dokumentation

36 Bundeswettbewerb Informatik  
2. Runde / Aufgabe 1 und 3

# Aufgabe 1: Die Kunst der Fuge

## Lösungsweg

Nachdem ich mich zuerst mathematisch mit der Aufgabe auseinandergesetzt hatte, hatte ich recht schnell die Formeln, um weitere Eigenschaften der Mauer in Abhängigkeit von N (Anzahl der Klötzchen) herzuleiten:

Danach habe ich sehr viele unterschiedliche Strategien ausprobiert, um eine Mauer mit N Klötzen direkt zu konstruieren. Jedoch hat kein Algorithmus für ein N größer als 7 funktioniert. Nach einiger Zeit habe ich im Internet nach ähnlichen Problemstellungen gesucht aber auch da nichts gefunden, was mir weiterhalf. Daraufhin wollte ich erstmal mit einem Programm alle möglichen Mauern der maximalen Höhe brute-forcen und ausgeben lassen, da ich hoffte in einer der ausgegebenen Mauer ein Muster zu erkennen, aus dem ich einen Algorithmus ableiten konnte. Allerdings war meine Brute-force Algorithmus für N = 10 bezüglich Rechenzeit und Speicherplatz trotz Optimierungen zu komplex und so entschloss ich mit letztendlich den Brute-force Algorithmus so abzuwandeln, dass er mir nicht alle möglichen fertigen Mauern ausgibt, sondern abbricht sobald er eine Mauer gefunden hat.

## Umsetzung

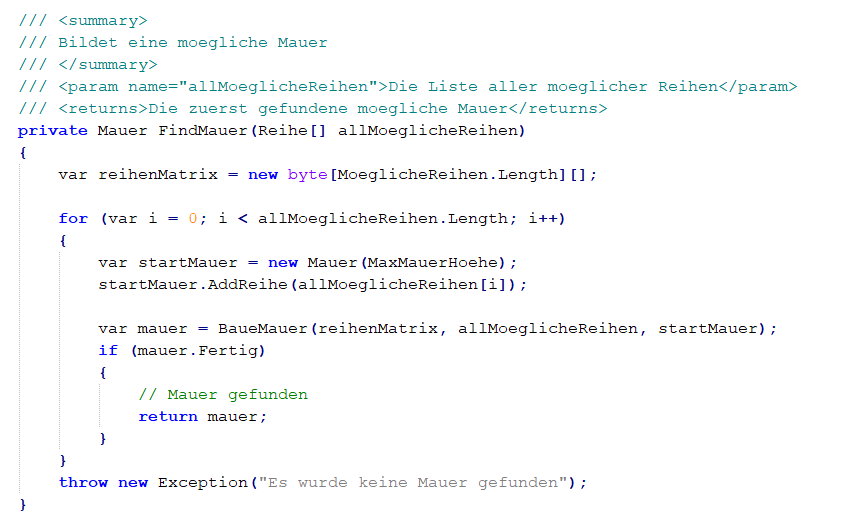
Das Programm ist eine Konsolenanwendung, geschrieben in C# mit Visual Studio. Zu Beginn wird der Anwender aufgefordert N zu definieren. Die Eingabe muss eine Zahl zwischen 2 und 22 sein. Bei 1 kann eine unendlich hohe Mauer gebildet werden, da eine Reihe mit nur einem Baustein keine Fugen besitzt und die Obergrenze von N = 22 ist durch die Wertebereiche mancher Datentypen bedingt. Kommt es zu einer ungültigen Eingabe, wird der Anwender erneut aufgefordert eine Zahl einzugeben.

Wurde eine gültige Zahl für die Anzahl der Klötzchen definiert, werden zunächst in der Methode BerechneEigenschaften der Klasse WallBuilder die Mauer Breite, die Anzahl der möglichen Fugenstellen, die maximal mögliche Mauerhöhe und die Anzahl der Fugen, die bei einer Mauer der maximalen Höhe benutzt werden müssen, berechnet.

Daraufhin wird die Methode StartAlgorithmus der Klasse WallBuilder ausgeführt. In dieser wird als erstes eine Liste aller Zahlen bis einschließlich N erstellt, die im nächsten Schritt für die Erstellung aller Permutationen verwendet wird. Die Erstellung der Permutationen erfolgt über eine optimierte Implementierung des Heaps Algorithmus. Die Methode SammlePermutationen der Klasse Utilities enthält als zweiten Parameter einen Delegaten der ein Byte Array als Übergabeparameter hat und einen Boolean zurückgibt. Somit kann nach jeder einzelnen Permutation über den Rückgabewert der referenzierten Methode entschieden werden, ob weitere Permutationen gesucht werden sollen. Im aktuellen Fall werden jedoch immer alle Permutationen gesammelt, sodass man am Ende ein Array aus Byte Arrays hat, wobei jedes Byte Array eine Permutation darstellt.

