

GYMNASIUM OTTOBRUNN

Oberstufenjahrgang 2017/19

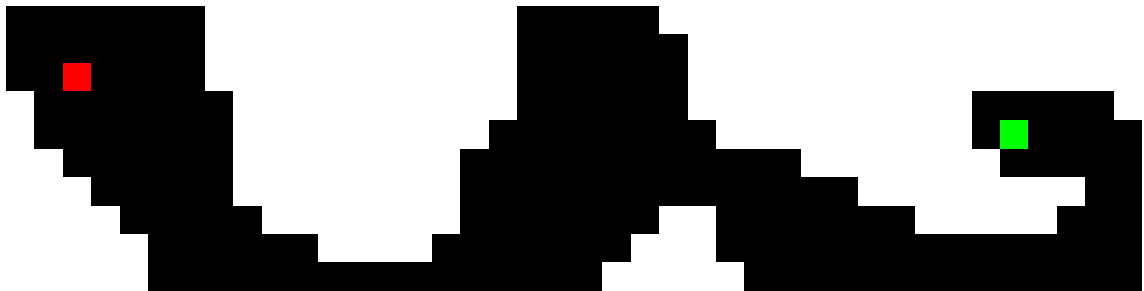
Seminarfach Softwareentwicklung

Seminararbeit

36. Bundeswettbewerb Informatik

Runde 2

Aufgabe 1 und 3



Verfasser: Jonas Fritsch

Seminarleiter: StD Peter Brichzin

Bewertung: Punkte

Unterschrift des Seminarleiters:

INHALTSVERZEICHNIS

1	EINLEITUNG	4
2	AUFGABE 1 – „Die Kunst der Fuge“	5
2.1	Aufgabenstellung	5
2.2	Auslegung der Aufgabe	5
2.3	Lösungsidee	6
2.3.1	Verstehen des Problems	6
2.3.2	Der Algorithmus	13
2.4	Implementierung	21
2.5	Optimierungsmöglichkeiten	26
2.6	Beispiele	26
3	AUFGABE 3 – „Quo vadis, Quax?“	30
3.1	Aufgabenstellung	30
3.2	Auslegung der Aufgabe	32
3.3	Lösungsidee	34
3.4	Teilaufgabe (a)	38
3.5	Teilaufgabe (b)	42
3.6	Teilaufgabe (c)	44
3.6.1	Implementierung	44
3.6.2	Optimierungsmöglichkeiten	50
3.6.3	Beispiele	51
3.7	Teilaufgabe (d)	51
4	FAZIT	53
5	LITERATURVERZEICHNIS	54
6	ABBILDUNGSVERZEICHNIS	55
7	CD-VERZEICHNIS	56
8	ANHANG	57
8.1	Installationshinweise	57

1 EINLEITUNG

Der aus drei Runden bestehende Bundeswettbewerb Informatik (BwInf) ist einer von vier bundesweiten Informatikwettbewerben. Mit dem 36. BwInf 2017 gab es für die erste Runde mit **1463** Teilnehmern eine so hohe Teilnahme wie seit 1993 nicht mehr [1]. Von den ursprünglich 1463 Teilnehmern der ersten Runde haben es **828** Teilnehmende geschafft, sich für die zweite Runde zu qualifizieren [2]. Von diesen haben jedoch nur noch **154** Teilnehmer der zweiten Runde Lösungen für zwei Aufgaben ihrer Wahl eingereicht. Dies lag nicht zuletzt an der stark gestiegenen Komplexität der Aufgaben gegenüber der ersten Runde. Zu der letzten dritten Runde des Bundeswettbewerbs Informatik wurden schlussendlich nur noch **25** Teilnehmer eingeladen. Das waren im Falle des BwInfs von 2017 nur rund **1,7%** der ursprünglichen Teilnehmer.

Im Folgenden wird eine eingereichte Lösung zu der ersten und dritten Aufgabe der zweiten Runde des 36. Bundeswettbewerbs Informatik erläutert.

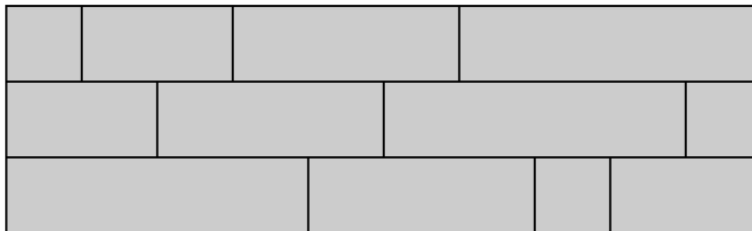
2 AUFGABE 1 – „Die Kunst der Fuge“

2.1 Aufgabenstellung

Ilona besitzt einen riesigen Haufen Holzklötzchen: Diese haben alle dieselbe Höhe und Tiefe, aber verschiedene Längen.

Ilona möchte eine Mauer bauen. Jede Reihe der Mauer soll aus n Klötzchen bestehen, die die Längen 1 bis n haben und lückenlos aneinanderliegen. Die Stellen zwischen den Klötzchen heißen Fugen. Ilona möchte, dass in der fertigen Mauer niemals zwei Fugen übereinander liegen, selbst wenn sich mehrere Reihen dazwischen befinden. Außerdem soll ihre Mauer möglichst hoch sein.

Für $n = 4$ gelingt es ihr recht schnell, eine Mauer mit drei Reihen zu bauen:



Aufgabe

Hilf Ilona, indem du ein Programm schreibst, das nach Eingabe von n eine nach ihren Vorgaben konstruierte, möglichst hohe Mauer ausgibt. Für $n = 10$ sollte dein Programm eine Mauer der Höhe 6 ausgeben können. Wie hoch werden die Mauern deines Programms für größere n ?

2.2 Auslegung der Aufgabe

Die grundlegende Aufgabe ist es, eine Mauer mit möglichst vielen Reihen, die aus verschieden länglichen Klötzchen bestehen, zu bauen.

Die Variable n ($n \in \mathbb{N} \setminus \{0\}$) definiert dabei die Längen und auch die Gesamtanzahl der Klötzchen pro Reihe. Gesamtanzahl heißt, dass bei zum Beispiel $n = 5$ jede Reihe aus 5 einzelnen Klötzchen besteht. Jede Reihe setzt sich dann aus einem 1er-, einem 2er-, einem 3er-, einem 4er- und einem 5er-Klötzchen zusammen.

Der Trick bei der Aufgabe ist nun, dass niemals zwei Fugen **übereinanderliegen dürfen**. Eine Fuge ist die Lücke zwischen zwei Klötzchen (siehe Abbildung 1).

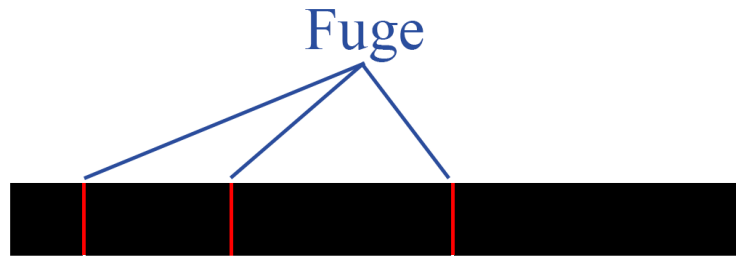


Abbildung 1: Darstellung der 3 Fugen einer Reihe mit 4 Klötzchen

Als Hilfestellung wurde außerdem erwähnt, dass eine Mauer für $n = 10$ eine maximale Höhe von 6 hat.

2.3 Lösungsidee

2.3.1 Verstehen des Problems

Um die Problemstellung der Aufgabe genauer zu verstehen, bietet es sich erstmal an, Mauern für kleinere n auf Papier zu zeichnen. Beginnt man mit $n = 1$, fällt auf, dass ganz gleich, wie viele Reihen übereinanderliegen, es nie zu einer Fugenüberlappung kommen kann, da eine Reihe mit einem einzelnen Klötzchen keine Fuge besitzt. Für $n = 1$ kann also theoretisch eine **unendlich** hohe Mauer gebaut werden.

Für $n = 2$ ist es auch noch recht einfach, eine Mauer zu bauen. Denn es gibt pro Reihe nur 2 mögliche Reihenfolgen von Klötzchen. Entweder kommt zuerst das 1er- oder das 2er-Klötzchen. Eine der zwei möglichen Lösungen ist in Abbildung 2 zu sehen. Es gibt keine Möglichkeit, noch ein Klötzchen in einer neuen dritten Reihe zu platzieren, ohne eine Fuge doppelt zu besetzen.

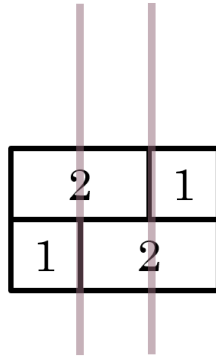


Abbildung 2: Eine der zwei möglichen Lösungen für $n = 2$. Die vertikalen Striche zeigen an, dass die jeweilige Fuge bereits besetzt ist.

Auch für $n = 3$ gibt es eine Mauer mit einer maximalen Höhe von 2 (siehe Abbildung 3). Hier wird es jedoch schon komplexer, da es nun für eine Reihe nicht mehr $2! = 2 \times 1 = 2$, sondern $3! = 3 \times 2 \times 1 = 6$ mögliche Klötzchen-Reihenfolgen gibt. Außerdem tritt bei $N = 3$ das erste Mal auf, dass eine **mögliche Fugenstelle nicht besetzt wird**.

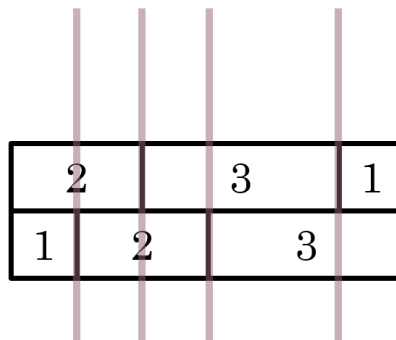


Abbildung 3: Eine mögliche Lösung für $n = 3$. Zu beachten ist, dass in der Mauer eine **mögliche** Fugenstelle, dort wo es eine größere Lücke zwischen den zwei letzten besetzten Fugenstellen gibt, **unbesetzt bleibt**.

Geht man noch weiter und bildet eine Mauer für $n = 4$ ($4! = 24$ mögliche Klötzchen Anordnungen pro Reihe), so kann man erstmals eine Mauer der Höhe 3 bilden (siehe Abbildung 4). Hier ist auch wieder jede einzelne Fugenstelle besetzt.

	4		3		1		2
2		3			4		1
1	2		3			4	

Abbildung 4: Eine mögliche Lösung für $n = 4$

Die Komplexität der Aufgabe steigt mit höheren n stark an. Während man bei $n = 2$ $2!$ Möglichkeiten pro Reihe und 2 Reihen hatte ($2! \times 2 = 4$ unterschiedliche Mauern), gibt es bei $n = 4$ bereits $4! \times 3 = 72$ verschiedene Mauern. Allgemein gibt es also für eine Mauer $n! \times h$ Lösungen, wobei h die Anzahl der übereinanderliegenden Reihen ist.

Es ist logisch, dass zwischen der Anzahl der maximal möglichen Reihen h und der Anzahl der Klötzchen pro Reihe n eine Abhängigkeit existiert.

Allgemein kann man sagen, dass die **maximale Mauerhöhe** dann erreicht ist, wenn **nicht** mehr **genügend Fugenstellen in der Mauer frei sind**, um eine **weitere Reihe zu bilden**.

Somit lässt sich die Formel

$$h = \frac{f_{Mauer}}{f_{Reihe}}$$

ableiten, wobei f_{Mauer} die maximale Anzahl an Fugen in der Mauer und f_{Reihe} die Anzahl der Fugen, die eine Reihe besetzt, beschreibt.

Um nun auf f_{Mauer} schließen zu können, braucht man nur die Länge der Mauer. Da jede Reihe der Mauer gleich lang ist, ist die Länge der Mauer gleich der Länge einer Reihe. Die Länge einer Reihe definiert sich wiederum durch die Menge und Breite ihrer Klötzchen. Das heißt also, die Länge einer Reihe/der Mauer ist nach der **Gaußschen Summenformel**

$$1 + 2 \dots + n = \sum_{k=1}^n k = \frac{n(n+1)}{2}$$

Um nun aber von der Länge der Mauer auf f_{Mauer} zu schließen, muss man noch 1 von der Länge der Mauer abziehen. Dies wird offensichtlich, wenn man zu einer Mauer eine Reihe aus lauter 1er-Klötzchen bildet (siehe Abbildung 5). Jedes 1er-Klötzchen, bis auf das Letzte, besetzt eine Fuge. Da das Ende des letzten Klötzchens nicht als Fuge gilt, wird von der Länge der Reihe noch 1 abgezogen.

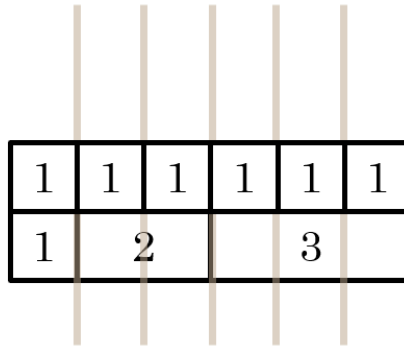


Abbildung 5: Eine Mauer der Länge 6 besitzt 5 mögliche Fugenstellen.

Ähnlich lässt sich auch f_{Reihe} berechnen. Um wissen zu wollen, wie viele freie Fugen eine Reihe mit n Klötzchen besetzen wird, zieht man einfach wieder 1 von der Anzahl der Klötzchen pro Reihe (n) ab (siehe Abbildung 6).

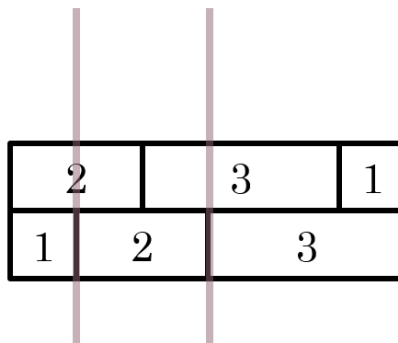


Abbildung 6: Eine Reihe bei $n = 3$ besetzt $n - 1 = 2$ freie Fugen

Daraus ergibt sich die fertige Formel für die Anzahl der Reihen h in einer maximal hohen Mauer in Abhängigkeit von n .

$$h = \frac{\frac{n \times (n+1)}{2} - 1}{n - 1} = \frac{n}{2} + 1$$

Der Graph der daraus folgenden Funktion $h(n) = \frac{n}{2} + 1$ verläuft dabei linear (siehe Abbildung 7).

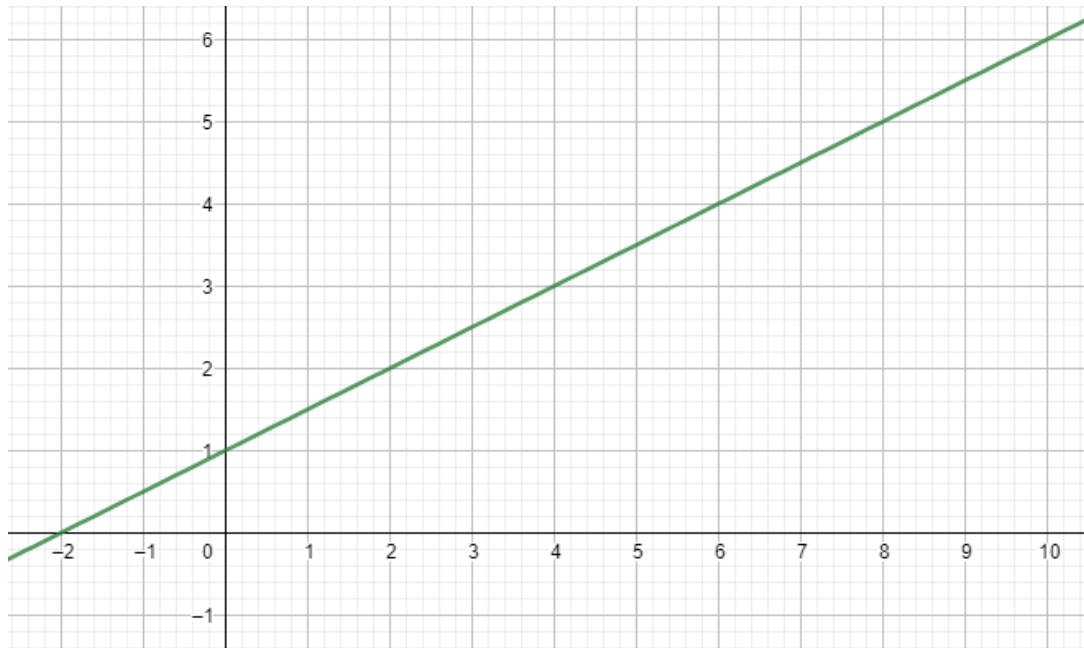


Abbildung 7: Graph der linearen Funktion $h(n) = \frac{n}{2} + 1$

Setzt man nun für $n = 10$ ein, ergibt sich der Funktionswert 6, der mit der in der Aufgabenstellung angegebenen Höhe für eine Mauer von $n = 10$ übereinstimmt. Testet man allerdings die maximale Mauerhöhe für zum Beispiel $n = 3$, so ergibt sich als Wert 2,5. Dies macht allerdings keinen Sinn, da die fertige Mauer nur aus vollen Reihen bestehen muss und keine halbfertige Mauer besitzen kann.

