Aufgabe 2: Simultane Labyrinthe

Teilnahme-ID: 73795

Bearbeiter/-in dieser Aufgabe:
Timon Retzlaff

26. April 2025

Inhaltsverzeichnis

Lösungsidee

Algorithmus

Man sieht die beiden Labyrinthe als ein 4D-Labyrinth, wobei die Position im 4D-Labyrinth aus den x- und y-Koordinaten der beiden 2D-Labyrinthe besteht.

Das 4D-Labyrinth nenne ich ab jetzt Superlabyrinth und die 4D-Position ab jetzt Superposition. Wenn ich von Position oder Labyrinth rede, meine ich eine 2D-Position / -Labyrinth.

Die Superposition ist also (x1 | y1 | x2 | y2), wobei (x1 | y1) die Position im einen und (x2 | y2) die Position im anderen Labyrinth ist. Wenn man zum Beispiel nach rechts geht und beide Personen nicht blockiert sind, ist die neue Superposition (x1 + 1 | y1 | x2 + 1 | y2).

In diesem Superlabyrinth kann man dann A*-Suche anwenden. Diese findet den optimalen Weg und ist dabei optimal effizient, was heißt, dass sie den kürzesten Weg findet und dabei eine minimale Anzahl an Knoten für eine bestimmte Heuristik expandiert. Dafür muss die verwendete Heuristik allerdings monoton sein. Zusätzlich muss bei einer monotonen Heuristik jeder Knoten nur einmal expandiert werden.

Die beste monotone Heuristik, welche ich gefunden habe, ist ein gewichteter Durchschnitt der Einzellösungen der Labyrinthe. Dabei hat die längere Einzellösung ein deutlich höheres Gewicht. In meinem Fall 0.99999. Also: $h(l_1, l_2) = max(l_1, l_2) \cdot 0.99999 + \min(l_1, l_2) \cdot 0.00001$.
Die Einzellösungen sind die Längen der Lösungen der beiden Labyrinthe.

Diese können vorberechnet werden, indem man vom Ziel aus Flood-Fill ausführt und sich für jedes Feld die kürzeste Schrittzahl bis dort merkt.

Mit diesen kann man auch schnell bestimmen, ob eine Lösung des Superlabyrinthes existiert. Wenn beide Labyrinthe eine Lösung haben, kann man zuerst die eine gehen, dann bei dem anderen den Weg zurück und dann den ursprünglichen Lösungsweg. Deshalb hat das Superlabyrinth immer eine Lösung, wenn beide Labyrinthe eine Lösung haben. Wenn eins oder beide der Labyrinthe keine Lösung hat, hat auch das Superlabyrinth keine Lösung.

Heuristiken

Ich musste eine A* Heuristik finden, welche die Restweglänge nie überschätzt, trotzdem möglichst nah an der echten Entfernung zum Ziel liegt und monoton ist. (siehe Wikipedia)

Manhattan-Distanz

Zuerst habe ich die Manhattan-Distanz der Superpositionen der Start- und Endfelder probiert. Diese Heuristik überschätzt nicht und ist monoton, aber leider hat diese Heuristik viel zu niedrig geschätzt.

Maximale Einzellösung

Wenn man stur den längeren Einzelweg (Länge L_{Max}) verfolgt, kann es sein, dass der kürzere Einzelweg (Länge L_{Min}) mitgelöst wird. Der längere Einzelweg ist also die kleinste mögliche Gesamtlösung. Deshalb darf eine Heuristik nicht höher schätzen.

Das Problem mit der Heuristik:

$h(l_1, l_2) = max(l_1, l_2) = L_{Max}$

ist, dass zwei Lösungen mit der Länge L_{max} immer gleich bewertet werden, ohne L_{Min} zu betrachten, weshalb von diesen eine zufällige ausgesucht wird.

Dementsprechend war auch diese Heuristik langsam.

OneOverMin

Diese Heuristik ist $h(l_1, l_2) = max(l_1, l_2) – 1/(min(l_1, l_2)+1)$.

Die Idee davon war, das Problem mit dem Ignorieren der kleineren Einzellösung zu eliminieren.

Leider war diese Heuristik aber nicht monoton. Deshalb mussten Knoten mehrfach expandiert werden, weshalb diese Heuristik keine gute Wahl ist.

WeightedAverage

Diese Heuristik ist $h(l_1, l_2) = max(l_1, l_2) \cdot 0.99999 + \min(l_1, l_2) \cdot 0.00001$.

Hier wird der längere Weg viel mehr gewichtet, aber der kürzere wird auch beachtet. Zudem ist diese Heuristik monoton.

Laufzeitbestimmung

Worst-Case

Das Einfügen einer neuen Superposition in den Heap, welcher die Open-List des A*-Algorithmus repräsentiert, hat eine Laufzeit von $O(log(n))$. n ist dabei die Größe des Heaps. Es muss zwar jeder seiner Nachbarn eingefügt werden, davon gibt es aber maximal 4, da es nur 4 mögliche Bewegungen gibt, was also auch $O(4log(n)) = O(log(n))$ hat.

Die Laufzeit einer Expandierung ist also $O(log(n)+log(n)) = O(2log(n)) = O(log(n))$.

Die Größe des Heaps ist maximal so groß, wie alle Knoten die man entdeckt. Da man sich maximal alle Superpositionen des Superlabyrinthes angucken muss, welches l^2m^2 hat, wenn l und m die Länge und Höhe der Labyrinthe sind, hat der Heap eine maximale Größe von l^2m^2, wodurch das Expandieren eines Knoten eine Worst-Case-Laufzeit von $O(log(l^2m^2)) = O(log(lm))$ hat.

Da man im Worst-Case alle Superpositionen expandieren muss, wird das jeweils erste Element des Heaps l^2m^2-mal extrahiert und dessen Nachbarn werden weniger oft als l^2m^2-mal hinzugefügt.

Daher liegt die Worst-Case-Laufzeit von A* in dem Superlabyrinth in $O(l^2m^2log(lm)+l^2m^2log(lm)) = O(l^2m^2 \cdot log(lm))$.

Best-Case

Im Best-Case ist die Größe des Heaps immer 1. Dadurch hat das Einfügen von Elementen in den Heap $O(log(1)) = O(1)$.

Die Anzahl der Superpositionen die man im Best-Case betrachtet ist die Manhattan-Distanz von Start und Ziel im 2D-Labyrinth. Diese ist in jenem $l+m$.

Dadurch ist die Best-Case-Laufzeit von dem Algorithmus $O((l+m) \cdot 1) = O(l+m)$.

Umsetzung

Übersicht

Mein Code besteht aus 12 Klassen und einem Interface: *Main, LabyrinthSolver, Labyrinths, Labyrinth, Move, Vector2, PositionData, StateTracker, State, StateMove, Field, WeightedAverage* und *Heuristic.*

Main liest die Input-Datei ein und startet die Suche.

Labyrinths repräsentiert beide Labyrinthe auf einmal und enthält Methoden, um die nächsten erreichbaren Felder zu generieren, die Start- / Endposition zu ermitteln oder die Labyrinthe anzuzeigen.

Labyrinth repräsentiert ein Labyrinth und enthält die gleichen Methoden wie *Labyrinths,* jedoch nur für ein Labyrinth.

LabyrinthSolver führt die Suche nach dem besten Weg im Superlabyrinth aus.

StateTracker speichert, welche Felder bereits besucht wurden und merkt sich woher man auf diese gekommen ist.

PositionData speichert eine Position und die Heuristik, sowie die Schrittzahl an dieser Stelle. Zusätzlich speichert sie, um den Weg später zu rekonstruieren, das Vorgänger-Feld und die Bewegung, die von dort gemacht wurde.

State speichert die vier Koordinaten einer Superposition.

StateMove speichert einen State, den dazugehörigen *Move* und die bisher gegangenen Schritte.

Move ist ein Enum, das die vier Züge enthält: *LEFT, RIGHT, UP, DOWN*.

Vector2 repräsentiert eine Position in einem der beiden Labyrinthe mit x- und y-Koordinate.

Field repräsentiert ein Feld eines Labyrinthes.

Heuristic ist ein Interface, das die Heuristiken der A*-Suche repräsentiert.

WeightedAverage ist die *Heuristic,* die ich benutze.

Details

Main-Klasse

Ich lese die Input-Datei ein und übergebe die Input-Zeilen an den *Labyrinths*-Konstruktor.

Dann initialisiere ich den *LabyrinthSolver* und die *Heuristic* und starte die Pfadsuche mit der *solveSimultaneously*-Methode der *LabyrinthSolver*-Klasse.

LabyrinthSolver-Klasse

SolveSimultaneously-Methode

Zuerst führe ich den Flood-Fill aus mit der *generateDists*-Methode der *Labyrinths*-Instanz.

Dann erstelle ich den *StateTracker* und die *PriorityQueue*, die die Open-List, also die noch nicht expandierten Knoten, darstellt und initialisiere diese.

Daraufhin kommt eine *while*-Schleife, die läuft, bis das Ziel gefunden wurde.

In dieser *polle* ich die erste *PositionData* und speichere den *State* dieser.

Wenn dieser noch nicht betrachtet wurde und nicht das Ziel ist, entferne ich ihn vom *StateTracker* mit der *removeFromMap*-Methode.

Die erreichbaren Nachbarn kriege ich durch die *getPossibleFields*-Methode der *Labyrinths*-Klasse und für jeden von jenen füge ich ihn der *Queue* hinzu, wenn nötig.

Es gibt eine Progress-Anzeige. Da die Heuristik nie überschätzt und immer näher an den eigentlichen Weg kommt, steigt sie kontinuierlich. Als Fortschritt gebe ich den linearen Fortschritt des aktuell geschätzten Weges von dem Best-Case (Maximum der Einzellösungen) zu einem schlechten Case (Beide Einzellösungen addiert) aus.
Dieser Progress erreicht meistens nicht 100%, aber er macht sichtbar, ob der Algorithmus schnell oder langsam Fortschritt macht und zeigt, wie groß der Umweg ist, den man gehen muss.

Nach der Schleife gebe ich den Weg zurück, den ich durch die *getPath*-Methode berechne.

GetPath-Methode

In dieser Methode folge ich den verketteten *previous*-Referenzen, um den Pfad zu rekonstruieren.

Labyrinths-Klasse

GetPossibleFields-Methode

Ich gehe alle Züge durch und kombiniere die Ergebnisse der Labyrinthe zu Superpositionen.

DrawSolution-Methode

Ich konvertiere den Weg in eine List von *Move*s und rufe die *drawSolution*-Methoden der Labyrinthe auf.

GenerateDists-Methode

In dieser Methode rufe ich die *generateDists*-Methode der Labyrinthe auf.

StateTracker-Klasse

Es gibt eine Map, die als keys die *States*, also Superpositionen, hat und als values die Vorgänger dieser, in Form von *PositionData*-Instanzen. Zusätzlich gibt es ein *MyBitSet*, das speichert, welche *States* bereits besucht wurden.

Get-Methode

Es wird zu einem *State* der Vorgänger aus der Map zurückgegeben.

HasSeen-Methode

Der Wert aus dem BitSet wird zurückgegeben. Der Index wird mit *getIndex* ausgerechnet.

Put-Methode

Der Vorgänger wird in der Map gespeichert. Im BitSet wird gespeichert, dass der *State* bereits betrachtet wurde.

GetIndex

Der Index wird ausgerechnet, indem die Koordinaten jeweils auf den Wert addiert werden, welcher dann um 8 Bits geshiftet wird.

Dadurch ist die Länge und Breite der Labyrinthe auf 256 begrenzt.