Für jede Permutation wird nun eine Reihe Instanz erstellt, die die Folge der Klötze als Byte Array speichert und in einem HashSet vom Typ Byte die Indizes der Fugen beinhaltet, die diese Reihe besetzt. Nachdem jede Reihe über das MoeglicheReihen Array referenziert wurde, beginnt der Mauerbau.

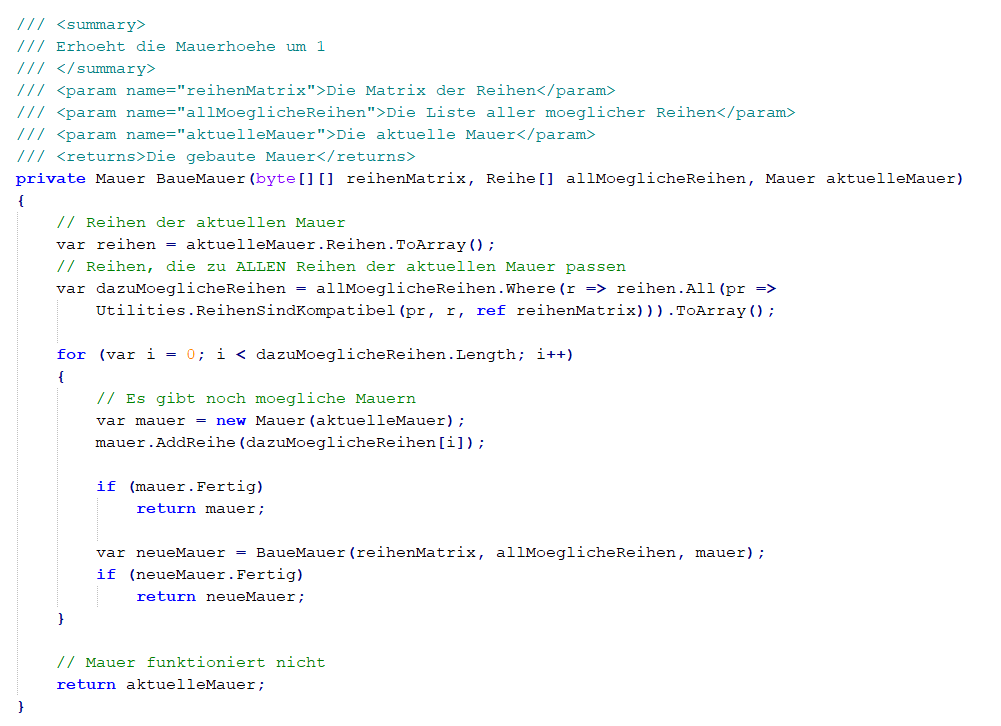
Zuerst wird die Methode FindMauer (siehe Abbildung 1) in der Klasse WallBuilder ausgeführt und ihr wird das Array aller Permutations-Reihen übergeben. Die Methode erstellt eine neue Variable reihenMatrix die ein verzweigtes Byte Array ist. In dieser Matrix wird eingetragen, ob zwei Reihen kompatibel sind (Keine Fugen überlappen sich). Ist dies der Fall hat das Feld der beiden Reihen den Wert 2. Sind die zwei Reihen nicht kompatibel bekommt das Feld den Wert 1. In die Matrix wird jedoch nur ein Wert eingetragen, wenn zwei Reihen auch verglichen werden.



