Sortiernetzwerke

Christian Brosy
Matrikelnummer: 2945100

22. Juli 2015

Institut für Theoretische Informatik
Leibniz Universität Hannover

Erstprüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Arne Meier

Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen genutzt habe.

Christian Brosy

Inhaltsverzeichnis

1 Einleitung

Die Theoretische Informatik untersucht grundlegende Konzepte, Modelle und Vorgehensweisen, die allen Bereichen der Informatik zugrunde liegen. Zu diesen Konzepten gehört auch das Sortieren. Sortiernetzwerke bilden die Grundlage, wenn Elemente durch Hardware optimal geordnet werden sollen. Sie können durch verschiedene Algorithmen erstellt werden, die unterschiedliche Laufzeiten beim Sortieren aufweisen. Die Komplexität dieser Laufzeiten kann durch Landau-Symbole beschrieben werden.

1.1 Sortieren

Sortieren – ein Gebiet das wohl schon jedem jungen Informatiker begegnet ist. Es ist um einiges schwerer ein Lexikon zu benutzen, wenn dies nicht alphabetisch sortiert wäre. Nach dem deutschen Wörterbuch ist die Definition von Sortieren nach Art, Farbe, Größe, Qualität o. Ä. zu sondern bzw. ordnen [5]. Programmierer denken beim Sortieren eher daran Elemente auf- oder absteigend zu sortieren. Die Ordnung von Listen, auf denen Operationen ausgeführt werden sollen, hat oft direkten Einfluss auf die Geschwindigkeit und Komplexität eines Sortieralgorithmus bei seiner Ausführung.

1.2 Vergleichernetz

Ein Vergleichernetz ist die Vorstufe eines Sortiernetzwerkes. Ein *Vergleicher* ist eine Abbildung $[i{:}j] : A^n \to A^n, i, j \in \{0, \ldots, n-1\}$ mit $[i{:}j](a)_i = \min(a_i, a_j)$ und $[i{:}j](a)_j = \max(a_i, a_j)$. Die Hintereinanderausführung von Vergleichern nennt man ein *Vergleichernetz*. Das Vergleichernetz ist dabei auf eine bestimmte Eingabefolge angewiesen, es kann also nicht jede beliebige Eingabefolge sortieren. Es wird *Standard-Vergleichernetz* genannt, wenn alle Vergleicher von oben nach unten gerichtet sind, also wenn für alle Vergleicher $[i{:}j]$ gilt: $i < j$. *Primitiv* wird es genannt, wenn alle Vergleicher nur zwischen zwei benachbarten Leitungen liegen, also wenn für alle Vergleicher gilt: $[i : i\text{-}1]$. [9, S. 293-296]

1.3 Sortiernetzwerke

Ein *Sortiernetzwerk* ist ein Vergleichernetz, das alle Eingabedaten sortieren kann. Es wird in der Regel in Hardware implementiert, damit eine schnelle Ausführung sichergestellt werden kann. Damit ein Netzwerk n Elemente sortieren kann, benötigt es die Leitungen $K_1 \ldots K_n$ auf denen Vergleicher angebracht werden können. Durch die Ausführung

eines Sortiernetzwerkes sind die Elemente zum Schluss aufsteigend sortiert. Abbildung 1.1 zeigt ein Sortiernetzwerk der Größe 4, welches jede Eingabefolge aufsteigend sortieren kann. Ein *Vergleicher* wird als eine Verbindung zwischen zwei Leitungen K_a und K_b dargestellt, für die gilt: $a, b \in 1, \ldots, n$ und $a \neq b$. Es ist NP-schwer herauszufinden, ob ein gegebenes Vergleichernetz ein Sortiernetzwerk ist. [9, S. 295]

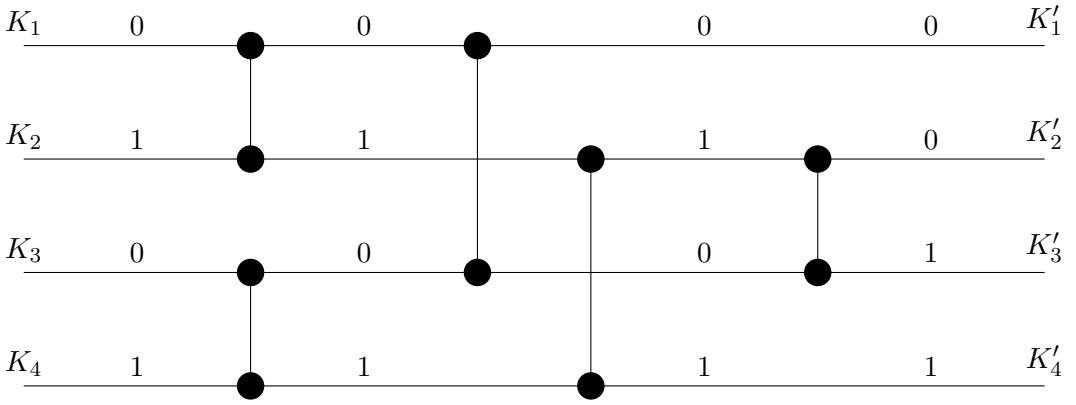

Abbildung 1.1: Ein Sortiernetzwerk, dass 4 Elemente sortieren kann.

Jedes Sortiernetzwerk kann durch einen Binärbaum dargestellt werden und umgekehrt. Jeder Vergleich von zwei Leitungen fügt eine Ebene im Binärbaum hinzu. Nach m Vergleichen hat man also 2^m mögliche Ergebnisse. Es gibt aber $n!$ verschiedene Möglichkeiten n Elemente zu sortieren. Daraus kann man schließen, dass einige Vergleiche redundant sind, da $n!$ kein Potenz von 2 ist für $n \geq 3$. Soll nun ein Netzwerk in ein Binärbaum umgewandelt werden, fängt man mit dem ersten Vergleich als Wurzel im Binärbaum an. Der linke Ausgang dieses Vergleichs hat nun die Eigenschaft $K_i > K_j$ und der rechte Ausgang $K_j > K_i$ (siehe Abbildung 1.2).[7, S.219-222]

Ein Algorithmus kann als Sortiernetzwerk implementiert werden, wenn er datenunabhängig ist. Das bedeutet, der Algorithmus darf sich nicht unterschiedlich bei verschiedenen Eingabedaten verhalten, sondern muss für eine bestimmte Größe immer gleich arbeiten.

1.4 Asymptotische Komplexität

Die *Größe* eines Netzwerks N wird anhand der Anzahl der Vergleicher gemessen, die in N vorkommen. Die *Tiefe* von N ist definiert durch die Leitung K, welche am meisten Vergleicher durchläuft. Die Tiefe sagt außerdem aus, wie viele Schritte das Sortiernetzwerk bei paralleler Ausführung maximal benötigt, bis dessen Sortierung fertig ist. Primitive Netzwerke benötigen immer $O(n^2)$ Vergleicher beim Sortieren. Effizientere Netzwerke, die $O(n \log(n)^2)$ Vergleicher benötigen, werden durch Verfahren wie Batcher's Odd Even Methode erzeugt. Solche Verfahren werden seit ihrer Entwicklung bis heute aktiv in der Praxis angewendet. Es existiert auch ein Verfahren von Ajtai, Komlós und Szemerédi,

welches Netzwerke mit einer Größe von $O(n \log(n))$ erzeugen kann. Dies ist allerdings aufgrund seiner hohen Konstanten erst bei Größen von über 2^{6100} besser als die in der Praxis verwendeten Verfahren [11]. Keines dieser Verfahren erzeugt aber optimale Netzwerke. Ein *optimales* Vergleicher-Sortiernetzwerk N_{optv} ist ein Netzwerk, bei dem kein anderes Netzwerk gleicher Größe existiert, das weniger Vergleicher braucht als N_{optv}. Ein optimales Tiefe-Sortiernetzwerk N_{optt} ist ein Netzwerk, bei dem kein anderes Netzwerk existiert, das eine geringere Tiefe hat. In der Regel ist $N_{optv} \neq N_{optt}$. Für die Netzwerke der Größe $N \leq 16$ sind optimale Netzwerke bekannt. [9] [7, S.225-229]

1.5 0-1-Prinzip

Es ist NP-schwer herauszufinden, ob ein gegebenes Vergleichernetzwerk ein Sortiernetzwerk ist. Hilfreich für diese Überprüfung ist das 0-1-Prinzip, wenn es um den Beweis geht, ob das vorhandene Vergleichernetzwerk ein Sortiernetzwerk ist. Es besagt, dass das Netzwerk nur von der Struktur abhängig ist und nicht vom Eingabedatenbereich A. Die *Struktur* eines Netzwerkes ist die Anzahl sowie Platzierung der Vergleicher.

Satz (0-1-Prinzip)**.** *Ein Vergleichernetz, das alle Folgen von Nullen und Einsen sortiert, ist ein Sortiernetz (d.h. es sortiert auch alle Folgen von beliebigen Werten).*

Für den Beweis wird das 0-1-Prinzip umgekehrt formuliert:

Satz (0-1-Prinzip)**.** *Sei N ein Vergleichernetz. Wenn es eine beliebige Folge gibt, die von N nicht sortiert wird, dann gibt es auch eine 0-1-Folge, die von N nicht sortiert wird.*

Beweis. Sei a mit $a_i \in A$ eine Folge, die von N nicht sortiert werden kann. Das bedeutet, dass $N(a) = b$ unsortiert ist und es keinen Index k mit $b_k < b_{k+1}$ gibt.
Definiert wird die Abbildung $f : A \to \{0, 1\}$ wie folgt: Für alle $c \in A$ sei

$$f(c) = \begin{cases} 0 & \text{falls } c < b_k \\ 1 & \text{falls } c \geq b_k \end{cases}$$

Anscheinend ist f monoton, folglich gilt:

$$f(b_k) = 0 \text{ und } f(b_{k+1}) = 1$$

Das bedeutet, dass $f(b) = f(N(a))$ unsortiert sind. Damit ist die Folge $N(f(a))$ auch unsortiert, was bedeutet, dass die 0-1-Folge ebenfalls unsortiert ist. [9][S.296-298]

□

1.6 Ziel der Arbeit

Ziel der Arbeit ist die Entwicklung eines modularen Programms, welches Daten visuell darstellen kann und in welchem neue Sortieralgorithmen einfach implementiert werden

können. Zu jedem gültigen Algorithmus kann das Programm automatisch ein Balkendiagramm und ein Sortiernetzwerkdiagramm erstellen. Das Balkendiagramm stellt zu dem gegebenen Zeitpunkt visuell die Werte dar, die zu sortieren sind. Das Sortiernetzwerkdiagramm kann den Verlauf der Werte auf den einzelnen Leitungen visuell darstellen. Dies ist auch für mehrere Sortiernetzwerke gleichzeitig möglich, es kann also direkt zur Laufzeit zwei Algorithmen miteinander vergleichen. Die visuell dargestellten Diagramme können auch zur Laufzeit in ihrer Größe, wie sie auf dem Bildschirm dargestellt werden, verändert werden. Zu dem Programm werden auch die in der Praxis verwendeten Algorithmen implementiert und auf ihre Laufzeit verglichen.

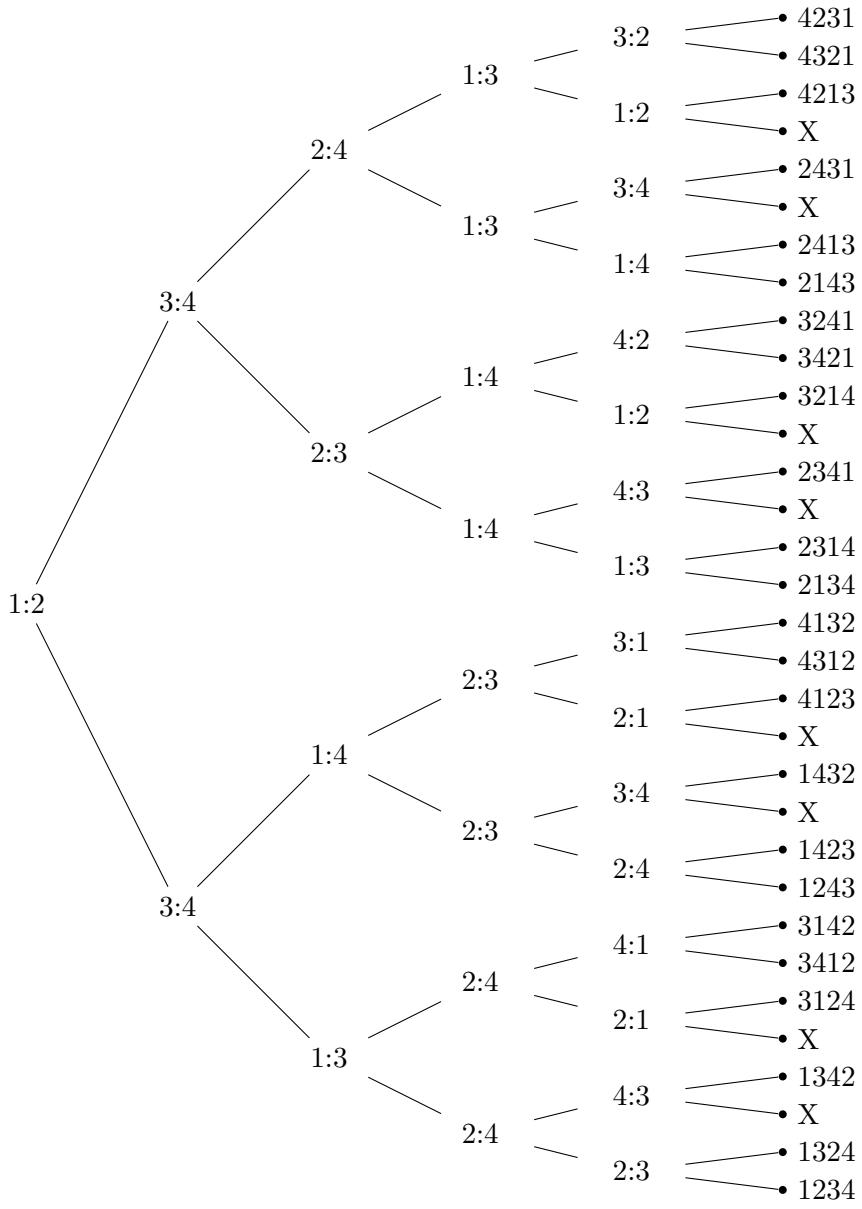

Abbildung 1.2: Der korrespondierende Binärbaum zu dem Sortiernetzwerk in Abb. 1.1

2 Algorithmen

In diesem Kapitel sollen die in dem Programm benutzten Algorithmen zum Erstellen eines Sortiernetzwerks gezeigt und anhand eines Beispiels deren Arbeitsweise erläutert werden. Es wurden Algorithmen ausgewählt, die in der Praxis aktiv eingesetzt werden. Im Folgendem wird auf die Algorithmen Batcher Bitonic, Batcher Odd Even, PairwiseSort und Shellsort eingegangen.

