Aufgabe 2: Die goldene Mitte

Team-ID: 00779
Team: tiantianquan
Autor: Florian Werth
17. November 2023

Inhaltsverzeichnis

Lösungsidee

Man stellt die Box als dreidimensionales integer Feld [X, Y, Z] dar. In der Mitte des Feldes markiert man den goldenen Würfel. Man legt die Quader von links nach rechts und von unten nach oben in die Box. Man fängt bei [0, 0, 0] an Quader zu platzieren, bis eine Ebene vollständig belegt ist, dann geht man in die nächste Ebene. Kann man eine Ebene nicht vollenden, geht man mittels Backtracking zurück und platziert auf andere Weise einen Quader. Ein Quader kann dabei auf 6 verschiedene Arten platziert werden (Es gibt 3 Flächen auf die man ihn legen kann und jeweils noch die Entscheidung ob längs oder quer).

Umsetzung

Die Lösung wurde in C++ programmiert.

Für die Lösungsidee wurde eine rekursive Backtracking-Funktion geschrieben.

Als Parameter nimmt sie ein 3D-Array, das die Box repräsentiert, ein Array, das die Indizes der noch benutzbaren Quader enthält und die derzeitige Position in der Box.

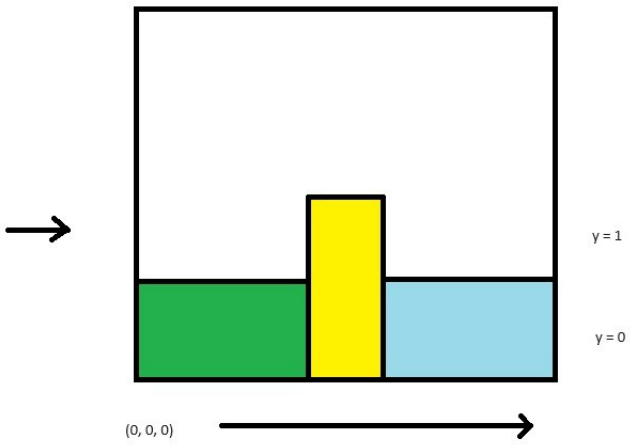
Ein Quader wird als Struct mit Breite, Höhe, Länge gespeichert.

Der Backtracking-Algorithmus funktioniert wie folgt:

Der erste Aufruf: backtracking(Box, Quaderliste, Position=0,0,0)

für jeden Quader wird in jeder möglichen Ausrichtung überprüft, ob man ihn an die derzeitige Position setzen kann:

1. Passen die Ausmaße des Quaders an der derzeitigen Position überhaupt noch in die Box?
2. Ist nach rechts hin Platz? Ein anderer Quader könnte die Fläche versperren. Das nachfolgende Bild versucht das zu veranschaulichen.



In der ersten Reihe (y=0) hat man bereits Quader platziert. Man geht immer von links nach rechts. Bei der nächsten Reihe könnte es nun sein, dass ein Quader aus der vorherigen Reihe herausragt.

3. Mit eine If-Abfrage wird überprüft, ob der Quader sich mit dem goldenen Würfel überschneidet.

```
if ((x <= mid && mid <= x + o.x - 1) &&
    (y <= mid && mid <= y + o.y - 1) &&
    (z <= mid && mid <= z + o.z - 1))
{
    continue;
}
```

4. Ist Platz für den Quader wird eine Kopie des Box-Arrays gemacht und der Quader wird in der Kopie mit seinem Index als Wert markiert. (nachfolgendes Bild)

```
for (int xScout = 0; xScout < o.x; xScout++)
{
    for (int yScout = 0; yScout < o.y; yScout++)
    {
        for (int zScout = 0; zScout < o.z; zScout++)
        {
            boxCopy[x + xScout][y + yScout][z + zScout] = c + 1;
        }
    }
}
```

   Es wird auch eine Kopie des Quader-Indizes-Array gemacht und in der Kopie wird der gerade platzierte Quader herausgelöscht.

   Mit den beiden Kopien wird backtracking(BoxKopie, IndizesKopie, X+1, Y, Z) aufgerufen.

Am Anfang des Funktionsaufrufs wird zuerst überprüft

Ist X >= der Boxlänge wird X auf 0 gesetzt und Y++

Ist Y >= der Boxlänge wird Y auf 0 gesetzt und Z++

Ist Z >= der Boxlänge ist die Lösung gefunden und wird anhand des derzeitigen BoxArrays ausgegeben

Optimierungen

Ausrichtungen

Für einen Quader gibt es zwar sechs verschiedene Ausrichtungen, doch sind alle Seiten gleich lang sind sie alle gleich. Anstatt alle möglichen Ausrichtungen zu benutzen sollte man nur die nicht identischen Ausrichtungen benutzen bei einem Würfel z.B. nur 1 und bei einem Quader mit quadratischer Grundfläche 3

Gruppieren

Quader die gleich sind kann man zu Gruppen zusammenfassen, z.B. bei

1 1 1

1 1 1

1 1 1

2 2 2

2 2 2

hätte man dann anstatt 6 Möglichen einen Quader zu setzen nur noch 2 Möglichkeiten, nämlich entweder 1 1 1 oder 2 2 2.