Abbildung

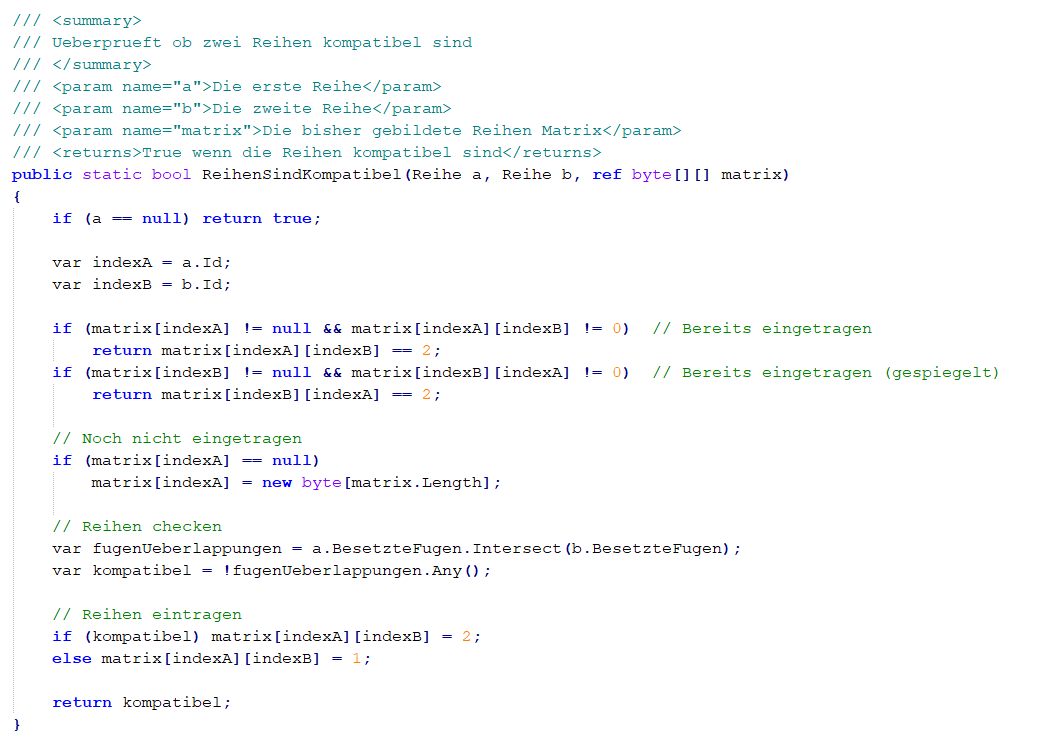
Der Brute-force Algorithmus beginnt indem er in einer Schleife alle möglichen Reihen durchgeht. Aus der jeweiligen Reihe wird dann eine Mauer startMauer erstellt. Die Klasse Mauer hat ein Array vom Typ Reihe, dass die Reihen der Mauer referenziert, ein HashSet vom Typ Byte, dass die Indizes der besetzten Fugen in der gesamten Mauer beinhaltet und einen Boolean Fertig der angibt, ob alle Reihen der Mauer gesetzt sind. Für die gebaute Mauer wird dann die Rekursive Methode BaueMauer aufgerufen.

Die Methode BaueMauer (siehe Abbildung 2) sammelt zuerst alle Reihen, die zur übergebenen Mauer passen und erstellt dann wiederrum für jede dazu passende Reihe eine neue Mauer Instanz mit den gleichen Reihen wie der übergebenen Mauer und fügt dieser Mauer die jeweilige passende Reihe hinzu. Ist die erstellte Mauer nun fertig, sprich alle ihre Reihen sind gesetzt, wurde eine Mauer der maximalen Höhe gefunden und diese Mauer kann zurückgegeben werden. Ansonsten ruft sich die Methode selber auf und übergibt die neu erstellte Mauer.



Abbildung

Die Methode ReihenSindKompatibel (siehe Abbildung 3) überprüft ob zwei Reihen kompatibel sind. Als erstes wird überprüft, ob es schon einen Eintrag für beide Reihen gibt. Gibt es noch keinen Eintrag wird überprüft ob beide Reihen eine Fuge mit dem gleichen Index in ihrem BesetzteFugen HashSet. Ist dies der Fall sind die Reihen nicht kompatibel.



Abbildung

Zum Schluss wird die gefundene Reihe noch in der Konsole ausgegeben.

Allerdings, kann selbst dieser Brute-force Algorithmus keine Mauer für N = 10 finden. Optimierungsmöglichkeiten des Algorithmus wären das Bilden einer Permutationsreihe erst, wenn sie gebraucht wird oder auch Multithreading.

Auch wage ich zu Vermuten, dass dieses Problem nicht effizient gelöst werden kann und somit als NP-Schwer gilt.

## Beispiele:

|  |  |
| --- | --- |
| N = 6  Maximale Höhe: 4  Laufzeit: Ø 30ms  Verbrauchter Speicherplatz: Ø 10MB |  |
| N = 8  Maximale Höhe: 5  Laufzeit: Ø 64.000ms  Verbrauchter Speicherplatz: Ø 70MB |  |
| N = 9  Maximale Höhe: 5  Laufzeit: Ø 10.000ms  Verbrauchter Speicherplatz: Ø 165MB |  |
| N = 10  Maximale Höhe: 6  Laufzeit: ?  Verbrauchter Speicherplatz: > 2GB |  |