MyBitSet-Klasse

Diese Klasse enthält ein *long[]*, in dem jedes Bit einen Boolean repräsentiert, der angibt, ob ein State bereits entdeckt wurde.

Dies war als Beschleunigung des Programms notwendig und spart gleichzeitig RAM und Rechenzeit, da man sonst ein HashSet benutzen müsste.

Labyrinth-Klasse

Diese Klasse speichert Breite und Höhe des Labyrinthes als *int*. Das Labyrinth an sich wird als zweidimensionales *Field*-Array dargestellt. Zusätzlich werden die Koordinaten des Start- und Zielfeldes als *Vector2* gespeichert.

Zudem gibt es ein zweidimensionales *int*-Array, welches die Entfernungen zum Ziel enthält.

GetPossibleFields-Methode

In einem *if*-Tree werden die möglichen Positionen ermittelt.

GetField-Methode

In einem *switch-case*, welches einen *if-else*-Tree enthält, wird das Feld ermittelt.

GenerateDists-Methode

Diese Methode benutzt Flood-Fill, um die Entfernung von jedem Feld zum Ziel zu bestimmen.

Es gibt eine *Queue*, die die neuen Positionen sortiert und ein *Set*, welches speichert, welche Positionen bereits gefunden wurde.

Ich füge der *Queue* und dem *Set* den Start hinzu und generiere die Nachbarn mit der *getPossibleFields*-Methode, um diese, wenn sie noch nicht im *Set* enthalten sind, diesem und der *Map* hinzuzufügen. Dabei zähle ich die Schritte und speichere sie in einem zweidimensionalen *int*-Array.

MyFrame-Klasse

Diese Klasse wird benutzt, um die Labyrinthe grafisch darzustellen. Dafür speichert sie das *Labyrinth*, die Größe eines Feldes in Pixeln und den Weg als Liste von *Move*s.

Paint-Methode

Das Labyrinth wird skaliert gezeichnet. Wenn der Weg gefunden wurde, wird dieser auch eingezeichnet. Für jede Bewegung, bei der man in eine Wand läuft, wird ein Punkt eingezeichnet.

Heuristic-Interface

Dieses Interface repräsentiert die Heuristiken für das Superlabyrinth.

GetScore-Methode

Gibt den geschätzten Weg zum Ziel zurück.

GetName-Methode

Gibt den Namen der Heuristik zurück.

WeightedAverage-Klasse

Diese Klasse implementiert *Heuristic*. Sie speichert die Gewichte.

GetScore-Methode

Die Einzelweglängen der Labyrinthe werden mithilfe von *getDist* ausgelesen. Die längere wird mit dem größeren Gewicht multipliziert und mit dem kürzeren, multipliziert mit dem kleineren Gewicht, addiert: $h(l_1, l_2) = max(l_1, l_2) \cdot 0.99999 + \min(l_1, l_2) \cdot 0.00001$

Genereller Hinweis

Beim Ausführen von großen Beispielen wird viel RAM benötigt.
Ich empfehle 10GB Heap (-Xmx10g).

Beispiele

Labyrinthe 0

Lab1 Best way: 8

Lab2 Best way: 6

Queue: bestWayLen = 1 + 6.99998 Progress: 0.0% // queueLen = 1 (1) // mapSize = 1

Time needed: 1.1936597s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 8

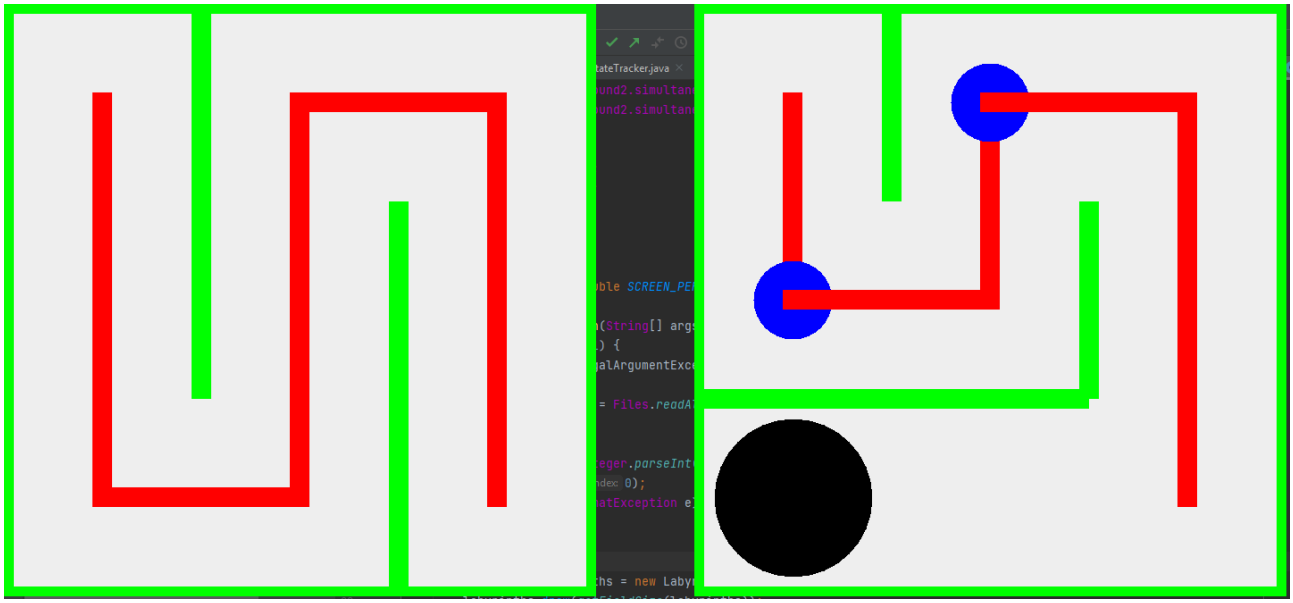


Schaubild 1: Grün = Wand; Rot = Weg; Blau = Gegen Wand gelaufen; Schwarz = Loch

Labyrinthe 1

Lab1 Best way: 19

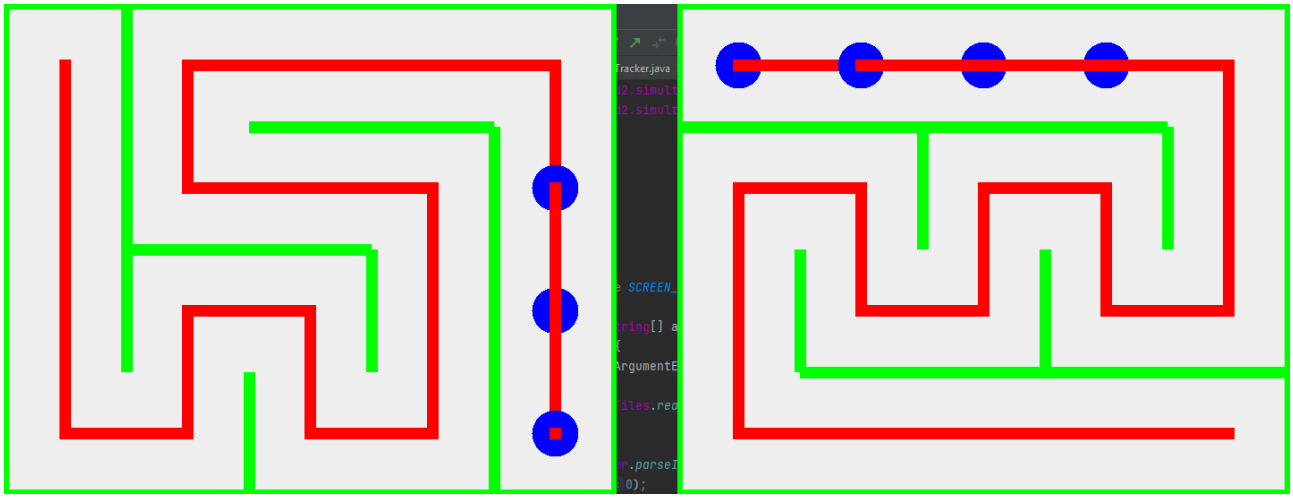
Lab2 Best way: 19

Queue: bestWayLen = 1 + 18.99999 Progress: 5.26% // queueLen = 2 (1) // mapSize = 2

Time needed: 0.76035535s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 31



Labyrinthe 2

Lab1 Best way: 48

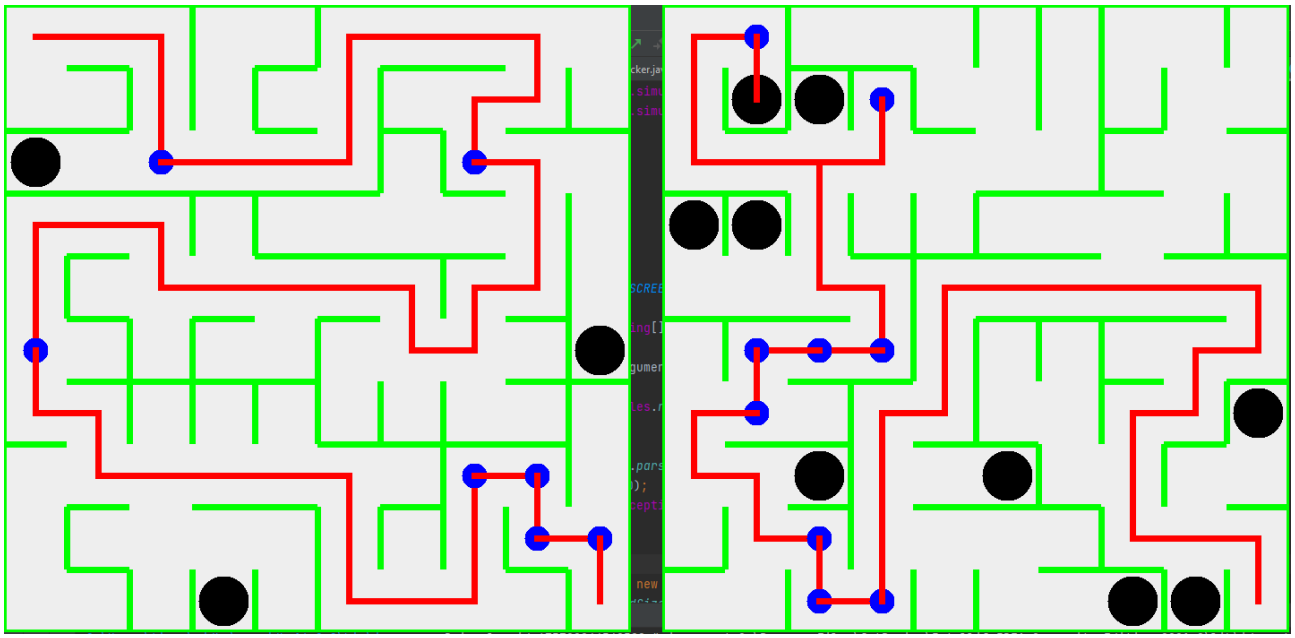
Lab2 Best way: 38

Queue: bestWayLen = 1 + 46.999919999999996 Progress: 0.0% // queueLen = 2 (1) // mapSize = 2