2.1 Batcher's Bitonic Verfahren

Batcher's Bitonic Verfahren ist ein parallel arbeitendes Sortierverfahren, das auch als Sortiernetzwerk implementiert werden kann. Die Konstruktion dieses Netzwerks benötigt $O(n \log(n)^2)$ Vergleicher und maximal $O(\log(n)^2)$ Schritte, wenn es parallel ausgeführt wird. Es besitzt die gleiche asymptotische Komplexität wie Batcher's Odd Even Verfahren, Pairwise-Sorting und Shellsort. Das Verfahren besteht aus zwei Komponenten, dem Bitonic-Sort und dem Bitonic-Merge. Die bitonische Folge (siehe Abschnitt 2.1), die für den Bitonic-Merge benötigt wird, wird rekursiv von Bitonic-Sort erzeugt, indem es die beiden Hälften der Folge gegenläufig sortiert (rekursiver Aufruf von Bitonic-Sort mit den beiden Hälften). Bitonic-Merge sortiert rekursiv eine bitonische Folge. Das Bitonic Verfahren kann so erweitert werden, dass es Sortiernetzwerke erstellen kann, deren Größe ungleich $\log_2(n) : n \in 2^b, b \in \mathbb{N}$ ist. [7, S.230-232] [9, S. 302-309]

Bitonische Sequenz

Eine Folge $A(a_0, \ldots, a_{n-1})$ heißt bitonisch, wenn für ihre Folgeglieder gilt: $a_0 \leq \cdots \leq a_k \geq \cdots \geq a_{n-1}$ für $0 \leq k < n$ oder es für einen beliebigen zyklischen Shift dieser Folge gilt. Ein zyklischer Shift ist eine Verschiebung der Elemente einer Folge, welcher sowohl links als auch rechts erfolgen kann. Wenn ein zyklischer Links-Shift auf die Folge A erfolgt, verschieben sich alle Elemente a_i nach a_{i-1} und a_0 verschiebt sich nach a_{n-1}. Bei einem zyklischen Rechts-Shift verschieben sich die Elemente von a_i nach a_{i+1} und a_{n-1} verschiebt sich nach a_0.

Beispiel:

$1, 5, 6, 8, 9, 6, 2, 1$ ist eine bitonische Folge für $k = 4$.

$1, 2, 3, 4, 5, 6, 7, 8$ ist eine bitonische Folge für $k = n - 1$.

$5, 4, 1, 3, 6, 8, 7, 6$ ist eine bitonische Folge für einen zyklischen Links Shift von 2 und $k = 3$.

$1, 8, 3, 4, 2, 5, 6, 8$ ist keine bitonische Folge.

Algorithmus

Sei die Größe des Netzwerks gegeben durch die Folge $A(a_0, \ldots, a_{n-1})$, wobei jedes Folgeglied eine Leitung des Sortiernetzwerks darstellt. Zusätzlich sei die Richtung der Vergleicher durch *direction* gegeben. Ein Vergleicher mit *direction* = *true* arbeitet normal, einer mit *direction* = *false* hat einen vertauschten Ausgang und wird *ReverseVergleicher* genannt. Der ReverseVergleicher tauscht die Werte, wenn $K_i < K_j$ und $i < j$. Ist *direction* beim ersten Aufruf *true*, dann sortiert das Sortiertnetzwerk aufsteigend, anderfalls absteigend. Die Funktion *length* gibt die Anzahl der Folgeglieder zurück. Die Variable Z beschreibt den Stack, in dem die Vergleicher eines Netzwerkes gespeichert werden. Sie ist auch von dem Algorithmus BitonicMerge aufrufbar (siehe Algorithmen 1 und 2). [9, S. 308-309]

Algorithmus 1 : BitonicSort

```
   input : Folge von A = (a_0, ..., a_i, ..., a_{n-1}), direction
   output : Position der Vergleicher als Stack Z
1  B ← A ;
2  if (B.length > 1) then
3  |  BitonicSort(b_0, ..., a_{B.length/2}, true);
4  |  BitonicSort(b_{B.length/2}, ..., b_{B.length-1}, false);
5  |  BitonicMerge(B, directionInput);
6  end
```

Algorithmus 2 : BitonicMerge

```
   input : Folge von A = (a_0, ..., a_i, ..., a_{n-1}), direction
1  B ← A ;
2  if (B.length > 1) then
3  |  for i ≤ b_0 < B.length/2 do
4  |  |  if (direction = true) then
5  |  |  |  Füge zu Z Vergleicher (b_i, b_{B.length/2+i}) hinzu;
6  |  |  else if (direction = false) then
7  |  |  |  Füge zu Z ReverseVergleicher (b_i, b_{B.length/2+i}) hinzu;
8  |  |  end
9  |  end
10 |  BitonicMerge(b_0, ..., b_{B.length/2}, direction);
11 |  BitonicMerge(b_{B.length/2}, ..., b_{B.langth-1}, direction);
12 end
```

Beispiel Bitonic Verfahren

Das Beispiel soll die Funktionsweise des Algorithmus genauer erläutern. Es soll ein Sortiernetzwerk der Größe 8 erstellt werden, welches aufsteigend sortiert. Dabei wird jeder Schritt genauer erläutert. Der Startaufruf erfolgt mit der Folge $a_0, \ldots, a_7$ und dem Directionwert *true*. Jedes Folgeglied repräsentiert eine Leitung auf dem Sortiernetzwerk.

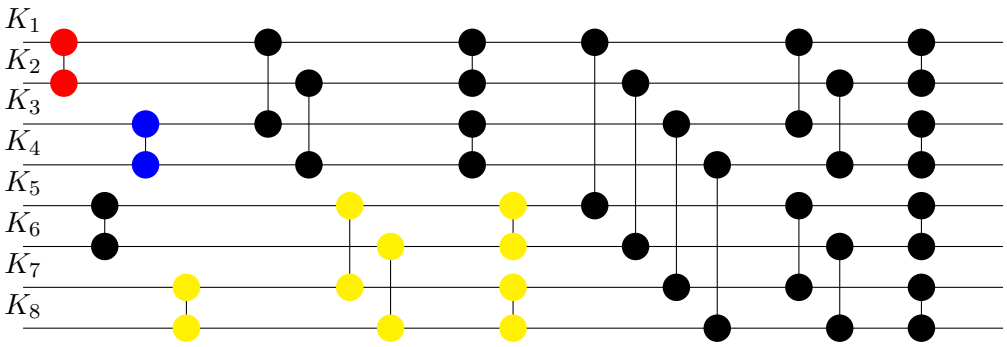

Abbildung 2.1: Bitonic Sortiernetzwerketzwerk. Reverse Vergleicher in Gelb dargestellt. In Rot der in Zeile 5 Alg. 2 und in Blau der in Zeile 7 Alg. 2 erstellte Vergleicher.

Zeile 3 Alg. 1 $n = 8$, Rekursiver Aufruf von BitonicSort mit $a_0, \ldots, a_3$ und *direction* = *true*.

Zeile 3 Alg. 1 $n = 4$, Rekursiver Aufruf von BitonicSort mit a_0, a_1 und *direction* = *true*.

Zeile 3-4 Alg. 1 $n = 2$, Rekursiver Aufruf von BitonicSort mit a_0 und *direction* = *true* und a_1 und *direction* = *false*. Return $n = 1$.

Zeile 5 Alg. 1 Aufruf von BitonicMerge mit a_0, a_1 und *direction* = *true*.

Zeile 5 Alg. 2 Füge Vergleicher(a_0, a_1) zu Z hinzu.

Zeile 10-11 Alg. 2 Rekursiver Aufruf von BitonicMerge mit a_0 und *direction* = *true* und a_1 und *direction* = *true*. Return, $n = 1$.

Zeile 4 Alg. 1 $n = 4$, Rekursiver Aufruf von BitonicSort mit a_2, a_3 und *direction* = *false*.

Zeile 3-4 Alg. 1 $n = 2$, Rekursiver Aufruf von BitonicSort mit a_2 und *direction* = *true* und a_3 und *direction* = *false*. Return, $n = 1$.

Zeile 5 Alg. 1 Aufruf von BitonicMerge mit a_2, a_3 und *direction* = *false*.

Zeile 7 Alg. 2 Füge ReverseVergleicher(a_2, a_3) zu Z hinzu.

Zeile 10-11 Alg. 2 u.s.w.

Abbildung 2.1 zeigt in rot den in Zeile 5 Alg. 2 erzeugten Vergleicher und in blau den in Zeile 7 Alg. 2 erzeugten ReverseVergleicher.

2.2 Batcher's Odd Even Methode

Batcher's Odd Even Mergesort ist ein Konstruktionalgorithmus für Sortiernetzwerke der nicht benachbarte Schlüssel (K_i, K_j) miteinander vergleicht. Das erstellte Sortiernetzwerk braucht $O(n \log(n)^2)$ Vergleicher und benötigt maximal $O(\log(n)^2)$ Schritte, wenn es parallel ausgeführt wird. Der Batcher's Odd Even Algorithmus kann auch Netzwerke aufbauen, deren Größe ungleich $\log_2(n) : n \in 2^b, b \in \mathbb{N}$ ist.

Algorithmus

Sei $(R_1, \ldots, R_N)$ die Anzahl der Leitungen, $t = \lceil \log(N) \rceil$ der aufgerundete Logarithmus von N und $\wedge$ der Operator, der zwei Zahlen bitweise logisch UND-verknüpft, sodass 13 $\wedge$ 21 $= (1101)_2 \wedge (10101)_2 = (00101)_2 = 5$ ergibt. Die Variable Z beschreibt den Stack, in dem die Vergleicher eines Netzwerkes gespeichert werden. Die Hilfsvariablen, die für den Algorithmus benötigt werden, werden p, q, r, d genannt (siehe Algorithmus 3). [7, S. 111-112]

Beispiel Batcher's Odd Even

Das Beispiel soll die Funktionsweise des Algorithmus schrittweise genauer erläutern. Es soll ein Sortiernetzwerk der Größe 8 erstellt werden.

Zeile 1 Alg. 3 Setzte p $\leftarrow 2^{3-1} = 4$.

Zeile 3-5 Alg. 3 Setzte q $\leftarrow 2^{3-1} = 4$, r $\leftarrow 0$ und d $\leftarrow p = 4$.

Zeile 9-10 Alg. 3 Vergleiche für $i = 0, 1, 2, 3$.

$0 \wedge 4 = 0 \equiv (0000)_2 \wedge (0100)_2 = (0000)_2$ Vergleicher für K_1 und K_5 einfügen.

$1 \wedge 4 = 0 \equiv (0001)_2 \wedge (0100)_2 = (0000)_2$ Vergleicher für K_2 und K_6 einfügen.

$2 \wedge 4 = 0 \equiv (0010)_2 \wedge (0100)_2 = (0000)_2$ Vergleicher für K_3 und K_7 einfügen.

$3 \wedge 4 = 0 \equiv (0011)_2 \wedge (0100)_2 = (0000)_2$ Vergleicher für K_4 und K_8 einfügen.

Zeile 19 Alg. 3 $q = p$. Setze $p \leftarrow \lfloor p/2 \rfloor = 2$.

Zeile 3-5 Alg. 3 Setzte q $\leftarrow 2^{3-1} = 4$, r $\leftarrow 0$ und d $\leftarrow p = 2$.

Zeile 9-10 Alg. 3 Vergleiche für $i = 0, 1, 2, 3, 4, 5$.

$0 \wedge 2 = 0 \equiv (0000)_2 \wedge (0010)_2 = (0000)_2$ Vergleicher für K_1 und K_3 einfügen.

$1 \wedge 2 = 0 \equiv (0001)_2 \wedge (0010)_2 = (0000)_2$ Vergleicher für K_2 und K_4 einfügen.

$4 \wedge 2 = 0 \equiv (0100)_2 \wedge (0010)_2 = (0000)_2$ Vergleicher für K_5 und K_7 einfügen.

Algorithmus 3 : Batcher's Odd Even Methode

input : Größe des Netzwerks
output : Position der Vergleicher als Stack Z

```
1  p ← 2^(t−1);
2  while p > 0 do
3  |   q ← 2^(t−1);
4  |   r ← 0;
5  |   d ← p;
6  |   pqcheck ← true;
7  |   while pqckeck do
8  |   |   pqcheck ← false;
9  |   |   for ∀i : 0 ≤ i < N − d und i ∧ p = r do
10 |   |   |   Füge Vergleicher (R_{i+1}, R_{i+d+1}) zu Z hinzu;
11 |   |   end
12 |   |   if q ≠ p then
13 |   |   |   pqcheck ← true;
14 |   |   |   d ← q − p;
15 |   |   |   q ← q/2;
16 |   |   |   r ← p;
17 |   |   end
18 |   end
19 |   p ← ⌊p/2⌋;
20 end
```

$5 \wedge 2 = 0 \equiv (0101)_2 \wedge (0010)_2 = (0000)_2$ Vergleicher für K_6 und K_8 einfügen.