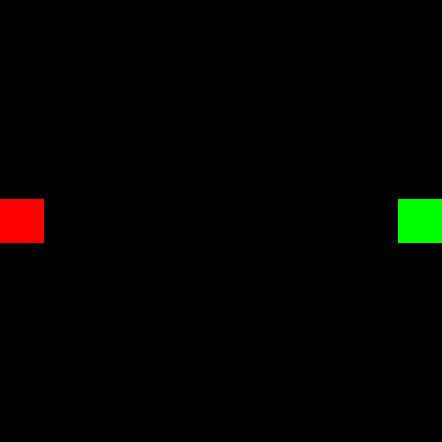
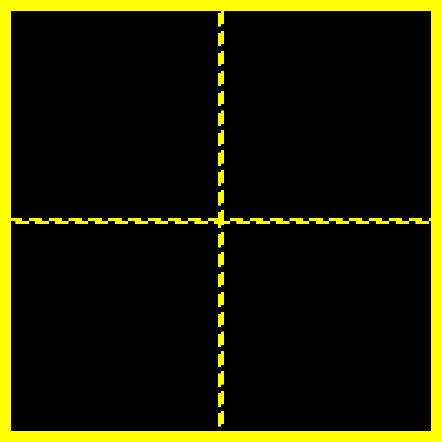
# Aufgabe 3: Quo vadis, Quax?

## Teilaufgabe (b)

### Lösungsweg:

Um die möglichst wenige Quadrate zu scannen überlegte ich mir, dass die Drohne zuerst ein möglichst großes Gebiet scannen muss und dann dieses Gebiet, wenn es gemischt ist, in kleinere Gebiete zerlegen muss, die dann wiederrum gescannt werden. Das heißt im einfachsten Fall, einer quadratischen Map ohne Wasser (siehe Abbildung 4), bräuchte es nur einen Flug, um den Weg von Quax zur Stadt zu finden.

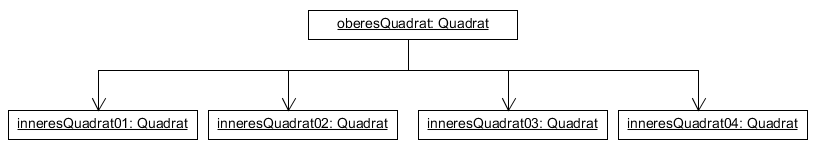
Wenn ein gescanntes Gebiet gemischt ist, ist es am effizientesten das Gebiet in 4 gleichgroße Teilgebiete aufzuteilen (siehe Abbildung 5).



Abbildung

Die daraus resultierende Datenstruktur (siehe Abbildung 6) heißt Quadtree oder auch Quaternärbaum.