Jedes ungerade n zieht einen nicht-ganzzahligen Funktionswert h nach sich. Der Zusammenhang ist dabei, dass wie beim vorherigen Beispiel $n = 3$ (siehe Abbildung 3) eine Fuge unbesetzt bleibt. Diese Fuge kann zwar mit einem weiteren Klötzchen besetzt werden, die Reihe könnte aber nicht mehr vollständig gebaut werden. Da die Mauerhöhe also eine natürliche Zahl sein muss, wird der Funktionsterm noch mit Abrundungsklammern umgeben:

$$h(n) = \left\lfloor \frac{n}{2} + 1 \right\rfloor$$

Der daraus resultierende Graph ist in Abbildung 8 zu sehen.

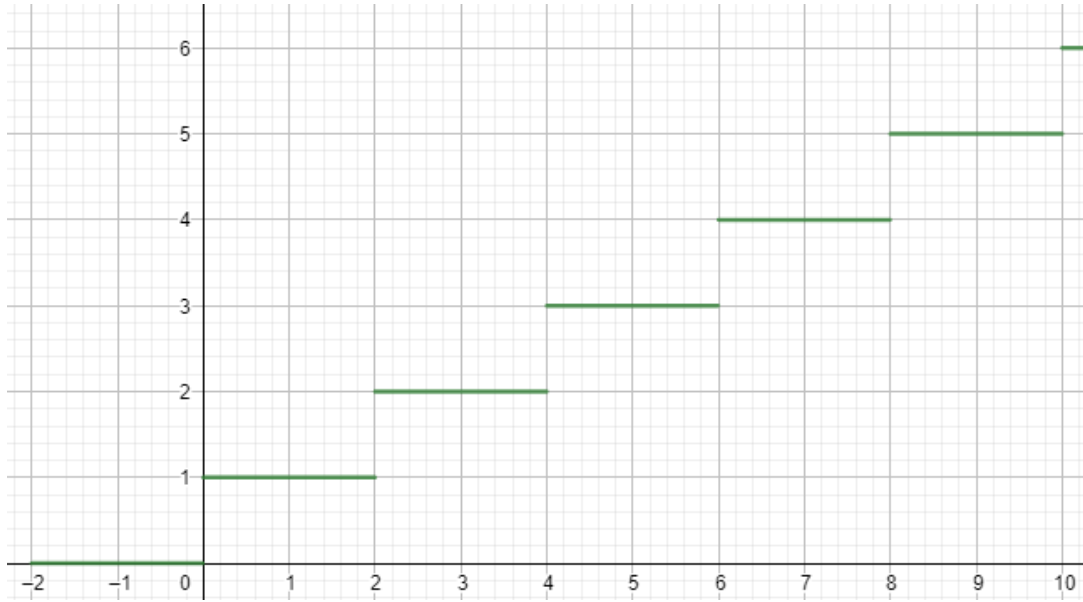


Abbildung 8: Graph der Funktion $h(n) = \lfloor \frac{n}{2} \rfloor + 1$

Somit gibt die Funktion $h(n)$ mit der Definitionsmenge $\mathbb{D}_h = \mathbb{N} \setminus \{0; 1\}$ und der Wertemenge $\mathbb{W}_h = \mathbb{N} \setminus \{0\}$ die jeweilige maximale Mauerhöhe (h) in Abhängigkeit von n an. Wobei in der Definitionsmenge auch 1 ausgeschlossen ist, weil für $n = 1$ aus logischer Sicht eine unendlich hohe Mauer und nicht nur eine Mauer der Höhe 1 gebildet werden kann.

Die naivste Lösung für die Aufgabe bestünde darin, per Brute-Force so viele Mauer-Permutationen durchzugehen, bis man eine fertige Mauer der maximalen Höhe ohne Fugenüberlappung erreicht hat.

Für die in der Aufgabenstellung festgelegte Mindestanforderung $n = 10$, gäbe es bereits

$$f(10) = 10!^6 \approx 2,28e+39$$

unterschiedliche Mauerkombinationen bei der maximalen Mauerhöhe von 6. Der Graph der allgemeinen Formel

$$f(n) = n!^{\lfloor \frac{n}{2} \rfloor + 1}$$

ist in Abbildung 9 zu sehen.

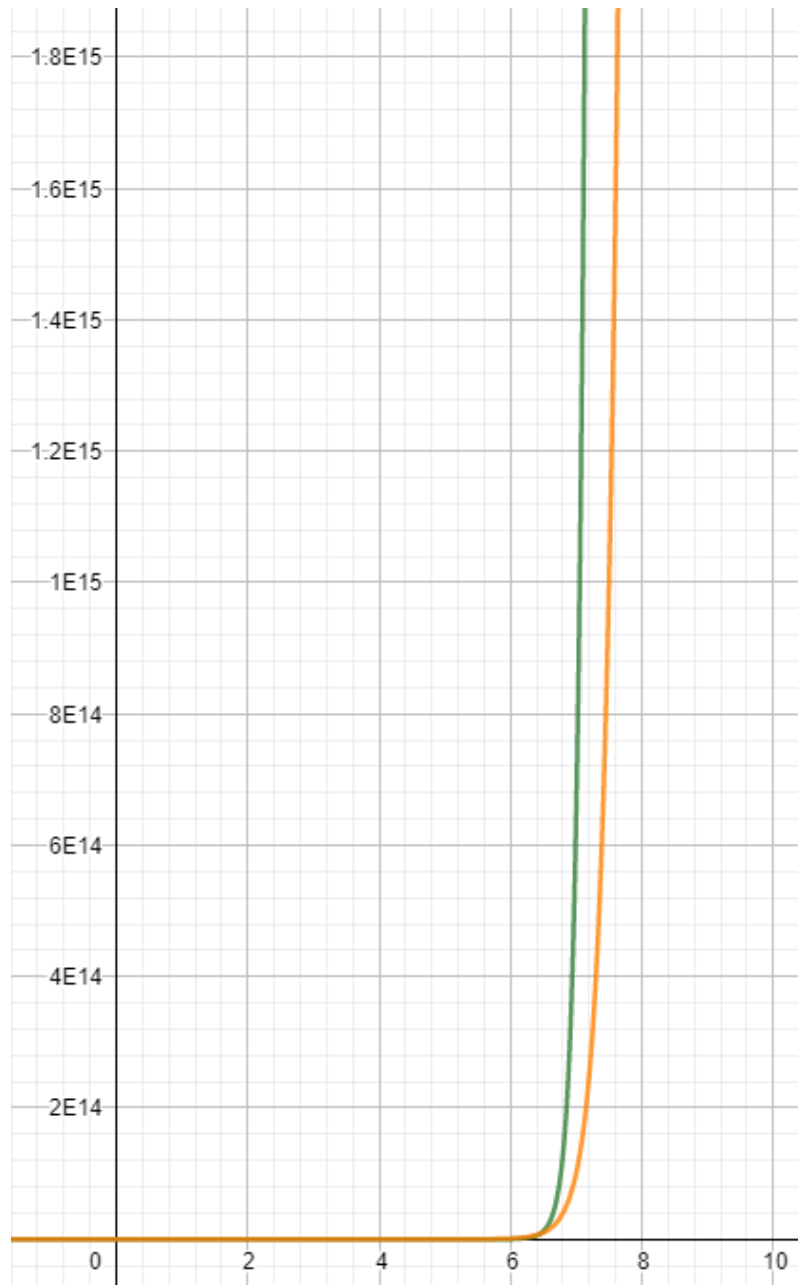


Abbildung 9: Der grüne Graph zeigt die extreme Steigung der Funktion $f(n) = n!^{\lfloor \frac{n}{2} + 1 \rfloor}$. Zum Vergleich der orange farbige Graph der Funktion $g(x) = 100^x$.

Die Anzahl der verschiedenen möglichen Mauern für n steigt somit mehr als **exponentiell**.

Nun ist hierbei allerdings zu beachten, dass es durchaus mehrere mögliche unterschiedliche Lösungen geben kann. Unter anderem werden bei der Gesamtmenge unterschiedlicher Mauern auch Mauern unterschieden, die die gleichen Reihen in unterschiedlicher Reihenfolge besitzen (siehe Abbildung 10). Dies führt zu einer Vielzahl möglicher Lösungen.

2	3	1
1	2	3

1	2	3
2	3	1

Abbildung 10: Obwohl beide Mauern aus denselben zwei Reihen bestehen, werden sie als unterschiedliche Mauern angesehen.

Weiterhin ist zu beachten, dass es für Mauern mit ungeraden n mehr unterschiedliche Lösungen gibt als für die wieder darauffolgende Mauer mit geraden n .

Dies liegt eben daran, dass bei ungeraden n immer mindestens eine Fugenstelle frei bleibt, was wiederum zu mehr richtigen Mauervariationen führt.

Trotzdem würde das Finden einer Lösung für $n = 10$, selbst wenn ein Computer nur **1 Millisekunde** für die Permutation einer **gesamten Mauer** benötigen würde, die maximal erwartete Algorithmuslaufzeit einer BwInf-Lösung bei Weitem überschreiten.

Damit ist eine naive Brute-Force Methode als Lösung ausgeschlossen und es muss nach einem effizienteren Algorithmus gesucht werden.

2.3.2 Der Algorithmus

Grundlegend gibt es zwei unterschiedliche Wege, eine Mauer wie in der Aufgabenstellung zu bauen.

Der offensichtlichere ist die Bauart „**Von unten nach oben**“. Bei dieser Bauart setzt man zuerst die Klötzchen zu einer Reihe zusammen. Hat man eine Reihe fertig, beginnt man mit der zweiten Reihe und so weiter (siehe Abbildung 11). Man setzt Reihe auf Reihe (von unten nach oben), bis man die Maximalhöhe erreicht hat.

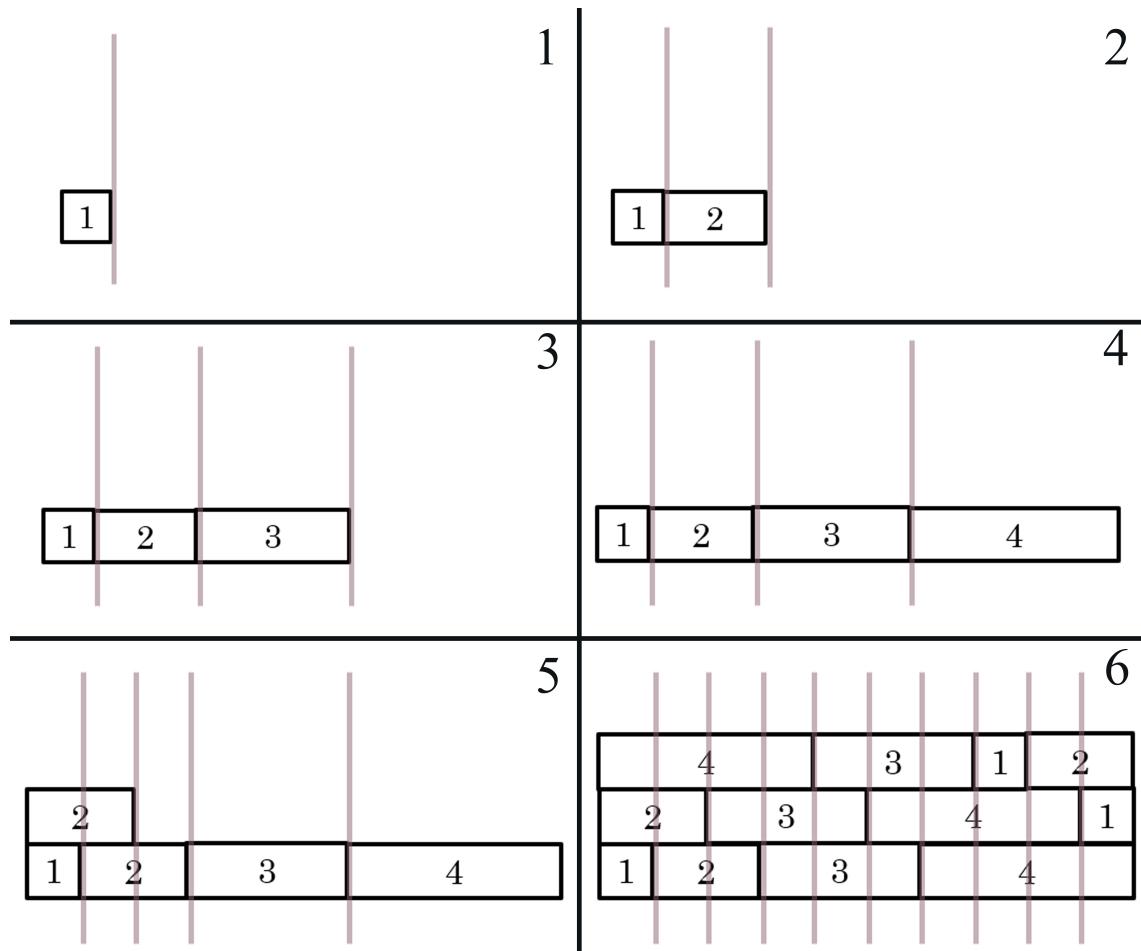


Abbildung 11: Der Bau einer Mauer nach der „**Von unten nach oben**“-Methode. Es wird Reihe nach Reihe gebaut (1-5), bis die maximale Höhe erreicht wurde (6).

Eine andere Möglichkeit ist, wenn man die Anzahl der Reihen (h) für eine Mauer der maximalen Höhe bereits weiß, die Mauer „**von links nach rechts**“ aufzubauen. Man kann die Aufgabe auch so formulieren, dass das **Ziel ist, möglichst alle Fugenpositionen in der Mauer zu füllen**. Nun bietet es sich an, anstatt wie bei „**von unten nach oben**“ immer ein Klötzchen zu setzen und **dann** zu schauen, ob die jeweilige Fugenposition noch frei ist, von den **Fugenpositionen auszugehen** und diese jeweils zu versuchen, **mit einem Klötzchen zu füllen** (siehe Abbildung 12).



Abbildung 12: Der Bau einer Mauer nach der „**Von links nach rechts**“-Methode. Es wird jeweils immer die nächste freie Fuge angeschaut und überprüft, welches Klötzchen in welcher Reihe diese Fuge füllen kann. Dieses Klötzchen wird dann gesetzt (1 bis 5). Dies wiederholt sich so lange, bis die letzte Fuge erreicht wurde und die Mauer fertig ist (6).

Diese „**Von links nach rechts**“-Bauart ist dadurch möglich geworden, dass man schon von vornherein die Anzahl an Reihen kennt, die für eine maximale Mauer benötigt wird.

Aber auch bei dieser Bauart bleibt die Frage, was passiert, wenn mehr als ein Klötzchen die nächste Fuge füllen kann. Dies passiert nämlich bei fast jeder Fuge (siehe Abbildung 13).

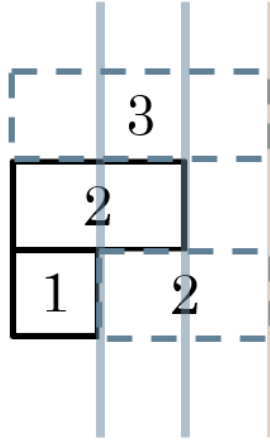


Abbildung 13: Zum Besetzen der dritten Fuge (orange markiert) bei $n = 4$ kann entweder ein 3er-Klötzchen in einer neuen Reihe, oder ein 2er-Klötzchen in der untersten Reihe platziert werden.

Wählt man ein zufälliges Klötzchen aus, so kann es schnell dazu kommen, dass eine Reihe so weit in „Rückstand“ gerät, dass kein mögliches Klötzchen mehr lang genug wäre, um die Reihe fortzuführen (siehe Abbildung 14).

