Aufgabe 1: Arukone

Team-ID: 00206

Team-Name: Julian

Bearbeiter/-innen dieser Aufgabe:
Julian Bents

19. November 2023

Inhaltsverzeichnis

1 Lösungsidee

Arukone Rätsel sind Denksportaufgaben. Mein erster Ansatz bestand deswegen darin, ein System zu entwickeln, mit dem ich in der Lage bin, zu 100% lösbare Rätsel mithilfe einer standardmäßigen Methode zu generieren.
Hierfür habe ich einen Generator geschrieben, der das n^2 große Gitter mit $n/2$ zufälligen Paaren füllt. Diese habe ich anschließend mit dem bereitgestellten Löser auf Lösbarkeit geprüft und die lösbaren Rätsel auf Gemeinsamkeiten hin untersucht. Leider konnte ich dabei keine signifikanten Gemeinsamkeiten identifizieren.
Daraufhin wurde mir klar, dass eine zufällige Generierung der Rätsel erforderlich ist. Mein Ansatz war es, einen Algorithmus zu entwickeln, der sich wie eine Pflanze auf dem Gitter verhält. Ähnlich dem Wachstum einer Pflanze beginnt der Algorithmus an einem zufälligen Punkt und breitet sich entlang der freien Felder im Gitter aus, wodurch lange Pfade entstehen. Die Endpunkte dieser Pfade können dann als Start und Ende eines Nummernpaares dienen, und das Arukone Rätsel ist fertig.

2 Umsetzung

Die Implementierung beginnt damit, zwei gleich große Gitter zu erstellen und beide mit Nullen zu initialisieren. Das erste Gitter dient dazu, den gesamten Pfad zu speichern, während das zweite nur die Endpunkte der Pfade festhält.
Die Erzeugung verschiedener Pfade erfolgt durch zufällige Generierung. Um einen Pfad zu generieren, wird zunächst ein zufälliger Punkt auf dem Gitter gewählt. Falls auf diesem Punkt noch kein Pfad existiert, wird ein Loop begonnen. In diesen wird erst eine zufällige Richtung generiert. Dann wird es überprüft, ob der Pfad in dieser Richtung wachsen kann. Dies beinhaltet das Überprüfen, ob sich in dieser Richtung eine Grenze befindet und ob sich bereits ein Pfad auf dem Feld befindet. Zusätzlich wird geprüft, ob sich relativ zum neuen Kopf des Pfades rechts und links bereits der Pfad selber befindet. Dies verhindert ungewollte Muster der Pfade. In Abbildung 1 ist dies zu sehen, es zeigt welche Felder genau überprüft werden, falls der Pfad sich im Bild nach unten bewegen würde:

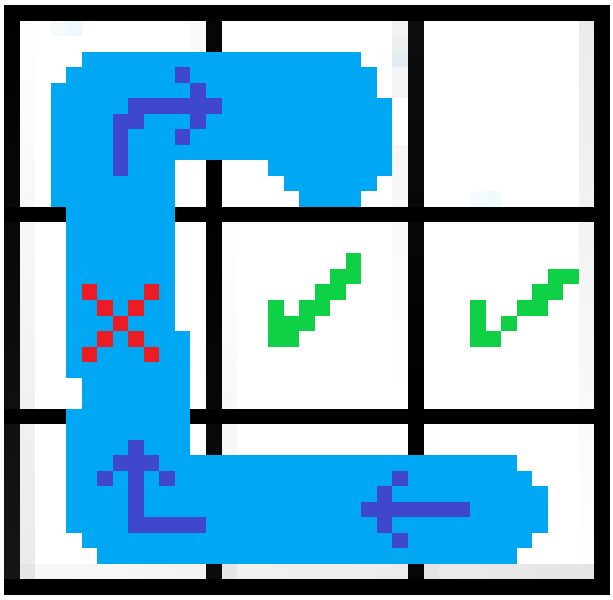
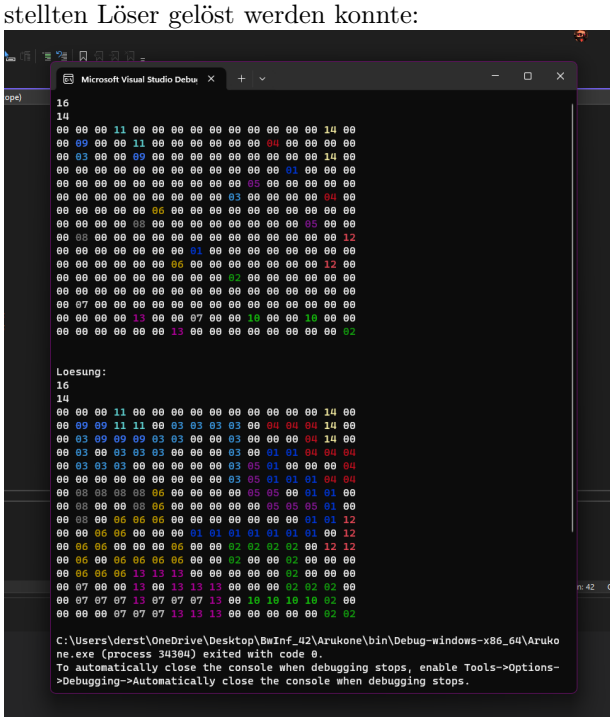


