POLYOMINO – Ein Spiel

Praktikumsbericht “Wissenschaftliches Rechnen”
Dozent: Prof. Dr. Peter Bastian
SS 2012 Universität Heidelberg

Apolline Pleniér- Motte, Petra Nichtburgerova

20. August 2012

Inhalt

1 Grundlagen des Spiels

Das Polyomino-Spiel ist ein Puzzle, in dem es darum geht Steine von verschiedener Größe auf einem Spielfeld so einzufügen, bis kein Platz frei bleibt. D.h. die ganze Fläche des Spielfelds mit Polyominos auszufüllen. Dabei sind alle möglichen Lösungen zu finden und alle unmöglichen Fälle auszuschließen.

1.1 Polyominos

Ein Polyomino (der Name wurde von Domino abgeleitet) ist eine Fläche, die aus *n* zusammenhängenden Quadraten besteht. Für kleine n sind auch die Bezeichnungen Triomino (*n=3*), Tetromino (*n=4*), Pentomino (*n=5*) und Hexomino (*n=6*) üblich. Quadrate müssen an sich anlehnen, d.h. zwei Quadrate müssen eine Kante gemeinsam haben.

Polyominos können beliebig gedreht werden und in beliebiger Ordnung in das Spielfeld eingefügt werden.

Abb. 1 Polyominos einzeln und in einem Feld (4x2)

Am Anfang des Spiels kann man nach Belieben die Größe des Spielfelds und auch die gewünschten Polyominos wählen. Das Programm überprüft, ob die gewünschten Polyominos in das Spielfeld passen und ob es Lösungen gibt. Falls ja, gibt es alle möglichen Lösungen, die Anzahl der Lösungen und die Rechenzeit aus.

1.2 Spielregeln

1. Steine wählen
2. Feld definieren
3. Steine einzeln einfügen, sodass sie sich nicht überlappen, dabei können die Steine gedreht werden
4. Möglichst alle Lösungen für eine Menge von Steinen finden

1.3 Anforderungen an das Programm

Das Ziel ist die ganze Spielfläche (ohne Lücken) mit Polyominos auszufüllen. Das Programm soll alle Polyomino-Lösungen mittels der Backtrackingmethode finden und gleichzeitig auch alle unmöglichen Fälle erkennen.

Eine Methode ist ein Polyomino auf der Fläche zu platzieren und alle gegebenen Möglichkeiten für die anderen Steine durchzuprobieren. D.h. alle gewählten Steine in allen möglichen Varianten im eingegebenen Feld auszuprobieren.

Das Hauptproblem ist, dass es so zu viele verschiedene Möglichkeiten gibt. Daher muss die „Backtracking“-Methode genutzt werden um die Anzahl von Versuchen und die Ausführungszeit zu reduzieren.

Triomino

Tetromino
(B)

Abb. Zwei Polyominos

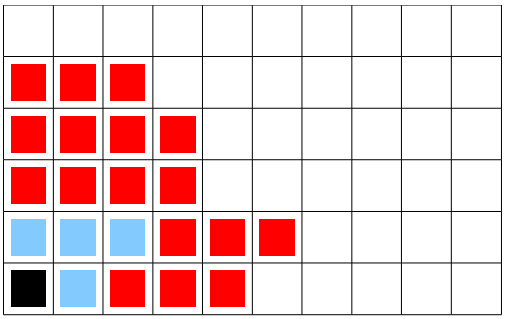


Abb. Eine Spielfeldfläche

Einlage des Polyominos **B** (irgendwo) => blau
Es wird probiert Polyomino **A** in den benachbarten Feldern einzulegen => rot
Wegen des schwarzen Feldes ist es unmöglich **A** zu platzieren => keine Lösung => Backtracking.
Ein anderer Polyomino wird gesucht, sodass es im schwarzen Feld platziert werden kann: Hier brauchen wir einen Monomino. Wenn es keinen Monomino gibt es keine Lösung => Backtracking
usw.

Die Spielfläche und die Steine werden gleichzeitig durch eine orthonormale Koordinatenebene und als eine Matrix repräsentiert. Polyominos werden zusätzlich mit Buchstaben versehen, sodass sie abgebildet werden können und erkennbar bleiben.