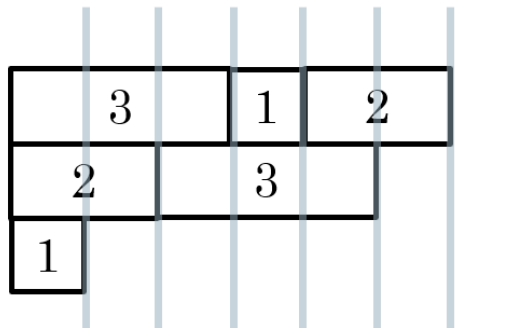


Abbildung 14: In der dargestellten Mauer für $n = 4$ gibt es für die **unterste Reihe keine Möglichkeit mehr, ein Klötzchen zu platzieren**. Da die oberen Reihen alle Fugen schon besetzt haben und die nächste freie Fuge **6 Längen** von der untersten Reihe entfernt ist, würde diese Reihe ein **6er-Klötzchen** benötigen. Bei $n = 4$ gibt es jedoch maximal **4er-Klötzchen**.

Es ist also wichtig, immer darauf zu achten, dass die kürzeste Reihe nicht den Anschluss verliert.

Die Lösung für dieses Problem ist es, bei dem Fall, wenn verschiedene Klötzchen mehrerer Reihen eine Fuge befüllen können, immer die **kürzeste Reihe zu bevor-**

zugen.

Mit diesem Algorithmus kommt man bis zu $n = 8$. Dort tritt ein weiteres Problem auf (siehe Abbildung 15).

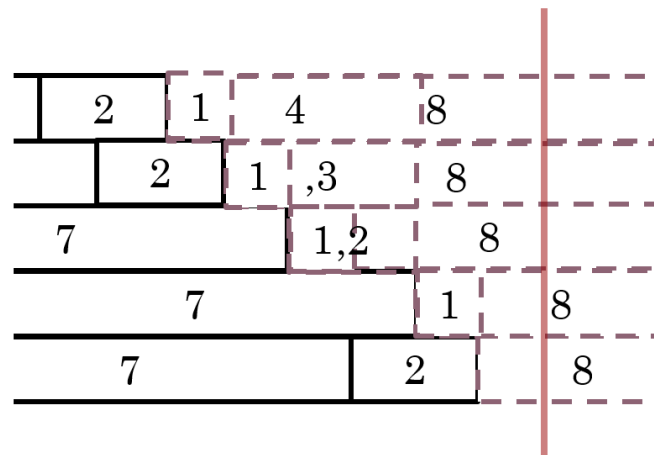
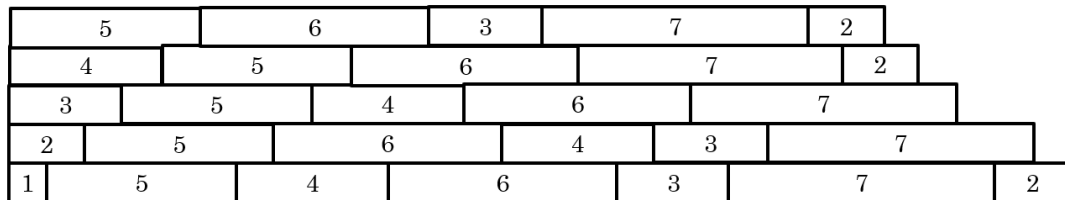


Abbildung 15: Bei $n = 8$ tritt das Problem auf, dass die **29. Fuge** (rot markiert) durch **kein einziges mögliches Klötzchen besetzt werden kann**. Da n jedoch eine **gerade Zahl** ist, wissen wir, dass für eine Mauer der maximalen Höhe **alle Fugen besetzt werden müssen**. Das untere Bild zeigt noch mal den hinteren Ausschnitt der Mauer mit allen noch möglichen Klötzchen für jede Reihe an. Keines davon würde die nächste Lücke füllen.

Um dennoch eine Lösung zu bekommen, wird **Backtracking** verwendet. Backtracking erlaubt es, **zur letzten Entscheidung zurückzugehen** und eben **eine andere zu treffen**. Mit „Entscheidung“ ist die Wahl eines Klötzchens, um eine freie Fuge zu besetzen, wenn es mehrere Möglichkeiten gibt, gemeint (siehe Abbildung 16).

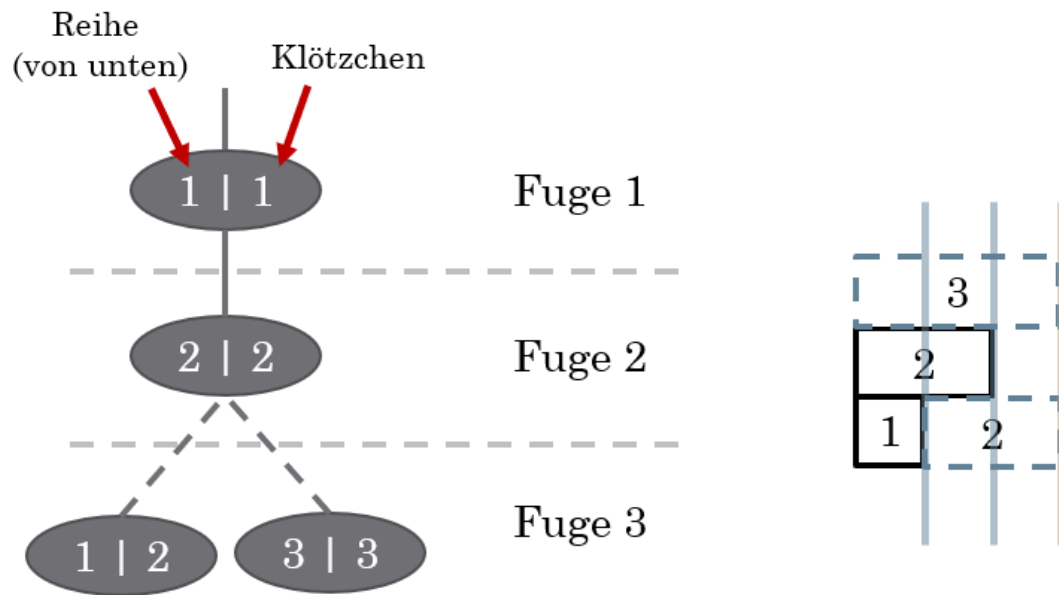


Abbildung 16: Bei einer Mauer von $n = 4$ tritt das vorher schon besprochene Problem auf, dass es zwei mögliche Klötzchen gibt, die die **3. Fuge** besetzen können. Der Baum auf der linken Seite veranschaulicht die möglichen Klötzchen pro Fuge. Die **Tiefe des Baumes** entspricht dabei einer **jeweiligen Fugenposition** und die **Knoten dieser Tiefe** stehen für die **möglichen Klötzchen**.

Nun wird das Klötzchen der kürzesten Reihe (das 3er-Klötzchen in der obersten, neuen Reihe) gewählt und **davon ausgegangen, dass die 4. Fuge danach nicht mehr gefüllt werden kann** (siehe Abbildung 17).

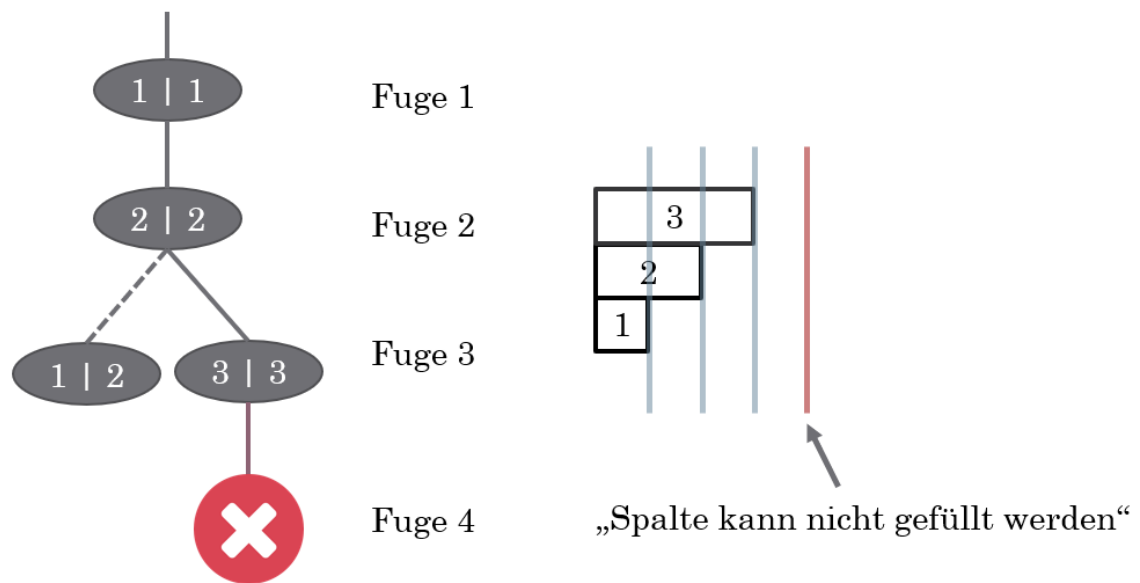


Abbildung 17: Angenommen, nach der Entscheidung, das 3. Klötzchen einzusetzen, kann die 4. freie Fuge nicht besetzt werden. Theoretisch könnte in der untersten Reihe ein 3er-Klötzchen die 4. Fuge füllen. Zur Vereinfachung der Darstellung wird dies jedoch außer Acht gelassen.

Daraufhin wird **einen Schritt zurück zur letzten Entscheidung** gegangen und **dort das andere Klötzchen** gewählt (siehe Abbildung 18).



Abbildung 18: Nach dem „Trial and error“-Prinzip wird beim Backtracking zur letzten Entscheidung zurückgegangen (oberes Bild) und ein anderes Klötzchen ausgewählt (unteres Bild). Dann wird mit dem Algorithmus fortgefahren.

Damit ist der grundlegende Algorithmus vollständig.

2.4 Implementierung

Das Programm wurde in der Sprache C# als Konsolen-Applikation mit dem .NET Framework 4.7.1 geschrieben. Das am Ende des Kapitels in Abbildung 21 stehende Klassendiagramm zeigt die Grundstruktur des Programms.

Im Folgenden soll nur die Implementierung der für den „Von links nach rechts“-Algorithmus wichtigsten Methode erklärt werden. Auf der beigelegten CD befindet sich zusätzlich eine vollständige Dokumentation aller Methoden und Attribute einmal in Form eines PDFs und einer HTML-Website.

Die Klasse `WallBuilder` ist für den Algorithmus die wichtigste Klasse. Sie baut die Mauer. Die für diesen Vorgang verantwortliche Methode ist `FillNextGap` (siehe Abbildung 20), die als Eingabeparameter einen Integer `nextGap`, ein `Wall` Objekt `curWall` und einen weiteren Integer `freeGaps` erwartet und ein Objekt der Klasse `Wall` zurückgibt.

Die Methode ist **direkt rekursiv** und teilt somit die Problemstellung, eine komplette Mauer zu bauen, in mehrere kleinere Probleme auf. Diese sind, für die jeweils nächste Fuge ein Klötzchen zu finden, das sie besetzt.

Als Erstes wird mit der Abbruchbedingung in Zeile 4 kontrolliert, ob die Mauer vollständig gebaut ist. Durch das Programmiermodell **LINQ** wird überprüft, ob die Summe aller platzierten Klötzchen jedes `Row` Objekts der übergebenen `Wall` Instanz `curWall` mit der Länge einer vollständigen Mauer übereinstimmt. Ist die Abbruchbedingung `true`, so kommt es zu einem Rekursionsende und das fertige `Wall` Objekt wird zurückgegeben.

Ansonsten wird das übergebene `Wall` Objekt „kopiert“, denn es ist sehr wichtig, dass am übergebenen `curWall` Objekt nichts geändert wird. Objekte der Klassen `Wall` und `Row` sind nämlich sogenannte Referenz-Typen (engl. Reference Types). Anders als bei Werte-Typen (engl. Value Types), bei denen bei einer erneuten Zuweisung der gespeicherte Wert kopiert und die Variable dann auf diesen kopierten Wert verweist, wird bei einer erneuten Zuweisung eines Referenz-Typs **nur der Verweis** (engl. Pointer) **kopiert**. Die Variable greift jedoch auf das **gleiche Objekt im Systemspeicher** zu wie die vorherige Variable.

Dies hätte zur Folge, dass wenn an dem übergebenen `Wall` Objekt zu späterem Zeitpunkt irgendwelche Änderungen vorgenommen werden, sich auch das `Wall` Objekt der Aufrufer-Methode ändert. Dies würde das Backtracking erheblich erschweren, da nicht mehr auf den Zustand der Mauer **vor einer Klötzchen-Entscheidung** zurückgegriffen werden könnte.

In Zeile 10 bis 14 wird dann die für den aktuellen Schritt zu befüllende Fugenposition gewählt. Der Parameter `nextGap` beschreibt also die **zuletzt gefüllte Fuge**.

Es wird überprüft, ob es nach `nextGap` überhaupt noch eine Fuge gibt. Ist dem so, wird die Variable `nextGapPos` mit dem Wert `nextGap + 1`, also der Position der nächsten Fuge, initialisiert. Es kann jedoch auch sein, dass die Mauer schon am Ende angelangt ist und im letzten Schritt bereits die letzte „Fugenposition“ und damit ist hierbei das Ende der Mauer gemeint, gefüllt wurde (siehe Abbildung 19). Wichtig ist, dass dies nicht direkt heißt, dass die Mauer fertig ist. Es muss nämlich jede andere Reihe auch noch die letzte „Fuge“ erreichen.

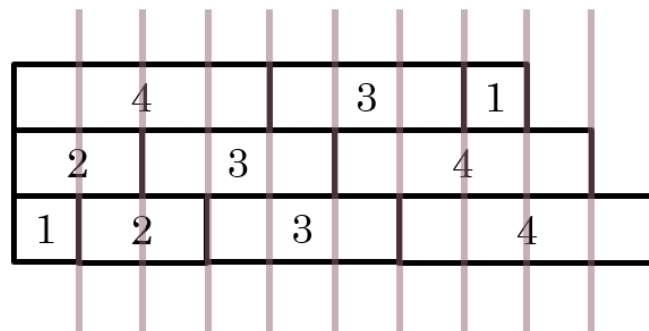


Abbildung 19: Bei dieser Mauer von $n = 4$ hat die unterste Reihe die letzte „Fugenposition“ bereits gefüllt. Diese End-„Fuge“ darf und muss jedoch mehrmals besetzt werden, um jede Reihe abzuschließen.

Deshalb wird in diesem Fall die Variable `nextGapPos` mit dem gleichen Wert wie `nextGap` initialisiert.

Im nächsten Ausdruck, der sich von Zeile 17 bis 19 erstreckt, werden wieder mittels **LINQ** alle `Row` Objekte der aktuellen `Wall` Instanz, die durch ein noch verfügbares Klötzchen die gewünschte Fugenposition befüllen können, in einem Array gesammelt.

Von Zeile 22 bis 34 wird nun geprüft, ob es keine einzige Reihe gibt, die die nächste Fuge besetzen kann. Hier wird der Sonderfall berücksichtigt, dass der Parameter `freeGaps` größer als 0 ist. Dies kann nämlich der Fall sein, wenn n **ungerade** ist. Bei ungeraden n kann es ja zu vollständigen Mauern kommen, **ohne**, dass **alle** Fugen besetzt sind. In der Tat kann man auch relativ einfach ausrechnen, wie viele Fugen für ein ungerades n frei bleiben können (siehe Methode `CalculateWallProperties` der Klasse `WallBuilder`). Somit kann es passieren, wenn kein Klötzchen in keiner Reihe zum Besetzen einer freien Fuge gefunden wurde und der „freie Fugen-Puffer“ der aktuellen Mauer noch größer als 0 ist, dass diese freie Fuge **nicht besetzt** und einfach **übersprungen** werden kann. Deshalb findet in Zeile 26, wenn `freeGaps` noch größere als 0 ist, einfach der rekursive Aufruf statt, mit dem `freeGaps` Parameter um 1 verringert. Der Rückgabewert dieses rekursiven Aufrufs wird in der Variable `result` gespeichert. Ist `result` ein `Wall` Objekt, wird dieses weiter zurückgegeben, ansonsten wird der restliche Code bis zum Rekursionsende in Zeile 51 ausgeführt. Tritt der Sonderfall `freeGaps > 0` nicht ein wird `null` zurückgegeben.

In Zeile 37 wird das Array der `Row` Objekte, die die nächste Fuge füllen können, nach ihrer `RowSum` sortiert. Mit `RowSum` ist hierbei einfach die Summe aller in der Reihe platzierten Klötzchen gemeint. Nach dem Sortieren ist das erste Objekt des `possibleRows` Arrays also die **kürzeste Reihe** und das letzte Objekt die bereits **längste Reihe**.

