Ein Puzzle aus zwölf Teilen … und einem weiteren

Einleitung

Neben dem üblichen Bild, das mit dem Begriff Puzzle assoziiert, einer Menge an kleinen Papp-Quasiquadraten mit Ein- und Ausbuchtungen, gibt es weitere -korrektur: eine schier unüberblickbare Anzahl an- Spielarten.

Um eine dieser dreht sich folgende Schrift.

Wo sind Puzzles erwerbar

Der Online-Shop iii-kuh230 bietet sehr viele Arten von Puzzles an. Im Segment der Legepuzzle gibt es jene, die mit dem Schwierigkeitsgrad "fast unmöglich" prämiert werden. Und um ein solches dreht sich folgende Schrift.

Aufgabe

Gegeben sei ein Puzzle, bestehend aus zwölf Teilen und einem weiteren, welches wir vorerst ignorieren können.

Die Puzzleteile werden im folgenden stets als *Elemente* bezeichnet, wenn es sich um die virtuellen Äquivalente handelt.

Diese zwölf Teile wirken erst einmal nicht besonders beeindruckend. Nach einigem hin und her ist es gar keine Schwierigkeit mehr, zehn Teile zu platzieren, so dann und wann auch mal elf. Wenn es dann noch richtig gut läuft, wurden elf Teile gelegt und es gibt nur eine verbleibende Lücke und nicht fünf, sechs kleinere Lücken.

Konsequenz der Puzzle-Struktur

Aber hier endet es dann auch wieder und wieder.

Das Problem ist, dass es keine Rückmeldung gibt, ob zwei Teile, die nebeneinander liegen, zusammenpassen oder nicht. Dies führt zu einem gewaltigen Komplexitäts-Problem, denn jedes Teil könnte nun überall liegen und jedes andere beliebige Teil in beliebiger Rotation an einem beliebigen Nachbarfeld liegen.

Herumprobieren und die menschliche Rechenkapazität werden dem Problem wahrscheinlich nicht herr. Folglich lassen wir den Computer für uns rechnen. Welche Aufgabenstellung wird in welcher Art und Weise formuliert, damit eine sinnvolle Lösung in den Bereich des Möglichen rückt. Es wird benötigt:

- Ausgabe
  - Digitalrekonstruktion des Puzzles
  - Nachbau des hölzernen Originals ermöglichen
- Koordinatensystem
  - Spielfeld
  - *Elemente*
  - Kompatibilität zwischen Spielfeld und *Elementen* sicherstellen
- Algorithmen
  - Generierung der Lösungen
  - hochperformant
- Automatisierung

Ans Werk

Die wahrscheinlich vorerst am wenigsten wichtigste, jedoch leichteste Übung ist die Umsetzung der **Ausgabe**.

Hierfür können tausende und abertausende verschieden Wege begangen werden. Im *worst case* würde es in ASCII-Art-Manier im Terminal gedruckt werden können.

Python, mit vielen seiner kleinen Probleme, wie z.B. der Scope-Unzuverlässigkeit bei Arrays in Rekursionen (hierzu folgt ein Artikel), bietet jedoch mit der *Pygame*-Bibliothek einen niederschwelligen Zugang zu ansehnlicheren Darstellungsmöglichkeiten.

Zeichnen

Gezeichnet werden soll die Digitalrekonstruktion des Puzzles, das dem Original dahingehend ähnlich ist, dass der Nachbau, wenn denn eine Lösung vorliegt, mit der hölzernen Vorlage durchführbar ist.

Die Struktur zeigt eine Regelmäßigkeit, die nutzbar gemacht werden sollte. Ob nun der Spielfeldrand oder die zwölf Teile. Alles ist auf der Beschaffenheit des Sonderteils aufgebaut. Dieses Zwölfeck wird im Verlauf dieses Textes als *Atom* bezeichnet.

Atome sind nicht klein genug

Das *Atom* wird im Kontext des Koordinatensystems, weiter unten im Text, als kleinste Einheit betrachtet. Im Rahmen des Zeichnens wird es in *Subatome* unterteilt.

Bei genauer Betrachtung fällt auf, dass das *Atom* aus drei Sechsecken besteht. Für diese lassen sich Funktionen zur Bestimmung der Eckpunkte, die die Grundlage für das spätere Zeichnen sein werden, wesentlich leichter implementieren.