3 Backtracking

3.1 Begriffserklärung

Der Begriff Rücksetzverfahren oder englisch Backtracking (Rückverfolgung) bezeichnet eine mathematische Problemlösungsmethode innerhalb der Algorithmik. Lösungen werden systematisch geprüft. Es wird gleichzeitig aber eine Heuristik entwickelt, durch die früh erkannt wird, dass ein „Zweig“ zu keiner Lösung führt und der Prozess abgebrochen wird, der letzte Stein entnommen und ein anderer probiert.

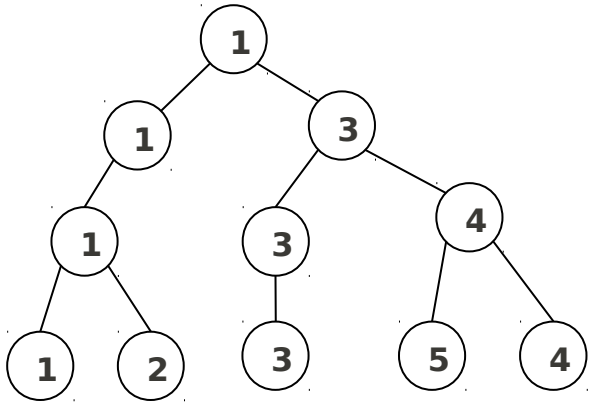


Abb.2 Jeder Zweig steht für eine Lösung.

Ein heuristisches Verfahren reduziert die Möglichkeiten und somit auch die Zeit, indem aussichtslos erscheinende Varianten von vornherein ausgeschlossen werden. Anwendungsgebiete für Backtracking sind im Allgemeinen z.B. das Damenproblem, die Beladung von Containerschiffen, oder das Problem des Handlungsreisenden u.a.

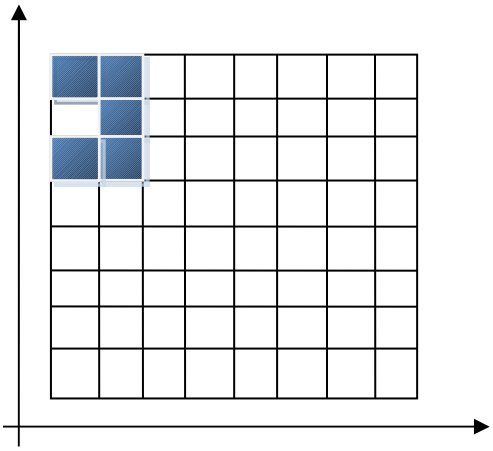


Abb. 3 Falls es kein Monomino gibt, ist diese Teillösung nicht möglich.

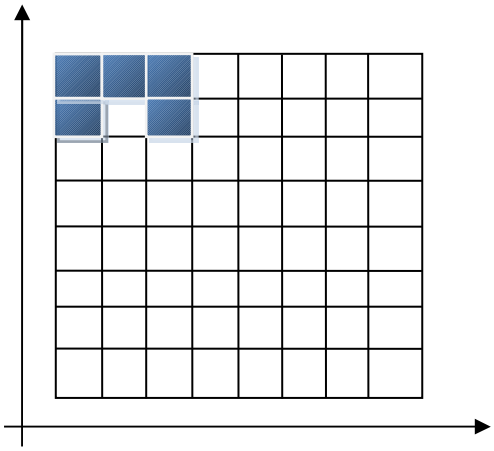


Abb. 4 Diese Teillösung ist möglich.

3.2 Backtracking im Spiel

Praktisch wird Backtracking im Spiel folgendermaßen angewandt:

1. Überprüfe, ob das erste Feld (links oben) leer ist! Nimm den ersten Stein und platziere ihn! Passt der Stein in das Spielfeld oder sind Teile des Steins außerhalb des Feldes? Ist diese Lösung aus anderen Gründen nicht möglich? Wenn ja, versuche den Stein zu drehen und zu bewegen bis es passt oder wenn auch das zu keiner Lösung führt, nimm den Stein weg und probiere einen anderen Stein… Wiederhole bis es passt.

2. Geh zum nächsten leeren Feld über. Probiere den nächsten Stein, ob es in die Umrisse des Feldes und mit den Grenzen anderer Steine passt.

3. Wenn das ganze Feld besetzt ist, speichert es diese Lösung und das Programm fängt von Anfang an, eine neue Kombination von Steinen zusammenzustellen.

4. Wenn kein passender Stein mehr gefunden werden kann und das Feld nicht ganz besetzt werden kann, fängt das Programm an, eine neue Kombination von Steinen zu suchen bis keine neue Kombination gefunden werden kann.