In der For-Schleife von Zeile 40 bis 48 werden nun die `Row` Objekte des `possibleRows` Array einzeln iteriert. Zuerst wird in Zeile 42 das Klötzchen platziert, das die gesuchte Fuge füllt. (Das Klötzchen wurde in der vorher aufgerufenen Methode `ContainsPossibleRowSum` der Klasse `WallBuilder` herausgefiltert.)

Danach kommt ein rekursiver Aufruf, dessen Rückgabewert in der lokalen Variable `result` gespeichert wird. In der Abbruchbedingung von Zeile 44 wird wieder überprüft, ob `result` ein `Wall` Objekt ist. Ist dies der Fall, wird es weiter zurückgegeben.

Ist im Gegensatz dazu `result` gleich `null`, heißt das, dass mit dieser Klötzchen-Entscheidung keine fertige Mauer gebaut werden konnte und deshalb bis zu dieser Entscheidung zurückgegangen wurde, um dort ein anderes Klötzchen zu wählen (**Backtracking**).

Deshalb wird das zuvor platzierte Klötzchen des aktuellen Row Objekts in Zeile 47 wieder entfernt.

Gibt es danach noch weitere Row Objekte im `possibleRows` Array, wird die Schleife mit dem nächsten Row Objekt durchlaufen.

Als letzte Anweisung der Methode wird mit einem Rekursionsende `null` zurückgegeben.

```
1  public Wall FillNextGap(int nextGap, Wall curWall, int freeGaps)
2  {
3      // Check if wall is finished
4      if (curWall.Rows.All(r => r.RowSum == WallLength))
5          return curWall;
6
7      var wall = curWall.Clone();
8
9      // Check if there is a next gap
10     int nextGapPos;
11     if (nextGap < GapCount + 1)
12         nextGapPos = nextGap + 1;
13     else
14         nextGapPos = nextGap;
15
16     // Get all rows that can reach the next gap
17     var possibleRows = wall.Rows
18         .Where(r => r.NextPossibleRowSums.Any(nrs => ContainsPossibleRowSum(
19             r, nrs, nextGapPos))).ToArray();
20
21     // If no row can reach next gap and FreeGaps > 0 continue to next gap
22     if (possibleRows.Length == 0)
23     {
24         if (freeGaps > 0)
25         {
26             var result = FillNextGap(nextGapPos, wall, freeGaps - 1);
27             if (result != null) return result;
28         }
29         else
30         {
31             // track back
32             return null;
33         }
34     }
35
36     // Get row with lowest row sum and call FillNextGap()
37     Array.Sort(possibleRows);
38
39     // Backtracking
40     for (var i = 0; i < possibleRows.Length; i++)
41     {
42         possibleRows[i].PlaceNextBrick();
43         var result = FillNextGap(nextGapPos, wall, freeGaps);
44         if (result != null) return result;
45
46         // remove wrong placed brick and try next branch
47         possibleRows[i].RemoveLastBrick();
48     }
49
50     // track back
51     return null;
52 }
```

Abbildung 20: Quellcode der Methode `FillNextGap` der Klasse `WallBuilder`

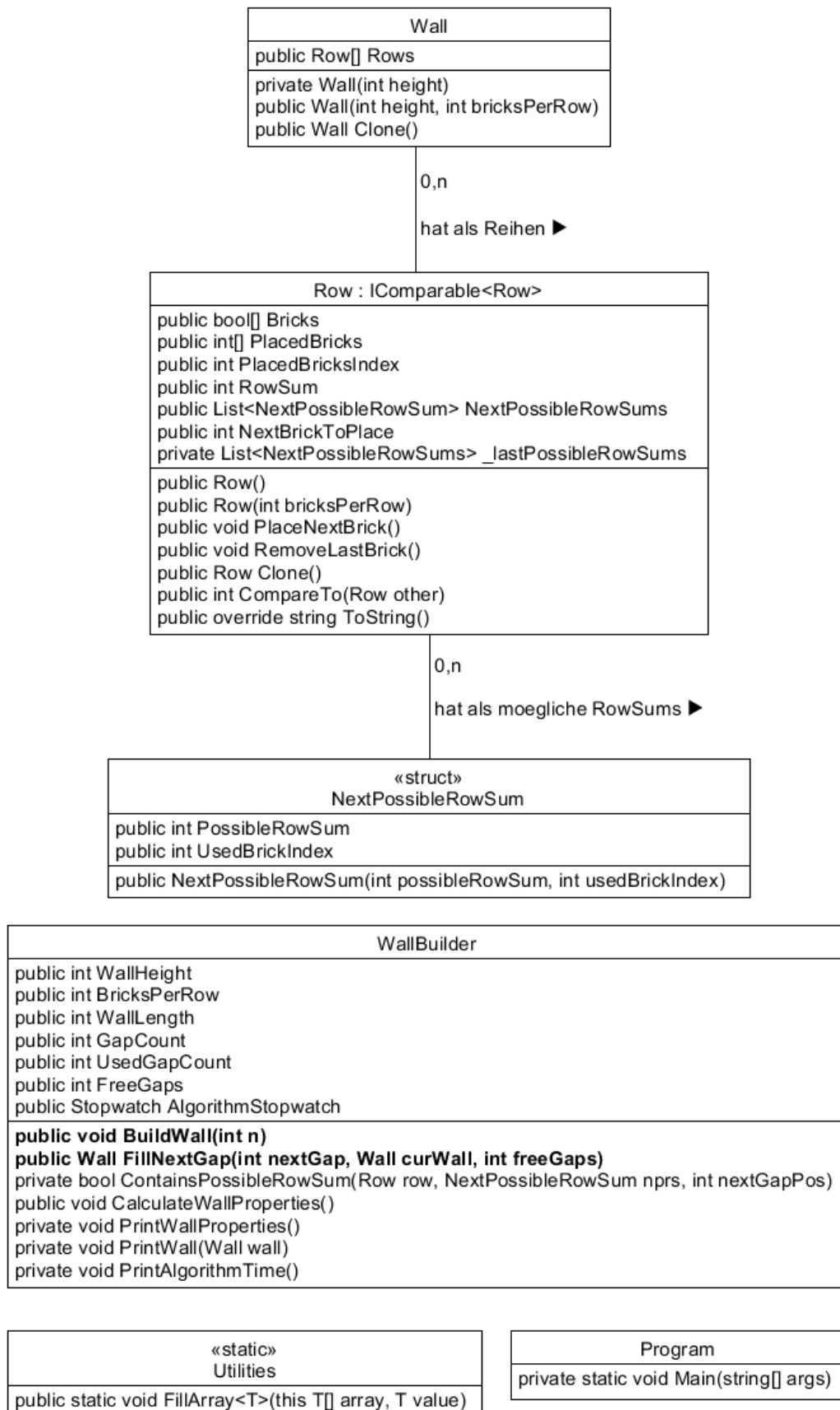


Abbildung 21: Erweitertes Klassendigramm des Programms. Die wichtigsten zwei Methoden des Algorithmus wurden in der Klasse `WallBuilder` hervorgehoben.

2.5 Optimierungsmöglichkeiten

Mit der Implementierung dieses Algorithmus können maximale Mauern bis zu $n = 17$ in wenigen Sekunden erzeugt werden. Jedoch steigt die durchschnittliche Algorithmuslaufzeit ab $n = 17$ sehr stark an. Für $n = 18$ dauert das Finden einer Mauer schon mehrere Minuten.

Dieser Algorithmus lässt sich jedoch noch wesentlich optimieren. In den vom Bundeswettbewerb Informatik offiziell veröffentlichten Lösungshinweisen werden zur ersten Aufgabe noch mehrere Optimierungsmöglichkeiten genannt:

- Randomized Algorithm
- Kopplung und Spiegelung einer halben Mauer
- Einschränkung des Suchraums
- Ergänzung vorheriger Permutationen

Mithilfe dieser Optimierungen kann der Algorithmus so erweitert werden, dass er bis $n = 110$ eine Mauer wenigen Minuten generieren kann.

Dies ist vor allem eindrucksvoll, wenn man bedenkt, dass es für $n = 100$ schon

$$100!^{51} \approx 2,95\text{e}+8,056$$

unterschiedliche Mauern der maximalen Höhe gibt.

Das ist eine Zahl mit **8057 Stellen**.

2.6 Beispiele

Wie bereits erwähnt, kann der Algorithmus Mauern von $n = 2$ bis $n = 17$ in wenigen Sekunden erstellen. Weitere Test mit größeren n wurden nach mehreren Minuten abgebrochen.

Außerdem können Mauern von $n = 2$ bis $n = 7$ komplett **ohne Backtracking** generiert werden. Dies sorgt dafür, dass die Algorithmuslaufzeiten für diese n in etwa gleich sind.

Weiterhin erwähnenswert ist, dass das Finden von Mauern für ungerade n meist

kürzer dauert als das Generieren der $n - 1$ Mauer, da dort Fugen freigelassen werden können und es dadurch deutlich mehr mögliche Mauerkombinationen gibt.

Im Folgenden sind die durchschnittlichen Algorithmuslaufzeiten für die verschiedenen n in einer Tabelle aufgelistet.

n	Algorithmuslaufzeit
2	10,25 ms
3	10 ms
4	11 ms
5	10 ms
6	10,75 ms
7	10,75 ms
8	10 ms
9	10,75 ms
10	15 ms
11	14,75 ms
12	29,25 ms
13	42 ms
14	372 ms
15	125 ms
16	11348 ms
17	636 ms

Die linke Spalte gibt den eingegeben Wert für n an, die rechte die durchschnittliche Algorithmuslaufzeit in Millisekunden.

Außerdem zeigen Abbildung 22 bis 24 Konsolen-Ausgaben des Programms für verschiedene n . Die Mauerlösung ist dabei die hellblaue Ausgabe. Die verschiedenen Klötzchen einer Reihe werden mit dem Trennstrich (|) abgetrennt. Die Zahl innerhalb der Trennstriche steht für die Länge eines Klötzchens.

```

Aufgabe1_DieKunstDerFuge
BwInf36 | Runde 2 | Aufgabe 1 (Die Kunst der Fuge)
=====

Anzahl der Kloetzchen in einer Reihe: 1

Die Anzahl der Kloetze muss zwischen 2 (eingeschlossen) und 22 (eingeschlossen) liegen!
Bitte waehlen Sie eine andere Anzahl von Kloetzchen in einer Reihe: abc

Die Anzahl der Kloetze muss zwischen 2 (eingeschlossen) und 22 (eingeschlossen) liegen!
Bitte waehlen Sie eine andere Anzahl von Kloetzchen in einer Reihe: 6

Anzahl Kloetzchen in einer Reihe: 6
Breite der Mauer: 21
Maximale Hoehe der Mauer: 4
Anzahl verfuegbarer Stellen fuer Fugen: 20
Anzahl benoetigter Fugen fuer Mauer der maximalen Hoehe: 20

The algorithm took 9ms to complete.

| 1 | 4 | 3 | 2 | 6 | 5 |
| 2 | 4 | 5 | 3 | 6 | 1 |
| 3 | 4 | 5 | 1 | 6 | 2 |
| 4 | 5 | 6 | 2 | 1 | 3 |

Druecke ENTER um das Programm zu beenden

```

Abbildung 22: Programm-Ausgabe für $n = 6$ mit vorherigen fehlerhaften Eingaben für n (1 und abc)

```

Aufgabe1_DieKunstDerFuge
BwInf36 | Runde 2 | Aufgabe 1 (Die Kunst der Fuge)
=====

Anzahl der Kloetzchen in einer Reihe: 10

Anzahl Kloetzchen in einer Reihe: 10
Breite der Mauer: 55
Maximale Hoehe der Mauer: 6
Anzahl verfuegbarer Stellen fuer Fugen: 54
Anzahl benoetigter Fugen fuer Mauer der maximalen Hoehe: 54

The algorithm took 11ms to complete.

| 1 | 6 | 5 | 4 | 7 | 8 | 9 | 3 | 2 | 10 |
| 2 | 6 | 7 | 4 | 5 | 3 | 8 | 1 | 10 | 9 |
| 3 | 6 | 5 | 8 | 7 | 9 | 1 | 10 | 2 | 4 |
| 4 | 6 | 7 | 3 | 5 | 8 | 9 | 10 | 1 | 2 |
| 5 | 6 | 7 | 8 | 2 | 4 | 9 | 3 | 10 | 1 |
| 6 | 7 | 8 | 9 | 4 | 3 | 10 | 1 | 2 | 5 |

Druecke ENTER um das Programm zu beenden

```

Abbildung 23: Programm-Ausgabe für $n = 10$

```
Aufgabe1_DieKunstDerFuge
BwInf36 | Runde 2 | Aufgabe 1 (Die Kunst der Fuge)
=====
Anzahl der Kloetzchen in einer Reihe: 17

Anzahl Kloetzchen in einer Reihe: 17
Breite der Mauer: 153
Maximale Hoehe der Mauer: 9
Anzahl verfuegbarer Stellen fuer Fugen: 152
Anzahl benoetigter Fugen fuer Mauer der maximalen Hoehe: 144

The algorithm took 629ms to complete.

| 1 | 9 | 8 | 10 | 7 | 11 | 6 | 12 | 13 | 5 | 4 | 14 | 3 | 2 | 17 | 15 | 16 |
| 2 | 9 | 10 | 8 | 7 | 11 | 6 | 5 | 13 | 4 | 14 | 12 | 3 | 16 | 1 | 17 | 15 |
| 3 | 9 | 8 | 10 | 11 | 7 | 6 | 5 | 13 | 4 | 14 | 12 | 15 | 1 | 16 | 17 | 2 |
| 4 | 9 | 10 | 8 | 11 | 7 | 12 | 5 | 13 | 6 | 14 | 15 | 2 | 3 | 16 | 17 | 1 |
| 5 | 9 | 8 | 10 | 7 | 11 | 12 | 3 | 13 | 6 | 14 | 15 | 16 | 1 | 2 | 17 | 4 |
| 6 | 9 | 10 | 8 | 7 | 11 | 12 | 4 | 13 | 3 | 14 | 15 | 16 | 17 | 1 | 2 | 5 |
| 7 | 9 | 8 | 10 | 11 | 12 | 13 | 4 | 14 | 3 | 2 | 15 | 16 | 1 | 17 | 5 | 6 |
| 8 | 9 | 10 | 11 | 6 | 12 | 13 | 4 | 14 | 5 | 2 | 15 | 1 | 16 | 17 | 7 | 3 |
| 9 | 10 | 7 | 11 | 6 | 12 | 5 | 8 | 13 | 14 | 1 | 15 | 4 | 16 | 2 | 3 | 17 |

Druecke ENTER um das Programm zu beenden
-
```

Abbildung 24: Programm-Ausgabe für $n = 17$

3 AUFGABE 3 – „Quo vadis, Quax?“

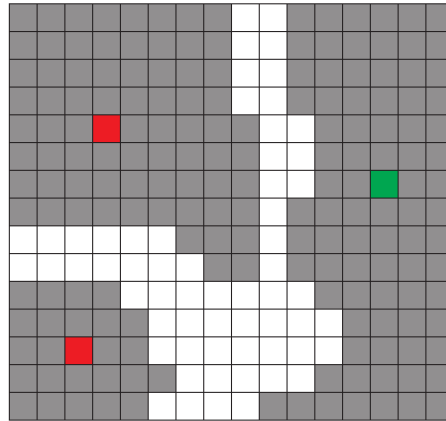
3.1 Aufgabenstellung

In einer abgelegenen und unwegsamen Gegend will Quax die Fähigkeiten seines neuen Quadcopters ausprobieren. Da wird aus dem Spiel Ernst: Ein Tornado mit anschließender Überschwemmung vernichtet seine Essenvorräte, so dass er seinen Urlaub abbrechen und sofort die lange Wanderung zurück zur einzigen Stadt der Provinz antreten muss. Außerdem hat sich die Landschaft vollkommen verändert – es haben sich Flüsse und Seen gebildet, wo früher keine waren. Es ist nicht klar, ob es überhaupt noch einen gangbaren Weg zurück zur Stadt gibt und, falls ja, wo sich ein solcher Weg zwischen den Gewässern durchschlängelt.

Quax möchte die Landschaft mit dem Quadcopter erkunden, bevor er loslegt, um lange Umwege zu vermeiden, die er sich nicht leisten kann. Der Quadcopter kann in einer frei wählbaren Flughöhe einen beliebigen Punkt der Landschaft anfliegen. Allerdings ist auch er beschädigt: Die Speicherkarte ist hin, und es funktioniert nur ein einziger Sensor. Dieser erfasst ein quadratisches, nach Norden ausgerichtetes Gebiet unter dem Quadcopter, dessen Seitenlänge proportional zur Flughöhe ist. Der Sensor kann für dieses Testgebiet entscheiden, ob es ganz von Wasser bedeckt ist („Wasser“), ganz aus trockenem Land besteht („Land“) oder sowohl Wasser als auch Land enthält („Gemischt“).