Beispiele

Raetsel1.txt

i: 1 | 1 1 3

i: 2 | 1 1 3

i: 3 | 1 1 2

i: 4 | 1 1 2

i: 5 | 1 2 2

i: 6 | 1 2 2

i: 7 | 1 2 2

i: 8 | 1 2 2

Found solution in 511 steps

Ebene 0

2 8 8

4 8 8

1 1 1

Ebene 1

2 7 7

4 G 6

3 3 6

Ebene 2

2 7 7

5 5 6

5 5 6

Raetsel2.txt

i: 1 | 1 3 3
i: 2 | 1 3 3
i: 3 | 1 1 1
i: 4 | 1 1 1
i: 5 | 1 1 3
i: 6 | 1 1 3
Found solution in 61 steps
Ebene 0

```
2  2  2
6  4  5
1  1  1
```

Ebene 1

```
2  2  2
6  G  5
1  1  1
```

Ebene 2

```
2  2  2
6  3  5
1  1  1
```

Raetsel3.txt

keine Lösung gefunden

Raetsel4.txt

i: 1 | 1 2 4

i: 2 | 2 2 2

i: 3 | 2 2 2

i: 4 | 2 2 2

i: 5 | 2 2 2

i: 6 | 1 1 5

i: 7 | 1 1 5

i: 8 | 1 1 5

i: 9 | 1 1 5

i: 10 | 1 1 5

i: 11 | 1 1 5

i: 12 | 1 1 5

i: 13 | 1 1 5

i: 14 | 1 1 5

i: 15 | 1 2 3

i: 16 | 1 2 3

i: 17 | 1 2 3

i: 18 | 1 2 3

i: 19 | 1 2 3

i: 20 | 1 3 3

Found solution in 827881 steps

Ebene 0

1 1 3 3 11
5 5 3 3 11
5 5 19 14 11
4 4 19 13 11
4 4 19 12 11

Ebene 1

1 1 3 3 10
5 5 3 3 10
5 5 19 14 10
4 4 19 13 10
4 4 19 12 10

Ebene 2

1 1 16 16 16
9 9 9 9 9
18 18 G 14 20
17 17 17 13 20
17 17 17 12 20

Ebene 3

1 1 16 16 16
8 8 8 8 8
18 18 15 14 20
2 2 15 13 20
2 2 15 12 20

Ebene 4

7 7 7 7 7
6 6 6 6 6
18 18 15 14 20
2 2 15 13 20
2 2 15 12 20

Raetsel5.txt

i: 1 | 2 2 3
i: 2 | 2 2 3
i: 3 | 2 2 3
i: 4 | 2 2 3
i: 5 | 2 2 3
i: 6 | 2 2 3
i: 7 | 1 2 4
i: 8 | 1 2 4
i: 9 | 1 2 4
i: 10 | 1 2 4
i: 11 | 1 2 4
i: 12 | 1 2 4
i: 13 | 1 1 1
i: 14 | 1 1 1
i: 15 | 1 1 1
i: 16 | 1 1 1
Found solution in 16938508 steps

Ebene 0

12	12	5	5	10
6	6	5	5	10
6	6	5	5	10
11	11	11	11	10
11	11	11	11	16

Ebene 1

12	12	5	5	10
6	6	5	5	10
6	6	5	5	10
4	4	4	15	10
4	4	4	9	9

Ebene 2

12	12	3	3	3
6	6	3	3	3
6	6	G	2	2
4	4	4	2	2
4	4	4	9	9

Ebene 3

12	12	3	3	3
8	14	3	3	3
8	1	1	2	2
8	1	1	2	2
8	1	1	9	9

Ebene 4

13	7	7	7	7
8	7	7	7	7
8	1	1	2	2
8	1	1	2	2
8	1	1	9	9

Quellcode

Anmerkung: Der Code ist beim Umsetzen der Optimierungen etwas unleserlich geworden.