Aus Kantenlänge L wird, Pythagoras sei Dank, ohne Winkelberechnungen die Höhe H ermittelt.

$$H = \sqrt{L^2 - (\frac{L}{2})^2}$$

Es werden nun von einem Zentralpunkt (x, y) aus die Berechnungen vorgenommen und folgende neutrale Ergebnisse werden dabei ermittelt:

- $[x - H, y + \frac{L}{2}]$
- $[x - H, y - \frac{L}{2}]$
- $[x, y - L]$
- $[x + H, y - \frac{L}{2}]$
- $[x + H, y + \frac{L}{2}]$
- $[x, y + L]$

Pygame erledigt mittels `gfxdraw()` den Rest.

Da jedes Atom aus drei *Subatomen* besteht, die stets gleich zu einander positioniert sind, kann hier bereits bestehender Code verwendet werden.

Zirkelblume

Die Koordinaten (x, y) jedes *Subatom*-Zentrums kann durch ein viertes, virtuelles, *Subatom* dargestellt werden. Jede zweite äußere Koordinate dieses virtuellen *Subatoms* wird verwendet, um diese als Zentralkoordinaten der drei *Subatome* zu definieren. So wird erreicht, dass ein Koordinate ausreicht ein *Atom* an der richtigen Stelle zu zeichen.

Dies bildet die Grundlage einer *Schnittstelle* zwischen dem grafischen und kalkulatorischen Teil dieses Projekts. Um aus dieser Grundlage eine funktionale *Schnittstelle* zu entwickeln, muss die Gegenseite analysiert werden, damit abgeleitet werden kann, welche Daten zwingend übertragen werden müssen.

Spielfeld

Die Untersuchung der Struktur des Spielfeldes führt zu folgenden Ergebnissen:

- 61 Felder
  - Diagonale: neun Felder
  - nutzbare Außenkante: fünf Felder (Ecken überlagernd)
- Rahmen, bestehend aus *Atomen*
- symmetrischer Aufbau
  - Spiegelsymmetrie: negativ
  - Rotationssymmetrie: positiv, 120°

Koordinatensystem

Aus den oben genannten Erkenntnissen kann ein Koordinatensystem konstruiert werden.

Unter der Annahme, das zentrale Feld erhielt die Koordinate **0**, wäre die maximale Sprunganzahl um jedes beliebige andere Feld auf kürzestem Weg zu erreichen, vier.

Diese Sprünge bedürften eines definierten Bezugsrahmens. Jener könnte so konzipiert sein, dass eine Koordinate, nicht als Punkt verstanden würde, sondern als Pfad. Der Vorteil wäre, dass von jedem Punkt aus lediglich die Nachbarschaft relevant wäre und eine kleine festgelegte Statik, große dynamische Wirkkraft hätte.

Diesem skizzierten Konjunktiv folgend, wird das Koordinatensystem ein relatives Pfadsystem. Die Nachbarn eines Feldes werden im Uhrzeigersinn, rechts-oben beginnend, von **1** - **6** durchnummeriert und das Zentrum bleibt **0**.

Ein Sprung wird dabei immer von einem festgelegten Zentrum aus betrachtet und aufeinander folgende Sprünge werden aufgereiht in einem Objekt zusammengefasst.

Das Koordinatensystem für das Spielfeld selbst ist:

```
koordinatenInReihen =[[6, 6, 6, 6], [6, 6, 6, 1], [6, 6, 1, 1], [6, 1, 1, 1], [1, 1,
1, 1],
                      [6, 6, 6, 5], [6, 6, 6], [6, 6, 1], [6, 1, 1], [1, 1, 1], [1, 1,
1, 2],
                      [6, 6, 5, 5], [6, 6, 5], [6, 6], [6, 1], [1, 1], [1, 1, 2], [1,
1, 2, 2],
                      [6, 5, 5, 5], [6, 5, 5], [6, 5], [6], [1], [1, 2], [1, 2, 2],
[1, 2, 2, 2],
                      [5, 5, 5, 5], [5, 5, 5], [5, 5], [5], [0], [2], [2, 2], [2, 2,
2], [2, 2, 2, 2],
                      [4, 5, 5, 5], [4, 5, 5], [4, 5], [4], [3], [3, 2], [3, 2, 2],
[3, 2, 2, 2],
                      [4, 4, 5, 5], [4, 4, 5], [4, 4], [4, 3], [3, 3], [3, 3, 2], [3,
3, 2, 2],
                      [4, 4, 4, 5], [4, 4, 4], [4, 4, 3], [4, 3, 3], [3, 3, 3], [3, 3,
3, 2],
                      [4, 4, 4, 4], [4, 4, 4, 3], [4, 4, 3, 3], [4, 3, 3, 3], [3, 3,
3, 3]]
```

