Primzahlregression mit Hilfe genetischer Algorithmen

Björn Boss Henrichsen und Simon Budinsky
Betreut von Herrn Christoph Fischer

6. Februar 2017

Inhaltsverzeichnis

1 Einleitung **3**
- 1.1 Problemstellung . . . 3
- 1.2 Ziel des Projektes . . . 3

2 Relevanz von Primzahlen **4**
- 2.1 RSA-Verschlüsselung . . . 4
  - 2.1.1 Funktionsweise RSA . . . 4
  - 2.1.2 Der RSA-Wettbewerb . . . 4
  - 2.1.3 Folgen einer geknackten Verschlüsselung . . . 5
- 2.2 Primzahltests . . . 5
- 2.3 Finden der $n$ten Primzahl . . . 5
- 2.4 Die größte Primzahl . . . 6

3 Das Sieb des Eratosthenes **7**
- 3.1 Allgemein . . . 7
- 3.2 Funktionsweise . . . 7
- 3.3 Offizielle Variante . . . 8
  - 3.3.1 Programmablaufplan . . . 8
  - 3.3.2 Komplexität . . . 8
- 3.4 Optimierungsmaßnamen . . . 9
  - 3.4.1 Weglassen der eins . . . 9
  - 3.4.2 Quadrieren statt Radizieren . . . 9
  - 3.4.3 Zahlen im Vorraus streichen . . . 9
- 3.5 Messungen . . . 12
  - 3.5.1 Methodik . . . 12
  - 3.5.2 Vergleich zwischen offizieller und optimierter Variante . 15
  - 3.5.3 Messung mit Ausgabe aller Primzahlen . . . 18

4 Genetischer Algorithmus für Regressionen **19**
- 4.1 Definition genetischer Algorithmus . . . 19
- 4.2 Funktionsweise eines genetischen Algorithmus . . . 19
- 4.3 Unsere Implementierung . . . 22
  - 4.3.1 Die Funktionssyntax . . . 22
    - 4.3.1.1 Beispiel . . . 23
  - 4.3.2 Definitionen . . . 24
    - 4.3.2.1 Headerdatei primereg.h . . . 24
    - 4.3.2.2 Erläuterung . . . 25
  - 4.3.3 Wahl des Funktionsparameter Datentyps . . . 26
  - 4.3.4 Funktionsweise . . . 26

4.3.4.1 Programmablaufplan 26
4.3.4.2 Erläuterung 28
4.4 Vor-/Nachteile gegenüber herkömmlichen Regressionsmethoden 31
4.4.1 Vorteile 31
4.4.2 Nachteile 31

5 Ergebnisse **32**

6 Fazit **34**

A Quellen **35**

1 Einleitung

1.1 Problemstellung

In der Mathematik gibt es einige charakteristische Zahlen, die auf Grund der Komplexität ihrer Bestimmung eine besondere Anwendung finden. Eine solche Zahl stellen unter anderem Primzahlen dar. Unter einer Primzahl versteht man eine natürliche Zahl größer eins, welche nur durch sich selbst und eins teilbar ist. Somit ist die Mächtigkeit der Teilermenge einer Primzahl zwei.

Über 2000 Jahre lang konnte man keinen praktischen Nutzen aus dem Wissen über die Primzahlen ziehen. Dies änderte sich erst mit dem Aufkommen elektronischer Rechenmaschinen, bei denen die Primzahlen beispielsweise in der Kryptographie eine zentrale Rolle spielen.

Primzahlen sind hierfür deshalb so gut geeignet, weil es bisher keine Formel gibt, welche Primzahlen effizient berechenbar generiert.

1.2 Ziel des Projektes

Das Ziel unseres Projektes besteht darin, eine Folge zu finden, die Primzahlen möglichst gut beschreibt. Dazu setzen wir einen genetischen Algorithmus ein, welcher zu vorgegebenen Datenpunkten eine Regressionsfunktion bestimmen soll. Diese Datenpunkte sollen Primzahlen sein, welche wir mit Hilfe einer optimierten Variante des Siebes des Eratosthenes erzeugen.

2 Relevanz von Primzahlen

2.1 RSA-Verschlüsselung

In der Kryptographie gibt es Algorithmen, welche auf der Primfaktorzerlegung basieren. Ein modernes Beispiel hierfür ist der asymmetrische Verschlüsselungsalgorithmus RSA. Da bisher offiziell keine effiziente Methode gefunden wurde, Primfaktorzerlegung zu betreiben, ist eine Verschlüsselung mit RSA sicher.

2.1.1 Funktionsweise RSA

$\varphi(n)$: Eulersche Phi-Funktion
$\varphi(n) := |\{a \in \mathbb{N} | 1 \leq a \leq n \wedge ggT(a,n) = 1\}|$

1. Nimm zwei (sehr, sehr große) Primzahlen p und q
2. Bilde das RSA-Modul $N = p * q$
3. Bestimme $\varphi(N) = \varphi(p * q) = \varphi(p) * \varphi(q) = (p-1) * (q-1)$
4. Wähle eine Zahl e mit $1 < e < \varphi(N)$ mit $ggT(e, \varphi(N)) = 1$
5. Bestimme d mit $e * d \equiv 1 \ mod \ \varphi(N)$

Jetzt bilden (e, N) den öffentlichen und (d, N) den privaten Schlüssel.
Mit Hilfe des privaten Schlüssels lässt sich ein Klartext T zum Geheimtext G verschlüsseln:

$$G = T^e \ mod \ N$$

Der Geheimtext lässt sich mit dem öffentlichen Schlüssel wieder entschlüsseln:

$$T = G^d \ mod \ N$$

2.1.2 Der RSA-Wettbewerb

Aktuell gibt es einen RSA-Wettbewerb. Ziel ist es, die zwei Primfaktoren p und q einer RSA Zahl N zu finden. Denn findet man diese, so ist die Verschlüsselung geknackt.
Dies wird mit einem entsprechenden Preisgeld belohnt. Die bisher größte zerlgte RSA Zahl hat 232 Dezimalstellen (768 Bits, RSA-768) und wurde

am 12. Dezember 2009 von Thorsten Kleinjung, Paul Zimmermann u.a. für ein Preisgeld von $ 50000 USD faktorisiert. Hierfür wurde ein Rechencluster verwendet, welches etwa tausend Mal schneller als ein 2,2GHz AMD Opteron single-core Prozessor ist. Trotzdem dauerte die Faktorisierung zwei Jahre.
Dabei muss bedacht werden, dass heutzutage eher längere RSA-Zahlen verwendet werden. Häufig verbreitet sind Zahlen im Bereich von 309 bis 617 Dezimalstellen (1024-2048 Bits (das Preisgeld für eine RSA-2048-Zahl beträgt aktuell $ 200000 USD (02.01.17))).

2.1.3 Folgen einer geknackten Verschlüsselung

Zusätzlich sollte die Verschlüsselung in ein paar Minuten, am besten wenigen Sekunden, erfolgen. So könnte das Passwort für den Bankaccount per Man-In-The-Middle-Angriff einfach ausgelesen werden. Dies ist natürlich nur eine von den vielen fatalen Folgen, wenn RSA geknackt werden würde. Der Mensch könnte seine Privatsphäre verlieren und zum „gläsernen Menschen“ werden.

2.2 Primzahltests

Eine primzahlbeschreibende Folge wird vor allem in der Kryptographie benötigt. Damit RSA funktioniert, werden zunächst zwei Primzahlen erzeugt. Diese müssen zunächst gefunden werden.
Hierfür werden Test wie der Rabin-Miller-Test verwendet, der allerdings nur über eine Zahl zwei Aussagen treffen kann: zusammengesetzt oder wahrscheinlich prim.
Er liefert also nicht mit hundertprozentiger Sicherheit Zahlen, die anschließend für RSA verwendet werden. Ist eine Zahl nicht prim, funktioniert der RSA-Algorithmus nicht und kann geknackt werden.
Dies kann verhindert werden, indem stets mit hundertprozentiger Sicherheit über eine Zahl ausgesagt werden kann, ob sie prim ist. Hierfür könnte eine Folge für Primzahlen verwendet werden und somit die Primzahltest ablösen.

2.3 Finden der $n$ten Primzahl

Ein weiteres Problem ist folgendes: Finde die $n$te Primzahl.
Beim Sieb des Eratosthenes zum Beispiel müssen zunächst alle Primzahlen vor der $n$ten gefunden werden. Gäbe es eine Folge für Primzahlen, könnte dieses Problem wesentlich effizienter gelöst werden. Folglich wären Algorithmen zum Finden von Primzahlen wie das Sieb des Eratosthenes obsolet.