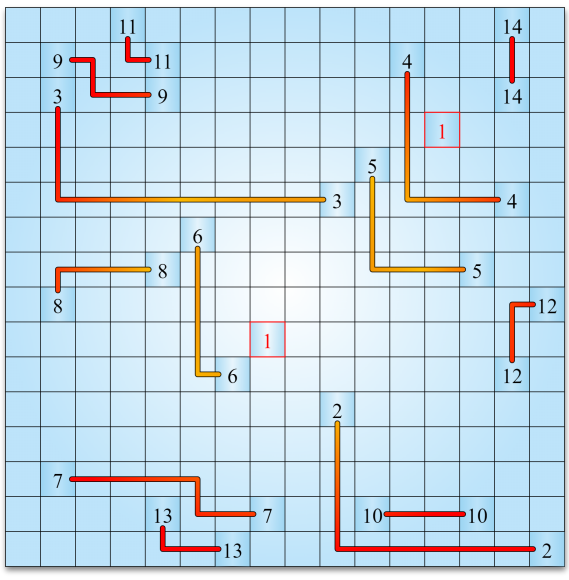
Abbildung 1: Darstellung zur Pfadgenerierung

In der Darstellung ergibt sich, dass der Pfad sich nicht in die Richtung weiterbewegen kann, da links vom neuen Kopf sich der Pfad selber befindet. Falls der Pfad sich fünfmal hintereinander nicht weiterbewegen kann, wird er als abgeschlossen betrachtet, im Gitter final gespeichert und der Loop wird beendet. Sollte der Pfad eine Länge von weniger als zwei haben, wird er aus dem Gitter entfernt. Der Pfad gilt ebenfalls als abgeschlossen, wenn er länger als $2n$ ist. Dieser Prozess zur Pfadgenerierung wird $2n$ mal wiederholt. Falls nicht mindestens $n/2$ Pfade erstellt wurden, fängt der Prozess von vorne an.

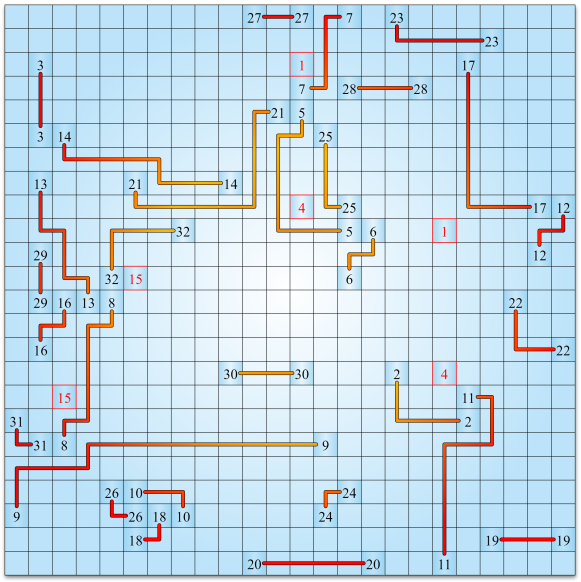
3 Beispiele

In folgenden ist ein gelöstes Arukone Rätsel mit $n = 16$ erstellt worden, welches nicht von den bereitgestellten Löser gelöst werden konnte:

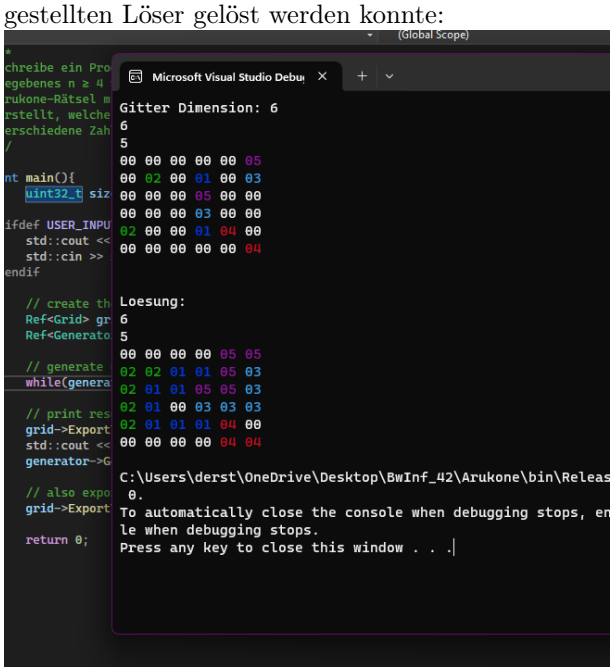


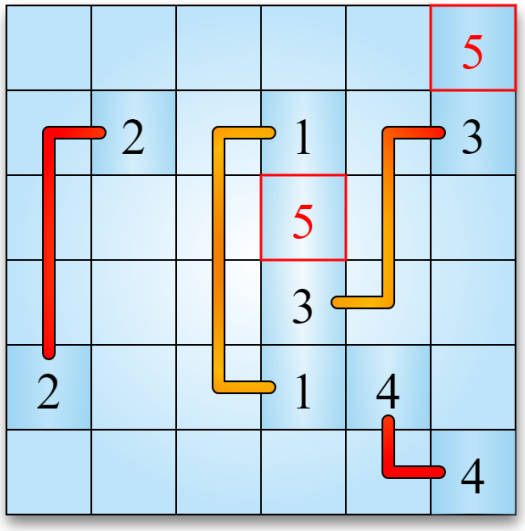


In folgenden ist ein gelöstes Arukone Rätsel mit $n = 24$ erstellt worden, welches nicht von den bereitgestellten Löser gelöst werden konnte. Die blauen Balken können ignoriert werden, denn Windows füllt die ganze Zeile mit der Hintergrundfarbe, wenn eine neue Linie gedruckt wird:

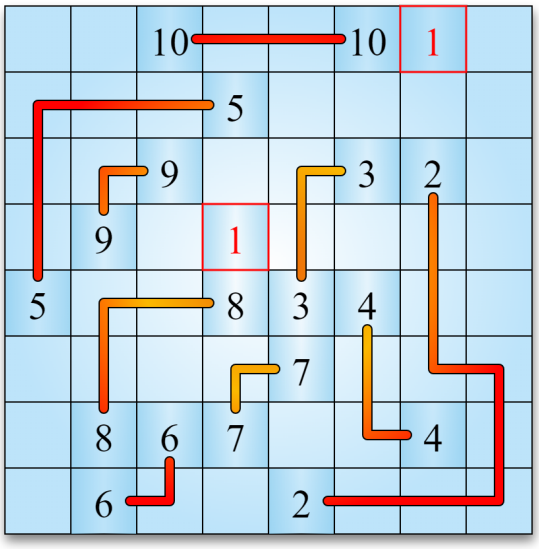


In folgenden ist ein gelöstes Arukone Rätsel mit $n = 6$ erstellt worden, welches nicht von den bereitgestellten Löser gelöst werden konnte:





In folgenden ist ein gelöstes Arukone Rätsel mit $n = 8$ erstellt worden, welches nicht von den bereitgestellten Löser gelöst werden konnte:



4 Quellcode

Die Main Funktion des Programmes, hier wird der Input genommen, die einzelnen Objekte erstellt, der Generator ausgelöst und am Ende wird das Ergebnis zur Konsole gedruckt. Die Grid Klasse ist lediglich ein Wrapper für eine 2D Array.

```
int main() {
#ifdef USER_INPUT
    std::cout << "Gitter Dimension: ";
    uint32_t size;
    std::cin >> size;
#else
    constexpr uint32_t size = 16;
#endif

    // create the grid as shared pointer to controle ownership correctly
    Ref<Grid> grid = std::make_shared<Grid>(uvec2{ size, size });

    Generator generator = Generator(grid);

    // generate until valid
    while(generator.Generate() < (uint32_t)std::ceil((float)size/2.0f)){}

    // print result in a fancy way to console
    grid->ExportToConsole();
    std::cout << "\n\nLoesung:\n";
    generator.GetPathGrid()->ExportToConsole();

    // also export to file
    grid->ExportToFile("export.txt");

    return 0;
```

```
}
```