Time needed: 0.7126518s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 65



Labyrinthe 3

Lab1 Best way: 106

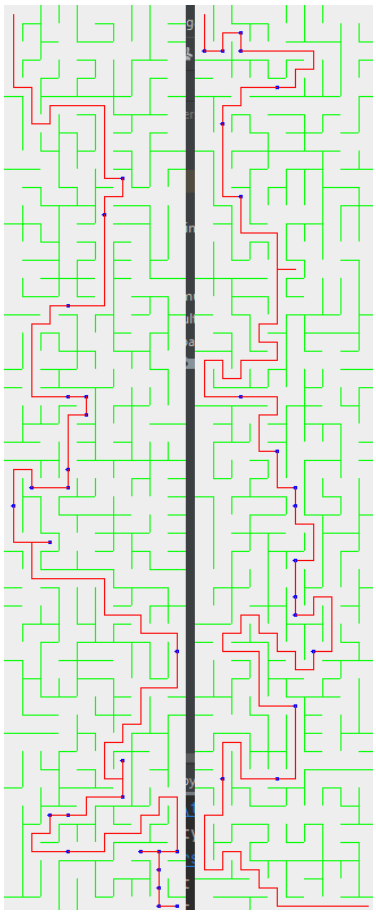
Lab2 Best way: 122

Queue: bestWayLen = 1 + 120.99983999999999 Progress: 0.0% // queueLen = 2 (1) // mapSize = 2

Time needed: 0.810792s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 164



Labyrinthe 4

Lab1 Best way: 10200

Lab2 Best way: 10200

Queue: bestWayLen = 1 + 10199.0 Progress: 0.0% // queueLen = 1 (1) // mapSize = 1

Queue: bestWayLen = 1792 + 9686.98797 Progress: 12.53% // queueLen = 6593 (4194305) // mapSize = 6053

Queue: bestWayLen = 5787 + 6215.997879999999 Progress: 17.67% // queueLen = 9298 (8388609) // mapSize = 8556

Queue: bestWayLen = 6556 + 5851.9943 Progress: 21.64% // queueLen = 11524 (12582913) // mapSize = 10559

Queue: bestWayLen = 3060 + 9686.97494 Progress: 24.97% // queueLen = 13588 (16777217) // mapSize = 12351

Queue: bestWayLen = 9563 + 3479.9985699999997 Progress: 27.87% // queueLen = 15160 (20971521) // mapSize = 13880

Queue: bestWayLen = 8096 + 5217.98551 Progress: 30.52% // queueLen = 16518 (25165825) // mapSize = 15199

Queue: bestWayLen = 8067 + 5495.98098 Progress: 32.97% // queueLen = 17628 (29360129) // mapSize = 16217

Queue: bestWayLen = 11542 + 2250.99496 Progress: 35.22% // queueLen = 18800 (33554433) // mapSize = 17384

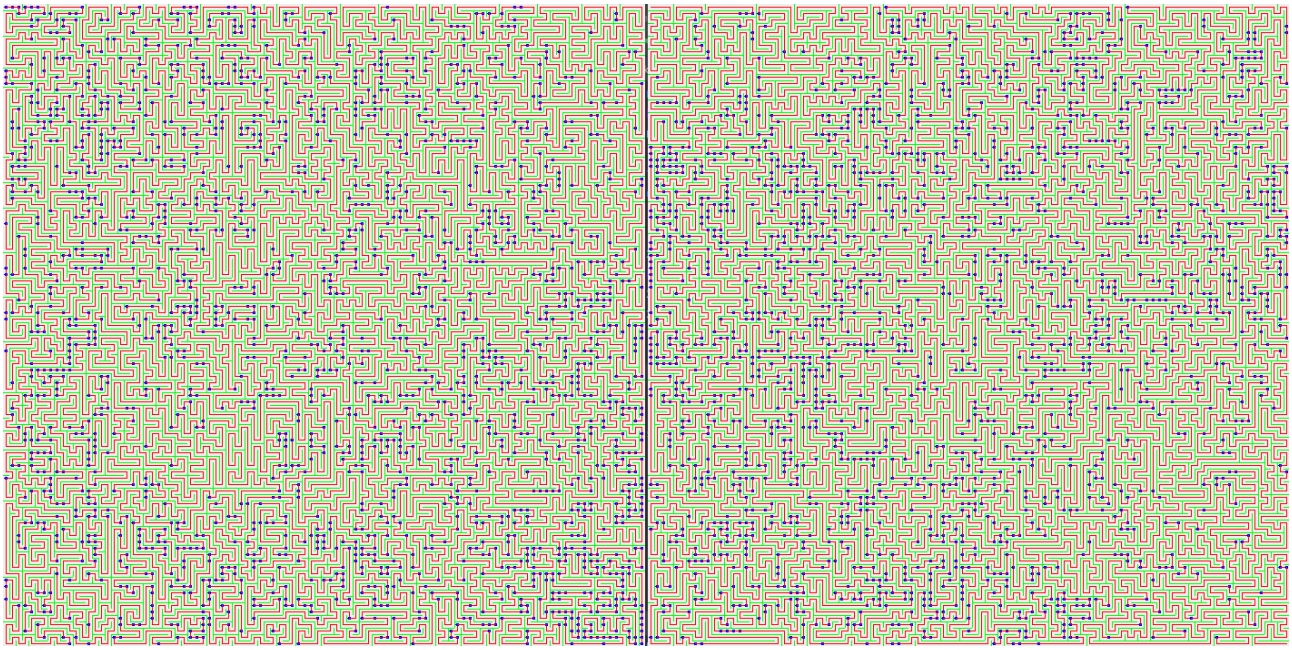
Queue: bestWayLen = 7908 + 6101.97294 Progress: 37.35% // queueLen = 20133 (37748737) // mapSize = 18491

Queue: bestWayLen = 12549 + 1665.99244 Progress: 39.36% // queueLen = 20763 (41943041) // mapSize = 19183

Time needed: 31.080935s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 14384



Labyrinthe 5

Lab1 Best way: 938

Lab2 Best way: 808

Queue: bestWayLen = 1 + 936.9987199999999 Progress: 0.0% // queueLen = 2 (1) // mapSize = 2

Queue: bestWayLen = 353 + 697.9989499999999 Progress: 13.98% // queueLen = 162852 (4194305) // mapSize = 126294

Queue: bestWayLen = 367 + 720.9995800000002 Progress: 18.56% // queueLen = 254987 (8388609) // mapSize = 192353

Queue: bestWayLen = 328 + 785.99845 Progress: 21.78% // queueLen = 348418 (12582913) // mapSize = 262382

Queue: bestWayLen = 218 + 915.99783 Progress: 24.25% // queueLen = 425450 (16777217) // mapSize = 318037

Queue: bestWayLen = 219 + 931.99768 Progress: 26.36% // queueLen = 484446 (20971521) // mapSize = 358711

Queue: bestWayLen = 279 + 886.99876 Progress: 28.21% // queueLen = 526204 (25165825) // mapSize = 385078

Queue: bestWayLen = 328 + 851.99937 Progress: 29.95% // queueLen = 576049 (29360129) // mapSize = 421603

Queue: bestWayLen = 649 + 543.9995100000001 Progress: 31.55% // queueLen = 625055 (33554433) // mapSize = 458676

Queue: bestWayLen = 425 + 779.99962 Progress: 33.04% // queueLen = 666039 (37748737) // mapSize = 488260

Queue: bestWayLen = 315 + 901.99692 Progress: 34.52% // queueLen = 704972 (41943041) // mapSize = 516676

Queue: bestWayLen = 307 + 919.9993300000001 Progress: 35.76% // queueLen = 746720 (46137345) // mapSize = 547603

Queue: bestWayLen = 490 + 746.99953 Progress: 37.0% // queueLen = 785136 (50331649) // mapSize = 575541

Queue: bestWayLen = 316 + 930.9982399999999 Progress: 38.24% // queueLen = 827370 (54525953) // mapSize = 608302

Queue: bestWayLen = 575 + 680.99848 Progress: 39.35% // queueLen = 864209 (58720257) // mapSize = 635372

Queue: bestWayLen = 396 + 868.99739 Progress: 40.46% // queueLen = 904329 (62914561) // mapSize = 665102

Queue: bestWayLen = 566 + 706.99863 Progress: 41.46% // queueLen = 939654 (67108865) // mapSize = 689672

Queue: bestWayLen = 552 + 728.99894 Progress: 42.45% // queueLen = 970952 (71303169) // mapSize = 709670

Queue: bestWayLen = 297 + 991.9982300000001 Progress: 43.44% // queueLen = 1007701 (75497473) // mapSize = 734715

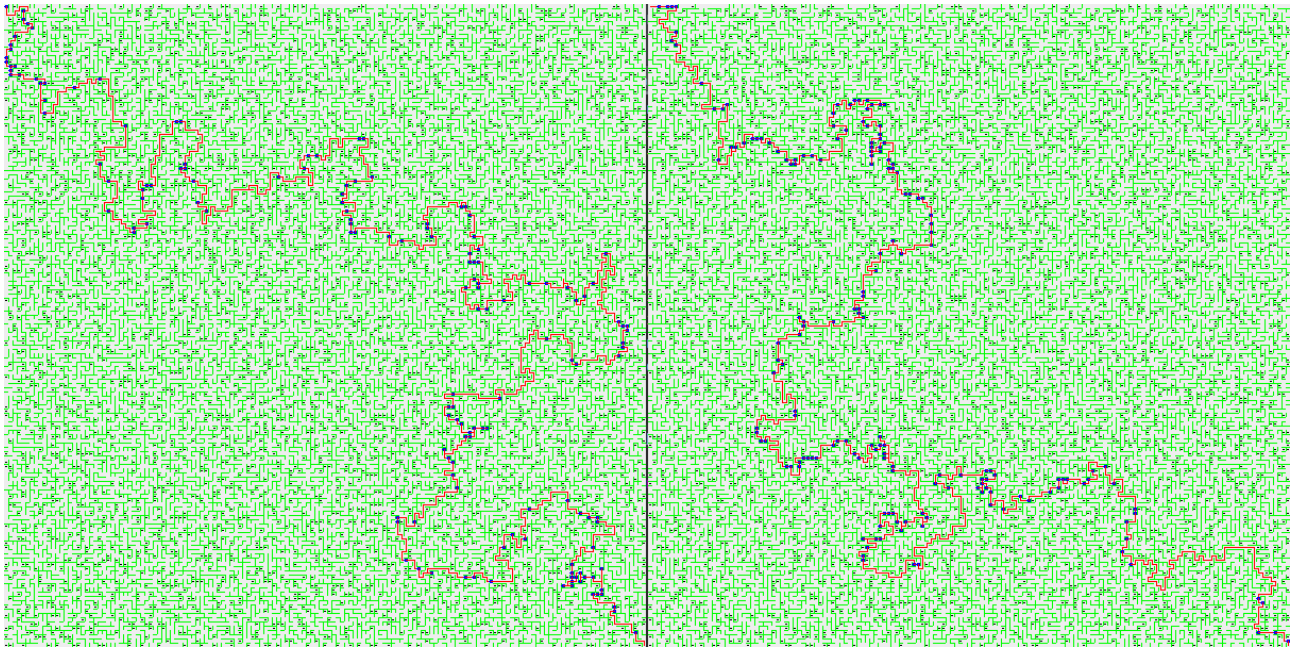
Queue: bestWayLen = 1080 + 215.99988000000008 Progress: 44.3% // queueLen = 1053314 (79691777) // mapSize = 768580

Queue: bestWayLen = 686 + 617.99678 Progress: 45.29% // queueLen = 1100305 (83886081) // mapSize = 803818

Time needed: 126.44973s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 1308