Zeile 12-16 Alg. 3 $q \neq p$. Setze $pqcheck \leftarrow true, d \leftarrow 4 - 2 = 2, q \leftarrow 4/2 = 2, r \leftarrow 2$.

Zeile 9-10 Alg. 3 Vergleiche für $i = 0, 1, 2, 3, 4, 5$ u.s.w. ...

Abbildung 2.2 zeigt in Rot die für $i = 0, 1, 2, 3$ erzeugten Vergleicher und in Gelb die für $i = 0, 1, 2, 3, 4, 5$ erzeugten Vergleicher.

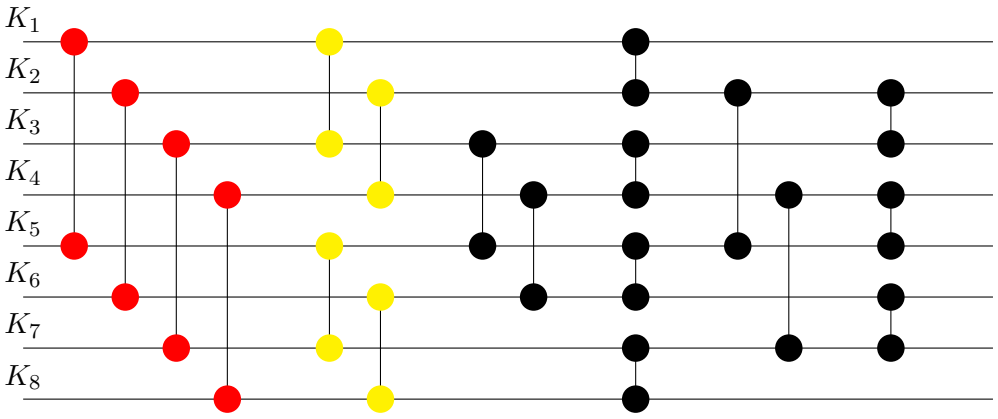

Abbildung 2.2: Batchers Odd Even Netzwerk. In Rot die für $i = 0, 1, 2, 3$ und in Gelb die für $i = 0, 1, 2, 3, 4, 5$ erzeugten Vergleicher.

2.3 Pairwise Sorting

Das Pairwise Sorting Netzwerk ist ein Sortiernetzwerk, welches $O(n \log(n)^2)$ Vergleicher benötigt und maximal $O(\log(n)^2)$ Schritte benötigt, wenn es parallel ausgeführt wird. Das konstruierte Netzwerk hat exakt so viele Vergleiche, wie das mit dem Odd Even Verfahren erstellte Netzwerk. Die Konstruktion erfolgt über einen rekursiven Algorithmus, dessen Kernverfahren ein Merge ist, der aus einer Folge $(a_0, a_1, \ldots, a_{n-1})$, deren geraden $(a_0, a_{i\cdot 2}, \ldots, a_{n-2} i \in \mathbb{N})$ und ungeraden $(a_1, a_{i\cdot 2+1}, \ldots, a_{n-1} i \in \mathbb{N})$ Teilfolgen bereits sortiert sind, eine insgesamt sortierte Folge macht. Zusätzlich gilt die Bedingung, dass alle Elemente der geraden Teilfolge kleiner oder gleich den Elementen der ungeraden Teilfolgen sind. Es können nur Netzwerke mit 2^n Elemente konstruiert werden. [10, 11]

Algorithmus

Sei die Größe des zu erstellenden Netzwerks als Folge $A(a_0, \ldots, a_{n-1})$, die gerade Teilfolge als $a_{i\cdot 2}$ für $0 \leq i \leq \lfloor (n-1)/2 \rfloor$ und die ungerade Teilfolge als $a_{i\cdot 2+1}$ für $0 \leq i \leq \lfloor (n-1)/2 \rfloor$ gegeben. Die Variable Z beschreibt den Stack, in dem die Vergleicher eines Netzwerkes gespeichert werden. Dieser Stack ist global, das bedeutet, dass der Algorithmus 5 auf ihn auch zugreifen kann (siehe Algorithmus 4 und 5). [10]

Algorithmus 4 : PairwiseSort

input : Größe des Netzwerks als Folge A
output : Position der Vergleicher als Stack Z

```
1 if n > 1 then
2 |   for ∀i : i mod 2 = 0 do
3 |   |   Füge Vergleicher (a_i, a_{i+1}) zu Z hinzu;
4 |   end
5 |   PairwiseSort(n/2) rekursiv auf die gerade Teilfolge anwenden;
6 |   PairwiseSort(n/2) rekursiv auf die ungerade Teilfolge anwenden;
7 |   PairwiseMerge(A);
8 end
```

Algorithmus 5 : PairwiseMerge

input : Teilfolge von $A = (a_0, \ldots, a_{n-1})$

```
1 d ← n/2;
2 while d > 1 do
3 |   k ← d;
4 |   while k < n do
5 |   |   Füge Vergleicher (a_{k−d+1}, k) zu Z hinzu;
6 |   |   k ← k + 2 ;
7 |   end
8 |   d ← d/2;
9 end
```

Beispiel Pairwise Sorting

Das Beispiel soll die Funktionsweise des Algorithmus schrittweise genauer erläutern. Es soll ein Sortiernetzwerk der Größe 8 erstellt werden. Die Folgeglieder von A entsprechen jeweils einer Leitung des Sortiernetzwerks.

Zeile 2-3 Alg. 4 Vergleiche für $i = 0, 2, 4, 6$.

$i = 0$ Füge Vergleicher(a_0, a_1) zu Z hinzu.

$i = 1$ Füge Vergleicher(a_2, a_3) zu Z hinzu.

$i = 1$ Füge Vergleicher(a_4, a_5) zu Z hinzu.

$i = 1$ Füge Vergleicher(a_6, a_7) zu Z hinzu.

Zeile 5 Alg. 4 Rufe PairwiseSort für die gerade Teilfolge a_0, a_2, a_4, a_6 auf.

Zeile 2-3 Alg. 4 Vergleiche für $i = 0, 4$.

$i = 0$ Füge Vergleicher(a_0, a_2) zu Z hinzu.

$i = 1$ Füge Vergleicher(a_4, a_6) zu Z hinzu.

Zeile 5 Alg. 4 Rufe PairwiseSort für die gerade Teilfolge a_0, a_4 auf.

Zeile 2-3 Alg. 4 Vergleiche für $i = 0$.

$i = 0$ Füge Vergleicher(a_0, a_4) zu Z hinzu.

Zeile 5 Alg. 4 Rufe PairwiseSort für die gerade Teilfolge a_0 auf. Return $n = 1$.

Zeile 6 Alg. 4 Rufe PairwiseSort für die ungerade Teilfolge a_4 auf. Return $n = 1$.

Zeile 7 Alg. 4 u.s.w.

Abbildung 2.3 zeigt in Rot die für $i = 0, 2, 4, 6$, in Gelb die für $i = 0, 4$ und in Blau die für $i = 0$ erzeugten Vergleicher.

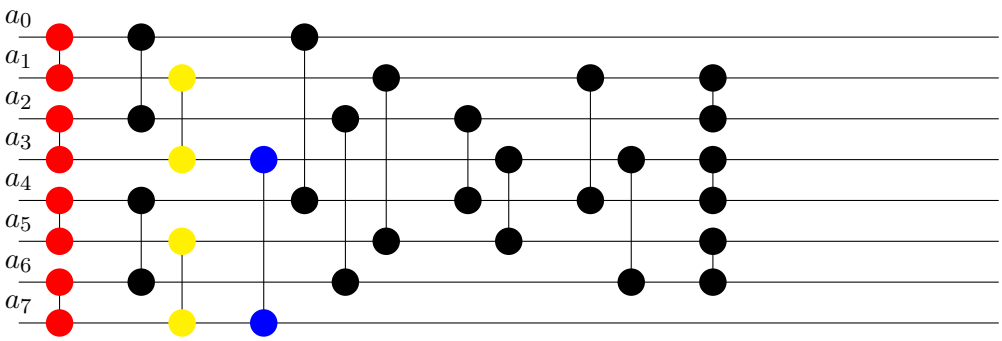

Abbildung 2.3: Pairwise Sortiernetzwerk. In Rot die für $i = 0, 2, 4, 6$, in Gelb die für $i = 0, 4$ und in Blau die für $i = 0$ erzeugten Vergleicher.

2.4 Shells Methode

Shellsort ist ein Sortierverfahren, das versucht, durch eine Grobsortierung den Aufwand eines Insertionssort zu minimieren. Es ist datenabhängig, das bedeutet, dass die Sortierung bei verschiedenen Daten anders verlaufen kann. Damit das Verfahren als Sortiernetzwerk implementiert werden kann, muss der datenabhängige Insertionsort durch ein Bubblesort ersetzt werden. Die Grobsortierung wird mithilfe von Feldbreiten (siehe Abschnitt 2.4) durchgeführt. Es wird die größte Feldbreite genommen und dementsprechend viele Spalten gebildet. Die Spalten werden dann einzeln sortiert und die Feldbreite nimmt mit jedem Schritt ab. Durch die Vorsortierung muss der Bubblesort allerdings nicht auf das gesamte Netzwerk angewendet werden. Die Effizienz von Shellsort hängt von den gewählten Feldbreiten ab. Die hier benutzte Feldbreite ist die 2^p3^q Sequenz (siehe Tabelle 2.1). Damit besitzt es dieselbe Komplexität wie Batcher's Odd Even, Batcher's Bitonic und das PairwiseSort Verfahren. Die Anzahl der Vergleicher ist aber höher als bei dem Odd Even Verfahren oder das PairwiseSort Verfahren. [9, S. 314-317]

Feldbreiten

Die Feldbreite beeinflusst die Grobsortierung und die asymptotische Komplexität von Shellsort. Bei der Grobsortierung startet man mit der größten Feldbreite h in einer Feldbreitensequenz, die kleiner als die Anzahl der zu sortierenden Elemente ist. Dann werden die zu sortierenden Daten spaltenweise aufgeschrieben und spaltenweise sortiert. Die zu sortierenden Daten sind nun h-sortiert. Im nächsten Schritt wird die Feldbreite verringert und die vorherigen Spalten teilen sich in neue Spalten auf. Dabei bilden die zu sortierenden Elemente zeilenweise von links nach rechts neue Spalten. Jede Feldbreitensequenz, die eine 1 beinhaltet, ist eine gültige Sequenz, weil hierfür ein Insertionsort stattfindet. Das bedeutet, die Elemente werden solange verglichen, bis sie an der richtigen Stelle sind. Durch die gewählte Feldbreite können Grenzen gesetzt werden, wie oft maximal noch verglichen werden muss, bis das Element an der richtigen Stelle ist. Für verschiedene Feldbreiten wurden verschiedene Obergrenzen der Komplexität bewiesen (siehe Tabelle 2.1).

Beispiel

Das Beispiel zeigt wie die Zahlenfolge $15, 0, 1, 14, 2, 13, 3, 12, 4, 11, 5$, die mit der ursprünglichen $2^k = 16, 8, 4, 2, 1$ Feldbreitensequenz sortiert wird. Die zu sortierenden Daten sind erst 8, 4, 2 und letztendlich 1 sortiert. Eine Sequenz die 8-sortiert ist, hat 8 Spalten die in sich sortiert sind (siehe nachfolgende Matrizen). Die $4, 2, 1$ Sortierung sind dazu äquivalent.

$$\begin{matrix} 15 & 0 & 1 & 14 & 2 & 13 & 3 & 12 \\ 4 & 11 & 5 & 10 & & & & \end{matrix} \rightarrow \begin{matrix} 4 & 0 & 1 & 10 & 2 & 13 & 3 & 12 \\ 15 & 11 & 5 & 14 & & & & \end{matrix}$$

8 Sortiert.

$$\begin{array}{cccc} 4 & 0 & 1 & 10 \\ 2 & 13 & 3 & 12 \\ 15 & 11 & 5 & 14 \end{array} \rightarrow \begin{array}{cccc} 2 & 0 & 1 & 10 \\ 4 & 11 & 3 & 12 \\ 15 & 13 & 5 & 14 \end{array}$$

4 Sortiert.

$$\begin{array}{cc} 2 & 0 \\ 1 & 10 \\ 4 & 11 \\ 3 & 12 \\ 15 & 13 \\ 5 & 14 \end{array} \rightarrow \begin{array}{cc} 1 & 0 \\ 5 & 10 \\ 3 & 11 \\ 4 & 12 \\ 5 & 13 \\ 15 & 14 \end{array}$$

2 Sortiert.

Verfügbare Feldbreiten

Die Wahl der richtigen Feldbreite stellt heute immer noch ein Problem dar. Tabelle 2.1 zeigt die am meisten genutzten Feldbreiten im Vergleich. [15]

Feldbreiten			
Sequenz	Zahlenfolge	Komplexität	Author
$\lfloor N/2^k \rfloor$	$\lfloor N/2 \rfloor, \lfloor N/4 \rfloor \ldots, 1$	$O(N^2)$	Shell [14, S.30-32]
$2\lfloor N/2^{k+1} \rfloor + 1$	$2\lfloor N/2^{k+1} \rfloor + 1, \ldots, 1$	$O(N^{3/2})$	Frank und Lazarus [3, S.20-22]
$2^k - 1$	$1, 3, 7, 15, \ldots$	$O(N^{3/2})$	Hibbard [6, S.206-213]
$2^p 3^q p, q \in \mathbb{N}_0$	$1, 2, 3, 4, 6, 8, \ldots$	$O(n \log(n)^2)$	Pratt [12]
$4^k + 3 \cdot 2^{k-1} + 1$	$1, 8, 23, 77, 281, \ldots$	$O(N^{4/3})$	Sedgewick [13]

Tabelle 2.1: Feldbreiten

Algorithmus

Sei die Größe des zu erstellenden Netzwerks gegeben als Folge $A(a_1, \ldots, a_n)$, wobei jedes Folgemitglied eine Leitung des Sortiernetzwerkes darstellt. Die Feldbreitensequenz sei mit $h = (2^p 3^q) \ldots, 6, 4, 3, 2, 1$ in absteigender Reihenfolge gegeben. Die Funktion *length* gibt die Anzahl der Elemente in der Feldbreitensequenz bzw. der Folge A zurück. Die Variable Z beschreibt den Stack, in dem die Vergleicher eines Netzwerkes gespeichert werden (siehe Algorithmus 6).