In einer modifizierten Version, in der Einzelsprünge nicht in Integer-Arrays, sondern in Integer hinterlegt wird, ist das Koordinatensystem für *Elemente*:

```
allePuzzleElemente = [[0, 3, 5, 1, [1, 1]],
                      [0, 5, 6, 2, [2, 3]],
                      [0, 4, 2, 1, [1, 6]],
                      [0, 5, 6, 3, [3, 4]],
                      [0, 3, 1, 6, [6, 5]],
                      [0, 2, 3, 5, [5, 6]],
                      [0, 1, 6, 3, [3, 3]],
                      [0, 2, 1, 6, [6, 6]],
                      [0, 5, 4, 3, [3, 3]],
                      [0, 3, [3, 3], 5, [5, 6]],
                      [0, 6, [6, 6], 3, [3, 2]],
                      [0, 6, [6, 6], 4, [4, 4]]]
```

`allePuzzleElemente` enthält die Repräsentationen aller zwölf *Elemente*.

Rotation

Wenn ein *Atom* um 120° rotiert wird, ist es, aufgrund der Struktur, wieder in der Ausgangslage. Bei *Elementen* ist dies nicht der Fall. Nach 120° Rotation ist das *Element* wieder im Einklang mit der atomaren Ausrichtung des Puzzles, jedoch muss das *Element*, im Sinn der `allePuzzleElemente`-Liste, als neues *Element* betrachtet werden. Die Anzahl der *Elemente* verdreifacht sich auf 36.

Dies wird bei der weiter unten stehenden Betrachtung der Kombinationsmöglichkeiten bedeutsam sein.

Um ein *Element* zu rotieren, werden die Relationen zu den Nachbarn neu bestimmt.

```
def rotationsFaktor(rf):
    return (rf + 2) % 6 or 6
```

`rotationsFaktor()`, als Kernfunktion, übernimmt diese Aufgabe. Diese wird auf jede *Atom*-Position eines *Elements* angewandt und rotiert das *Element* somit um 120° in Uhrzeigersinn.

Elemente platzieren

Um den Akt, ein Puzzleteil auf das Spielfeld zu legen, als Funktionalität zur Verfügung zu haben, wird ein Array aus dem `koordinatenInReihen`-Array ausgewählt und jedem Eintrag des Koordinatensystems des *Elements* hinzugefügt.

Zur Veranschaulichung ein Beispiel:
`[0, 5, 6, 2, [2, 3]]` wird mit dem Zentrum **0** auf `[4, 4, 3]` positioniert. Der neue Koordinatensatz für dieses *Element* ist `[[4, 4, 3], [4, 4, 3, 5], [4, 4, 3, 6], [4, 4, 3, 2], [2, 3, 4, 4, 3]]`.

Pfad abkürzen

Diese Koordinaten, oder an dieser Stelle nochmals erwähnt, Pfade, sind in einem nicht-formalisierter Zustand. Ein Abgleich, ob eine dieser Positionen bereits von einem anderen *Element* belegt ist, erweist sich als schwierig, da ein reiner Vergleich der Sprünge des Pfades mit denen eines anderen *Elements* nicht zwangsläufig sichere Aussagen über den Zustand eines Feldes (frei, belegt) bringen würde. Anzunehmen ist ein anderer Pfad, der das gleiche Feld belegt.

Die Formalisierung, die vorzunehmen ist, sieht eine Kürzung der Einträge in den Koordinatensätzen vor, die durch das Platzieren der *Elemente* erzeugt werden, bis die Einträge der Norm der `koordinatenInReihen` (siehe oben) enstpricht.