Labyrinthe 6

Lab1 Best way: 776

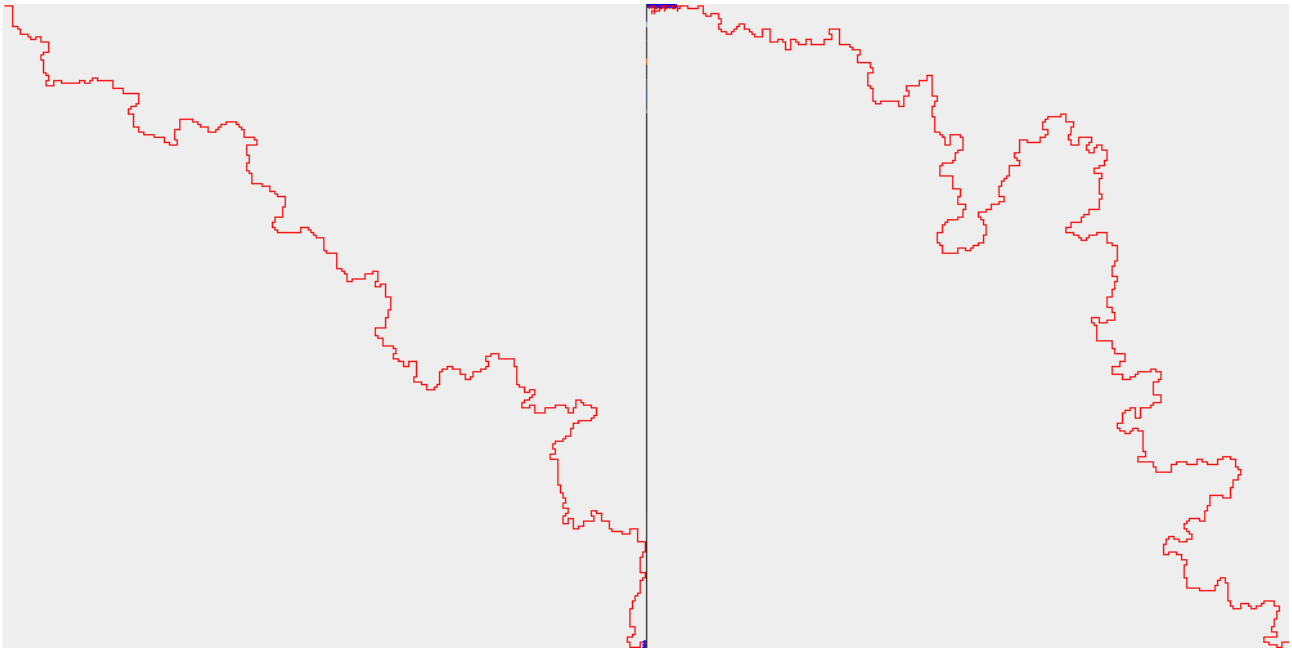
Lab2 Best way: 1076

Queue: bestWayLen = 1 + 1074.997 Progress: 0.0% // queueLen = 2 (1) // mapSize = 2

Time needed: 1.0596135s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 1844



Labyrinthe 7

Lab1 Best way: 74

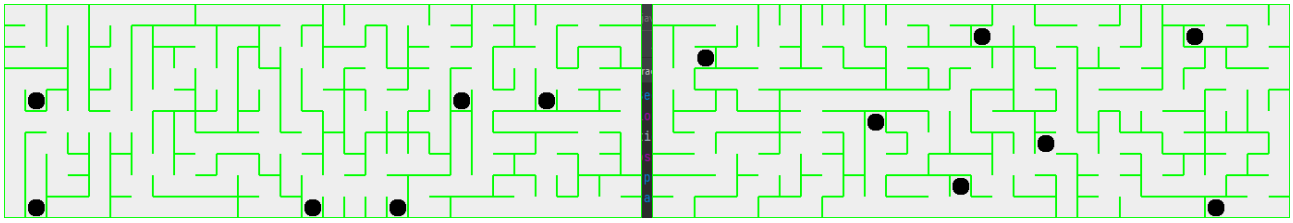
Lab2 Best way: 0

There is no solution

Time needed: 0.0023126s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: -1



Labyrinthe 8

Lab1 Best way: 404

Lab2 Best way: 330

Queue: bestWayLen = 1 + 402.99926 Progress: 0.0% // queueLen = 1 (1) // mapSize = 1

Queue: bestWayLen = 198 + 235.99971000000005 Progress: 9.09% // queueLen = 562638 (4194305) // mapSize = 530383

Queue: bestWayLen = 134 + 307.99960999999996 Progress: 11.51% // queueLen = 876115 (8388609) // mapSize = 826290

Queue: bestWayLen = 231 + 215.99995 Progress: 13.03% // queueLen = 1150992 (12582913) // mapSize = 1084433

Queue: bestWayLen = 178 + 273.99962000000005 Progress: 14.54% // queueLen = 1405158 (16777217) // mapSize = 1324063

Queue: bestWayLen = 165 + 290.99917 Progress: 15.75% // queueLen = 1615804 (20971521) // mapSize = 1524241

Queue: bestWayLen = 220 + 238.99966 Progress: 16.66% // queueLen = 1806349 (25165825) // mapSize = 1704766

Queue: bestWayLen = 159 + 302.99965000000003 Progress: 17.57% // queueLen = 1981830 (29360129) // mapSize = 1870750

Queue: bestWayLen = 290 + 174.99941 Progress: 18.48% // queueLen = 2120897 (33554433) // mapSize = 2003248

Queue: bestWayLen = 308 + 158.99987 Progress: 19.09% // queueLen = 2254671 (37748737) // mapSize = 2129780

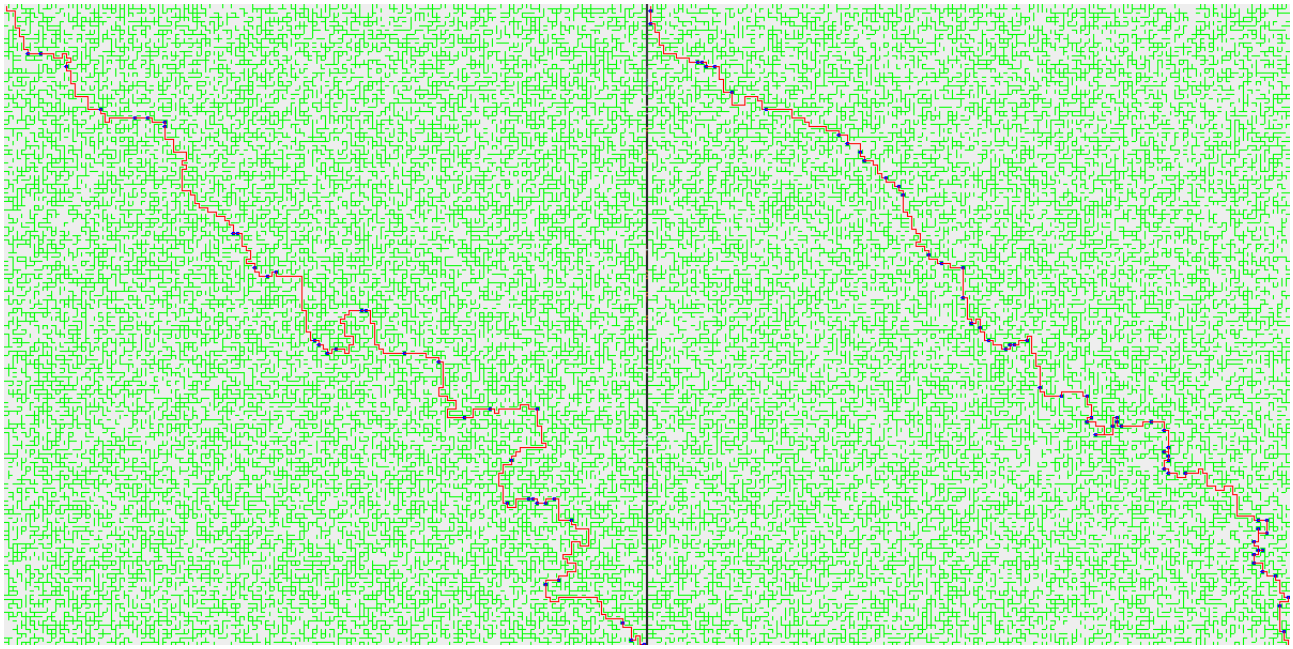
Queue: bestWayLen = 212 + 257.99946 Progress: 19.99% // queueLen = 2390213 (41943041) // mapSize = 2257531

Queue: bestWayLen = 285 + 186.99971 Progress: 20.6% // queueLen = 2517352 (46137345) // mapSize = 2377268

Time needed: 92.9144s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 472



Labyrinthe 9

Lab1 Best way: 588

Lab2 Best way: 812

Queue: bestWayLen = 1 + 810.99776 Progress: 0.0% // queueLen = 1 (1) // mapSize = 1

Queue: bestWayLen = 102 + 772.99729 Progress: 10.71% // queueLen = 152528 (4194305) // mapSize = 147968

Queue: bestWayLen = 142 + 762.99838 Progress: 15.81% // queueLen = 226025 (8388609) // mapSize = 219534

Queue: bestWayLen = 646 + 280.99983999999995 Progress: 19.55% // queueLen = 293932 (12582913) // mapSize = 285516

Queue: bestWayLen = 443 + 501.9987 Progress: 22.61% // queueLen = 376296 (16777217) // mapSize = 365858

Queue: bestWayLen = 409 + 549.99816 Progress: 24.99% // queueLen = 458942 (20971521) // mapSize = 446280

Queue: bestWayLen = 523 + 446.9996 Progress: 26.87% // queueLen = 543000 (25165825) // mapSize = 528048

Queue: bestWayLen = 482 + 497.99965 Progress: 28.57% // queueLen = 616783 (29360129) // mapSize = 599899

Queue: bestWayLen = 358 + 630.99949 Progress: 30.1% // queueLen = 689823 (33554433) // mapSize = 671016

Queue: bestWayLen = 329 + 667.9995700000001 Progress: 31.46% // queueLen = 753068 (37748737) // mapSize = 732819

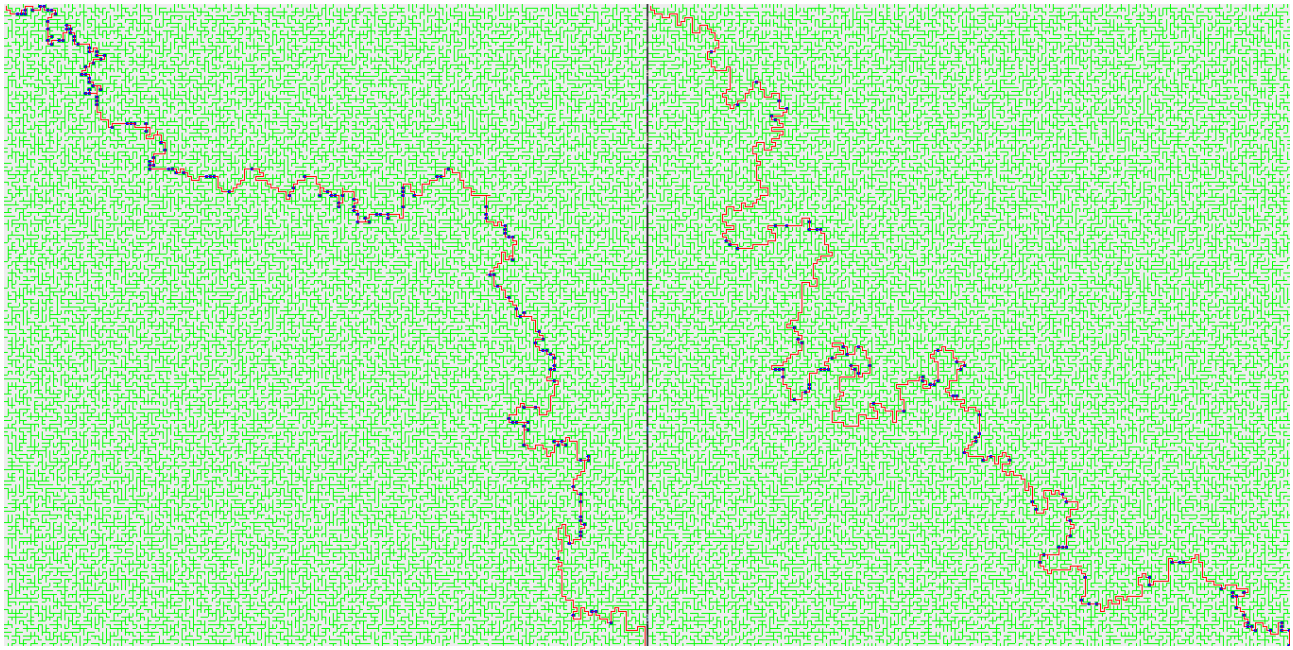
Queue: bestWayLen = 282 + 722.9981300000001 Progress: 32.82% // queueLen = 805080 (41943041) // mapSize = 783714