Beispiel

Das Beispiel soll die Funktionsweise des Algorithmus schrittweise erläutern, wobei ein Sortiernetzwerk der Größe 8 erstellt werden soll. Die Folgeglieder von A entsprechen jeweils einer Leitung des Sortiernetzwerks.

Algorithmus 6 : Shellsort

input : Folge A
output : Position der Vergleicher als Stack Z

```
1  t ← 0;
2  j ← 0;
3  for k ← 0; k < h.length; k ← k + 1 do
4  |  t ← h[k];
5  |  for i ← t; i < A.length; ← i + 1 do
6  |  |  j ← i;
7  |  |  x ← 0;
8  |  |  while j ≥ h ∧ x < (h[k − 1] − 1) · (h[k − 2] − 1) − 1 do
9  |  |  |  Füge Vergleicher(j, j − h) zu Z hinzu ;
10 |  |  |  x ← x + h;
11 |  |  end
12 |  end
13 end
```

Zeile 1-2 Alg. 6 $t \leftarrow 0; j \leftarrow 0$.

Zeile 4-5 Alg. 6 $t \leftarrow h[0] = 8; break$.

Zeile 4 Alg. 6 $t \leftarrow h[1] = 6$.

Zeile 5 Alg. 6 $i \leftarrow t = 6 \wedge 0 < 8$.

$j = 7$ Füge Vergleicher(a_1, a_7) zu Z hinzu.

$j = 8$ Füge Vergleicher(a_2, a_8) zu Z hinzu.

Zeile 4 Alg. 6 $t \leftarrow h[2] = 4$.

Zeile 5 Alg. 6 $i \leftarrow t = 4 \wedge 0 < 8$.

$j = 5$ Füge Vergleicher(a_1, a_5) zu Z hinzu.

$j = 6$ Füge Vergleicher(a_2, a_6) zu Z hinzu.

$j = 7$ Füge Vergleicher(a_3, a_7) zu Z hinzu.

$j = 8$ Füge Vergleicher(a_4, a_8) zu Z hinzu.

Zeile 4 Alg. 6 u.s.w.

Abbildung 2.4 zeigt in Rot die für $j = 7, 8$ und in Gelb die für $j = 5, 6, 7, 8$ erzeugten Vergleicher.

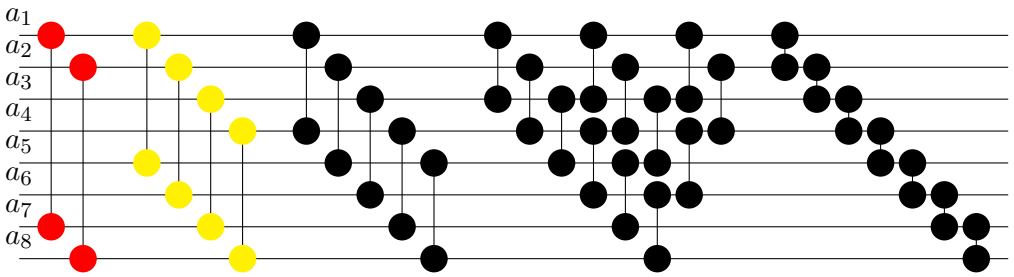

Abbildung 2.4: Shellsort Netzwerk. In Rot die in Schritt 4 und und Gelb die in Schritt 6 erzeugten Vergleicher.

2.5 Größenvergleich

In diesem Abschnitt soll die Größe der verschiedenen Netzwerke miteinander verglichen werden. Dabei soll ein Netzwerk der Größe 2^k konstruiert und deren Vergleicher gezählt werden. Die Werte werden aus einer modifizierten Version des Programms entnommen. Netzwerke der Größe 2^0 und 2^1 sind immer gleich und in der Liste nicht mit aufgenommen.

Batcher's Odd Even & Pairwise-Sorting Algorithmus

Die folgenden Werte sind für das Batcher's Odd Even und den Pairwise-Sorting Algorithmus entstanden. Beide Verfahren benötigen die gleiche Anzahl von Vergleicher. Das Odd Even sowie Pairwise-Sorting Verfahren wird in Abbildung 2.5 in Rot dargestellt. Der Verlauf der Kurve zeigt, dass die durchschnittliche Anzahl der Vergleiche pro Vergleicher gering zur Größe des Sortiernetzwerkes zunimmt. Durch diese geringe Anzahl stellen diese beiden Verfahren die beste Möglichkeit in der Praxis dar.

Größe des Netzwerks	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Benötige Vergleicher	5	19	63	191	543	1471	3839	9727	24063
Größe des Netzwerks	2^{11}		2^{12}		2^{13}		2^{14}	2^{15}	
Benötige Vergleicher	58367		139263		327679		761855	1753087	
Größe des Netzwerks	2^{16}		2^{17}		2^{18}		2^{19}	2^{20}	
Benötige Vergleicher	3997695		9043967		20316159		45350911	100663295	

Tabelle 2.2: Größe Batcher's Odd Even & Pairwise-Sort

Bitonic Algorithmus

Die folgenden Werte sind für den Batcher's Bitonic Verfahren entstanden. Das Batcher Bitonic Verfahren wird in Abbildung 2.5 in Blau dargestellt. Der Verlauf der Kurve zeigt, dass die Anzahl der Vergleicher ähnlich dem der Batcher's Odd Even und PairwiseSort Verfahren sind.

Größe des Netzwerks	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Benötige Vergleicher	5	24	80	240	672	1792	4608	11520	28160
Größe des Netzwerks	2^{11}		2^{12}		2^{13}		2^{14}		2^{15}
Benötige Vergleicher	67584		159744		372736		860160		1966080
Größe des Netzwerks	2^{16}		2^{17}		2^{18}		2^{19}		2^{20}
Benötige Vergleicher	4456448		10027008		22413312		49807360		110100480

Tabelle 2.3: Größe Batcher's Bitonic Verfahren

Shellsort

Die folgenden Werte sind für das Shellsort Verfahren entstanden. Das Shellsort Verfahren wird in Abbildung 2.5 in Grün dargestellt. Der Verlauf der Kurve zeigt, dass der Anstieg der Vergleicher ebenfalls so stark wie bei dem BubbleSort Verfahren ist.

Größe des Netzwerks	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Benötige Vergleicher	6	32	143	573	2153	7510	25178	82283	262960
Größe des Netzwerks	2^{11}		2^{12}		2^{13}		2^{14}		2^{15}
Benötige Vergleicher	818373		2525444		7694729		23129189		69050176

Tabelle 2.4: Größe Shellsort Verfahren

Ergebnis

Die in der Praxis verwendeten Verfahren Batcher Odd Even, Batcher Bitonic und PairwiseSort zeigen, dass sie bei relevanten Netzwerkgrößen von $n \leq 2^{10}$ Sortiernetzwerke wesentlich besser erstellen können als andere Verfahren. Abbildung 2.5 zeigt, dass das Verfahren Batcher Odd Even und PairwiseSort besser sind, als das Batcher Bitonic Verfahren, weil die benötigten Anzahl der Vergleicher geringer ist.

Das Shellsort Verfahren ist durch seinen datenabhängigen Teil gut in Software zu implementieren, hat aber bei der Hardwareimplementation Probleme. Der datenabhängige Insertionsort muss durch einen datenunabhänigen Bubblesort ersetzt werden. Dieser unabhängige Teil ist dafür verantwortlich, dass Shellsort wesentlich mehr Vergleiche braucht als die anderen in der Praxis verwendeten Verfahren.

Abbildung 2.5 zeigt, dass Shellsort am Anfang mehr Verleiche als Bubblesort braucht, aber bei größeren Netzwerk die Anzahl abnimmt.

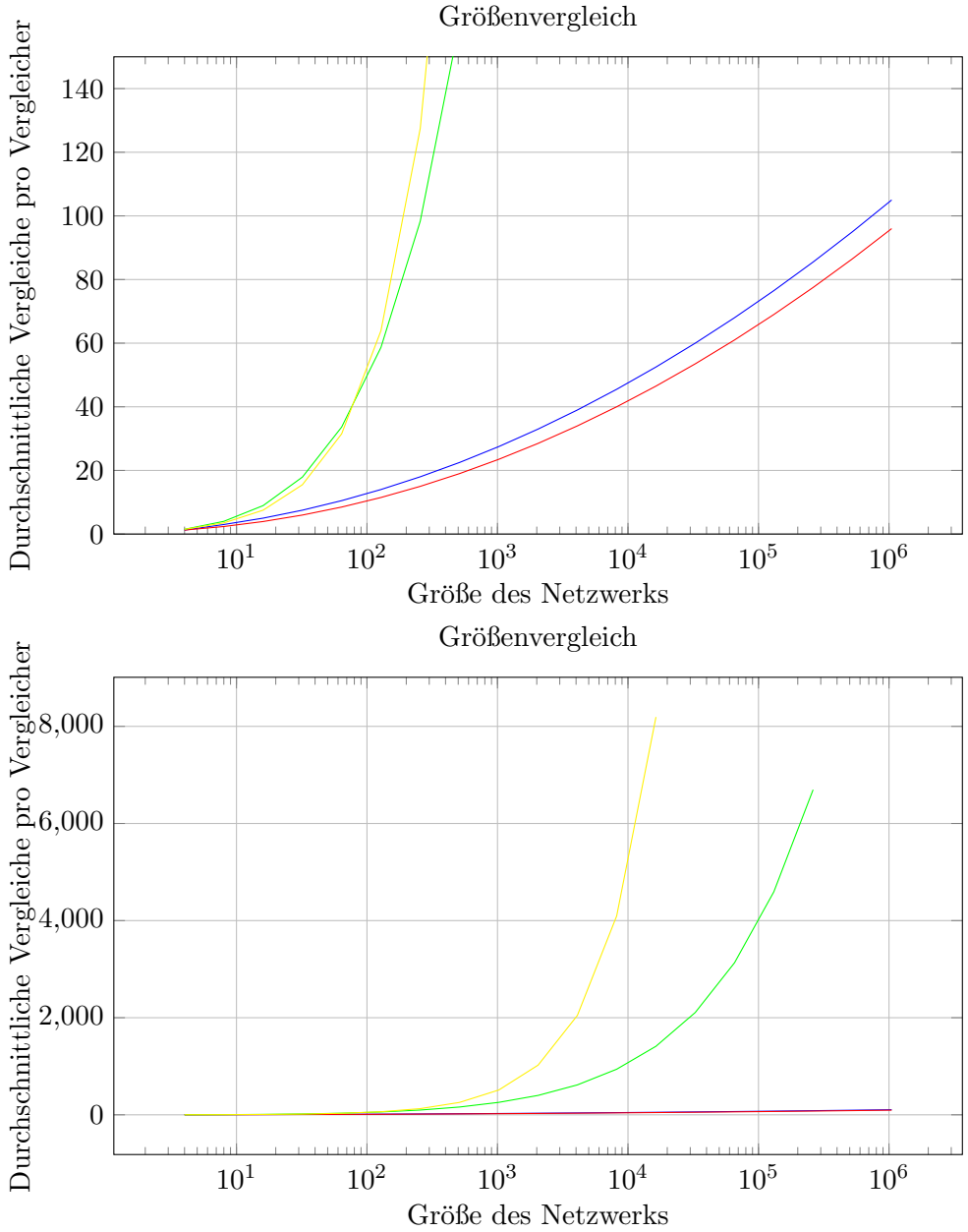

Abbildung 2.5: Größenvergleich: In Gelb Bubblesort, Grün Shellsort, Blau BitonicSort und Rot Odd Even sowie PairwiseSort.

3 Implementation

In diesem Kapitel soll die Implementation des Programms genauer erläutert werden. Des weiteren wird auf die einzelnen Teilkonstrukte eingegangen, sowie die verwendeten Pattern, welche für die Entwicklung der Software genutzt werden, genauer erklärt.

3.1 Allgemeiner Aufbau

Der allgemeine Aufbau des Programms beschreibt die Klassen und deren Abhängigkeiten sowie den Programmablauf.

Klassen

Dieser Abschnitt beschreibt die Klassen des Programms. Es werden die Klassen innerhalb des sortings_network Namespaces beschrieben. Ein *Namespace* ist ein Bereich, indem Klassen ihre Gültigkeit haben. Innerhalb diesen Bereichs muss jeder Objektname eindeutig sein.

main Dies ist der Einstiegspunkt des Programms. Er beinhaltet die Klassen für die GUI und das Controlling um einen neuen Datensätze zu erzeugen. Neue Datensätze werden durch den Aufruf von *generate_data* angelegt.

generate_data In dieser Komponente wird ein neuer zufälliger Datensatz durch die Eingabe von Werten angelegt. Es beinhaltet die Klassen für die GUI und das Controlling dieser Oberfläche. Die erzeugten Daten werden an die Klasse *data* weitergegeben.

data Diese Komponente enthält die Klassen für die GUI und das Controlling zur Steuerung der Datensätze. Hier können die Sortiernetzwerke erzeugt und gestartet werden. Außerdem beinhaltet es die Funktion zum gleichzeitigen Ausführen mehrerer Datensätze und das Controlling von der Klasse *image_data.*

image_data In dieser Komponente sind die Klassen für die GUI und die Methoden für die Steuerung des Balken- und Netzwerkdiagramms enthalten.

coredata Dies ist die Klasse für das Datenmodell eines Datensatzes. Es beinhaltet die zu sortierenden Daten, Informationen über diese und die Vergleicher, wie diese Daten zu sortieren sind.

sortingalgorithm Die abstrakte Klasse, von der jede Klasse, die ein Sortieralgorithmus darstellt, erben muss. In ihr sind die Methoden vorgeschrieben, welche jeder Sortieralgorithmus implementiert haben muss.

sortingfactory Dies ist die Klasse, welche für die Klasse *data*, Sortieralgorithmen zurück gibt. Für diese Klasse wurde das Factory Design-pattern verwendet.

variables Die Klasse die Variablen für Einstellungen und Debug-Optionen beinhaltet und eine Möglichkeit beinhaltet, Informationen zu speichern, die allen Klassen zur Verfügung gestellt werden können.

sortingstep Die Klasse, die einen Vergleicher modelliert.