2.4 Die größte Primzahl

Vor etwa 2300 Jahren hat Euklid bewiesen: Es gibt unendlich viele Primzahlen. Dies wurde 1737 durch Leonhard Eulers Beweis bestärkt, der folgendes zeigte:

$$\sum_{p \in \mathbb{P}} \frac{1}{p} = \infty$$

Heutzutage versucht man stets größere Primzahlen zu finden. Daraus wurde eine Art Wettbewerb. Die Aktuell größte Primzahl (Stand Januar 2017) hat 22.338.618 Dezimalstellen und wurde 2016 vom *Great Internet Mersenne Prime Search*-Projekt gefunden.
Bei der Suche kann jeder teilnehmen. Hierfür muss lediglich eine kostenlose Software heruntergeladen werden, welche nach Primzahlen sucht. Findet der Comupter eine neue ,,größte"Primzahl, erhält man $3000 USD.
Mit Hilfe einer Folge für Primzahlen könnte diese direkt gefunden werden.

3 Das Sieb des Eratosthenes

3.1 Allgemein

Das Sieb des Eratosthenes ist ein Algorithmus, mit welchen man Primzahlen findet kann.
Er wurde nach dem griechischen Mathematiker Eratosthenes von Kyrene benannt, welcher im 3. Jahrhundert v. Chr. lebte.
Laut wikipedia.org habe dieser selbst nicht das Verfahren entdeckt. Eratosthenes habe lediglich die Bezeichnung „Sieb“ eingeführt.
Für Primzahlen kleiner 10.000.000 zählt der Algorithmus zu den effizientesten Methoden. Für größere Zahlen empfehlen sich effizientere Algorithmen. Beispiele sind der Lehmann-Test oder der Rabin-Miller-Test, welcher auch in modernen Verschlüsselungsalgorithmen (RSA) verwendet wird.

3.2 Funktionsweise

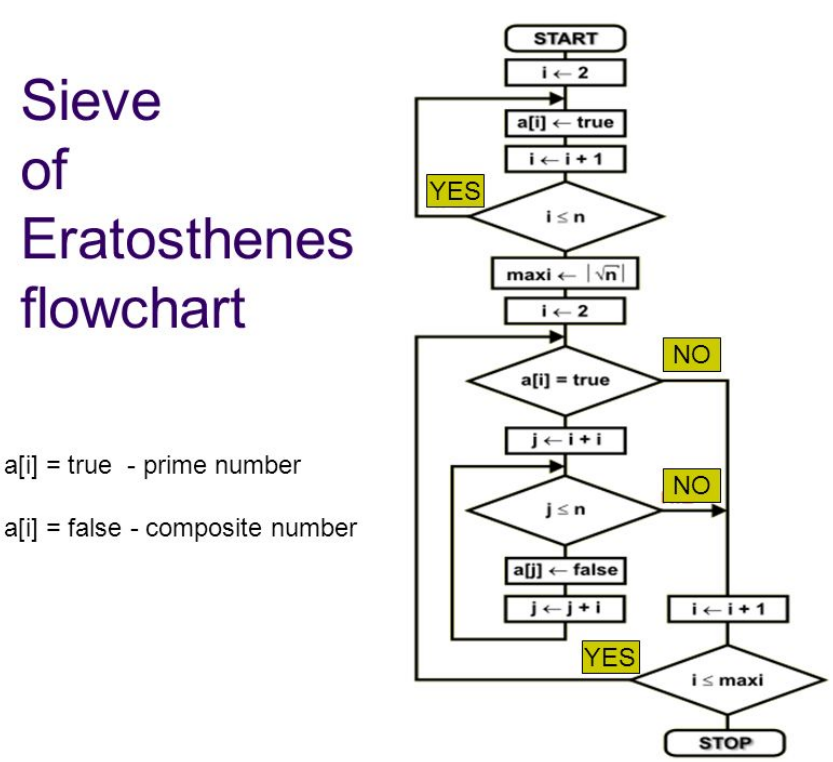
Zu Beginn werden alle natürlichen Zahlen von 2 bis einer Grenze n aufgeschrieben. Anfangs sind all diese nicht gestrichen und somit potentielle Primzahlen. Nun durchläuft man das Intervall. Trifft man auf eine ungestrichene Zahl, so ist diese eine Primzahl. Jetzt werden alle Vielfachen dieser gestrichen, die größer oder gleich deren Quadrat sind. Ist man am Ende des Intervalls angekommen, liegen sowohl gestrichene als ungestrichene Zahlen vor. Dabei ist jede ungestrichene Zahl prim.

Es genügt, die Vielfachen von Primzahlen zu streichen, die kleiner oder gleich der Wurzel der Grenze n sind. Denn ist ein Primfaktor einer zusammengesetzten Zahl immer kleiner gleich der Wurzel der zusammengesetzten Zahl.

Außerdem muss beim Streichen der Vielfachen erst mit dem Quadrat der Primzahl begonnen werden, da alle kleineren Vielfachen bereits getrichen wurden.

3.3 Offizielle Variante

3.3.1 Programmablaufplan



3.3.2 Komplexität

Die asymptotische Laufzeitkomplexität O in Abhängigkeit von einer oberen Grenze n ergibt sich wie folgt:

Sei $p_j \in \mathbb{P}$
So werden für jede Primzahl

$$p_j \leq \sqrt{n}$$

höchstens $\frac{n}{p_j}$ Zahlen gestrichen. Somit erhält man für die Anzahl an ,,Streichungen”:

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + ... = \sum_{p_j \leq \sqrt{n}} \frac{n}{p_j} = n * \sum_{p_j \leq \sqrt{n}} \frac{1}{n}$$

Nun muss die asymptotische Ordnung O vom zweiten Faktor gefunden werden. Diese ist $O(\log \log n)$ (Der Beweis hierfür ist nicht trivial und würde den Rahmen der Facharbeit sprengen). Daher ergibt sich eine Gesamtkomplexität von $O(n \log \log n)$.

3.4 Optimierungsmaßnamen

3.4.1 Weglassen der eins

Da die eins keine Primzahl ist, kann man sie vorab ignorieren. Somit wird die Länge des primes-Puffers um ein Byte verringert (Speichereffizienz). Zusätzlich muss zu Beginn ein Byte weniger initialisiert werden. In der äußeren for()-Schleife wird ebenfalls ein Durchlauf weniger benötigt (Zeiteffizienz).

3.4.2 Quadrieren statt Radizieren

Des Weiteren ist das Quadrieren an einem Computer effizienter als das Radizieren. Dies erklärt sich darin, dass für das Radizieren eine Bisektion (Software) benötigt werden kann. Dahingegen ist das Quadrieren lediglich eine Multiplikation, welche direkt in der ALU (Hardware) ausgeführt wird.
Somit kann man schreiben:

i: Aktuell betrachtete Zahl

$$i \leq \sqrt{n} \Rightarrow i * i \leq n$$

3.4.3 Zahlen im Vorraus streichen

Der Algorithmus kann weitergehend optimiert werden, indem man Zahlen vorab ,,streicht“. So haben wir alle Vielfachen von zwei bereits ,,gestrichen“. Daraus ergibt sich folgende Speichereinsparung:

