Aufgabe 2: Dreieckspuzzle

Team-ID: 00129

Team: Teamname PIE_Team

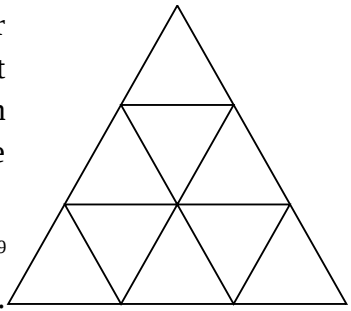
Bearbeiter dieser Aufgabe:
Jonathan Busch

14. November 2020

Inhaltsverzeichnis

Lösungsidee

Das Puzzle wird mit Brute-Force gelöst. Um die Anzahl der Möglichkeiten (= Anzahl der Permutationen der Teile, multipliziert mit Drehmöglichkeiten = n! * 3^n) dabei gering zu halten, werden an bestehende Teile nur passende Teile rekursiv angesetzt. Sobald eine Lösung gefunden ist, wird die Suche abgebrochen.



Von dieser Anordnung wird ausgegangen

Bei n=9 (wie in der Aufgabenstellung gegeben) gibt es insgesamt 9! * 3^9 = 7.142.567.040 ≈ 7,1 * 10^9 Möglichkeiten, die Puzzleteile anzuordnen. Diese Abschätzung ist jedoch die absolute Obergrenze und basiert auf der Annahme, dass jedes Puzzleteil nur gleiche Figurenteile zeigt, sodass es beliebig gedreht werden kann. Das bedeutet aber in diesem Fall auch, dass es viel weniger Möglichkeiten gibt, die Teile anzuordnen, da die Drehung keine Rolle spielt. Durch das beschriebene Verfahren wird die Anzahl der Möglichkeiten tendenziell gering gehalten (verhält sich aber trotzdem zur Teilezahl noch exponentiell), weshalb sich die Anzahl der zu überprüfenden Möglichkeiten für n=9 im für moderne Computer realistischen Bereich befindet.

Umsetzung

Das Programm wird in Java implementiert. Für die Repräsentation des Puzzles wird ein Graph verwendet. Die Knoten des Graphen sind die Puzzleteilpositionen; die Kanten des Graphen sind gemeinsame Kanten der Teile und damit die Figurenpositionen. Temporär angelegte Teile werden

also praktisch als Kanteninformationen gespeichert. Die Reihenfolge, in der die Puzzleteile angelegt werden, ist die Reihenfolge, in der sie von einer Tiefensuche vom obersten Knoten aus erreicht werden.

Ausgehend von dieser Reihenfolge wird das Puzzle mittels rekursivem Backtracking gefüllt.

Optimierungen des Backtrackings

Beim Einlesen werden gleiche Puzzleteile zusammengefasst, indem in einer Variable die Anzahl der verfügbaren Teile mit gleichen Figuren gespeichert wird. So wird verhindert, dass praktisch gleiche Teile an derselben Stelle mehrmals angelegt werden.

Bevor ein Teil zum Anlegen rotiert wird, wird überprüft, ob das Teil drei Mal die gleiche Teilfigur enthält. Wenn das der Fall ist, wird das Teil nur in seiner Ursprungsrotation verwendet.

Darstellung des Graphen

Der Graph wird als Adjazenzliste dargestellt. Jede Liste hat die Länge 3; die Sortierung der Kanten ist immer gleich. Die waagerechte Puzzlekante _ hat immer den Index 2; die Puzzlekante \ hat den Index 1 und / hat Index 0. Auf diese Weise kann man sehr einfach von der anderen Seite auf die gleiche Kante zugreifen, da die Kante auf der anderen Seite immer den gleichen Index hat.