Quax kann wiederholt den Quadcopter für eine bestimmte Kombination aus Flughöhe und Zielpunkt programmieren, ihn entsenden und bei seiner Rückkehr das Ergebnis („Wasser“, „Land“ oder „Gemischt“) in Empfang nehmen. Liefert der Quadcopter auch für ein kleinstmögliches Testgebiet mit Seitenlänge 20m immer noch das Ergebnis „Gemischt“, geht Quax davon aus, dass er dieses Testgebiet überqueren kann – zur Not schwimmt er ein paar Meter mit dem Gepäck.

Das folgende Pixelbild zeigt eine Beispiel-Landschaft. Jedes Pixel repräsentiert einen quadratischen Bereich der Landschaft mit Seitenlänge 10 m. Das grüne Pixel zeigt die Position der Stadt. Vom oberen roten Startpunkt kann Quax die Stadt erreichen, vom unteren aber nicht.



Aufgabe

- (a) Schreibe ein Programmmodul, das aus den Ergebnissen durchgeführter Flüge bestimmt, ob es für Quax einen gangbaren Weg zurück zur Stadt gibt, ob es definitiv keinen solchen Weg gibt oder ob mehr Flüge nötig sind, um eine Entscheidung herbeizuführen.
- (b) Beschreibe eine Strategie für das Entsenden des Quadcopters, die mit möglichst wenig Flügen einen Weg zur Stadt findet, falls es einen gibt.
- (c) Schreibe ein Programm, das eine Landschaft einliest, in welcher die Stadt sowie mehrere mögliche Standorte von Quax markiert sind. Ein Pixel repräsentiert ein Quadrat der Seitenlänge 10 m. Dann soll das Programm die anhand der Strategie aus (b) festgelegten Entsendungen – für jeden Standort des Quadcopters einzeln – simulieren. Ausgeben soll es je eine Darstellung der Landschaft, in der die Testgebiete mit ihren jeweiligen Ergebnissen geeignet eingetragen sind. Falls es einen gangbaren Weg zur Stadt gibt, soll dieser in der Darstellung erkennbar sein.

Wende dein Programm mindestens auf die auf der BWINF-Website gegebenen Beispiele an und dokumentiere die Ergebnisse. Gehe davon aus, dass sich außerhalb der eingelesenen Landschaft nur Wasser befindet.

- (d) Gibt es Verteilungen von Flüssen und Seen, bei denen deine Strategie aus (b) zu sehr vielen Flügen führt? Falls nein, erkläre warum nicht. Falls ja, gib ein Beispiel dafür an und erkläre, worin das Problem besteht.

3.2 Auslegung der Aufgabe

Die Aufgabenstellung ist größtenteils sehr klar. Das grundlegende Problem ist eine Wegfindung. Jedoch reicht als Lösung kein einfacher Wegfindungs-Algorithmus, da es bestimmte Bedingungen gibt, wie das der Quadcopter minimal 20x20 Gebiete scannen kann. Beim Scannen eines Gebiets gibt es 3 Mögliche Resultate. Entweder das Gebiet besteht eindeutig aus Wasser oder aus Land, oder das Gebiet liefert den Wert gemischt und besitzt damit sowohl Wasser als auch Land.

Das Endziel des Algorithmus ist nicht wie bei den meisten normalen Pathfinding Algorithmen möglichst schnell einen möglichst kurzen Weg zu finden, sondern mit **möglichst wenig Flügen** einen Pfad zur Stadt zu finden.

Angenommen man verwendet einen auf einem 20x20 Gitter basierenden Wegfindungs-Algorithmus, so müsste der Quadcopter zuerst

$$\lceil \frac{w}{2} \rceil \times \lceil \frac{h}{2} \rceil$$

wobei w die Breite und h die Höhe der Karte sind, Flüge durchführen, um das benötigte Gitter zu erstellen. Im Falle der ersten offiziellen Test-Map des BwInfs (siehe Abbildung 25) mit den Dimensionen 1674x1102 wären das **461,187 Flüge**.



Abbildung 25: quax1.png des zur Aufgabe 3 verfügbar gestellten Materials

Hier könnte man allerdings allein schon zehntausende Flüge einsparen, wenn man

den Quadcopter die linke obere Ecke von einer höheren Position aus scannen lässt, so dass er ein größeres Gebiet auf einmal betrachtet (siehe Abbildung 26).

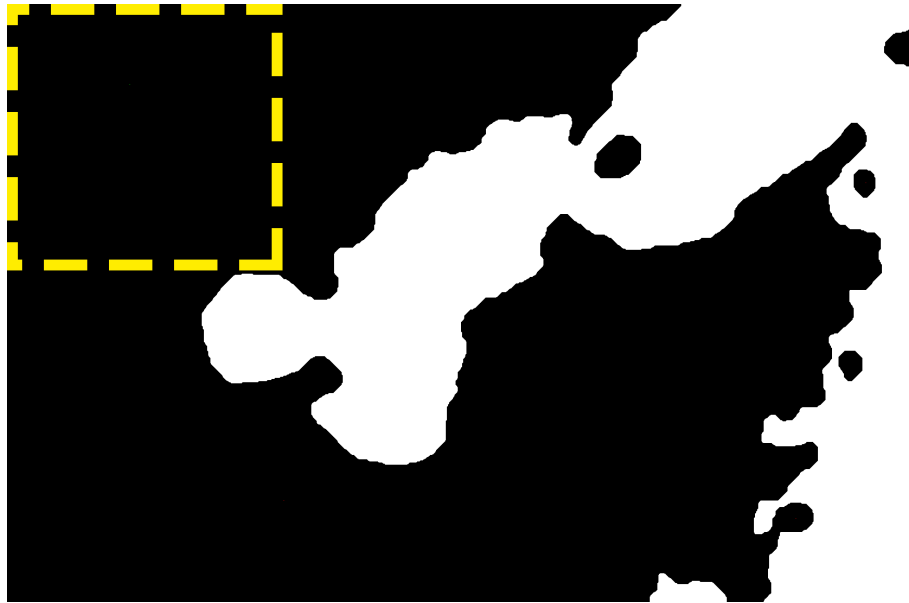


Abbildung 26: Das einzelne Scannen dieses Bereiches würde ungefähr **63 tausend Flüge einsparen**.

Es lohnt sich also groß anzufangen und nicht direkt auf der niedrigsten Höhe die Map zu untersuchen.

Zusätzlich würde es nicht ausreichen das Gebiet in nebeneinander liegende 20x20 Quadrate zu unterteilen, da es so zu einem Sonderfall kommen könnte.

In der Aufgabenstellung ist definiert, dass Quax zwar ein 20x20 gemischtes Quadrat überqueren kann, jedoch kein 20x20 Wasser-Quadrat.

Nun kann es dazu kommen, dass zwei benachbarte, gemischte 20x20 Quadrate ein quasi „unentdecktes“ 20x20 Wasser-Quadrat zwischen sich haben (siehe Abbildung 27). In diesem Fall würde ein normaler Wegfindungs-Algorithmus Quax diese zwei 20x20 gemischten Quadrate überqueren lassen. Jedoch müsste Quax dafür auch ein 20x20 Wasser-Quadrat überqueren, was nicht geht.

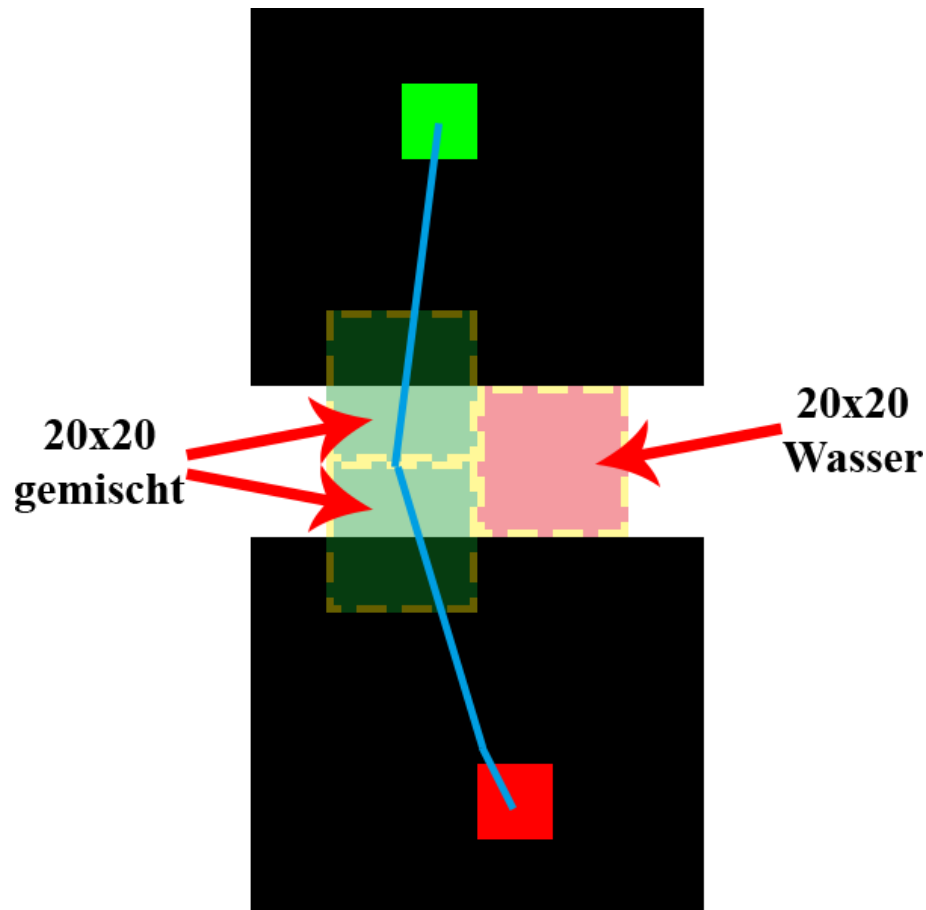


Abbildung 27: Bei einer Überquerung der beiden Linken 20x20 Quadrate, würde Quax das eigentliche 20x20 Wasser-Quadrat einfach „überspringen“.

3.3 Lösungsidee

Um die Map mit möglichst wenig Flügen zu untersuchen, eignet es sich einen Quadtree („Quaternärbaum“) zu verwenden. Ein Quadtree ist eine Baum-Datenstruktur, bei der jeder innere Knoten genau vier Tochterknoten hat. Quadrees werden in der Informatik oft zur Unterteilung von Bildern verwendet.

Diese Datenstruktur eignet sich hervorragend für die Unterteilung der Karte. Ein größeres Quadrat wird dabei in jeweils 4 gleichgroße kleinere Quadrate unterteilt (siehe Abbildung 28).

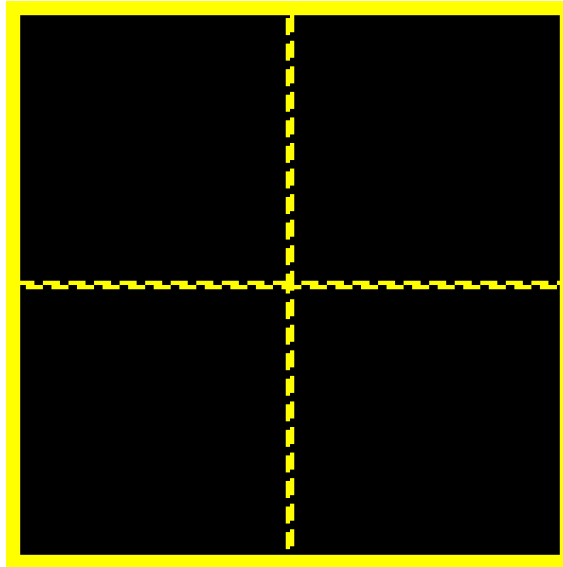


Abbildung 28: Das größere Quadrat wird in jeweils 4 gleichgroße, kleinere Quadrate unterteilt. Das große Quadrat Objekt, hält dabei eine Referenz auf die Kleineren.

Man teilt ein Quadrat immer dann auf, wenn der jeweilige Inhalt nicht eindeutig als Land oder Wasser bestimmt werden kann, und das Quadrat noch größer als 20x20 ist.

Abbildung 29 zeigt eine solche Unterteilung für eine Beispielkarte. Es wird nach Quaxs Position (1, 10) gesucht. Hierbei wird zuerst die Wurzel des Quadtree gebildet, welche die gesamte Karte beinhaltet. Nachdem diese als gemischt identifiziert wird (1. Flug), wird die Wurzel unterteilt. Da das Süd-Westliche Quadrat den Punkt (1, 10) enthält, wird dieses als nächstes angeflogen (2. Flug). Da jedoch auch dieses das Ergebnis „Gemischt“ zurückliefert, wird es auch unterteilt. Das Süd-Westliche innere Quadrat ist wieder das, welches die Position (1, 10) enthält und wird somit untersucht (3. Flug). Da dieses jedoch eindeutig Land enthält, ist die Unterteilung abgeschlossen.

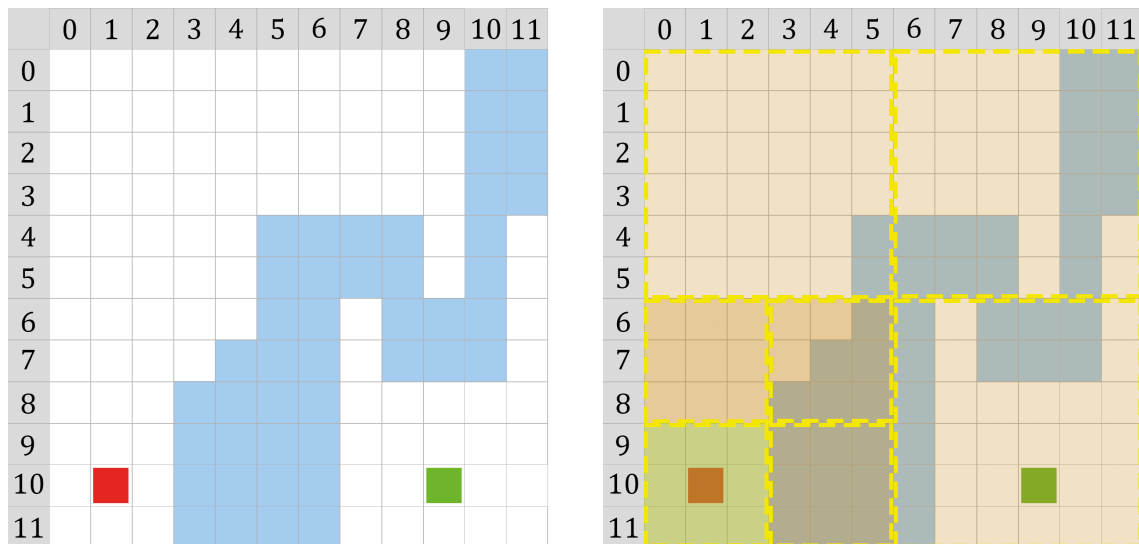


Abbildung 29: Für die links abgebildete Karte wird ein Quadtree (rechts) so gebildet, dass Quaxs Umgebung eindeutig bestimmt werden kann.

Nun stellt diese Quadtree Struktur aber auch eine Bedingung an die zu untersuchende Karte. Da der Quadcopter immer Viereckige Gebiete untersucht und die Wurzel des Quadtree-Baums die gesamte Karte umfassen muss, muss die Karte auch quadratisch sein. Dies ist jedoch kein größeres Problem, da in der Aufgabenstellung festgelegt wurde, dass alle Bereiche außerhalb der definierten Karte als Wasser angenommen werden können.

Jetzt gibt es aber auch beim Quadtree den Sonderfall, dass ein Quadrat mit ungerader Größe unterteilt werden muss. Dabei, wenn man alle Bereiche abdecken will, kommt es zu Überlappungen der inneren Quadrate (siehe Abbildung 29).

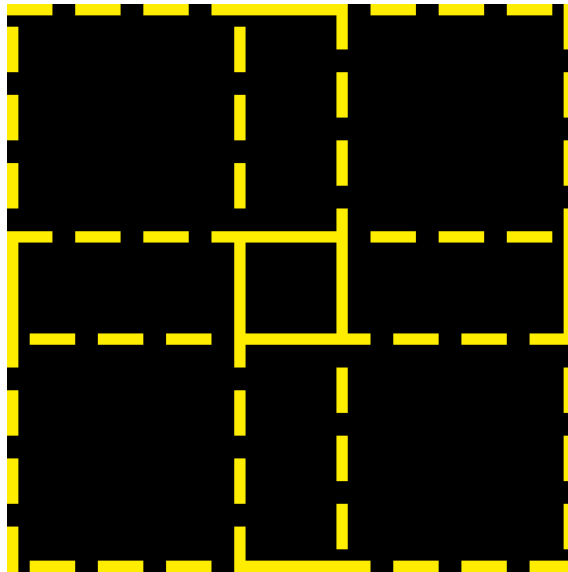


Abbildung 30: Bei einer Aufteilung eines Quadrats mit ungerader Größe überlappen sich die inneren Quadrate.