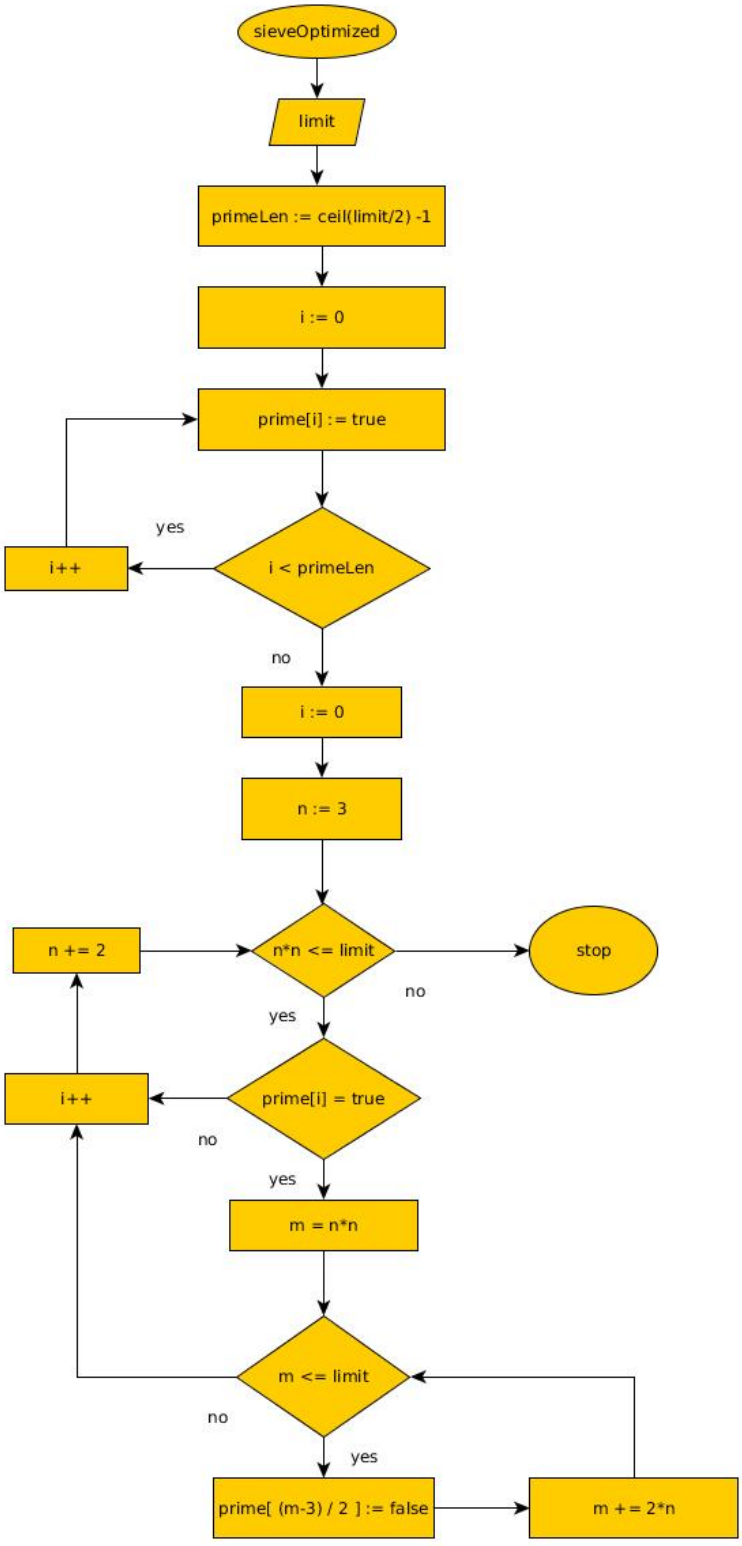
a: Belegter RAM-Speicher mit allen Zahlen [0;n]
b: Belegter RAM-Speicher ohne eins und geraden Zahlen
n: Obere Grenze des Siebes
k: Proportionalitätsfaktor

$$a = k * b \tag{1}$$
$$\Leftrightarrow n = k * (\frac{n}{2} - 1) \tag{2}$$
$$\Leftrightarrow k = \frac{n}{(\frac{n}{2} - 1)} \tag{3}$$
$$= \frac{2n}{n-2} \tag{4}$$

Foglich verwendet die optimierte Variante $\frac{n-2}{2n}$ Mal so viel Speicher, im Vergleich zur offiziellen. Somit können mehr Primzahlen auf dem gleichen Rechner gefunden werden.

Auch zeigen unsere Messungen, dass, wenn man alle Vielfachen von zwei weglässt, der Algorithmus mehr als doppelt so schnell terminiert (Zeiteffizienz).

Nach den oben genannten Optimierungen könnte ein Programmablaufplan so aussehen:


sieveOptimized
limit
primeLen := ceil(limit/2) -1
i := 0
prime[i] := true
yes
i++
i < primeLen
no
i := 0
n := 3
n += 2
n*n <= limit
stop
no
yes
i++
prime[i] = true
no
yes
m = n*n
m <= limit
no
yes
prime[(m-3) / 2] := false
m += 2*n

Nun könnte man immer mehr Zahlen im Vorraus ,,streichen". Hierfür gibt es allerdings andere Algortihmen. Das *Sieb des Atkin* zum Beispiel schließt vorher Zahlen bestimmter Restklassen aus und hat eine lineare Laufzeitkomplexität $O(n)$.

3.5 Messungen

3.5.1 Methodik

Um die Laufzeit des Siebes für verschiedene Grenzen *limit* zu messen, wurde das Programm für jedes *limit* jeweils neu kompiliert und ausgeführt. Nachdem die Primzahlen gefunden sind, gibt das Programm die böntigte Zeit zwischen Speicherallokation für den Primzahl-Puffer und dem Finden der Primzahlen in Sekunden aus. Diese Ausgabe wurde in eine .csv-Datei per Linux-Bash umgeleitet, um somit später mit LibreOfficeCalc Diagramme zu erzeugen. Dabei besteht diese Datei aus drei Spalten. Die erste Spalte repräsentiert die aktuelle Grenze *limit*, die zweite die gemessene Zeit in Sekunden und in der dritten Spalte steht die Anzahl gefundener Primzahlen.

Dafür verwendeten wir folgendes Bash-Script:

```
#!/bin/bash
#i: limit
for i in {100000000..6000000000..100000000}
do
    echo -n "$i," >> runtimes.csv
    gcc -std=c11 -DLIMIT=$i sieveRuntime.c -o sieveRuntime -lm \
                && ./sieveRuntime >> runtimes.csv
done
```

Wir implementierten folgende Algorithmen in C zur Zeitmessung:

Sieb ohne die eins:

```c
#include    <inttypes.h>
#include    <stdbool.h>
#include    <stdlib.h>
#include    <string.h>
#include    <sys/time.h>
#include    <stdio.h>

int main( void ) {
    struct timeval  start, end;
    gettimeofday(&start, NULL);

    bool* prime = (bool*)malloc(sizeof(bool) * (LIMIT-1));
    if( prime == NULL ) return 1;
    memset(prime, true, LIMIT-1);

    for(uint64_t i = 2; i*i <= LIMIT; i++) {
        if(prime[i-2]) {
            for(uint64_t j = i*i -2; j <= LIMIT; j += i)
                prime[j] = false;
        }
    }


    gettimeofday(&end, NULL);

    // Count primes
    uint32_t    nPrimes  = 0;
    for( uint32_t i = 0;  i < LIMIT-1; i++)
        if( prime[i] ) nPrimes++;

    printf("%.10f,%" PRIu32 "\n",
         end.tv_sec + end.tv_usec/1e6 - start.tv_sec - start.tv_usec/1e6,
          nPrimes);

        return 0;
}
```

Sieb ohne die eins und geraden Zahlen:

```
#include    <inttypes.h>
#include    <stdbool.h>
#include    <stdlib.h>
#include    <string.h>
#include    <sys/time.h>
#include    <stdio.h>
#include    <math.h>

int main( void ) {
    struct timeval  start, end;
    gettimeofday(&start, NULL);

    uint32_t primeLen = ceil(LIMIT / 2) - 1;
    bool*    prime    = (bool*)malloc(sizeof(bool) * primeLen);
    if( prime == NULL ) return 1;
        memset(prime, true, primeLen);

    for(uint64_t i = 0, n = 3;  n*n <= LIMIT;  i++, n += 2) {
        if(prime[i]) {
            for(uint64_t m = n*n;  m <= LIMIT;  m += 2*n)
                            prime[ (m-3) / 2 ] = false;
        }
    }

    gettimeofday(&end, NULL);

    // Count primes
    uint32_t    nPrimes  = 1;
    for( uint32_t i = 0;  i < primeLen; i++)
        if( prime[i] ) nPrimes++;

    printf("%.10f,%" PRIu32 "\n",
     end.tv_sec + end.tv_usec/1e6 - start.tv_sec - start.tv_usec/1e6,
      nPrimes);
      return 0;
}
```

[Anmerkung:]
Es wird bewusst nicht die Zeit zum Ausgeben (printf()) mit einbezogen, da diese allgemein einen großen Anteil an Laufzeit beansprucht. So ergibt sich bei unserem optimierten Algorithmus für n = 1.000.000 eine Zeit von etwa 1389.054 Sekunden mit Ausgabe aller Primzahlen und etwa 8.0484920487 Sekunden ohne.