Abhängigkeiten

Die Abhängigkeiten der Klassen zueinander kann durch ein Abhängigkeitsgraphen modelliert werden (siehe Abbildung 3.1). Eine Abhängigkeit ist der Aufruf einer Klasse von einer anderen oder derselben Klasse. Der Einstiegspunkt ist die Klasse *Program*, welches die Klasse *main* aufruft, welches der Startpunkt des Programms ist. Durch diese Klasse können alle weiteren Klassen mit den entsprechenden Aufrufen angesprochen werden. Der Namespace *sorting_algorithm* enthält die Klassen, welche für die Erzeugung der Sortiernetzwerke verantwortlich sind. Die Aufrufe zwischen den beiden Namespaces ist vereinfacht dargestellt, indem nur Pfeile zwischen den beiden Namespaces *sorting_network* und *sorting_algorithmen* gezeichnet sind. Der Aufruf von *Externals* geschieht nur in einer Richtung, weil dort systemrelevante Bibliotheken zur Verfügung stehen, die z.B. für die Zeichnung eines Fenster auf Betriebssystemebene zuständig sind.

Programmablauf

Die Erzeugung der Objekte innerhalb des Programms kann wie in Abbildung 3.2 beschrieben werden. Das Diagramm zeigt die beteiligten Klassen und wann sie erzeugt werden. Es soll eine vereinfachte Darstellung des Programms sein, da nicht alle Klassen behandelt werden. Die Klasse *main* erzeugt ein Objekt *generate_data* in dem der Benutzer Einstellungen über seinen zufälligen Datensatz macht. Wenn der zufällige Datensatz (*coredata*) erzeugt wurde, wird auch das Objekt *data* erstellt. In *data* können alle Einstellungen zu dem Datensatz gewählt werden. Als Einstellungen zählen beispielsweise: Welche Sortieralgorithmen angewendet werden sollen, welche Grafikoptionen gesetzt werden und wie schnell die Sortierung erfolgen soll (Zeit zwischen jedem Schritt). Wird die Option “Grafik anzeigen“gesetzt, wird ein Objekt vom *image_data* erzeugt, welches die grafische Darstellung der Sortiernetzwerke beinhaltet. Werden mehrere Sortieralgorithmen in den Einstellungen ausgewählt, so werden auch mehrere Objekte von dem Typ *image_data* erzeugt.

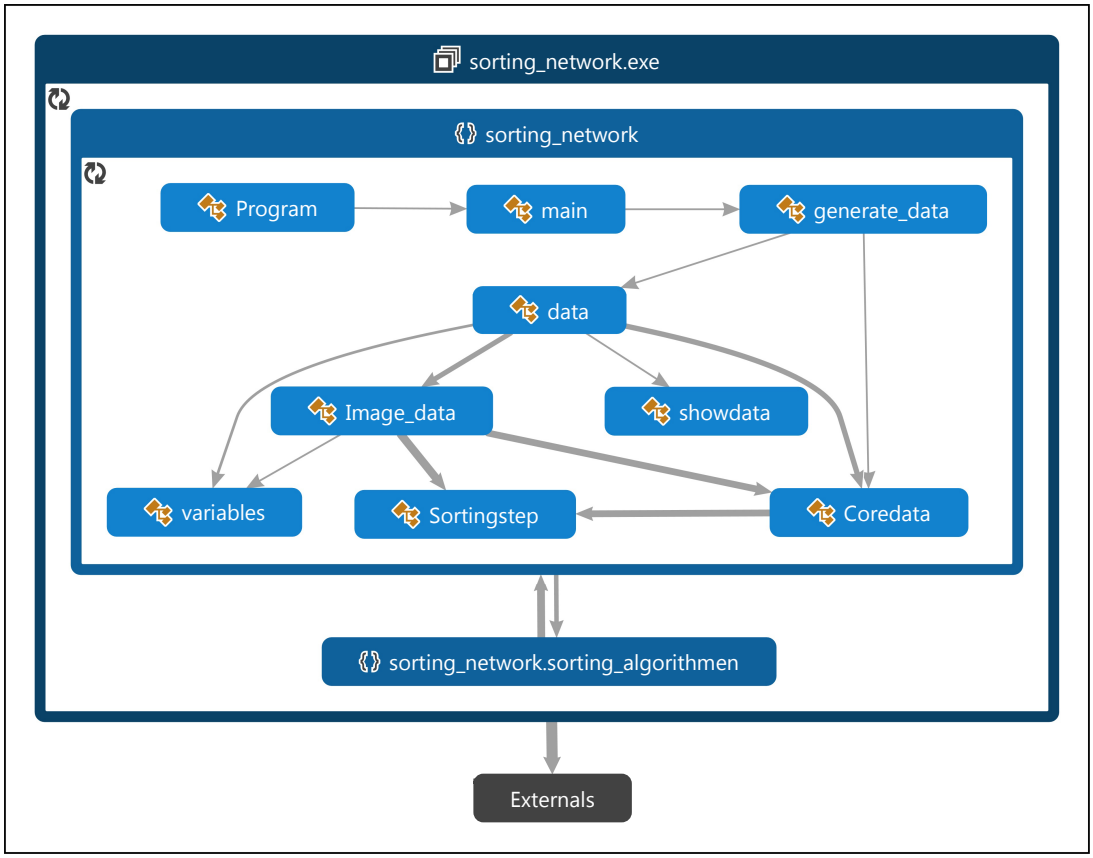

Abbildung 3.1: Abhängigkeitsgraph der Klassen

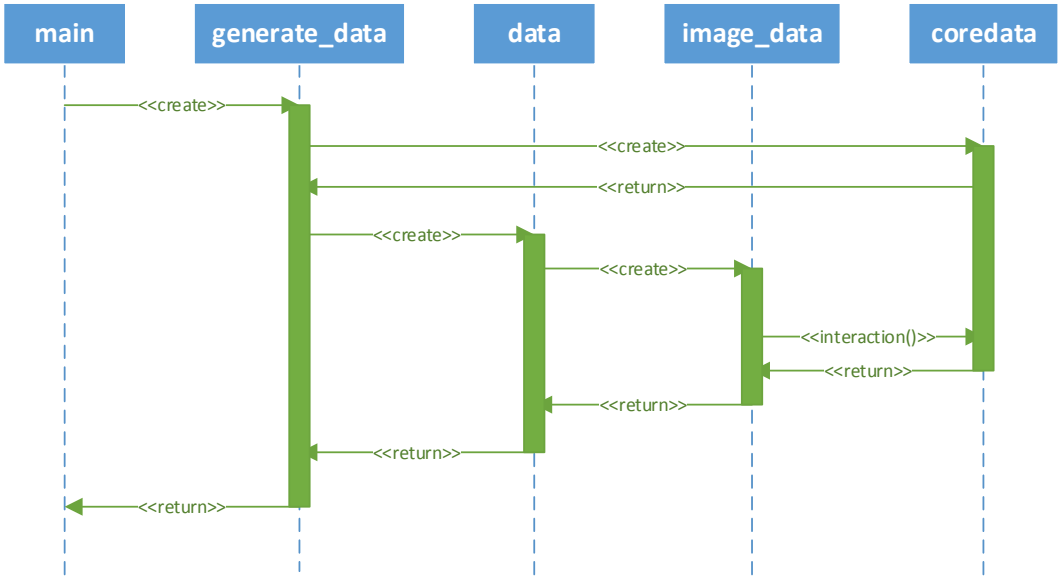

Abbildung 3.2: Sequenzdiagramm

Entwurfsmuster

Die Architektur des Programms wird durch den Einsatz von Entwurfsmustern (Pattern) vereinfacht. Ein Entwurfsmuster ist ein bewährtes Lösungsverfahren in der Softwareentwicklung für immer wiederkehrende Probleme. Durch die Anwendung eines Entwurfsmusters wird die Wartbarkeit und die zukünftige Weiterentwicklung des Programms gefördert. In dem Programm wurde das Model-View-Vontrol (MVC) und das Factory Entwurfsmuster eingesetzt.

MVC-Pattern

Das MVC-Pattern ist die Strukturierung der Software in die drei Einheiten Modell, Präsentation und Steuerung. Das Modell enthält die darzustellenden Daten und Informationen und wird deswegen auch oft Datenmodell genannt. Dies wird durch die Klasse *coredata* bereitgestellt. Die Steuerung verwaltet die Präsentationsschicht und das Modell. Sie nimmt Benutzereingaben entgegen und bewirkt Änderungen auf das Datenmodell. Diese Änderungen werden von der Klasse *data* ausgeführt. Die Präsentationsschicht stellt die Daten dem Benutzer dar. Sie nimmt Eingaben an und leitet diese an die Steuerung weiter, da sie für die Weiterverarbeitung nicht zuständig ist. Die Weiterleitung und Annahme der Benutzereingabe erfolgt durch die Klasse *image_data*. Das MVC-Pattern wird nur für die Anzeige der Diagramme umgesetzt. Bei den verschiedenen anderen Darstellungen teilen sich die Steuerung und die Präsentationsschicht eine Klasse, beispielsweise das Layout zum Erzeugen von Daten. [8]

Factory-Pattern

Das Factory-Pattern ist ein Entwurfsmuster, mit dem Objekte durch den Aufruf einer Methode anstatt durch den Konstruktor angelegt werden. Die Sortieralgorithmen werden durch dieses Verfahren erzeugt, indem sie eine zentrale Methode in der Klasse *SortingFactory* haben, die alle Sortieralgorithmen anlegen kann. Eine einheitliche Klasse zum Erzeugen der Sortieralgorithmen erleichtert die Wartung des Programms und die Implementation weiterer Sortieralgorithmen. [4]

3.2 Parallelisierung von Vergleichern

Ein Algorithmus, der ein Sortiernetzwerk erzeugt, erstellt eine Liste von Vergleichern, die hintereinander ausgeführt werden müssen. Das Programm kann automatisch die optimale Position jedes Vergleichers in einem Netzwerk ermitteln und sie dann so gruppieren, dass möglichst wenige Ebenen vorhanden sind. Die Anzahl der Ebenen hat Einfluss auf die Geschwindigkeit der Sortierung des Sortiernetzwerkes, wenn dieses parallel ausgeführt wird. Wenn ein Sortiernetzwerk parallel ausgeführt wird, kann pro Schritt eine Ebene ausgeführt werden, weil es innerhalb einer Ebene keine Abhängigkeiten von Folgezuständen auf einer Leitung gibt. Für die Anwendung in der Praxis ist dies sehr interessant, weil z.B. neue Grafikkarten genug Rechenkerne besitzen um Netzwerke der

Größe 2^{12} parallel ausführen zu können. Voraussetzung dafür ist, das berücksichtigt wird, dass jeder Grafikkartenkern einen Vergleich übernimmt und es keine Probleme mit der Kommunikation zwischen diesen Kernen gibt. Sortiernetzwerke bilden die Grundlage, wenn mit einer Grafikkarte effektiv sortiert werden soll. [16]

Die Ermittlung der optimalen Position der Vergleicher erfolgt über den Algorithmus 7. Der Algorithmus arbeitet korrekt, wenn folgende Bedingung erfüllt ist: Für jeden Vergleicher $(a_0, \ldots, a_{n-1})$, der von dem Sortieralgorithmus des Sortiernetzwerkes erzeugt wird gilt, a_{i-1} muss vor a_i ausgeführt werden.

Algorithmus

Der Algorithmus arbeitet nach dem *Drop Down* Prinzip. Er bekommt einen Vergleicher und versucht diesen in einer entsprechende Ebene einzufügen. Der Vergleicher wird in die kleinste Ebene eingeordnet, bei welcher alle darüber liegenden Ebenen dieser Leitung noch frei sind, dass bedeutet, er wird an der Position angelegt, wo nachfolgend keine Abhängigkeiten mehr auf den Leitungen vorhanden sind. Eine *Ebene* ist die Anzahl der Leitungen eines Sortiernetzwerkes, also dessen Größe.

Algorithmus 7 : Optimale Vergleicherposition

```
   input  : Liste der Vergleicher als Array V_new
   output : Liste von Ebenen mit Vergleicher V_sol
 1 maxlayer ← 0;
 2 foreach Vergleicher i in V_new do
 3 |   t ← maxlayer;
 4 |   if V_sol[t][i.wireA] = null ∧ V_sol[t][i.wireB] = null then
 5 |   |   t ← t − 1;
 6 |   else if V_sol[t][i.wireA] ≠ null ∨ V_sol[t][i.wireB] ≠ null then
 7 |   |   if t ≠ 0 then
 8 |   |   |   V_sol[t + 1][i.wireA] ← i;
 9 |   |   |   V_sol[t + 1][i.wireB] ← i;
10 |   |   end
11 |   |   else if t = 0 then
12 |   |   |   V_sol[t][i.wireA] ← i;
13 |   |   |   V_sol[t][i.wireB] ← i;
14 |   |   end
15 |   |   if t ≥ maxlayer then
16 |   |   |   maxlayer ← t;
17 |   |   end
18 |   end
19 end
```

Gegeben sei die Liste der Vergleicher als V_{new} in einem Array. Die Lösung wird als V_{sol} ausgegeben, welches ein Array von einer Ebene ist. Ein Vergleicher hat die Funktio-

nen *wireA* und *wireB*, welche die Leitungen ausgeben, die verglichen werden sollen. Die Allokation von Speicher für eine neue Ebene ist in diesem Algorithmus nicht abgebildet und wird als automatisch gegeben vorausgesetzt.