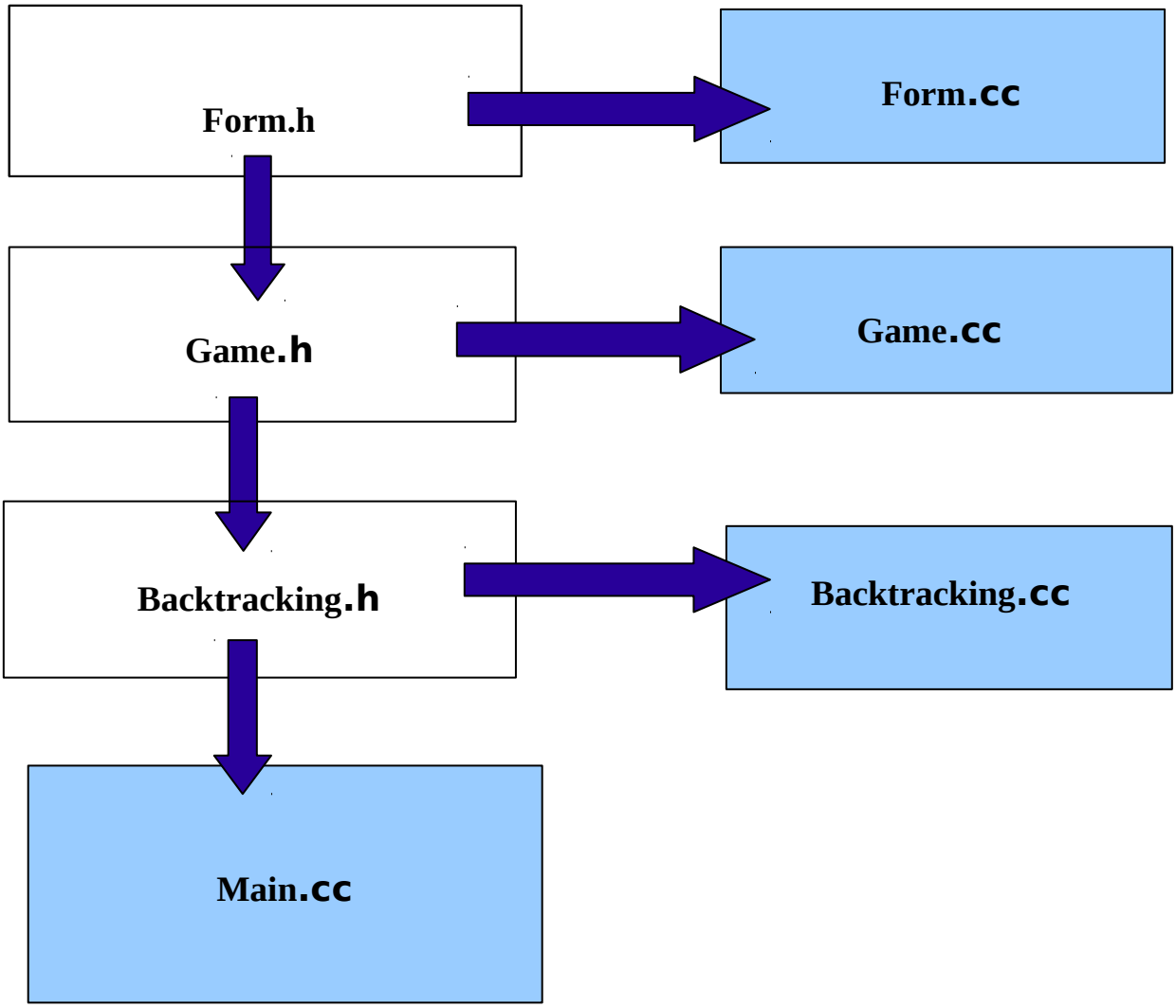
5. Im Prinzip „Branch-and-Bound“ können in jedem Schritt alle möglichen Lösungen aufgezählt werden. D.h. das Programm sucht systematisch nach einer Lösung unter allen vorhandenen Möglichkeiten und geht jedes Mal zu dem nächsten freiliegenden Feld über.

6. Im Prinzip „Backtracking“ können Steine zurückgezogen werden und andere anstatt von diesen gewählt bei einer nicht erfolgreichen Lösung. D.h. In jeder Phase sucht es nach Teillösungen. Wenn die Teillösung nicht gefunden werden kann, das Algorithmus geht ein Schritt zurück und löscht den letzten Schritt und mit dem nächsten Schritt wird ein anderer Schritt ausprobiert.