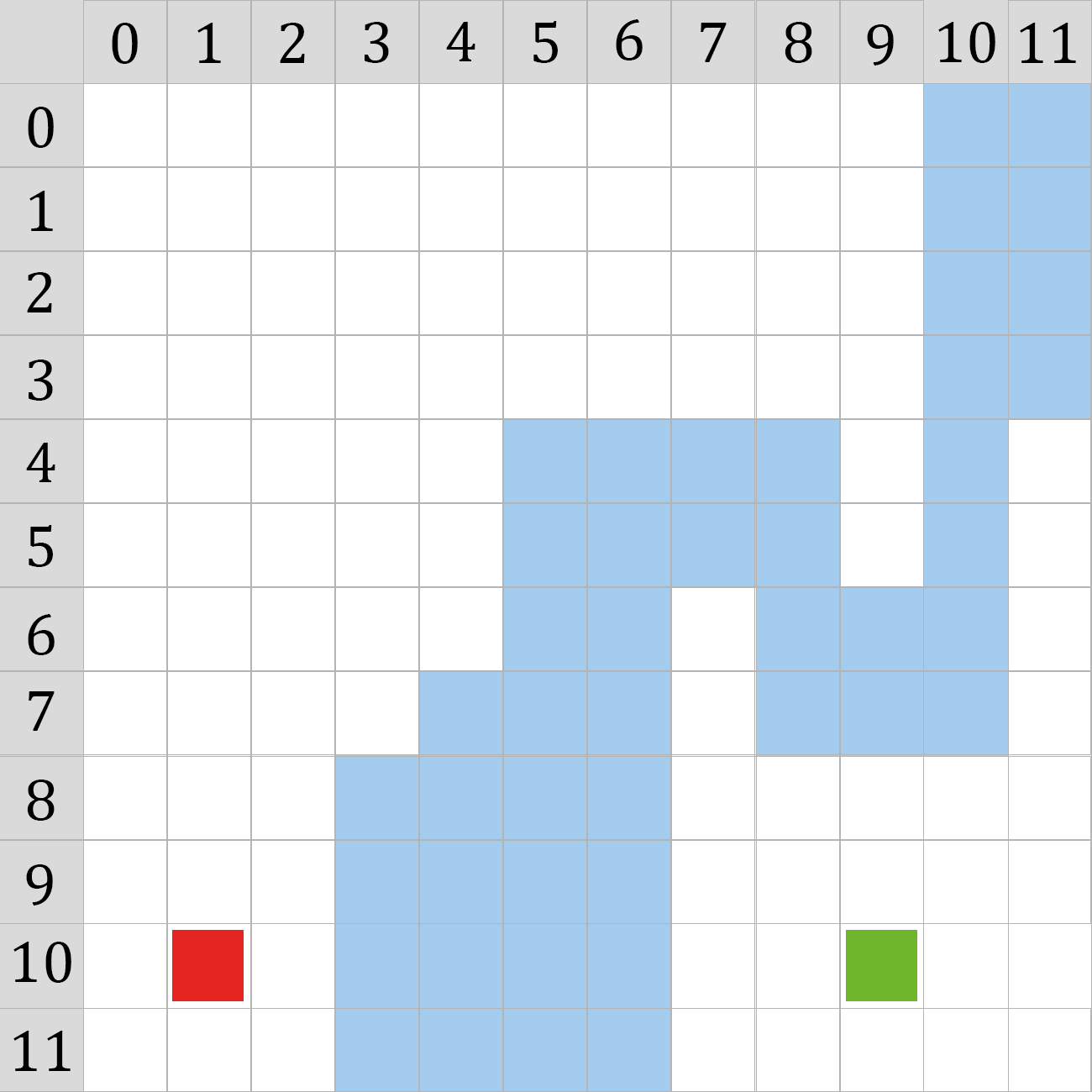
Abbildung



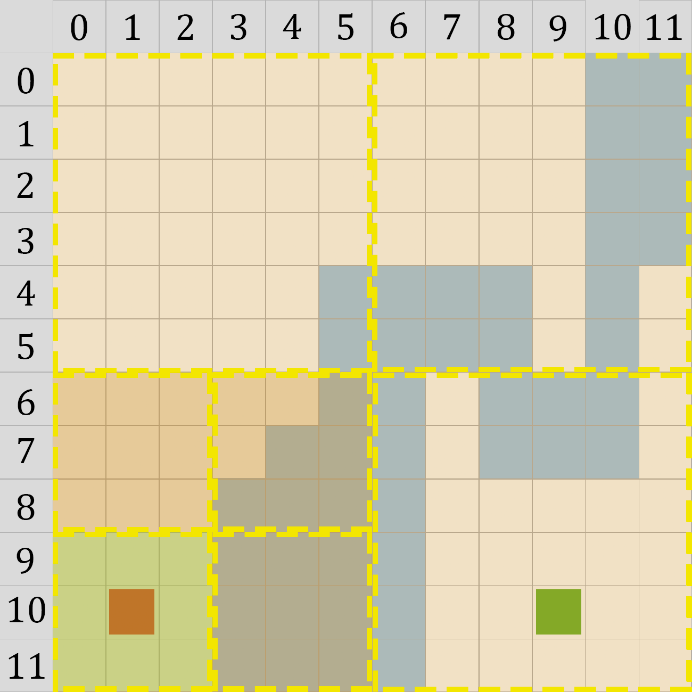
Abbildung

Nun wollte ich diese Datenstruktur für meinen Algorithmus verwenden. Allerdings war der Start anders als bei anderen Pathfinding Algorithmen. Während andere Algorithmen schon beim Ausgangspunkt, also hier die Position von Quax, anfangen, fängt mein Algorithmus erst beim größtmöglichen Quadrat an. Das heißt also, dass mein Quadtree als erstes Quax suchen muss. Das geht noch relativ einfach, indem man einfach so lange das Quadrat indem Quax sich befindet in innere Quadrate aufteilt, bis das Quadrat das Quax enthält Land ist, oder gemischt und eine Größe von 20x20 hat. Hierbei ist anzumerken, dass alle Pixel, die nicht die Farbe Weiß (#FFFFFF) besitzen, als Land gewertet werden.

Das Start Quadrat der Suche ist dann das ober Quadrat des gefundenen Quadrates. Nun muss der Weg vom Start Quadrat zur Stadt gefunden werden. Um nun diesen Algorithmus zu finden, hatte ich mir ein kleineres Beispiel (siehe Abbildung 7) überlegt, nach welchem ich den Algorithmus entwerfen konnte.



Abbildung

Zuerst wird die Suche nach Quax wie oben geschildert durchgeführt. Das Ergebnis mit eingezeichnetem Quadtree sehen Sie in Abbildung 8.

Ein gemischtes Quadrat hat eine hellbraune Füllfarbe, ein passierbares Quadrat (Nur Land oder gemischt und größer als 20x20) eine grüne und ein nicht passierbares Quadrat ist rot.

Das Start Quadrat des Pathfindings zur Stadt ist hierbei dann das Quadrat mit den Eckpunktkoordinaten (0 | 6) links oben und (5 | 11) rechts unten. Das Quadrat (0 | 9), (5 | 11) ist nun das erste Quadrat des End Weges.