3.3 Farbverlauf

Damit eine bessere Erkennbarkeit der einzelnen zu sortierenden Werte erfolgt, wird ein Farbverlauf mithilfe des HSV-Farbmodell (hue-saturation-value) eingesetzt. So kann der Benutzer bei der Ausführung einer Sortierung anhand der Farbe abschätzen, wo gerade welcher Wert der unsortierten Folge auf der Leitung anliegt (siehe farblichen Verlauf bei Abbildung 3.4 und 3.6). Nach Abschluss der Sortierung kann der Benutzer sich durch die farbliche Darstellung ebenfalls von der richtigen Sortierung der Daten überzeugen.

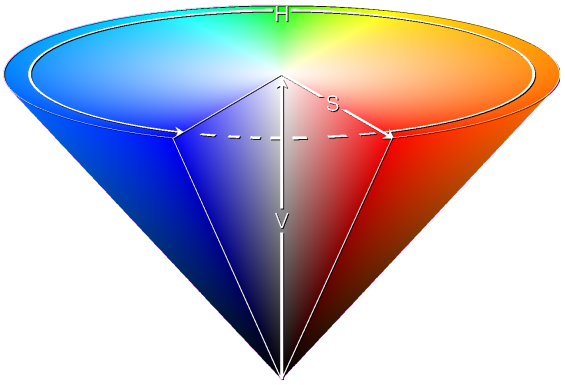

Abbildung 3.3: HSV Kegel [2]

HSV-Farbmodell

Der HSV-Farbraum ist die Grundlage verschiedener Farbmodelle. Eine Farbe wird in diesem Raum mit den Parametern $H \in [0, 360]$ (Farbwinkel, hue), $S \in [0, 1]$ (Sättigung, saturation) und $V \in [0, 1]$ (Helligkeitswert, value) beschrieben. Diese drei Werte bilden einen kegelförmigen Bereich auf welchem die Farben liegen (siehe Abbildung 3.3). Der Farbwinkel H gibt den Farbton an, wobei etwa 0° Rot, 120° Grün und 240° Blau ist. Durch S wird der Sättigungswert beschrieben, bei dem 0% Neutralgrau und 100% die reine Farbe ist. Der Wert V beschreibt die Helligkeit der Farbe, wobei 0% keine Helligkeit und 100% die volle Helligkeit bedeutet.

Implementierung

Eine Farbe auf einem Bildschirm wird mithilfe des RGB-Modells dargestellt. Da das RGB-Modell keine gute Möglichkeit zur Darstellung des Farbverlaufs bietet, wird zunächst das HSV-Modell benutzt, welches anschließend ins RGB-Modell umgewandelt wird. Der Farbverlauf findet zwischen den beiden Farbwinkeln H_1 und H_2 statt, die zuvor festgelegt werden müssen. Diese Winkel sind der Start und Endpunkt, zwischen denen der Farbverlauf stattfindet.

Die Umwandlung von HSV nach RGB erfolgt mit dem Algorithmus 8. Es gibt konstante und dynamische Komponenten in diesem Algorithmus. Die konstanten Elemente sind bereits vor der Programmausführung bekannt, während die dynamischen erst zur

Laufzeit des Programms bestimmt werden können. Konstant gegeben sind die beiden Farbwinkel H_1 und H_2 zwischen denen der Verlauf stattfindet, die Sättigung S und die Helligkeit V des Farbverlaufs. Dynamisch gegeben sind der Minimalwert k_{min} und Maximalwert k_{max}, der zu sortierenden Elemente sowie der Wert k, für die die Farbe bestimmt werden soll. Der Wert k_{win} stellt den Farbwinkel für den Wert dar, wobei sich k_{min} proportional zu H_1, k_{max} sich proportional zu H_2 und k_{win} sich proportional zu k verhält.

Algorithmus 8 : HSV-RGB Umwandlung [1, S. 259-267]

input : k_{win}
$h_i \leftarrow \lfloor \frac{k_{win}}{60°} \rfloor$;
$f \leftarrow (\frac{k_{win}}{60°} - h_i)$;
$p \leftarrow V \cdot (1 - S)$;
$q \leftarrow V \cdot (1 - S \cdot f)$;
$t \leftarrow V \cdot (1 - S \cdot (1 - f))$;
switch h_i **do**
  case $h_i = 0$
    $R \leftarrow V, G \leftarrow t, B \leftarrow p$;
  end
  case $h_i = 1$
    $R \leftarrow q, G \leftarrow V, B \leftarrow p$;
  end
  case $h_i = 2$
    $R \leftarrow p, G \leftarrow V, B \leftarrow t$;
  end
  case $h_i = 3$
    $R \leftarrow p, G \leftarrow q, B \leftarrow V$;
  end
  case $h_i = 4$
    $R \leftarrow t, G \leftarrow p, B \leftarrow V$;
  end
  case $h_i = 5$
    $R \leftarrow V, G \leftarrow p, B \leftarrow q$;
  end
endsw
output : Farbe in RGB

Beispiel

Ein Beispiel soll zeigen, wie die Umwandlung funktioniert. Gegeben seien: $H_1 = 0°$, $H_2 = 100°$, $S = 1$, $V = 1$, $k_{min} = 0$, $k_{max} = 20$ und $k = 9$.

Input bestimmen k_{win} bestimmen : $k_{win} = 45°$.

Zeile 1-5 Alg. 8 Hilfsvariablen bestimmen : $h_i = 0$, $f = \frac{45}{60}$, $p = 0$, $q = 0.25$, $q = 0.25$ und $t = 0.75$.

Zeile 8 Alg. 8 Farbe bestimmen : $h_i = 0$, RGB $= (1, 0.75, 0.25)$.

Der RGB-Wert $= (1, 0.75, 0.25)$ entspricht einem dunklen Gelb.

3.4 Balkendiagramm

Das Programm kann die Daten visuell als Balkendiagramm darstellen (siehe Abbildung 3.4). Die Position der Balken im Balkendiagramm ist äquivalent zu den Leitungen im Sortiernetzwerk. Der Balken links entspricht der Leitung K_0 und rechts K_{n-1} im Balkendiagramm. Zur Laufzeit kann das Programm die Balken durch Vergleiche vertauschen. Hierfür werden die für den Vergleicher benötigten Balken markiert und verglichen (siehe Abbildung 3.5). Sortiert das Netzwerk aufsteigend und hat der Balken links einen höheren Wert als der rechts, werden beide vertauscht. Der Wert eines Balken wird durch seine Höhe bzw. durch seine Farbe dargestellt (siehe Abschnitt 3.3).

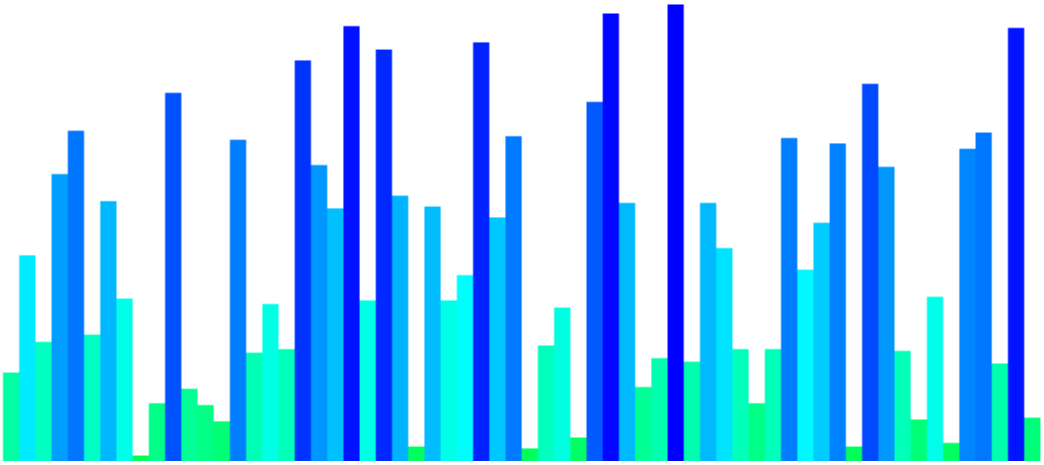

Abbildung 3.4: Unsortiertes Balkendiagramm.

Implementation

Das Balkendiagramm wird als eigenes komplettes Bild im Programm erzeugt, dass dann in die Oberfläche eingebunden wird. Es wird durch die Veränderung der Größe des Fenster automatisch in seiner sichtbaren Größe mit skaliert, das bedeutet, es passt sich automatisch der aktuellen Größe des Fensters an. Das Programm rechnet zuerst die Breite eines jeden Balken durch die Anzahl der Leitungen im Sortiernetzwerk aus, indem die Gesamtbreite des Bildes durch die Anzahl der Leitungen geteilt wird. Jeder Balken hat die gleiche Breite und ist mindestens ein Pixel breit. Pixel, die nicht benötigt werden,

falls die Anzahl der Balken kein Teiler von der Breite des Bildes ist, werden als weißer Rand links und rechts vom Diagramm dargestellt. Werden zwei Werte miteinander verglichen, werden sie Rot markiert. Die rote Markierung kann auf dem aktuellen Bild über dem farblichen Balken gezeichnet werden, weil die Balkenhöhe bei der Markierung gleich bleibt. Werden die Balken aufgrund des Vergleiches getauscht, werden zuerst die markierten Balken weiß und dann an den vertauschten Stellen farblich neu gezeichnet. Wird dies nicht getan, kann es vorkommen, das rote Markierungen bleiben und nicht weiter überzeichnet werden, also würden am Ende der Sortierung rote Balken im Diagramm bleiben. Dies liegt da dran, dass die rote Balkenmarkierung größer ist als alle zukünftigen Balken, die an dieser Stelle neu gezeichnet werden. Dadurch besteht keine Möglichkeit mehr, das Rot neu zu überzeichnen. Effizienz-technisch hat dieses Verfahren Vorteile, weil nur auf Teile des Bildes gezeichnet wird, die der Algorithmus benötigt. Das Bild wird somit nicht bei jeder Veränderung komplett neu berechnet.

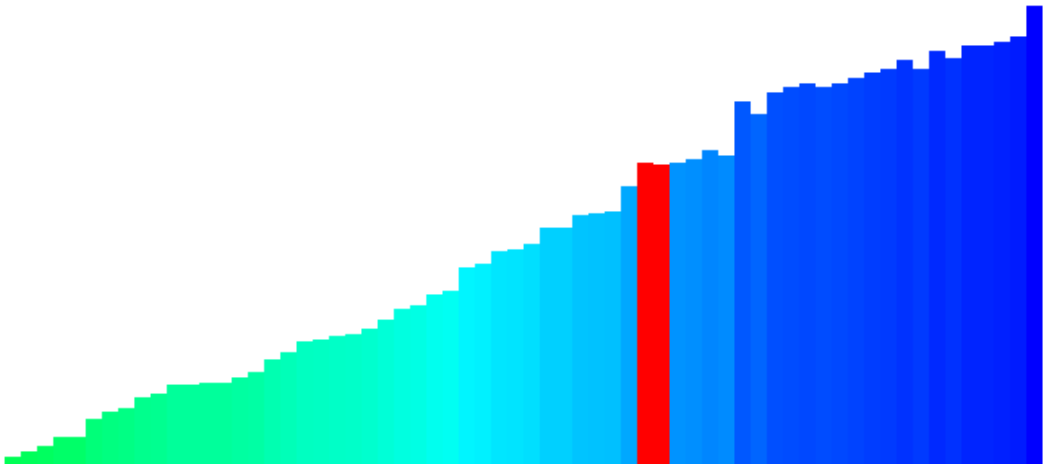

Abbildung 3.5: Fast sortiertes Balkendiagramm mit Markierung.

3.5 Netzwerkdiagramm

Das Programm kann die Daten visuell als Netzwerkdiagramm darstellen (siehe Abbildung 3.6 und 3.7). Die horizontal verlaufenden Linien stellen die Leitungen K_1 bis K_{n-1} dar, wobei K_1 die oberste horizontale und K_{n-1} die unterste horizontale Linie ist. Der horizontale Verlauf einer Linie kann durch einen Vergleicher, der als vertikale Linie dargestellt ist, unterbrochen werden. So lässt sich eine horizontale Linie in mehrere Abschnitte unterteilen. Jeder dieser Abschnitte kann eine Farbe annehmen, die äquivalent zum Wert ist, der auf der Leitung zu dem gegebenen Zeitpunkt anliegt (siehe Abschnitt 3.3). Die Farbe einer horizontalen Linie wird immer bis zum nächsten Vergleicher gefärbt, der an

dieser Leitung anliegt. Vergleicher können in der Farbe: Rot, Gelb und Schwarz dargestellt werden. Ein in Rot dargestellter Vergleicher arbeitet normal, ein in Gelb dargestellter Vergleicher ist ein Reverse-Vergleicher und ein schwarz dargestellter Vergleicher ist ein aktiver Vergleicher. Ein aktiver Vergleicher ist der aktuelle Vergleicher, an dem zwei Werte zeitlich gesehen gerade miteinander vergleichen werden.

