1 Lösungsidee

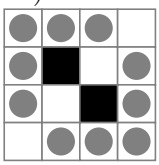
1.1 Kernidee

Rominos mitt n Blöcken können gefunden werden, in dem zu Rominos mit (n-1) Blöcken ein Block angefügt wird. Hierbei muss beachtet werden das der Rominostein zusammenhängend bleiben muss, und dass mindestens eine Diagonale bleiben muss.

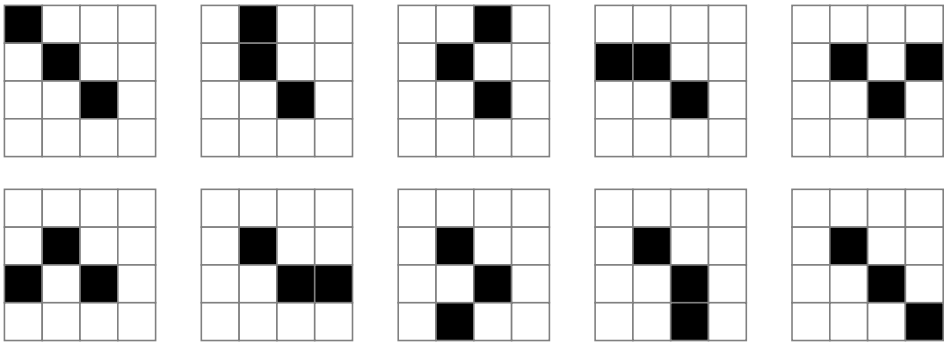
Um alle möglichen Rominos mit n Blöcken zu finden, muss man also alle Rominos mit (n-1) Blöcken finden, und für diese alle Rominos die durch hinzufügen eines weiteren Blocks enstehen können ermitteln. Dabei wird es Duplikate geben. Eliminiert man diese hat man alle möglichen n-Rominos eindeutig gefunden.

1.1.1 Beispiel

Nehme man beispielsweise das 2er-Romino, kann man zum finden aller 3 (= 2 + 1) - Rominos wie folgt Blöcke anfügen:



Somit ergeben sich folgende 3-Rominos:



Da Rominos mindestens zwei Steine haben müssen um eine Diagonale zu besitzen, ist der Rominostein mit den wenigsten Blöcken eine 2er Diagonale.

Kleinster Rominostein

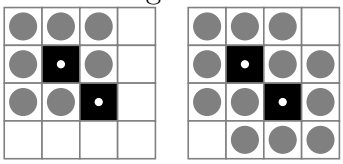
Um alle n-Rominos für ein beliebiges n zu finden, würde man den obigen Algorithmus verwenden um aus dem 2er-Romino alle 3-Rominos zu folgern, dann aus diesen alle 4-Rominos etc. bis man alle n-Rominos errechnet hat.

1.2 Hinzufügen von Blöcken

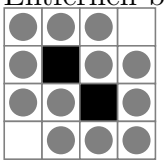
Um Blöcke hinzuzufügen, werden zuerst die Stellen ermittelt, wo Blöcke angefügt werden können, sodass das Romino zusammenhängend bleibt. Hierfür werden die Nachbarn jedes Blocks des Rominos ermittelt, daraufhin werden Duplikate und bereits belegte Blöcke eliminiert.

1.2.1 Beispiel

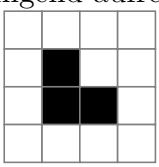
Nehme man beispielsweise wieder das 2er-Romino, würden die Nachbarn aller Blöcke wie folgt ermittelt werden:



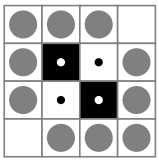
Entfernen bereits existierender Blöcke



Es lässt sich hier erkennen, das die Existenz einer echten Diagonale nicht zwingend aufrecht erhalten wird;



Um dafür zu sorgen, dass diese echte Diagonale immer existiert, wird eine spezifische Diagonale immer beschützt. Bei den Möglichen Block-Additionen beim 2er-Romino beispielsweise würden hierfür die für die Diagonale relevanten Blöcke aus den Block-Additionsmöglichkeiten entfernt:



Diese 4 beschützten Blöcke werden auch bei Spiegelungen, Verschiebungen und Rotationen mitverfolgt, sodass diese eine Diagonale immer besteht.