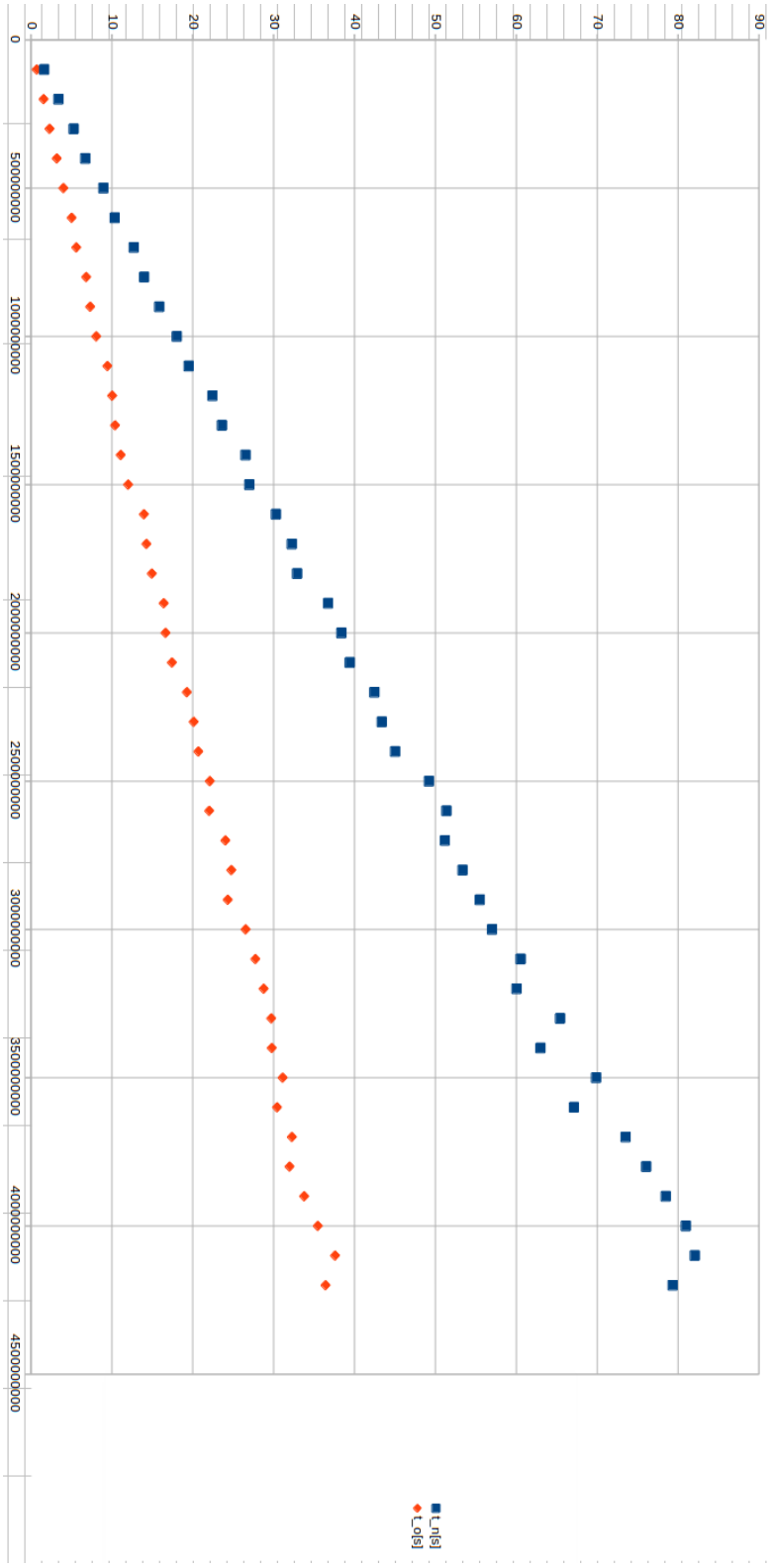
Die Messungen fanden in folgender Umgebung statt:

Operating System	*Linux Mint 18.1 Cinnamon 64-Bit*
Cinnamon Version	*3.2.7*
Linux Kernel	*4.4.0-53-generic*
Processor	*Intel Core i5-4440 CPU @ 3.10GHz x 4*
Memory RAM	*7.7GiB*

3.5.2 Vergleich zwischen offizieller und optimierter Variante

t_g: Benötigte Zeit in Sekunden, offizielle Variante (blau)
t_u: Benötigte Zeit in Sekunden, optimierte Variante (rot)

$limit$	t_g[s]	t_u[s]	$\frac{t_g}{t_u}$
100000000	1,6142570771	0,6927529358	2,3302060427
200000000	3,3796610065	1,5407490117	2,193518205
300000000	5,2564000593	2,2783710453	2,3070869296
400000000	6,7034089816	3,1714290402	2,1136872043
500000000	8,9381100896	3,9901179841	2,2400616035
600000000	10,3455499889	5,011878017	2,0642062624
700000000	12,6841129465	5,5832778916	2,271803982
800000000	13,9744109066	6,8093409411	2,0522413296
900000000	15,8358931078	7,3026679844	2,1685078853
1000000000	18,0069118877	8,0484920487	2,2373025629
1100000000	19,4879198822	9,4263940965	2,0673780114
1200000000	22,4203640433	10,0174689825	2,2381266248
1300000000	23,5954410898	10,3775840185	2,2736930915
1400000000	26,5114490883	11,0751651153	2,393774613
1500000000	26,9755919041	11,9925879208	2,2493553587
1600000000	30,2582220339	13,9449501191	2,1698336513
1700000000	32,2235829942	14,2521269333	2,2609666013
1800000000	32,8872118869	14,924550968	2,2035645801
1900000000	36,7206629502	16,3945279494	2,2398121534
2000000000	38,3611690887	16,617560026	2,3084718231
2100000000	39,3934800971	17,4123429244	2,2623882534
2200000000	42,4355878925	19,252283056	2,2041847073
2300000000	43,353328041	20,0979319791	2,1571039292
2400000000	45,0081739355	20,6734950301	2,1770955453
2500000000	49,1815101048	22,0877529756	2,2266416217
2600000000	51,3432890466	22,012063034	2,3325069062
2700000000	51,1586249138	24,0191830281	2,129906952
2800000000	53,3409920051	24,7522308979	2,1549973505
2900000000	55,47018401	24,3075119497	2,282018173
3000000000	56,9847639827	26,5199970692	2,148746994
3100000000	60,5142700262	27,7222269376	2,1828791086
3200000000	60,012045889	28,7487220151	2,0874683006
3300000000	65,3936210892	29,68840601	2,2026652784
3400000000	62,9704729115	29,7541698854	2,116357914
3500000000	69,8409569373	31,0977290192	2,2458539302
3600000000	67,1172831177	30,4034039054	2,2075581842
3700000000	73,4939509822	32,2375419811	2,2797628624
3800000000	76,0375679339	31,9458291063	2,3802033023
3900000000	78,455664996	33,7550660629	2,3242634113
4000000000	80,9324689933	35,4401829989	2,2836357531
4100000000	82,0384289596	37,5744409737	2,1833572725
4200000000	79,314230075	36,3976849106	2,1791308935


0
10
20
30
40
50
60
70
80
90
0
500000000
1000000000
1500000000
2000000000
2500000000
3000000000
3500000000
4000000000
4500000000
t_n[s]
t_o[s]

3.5.3 Messung mit Ausgabe aller Primzahlen

$limit$	t_g[s]	t_u[s]	Δt	$anzahl$
100.000	0,003	0,002	0,001	9592
1.000.000	1,216	1,205	0,011	78498
10.000.000	14,463	14,387	0,076	664579
100.000.000	143,18	142,439	0,741	5761455
1.000.000.000	1397,5	1389,054	8,446	50847534

