Unser Algorithmus:

Unser Syntax

Wir haben uns für eine Assembler-Artige Darstellung von mathematischen Funktionen entschieden.

Funktion Parameter\_A opt. Parameter\_B Solution

Als Syntax haben wir folgendes aufgestellt:

VAR = 0x00, //Usage of Variable x

SLA = 0x01, //Temp. Solution A

SLB = 0x02, //Temp. Solution B

NUM = 0x03, //following is a double

ADD = 0x10, //double add(double a, double b) -> a + b

SUB = 0x11, //double substract(double a, double b) -> a - b

MUL = 0x12, //double multiply(double a, double b) -> a \* b

DIV = 0x13, //double divide(double a, double b) -> a / b

POW = 0x14, //double power(double a, double b) -> a ^ b

LOG = 0x15, //double logarithm(double a, double b) -> log(a) / log(b)

MOD = 0x16, //double modulo(double a, double b) -> (int64\_t)a % (int64\_t)b

RND = 0x17, //double round(double a) -> (int64\_t)(a + 0.5)

ABS = 0x18, //double absolute(double a) -> +a

SIN = 0x19, //double sin(double a) -> sin(a)

COS = 0x1a, //double cos(double a) -> cos(a)

TAN = 0x1b, //double tan(double a) -> tan(a)

ASIN = 0x1c, //double arcsin(double a) -> asin(a)

ACOS = 0x1d, //double arccos(double a) -> acos(a)

ATAN = 0x1e, //double arctan(double a) -> atan(a)

EXP = 0x1f, //double exponent(double a) -> e ^ a

RET = 0xff //end of function

So wird Folgende Funktion: (sin(x) + cos(x))tan(6)

Wie folgt in unserem Syntax aussehen:

SIN VAR SLA

COS VAR SLB

ADD SLA SLB 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. Nun wird der Tangens der Dezimalzahl 6.0 berechnet und in Register B gespeichert. Danach wird der Wert von Register A hoch den Wirt 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.

Folgende Definitionen gelten in unserem Programm:

#define N\_FUNCTIONS 128 // Number of Functions per generation

#define N\_PARENTS 8 // Number of parents to pick

#define N\_PRIMES 16 // Number of primes

#define SIEVE\_LIMIT 100

#define N\_PRINTDATA 256 // Number of loops til print of data

#define PERC\_MUTATION 2 // Chance in percent of a mutation of a function

#define PERC\_INHERIT 65 // Chance of child taking over data from parent B

#define FUNCTION\_START 0x10 // Number of first function

#define FUNCTION\_ONEARG 0x17 // Number of first function to take one argument

#define FUNCTION\_END 0x1f // Number of last function

#define MAX\_RAND\_DOUBLE 65536 // Maximum double to be picked as random number

#define MAX\_RAND\_DOUBLE\_DIG 5 // Digits of MAX\_RANDOM\_DOUBLE

#define PERC\_ADD\_NEW 60 // Percentage of adding a new function on creation

#define DEBUG // If defined, every generation results will be printed to stdout

#ifdef DEBUG

#undef N\_PRINTDATA

#define N\_PRINTDATA 1

#endif // DEBUG

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.

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 gut umgehen kann und er nur 8 Byte aufnimmt pro Variable. Zudem stellt die math Library, welche wir für die mathematischen Funktionen verwendet haben, Funktionen bereit welche mit double arbeiten.

Unser Algorithmus startet damit den Speicher für die Population zu allocaten. Anschließend allocated er den Speicher für die Primzahlen, deren Erzeugung noch beschrieben wird. Danach wird der Zufallsgenerator mit der 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() % 101 <= PERC\_ADD\_NEW gilt, mit rand() % 101 erhält man einen Zufallswert zwischen 0 und 100, und wenn ja, 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 double Nummer 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^MAX\_RAND\_DOUBLE\_DIG liegt. Dieser wird nun solange in einem Loop 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.

Falls die Definitionen wie folgt lauten:  
MAX\_RAND\_DOUBLE 65536

MAX\_RAND\_DOUBLE\_DIG 5

So werden als Zufallszahlen Zahlen zwischen -65535.9999 und 65535.9999 erzeugt werden.

Nachdem die gesamte Population mit funktionstüchtigen Elementen gefüllt worden ist geht es in den 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.

In der evolve-Funktion wird zuerst die Fitness der einzelnen Elemente der Funktion berechnet. Die 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\_PRIMES berechnet und aufaddiert. Der dann erhaltene Wert repräsentiert die Fitness. Umso geringer dieser Wert ist, umso 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 wird gemacht mit dem 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 0.0 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 Zufalls Generator erneut geseedet, da es in diesem eine gewisse Wiederholung nach ungefähr 10^5 Zahlen gibt, die wir verhindern möchten. Dann kommt es zur Repopulation der Population. Hierfür werden immer zwei der N\_PARENTS gewählt. Es kann auch vorkommen, dass ParentA und ParentB gleich sind, das wirkt sich aber nicht negativ auf den Algorithmus aus und kann somit enthalten bleiben.

Die Repopulations-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 allocated 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 jedes Mal, wenn rand() % 101 <= PERC\_MUTATION gilt ersetzt sie sowohl die mathematische Funktion an dieser Stelle im Array, als auch deren 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() % 101 <= PER\_INHERIT gilt. Wenn es das tut, erbt das neue Element diese Mathematische Funktion von denen der Eltern, falls dies von den Parametern her zutrifft. Sie kann somit 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 allocatete 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 der Loop wieder von vorne und die Fitness der neuen Funktionen wird erneut berechnet.