Queue: bestWayLen = 490 + 521.99807 Progress: 34.01% // queueLen = 853757 (46137345) // mapSize = 830927

Time needed: 63.63466s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 1012



Quellcode

```
public class LabyrinthSolver {

    public List<PositionData> solveSimultaneously(final Labyrinths labyrinths, Heuristic heuristic) {
        labyrinths.generateDists();
        StateTracker tracker = new StateTracker();
        Queue<PositionData> toCheck = new PriorityQueue<>(Comparator.comparingDouble(PositionData::getScore));

        final State start = labyrinths.getStartPos();
        final PositionData positionData = new PositionData(start, heuristic.getScore(start, labyrinths),
                                                           0, null, Move.DOWN);
        tracker.put(start, positionData);
        toCheck.add(positionData);
        State finish = labyrinths.getFinishPos();
        PositionData finishData = null;
        boolean finishFound = false;
        while (!finishFound) {
            final PositionData curr = toCheck.poll();
            final State state = curr.getState();
            if (tracker.isExpanded(state)) {
                continue;
            }
            if (state.equalsIgnoreJumpCount(finish)) {
                finishFound = true;
                finishData = curr;
                continue;
            }
            tracker.removeFromMap(state);
            StateMove[] possibleNextFields = labyrinths.getPossibleFields(state);
            final int stepCount = curr.getStepCount() + 1;
            for (StateMove next : possibleNextFields) {
                if (tracker.hasSeen(next.state())) {
```

```
                final PositionData oldPositionData = tracker.get(next.state());
                if (oldPositionData != null && oldPositionData.getStepCount() > stepCount) {
                    final PositionData nextScore = new PositionData(next.state(),
                            heuristic.getScore(next.state(), labyrinths) + stepCount, stepCount, curr, next.move());
                    tracker.put(next.state(), nextScore);
                    toCheck.add(nextScore);
                }
            } else {
                final State nextState = next.state();
                final PositionData nextScore = new PositionData(nextState,
                        heuristic.getScore(nextState, labyrinths) + stepCount, stepCount, curr, next.move());
                tracker.put(nextState, nextScore);
                toCheck.add(nextScore);
            }
        }
    }

    return getPath(finishData);
  }
}
```

Erweiterungen

Verschiedene Größen

Idee

Die Labyrinthe sollen unterschiedlich groß sein können.

Dafür muss man am Algorithmus eigentlich nichts ändern, aber am Einlesen. Deshalb habe ich den Code getrennt.

Beispiele

Labyrinth 0-7

Lab1 Best way: 8

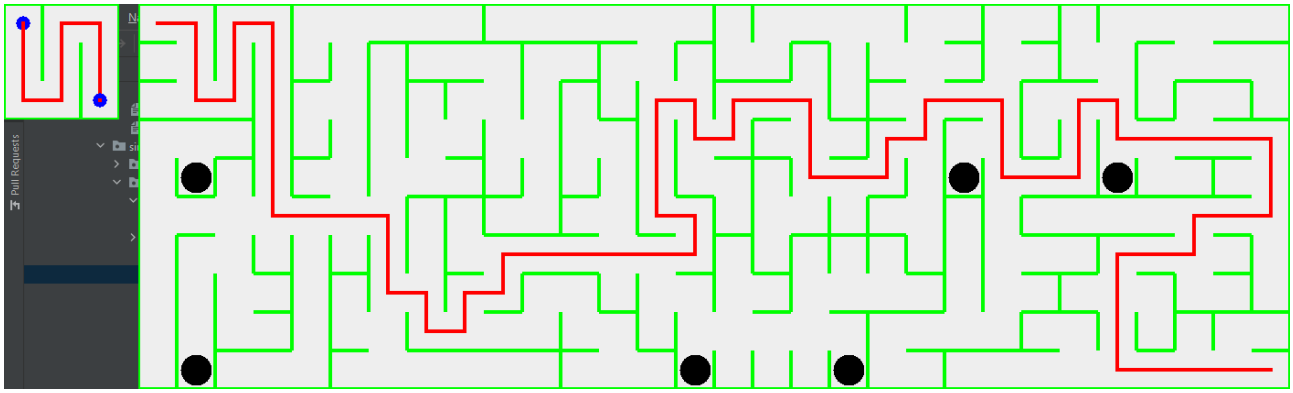
Lab2 Best way: 74

Queue: bestWayLen = 1 + 72.935 Progress: -0.81% // queueLen = 2 (1) // mapSize = 2

Time needed: 0.091873296s

Using heuristic: WeightedAverage: 0.999-0.001

Length: 74



Labyrinth 3-6

Lab1 Best way: 106

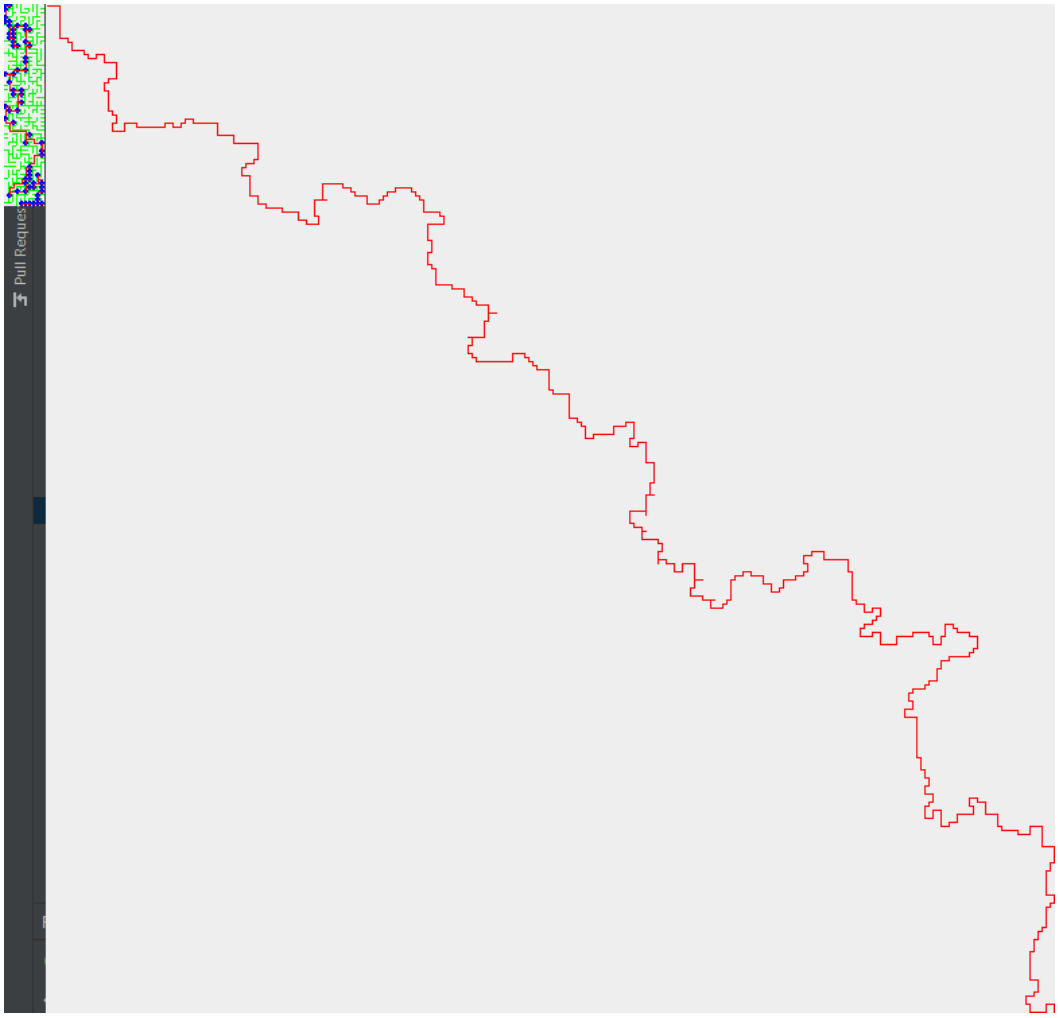
Lab2 Best way: 776

Queue: bestWayLen = 1 + 774.332 Progress: -0.63% // queueLen = 2 (1) // mapSize = 2