Um dies zu verhindern könnte man die Karte nicht einfach nur quadratisch machen, sondern auf eine Größe bringen, so dass die Breite und Höhe der Map einer Zweierpotenz entsprechen. Dadurch könnte jede Aufteilung bis zum kleinst-möglichen 20x20 Quadrat ohne Überlappungen durchgeführt werden.

Der Nachteil dieser Skalierung ist jedoch eine ziemliche Vergrößerung der Karte im Vergleich zur ersten Methode (siehe Abbildung 31).

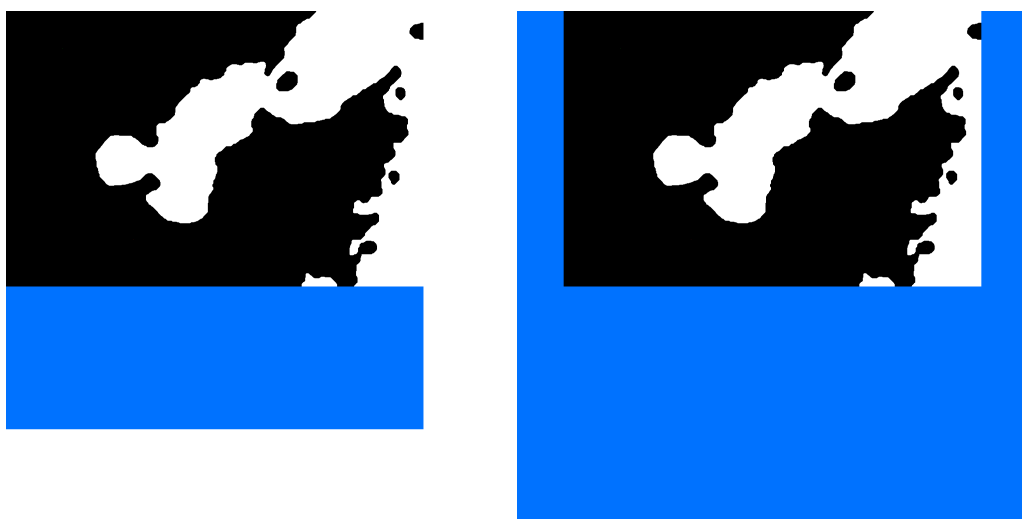


Abbildung 31: Links die einfache Umformung der Map zu einem Quadrat so dass Breite = Höhe. Rechts die Umformung zu einer Zweierpotenz-Größe. Der blau markierte Bereich zeigt jeweils die zusätzlich hinzugefügten Gebiete an.

Für noch größere Karten steigt der Unterschied zwischen den beiden Methoden noch mehr und je größer die Karte, desto mehr Quadcopter Flüge bis man das eigentliche Gebiet erreicht hat.

Um nun auch auf den Sonderfall der zwei angrenzenden, gemischten 20x20 Quadrate, zwischen denen ein 20x20 Wasser Quadrat liegt, Rücksicht zu nehmen, darf der Weg zur Stadt nicht aus 20x20 aneinanderliegenden Quadraten bestehen, sondern aus 20x20 überlappenden Quadraten.

Auf diese Weise kann sichergestellt werden, dass keine 20x20 Wasser-Quadrate, die zwischen zwei 20x20 gemischten Quadraten liegen übersprungen werden.

3.4 Teilaufgabe (a)

Ein Quadrat in der Landschaft kann für Quax drei mögliche Zustände Wasser („W“), Land („L“) oder Unbekannt („U“) haben.

Als Wegfindungs-Algorithmus wird **A*** (**A-Star**) verwendet. A* ist eine Variante des Dijkstra-Algorithmus, die auf einem vorher schon angesprochenem Grid-System beruht (siehe Abbildung 32). Die einzelnen Elemente des Grids werden als „Nodes“ bezeichnet.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Abbildung 32: Findung des kürzesten Wegs vom grünen zum blauen Quadrat durch A*. Hierbei ist nur der Weg markiert und nicht alle untersuchten Nodes. Die Zahlen geben die Entfernung in Nodes zum Ausgangspunkt an.

Die generelle Funktionsweise des A* Algorithmus soll hier nur kurz erklärt werden. Für eine umfassendere Erklärung werden die zu diesem Thema im Literaturverzeichnis angegebenen Quellen empfohlen.

Der A* Algorithmus beginnt mit einer Start-Node (S), einer Target-Node (T), einem `openSet` und einem `closedSet`. Jede Node hat dabei entweder den Status „Walkable“ oder „Unwalkable“. Es werden zuerst alle Nachbar-Nodes von S mit dem Zustand „Walkable“ zum `openSet` hinzugefügt. Für sie werden jeweils 3 Werte berechnet. **G-Cost** beschreibt die Entfernung zu S. **H-Cost** beschreibt die Entfernung zu T. **F-Cost** entspricht **G-Cost** + **H-Cost**. (Für die Berechnung dieser **F-Cost** gibt es mehrere Varianten. Hier wird die sogenannte „Manhattan-Distanz“ verwendet.)

Als nächstes wird die Node im `openSet` mit der geringsten **F-Cost** aus dem `openSet` entfernt und dem `closedSet` hinzugefügt. Für diese Node werden wieder alle Nachbarn mit dem Status „Walkable“ entweder in das `openSet` aufgenommen und ihre Werte berechnet oder, falls sie schon im `openSet` vorhanden sind, werden ihre Werte aktualisiert.

Dieser Vorgang wiederholt sich so lange, bis T erreicht wurde.

Das hier auftretende Problem ist, dass bei einem normalen A* Algorithmus vorher schon das gesamte Grid definiert sein muss. Der Algorithmus muss wissen, welche Nodes „Walkable“ und welche „Unwalkable“ sind um einen Pfad zu finden.

Da das Scannen der gesamten Map jedoch auch mit der Quadtree Struktur viel zu viele letztendlich Quadcopter Flüge beanspruchen würde, gibt es bei der hier verwendeten A* Variante noch den dritten möglichen Node Status „Unknown“.

Wie genau der Algorithmus mit einem dritten Status abläuft, wird nun an einem kleinen Beispiel veranschaulicht.

Zu Beginn hat jede Grid-Node außer S und T, die beide den Status „Walkable“ haben, den Status „Unknown“ (siehe Abbildung 33).

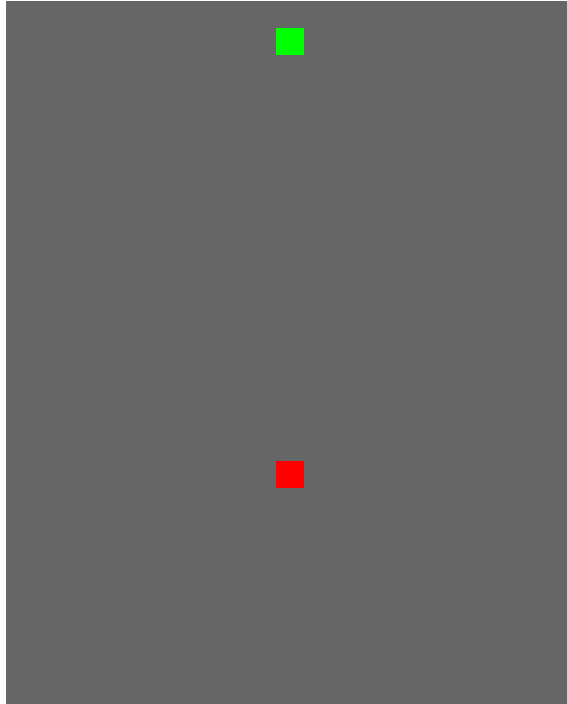


Abbildung 33: Die Map aus A* Sicht. Alle Nodes des Status „Unknown“ sind grau markiert.

Nun wird der erste Flug durchgeführt und die Grid-Nodes dementsprechend aktualisiert (siehe Abbildung 34).

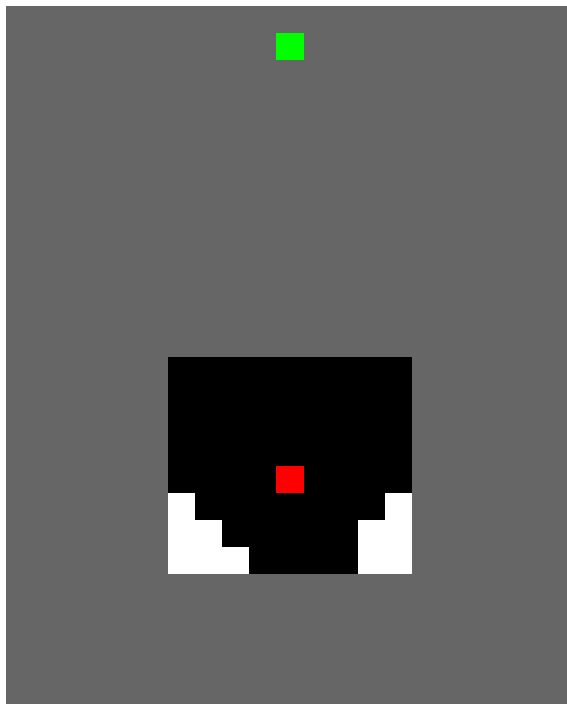


Abbildung 34: Durch den ersten Flug wurde der Status von einigen Nodes um Quax aufgedeckt. Schwarz steht für „Walkable“, Weiß für „Unwalkable“.

Danach wird zuerst der A* Pathfinding Algorithmus normal durchgeführt. Das heißt, dass nur Nodes mit dem Status „Walkable“ auch als tatsächlich begehbar angenommen werden. Die restlichen Nodes mit dem Status „Unknown“ oder „Unwalkable“ gelten als nicht begehbar.

Wird so ein Weg zum Ziel gefunden ist der Algorithmus fertig und es steht fest, dass es einen Pfad zur Stadt gibt. Umgekehrt heißt jedoch nicht, dass wenn kein Weg gefunden wurde es keinen Weg gibt. Stattdessen führt man eine zweite A* Suche durch bei der diesmal nicht nur Nodes des Status „Walkable“, sondern auch die des Status „Unknown“ als begehbar betrachtet werden.

Findet diese Suche wiederum einen Pfad zum Ziel, bedeutet das, dass noch weitere Flüge zur Erkundung dieser „Unknown“-Nodes notwendig sind, um sagen zu können, ob es einen Pfad gibt oder nicht. Findet dagegen auch die zweite Suche keinen Weg zur Stadt, so bedeutet das, dass es auch wirklich keinen Weg gibt.

Abbildung 35 zeigt die notwendige Durchführung weiterer Flüge zur Findung eines Pfades.

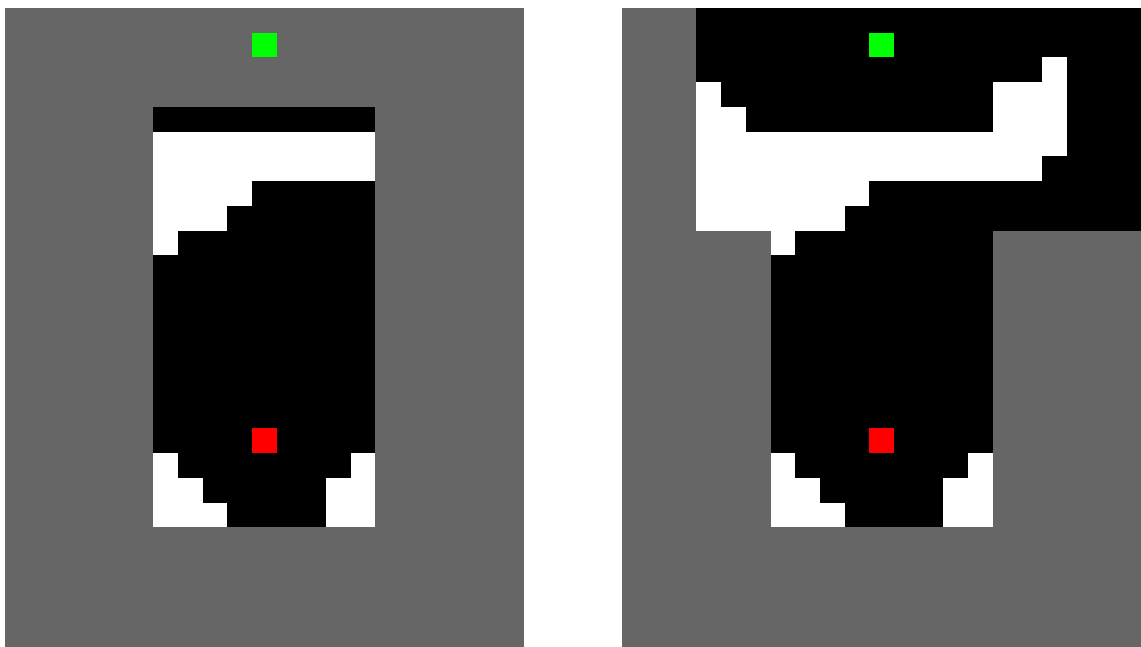


Abbildung 35: Da auch nach einem weiteren Flug (links) noch nicht offensichtlich ist, ob es einen Weg gibt oder nicht, wird ein dritter (rechts) durchgeführt.

Bei der Durchführung des ersten A* Algorithmus wird nun ein Weg aus nur „Walkable“-Nodes gefunden. Damit ist das Pathfinding abgeschlossen. Es wurde ein Weg gefun-

den (siehe Abbildung 36).

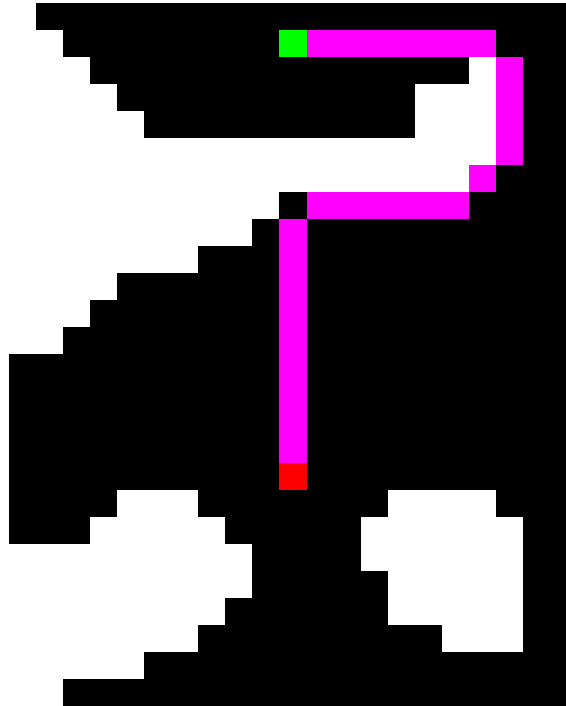


Abbildung 36: Die Vollständige Map. Der Pfad zur Stadt ist pink eingezeichnet.

3.5 Teilaufgabe (b)

Der Schlüssel zu einem möglichst effizienten Algorithmus, welcher nur wenige Quadcopter-Flüge benötigt, ist die Koordinierung zwischen dem A* Pathfinding und der Quadtree Datenstruktur.

Das A* Grid wird nur mit den nötigsten „Informationen versorgt“. Der Quadtree muss darauf vorbereitet sein vom A* Pathfinding aus eine Anforderung für das Scannen eines Bereiches zu bekommen. Dieser Bereich muss dann identifiziert werden und zurück an das Pathfinding System gegeben werden, welches mit dem Resultat das eigene A* Grid aktualisiert und die Wegsuche neu startet.

Ein weiterer nicht zu vernachlässigender Aspekt ist das Umgehen mit dem Sonderfall, dass Quax über ein 20x20 Wasser-Quadrat, durch zwei aneinanderliegende, gemischte 20x20 Quadrate, drüber kommt.

Dieses Problem ist deshalb so kompliziert, weil man bei einem Quadtree nicht davon ausgehen kann, dass sich Quadrate immer überlappen. Bei einem perfekten Quadtree würden sich alle Quadrate nur berühren. Jedoch muss sich ja auch nicht

jedes Quadrat überlappen, sondern nur die 3 20x20 Quadrate bei einem Sonderfall.