Generierung der Ausrichtungen

```
std::vector<cuboid> generate_orientations(const std::array<int, 3>& c)
{
	if ((c[0] == c[1]) && (c[0] == c[2]))
	{
		std::vector<cuboid> orientations
		{
			{c[0], c[1], c[2]},
		};
		return orientations;
	}

	else if ((c[0] == c[1]) && (c[0] != c[2]))
	{
		std::vector<cuboid> orientations
		{
			{c[0], c[1], c[2]},
			{c[0], c[2], c[1]},
			{c[2], c[1], c[0]},
		};
		return orientations;
	}

	else if ((c[0] == c[2]) && (c[0] != c[1]))
	{
		std::vector<cuboid> orientations
		{
			{c[0], c[1], c[2]},
			{c[0], c[2], c[1]},
			{c[1], c[0], c[2]},
		};
		return orientations;
	}

	else if ((c[1] == c[2]) && (c[1]) != c[0])
	{
		std::vector<cuboid> orientations
		{
			{c[0], c[1], c[2]},
			{c[1], c[0], c[2]},
			{c[2], c[1], c[0]},
		};
		return orientations;
	}

	else
	{
		std::vector<cuboid> orientations
		{
			{c[0], c[1], c[2]},
			{c[1], c[0], c[2]},

			{c[0], c[2], c[1]},
			{c[2], c[0], c[1]},

			{c[2], c[1], c[0]},
			{c[1], c[2], c[0]},

		};
		return orientations;
	}
}
```

Backtracking-Funktion

```
int iteration = 0;
int found = false;

void backtracking_box(std::vector<std::vector<std::vector<int>>> box, int x, int y,
int z, std::vector<std::vector<int>> cuboidGroupedIndices, const
std::vector<std::array<int, 3>>& cuboids, int mid, int size)
{
	if (found)
		return;


	if (x >= size)
	{
		x = 0;
		y += 1;
	}
	if (y >= size)
	{
		y = 0;
		z += 1;
	}

	if (z >= size)
	{
		//Lösung wurde hier ausgegeben
		found = true;
	}

	if (box[x][y][z] > 0)
	{
		backtracking_box(box, x + 1, y, z, cuboidGroupedIndices, cuboids, mid,
		size);
	}

	for (int groupIndex = 0; groupIndex < cuboidGroupedIndices.size(); groupIndex++)
	{
		if (cuboidGroupedIndices[groupIndex].size() == 0)
		{
			continue;
		}
		std::vector<cuboid> orientations =
		 generate_orientations(cuboids[cuboidGroupedIndices[groupIndex][0]]);
		for (const auto& o : orientations)
		{
			if (x + o.x > size || y + o.y > size || z + o.z > size)
			{

				continue;
			}

			if ((x <= mid && mid <= x + o.x - 1) &&
				(y <= mid && mid <= y + o.y - 1) &&
				(z <= mid && mid <= z + o.z - 1))
			{
				continue;
```

```
                        }

                        bool canPlace = true;
                        for (int xScout = 0; xScout < o.x; xScout++)
                        {

                                    if (box[x + xScout][y][z] > 0)
                                    {
                                          canPlace = false;
                                    }

                        }

                        if (canPlace)
                        {
                              auto boxCopy(box);
                              for (int xScout = 0; xScout < o.x; xScout++)
                              {
                                    for (int yScout = 0; yScout < o.y; yScout++)
                                    {
                                          for (int zScout = 0; zScout < o.z; zScout++)
                                          {
                                                boxCopy[x + xScout][y + yScout][z+zScout]
                                            = cuboidGroupedIndices[groupIndex].back() + 1;
                                          }
                                    }
                              }

                              auto cuboidGroupedIndicesCopy(cuboidGroupedIndices);
                              cuboidGroupedIndicesCopy[groupIndex].pop_back();

                              backtracking_box(boxCopy, x + 1, y, z,
                              cuboidGroupedIndicesCopy, cuboids, mid, size);
                        }

                  }
            }
      }
```

Gruppieren

```
std::vector<std::vector<int>> cuboidsGrouped;
std::vector<std::vector<std::vector<int>>> groupMapping(size,
std::vector<std::vector<int>>(size, std::vector<int>(size, -1)));

for (int i = 0; i < cuboids.size(); i++)
{
	std::sort(cuboids[i].begin(), cuboids[i].end());
	if (groupMapping[cuboids[i][0] - 1][cuboids[i][1] - 1][cuboids[i][2] - 1] > - 1)
	{
		cuboidsGrouped[groupMapping[cuboids[i][0] - 1][cuboids[i][1] - 1]
		[cuboids[i][2] - 1]].push_back(i);
	}
	else
	{
		cuboidsGrouped.push_back(std::vector<int>{{i}});
		groupMapping[cuboids[i][0] - 1][cuboids[i][1] - 1][cuboids[i][2] - 1] =
		cuboidsGrouped.size() - 1;
	}

}
```