Time needed: 0.5237112s

Using heuristic: WeightedAverage: 0.999-0.001

Length: 798



Labyrinth 7-5

Lab1 Best way: 74

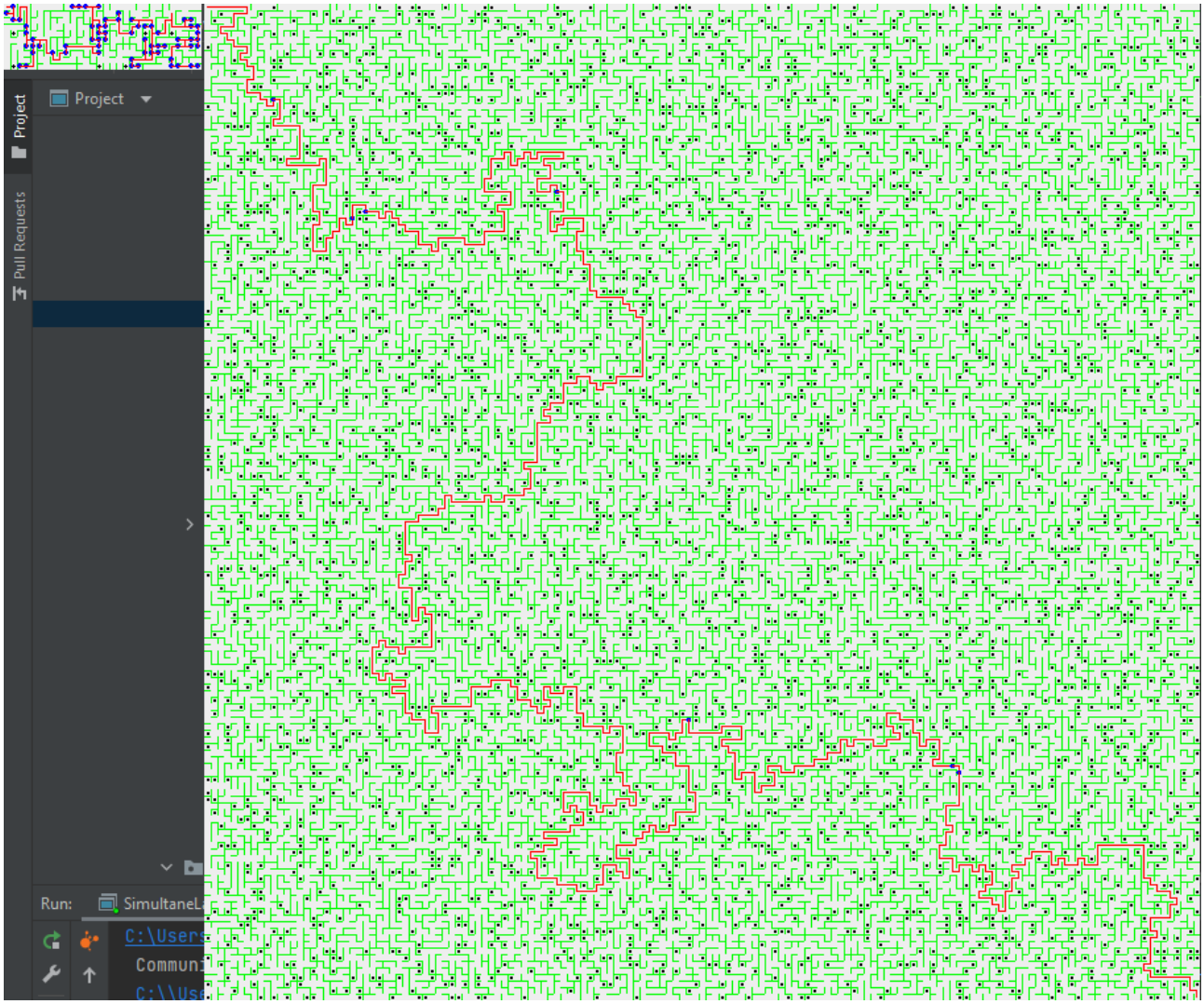
Lab2 Best way: 808

Queue: bestWayLen = 1 + 806.266 Progress: -0.99% // queueLen = 2 (1) // mapSize = 2

Time needed: 0.3193529s

Using heuristic: WeightedAverage: 0.999-0.001

Length: 815



Sprünge

Idee

Problem

Es sollen vier weitere Bewegungen eingeführt werden, die den Leuten in den Labyrinthen erlauben über Wände zu springen. Sie müssen dabei beide in die gleiche Richtung, aber nicht zwingend über eine Wand springen.

Algorithmus

Man fügt eine 5. Dimension hinzu, die angibt wieviele Sprünge noch übrig sind und fügt in der Zuggenerierung die neuen Bewegungen hinzu. Dann kann man wieder A* mit der gleichen Heuristik verwenden.

Umsetzung

Da keine großen Änderungen nötig war, habe ich den Code im normalen Code integriert.

Bei der Eingabedatei muss die erste Zeile nur aus der Sprungzahl bestehen und diese darf maximal *3* betragen.

Folgende Klassen wurden geändert:

Main-Klasse
Die *Main*-Klasse liest die Sprungzahl ein.

LabyrinthSolver-Klasse
Gibt die Sprungzahl aus, falls sie größer als *0* ist.

Vector3-Klasse
Ergänzt *Vector2* und speichert zusätzlich die übrige Sprungzahl.

Labyrinth-Klasse
Es wurden zusätzliche *Vector3* für *start* und *finish* eingeführt.

Das *int[][]*, welches die Entfernungen zum Ziel speichert ist jetzt dreidimensional, um die Sprungzahl mit einzubeziehen.

Beinahe alle *Vector2*-Usages wurden durch *Vector3* ersetzt.

Die neuen *Move*s wurden in *getField* und *getPossibleFields* beachtet.

Move-Klasse
Die 4 neuen *Move*s wurden hinzugefügt: LEFT_JUMP, RIGHT_JUMP, UP_JUMP, DOWN_JUMP

StateTracker-Klasse
Am Ende der *getIndex*-Methode wird jetzt nochmal um *2* Bits geshiftet und die aktuelle Sprungzahl addiert. Dadurch sind die Sprünge auf maximal 3 limitiert. Das kommt daher, dass die obere Grenze für die Länge von Arrays in Java zwischen 2^{30} und 2^{31}. Deshalb brauchen wir noch 6 Bits von dem *long*-Datentyp, wodurch wir 2^{35} *Booleans* speichern können.

State-Klasse
Es wird jetzt auch noch die verbliebene Sprungzahl gespeichert.

Genereller Hinweis
Beim Ausführen von großen Beispielen wird sehr viel RAM benötigt.
Ich empfehle 20GB Heap (-Xmx20g).

Quellcode

```
public class Labyrinth {

public void generateDists() {
    Queue<Vector3> queue = new ArrayDeque<>();
    Set<Vector3> found = new HashSet<>();

    final Vector3 finishPos = getFinishPos();
```

```
    queue.add(finishPos);
    found.add(finishPos);
    while (queue.size() > 0) {
        Vector3 curr = queue.poll();
        int dist = dists[curr.x][curr.y][curr.z] + 1;
        if (fields[curr.x][curr.y].isHole) {
            dist--;
        }
        List<Vector3> neighbours = getPossibleFields(curr);
        if (curr.equalsIgnoreZ(start)) {
            for (int x = 0; x < fields.length; x++) {
                final Field[] row = fields[x];
                for (int y = 0; y < row.length; y++) {
                    if (row[y].isHole) {
                        neighbours.add(new Vector3(x, y, curr.z));
                    }
                }
            }
        }
        for (Vector3 neighbour : neighbours) {
            if (!found.contains(neighbour)) {
                if (!fields[neighbour.x][neighbour.y].isHole) {
                    found.add(neighbour);
                    queue.add(neighbour);
                    dists[neighbour.x][neighbour.y][neighbour.z] = dist;
                }
            }
        }
    }
    for (final int[][] row : dists) {
        for (final int[] column : row) {
            for (int z = 0; z < column.length - 1; z++) {
                final int value = column[z + 1];
                if ((column[z] < value || value == 0) && column[z] != 0) {
                    column[z + 1] = column[z];
                }
            }
        }
    }
}

public List<Vector3> getPossibleFields(final Vector3 curr) {
    List<Vector3> result = new ArrayList<>(8);
    final int y = curr.y;
    final int x = curr.x;
    final int z = curr.z;
    if (y > 0) {
        if (!fields[x][y - 1].hasLowerWall) {
            result.add(new Vector3(x, y - 1, z));
        }
        if (z < startJumpCount) {
            result.add(new Vector3(x, y - 1, z + 1));
        }
    }
    if (y < height - 1) {
        if (!fields[x][y].hasLowerWall) {
            result.add(new Vector3(x, y + 1, z));
        }
```

```
        if (z < startJumpCount) {
            result.add(new Vector3(x, y + 1, z + 1));
        }
    }
    if (x > 0) {
        if (!fields[x - 1][y].hasRightWall) {
            result.add(new Vector3(x - 1, y, z));
        }
        if (z < startJumpCount) {
            result.add(new Vector3(x - 1, y, z + 1));
        }
    }
    if (x < width - 1) {
        if (!fields[x][y].hasRightWall) {
            result.add(new Vector3(x + 1, y, z));
        }
        if (z < startJumpCount) {
            result.add(new Vector3(x + 1, y, z + 1));
        }
    }
    return result;
}

}
```