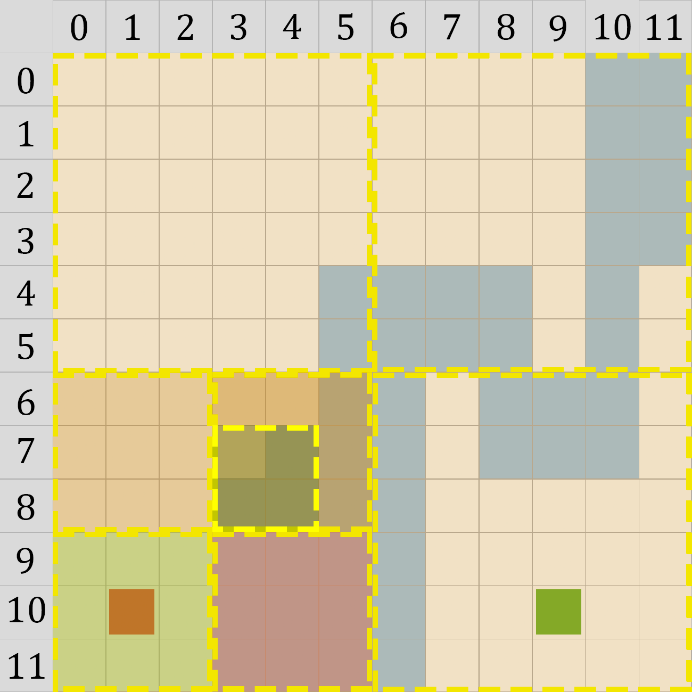
Nach meiner Strategie, wird nun das innere Quadrat überprüft, dass das letzte Quadrat des End Weges berührt und am nächsten zur Stadt ist.

Dies wäre in diesem Fall das Quadrat (3 | 9), (5 | 11), welches jedoch nicht passierbar ist. Deswegen wird das zweit nächste Quadrat (3 | 6), (5 | 8) untersucht. Dieses ist gemischt und größer als 20x20, was heißt, dass es in innere Quadrate zerlegt wird. Da es jedoch keine 1,5 Pixel große Quadrate geben kann, wird jedes Quadrat auf eine Breite von 2 aufgerundet.

Abbildung

Nun beginnt der Algorithmus wieder von vorne und da es nur ein Quadrat gibt, das den letzten Weg berührt, wird dieses überprüft. Das Quadrat (3 | 7), (4 | 8) ist hierbei passierbar, da Quax laut Aufgabenstellung ein gemischtes Quadrat der Größe 20x20 passieren kann.

Abbildung 9 zeigt den aktuellen Stand des Quadtrees.



Abbildung

Nun brauchen wir aber noch ein weiteres inneres Quadrat vom Quadrat (3 | 6), (5 | 8), das passierbar ist. Hat ein Quadrat nämlich mindestens zwei passierbare, innere Quadrate, so ist dieses Quadrat passierbar. Hier gibt es auch Sonderfälle, auf die ich in Teilaufgabe (c) zu sprechen komme.

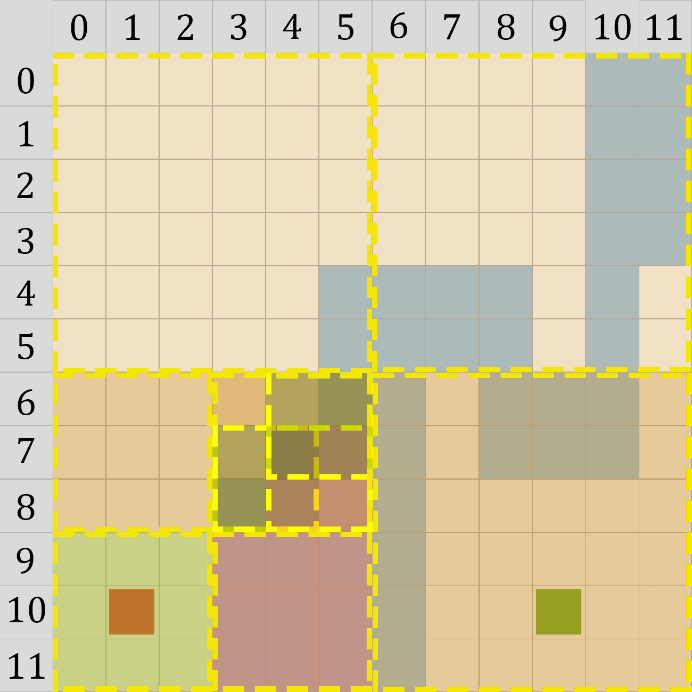
Das Quadrat (3 | 6), (5 | 8) besitzt bisher nur ein passierbares inneres Quadrat, also muss das nächste innere Quadrat untersucht werden. Das Quadrat (4 | 7), (5 | 8) ist hierbei jedoch wieder nicht passierbar und erst das dritte innere Quadrat (4 | 6), (5 | 7) ist passierbar.