1.3 Eliminierung von Duplikaten

Zur Eliminierung von Duplikaten werden die Rominos zuerst eindeutig orientiert, um Vergleiche zwischen gleichen, aber transformierten Rominos zu erleichtern.

1.3.1 Verschiebung

Die Verschiebung wird eliminiert durch Verschiebung des Rominos in die linke obere Ecke des Gitters; also wird der Block mit der geringsten x-Koordinate auf x=0 verschoben, und der Block mit der geringsten y-Koordinate auf y=0.

1.3.2 Rotation und Spiegelung

Um Rotation und Spiegelung eines Rominos zu eliminieren, werden zuerst alle seine Permutationen (also alle Kombinationen von Rotation und Spiegelung) ermittelt, und denen wird ein eindeutiger Wert zugewiesen. Daraufhin wird das Romino mit dem höchsten dieser eindeutigen Werte ausgewählt. Hierbei ist es eigentlich egal, ob der niedrigste oder höchste Wert genommen wird, solange das Ergebnis eindeutig ist.

Die Bestimmung dieses eindeutigen Werts haben wir einen trivialen Algorithmus verwendet wie folgt:

1. Nehme einen Block b aus der Permutation des Rominos
2. Seien die Koordinaten (x, y) die Koordinaten des Blocks b, wobei die minimale x-Koordinate und die minimale y-Koordinate aus allen Blöcken der Permutation 0 ist.

3. Man weise dem Block b den Wert $2^{(y*<\text{Anzahl an Blöcken}>)+x}$ zu

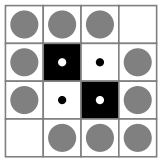
4. Addiere die Werte aller Blöcke der Permutation, sei dies der Wert der Permutation

Dabei ist zwar noch viel Raum für Optimierung, aber dieser Algorithmus ist ausreichend und O(n).

1.3.3 Endgültige Duplikat-Eliminierung

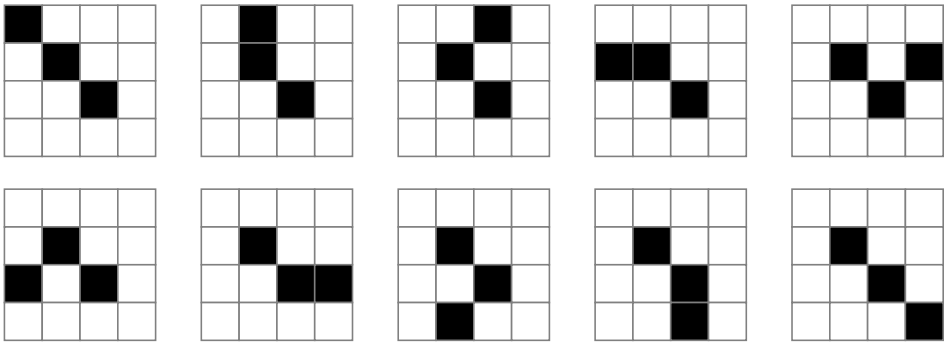
Zum endgültigen eliminieren der Duplikate werden zuerst alle Rominos wie oben beschrieben orientiert, dann werden die eindeutigen Werte dieser verglichen, um schnell Gleichheit zu ermitteln. Durch Verwendung dieser Vergleichsmethode lassen sich schnell Duplikate entfernen.