Beispiele

Jump-Labyrinthe 0

Lab1 Best way: 4

Lab2 Best way: 4

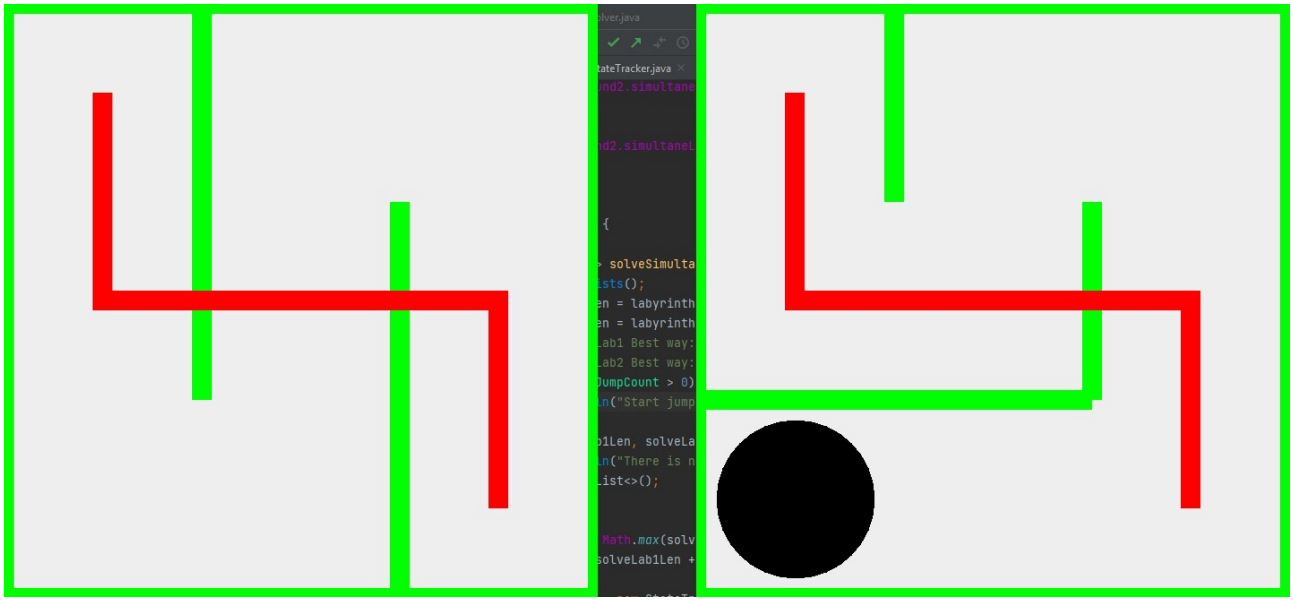
Start jump count: 3

Queue: bestWayLen = 1 + 3.0 Progress: 0.0% // queueLen = 4 (1) // mapSize = 4

Time needed: 0.6040514s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 4



Jump-Labyrinthe 1

Lab1 Best way: 7

Lab2 Best way: 7

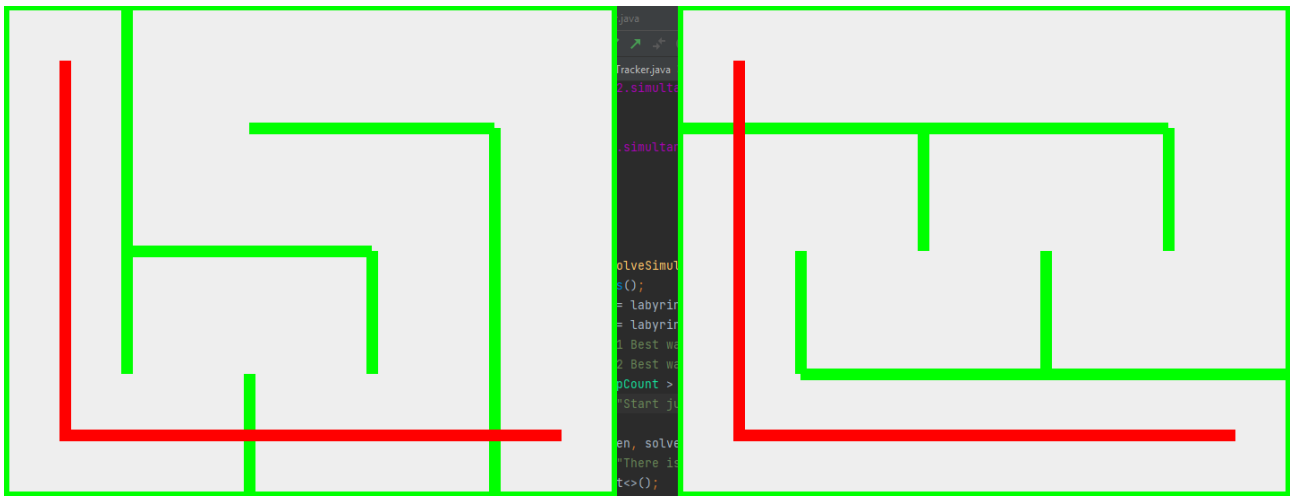
Start jump count: 3

Queue: bestWayLen = 1 + 6.0 Progress: 0.0% // queueLen = 5 (1) // mapSize = 5

Time needed: 0.6033564s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 7



Jump-Labyrinthe 2

Lab1 Best way: 18

Lab2 Best way: 18

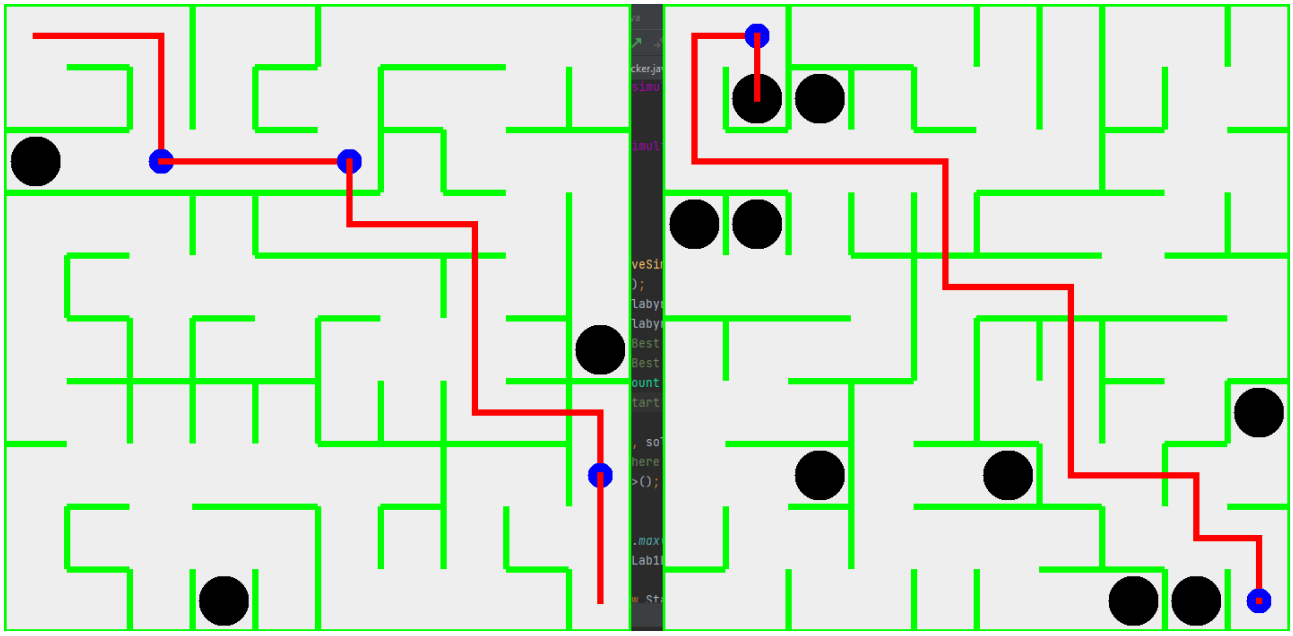
Start jump count: 3

Queue: bestWayLen = 1 + 17.0 Progress: 0.0% // queueLen = 5 (1) // mapSize = 5

Time needed: 0.616566s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 22



Jump-Labyrinthe 3

Lab1 Best way: 72

Lab2 Best way: 86

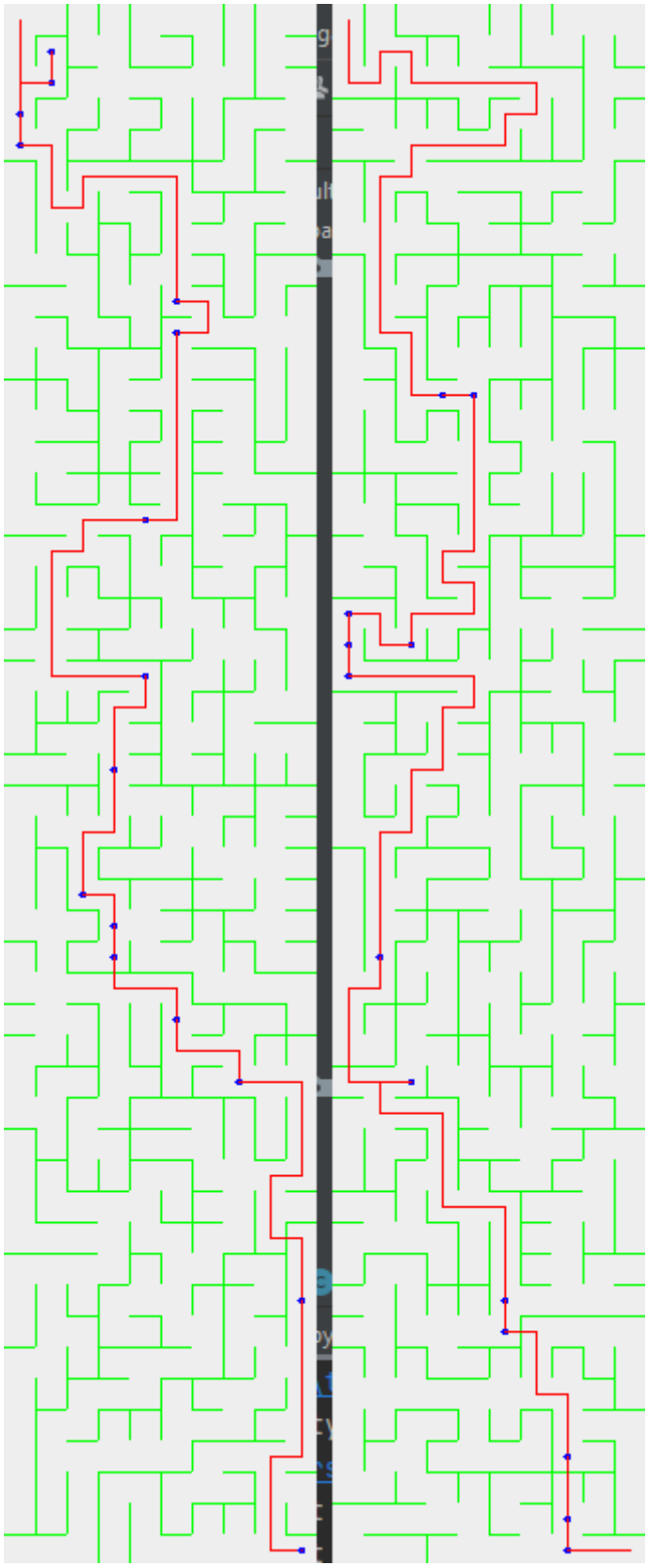
Start jump count: 3

Queue: bestWayLen = 1 + 84.99986 Progress: 0.0% // queueLen = 5 (1) // mapSize = 5

Time needed: 0.98545635s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 113



Jump-Labyrinthe 4

Lab1 Best way: 904

Lab2 Best way: 1430

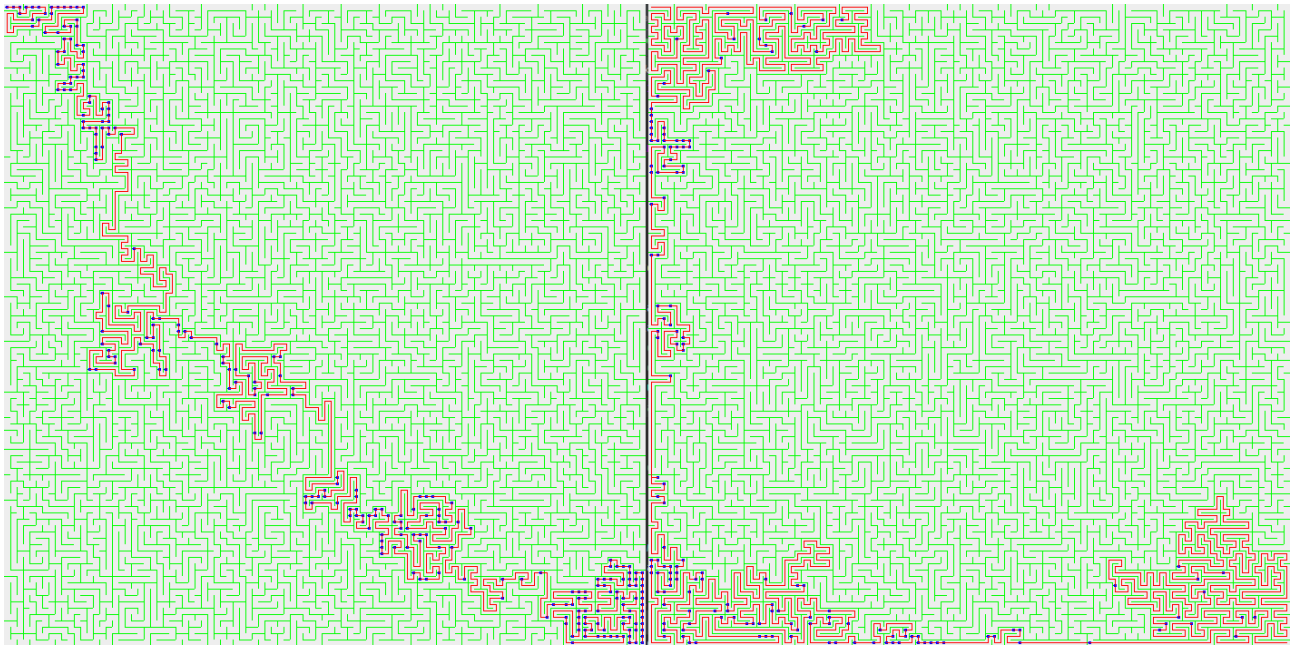
Start jump count: 3

Queue: bestWayLen = 1 + 1428.99474 Progress: 0.0% // queueLen = 4 (1) // mapSize = 4

Time needed: 11.118489s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 1834



Jump-Labyrinthe 5

Lab1 Best way: 484

Lab2 Best way: 424

Start jump count: 3

Queue: bestWayLen = 1 + 482.99942 Progress: 0.0% // queueLen = 5 (1) // mapSize = 5

Queue: bestWayLen = 252 + 342.99999 Progress: 26.17% // queueLen = 5791795 (4194305) // mapSize = 4147277

Queue: bestWayLen = 169 + 447.99960999999996 Progress: 31.36% // queueLen = 10994440 (8388609) // mapSize = 7954811

Queue: bestWayLen = 172 + 457.99905 Progress: 34.43% // queueLen = 16254471 (12582913) // mapSize = 11824340

Queue: bestWayLen = 227 + 411.99897 Progress: 36.55% // queueLen = 21447693 (16777217) // mapSize = 15550530

Queue: bestWayLen = 338 + 307.99924 Progress: 38.2% // queueLen = 26483684 (20971521) // mapSize = 19102854

Queue: bestWayLen = 242 + 409.99929 Progress: 39.62% // queueLen = 31396662 (25165825) // mapSize = 22558449