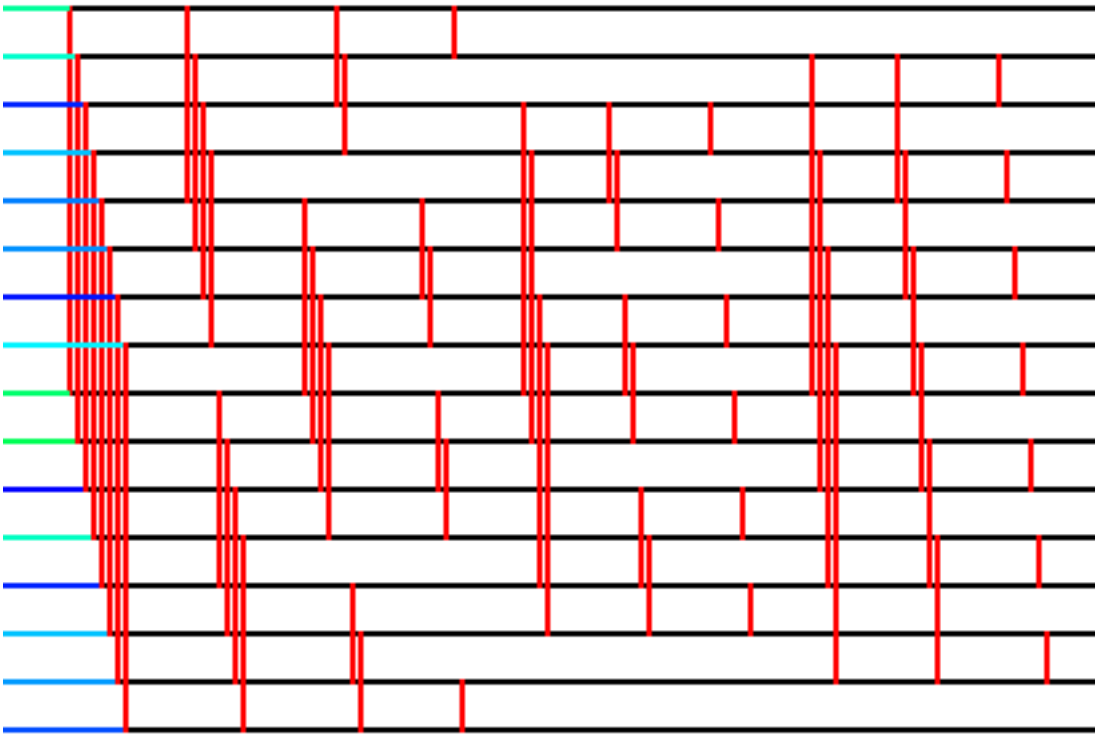

Abbildung 3.6: Unsortiertes Netzwerkdiagramm.

Implementation

Das Netzwerkdiagramm wird wie das Balkendiagramm als eigenes komplettes Bild erzeugt, welches mit der Größe des Fenster automatisch mit skaliert wird. Die vertikale Größe des Bildes wird automatisch durch die Anzahl der Leitungen und den Abstand zwischen den Leitungen bestimmt, indem die Breite und der Abstand zwischen den Leitungen zu der vertikalen Größe zusammengerechnet werden. Diese Größe gibt die Anzahl der Pixel an, die vertikal für das Bild benötigt werden. Die horizontale Größe wird durch die Anzahl der Vergleicher, der Abstand zwischen den Vergleicher sowie der Abstand zwischen den Vergleicherebenen bestimmt (Berechnung ist äquivalent zu der vertikalen Größe). Eine *Vergleicherebene* besteht aus allen Vergleichern, die parallel in einem Schritt ausgeführt werden können. Die Abstände sowie die Dicke der Leitungen und Vergleicher können beliebig innerhalb des Programmcodes durch Variablen einge-

stellt werden. Nach der Berechnung der Größe des Bildes wird dieses nun erstellt. Es werden zuerst die Leitungen und dann die Vergleicher gezeichnet. Danach werden die Farben für die einzelnen über die bereits vorhanden Leitungen darüber gezeichnet, bis ein Vergleicher die Leitung schneidet. Wenn das Programm ausgeführt und eine Sortierung gestartet wird, werden nacheinander die Vergleicher aktiviert. Ein aktivierter Vergleicher wird eingefärbt und vergleicht die beiden anliegenden Werte. Danach werden die farblich Markierungen dahinter bis zum nächsten Vergleicher, oder wenn keiner mehr vorhanden ist, bis zum Ende des Bildes gezeichnet. Abbildung 3.7 zeigt den farblichen Verlauf der Leitungen eines Netzwerkdiagrammes.

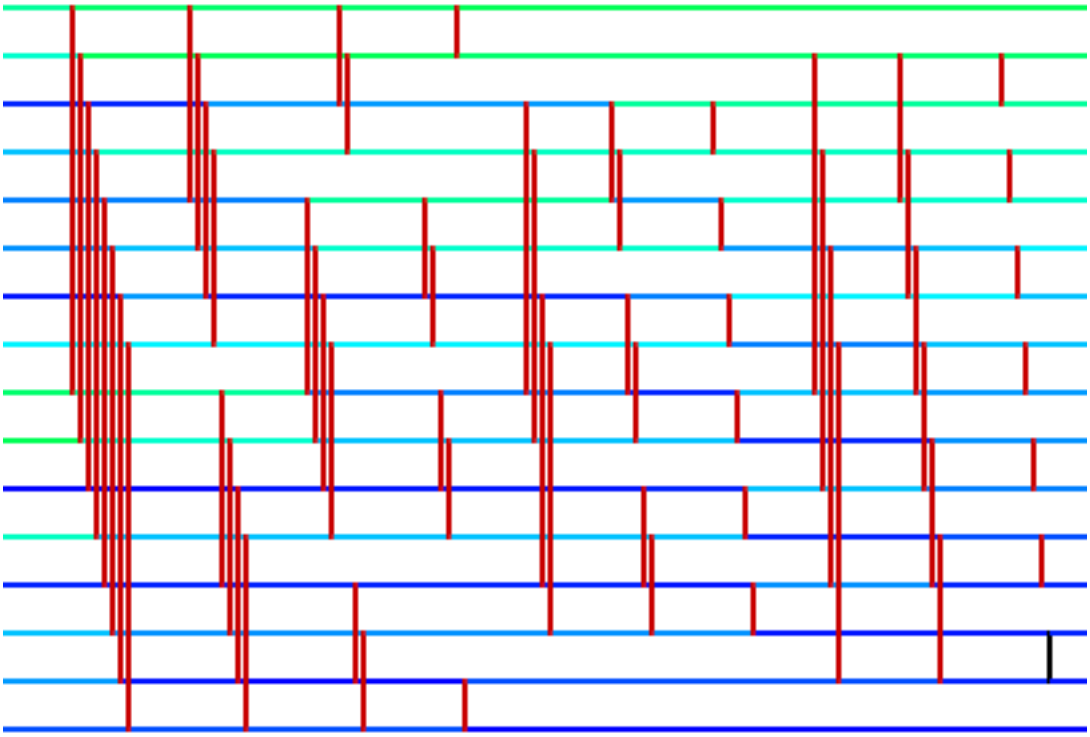

Abbildung 3.7: Sortiertes Netzwerkdiagramm.

3.6 Algorithmus implementieren

Die Implementation eines neuen Sortieralgorithmus ist einfach gestaltet und erfordert drei Schritte.

Im ersten Schritt muss der Sortieralgorithmus geschrieben werden. Jeder Sortieralgorithmus ist in einer eignen Klasse gekapselt und wird unter dem Namespace *Sorting_algorithmen* angelegt. Ein Sortieralgorithmus muss von der Klasse *SortingAlgorithm* erben. Durch die Vererbung wird sichergestellt, dass der neue Sortieralgorithmus die

Funktionen `get_sortingsteps()` und `get_sortingname()` beinhaltet. Diese Funktionen müssen bei der Implementation des neuen Sortieralgorithmus mit dem passenden Inhalt, also der Arbeitsweise des neuen Sortieralgorithmus, überschrieben werden. Die Funktion `get_sortingsteps()` nimmt ein Integer als Größe des Netzwerkes an und berechnet damit die Position der Vergleicher. Ein neuer Vergleicher wird mit dem Befehl `new Sortingstep()` erzeugt. Der Befehl benötigt beim Aufruf vier Übergabeparameter. Parameter eins und zwei geben die zu vergleichenden Leitungen an. Der Parameter drei sagt aus, ob der Vergleicher bei der Sortierung benutzt werden soll oder nicht. Bei dem letzten Parameter wird die Richtung angegeben, in die der Vergleicher die Elemente tauschen soll, also ob es sich bei dem Vergleicher um einen ReverseVergleicher handelt oder nicht. Die Vergleicher werden in einer Queue an das Hauptprogramm zurückgegeben. Die Funktion `get_sortingname` gibt den Namen des Sortieralgorithmus zurück, der benötigt wird, um den Sortieralgorithmus eindeutig zuzuordnen.

Im zweiten Schritt muss der neue Sortieralgorithmus in die Klasse *SortingFactory* eingetragen werden. Dabei muss ein neuer Eintrag erzeugt werden, der für den Namen des Sortieralgorithmus diesen als erzeugtes Objekt zurückgibt. Die Klasse *SortingFactory* ist der Anlaufpunkt für andere Klassen, wenn ein neuer Sortieralgorithmus instanziert werden soll.

Im letzten Schritt wird der Name des Sortieralgorithmus in der Klasse *data* in das readonly-Array `SORTINGMETHODS` eingetragen. Durch diesen Eintrag wird auf der GUI automatisch eine Box erzeugt, indem der neue Algorithmus dann zur Laufzeit ausgewählt werden kann.

3.7 Hinweise bei der Benutzung

In diesem Abschnitt wird das benötigte Framework und die allgemeine Hinweise zur Benutzung der Software gegeben. Es werden zudem die einzelnen Optionen der Haupt-GUI innerhalb des Programm erläutert.

Framework

Die Software wurde mit der Programmiersprache C-Sharp entwickelt. Damit die Software mit dem Betriebssystem Mac OS und Linux läuft, wird das Framework Mono benötigt. Mono stellt eine plattformunabhängige Laufzeitumgebung für C-Sharp Code zur Verfügung und wird primär von Xamarin weiterentwickelt. Für Windows wird das Framework .NET benötigt. Es wird von Microsoft zu Verfügung gestellt und weiterentwickelt.

GUI-Beschreibung

Abbildung 3.8 zeigt die Haupt-GUI mit den verschiedenen Funktionen zu dem Programm. Diese GUI bezieht sich immer auf einen *Datensatz*, welcher eine unsortierte Folge von Zahlen $a_0, \ldots, a_{n-1}$ in dem Programm ist. Dieser Datensatz kann durch

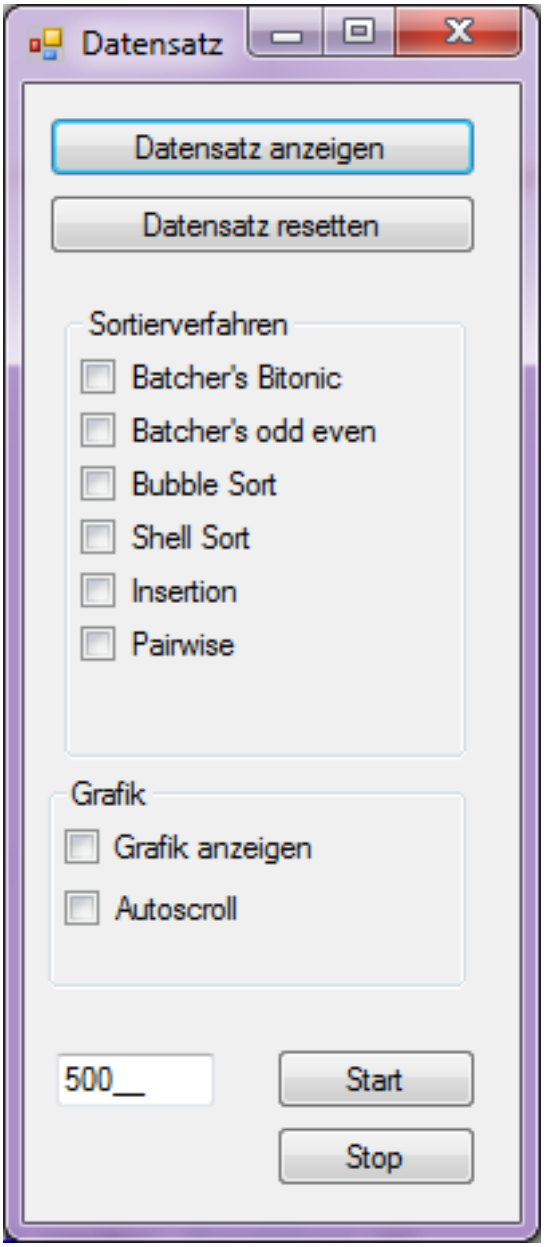

Abbildung 3.8: GUI Hauptoptionen(Screenshot).

den Button „Datensatz anzeigen“ angezeigt und durch den Button „Datensatz resetten“ zurückgesetzt werden. Ein zurückgesetzter Datensatz ist immer die ursprüngliche unsortierte Folge $a_0, \ldots, a_{n-1}$. Die implementierten Sortieralgorithmen können mithilfe der Boxen im Bereich Sortierverfahren ausgewählt werden. Sobald eine Box ausgewählt wurde, wird der Datensatz für das Sortierverfahren kopiert und der Sortieralgorithmus berechnet das dazugehörige Sortiernetzwerk. Es besteht auch die Möglichkeit mehrere Boxen auszuwählen. Dann werden mehrere Datensätze von dem Programm parallel abgearbeitet und visuell dargestellt. Die Box „Grafik anzeigen“ erzeugt ein Balken und ein Netzwerkdiagramm für alle ausgewählten Sortierverfahren. Zusätzlich wird zu jedem dieser Verfahren die Anzahl der Vergleiche mitgezählt. Die Option „Autoscroll“ ermöglicht es dem Programm automatisch das Netzwerkdiagramm mitzuscrollen. Das bedeutet, das Programm zentriert automatisch das Netzwerkdiagramm auf den aktuell ausgewählten Vergleicher. Die Textbox links neben dem Button „Start“ enthält die Information, wie schnell das Netzwerk sortieren soll. Es gibt die Pause nach jedem Vergleicher in Millisekunden an, die das Programm wartet, bis es den nächsten Vergleicher abarbeitet. Die Buttons „Start“ und „Stop“ starten und beenden den aktiven Sortiervorgang. Der Reset eines Datensatzes, die Auswahl eines neuen Sortierverfahren und die Anzeige der Grafik kann nur ausgewählt werden, wenn gerade kein aktives Sortierverfahren läuft.