Da es wenige aber gewichtige Regeln gibt, die sich aus der Pfadgenerierung anhand von Nachbarn ableiten lassen, kommt ein Switch zu Einsatz.

```
def moeglicheKombinationen(a, b):
    x = sorted((a, b))
    match x:
        case (1, 4):
            return 0
        case (2, 5):
            return 0
        case (3, 6):
            return 0
        case (1, 3):
            return 2
        case (2, 4):
            return 3
        case (3, 5):
            return 4
        case (4, 6):
            return 5
        case (1, 5):
            return 6
        case (2, 6):
            return 1
        case _:
            return a, b
```

Ein alternativer Ablauf innerhalb von `moeglicheKombinationen()` wäre möglich gewesen, aber das `match`-Statement funktioniert an dieser Stelle sehr gut und ist seht leicht zu überblicken. `moeglicheKombinationen()` wird durch eine andere rekursive Funktion aufgerufen, die alle Arrays eines *Elements* separat iteriert, und je Paare auswählt und übergibt. Aus `[1, 1, 3, 4, 6]` wird **1**.

Das bereits begonnene Beispiel wird an dieser Stelle fortgesetzt und der Koordinatensatz `[[4, 4, 3], [4, 4, 3, 5], [4, 4, 3, 6], [4, 4, 3, 2], [2, 3, 4, 4, 3]]` weiterhin exemplarisch betrachtet.

```
[4, 4, 3] --> [4, 4, 3]
[4, 4, 3, 5] -case (3, 5)-> [4, 4, 4]
[4, 4, 3, 6] -case (4, 6)-> [5, 4, 3] -case (3, 5)-> [4, 4]
[4, 4, 3, 2] -case (2, 4)-> [3, 4, 3]
[2, 3, 4, 4, 3] -case (2, 4)-> [3, 3, 4, 3]
result = [[4, 4, 3], [4, 4, 4], [4, 4], [3, 4, 3], [3, 3, 4, 3]]
```

`result` enthält ausschließlich Koordinaten, die in `koordinatenInReihen` vorhanden sind. Die Reihenfolge der Pfade wird im Weiteren durch `sort()` keinen Einfluss mehr haben.

Abstraktion 1

Da von diesem Stand an, keine Notwendigkeit mehr besteht, dass mit den konkrete Koordinaten gearbeitet werden muss, reicht der Verweis auf die Position innerhalb von `koordinatenInReihen`. Das Beispiel hat somit die Koordinaten `[52, 51, 45, 53, 59]`.

Zwischenfazit

Der Ablauf bisher:

1. Koordinaten werden bereitgestellt
2. aus `koordinatenInReihen` werden die Pfade ermittelt
3. Pfade werden vom Zentrum des Spielfeldes betrachtet und je Integer um eine *Atom*größe in die jeweilige Richtung verschoben um die Zentren der *Atome* zu berechnen
4. Von den Zentren ausgehend werden die *Atome* durch die Erstellung der *Subatome* gezeichnet

<details>
<summary>Todo</summary>

- Ausgabe (erledigt)
  - Digitalrekonstruktion des Puzzles (erledigt)
  - Nachbau des hölzernen Originals ermöglichen (erledigt)
- Koordinatensystem (erledigt)
  - Spielfeld (erledigt)
  - *Elemente* (erledigt)
  - Kompatibilität zwischen Spielfeld und *Elementen* sicherstellen (erledigt)
- Algorithmen
  - Generierung der Lösungen
  - hochperformant
- Automatisierung

</details>

Es wurde ein Koordinatensystem entwickelt, welches sowohl abstrahiert werden kann, als auch von den grafischen Funktionen interpretiert werden kann. Dieses ist für das Spielfeld und die einzelnen Puzzleteile anwendbar. Unnötige Information werden durch Transformationen aufgelöst.

Die Ausgabe erzeugt ein Abbild des Puzzles, welches geeignet ist, genaue Instruktionen für den Nachbau mit dem hölzernen Original.

Algorithmisch eine Lösung zu finden wurde noch nicht versucht.

Konstruktion eines Lösungsalgorithmus

Bei theoretischen $3 \cdot 61 = 183$ Positionen pro *Element* und 12 *Elementen* an der Zahl, sind dies 183^{12} Kombinationen, was etwa $1.4 \cdot 10^{27}$ möglichen Lösungen entspricht. Um diese Größenordnung zu erreichen braucht es: 1 Milliarde · 1 Milliarde · 1 Milliarde.

