DOKUMENTATION

36 Bundeswettbewerb Informatik
2. Runde / 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:

$$MauerBreite = \frac{N^2 + N}{2}$$

$$AnzahlStellenFuerFugen = MauerBreite - 1$$

$$MaximaleMauerHoehe = \frac{AnzahlStellenFuerFugen}{N - 1}$$

$$MaximaleAnzahlBesetzterFugen = (N - 1) \cdot MaximaleMauerHoehe$$

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.

```
/// <summary>
/// Bildet eine moegliche Mauer
/// </summary>
/// <param name="allMoeglicheReihen">Die Liste aller moeglicher Reihen</param>
/// <returns>Die zuerst gefundene moegliche Mauer</returns>
private Mauer FindMauer(Reihe[] allMoeglicheReihen)
{
    var reihenMatrix = new byte[MoeglicheReihen.Length][];

    for (var i = 0; i < allMoeglicheReihen.Length; i++)
    {
        var startMauer = new Mauer(MaxMauerHoehe);
        startMauer.AddReihe(allMoeglicheReihen[i]);

        var mauer = BaueMauer(reihenMatrix, allMoeglicheReihen, startMauer);
        if (mauer.Fertig)
        {
            // Mauer gefunden
            return mauer;
        }
    }
    throw new Exception("Es wurde keine Mauer gefunden");
}
```

ABBILDUNG 1

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.

```
/// <summary>
/// Erhoeht die Mauerhoehe um 1
/// </summary>
/// <param name="reihenMatrix">Die Matrix der Reihen</param>
/// <param name="allMoeglicheReihen">Die Liste aller moeglicher Reihen</param>
/// <param name="aktuelleMauer">Die aktuelle Mauer</param>
/// <returns>Die gebaute Mauer</returns>
private Mauer BaueMauer(byte[][] reihenMatrix, Reihe[] allMoeglicheReihen, Mauer aktuelleMauer)
{
    // Reihen der aktuellen Mauer
    var reihen = aktuelleMauer.Reihen.ToArray();
    // Reihen, die zu ALLEN Reihen der aktuellen Mauer passen
    var dazuMoeglicheReihen = allMoeglicheReihen.Where(r => reihen.All(pr =>
        Utilities.ReihenSindKompatibel(pr, r, ref reihenMatrix))).ToArray();

    for (var i = 0; i < dazuMoeglicheReihen.Length; i++)
    {
        // Es gibt noch moegliche Mauern
        var mauer = new Mauer(aktuelleMauer);
        mauer.AddReihe(dazuMoeglicheReihen[i]);

        if (mauer.Fertig)
            return mauer;

        var neueMauer = BaueMauer(reihenMatrix, allMoeglicheReihen, mauer);
        if (neueMauer.Fertig)
            return neueMauer;
    }

    // Mauer funktioniert nicht
    return aktuelleMauer;
}
```

ABBILDUNG 2

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.

```
/// <summary>
/// Ueberprueft ob zwei Reihen kompatibel sind
/// </summary>
/// <param name="a">Die erste Reihe</param>
/// <param name="b">Die zweite Reihe</param>
/// <param name="matrix">Die bisher gebildete Reihen Matrix</param>
/// <returns>True wenn die Reihen kompatibel sind</returns>
public static bool ReihenSindKompatibel(Reihe a, Reihe b, ref byte[][] matrix)
{
    if (a == null) return true;

    var indexA = a.Id;
    var indexB = b.Id;

    if (matrix[indexA] != null && matrix[indexA][indexB] != 0)  // Bereits eingetragen
        return matrix[indexA][indexB] == 2;
    if (matrix[indexB] != null && matrix[indexB][indexA] != 0)  // Bereits eingetragen (gespiegelt)
        return matrix[indexB][indexA] == 2;

    // Noch nicht eingetragen
    if (matrix[indexA] == null)
        matrix[indexA] = new byte[matrix.Length];

    // Reihen checken
    var fugenUeberlappungen = a.BesetzteFugen.Intersect(b.BesetzteFugen);
    var kompatibel = !fugenUeberlappungen.Any();

    // Reihen eintragen
    if (kompatibel) matrix[indexA][indexB] = 2;
    else matrix[indexA][indexB] = 1;

    return kompatibel;
}
```