4 Programmstruktur

Die Struktur des Programms wurde in verschiedene Objekte, Klassen aufgeteilt. Die Steine, das Spielfeld, und das Backtrackingverfahren sind jeweils in einer einzelnen Klasse. Die Klassen werden auch als separate Files gespeichert und durch Verweise zusammengebunden.

4.1 Die Einteilung der Files



4.2 Die Klassen

<u>die Klasse « Form »</u>

Zum Ansatzpunkt wurde die Klasse „Form“, mit der man Steine erstellt. Die Steine werden durch eine Menge von Punkten symbolisiert. Jeder Punkt korrespondiert mit zwei ganzen Zahlen (x,y), die die Koordinaten des Grundrisses bilden. Um die Punkte zu modellieren, haben wir eine Struktur konstruiert:

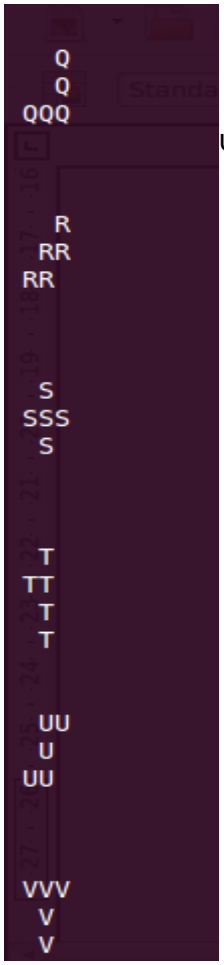
```
struct Point
{
int x,y;
```

```
};
```

Für jeden Stein gibt es drei Merkmale: die Menge von Punkten (eine Matrix von Koordinaten), die Anzahl von Punkten (eine ganze Zahl), und die Kennzeichnungsnummer des Steins (eine ganze Zahl, die in einen Buchstaben transformiert wurde).

In dieser Klasse gibt es alle Funktionen, die mit den Formen/Steinen in Interaktion sind. Zum Beispiel existiert eine Funktion „print_piece“ gennant, die einen Stein auf dem Terminal anzeigt.

Die Ergebnisse der Funktion „print_piece“ nach der Ausführung des Programms:



In diesem Beispiel gibt es sechs verschiedene Formen 'Q' 'R' 'S' 'T' 'U' und 'V'.

Jede von Ihnen besitzt 5 Punkte.

Für 'S' ist es die Menge M:= { (1,0) , (0,1) , (1,1) , (1,2), (2,1) }

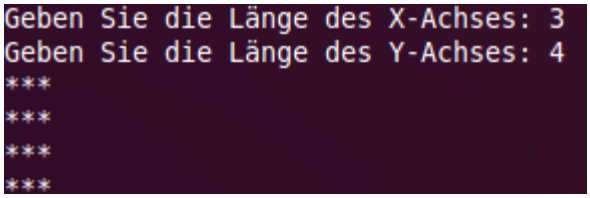
Ein Punkt.

Die Klasse « Game »

Es musste auch die Klasse „Game“ erstellt werden, um ein Spielfeld als Objekt zu schaffen. Das Spielfeld wurde durch eine Matrix von Koordinaten erstellt. Jeder Punkt der Matrix stimmt mit einem Platz überein, an dem ein Punkt des Steins platziert werden kann. Die Breite (m_width Variable) und die Länge (m_length Variable), beide Ganzzahlen, stellen die Dimensionen des Spielfelds dar. Es wurde so eine N*M Matrix, mit N = m_width und M = m_length gebildet.

Die Klasse «Game» enthält alle Funktionen, die mit dem Spielfeld verbunden sind. Zum Beispiel gibt es die Funktion print_game, die das Spielfeld auf dem Terminal anzeigt.

Die Funktion wird im Programm folgendermaßen ausgeführt:



die Klasse « Backtracking »

Diese letzte Klasse dient dazu, die Lösungen zu liefern. Steine werden in das Spielfeld eingesetzt, Ergebnisse gespeichert und angezeigt. In dieser Klasse befinden sich:

- ein Spielfeld, weshalb wir das Objekt „Game“ verwenden,

- mehrere Steine und ihre Varianten, diese sind in dem Vektor m_v enthalten und werden aus dem Vektor aufgerufen,

- Lösungen (d.h. mehrere Spielfelder mit Steinen), die in einem anderen Vektor gespeichert werden,

- Anfangspositionen der Steine im Vektor m_v, die in einem anderen Vektor gespeichert sind,

Die Auslösefunktionen des Polyominos gehören ebenfalls zu dieser Klasse. Als Beispiel kann die Funktion free_place angegeben werden:

Point Backtracking : : free_place ( ) const // Die Benennung der Funktion ist free_place,

{ // ihre Aufgabe ist es einen freien

Platz im Spielfeld zu suchen.

int i,j ; // Deklaration von zwei Variablen zur Prüfung aller Koordinatenkombinationen des Spielfelds.