Queue: bestWayLen = 230 + 426.99962000000005 Progress: 40.8% // queueLen = 36133139 (29360129) // mapSize = 25824590

Queue: bestWayLen = 314 + 347.99920999999995 Progress: 41.98% // queueLen = 40732370 (33554433) // mapSize = 28932099

Queue: bestWayLen = 286 + 379.99963 Progress: 42.92% // queueLen = 45154821 (37748737) // mapSize = 31871302

Queue: bestWayLen = 276 + 393.99963 Progress: 43.86% // queueLen = 49486679 (41943041) // mapSize = 34721678

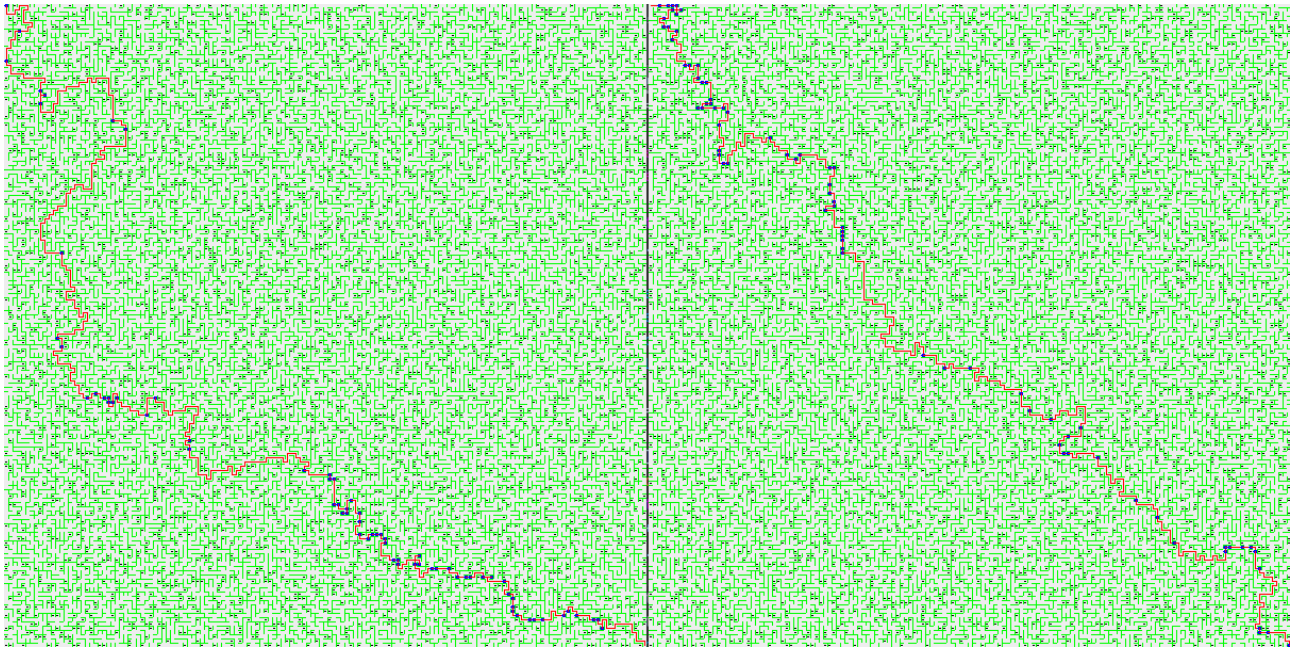
Queue: bestWayLen = 106 + 567.99931 Progress: 44.81% // queueLen = 53759832 (46137345) // mapSize = 37535405

Queue: bestWayLen = 310 + 366.99996 Progress: 45.51% // queueLen = 57894956 (50331649) // mapSize = 40254592

Time needed: 410.05057s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 680



Jump-Labyrinthe 6

Lab1 Best way: 776

Lab2 Best way: 1076

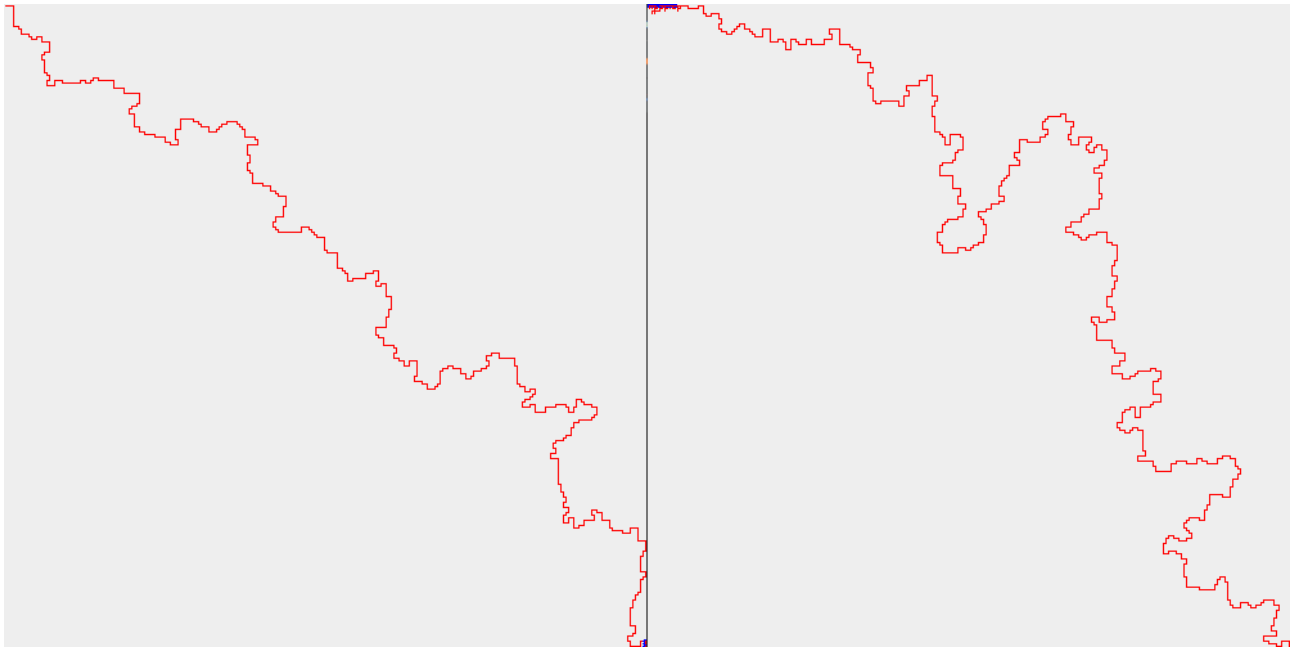
Start jump count: 3

Queue: bestWayLen = 1 + 1074.997 Progress: 0.0% // queueLen = 5 (1) // mapSize = 5

Time needed: 2.3727508s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 1844



Jump-Labyrinthe 7

Lab1 Best way: 62

Lab2 Best way: 68

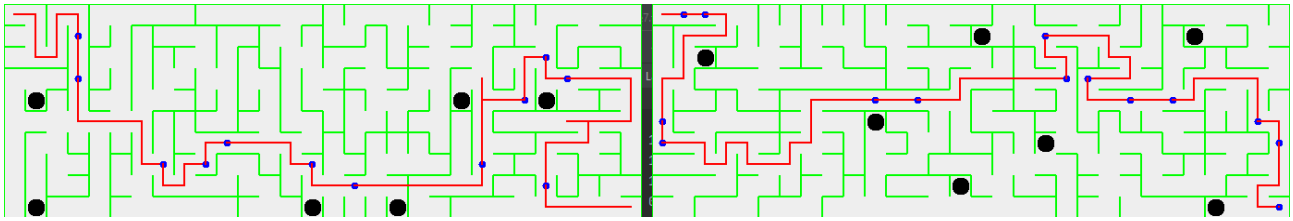
Start jump count: 1

Queue: bestWayLen = 1 + 66.99994 Progress: 0.0% // queueLen = 4 (1) // mapSize = 4

Time needed: 0.6754666s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 93



Jump-Labyrinthe 9

Lab1 Best way: 546

Lab2 Best way: 700

Start jump count: 1

Queue: bestWayLen = 1 + 698.99846 Progress: 0.0% // queueLen = 4 (1) // mapSize = 4

Queue: bestWayLen = 174 + 597.99952 Progress: 13.18% // queueLen = 6333909 (4194305) // mapSize = 4781004

Queue: bestWayLen = 135 + 657.99847 Progress: 17.03% // queueLen = 9789452 (8388609) // mapSize = 6458102

Queue: bestWayLen = 263 + 543.9989999999999 Progress: 19.59% // queueLen = 12619093 (12582913) // mapSize = 7530331

Queue: bestWayLen = 144 + 674.9987 Progress: 21.79% // queueLen = 15273769 (16777217) // mapSize = 8721207

Queue: bestWayLen = 218 + 610.99901 Progress: 23.62% // queueLen = 17452724 (20971521) // mapSize = 9910183

Queue: bestWayLen = 210 + 627.99814 Progress: 25.27% // queueLen = 19490796 (25165825) // mapSize = 11167829

Queue: bestWayLen = 360 + 484.9993400000001 Progress: 26.55% // queueLen = 21558861 (29360129) // mapSize = 12414179

Queue: bestWayLen = 162 + 689.99773 Progress: 27.83% // queueLen = 23665941 (33554433) // mapSize = 13689653

Queue: bestWayLen = 588 + 268.99987 Progress: 28.75% // queueLen = 25725707 (37748737) // mapSize = 14930661

Queue: bestWayLen = 255 + 607.99832 Progress: 29.85% // queueLen = 27844727 (41943041) // mapSize = 16208145

Queue: bestWayLen = 148 + 719.99762 Progress: 30.76% // queueLen = 29952669 (46137345) // mapSize = 17489418

Queue: bestWayLen = 489 + 382.99951 Progress: 31.5% // queueLen = 31939950 (50331649) // mapSize = 18652766

Queue: bestWayLen = 357 + 518.99996 Progress: 32.23% // queueLen = 33983284 (54525953) // mapSize = 19892241

Queue: bestWayLen = 214 + 666.99766 Progress: 33.14% // queueLen = 35998849 (58720257) // mapSize = 21083655

Queue: bestWayLen = 381 + 503.0 Progress: 33.69% // queueLen = 38028451 (62914561) // mapSize = 22317436

Queue: bestWayLen = 530 + 357.99969 Progress: 34.43% // queueLen = 40012457 (67108865) // mapSize = 23487592

Queue: bestWayLen = 249 + 642.99894 Progress: 35.16% // queueLen = 42073992 (71303169) // mapSize = 24730284

Queue: bestWayLen = 215 + 679.9998499999999 Progress: 35.71% // queueLen = 44139376 (75497473) // mapSize = 26030499

Queue: bestWayLen = 399 + 499.99852 Progress: 36.44% // queueLen = 46206959 (79691777) // mapSize = 27291428

Queue: bestWayLen = 582 + 319.99926000000005 Progress: 36.99% // queueLen = 48262273 (83886081) // mapSize = 28577308

Queue: bestWayLen = 726 + 178.99955 Progress: 37.54% // queueLen = 50368306 (88080385) // mapSize = 29918006

Queue: bestWayLen = 560 + 347.99963 Progress: 38.09% // queueLen = 52518440 (92274689) // mapSize = 31292285

Queue: bestWayLen = 513 + 397.99954 Progress: 38.64% // queueLen = 54691730 (96468993) // mapSize = 32687564

Time needed: 594.9879s

Using heuristic: WeightedAverage: 0.99999-1.0E-5

Length: 911