4 Genetischer Algorithmus für Regressionen

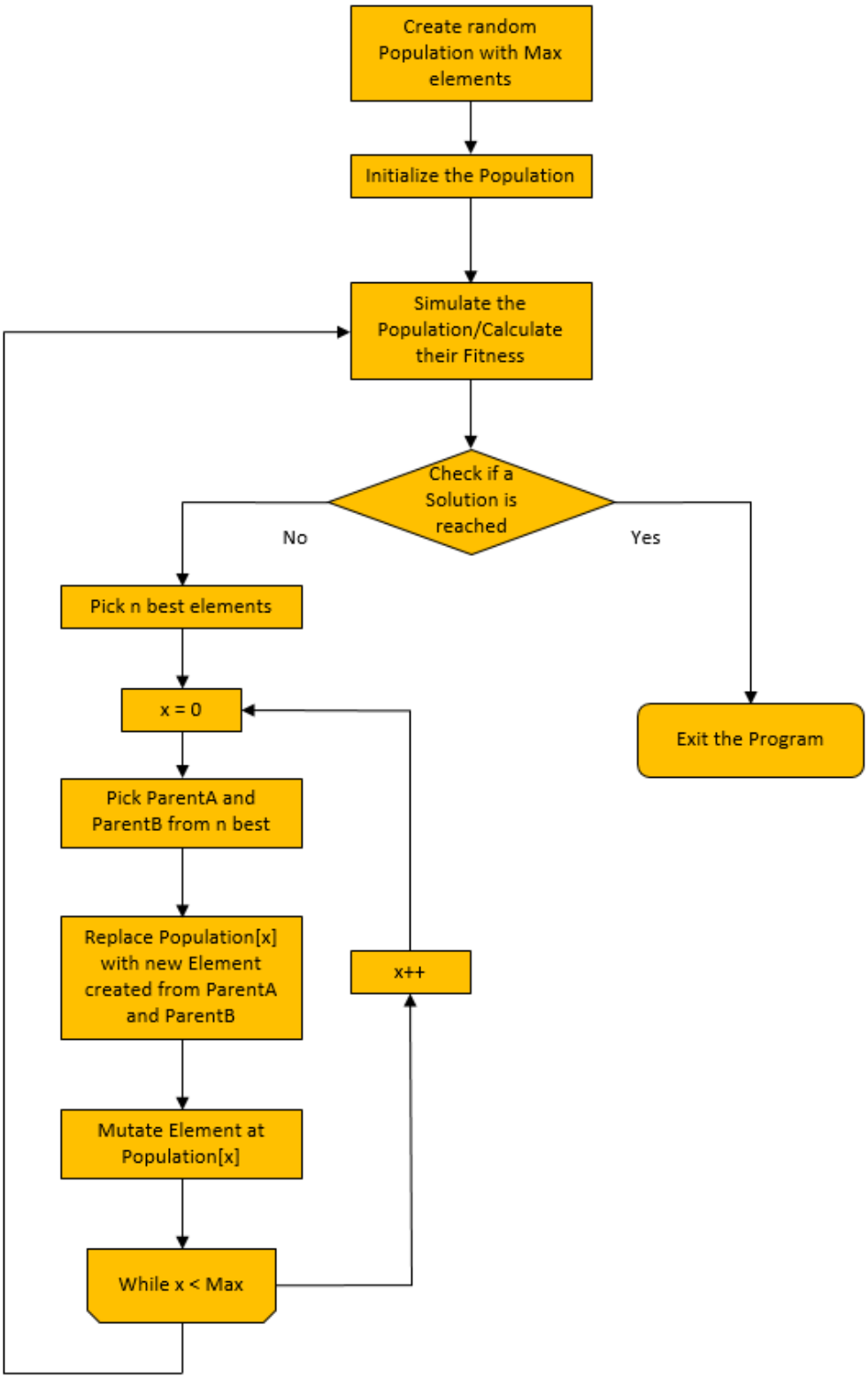
4.1 Definition genetischer Algorithmus

Ein genetischer Algorithmus, oder auch Evolutionärer Algorithmus, ist eine Klasse von Algorithmen, welche an die Funktionsweise der Natur angelehnt sind.
Hierbei werden aus einer Datenmenge, die dafür bestimmt ist, ein bestimmtes Problem zu lösen, die effizientesten ausgewählt und dafür verwendet die Anzahl an Datenmengen erneut zu erweitern mit den effizientesten als Grundgerüst.
Somit „lernt“ der Algorithmus aus seinen Fehlern, da nur die effizientesten Datenmengen „überleben“.

4.2 Funktionsweise eines genetischen Algorithmus

Zu Beginn wird erst einmal eine Darstellungsweise für das zu lösende Problem aufgestellt.
Anschließend wird eine Population generiert mit einer zufällig aufgebauten, oder bereits leicht geordneten, Struktur dieser Darstellungsweise. Anschließend muss jedes Element dieser Population von einer bewertenden Funktion, meist der Fitness-Funktion, die Fitness des Elements berechnen/bewerten.
Diese Funktion ist notwendig da es sonst, wie in der Natur auch, keinen Grund gäbe sich zu verändern, wenn es nicht für das Überleben beziehungsweise das Wohlergehen notwendig ist. Die Fitness-Funktion wird basierend auf das Problem festgelegt und ist in ihrer Struktur meistens einfacher. So kann diese beispielsweise bei einem Algorithmus, welcher ein simuliertes Auto zu fahren lernen soll, einfach nur eine Division von der zurückgelegten Strecke durch die benötigte Zeit sein.
Nachdem jedes Element der Population bewertet wurde und jedem ein Fitness Wert zugeordnet worden ist müssen die Elemente die am besten abgeschnitten haben aussortiert werden. In unserem Beispiel von oben wären dies die Elemente, welche die höchste Fitness erreicht haben. Dies ist aber nicht zwingend. So kann es auch sein, dass eine kleinere Fitness besser, als eine größere ist. Das hängt von der Fitness-Funktion und deren Funktionsweise ab.
Diese besseren ausgewählten Elemente repräsentieren nun die Eltern. Diese werden nun aus der Population entfernt und außerhalb gespeichert. Anschließend werden alle Elemente der Population gelöscht.
Nun wird für jeden Platz in der Population durch Zufall n, vorher festgelegte, Eltern aus den beiseitegelegten Elementen ausgewählt. Von diesen n

Eltern werden nun, meist ebenfalls durch Zufall, Teile aus der Datenmenge entnommen und in das neue leere Element der Population getan.
Anschließend kann es noch zu einer geringen Wahrscheinlichkeit, die meist als 1 bis 2 Prozent angelegt ist, zu einer Mutation kommen. Diese Mutation ist notwendig, damit sich die Population nicht auf den am Anfang durch Zufall erstellten Datenmengen festfährt, sondern immer was Neues hinzukommt und was Altes entfernt wird. Anschließend können die beiseitegelegten Elemente ebenfalls entfernt werden, da man für diese keinen Nutzen mehr hat.
Nun kann der gesamte Prozess von vorne beginnen mit der neu erstellten Population, die auf der alten Population aufbaut und die statistisch betrachtet nun bessere Fitness Werte erzielen wird.


Create random Population with Max elements
Initialize the Population
Simulate the Population/Calculate their Fitness
Check if a Solution is reached
No
Yes
Exit the Program
Pick n best elements
x = 0
Pick ParentA and ParentB from n best
Replace Population[x] with new Element created from ParentA and ParentB
x++
Mutate Element at Population[x]
While x < Max

4.3 Unsere Implementierung

4.3.1 Die Funktionssyntax

Damit der Algorithmus mit mathematischen Funktionen umgehen kann, bedarf es einer speziellen Funktionssyntax, welche von diesem interpretiert werden muss.
Wir haben uns für eine Assembler-Artige Darstellung von mathematischen Funktionen entschieden, wobei ein „Term" wie folgt aufgebaut ist:

$$Funktion\ Parameter_a \text{ optional } Parameter_b\ Solution$$

Dabei repräsentieren konstante Bytes mathematische Funktionen, um den Speicherverbrauch möglichst gering zu halten (im Vergleich zur Verwendung von Zeichenketten):

```
1  VAR  = 0x00, //Usage of Variable x
2  SLA  = 0x01, //Temp. Solution A
3  SLB  = 0x02, //Temp. Solution B
4  NUM  = 0x03, //following is a double
5  ADD  = 0x10, //double add(double a, double b)       -> a + b
6  SUB  = 0x11, //double substract(double a, double b) -> a - b
7  MUL  = 0x12, //double multiply(double a, double b)  -> a * b
8  DIV  = 0x13, //double divide(double a, double b)    -> a / b
9  POW  = 0x14, //double power(double a, double b)     -> a ^ b
10 LOG  = 0x15, //double logarithm(double a, double b) -> log(a) / log(b)
11 RND  = 0x16, //double round(double a)               -> (int64_t)(a + 0.5)
12 ABS  = 0x17, //double absolute(double a)            -> +a
13 SIN  = 0x18, //double sin(double a)                 -> sin(a)
14 COS  = 0x19, //double cos(double a)                 -> cos(a)
15 TAN  = 0x1a, //double tan(double a)                 -> tan(a)
16 ASIN = 0x1b, //double arcsin(double a)              -> asin(a)
17 ACOS = 0x1c, //double arccos(double a)              -> acos(a)
18 ATAN = 0x1d, //double arctan(double a)              -> atan(a)
19 EXP  = 0x1e, //double exponent(double a)            -> e ^ a
20 RET  = 0xff  //end of function
```

4.3.1.1 Beispiel

$$(|sin(x) + cos(x)|)^{tan(6)}$$

Dies sieht schließlich wie folgt aus:

SIN VAR SLA
COS VAR SLB
ADD SLA SLB SLA
ABS SLA SLA
TAN NUM 6.0 SLB
POW SLA SLB SLA
RET SLA

Das würde folgendes bewirken:
Zuerst wird der Sinus von der Variable berechnet und im temporären Register A gespeichert.
Anschließend wird der Cosinus der unabhängigen Variable berechnet und im temporären Register B gespeichert.
Die Werte dieser Register werden nun miteinander addiert und wieder im Register A gespeichert.
Von dem Wert in Register A wird nun der Betrag genommen und wieder in Register A zurückgespeichert.
Nun wird der Tangens der Dezimalzahl 6.0 berechnet und in Register B gespeichert.
Danach wird der Wert von Register A hoch den Wert von Register B gerechnet und dies im Register A gespeichert.
Zum Schluss wird Register A als das mit dem Ergebnis anerkannt und die Funktion wird beendet.