Point place ; // Eine Variable, in der Koordinaten eines freien Platzes gespeichert werden.

for(i=0 ; i < m_game.get_width ( ) ; i++)

{

for(j=0 ; j < m_game.get_length ( ) ; j++)

{

if (m_game.get_case(i , j)==0) // Es gibt einen freien Platz <=> wenn das Ergebnis eine '0' ergibt.

{

place.x=i ; // Die Variable place nimmt die Werte von 'i' und 'j' an.

place.y=j;

return place; // Wir geben die Variable 'place' wieder, um den gefundenen freien Platz in einer anderen verwenden zu können.

}

}

}

}

4.3 Der Backtracking-Algorithmus

Die Funktion „step“, die sich in der Klasse Backtracking befindet, sucht alle Lösungen für ein beliebig definiertes Spielfeld und Steine. Die Funktion wird in folgenden Schritten aufgerufen:

```
 void Backtracking : : step ( )
{
  int j,k,z ;
  int size;
  Point place;
```

(0)

```
  if ( full_game ( ) )
     {
       m_solutions.push_back(m_game);
       m_game.print_game();
       cout<<endl<<endl;
     }
```

(1)

```
  else
{
  place=free_place( );
```

(2)

```
  for( j=0 ; j < m_start.size ( ) ; j++)
  {
    if( m_used[j] ==0 )
    {
    size = m_v[j].get_size();
```

(3)

```
   for( k = m_start[j]; k < m_start[j+1] ; k++)
    {
       for(z=0 ;  z<size ; z++)
       {
        if ( m_game.goodposition ( m_v[k], place.x, place.y, z) )
        {
          m_game.insert_piece (m_v[k], place.x, place.y, z) ;
          m_used[j]=1;
          step();
          m_game.delete_piece(m_v[k], place.x, place.y, z);
          m_used[j]=0;
        }
    }
   }
  }
 }
}
return;
}
```

(4)

(5)

Im Folgenden ist die Funktion mit ihren Schritten beschrieben:

0) Variablen werden deklariert, die in der Funktion step vorkommen.

1) Mithilfe der Funktion full_game wird getestet, ob das Spielfeld voll ist:

 Ist das Spielfeld voll => gibt man diese Lösung auf dem Terminal aus und speichert man diese Lösung in m_solutions.

 Das Spielfeld ist nicht voll => (2)

2) Das Spielfeld ist nicht voll, weswegen ein freier Platz im Spielfeld gesucht wird, auf dem ein Stein platziert werden könnte.

3) Falls ein freier Platz gefunden werden konnte, versuchen wir einen Stein

einzulegen, dieser Stein muss jedoch mehrere Bedingungen erfüllen:

- für jeden Stein <=> for(j=0 ; j < m_start.size ( ) ; j++)

- der noch nicht benutzt wurde <=> if(m_used[j] ==0)

- für jede Variante dieses Steins <=> for(k = m_start[j]; k < m_start[j+1] ; k++)

- for(z=0 ; z<size ; z++) <=> für alle Punkte, die zu dieser Variante dieses Steins gehören.

4) Wir benutzen die Funktion goodposition, die in der Klasse Game ist, um zu testen ob der Punkt 'z' des Steins 'm_v[k]' an die Koordinaten ('place.x','place.y') platziert werden kann <=> if (m_game.goodposition (m_v[k], place.x, place.y, z)).

Wenn es möglich ist, platziert man diese Variante an den freien Platz mithilfe der Funktion insert_piece.

Anschließend nimmt m_used[j] den Wert 1 an, da der Stein eingefügt wurde.

Die Funktion step wird nochmal eingesetzt, um zu testen, ob das Spielfeld voll ist, und wenn nicht, um einen anderen Stein an einem neuen Platz zu platzieren. Damit kommen wir zu Schritt (1) zurück.

5) Wir löschen den Stein, den wir platziert haben und initialisieren m_used[j] zu 0, um alle Möglichkeiten zu probieren.