Filter

Aufgrund der Unzahl an Kombinationsmöglichkeiten muss der Ausgangsdatensatz reduziert werden.

Unmögliches

Ein erster Schritt ist, zu verwerfen, was nicht möglich ist.

Hierzu werden, wie oben bereits vorbereitet, die *Elemente* nicht weiterverarbeitet, die über das Spielfeld hinausragen. Dies reduziert die realistischen Positionen pro *Element* auf die Hälfte, je zwischen 84 und 108.

Die Gesamtanzahl der Kombinationen ist $641.805.762.426.921.958.440.960$.

Rotation

Der zweite Schritt ist, bei EINEM *Element* die Rotation nicht durchzuführen, denn die Rotation für dieses *Element* kann nachträglich mit dem Drehen des physischen Puzzle nachgeholt werden.

Die Gesamtanzahl der Kombinationen ist $213.935.254.142.307.319.480.320$.

Abstraktion 2

Von diesem, dem dritten, Schritt an wird die Programmiersprache gewechselt. Eine möglichst Hardware-nahe Sprache, wie *C* böte sich an. Die Entscheidung fiel auf *Rust*

Da das Spielfeld 61 Felder hat, besteht die Möglichkeit, die Arrays mit den fünf Integer, die in Kapitel *Abstraktion 1* generiert wurden weiter zu abstrahieren. Dies geschieht, indem für jedes Array ein `u64`, unsigned Integer, erzeugt wird, und nur die Bit-Positionen auf `TRUE` gesetzt werden, die als Integer in dem Array vorkommen.

```
fn set_bit(value: &mut u64, n: i32) {
    *value |= 1u64 << n;
}

fn collect_all_binarys(input: Vec<Vec<[i32; 5]>>) -> Result<Vec<Vec<u64>>, Box<dyn
std::error::Error>> {
...
    let mut value: u64 = 0;
    for k in 0..=4{
        set_bit(&mut value, input[i][j][k]);
    }
...
}
```

Diese Transformation eröffnet Möglichkeiten auf Bit-Ebene zu operieren, was die Prozesse enorm beschleunigt.

Unnötiges

Der vierte Schritt sieht vor, Teilfilterungen vorzunehmen.

Hierfür werden je ein `u64` von zwei verschiedenen *Elementen* miteinander addiert. Danach wird mittels `count_ones()` geprüft, ob die gesetzten Bits **10** sind. Ist dies der Fall, wurden beide *Elemente* auf dem virtuellen Spielfeld platziert, ohne dass eine Überschneidung stattfand. Mit diesem neuen `u64` wird weitergearbeitet.

```
fn binary_addition(input: Vec<Vec<u64>>) -> Result<Vec<Vec<u64>>, Box<dyn
std::error::Error>> {
...
    let value = input[i][l1] + input[i + in_len_half][l2];
    if value.count_ones() == bitcount_goal {
        list.push(value);
    }
...
}
```

Aus zwölf *Elementen* werden sechs hypothetische *Elemente*

Dieser Schritt wird wiederholt, um drei Listen mit hypothetischen *Elementen* zu erzeugen. Jedes `u64` hat danach **20** gesetzte Bits, enthält somit die Positionen von vier *Elementen*, die sich nicht überlagern. Die drei Listen haben je einige Millionen Einträge.

Die Gesamtanzahl der Kombinationen ist $68.186.544.323.915.282.760$.

60 Bits

Nachdem *Multithreading* implementiert und ein *Log*-System für verworfene Kombinationen eingerichtet wurde, wurde der Server vorbereitet. Nach kleineren Anpassungen ist *Multithreading* für diese Hardware-Software-Kombination optimiert.

Sobald das Programm auf dem Server eingerichtet ist und nach Lösungen sucht, werden gefundene Resultate, bei denen 60 Bits gesetzt sind in eine *txt*-Datei geschrieben. Ein Resultat besteht aus den drei `u64` mit je 20 gesetzten Bits, deren Genese weiter oben erläutert wurde. Ein Beispiel für ein solches Ergebnis ist `18161482951001600 224115895571578983 2063565630690589080`.

