problem sinnvoll und richtig definiert und modelliert werden, so dass ersichtlich wird, dass es in allen Einzelheiten verstanden wurde. Ein gangbarer Weg für Quax zur Stadt muss sinnvoll definiert sein.
- Die ermittelten Wege müssen immer eine korrekte Lösung darstellen. Falls es einen Weg gibt, soll auch einer gefunden werden.
- Der Sonderfall des mittigen Wassers auf einem Weg sollte angesprochen und behandelt werden.
- Die Erkundung und Wegbestimmung sollten nicht unnötig aufwändig sein und eine akzeptable Laufzeit haben. Die vorgegebenen Beispiele sollten rasch innerhalb weniger Sekunden bzw. Minuten verarbeitet werden können.
- Sollte der für Teilaufgabe (a) erläuterte zweite Durchlauf einer Wegsuche nach erfolgloser erster Wegsuche nicht in der Lösung der Aufgabe enthalten sein, gibt es hierfür kaum Minuspunkte, da dies für den Rest der Aufgabe nicht relevant war.

- Die in Teilaufgabe (b) beschriebene und in (c) realisierte Strategie sollte sinnvoll sein. Die Ergebnisse des entsprechenden Verfahrens sollten in den meisten Fällen eine akzeptable Fluganzahl beinhalten und sich an den in der Tabelle 3 für die Beispiele enthaltenen Anzahlen orientieren.
- Teilaufgabe (d) sollte sinnvoll und nicht zu knapp beantwortet werden.
- Vertiefte Wahrscheinlichkeitsüberlegungen und konkrete Wahrscheinlichkeitsberechnungen werden belohnt, z.B. dass höhere Flüge mit rasch ansteigender höherer Wahrscheinlichkeit meist nur mehr einen gemischten Geländewert liefern.
- Die Optimalität der Lösung wird weder in der Anzahl der Flüge noch bei der Weglänge erwartet. Es sollte hervorgehoben werden, dass das Problem nicht optimal gelöst wird, und die Grenzen des Verfahrens sollten aufgezeigt werden.
- Die Funktionsweise und Richtigkeit der verwendeten Algorithmen muss gut erklärt werden, sodass erkennbar ist, dass diese komplett verstanden wurden. Das vorgegebene Eingabeformat muss korrekt verarbeitet werden.
- Für die Implementierung des Quaternärbaums und der Prioritätswarteschlange darf auf bereits vorhandenen Quellcode aus externen Quellen zurückgegriffen werden. Allerdings sollte klar dokumentiert sein, warum der externe Quellcode seinen Zweck korrekt erfüllt, da es vielerlei Implementierungen eines Quaternärbaums mit teils unterschiedlicher Funktionsweise im Detail je nach Anwendungsgebiet gibt.
- Alle vorgegebenen Beispiele müssen verarbeitet werden und die jeweiligen Ergebnisse in der Dokumentation enthalten sein. Falls einzelne der in den drei Beispieldaten enthaltenen Startpunkte übersehen werden, ist das kein großes Problem.
- Zum Aufzeigen korrekter Implementierungen und Grenzen des vorgestellten Algorithmus ist die Darstellung von Ergebnissen anhand weiterer eigener Beispiele sinnvoll.
- Eine sinnvolle Ausgabe für Teilaufgabe (c) war selbst zu bestimmen, aber zumindest sind die Anforderungen der Aufgabenstellung zu erfüllen: Es soll eine Darstellung der Landschaft ausgegeben werden, in denen die getesteten Gebiete mit der jeweiligen Ergebnissen eingetragen sind und ein Weg erkennbar ist, falls vorhanden. Besonders gelungene Ausgaben werden belohnt (z.B. animierte GIF-Bilder). Die Anzahl der Flüge und die jeweilige Rechenzeit sollte ebenfalls ausgegeben werden, schön wäre auch die Angabe der Weglänge zur Stadt.
- Es sind viele Erweiterungen der Aufgabenstellung möglich, insbesondere falls über die Landschaft weitere Informationen vorliegen. Mit Höhenangaben für die Landflächen und einer maximalen Flughöhe des Quadkopters kann die Landschaftserkennung erschwert werden, ebenso durch weitere Hindernisse.