Da das A* Grid nun eine Node Größe von 10x10 hat müssen wir unseren Algorithmus zum Aktualisieren des Pathfinding Grids ändern. Der Status des Map-Ausschnitts kann nicht einfach auf alle sich im Ausschnitt befindliche Grid-Nodes übertragen werden. Dies würde den Sonderfall missachten. Stattdessen muss jede 10x10 Grid-Node mit einem eigenen 20x20 Map-Ausschnitt assoziiert werden.

Entspricht der Status jeder einzelnen Grid-Node jeweils dem Map-Status des 20x20 Map-Quadrats, dessen linke, untere Ecke die gleichen Koordinaten hat wie die Grid-Node (siehe Abbildung 37), so entsteht letztendlich ein ein Geflecht aus überlappenden 20x20 Quadraten, welches wiederum den Sonderfall abdeckt (siehe Abbildung 38).

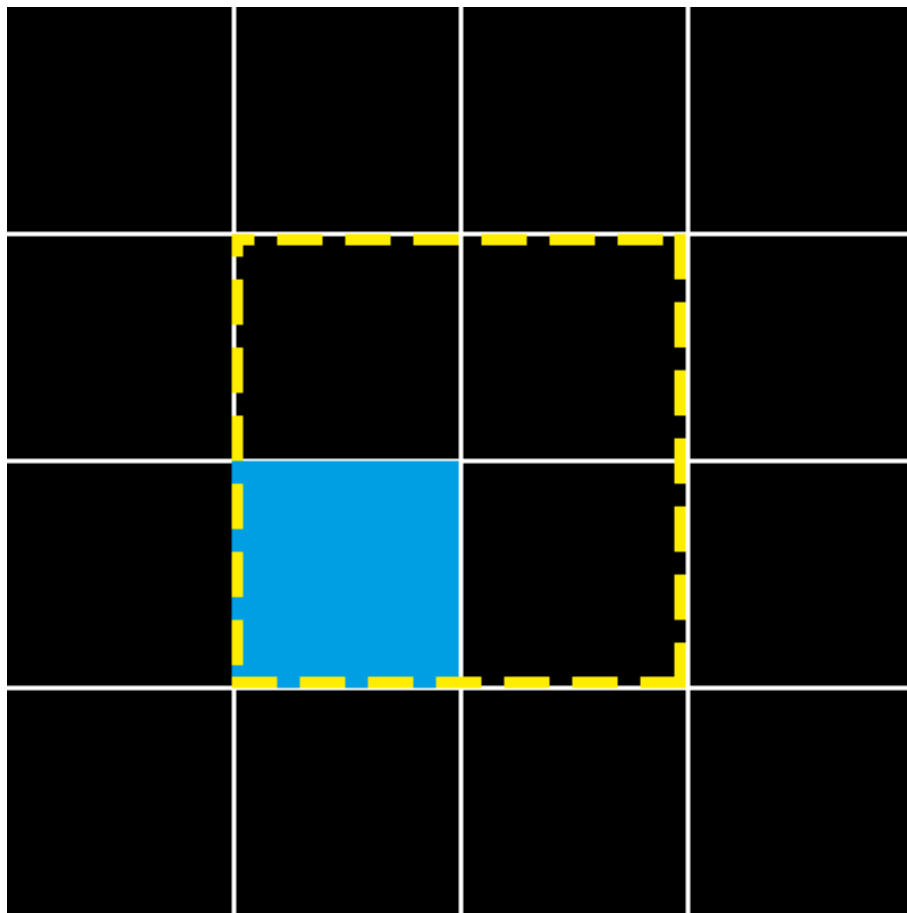


Abbildung 37: Der blaue Pixel repräsentiert eine Node im A*-Grid. Der Status dieser Node wird jedoch über den Map-Typ des gelben Quadrats definiert. Beinhaltet dieses zum Beispiel nur Land, ist die Grid-Node „Walkable“.

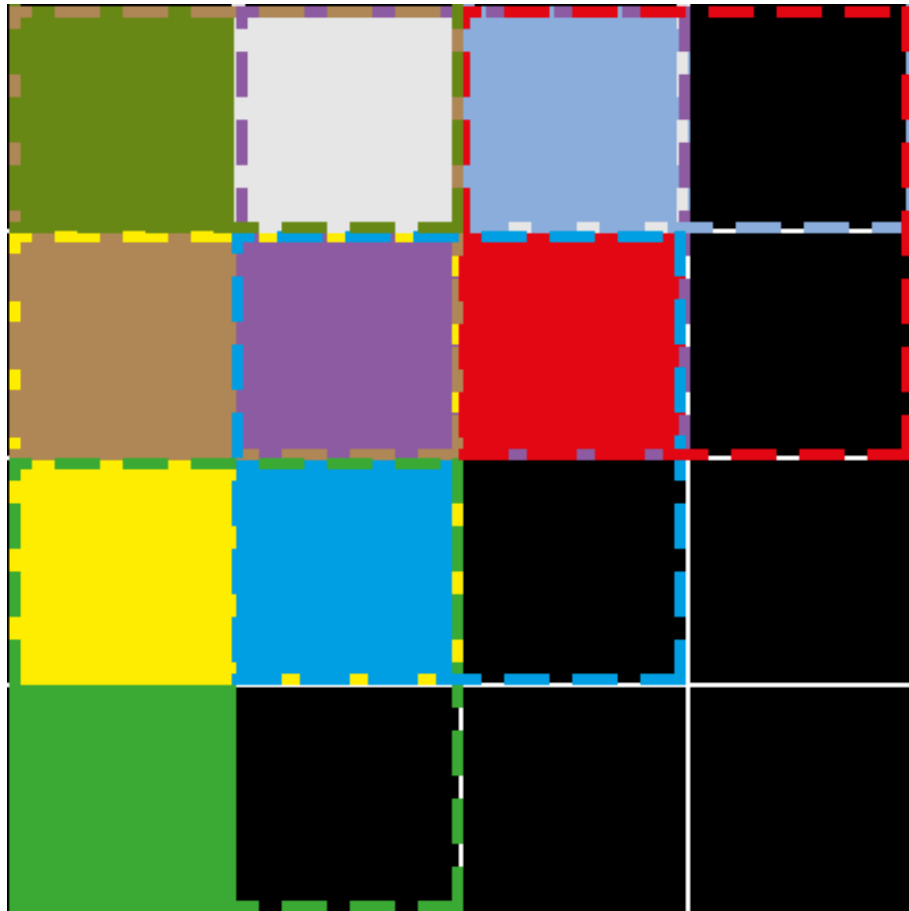


Abbildung 38: Ein Geflecht aus Grid-Nodes und den jeweils dazugehörigen Map-Quadraten. Die 2x2 Map-Quadrate überlappen sich sichern somit den Sonderfall ab.

Die Schnittstelle zwischen den beiden Systemen Pathfinding und Quadtree bildet also eine Methode im Quadtree, die von der Pathfinding Klasse mit einem Bestimmten Punkt im A*-Grid aufgerufen wird, von dem sie den Status will.

Die Methode sucht dann im Quadtree nach dem Punkt und beachtet dabei auch noch nach den zusätzlichen 3 weiteren Punkten zu suchen, so dass man am Ende den Map-Typ oder auch die Map-Typen des mit dem übergebenen Punktes assoziierten Map-Quadrats hat.

3.6 Teilaufgabe (c)

3.6.1 Implementierung

Das Programm wurde in der Game-Engine Unity entwickelt und die Skripte in der Sprache C# mit .NET 3.5 geschrieben.

Auf der beigelegten CD befindet sich eine vollständige Dokumentation aller Klassen in Form einer HTML-Website.

Verwendung einer Game-Engine

Die Game-Engine Unity wurde deshalb als Entwicklungsumgebung gewählt, da sie bereits Klassen und Methoden besitzt, um mit größeren Bildern schnell und effizient zu arbeiten.

Programmier-Stil

Um das Programm außerdem noch möglichst performant zu gestalten, wurde ein System zum Multithreading implementiert. Dieses ermöglicht es bestimmte Prozesse parallel laufen zu lassen und die erhöht die Performance. Allerdings erhöht es auch die Code-Komplexität, da Unity einige Methoden hat, die nur auf dem Main-Thread ausgeführt werden dürfen.

Bei Spielen gilt es allgemein als „Good-Practice“ viel mit **Events** zu arbeiten, um die verschiedenen Systeme möglichst modular miteinander interagieren zu lassen. Diese Philosophie wurde auch bei dieser Implementierung verfolgt, da Events auch gut als Kommunikationsmittel zwischen verschiedenen Threads genutzt werden können.

A* Pathfinding

PathfindingManager ist die Haupt-Klasse des A* Algorithmus. Sie kontrolliert welche Art Pathfinding durchgeführt wird und aktualisiert das A*-Grid.

Im Folgenden soll nur auf die Haupt-Methode (**FindPath**) der **PathfindingManager** Klasse eingegangen werden. Ein Klassendiagramm des gesamten Pathfinding-Systems mit den jeweils wichtigsten Methoden und Attributen ist am Ende des Kapitels in Abbildung ————— abgebildet.

Die **FindPath** Methode ist verantwortlich für das Suchen eines Pfades von der Start-Node zur Target-Node. Der Parameter des Typs **Boolean** gibt dabei an, ob Nodes des Status „Unknown“ auch als „Walkable“ betrachtet werden.

Zuerst (siehe Abbildung 39) werden die für das Pathfinding notwendigen Variablen **openSet** und **closedSet** deklariert. Diese werden je nachdem welche Art Pathfinding vorliegt initialisiert. Außerdem wird dem **openSet** entweder die gespeicherte letzte Path-Node oder die Start-Node hinzugefügt. Dies bestimmt die eigentliche

Start-Node der momentanen Wegsuche. Durch die Fortsetzung des Weges von der letzten „Walkable“-Node des letzten Weges zur Stadt wird Laufzeit eingespart.

```
1 public void FindPath(bool canWalkUnknown)
2 {
3     Heap<Node> openSet;
4     HashSet<Node> closedSet;
5
6     if (!canWalkUnknown)
7     {
8         // Use cached sets
9         openSet = _cachedOpenSet;
10        closedSet = _cachedClosedSet;
11        openSet.Add(_cachedLastPathNode);
12    }
13    else
14    {
15        // Start from the zero
16        openSet =
17            new Heap<Node>(PathfindingGrid.NodeGrid.GetLength(0) *
18                PathfindingGrid.NodeGrid.GetLength(1));
19        closedSet = new HashSet<Node>();
20        openSet.Add(_startNode);
21    }
22 }
```

Abbildung 39: Erste Teil der `FindPath` Methode. Deklaration und Initialisierung wichtige Variablen.

Nun kommt eine bedingte Wiederholung (siehe Abbildung 40), die sich so lange wiederholt bis keine Nodes mehr im `openSet` vorhanden sind. Im Körper der Wiederholung wird zuerst die oberste Node des `openSet` in der Variable `currentNode` gespeichert. Diese Node wird dann aus dem `openSet` entfernt und dem `closedSet` hinzugefügt.

Als nächstes wird geprüft, ob `currentNode` der Target-Node entspricht. Ist dem so wurde ein Pfad zur Target-Node gefunden. Dieser Pfad wird in Zeile 32 in der Variable `path` gespeichert. `path` ist damit eine Liste von Node Objekten.

Nun wird wieder unterschieden zwischen den zwei Pathfinding Arten. Wurden „Unknown“-Nodes als „Unwalkable“ betrachtet, so besteht der Pfad definitiv nur aus „Walkable“-Nodes. Der Algorithmus ist abgeschlossen. Ansonsten wird der Pfad Node für Node nachverfolgt. Sobald man auf die erste „Unknown“-Node stößt, wird die vorherige „Walkable“-Node gespeichert und es wird die Überprüfung der „Unknown“-Node vom Quadtree gefordert.

```

23     while (openSet.Count > 0)
24     {
25         // Remove node from the open set and add it to the closed set
26         var currentNode = openSet.RemoveFirst();
27         closedSet.Add(currentNode);
28
29         if (currentNode == _targetNode)
30         {
31             // Found a path to the target
32             var path = RetracePath(_startNode, _targetNode);
33
34             if (canWalkUnknown)
35             {
36                 // Get the first unknown node in the path
37                 for (var i = 0; i < path.Count; i++)
38                     if (path[i].NodeType == NodeTypes.Unknown)
39                     {
40                         // cache the last node on the path that is walkable
41                         _cachedLastPathNode = i - 1 < 0 ? _startNode :
42                             path[i - 1];
43
44                         // Request map information about the first unknown
45                         // node on the path
46                         if (RequestedMapTile != null)
47                             RequestedMapTile.Invoke(path[i].Position);
48                         break;
49                     }
50             }
51             else
52             {
53                 // Found a valid path from the start to the city
54
55                 /**
56                  * DONE
57                  */
58                 if (FinishedPathfinding != null)
59                     FinishedPathfinding.Invoke(path, true);
60             }
61
62             return;
63         }
64     }

```

Abbildung 40: Zweiter Teil der FindPath Methode. Verarbeiten eines gefundenen Pfads.

Entspricht die `currentNode` nicht der Target-Node, so werden alle angrenzenden Nachbar-Nodes der `currentNode` nach einander durch iteriert. Dabei wird überprüft, ob wird zuerst sichergestellt, dass die Nachbar-Node als „Walkable“ betrachtet wird und dass sie sich noch nicht im `closedSet` befindet.

Ist dem so, wird die G-Cost der Nachbar-Node über die `currentNode` berechnet. Ist diese G-Cost kleiner als die aktuelle G-Cost der Nachbar-Node oder die Nachbar-Node ist noch nicht im `openSet`, werden die Attribute der Nachbar-Node aktualisiert und die Node wird, wenn noch nicht vorhanden, in das `openSet` aufgenommen.

Wurden alle Nodes im `openSet` überprüft und trotzdem kein Weg zur Target-Node gefunden, bedeutet das, dass es für den momentanen Durchlauf keinen Pfad zur

Target-Node gibt. Hier wieder abermals zwischen den zwei Pathfinding-Arten unterschieden. Wurden auch „Unknown“-Nodes als „Walkable“ betrachtet und eben trotzdem kein Weg gefunden, bedeutet das, dass es auch keinen Weg zur Target-Node gibt. Der Algorithmus ist abgeschlossen.

Im anderen Fall wird eben dann die zweite Pathfinding Variante gestartet.

```

65     /**
66      * Check all neighbours
67      */
68     foreach (var neighbour in PathfindingGrid.GetNeighbours(currentNode))
69     {
70         // Is the neighbour important
71         if (!neighbour.IsWalkable(canWalkUnknown) || closedSet.Contains(neighbour))
72             continue;
73
74         // Calculate neighbours G-Cost
75         var newMovementCostToNeighbour = currentNode.GCost + GetDistance(currentNode, neighbour);
76         if (newMovementCostToNeighbour < neighbour.GCost || !openSet.Contains(neighbour))
77         {
78             // Update the neighbour
79             neighbour.GCost = newMovementCostToNeighbour;
80             neighbour.HCost = GetDistance(neighbour, _targetNode);
81             neighbour.Parent = currentNode;
82
83             // Add neighbour to the open set if it's not already in there
84             if (!openSet.Contains(neighbour))
85                 openSet.Add(neighbour);
86             else
87                 openSet.UpdateItem(neighbour);
88         }
89     }
90 }
91
92 // Unable to find a path
93 if (canWalkUnknown)
94 {
95     // There is no path
96     if (FinishedPathfinding != null)
97         FinishedPathfinding.Invoke(null, false);
98 }
99 else
100 {
101     // Try again with unknown nodes as walkable
102     _tryPathfinding02 = true;
103 }
104 }
```

Abbildung 41: Dritter Teil der FindPath Methode. Suche nach einem Pfad und Behandeln des Falls, dass kein Pfad gefunden wurde.