Die drei `u64` werden wieder in eine Form transformiert werden müssen, die der grafische Teil dieses Programms versteht. Dafür wird der Prozess der Erzeugung der `u64` wiederholt, um bei Übereinstimmung, die beiden Summanden festzustellen. Dies wird zuerst für die 20-Bit- und danach die 10-Bit-`u64` durchgeführt. Das Ermitteln der Bit-Position in den 5-Bit-`u64` wird mit `get_bit_positions()` durchgeführt. Diese druckt ein Array von fünf Integer als Array in die Konsole.

```
fn get_bit_positions(ns: Vec<u64>) {
    let result: Vec<Vec<u32>> = ns
        .iter()
        .map(|&n| (0..64).filter(|&i| n & (1 << i) != 0).collect())
        .collect();
    for (_index, positions) in result.iter().enumerate() {
```

```
        println!("{:?}", positions);
    }
}
```

Programmstart

Um das Programm unter Windows als *exe* starten zu können muss das *Rust*-Zurückrechnungsprogramm kompiliert werden und als `Subprocess` im *Python*-Hauptprogramm eingebunden werden. Mit der *Tkinter*-Bibliothek wird ein einfaches Eingabefenster erzeugt, das drei Zeilen zum Zahlenhineinkopieren hat und einen *Absenden*-Button. Die drei Zahlen werden dem `Subprocess` übergeben und innerhalb des *Rust*-Programms in `u64` geparst.

Die Rückrechnung wird durchgeführt und zwölf Integer-Arrays mit je einer Länge von fünf in die Konsole geschrieben. Diesen Druck nimmt Das Hauptprogramm wieder auf und dekodiert die Ausgaben.

```
current_dir = os.path.dirname(os.path.abspath(__file__))
puzzle_path = os.path.join(current_dir, "puzzle")
process = subprocess.Popen([puzzle_path, str(nums[0]), str(nums[1]), str(nums[2])]
                           , stdout=subprocess.PIPE
                           , stderr=subprocess.PIPE)
stdout, stderr = process.communicate()

loesung = []
for line in stdout.decode("utf-8").split('\n')[:-1]:
    l = line.strip('[]')
    int_array = list(int(num) for num in l.split(','))
    loesung.append(int_array)
```

Im weiteren wird der `Gameloop` gestartet, damit das Ergebnis gezeichnet werden kann. Innerhalb des *Loops* wird der grafische Teil des Programms aufgerufen, um die *Elemente* aus Puzzleteile zeichnen zu können.

```
while running:
    window.fill((0,0,0))

    # Spielfeld - Rahmen
    myPieces.spielfeldrahmen(window)
    for i, l in enumerate(loesung):
        farbe = (i * 15 + 45, i * 10 + 65, i * 5 + 100)
        for j in loesung[i]:
            myPieces.positioniereElement(window, myLogic.koordinatenInReihen[j],
farbe)

    # Refresh Ausgabe
    pygame.display.flip()

    # EVENTS
    for event in pygame.event.get():
```

```
        if event.type == pygame.QUIT:
            running = False

# Quit Pygame
pygame.quit()
```

Dauer

Im Vergleich zu den ursprünglichen 183^{12} Möglichkeiten wurde mit einem Datensatz von $\frac{1}{20.687.804}$ der Originalgröße gearbeitet.

Der Server, auf dem das Suchprogramm ausgeführt wird, schafft etwa ein Prozent pro Tag. Daraus folgt, dass der Originaldatensatz, in ein `u64`-Format transformiert, etwa 5.667.891 Jahre bräuchte und ohne diese, aufgrund der bis zu 20 Integer pro *Element* und einer mäßigen Vergleichsgeschwindigkeit bei Arrays, etwa 5.000.000.000 Jahre.

Fazit

Die Nichtbeachtung des Sonderteils war die erste richtige Entscheidung. Bei allen bisher gefunden Lösungen, war das Rotieren des Spielfeldes ausreichend, um die korrekte Ausrichtung zu finden, damit auch das Sonderteil seinen Platz in den Vorgaben findet.

Die 100 Tage, die es letztlich geworden sind, scheinen ein passables Ergebnis zu sein.

Es gibt sicherlich effizientere Methoden dieses Puzzle zu lösen. Dieser Weg war lediglich optimierte *BruteForce* und ein Wegbereiter für das Interesse an *Data Oriented Design (DoD)*.

Wer sich dieses Problems ebenfalls annehmen möchte, findet meinen Code unter meinem GitHub-Account. Und schauen Sie gern bei iii-kuh230.de vorbei.