1.3.4 Beispiel

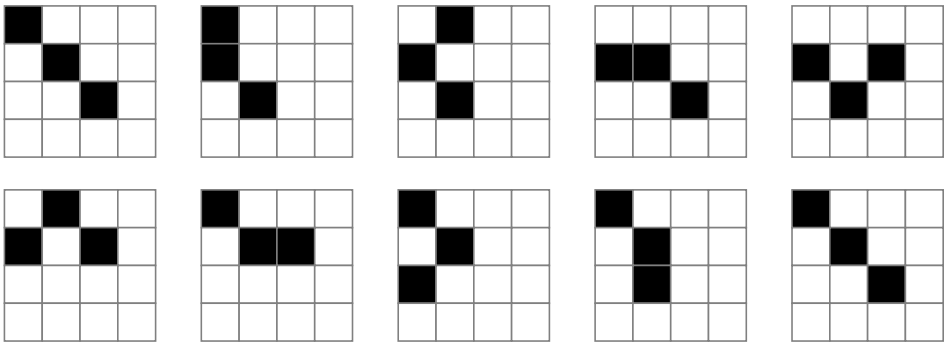


Ausgangsromino

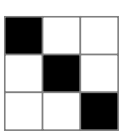
Nächste Rominos



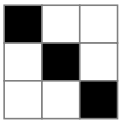
Verschiebung eliminieren

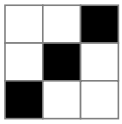


Rotation und Spiegelung eliminieren Ausgehend von dem Romino;



werden folgende Permutationen festgestellt:

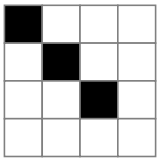

Permutation 1

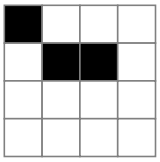

Permutation 2

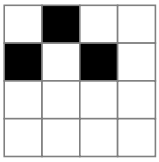
$$Wert_1 = 2^0 + 2^4 + 2^8 = 273 \qquad Wert_2 = 2^2 + 2^4 + 2^6 = 84$$

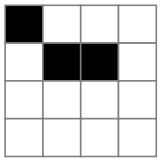
Hierbei ist $Wert_2 = 84 < 273 = Wert_1$. Da Permutation 1 mit $Wert_1$ den höchsten Wert hat, wird Permutation 1 als die eindeutige Rotierung festgelegt.

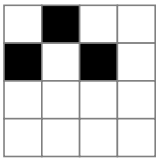
Analog auf alle Rominos angewendet ergibt sich:

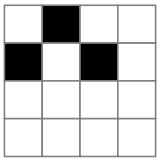

273

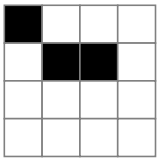

49

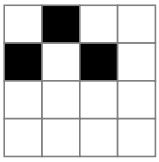

42

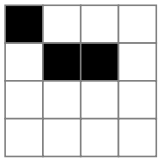

49

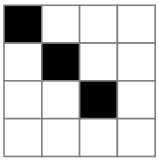

42


42

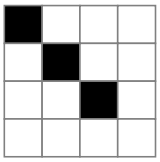

49

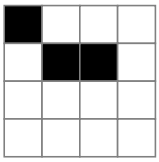

42

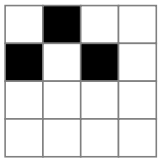

49


273

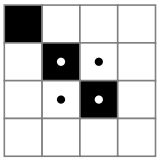
Nun lassen sich trivialerweise die Duplikate eliminieren;

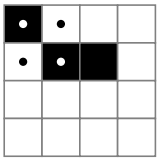

273

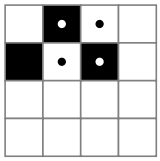

49


42

Über den gesamten Prozess hinweg wird auch die geschützte Diagonale mitverfolgt, bei den 3er-Rominos ist sie wie folgt plaziert:


273


49


42

2 Umsetzung

Zur Umsetzung haben wir den obigen Algorithmus in C# 8.0 mit .NET Core 3.0 implementiert.

Die Rominos werden in Form eines structs *Romino* gespeichert. das struct beinhaltet

1. *Vector2Int[] Blocks* - Das Array mit allen Blöcken des Rominos.
2. *List<Vector2Int> PossibleExtensions* - Die Liste mit allen Block-Additionsmöglichkeiten. Hierbei ist zu bemerken, dass die Größe der Liste konstant bleibt; es wird hier eine Liste statt einem Array verwendet, da bei der Erstellung die Größe unbekannt ist, und die Liste noch in ein Array zu konvertieren unnötig Rechenzeit kostet.
3. *Vector2Int DiagonalRoot* - Die linke obere Ecke der geschützten Diagonale.
4. *Vector2Int Max* - Die rechte untere Ecke des Rominos. Verwendet für korrigieren der Verschiebung ohne über alle Blöcke zu iterieren.
5. *BitBuffer512 _uniqueCode* - Der eindeutige Wert, errechnet wie in 1.3.2.