4.3.2 Definitionen

4.3.2.1 Headerdatei primereg.h

```
1  #pragma once
2
3  #define N_FUNCTIONS          256  //Number of Functions per
   ↪  generation
4  #define N_PARENTS            32   //Number of parents to pick
5  #define N_NUMBERS            32   //Number of Numbers needed
6  #define N_PRINTDATA          256  //Number of loops till print
   ↪  of data
7  #define PERC_MUTATION        5    //Chance in percent of a
   ↪  mutation of a function
8  #define PERC_INHERIT         60   //Chance of child taking
   ↪  over data from parent B
9  #define FUNCTION_START       0x10 //Number of first function
10 #define FUNCTION_ONEARG      0x16 //Number of first function
   ↪  to take one argument
11 #define FUNCTION_END         0x1e //Number of last function
12 #define MAX_RAND_DOUBLE      512  //Maximum double to be
   ↪  picked as random number
13 #define MAX_RAND_DOUBLE_DIG 5    //Digits of
   ↪  MAX_RANDOM_DOUBLE
14 #define PERC_ADD_NEW         60   //Percentage of adding a new
   ↪  function on creation
15
16 #define DEBUG                     //If defined, every
   ↪  generation results will be printed to stdout
17 #ifdef  DEBUG
18 #undef  N_PRINTDATA
19 #define N_PRINTDATA 1
20 #endif // DEBUG
```

4.3.2.2 Erläuterung

Diese stellen alle wichtigen Variablen des Quellcodes dar und können somit leicht verändert werden.
Hierbei Steht das N_ für eine Nummer/Anzahl, das PERC_ für ein Prozentwert und die anderen für vordefinierte Werte.
Falls DEBUG definiert ist, dann werden alle Funktionen nach einem Loop ausgegeben. Sonst werden sie nur alle N_PRINTDATA Durchläufe ausgegeben.

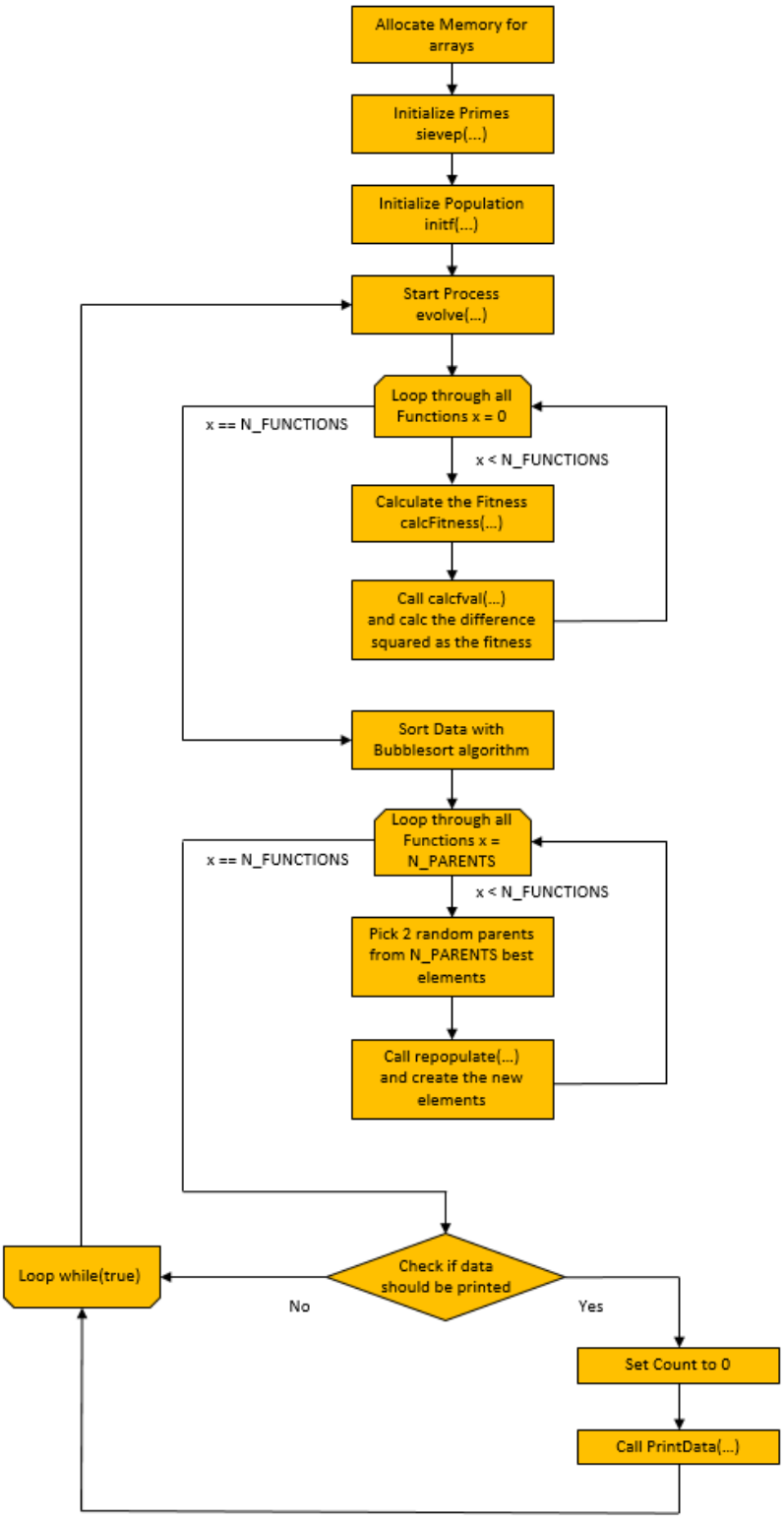
N_FUNCTIONS gibt an, wie viele Elemente in unserer Population sein sollen.
N_PARENTS gibt an, wie viele der besten Elemente in der Population erhalten bleiben sollen und als die Grundgerüste der neuen Elemente gewertet werden sollen.
N_NUMBERS hält die Anzahl an Nummern, mit denen die der Elemente abgeglichen werden sollen.
N_PRINTDATA gibt an, nach wie vielen Durchläufen des primären Loops die Daten ausgegeben werden sollen.
PERC_MUTATION ist ein Prozentwert, der die Wahrscheinlichkeit einer Mutation des Elementes angibt. Dieser Wert sollte gering gehalten werden, da es sonst sehr schnell zu dem Fall kommen kann, dass die neuen Elemente keine Ähnlichkeit zu den N_PARENTS Elementen mehr haben.
PERC_INHERIT ist ein Prozentwert, der angibt, wie groß die Wahrscheinlichkeit ist, dass ein neues Element die Eigenschaft eines der besseren Elemente erbt. Dieser Wert sollte nicht zu groß werden, da die neuen Elemente sonst sehr schnell größer und komplexer werden, da sie dann nahezu alles ihrer Eltern übernehmen werden.
FUNCTION_START gibt den Wert der ersten Funktion in unserem Syntax an.
FUNCTION_ONEARG gibt den Wert der ersten Funktion an, die nur ein Argument und nicht zwei nimmt.
FUNCTION_END gibt den Wert der letzten Funktion in unserem Syntax an.
MAX_RAND_DOUBLE stellt die maximale Größe eines durch Zufall generierten Zahlenwertes dar.
MAX_RAND_DOUBLE_DIG stellt die maximale Anzahl an Nachkommastellen der durch Zufall generierten Zahlenwerte dar.
PERC_ADD_NEW gibt in Prozent an, wie groß die Wahrscheinlichkeit ist, dass eine Funktion bei der Initialisierung eine weitere mathematische Funktion erhält. (Alle erhalten mindestens eine)

4.3.3 Wahl des Funktionsparameter Datentyps

Wir haben uns dazu entschieden, den Datentypen double als den primären Datentypen für unsere Funktionen zu wählen, da dieser eine hohe Präzision gewährleistet, jeder x86 und x86-64 Prozessor mit diesem umgehen kann und er nur 8 Byte pro Variable aufnimmt. Zudem stellt die Mathematik-Bibliothek, welche wir für die mathematischen Funktionen verwendet haben, Funktionen bereit, welche mit double arbeiten.