Nun hat das Quadrat (3 | 6), (5 | 8) zwei passierbare innere Quadrate. Somit ist dieses Quadrat passierbar. Außerdem ist eben das Quadrat (0 | 9), (2 | 11) passierbar, was bedeutet, dass das ober Quadrat (0 | 6), (5 | 11) passierbar ist.

Dies bedeutet, dass wir wieder ein Quadrat im Quadtree zurückgehen und beim größten Quadrat (0 | 0), (11 | 11) landen.

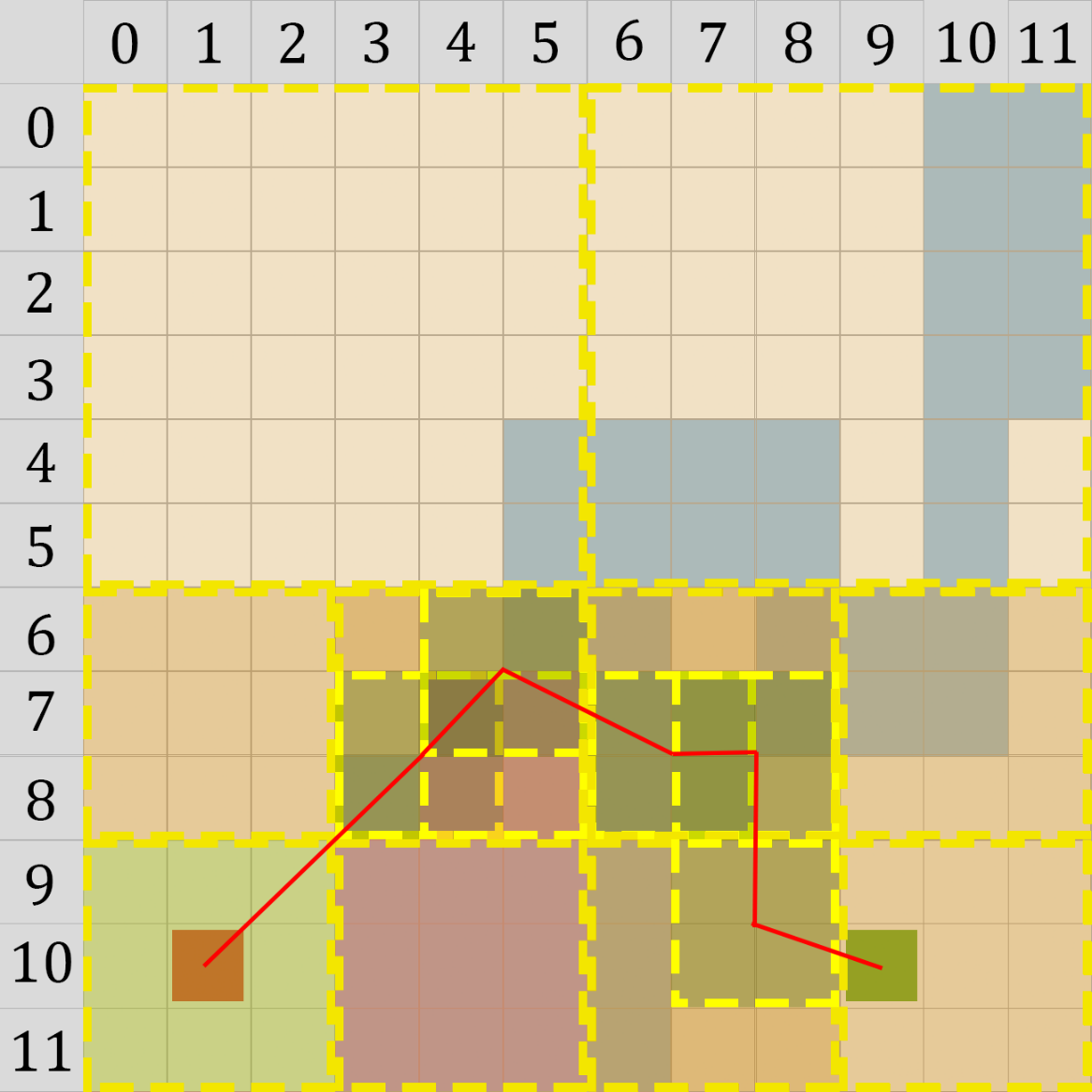
Dieses Quadrat hat nun ein passierbares inneres Quadrat. Folglich wird wieder das innere Quadrat überprüft, dass den letzten Weg berührt (Das Quadrat (4 | 6), (5 | 7) ist das zuletzt zum Weg hinzugefügte Quadrat) und am nächsten zur Stadt ist. Dies ist das Quadrat (6 | 6), (11, 11). Dieses ist wieder gemischt und es müssen die inneren Quadrate geprüft werden.