5 Programmeffizienz

Für ein Spielfeld mit den Dimensionen 6*10 und 12 Steinen von einer Größe von 5 Feldern muss das Programm auf dem verwendeten Rechner zwischen 50 Minuten und einer Stunde rechnen, um alle Lösungen zu gewinnen. Diese Dauer erscheint zwar nicht außergewöhnlich, die Effizienz des Programms könnte trotzdem verbessert werden:

1. In der Funktion full_game der Klasse Backtracking:

Annahme : die Fläche der Formen ist gleich wie die Fläche des Spielfelds.

Konsequenz: es ist ineffizient, das ganze Spielfeld durchzulaufen,

=> 6 (width) * 10 (length) = 60 Vergleiche zwischen int

gegeben, dass :

1) ein int entspricht 4 Bytes auf einem 32bit-System

2) diese Funktion wird in der Funktion step benutzt, und kann viele Male (z.B. 1 Million) für eine Auflösung wiederbenutzt werden.

2. Es wäre effizienter, einen Vektor mit boolschen Variablen mit dem Namen

m_used zu verwenden, um zu erfahren, ob alle Steine platziert worden sind oder nicht.

1) Die Grösse einer boolschen Variablen ist bei manchen Kompilern wie 1 Byte. Sonst ist die Größe gleich wie die einer Integervariable.

2) In unserem Fall geben wir nur 12 Vergleiche anstelle der 60 Vergleiche für jede Ausführung der Funktion step.

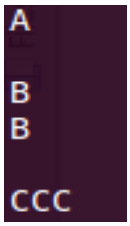
3. Lösungen, die nur durch Rotation (180°-Drehungen) des Spielfeldes erzeugt werden, müssten nicht einzeln behandelt werden. Bei quadratischen Feldern trifft dies auch für 90°-Drehungen zu.

6 Beispiele

6.1 Ein kleines Beispiel

Ein Spielfeld mit den Dimensionen : 2 (Breite) * 3 (Länge).

Wir wählen die Steine A B und D aus:



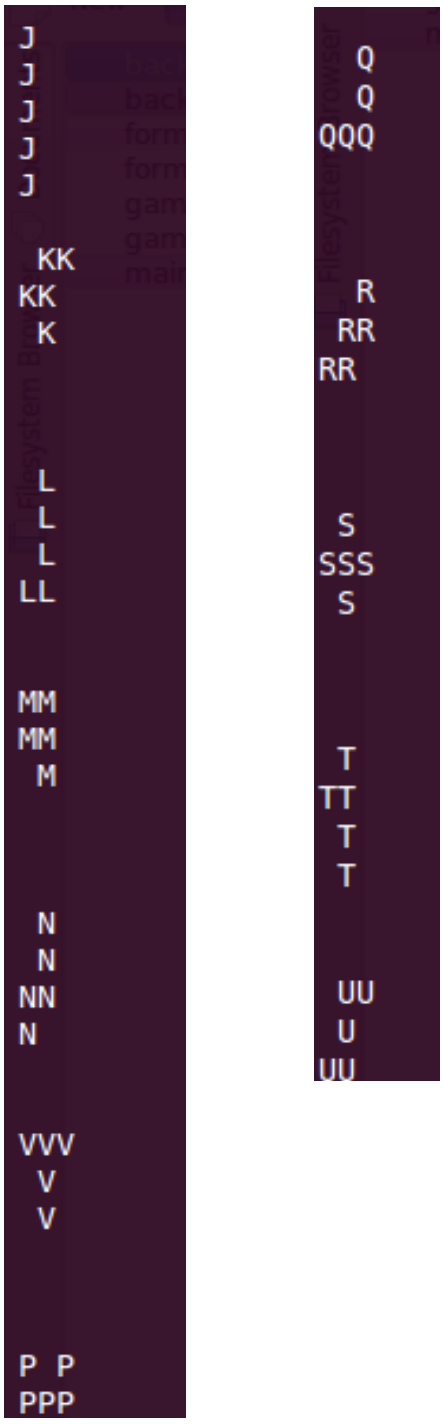
Ergebnisse :


AC
BC
BC
BC
BC
AC
CA
CB
CB
CB
CB
CA
Es gibt 4 Lösungen
die Dauer der Ausführung des Programms: 0.01 Sekunden

6.2 Ein größeres Beispiel

Ein Spielfeld mit den Dimensionen : 6 (Breite) * 10 (Länge).

Wir wählen diese Steine aus:



Einige Ergebnisse :