4.3.4 Funktionsweise

4.3.4.1 Programmablaufplan


Allocate Memory for arrays
Initialize Primes sievep(...)
Initialize Population initf(...)
Start Process evolve(...)
Loop through all Functions x = 0
x == N_FUNCTIONS
x < N_FUNCTIONS
Calculate the Fitness calcFitness(...)
Call calcfval(...) and calc the difference squared as the fitness
Sort Data with Bubblesort algorithm
Loop through all Functions x = N_PARENTS
x == N_FUNCTIONS
x < N_FUNCTIONS
Pick 2 random parents from N_PARENTS best elements
Call repopulate(...) and create the new elements
Check if data should be printed
No
Yes
Loop while(true)
Set Count to 0
Call PrintData(...)

4.3.4.2 Erläuterung

Unser Algorithmus startet damit, den Speicher für die Population zu allokieren. Anschließend allokiert er den Speicher für die Primzahlen, deren Erzeugung noch beschrieben wird. Danach wird der Zufallsgenerator mit der vergangenen Zeit seit dem 1. Januar 1970 in Millisekunden geseedet, damit wir möglichst zufällige Ergebnisse erhalten.

Nun werden die einzelnen Elemente der Population initialisiert. Hierfür wird die erste mathematische Funktion durch Zufall gewählt und deren Parameter ebenfalls als entweder VAR oder NUM. Anschließend wird geschaut, ob rand() mod 101 ¡= PERC_ADD_NEW gilt (Mit rand() mod 101 erhält man einen Zufallswert zwischen 0 und 100). Wenn dies gilt, dann wird eine weitere zufällige mathematische Funktion hinzugefügt. Falls nicht, dann wird geschaut, ob beide Register einen Wert beinhalten. Falls ja, dann wird noch eine zufällige Funktion, die zwei Argumente nimmt, gewählt und die nimmt dann als Parameter die beiden Register. Somit ist gewährleistet, dass kein Wert verloren geht und man nur noch ein volles Register am Ende hat, welches dann als das mit dem Ergebnis gewählt wird.

Falls als Parameter NUM gewählt wird, so heißt das, dass eine zufällige Fließkommazahl generiert werden muss. Diese wird wie folgt erzeugt:
Zuerst wird ein HighDouble Wert erzeugt. Dieser liegt innerhalb der Grenzen 0 und MAX_RAND_DOUBLE. Anschließend wird ein LowDouble Wert erzeugt, welcher in den Grenzen 0 und 10$\hat{}$MAX_RAND_DOUBLE_DIG liegt. Dieser wird nun solange in einer Schleife durch 10 geteilt, bis er kleiner als 1 ist. Der Generierte Wert setzt sich dann zusammen aus dem HighDouble und dem LowDouble, die miteinander addiert werden. Zum Schluss wird noch geschaut, ob der Wert negativ sein soll, was durch ein Aufruf der rand() Funktion geschieht.

Als Beispiel:
MAX_RAND_DOUBLE 512
MAX_RAND_DOUBLE_DIG 5
So werden als Zufallszahlen Zahlen zwischen -511.9999 und 511.9999 erzeugt.

Nachdem die gesamte Population mit validen Funktionen gefüllt worden ist, geht es in die Haupt-Schleife (*Main-Loop*). Hier wird zum einen die evolve Funktion aufgerufen, welche im Grunde den genetischen Algorithmus beinhaltet, und dann wird noch eine Zählervariable gehalten, welche angibt, wann die Daten alle ausgegeben werden sollen, und eine, die die derzeitige Gene-

ration mit sich trägt.

In der evolve Funktion wird zuerst die Güte (*Fitness*) der einzelnen Elemente der Funktion berechnet. Dabei wird diese anhand der Summe der Fehlerquadrate ermittelt: Diese Fitnessfunktion berechnet erst den Funktionswert des Elements für einen gegebenen x-Wert und anschließend seine Abweichung von der tatsächlichen Primzahl und quadriert diesen Wert. Diese Ergebnisse werden dann von x = 0 bis x = N_NUMBERS - 1 berechnet und aufaddiert. Der dann erhaltene Wert repräsentiert die Fitness. Umso geringer dieser Wert ist, desto besser ist die Funktion.
Wichtig ist, dass wir mathematische Fehler als vollständigen Eliminierungsfaktor ansehen. So werden Funktionen, die beispielsweise versuchen durch 0 zu teilen, mit dem Fitnesswert -1.0 versehen, was sie als nicht gültig kennzeichnet.

Sobald die Fitness für jede Funktion berechnet wurde, werden nun die N_PARENTS besten Elemente ausgewählt. Dies erfolgt mit Hilfe des Bubblesort Algorithmus, welcher dafür sorgt, dass die besten N_PARENTS Elemente die ersten der Population sein werden. Wir haben uns dafür entschieden, die N_PARENTS besten Elemente nicht für die neue Population zu verwerfen, da wir der Ansicht waren, dass es besser wäre, diese zu behalten. Somit werden in jedem Durchlauf lediglich N_FUNCTIONS – N_PARENTS Elemente der Population als neue Elemente aus den N_PARENTS Elementen generiert. Dies hat ebenfalls zur Folge, dass eine Funktion, die die Fitness 0 hat und somit perfekt ist, nicht durch Zufall verändert wird, zum Schlechteren und dann rausgeworfen wird. Dadurch ist gewährleistet, dass die N_PARENTS ersten Elemente der Population immer gute Werte erzielen werden.
Anschließend wird der Zufallsgenerator erneut geseedet, da es in dem von uns gewählten eine gewisse Wiederholung nach ungefähr $10\hat{5}$ Zahlen gibt, die wir verhindern möchten (es werden keine *true random numbers* erzeugt). Dann kommt es zur Neubildung der Population. Hierfür werden immer zwei der N_PARENTS gewählt. Es kann auch vorkommen, dass diese beiden gleich sind. Das wirkt sich aber nicht negativ auf den Algorithmus aus und kann somit enthalten bleiben.
Die *repopulate*-Funktion ist die Zeitaufwändigste Funktion und zugleich die komplexeste. Zuerst geht sie durch die Funktion der Eltern und zählt alle mathematischen Funktionen, die in diesen aufgerufen werden. Anschließend allokiert sie genug Speicher, um einen Array von allen mathematischen Funktionen und deren Parametern zu erstellen, die bei den Eltern vorkommen. Nachdem sie diese mathematischen Funktionen der Eltern alle eingelesen hat, mutiert sie diese. Dies geschieht, indem die Funktion durch den Array geht

und, sobald rand() mod 101 ¡= PERC_MUTATION gilt, ersetzt sie entweder die mathematische Funktion oder einen der Parameter. Nachdem dieser Schritt getan ist, findet die Funktion ein verwendbares Erst-Element, dass keine Register als Parameter nimmt. Falls es keine mathematische Funktion findet, auf die dies zutrifft, verändert sie das erste Element des Arrays so, dass dieses keine abhängigen Parameter mehr nimmt. Zum Schluss geht es durch den Array von Funktionen und schaut, ob rand() mod 101 ¡= PER_INHERIT gilt. Wenn dies der Fall ist, erbt das neue Element diese Mathematische Funktion von denen der Eltern, falls dies von den Parametern her zutrifft. Sie kann aber nicht vererbt werden, wenn die neue mathematische Funktion als Parameter Register A nimmt und Register B, aber Register A zu diesem Zeitpunkt leer ist. Dann wird diese Funktion übersprungen. Nach Durchgehen des Arrays wird, wie bei der Initialisierung, geschaut, ob in beiden Registern ein Wert ist und falls ja, dann wird erneut eine mathematische Funktion zufällig ausgewählt, die beide Parameter nimmt und nur ein Register zurückgibt. Zum Schluss wird noch der Schluss, also RET, an die Funktion angehangen und der reservierte Speicher wird freigegeben beziehungsweise der von dem neuen Element verkleinert.

Diese Funktion wird nun für jedes Element der Population ausgeführt, bis die gesamte Population erneut gefüllt wurde. Dann beginnt die Schleife wieder von vorne und die Fitness der neuen Funktionen wird erneut berechnet.