Abbildung 10 zeigt den aktuellen Stand des Quadtrees.



Abbildung

Der Algorithmus geht nun immer so weiter, bis zum passierbaren Quadrat (7 | 9), (8 | 10), welches die Stadt berührt. Damit ist das Pathfinding abgeschlossen und die gesammelten Quadrate für den Weg ergeben den Pfad von Quax zur Stadt (siehe Abbildung 11).



Abbildung

### Spezialfälle:

#### Nicht quadratische Map

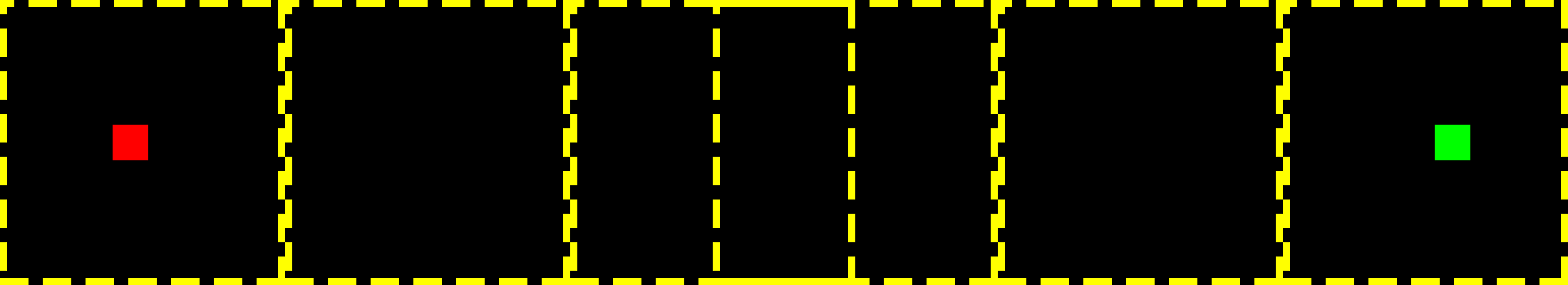
Alle BwInf Beispiele sind nicht quadratische Bilder. Dies ist deshalb ein Problem, da Quax nur quadratische Bereiche scannen kann. Mir sind zwei mögliche Lösungen für dieses Problem eingefallen:

1. Ich erweitere die Map zu einem Quadrat, indem ich den hinzugefügten Bereich mit Wasser fülle, wie es in der Aufgabenstellung festgelegt ist.
2. Ich definiere mehrere Start Quadrate, die so groß wie möglich, das Rechteck füllen (siehe Abbildung 12).

Ich habe mich für die zweite Lösung entschlossen, da ich glaube, dass diese weniger zusätzliche Quax Flüge benötigt, als wenn ich die Map mit Wasser erweitere und somit ein mögliches Land Quadrat in ein Gemischt Quadrat umwandle.

Die zweite Lösung erfordert jedoch auch nochmal einen extra Algorithmus, da ich damit nichtmehr nur einen Quadtree pro Map habe, sondern N Quadtrees wobei N die Anzahl der Start Quadrate ist. Diese N Quadtrees, sind im Worst-case jedoch alle am Pathfinding beteiligt. Auch ist es wahrscheinlich, dass die Start Quadrate sich über mehrere Pixel überschneiden.

Der Algorithmus für ein Pathfinding über mehrere Quadtrees hinweg, ist auch nicht allzu komplex. Bei der Quax Suche, wird der erste Quadtree ausgewählt, der die Quax Position enthält. Bei diesem wird dann auch das Pathfinding zur Stadt ausgeführt, bis das Start Quadrat fertig gescannt ist. Dann wird das Start Quadrat als nächstes gewählt, welches den letzten Weg berührt und am nächsten zur Stadt ist.



Abbildung

## Teilaufgabe (c)

## Teilaufgabe (d)

Rückblickend war es wohl ein Fehler, den Algorithmus basierend auf nur einer einzigen Map zu konzipieren, da der Algorithmus nun zwar für die gewählte Test Map funktioniert, jedoch nicht für Maps mit spezielleren Strukturen, die die Test Map nicht beinhaltet hat. Leider hatte ich keine Zeit mehr den Algorithmus an weitere Spezialfälle anzupassen und so funktioniert er für die meisten Maps noch nicht.

Bisher sind mir folgende Verbesserungsmöglichkeiten aufgefallen, die ich jedoch weder getestet noch implementiert habe. Deshalb sind die meisten nur Vermutungen:

1. Eine Node muss nicht immer speziell den zuletzt hinzugefügten Weg berühren, sondern immer nur eine der Nodes im Weg

Bei einer Test Map (siehe Abbildung ABBILDUNGSNUMMER)