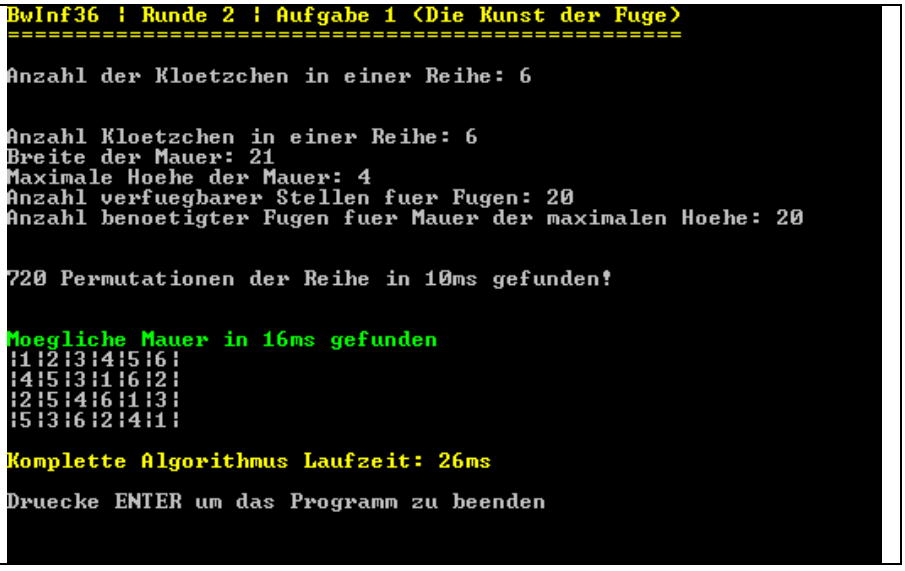
ABBILDUNG 3

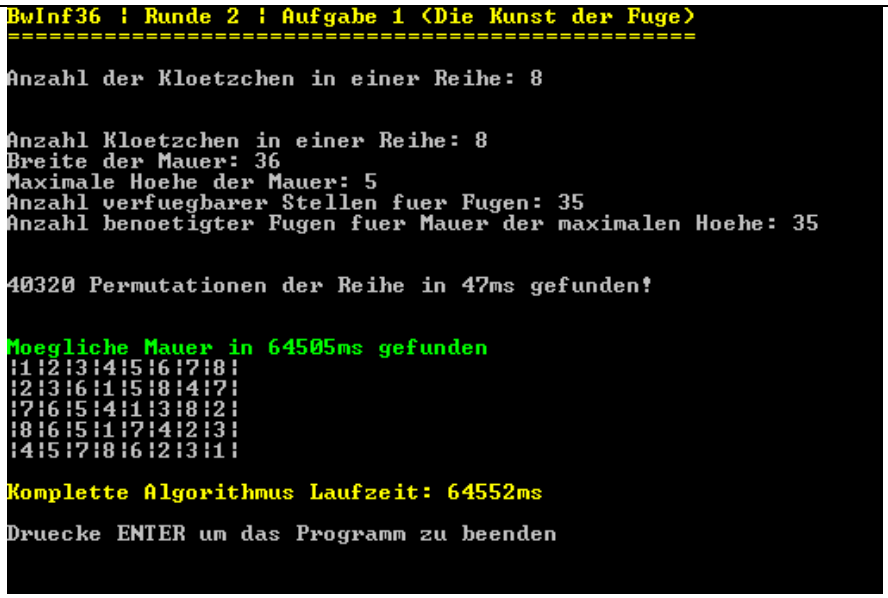
Zum Schluss wird die gefundene Reihe noch in der Konsole ausgegeben.

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