Beispiele

Da zu jedem Beispiel eine Lösung existiert, wurde ein einfaches unlösbares Beispiel erzeugt, indem beim letzten Beispiel alle Vorzeichen entfernt wurden. So wird die Reaktion des Programms auf unlösbare Puzzles gezeigt.

```
$ java A2Main
3
9
-1 -2 1
2 -1 -1
-1 -2 2
-1 3 1
2 -3 3
-1 3 -2
2 2 -1
-3 2 -1
-2 1 -3
Solution found!
              / 1  -1\
                 -2
                 --
                  2
       / 2   2\-2  -1/ 1  -1\
          -1             3
          --            --
           1            -3
/-1   2\-2  -3/ 3  -2\ 2   3/-3   2\
   -1            -1            -1
   --            --            --
```

```
$ java A2Main
3
9
1 -1 2
2 -3 -1
-1 -1 3
1 -1 -2
-3 3 -1
-1 3 3
1 -2 -3
-1 -2 3
3 -2 2
Solution found!
              /-1   3\
                 -1
                 --
                  1
       /-1   2\-2  -1/ 1  -2\
          -3            -3
          --            --
           3             3
/-1  -2\ 2  -2/ 2   1\-1   3/-3   3\
    3            -1            -1
   --            --            --

$ java A2Main
4
9
-3 -2 -1
-2 -4 1
1 2 -4
-3 -1 1
2 3 1
-3 -2 -4
-3 1 -2
3 4 -1
-2 -3 4
Solution found!
              /-3  -2\
                 -1
                 --
                  1
       /-3   4\-4   2/-2  -4\
          -2            -3
          --            --
           2             3
/-3  -1\ 1   3/-3   1\-1   4/-4   1\
    1            -2            -2
   --            --            --

$ java A2Main
10
9
10 10 4
9 8 -7
10 -5 10
-2 10 7
6 5 -2
2 3 -4
-6 -8 10
-9 10 10
10 -3 2
Solution found!
```

```
                    / 10    10\
                          4
                         ---
                          -4
          / 10    -3\  3     2/ -2    10\
                2                   7
               ---                 ---
                -2                  -7
/ 10    -5\  5     6/ -6    -8\  8     9/ -9    10\
     10                  10                  10
    ---                 ---                 ---
```

```
$ java A2Main
10
9
10 10 4
9 8 7
10 5 10
2 10 7
6 5 2
2 3 4
6 8 10
9 10 10
10 3 2
No solution found!
```