4.4 Vor-/Nachteile gegenüber herkömmlichen Regressionsmethoden

4.4.1 Vorteile

Da unser Algorithmus lediglich die neu erstellten Funktionen mit denen der eingegebenen Funktionen abgleicht, ist er dazu in der Lage, eine Regression für jegliche Funktionstypen durchzuführen, sofern man ihm die richtigen Funktionswerte zum Vergleichen gibt. So kann der Algorithmus beispielsweise ohne Vorwissen über den möglichen Funktionstyp eine lineare oder quadratische oder exponentielle Funktion finden. In unserem Beispiel füllen wir den Array mit Primzahlen, um eine Funktion zu finden, welche diese beschreibt. Wir haben ihn aber auch mit den Zahlen einer quadratischen Funktion gestartet, welche er innerhalb von 3 Generationen, also 3 Aufrufen der *evolve*-Funktion, erfolgreich fand.

4.4.2 Nachteile

Der Speicherverbrauch und die Laufzeit lassen sich nicht einschätzen, da diese beiden Werte abhängig sind von der Rückgabe der rand() Funktion. Aber es lässt sich sagen, dass der Algorithmus für höhere N_FUNCTIONS oder höhere N_PARENTS mehr Zeit in Anspruch nehmen wird. Des Weiteren wird der Speicherverbrauch für höhere PERC_ADD_NEW und PERC_INHERIT ebenfalls schneller ansteigen. Da für höhere PERC_INHERIT die Tendenz des Speicherverbrauchs schnell steigend ist, muss man mit diesem Wert vorsichtig umgehen, da es sonst sehr schnell zu einem Speicherüberlauf kommen kann.
Auch liefert der Algorithmus keine Folge eines vorgegebenen Grades (wie etwa bei einer linearen, quadratischen, ... Regression), was für manche Sachverhalte unerwünscht sein kann.

5 Ergebnisse

Um Folgen einer ausreichenden Güte zu erhalten, muss der Algorithmus über längere Zeit laufen. Im Rahmen der Arbeitsprojektwoche stand uns leider nur sehr wenig Zeit zur Verfügung. Trotzdem möchten wir einige Durchläufe dokumentieren.

Generation	Time	Size	Fitness	Term
0	3ms	23	0	ABS -14.556 SLA EXP SLA SLA TAN VAR SLB DIV SLB SLA SLA RET SLA f(x) = tan(x) /(10^abs(-14.556))
255	2524ms =0.04min	37	38.2706	ATAN VAR SLA MUL SLA VAR SLA ATAN VAR SLB MUL SLA SLB SLA ATAN VAR SLB ATAN SLA SLA MUL SLB SLA SLA ATAN VAR SLB MUL SLA VAR SLA MUL SLB SLA SLA RET SLA f(x) = (atan(atan(x) * x * atan(x)) * atan(x)) * x * atan(x)
767	8069ms =0.13min	30	28.8051	ATAN VAR SLA ATAN VAR SLB ADD VAR SLA SLA ATAN SLA SLA MUL SLA VAR SLA MUL SLA SLB SLA ATAN VAR SLB MUL SLA SLB SLA RET SLA f(x) = atan(atan(x) + x) * x * atan(x)^2
69119	698619ms =11.64min	46	25	ATAN VAR SLA ATAN SLA SLA ATAN VAR SLB MUL SLA SLB SLA ATAN VAR SLB MUL SLA VAR SLA MUL SLA SLB SLA ATAN VAR SLB MUL SLA SLB SLA ATAN VAR SLB RND SLA SLA ADD SLA SLB SLA RND SLA SLA RET SLA f(x) = md(md(atan(atan(x))*atan(x)^3 * x) + atan(x))
273151	2934391ms =48.91min	89	17.6869	ATAN 0.7912 SLA RND SLA SLA MUL SLA VAR SLA ATAN VAR SLB MUL SLA VAR SLA MUL SLA VAR SLA RND SLA SLA MUL SLA SLB SLA MUL SLA VAR SLA MUL SLA VAR SLA RND SLA SLA ATAN SLA SLA ATAN VAR SLB MUL SLA VAR SLA MUL SLA VAR SLA ADD SLB SLA SLA ATAN 0.7912 SLB POW SLA SLB SLA ATAN VAR SLB ADD SLB SLA SLA RET SLA f(x) = (atan(md(md(md(atan(0.7912)) * x^3) * atan(x) * x^2)) * x^2 + atan(x))^atan(0.7912) + atan(x)
366847	4166971ms =69.45min	70	13	SIN 0.7912 SLA MUL SLA VAR SLA MUL SLA VAR SLA SIN 0.7912 SLB POW SLA SLB SLA SIN 0.7912 SLB ADD SLB SLA SLA RND SLA SLA RND SLA SLA RND SLA SLA ADD SLA VAR SLA RND SLA SLA RET SLA f(x) = md((sin(0.7912) * x^2)^sin(0.7912) + sin(0.7912)) + x
373247	4264629ms =71.08min	88	12	ATAN 0.7912 SLA ATAN 0.7912 SLB RND SLA SLA ADD SLA VAR SLA ADD SLB SLA SLA MUL SLA VAR SLA RND SLA SLA RND SLA SLA ADD SLA VAR SLA RND SLA SLA SIN 0.7912 SLB RND SLA SLA ADD SLB SLA SLA SIN 0.7912 SLB POW SLA SLB SLA RND SLA SLA RET SLA f(x) = md((md((md(atan(0.7912)) + x + atan(0.7912)) * x) + x + sin(0.7912))^sin(0.7912))
392959	4557437ms =75.96min	100	11	ATAN 0.7912 SLA TAN 0.7912 SLB MUL SLA VAR SLA MUL SLA VAR SLA RND SLA SLA RND SLA SLA RND SLA SLA POW SLA SLB SLA SIN 0.7912 SLB RND SLA SLA POW SLA SLB SLA TAN 0.7912 SLB RND SLA SLA RND SLA SLA ADD SLB SLA SLA RND SLA SLA ADD SLA VAR SLA RND SLA SLA RND SLA SLA RND SLA SLA RET SLA f(x) = md(md(md(md(atan(0.7912) * x^2)^tan(0.7912))^sin(0.7912)) + tan(0.7912)) + x

Functions: 1024
Parents: 48
Numbers: 16
Mutation: 6
Inherit: 50 Add New: 75

6 Fazit

Unser Algorithmus ist flexibel. Er versucht, alle ihm gegebene Datenpunkte möglichst gut zu approximieren, unabhängig davon, welchem Wachstum diese entstammen.
Allerdings muss man stets Glück haben, eine passende Folge zu finden. Dies ist darin zu begründen, dass diese per Zufall generiert werden. Somit ist das Verhalten des Algorithmus nicht vorhersehbar, weswegen konventienelle Regressionsmethoden schneller zu einem Ergebnis kommen.
Aber vielleicht hilft genau dieses Prinzip genetischer Algorithmen, Folgen für bisher unbekannte Wachstumsklassen, wie etwa Primzahlen, zu finden. Unsere bisherigen Messungen zeigen, dass unser Programm während der Laufzeit immer bessere Funktionen generiert. Somit könnte der Algorithmus irgendwann eine Folge für Primzahlen finden. Hierfür benötigen wir aber mehr Zeit und Rechenleistung.
Zusammenfassend konnte das volle Potential des Algorithmus im Rahmen der Arbeitsprojektwoche nicht völlig ausgeschöpft werden. Daher können wir zum aktuellen Zeitpunkt keine endgültige Aussage über die Eignung eines genetischen Algorithmus zur Regression treffen. Deshalb werden wir den Algorithmus weitergehend ausführen und sind auf dessen Ergebnisse gespannt.

A Quellen

- https://en.wikipedia.org/wiki/Formula_for_primes
- https://de.wikipedia.org/wiki/Sieb_des_Eratosthenes
- https://de.wikipedia.org/wiki/Primzahl
- https://www.youtube.com/watch?v=ITLjfRiAc10
- http://www.iti.fh-flensburg.de/lang/algorithmen/asymp.htm
- http://www.programming-algorithms.net/article/41297/Sieve-of-Eratosthenes
- https://de.wikipedia.org/wiki/Miller-Rabin-Test
- http://www-cs-students.stanford.edu/~jl/Essays/ga.html
- https://en.wikipedia.org/wiki/RSA_numbers#RSA-768
- https://en.wikipedia.org/wiki/RSA_Factoring_Challenge
- https://de.wikipedia.org/wiki/Eulersche_Phi-Funktion
- https://www.youtube.com/watch?v=oXlY-yx1oIw
- https://en.wikipedia.org/wiki/Sieve_of_Atkin
- https://en.wikipedia.org/wiki/Largest_known_prime_number
- https://en.wikipedia.org/wiki/Divergence_of_the_sum_of_the_reciprocals_of_the_primes
- https://de.wikipedia.org/wiki/Evolution%C3%A4rer_Algorithmus
- THE NATURE OF CODE by Daniel Shiffman