3.8 Ausblick

Reflection und Dynamic Link Libary

Durch Reflection und Dynamic Link Libraries (DLL) lässt sich die Implementation von neuen Sortieralgorithmen vereinfachen. Reflection ist eine Möglichkeit in C-Sharp, dass ein Programm seinen eignen Code lesen und erkennen kann und dadurch selbst dynamisch zur Laufzeit damit Operationen ausführen kann, ohne das weiteres eingreifen erforderlich ist. Dies würde die Möglichkeit bieten, dass bei der Implementation eines neuen Algorithmus, nur noch dieser geschrieben werden muss, aber keine weiteren Änderungen mehr an dem vorhandenen Programm durchgeführt werden müssen. Das Programm kann zur Laufzeit diese neue Klasse (Sortierverfahren) entdecken und somit die Boxen auf der Haupt-GUI durch die Informationen aus dem Algorithmus selbst anlegen. Dieses Verfahren kann noch durch Dynamic Link Libraries erweitert werden. Ein neuer Sortieralgorithmus könnte durch eine Dynamic Link Library implementiert werden. Durch Reflection kann das Programm automatisch alle DLL's erkennen und in das Programm einbinden. Der Vorteil durch das Hinzufügen von DLL's ist, dass nur noch dieser Programmteil, aber nicht mehr das gesamte Programm neu kompiliert werden muss, wenn ein neuer Sortieralgorithmus hinzugefügt werden soll. Durch die Kombination dieser Verfahren besteht also die Möglichkeit sehr einfach einen neuen Sortieralgorithmus zu implementieren, indem er als DLL erstellt und kompiliert wird und von dem Programm dann automatisch erkannt wird.

Multithreading

Das Programm hat das Potential durch Multithreading bessere Laufzeiten erzielen zu können. Ein Thread kann immer nur von einem CPU-Kern bearbeitet werden. Bei mehreren Threads können also Multicore-CPU's besser ausgenutzt werden. Das Programm unterstützt zurzeit 2 Threads, einen für die Funktionalitäten der GUI sowie Benutzereingabe und einen für die Durchführung der Algorithmen, sowie das Zeichnen der neuen Bilder für die Diagramme. Eine Möglichkeit das Programm zu erweitern ist, das jedes ausgewählte Sortierverfahren einen eignen Thread bekommt. Dies würde das Programm bei mehreren ausgewählten Sortierverfahren entlasten, in dem die Arbeit auf mehrere CPU's aufgeteilt werden kann. Eine weitere Möglichkeit wäre, einem Sortierverfahren mehreren Threads zuzuweisen. Das Sortiernetzwerk könnte dann die Vergleicher, die in einem Schritt ausgeführt werden können, durch mehrere CPU's zeitgleich ausführen. Dies ergibt allerdings erst bei größeren Sortiernetzwerken Sinn, weil die Kommunikation der verschiedenen CPU's aufeinander abgestimmt sein muss, da ansonsten z.B. ein Vergleicher doppelt ausgeführt werden könnte. Die doppelte Ausführung ist in einer Ebene kein Problem, weil es die Zustände der Leitungen nicht verändern würde, allerdings in mehren Ebenen durch Folgezustände zu Fehlern kommen kann. Das Programm verfügt bereits bei dem Zeichnen und Erstellen der Bilder über Semaphoren. Eine *Semaphore* ist eine Möglichkeit kritischen Code maximal von einem Thread ausführen zulassen. Ein kritischer Codeblock besteht aus Anweisungen, dessen Code inkonsistente Zustände darstellen. Im Programm ist der kritische Code der Zugriff und die Bearbeitung der Diagramme, da diese nicht threadsicher sind. Wenn die Threadsicherheit nicht gewährleistet ist, können mehrere Threads sich bei der Benutzung eines Objektes behindern. Wenn das Programm mit Multithreading erweitert werden soll, muss ebenfalls auf die Kommunikation der Threads untereinander geachtet werden, da sich die Threads sonst gegenseitig behindern könnten.

Netzwerkdiagramm

Das Netzwerkdiagramm wird zurzeit als gesamtes Bild erzeugt. Bei größeren Netzwerken ist das Bild dementsprechend groß und verbraucht dadurch mehr Speicher. Damit größere Netzwerke visuell dargestellt werden können, sollte die Erstellung des Netzwerkdiagramm überarbeitet werden. Optimal wäre es, wenn das Bild dynamisch aus dem anzuzeigenden Bereich erzeugt werden kann. Das Datenmodell speichert bereits die Informationen über die Position der Vergleicher. Es müsste zusätzlich noch erweitert werden, das jeder Vergleicher die Werte speichert, die an ihm anliegen, damit das Sortiernetzwerk als Teilbild rekonstruiert werden kann. Ein Teilbild jedes mal neu zu erstellen verbrauchtwird mehr CPU Leistung als auf das aktuelle Bild die neuen Informationen zu zeichnen. Der Vorteil des Speicherverbrauches wird sich allerdings schnell bemerkbar machen, da damit größere Bilder gezeichnet werden können, die sonst nicht möglich wären.

Konfigurationsdatei

Durch eine Konfigurationsdatei können die Parameter für das Programm leichter eingestellt werden. Diese sind vor allem die Größe der Fenster, die Werte für das Farbspektrum und die Einstellungen zu Abständen und Farben der Leitungen und Balken in den Diagrammen. Im aktuellen Zustand müsste für jede Änderung die Variable im Programm geändert werden. Im Programm sind bereits diese Parameter als eine Variable in der jeweiligen Klasse vorhanden und mit einem vordefinierten Wert gesetzt. Es wäre möglich, sämtliche Konfigurationsvariablen in eine eigene Klasse auszulagern, die von anderen Klassen gelesen werden kann. Der Vorteil wäre, dass die Variablen leicht zu lesen und aus einer Konfigurationsdatei zu laden wären. Allerdings wäre durch eine Klasse, die alle Einstellungen speichert, das Prinzip des „Information Hiding“ verletzt, was ein wichtiger Designpunkt in der Softwareentwicklung ist. Information Hiding sagt aus, das Klassen nur die Informationen bekommen sollen, die sie auch bei ihrer Funktion benötigen, also das sie gar keinen Zugriff auf nicht benötigte Variablen haben.

Algorithmen

Das Programm verfügt über die in der Praxis verwendeten Sortieralgorithmen, die Sortiernetzwerke erstellen. Die Implementation weiterer Sortieralgorithmen ist vorteilhaft, um eine größere Auswahl zu ermöglichen. Zusätzlich sollte geprüft werden, ob der datenunabhängige Insertionsort bei Shellsort besser implementiert werden kann. Für die Implementation des AKS Sortiernetzwerkes (Ajtai, Komlós und Szemerédi) sollte sich darüber Gedanken gemacht werden, wie Expandergraphen gut und in der passenden Größe erzeugt werden können.

Akustik

Das Programm kann um die zusätzliche akustische Funktion erweitert werden, bei dem jedes mal, wenn auf ein Wert zugriffen wird bzw. wenn der Vergleicher zwei Werte vergleicht, ein Ton abgespielt werden kann. Die Höhe des Tons wird passend zum Farbverlauf bzw. den Werten an den Leitungen angepasst. Da bei einem Vergleich zwei Werte anliegen, könnte dieser z.B. zu einem Ton gemittelt werden, da es bei einem Sortiernetzwerk vorkommen kann, das auch über viele Leitungen hinweg verglichen wird. Dadurch kann das Sortiernetzwerk nicht nur visuell dargestellt werden, sondern auch akustisch gehört werden.

Beschränkungen

Die Größe der Netzwerke, die das Programm erstellen kann, ist abhängig von der Menge des verfügbaren Speichers. Des Weiteren besteht das Problem, dass es zu einem threadübergreifenden Fehler kommen kann, wenn das Programm zu schnell arbeitet. Dies liegt daran, dass die verwendeten Funktionen für die Bearbeitung der Diagramme nicht threadsicher sind. Das Problem ist, dass während einer Sortierung auf das Bild zugegriffen wird, weil bestimmte Abschnitte neu gezeichnet werden müssen, damit die

Diagramme aktuell bleiben. Durch ein Event, z.B. die Vergrößerung des Fensters oder das manuelle Scrollen im Netzwerkdiagramm, wird auch eine Neuzeichnung des Bildes ausgelöst. Wenn die Sortierung aktiv ist und ein Event ausgelöst wird, kann es zu diesen threadübergreifenden Fehlern kommen, da auf die Diagramme immer nur ein Thread zeitgleich zugreifen darf. Es sollte also darauf geachtet werden, das während einer aktiven Sortierung, die Veränderung der Größe des Fensters oder das manuelle Scrollen unterlassen wird.

4 Zusammenfassung

Mit dem Programm wurde eine Grundlage geschaffen, mit dem Sortiernetzwerke visuell dargestellt werden können. Durch das konstante Feedback des Betreuers zu dem Programm konnte es so für das Theoretische Institut für Informatik angepasst werden, wie es eingesetzt werden soll. Das Programm soll Studenten anschaulich vermitteln, wie ein Sortiernetzwerk arbeitet. Durch den Aufbau ist es dafür geeignet, neue Sortierverfahren einfach zu implementieren. Des Weiteren ermöglicht die einfache Bedienung der Software auch unerfahrenen Benutzer damit zu arbeiten.

Der Vergleich der praxistauglichen Algorithmen zeigt, dass der von Batcher entwickelte Odd Even Sort zusammen mit Pairwise Sort die besten Ergebnisse liefert. Das Bitonic Verfahren erzeugt ebenfalls gute Ergebnisse, allerdings benötigt es einige Vergleiche mehr. Die Verfahren Shellsort und Bubblesort sind für Sortiernetzwerke nicht geeignet, weil sie zu viele Vergleicher benötigen. Der Vergleich der Algorithmen zeigt, dass die Kriterien beim Sortieren in einem Sortiernetzwerk anders sind, als wenn ein Sortierverfahren softwaretechnisch implementiert werden soll. Dies liegt vor allem daran, dass in Hardware keine datenabhängigen Sortieralgorithmen gut implementiert werden können. Es zeigt sich ebenfalls, dass die guten Sortiernetzwerke nicht primitiv arbeiten und die durchschnittliche Anzahl der Vergleiche pro Vergleicher bei steigender Größe nur langsam zunimmt.

Die Entwicklung der Software war durch das kontinuierliche Feedback optimal. Dabei wurde zuerst das Grundgerüst der Software, also die GUI entwickelt und mit der Zeit dann weitere Funktionen hinzugefügt. Die Auswahl der Sortierverfahren war am Anfang gering und als visuelle Darstellung war nur das Balkendiagramm zur Verfügung. Das Netzwerkdiagramm, die weiteren Algorithmen und Funktionen wurden erst implementiert, als das Grundgerüst des Programms funktionsfähig war.

Das Verfahren von Ajtai, Komlós und Szemerédi (AKS) wurde am Anfang ebenfalls berücksichtigt, aber aufgrund seiner schlechten Praxistauglichkeit und Implementationsmöglichkeit dann wieder zurückgestellt. Die aktuelle Vermutung ist, dass das Verfahren erst bei Größen von 2^{6100} besser ist, als das Odd Even Verfahren [11]. Die Konstruktion von guten Expandergraphen, die für das AKS Verfahren benötigt werden, ist ebenfalls ein Problem.

Literatur

[1] Wilhelm Burger und Mark J. Burge. *Digital Image Processing: An Algorithmic Introduction Using Java*. 2009.

[2] Wikimedia Commons. *HSV Cone*. 2006. URL: https://upload.wikimedia.org/wikipedia/commons/e/ea/HSV_cone.png.

[3] R. M. Frank und R. B. Lazarus. "A High-speed Sorting Procedure". In: *Commun. ACM* 3.1 (Jan. 1960), S. 20–22. ISSN: 0001-0782. DOI: 10.1145/366947.366957. URL: http://doi.acm.org/10.1145/366947.366957.

[4] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.

[5] Bibliographisches Institut GmbH. *Duden: sortieren*. 2015-05. URL: http://www.duden.de/rechtschreibung/sortieren.

[6] Thomas N. Hibbard. "An Empirical Study of Minimal Storage Sorting". In: *Commun. ACM* 6.5 (Mai 1963), S. 206–213. ISSN: 0001-0782. DOI: 10.1145/366552.366557. URL: http://doi.acm.org/10.1145/366552.366557.

[7] Donald E. Knuth. "The Art of Computer Programming". In: *Sorting and Searching*. 1998.

[8] Gleen E. Krasner und Stephen T.Pope. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*. 1988.

[9] Hans Werner Lang. *Algorithmen in Java*. 2012.

[10] Hans Werner Lang. *Pairwise Sorting Network*. 2015. URL: http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/networks/pairwise.htm.

[11] Ian Parberry. *The Pairwise Sorting Network*. Center for Research in Parallel und Distributed Computing.

[12] Vaughan R. Pratt. *Shellsort and Sorting Networks*. 1980.

[13] Robert Sedgewick. *Algorithms in C 1*. 1998.

[14] D. L. Shell. "A High-speed Sorting Procedure". In: *Commun. ACM* 2.7 (Juli 1959), S. 30–32. ISSN: 0001-0782. DOI: 10.1145/368370.368387. URL: http://doi.acm.org/10.1145/368370.368387.

[15] wikipedia.org. *shellsort*. 2015-05. URL: http://en.wikipedia.org/wiki/Shellsort.

[16] Xiaochun Ye u. a. *High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs*. 2010.

Liste der Algorithmen

Abbildungsverzeichnis