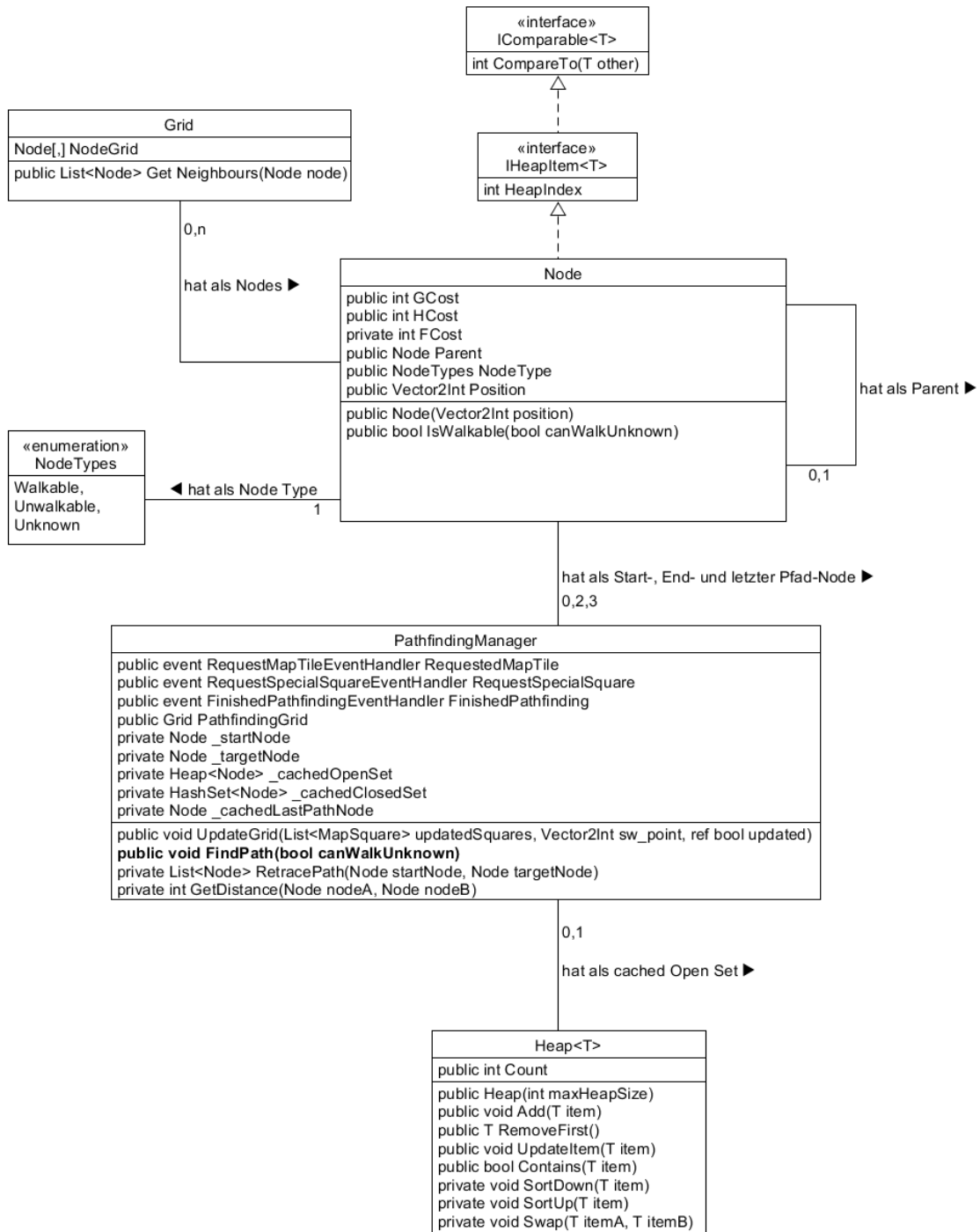


Abbildung 42: Erweitertes Klassendiagramm der wichtigsten Pathfinding Klassen. Die Haupt-Methode des A* Algorithmus wurden in der Klasse **PathfindingManager** hervorgehoben.

Quadtree

3.6.2 Optimierungsmöglichkeiten

Die Algorithmuslaufzeit der aktuellen Implementierung ist gerade für größere Maps sehr schlecht. Dabei überrascht, dass die einzelnen Komponente eigentlich gar nicht viel Zeit verbrauchen. Das Suchen im Quadtree, das Aktualisieren des A* Grids und die erste Wegsuche verbrauchen alle zusammen nur wenige Millisekunden pro Durchgang.

Das was am meisten Zeit verbraucht ist die zweite Wegsuche. Nicht zuletzt, weil sie noch nicht durch Caching optimiert ist. Allerdings erklärt auch die Laufzeit des zweiten Pathfinding-Algorithmus nicht die Länge der Gesamtlaufzeit. Die Wahl einer Game-Engine als Entwicklungsumgebung hat die Performance letztendlich durch Hintergrundprozesse doch eher um einiges verschlechtert als verbessert.

Mögliche Optimierungsmöglichkeiten zur Verbesserung der Laufzeit oder Verringerung der benötigten Quadcopter Flüge wären:

Multithreading

Nicht alle Bestandteile des Algorithmus nutzen das Prinzip des Multithreadings optimal aus und können noch erheblich verbessert werden.

A* Pathfinding

A* ist ein Pathfinding-Algorithmus zum Finden eines kürzesten oder zumindest möglichst kurzen Weges. Jedoch ist für diese Aufgabenstellung die Länge des Weges irrelevant und viele der Quadcopter Flüge zum Erkunden sehr kleiner Gebiete sind nicht unbedingt notwendig. Um dieses Verhalten des A*-Algorithmus zumindest abzuschwächen, könnte man noch weitere Heuristiken implementieren.

Speichern von Pathfinding Zuständen

Das bereits implementierte Caching für eine Wegfindung, bei der noch unbekannte Nodes als nicht-betretbar angenommen werden, hat die Laufzeit erheblich minimiert. Nun ist es aber noch möglich Zustände für die Wegfindung, bei der unbekannte Nodes als betretbar angenommen werden, zu speichern, so dass dieser Algorithmus nicht jedes mal von der Startposition bis zur Zielposition suchen muss, sondern auf vorher durchgeführte Suchergebnisse zurückgreifen kann.

Verbindung beider A* Suchen

Eine Alternative zum Speichern des aktuellen Zustandes der zweiten A* Suche, wäre auch eine Integration dieser in die schon vorherige A* Suche. Indem man alle unbekannten Nachbar-Nodes in einem zweiten **openSet** speichert und nach fehlgeschlagener ersten Suche eine Zweite mit dem zusätzlich erstellten **openSet** und der Bedingung, unbekannte Nodes begehbar sind, lässt einiges an Laufzeit einsparen.

Bidirektionale Suche

Mit Hilfe einer „Bidirektionale Suche“, könnte sich sowohl die Laufzeit, als auch die benötigte Anzahl an Quadcopter Flügen weiter verbessern.




Unity Implementierung

Die Wahl einer Game-Engine als Entwicklungsumgebung hat auch einen sehr großen Einfluss auf die Laufzeit, durch viele für dieses Programm unnötige Hintergrundprozesse. Auch das einzelne Darstellen von Flügen auf der Map-Textur verbraucht viele Ressourcen. Die Laufzeit des puren Algorithmus ist daher wesentlich geringer als die Zeit, die das Programm für die fertige Abbildung des Weges benötigt.

Eine simplere Konsolen-Anwendung, die dem Benutzer die Flüge am Ende in eine PNG-Datei einträgt, würde zum Beispiel nochmals deutlich effizienter laufen.

3.6.3 Beispiele

3.7 Teilaufgabe (d)

Mit einer eher unnatürlichen Verteilung von Flüssen und Seen wie zum Beispiel einem Schachbrettmuster (siehe Abbildung ) , einem verzweigten Labyrinth (siehe Abbildung ) oder eine Anordnung mehrerer schmaler Wasser und Land streifen (siehe Abbildung ) , können kaum noch größere Flächen eindeutig eingeteilt werden.

Dies sorgt schnell für einen sehr umfangreichen Quadtree und damit auch für viele Quadcopter Flüge.

Der hier besprochene Algorithmus läuft am effizientesten, bei eher natürlichen Landschaften, mit größeren Wasser- und Landflächen. Dabei kann der Quadcopter den Typ größere Gebiete mit nur einem Flug identifizieren.

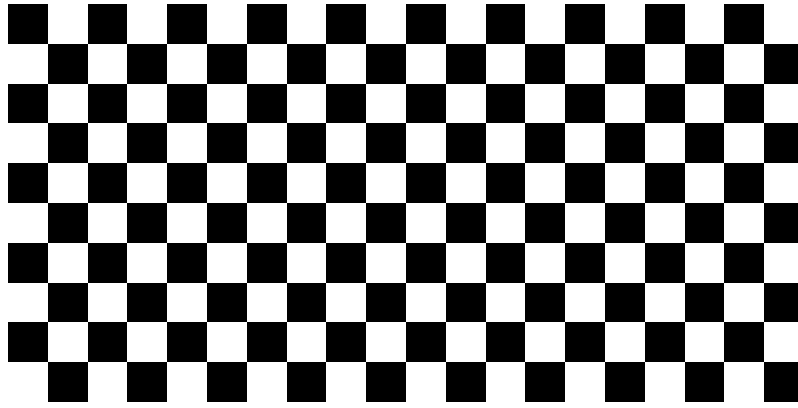


Abbildung 43: Schachbrettartige Landverteilung

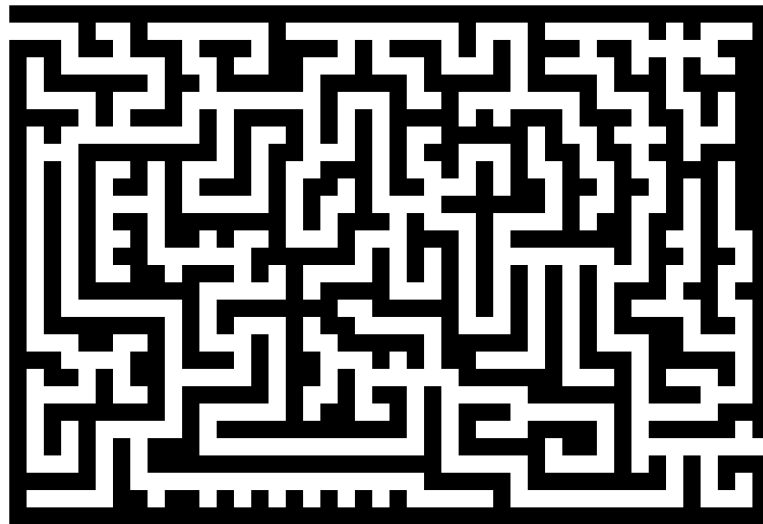


Abbildung 44: Irrgarten ähnliche Landverteilung

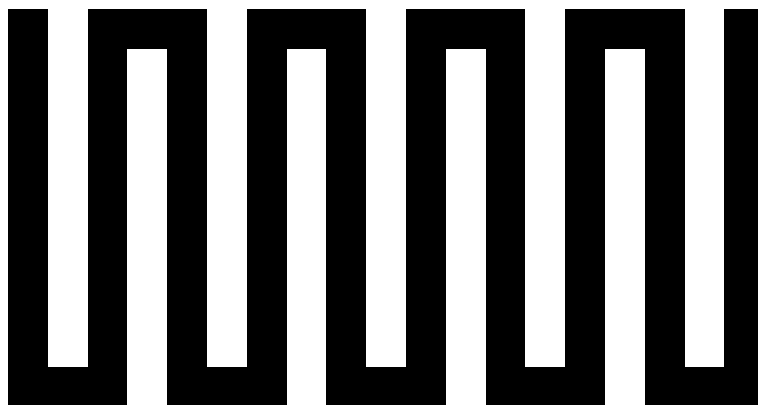


Abbildung 45: Linienartige Landverteilung

4 FAZIT

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed tempus consectetur lorem, imperdiet dignissim est auctor a. Vivamus convallis, leo et iaculis egestas, nunc massa porttitor tellus, id faucibus urna justo eget massa. Praesent quis feugiat odio. Nullam quis mattis enim. Fusce volutpat odio in enim sodales venenatis. Mauris consequat.

5 LITERATURVERZEICHNIS

Buchquellen:

- [1] Leslie Lamport, *TEX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.
- [2] Leslie Lamport, *TEX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.
- [3] Leslie Lamport, *TEX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.

PDF-Scans:

- [1] Leslie Lamport, *TEX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.
- [2] Leslie Lamport, *TEX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.

Internetquellen:

- [1] Leslie Lamport, *TEX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.
- [2] Leslie Lamport, *TEX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.
- [3] Leslie Lamport, *TEX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.

6 ABBILDUNGSVERZEICHNIS

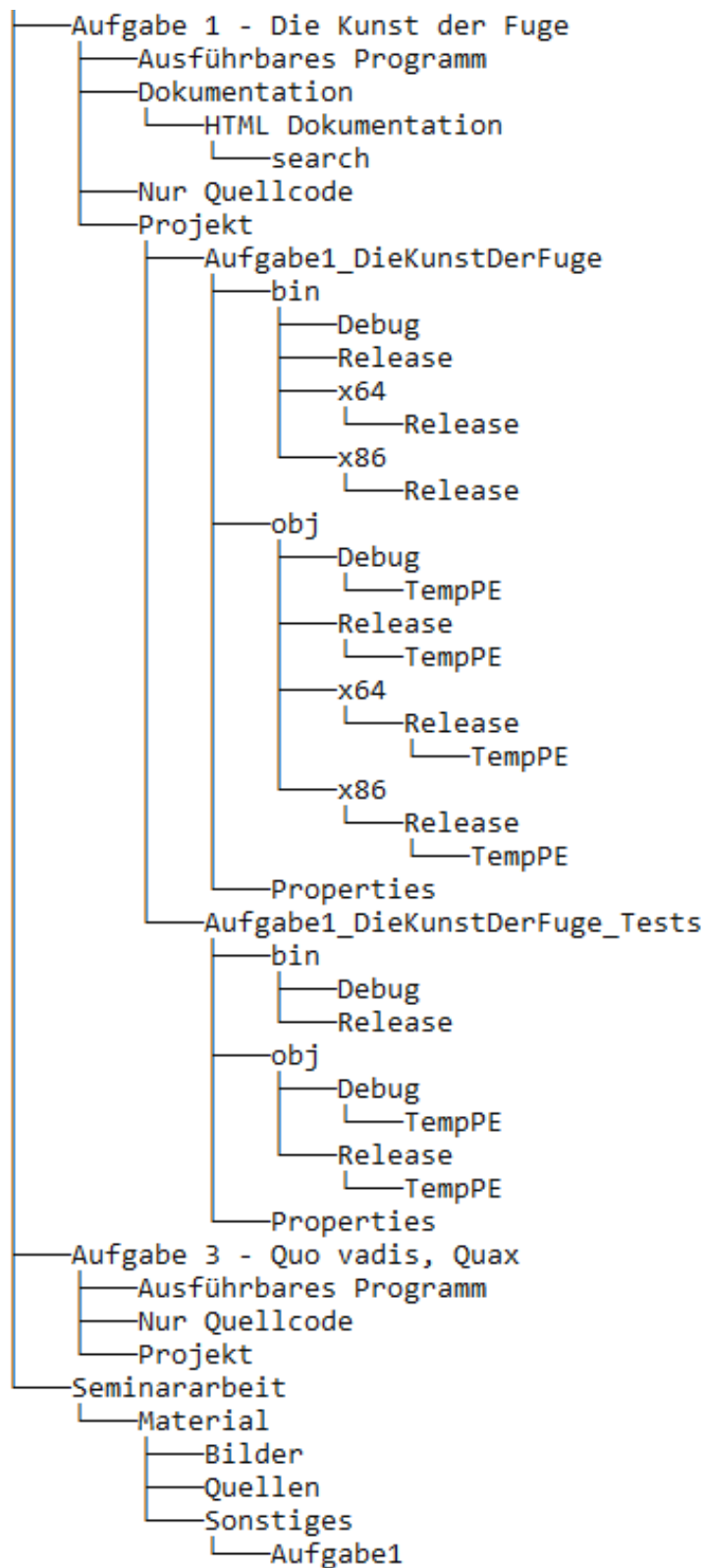
Die meisten enthaltenen Abbildungen wurden entweder selbst mit unten aufgelisteten Tools erstellt oder von unten genannten Programmen/Quellen aufgenommen.

- Microsoft PowerPoint
- Adobe Photoshop CC 2018
- UMLet
- Grafikrechner - GeoGebra
- BwInf36 Runde 2 Aufgabenstellung
- Windows 10 Konsole

Sonstige Bildquellen:

- <https://upload.wikimedia.org/wikipedia/commons/d/df/ZweiWegeIrrgarten.png>
- https://de.wikipedia.org/wiki/Datei:Pathfinding_A_Star.svg

7 CD-VERZEICHNIS



8 ANHANG

Aufgrund der Länge des Quellcodes beider Aufgaben, werden die einzelnen Klassen hier nicht weiter aufgelistet.

Die einzelnen Quellcode-Dateien befinden sich auf der beigelegten CD.

8.1 Installationshinweise

Die Programme beider Aufgaben wurden kompiliert und befinden sich als ausführbare .exe Dateien auf der beigelegten CD.

Für beide Aufgaben wurden jeweils für Windows 64bit und 86bit Dateien erstellt.

9 ERKLÄRUNG DES VERFASSERS

Ich erkläre hiermit, dass ich die Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benützt habe.

....., den
Ort Datum

.....
Unterschrift des Verfassers