In folgenden Code ist die Generate Methode der Generator Klasse zu finden. In ihr passiert alles, was in der Umsetzung beschrieben wurde.

```
uint32_t Generator::Generate() {
  const uvec2 size = m_grid->GetSize();

  m_grid->Clear();
  m_pathGrid = std::make_shared<Grid>(size);

  const uint32_t MaxNumTries = 2 * size.x;

  // we try a maximum amount of times to generate paths
  uint32_t path = 0;
  for (uint32_t itTry = 0; itTry < MaxNumTries; itTry++) {
    uint32_t sX, x;
    uint32_t sY, y;
    // generate a random starting point
    sX = x = random(size.x - 1);
    sY = y = random(size.y - 1);

    // if there is already a value assigned we just jump to the next iteration
    if (m_pathGrid->HasValue(x, y)) {
      continue;
    }

    // set the starting point in the grid
    m_pathGrid->SetValue(sX, sY, ++path);

    // keep track of some stats
    uint32_t repeatCounter = 0;
    uint32_t lengthCounter = 0;
    while (true) {
      // generare a random direction, move to it,
            // if there is no value, border and check for neighbouring cells
      uint32_t dir = random(3);
      if (dir == 0 && x < size.x - 1) {
        /*
        0 1>1 <-- Should not happen, check y-1 and y+1 for same
        0 1 1
        0 0 1
        */
        if (!m_pathGrid->HasValue(x + 1, y)) {
          if (m_pathGrid->GetValueSafe(x + 1, y + 1) != path &&
                        m_pathGrid->GetValueSafe(x + 1, y - 1) != path) {
            x++;
            lengthCounter++;
          }
        }
      }
      else if (dir == 1 && x > 1) {
        /*
        1<1 0 <-- Should not happen, check y+1 and y-1 for same
        0 1 1
        0 0 1
        */
        if (!m_pathGrid->HasValue(x - 1, y)) {
          if (m_pathGrid->GetValueSafe(x - 1, y + 1) != path &&
            m_pathGrid->GetValueSafe(x - 1, y - 1) != path) {
            x--;
            lengthCounter++;
          }
        }
      }
      else if (dir == 2 && y < size.y - 1) {
        /*
        1 1 1 <-- Should not happen, check x+1 and x-1 for same
        0 1 1
        0 0 0
        */
        if (!m_pathGrid->HasValue(x, y + 1)) {
```

```
            if (m_pathGrid->GetValueSafe(x + 1, y + 1) != path &&
              m_pathGrid->GetValueSafe(x - 1, y + 1) != path) {
              y++;
              lengthCounter++;
            }
          }
        }
        else if (dir == 3 && y > 1) {
          /*
          1 1 0
          0 1 1
          0 0 1 <-- Should not happen, check x-1 and x+1 for same
          */
          if (!m_pathGrid->HasValue(x, y - 1)) {
            if (m_pathGrid->GetValueSafe(x + 1, y - 1) != path &&
              m_pathGrid->GetValueSafe(x - 1, y - 1) != path) {
              y--;
              lengthCounter++;
            }
          }
        }

        // count how many times we have been on the same field and break if to many
        if (m_pathGrid->HasValue(x, y)) {
          repeatCounter++;
        }
        else {
          repeatCounter = 0;
        }

        m_pathGrid->SetValue(x, y, path);

        if (repeatCounter == 5 || lengthCounter > size.x) {
          // if path is to small remove it
          if (lengthCounter < 2) {
            m_pathGrid->Erase(path);
            path--;
          }
          else {
            // if path wasnt erased save it to the original grid
            m_grid->SetValue(sX, sY, path);
            m_grid->SetValue(x, y, path);
          }
          break;
        }
      }
    }
    // save the number of pairs
    m_grid->SetNumPairs(path);
    m_pathGrid->SetNumPairs(path);
    return path;
}
```