Quellcode

```
...
public final class A2Main {
  private static int     numParts;
  private static int[][] parts;         // Puzzledaten
  private static int[]   available;     // mehrfach vorhandene Teile
  private static int     distinctParts; // Anzahl der unterschiedlichen Teile
  private static int[][] graph;         // Graph
  private static int[][] data;          // Kanten
  private static int[]   order;         // Reihenfolge in der Tiefensuche

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    // Einleseprozedur
    int numFigures = sc.nextInt();
    numParts = sc.nextInt();
    parts = new int[numParts][3];
    available = new int[numParts];
    distinctParts = 0;
    reading: for (int i = 0; i < numParts; ++i) {
      for (int s = 0; s < 3; ++s) {
        parts[distinctParts][s] = sc.nextInt(); // Daten lesen
      }
      for (int j = 0; j < i; ++j) {
        for (int r = 0; r < 3; ++r) { // in allen Rotationen Matchings überprüfen
          boolean matches = true;
          for (int k = 0; k < 3 && matches; ++k) matches = parts[j][k] == parts[distinctParts][(k
                            + r) % 3];
          if (matches) {
            ++available[j]; // Zähler erhöhen
            continue reading;
          }
        }
      }
      ++available[distinctParts];
      ++distinctParts; // neues Teil
```

```
    }
    sc.close();
    graph = new int[numParts][3];
    data = new int[numParts][1];
    // Graphen initialisieren
    for (int i = 0; i < numParts; ++i) {
      int y = (int) Math.sqrt(i);
      int f = y * y; // erstes Teil der Reihe
      int yp1 = y + 1;
      int x = i - f;
      if ((x & 1) == 0) { // nur Teile mit Spitze nach oben betrachten
        int o = i - 1;
        if (o >= f) {
          graph[i][0] = o;
          graph[o][0] = i; // dafür jedes Mal beide Seiten initialisieren
        } else graph[i][0] = -1;
        o = i + 1;
        if (o < Math.min(numParts, yp1 * yp1)) {
          graph[i][1] = o;
          graph[o][1] = i;
        } else graph[i][1] = -1;
        o = i + 2 * (y + 1);
        if (o < numParts) {
          graph[i][2] = o;
          graph[o][2] = i;
        } else graph[i][2] = -1;
      }
    }
    order = new int[numParts];
    dfs(0, 0); // Reihenfolge suchen
    data = new int[numParts][3];
    boolean found = addPart(0); // Lösungen suchen
    System.out.println((found ? "S" : "No s") + "olution found!");
    int intLength = (int) (Math.log10(numFigures) + 2);
    if (found) { // Lösung ausgeben
      int height = (int) Math.sqrt(numParts);
      for (int y = 0; y < height; ++y) { // Zeilenweise durchgehen
        int w = y * 2 + 1;
        int firstIndex = y * y;
        String[] lines = new String[4]; // 4 Textzeilen pro Puzzlezeile
        int base = (height - 1 - y) * (intLength * 3 + 1); // Basiseinrückung
        for (int i = 0; i < lines.length; ++i) lines[i] = toLength(base, ' ', "");
        for (int x = 0; x < w; ++x) { // Alle Teile in der Puzzlezeile
          int index = firstIndex + x;
          boolean even = (x & 1) == 0; // Spitze nach oben / unten
          // Zeilen generieren
          lines[1]
              += (even ? "/" : "\\") + toLength(intLength, ' ', Integer.toString(data[index][even
                             ? 0 : 1]))
                  + toLength(intLength * 2, ' ', Integer.toString(data[index][even ? 1 : 0]));
          lines[even ? 0 : 2] += toLength(intLength * 3 + 1, ' ', "");
          lines[even ? 2 : 0] += toLength(intLength * 2 + 1, ' ', Integer.toString(data[index]
                             [2]))
              + toLength(intLength, ' ', "");
          lines[3] += even ? (toLength(1 + intLength * 2, ' ', toLength(intLength, '-', ""))
              + toLength(intLength, ' ', "")) : toLength(intLength * 3 + 1, ' ', "");
        }
        lines[1] += '\\';
        for (int i = y == 0 ? 1 : 0; i < lines.length; ++i)
                             System.out.println(lines[i].stripTrailing());
      }
    }
  }
```

```
  // Leerzeichen an String anfügen -> String verlängern
  private static String toLength(int length, char c, String end) {
    char[] a = new char[Math.max(0, length - end.length())];
    for (int i = 0; i < a.length; ++i) a[i] = c;
    return new String(a) + end;
  }

  // einfache Tiefensuche
  private static int dfs(int part, int i) {
    order[i] = part;
    data[part][0] = 1;
    ++i;
    for (int p : graph[part]) // alle Nachbarn
      if (p != -1 && data[p][0] == 0) // Nachbar existiert (nicht Rand) und nicht besucht
        i = dfs(p, i); // -> Tiefensuche hier fortfahren
    return i;
  }

  private static boolean addPart(int i) {
    if (i == numParts) return true; // Lösung gefunden
    int index = order[i];
    int[] given = data[index]; // gegebene Teilfiguren
    int[] rotations = new int[3];
    int numRot;
    for (int j = 0; j < distinctParts; ++j) { // alle Teile durchsuchen
      if (available[j] == 0) continue;
      if (parts[j][0] == parts[j][1] && parts[j][0] == parts[j][2]) {
        numRot = 1; // nur gleiche Figurenteile
        rotations[0] = 0;
      } else {
        numRot = 0;
        for (int r = 0; r < 3; ++r) {
          boolean matches = true; // alle Rotationen überprüfen
          for (int k = 0; k < 3 && matches; ++k) matches = given[k] == 0 || given[k] == parts[j]
                              [(r + k) % 3];
          if (matches) rotations[numRot++] = r;
        }
      }
      --available[j]; // Teil formal anlegen
      for (int ri = 0; ri < numRot; ++ri) { // alle passenden Rotationen nacheinander anwenden
        int r = rotations[ri];
        data[index] = new int[3];
        for (int k = 0; k < 3; ++k) {
          data[index][k] = parts[j][(k + r) % 3];
          int o = graph[index][k]; // alle Daten in die Arrays einschreiben, auch für Nachbarn
          if (given[k] == 0 && o != -1) data[o][k] = -data[index][k];
        }
        // Lösung gefunden? -> sofort abbrechen
        if (addPart(i + 1)) return true;
      }
      ++available[j]; // Teil formal wegnehmen
    }
    // aufräumen
    data[index] = given;
    for (int k = 0; k < 3; ++k) {
      int o = graph[index][k];
      if (given[k] == 0 && o != -1) data[o][k] = 0;
    }
    // keine Lösung gefunden
    return false;
  }
}
```