Code Generierung und Training eines CGP Klassifikators zur handschriftlichen Ziffernerkennung

Björn Piepenbrink

13.09.2018

Fakultät für Informatik
Algorithm Engineering (LS 11)
Technische Universität Dortmund

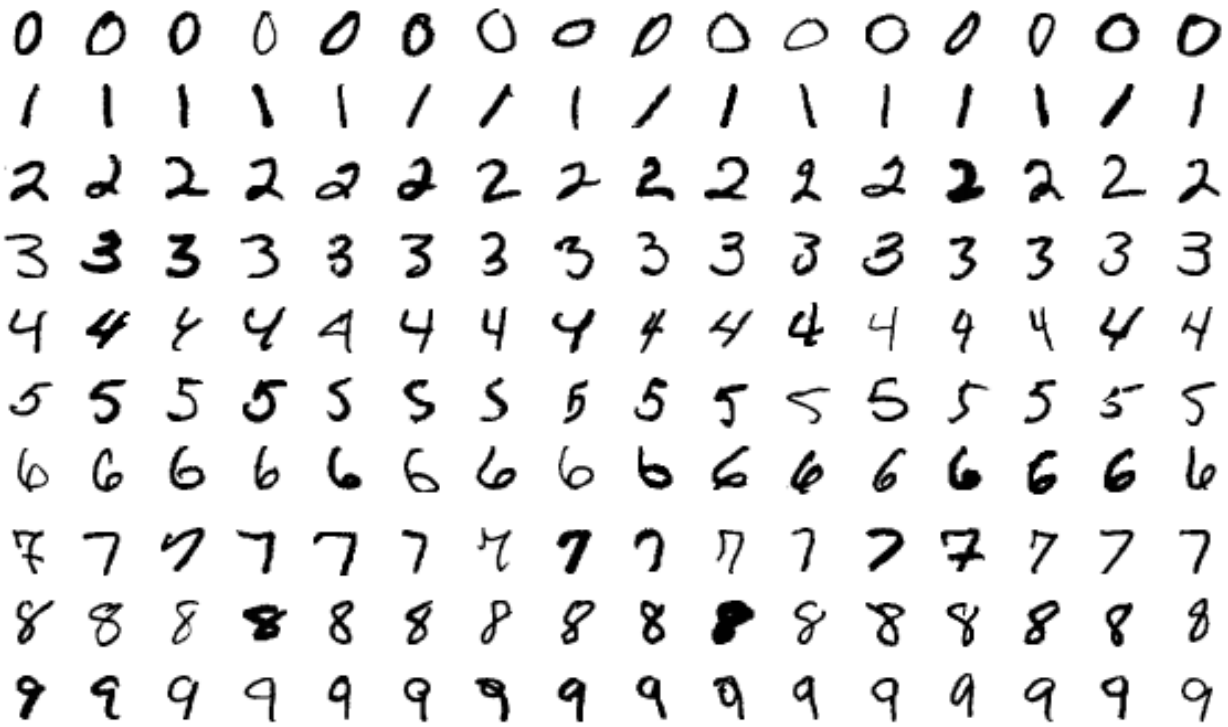
Gliederung

- Problemstellung
- Cartesian Genetic Programming
- Implementierungen
- Ergebnisse CGP
- Klassifikator-Generierung
- ECGP
- weitere Ergebnisse
- Fazit

Problemstellung

Handschriftliche Ziffern korrekt erkennen
Dafür: MNIST-Datensatz [1]

- 60.000 Trainings-Bilder
- zentriert in 28x28 Bild
- Klassifikation zu jedem Bild bekannt



[1] http://yann.lecun.com/exdb/mnist/

Cartesian Genetic Programming

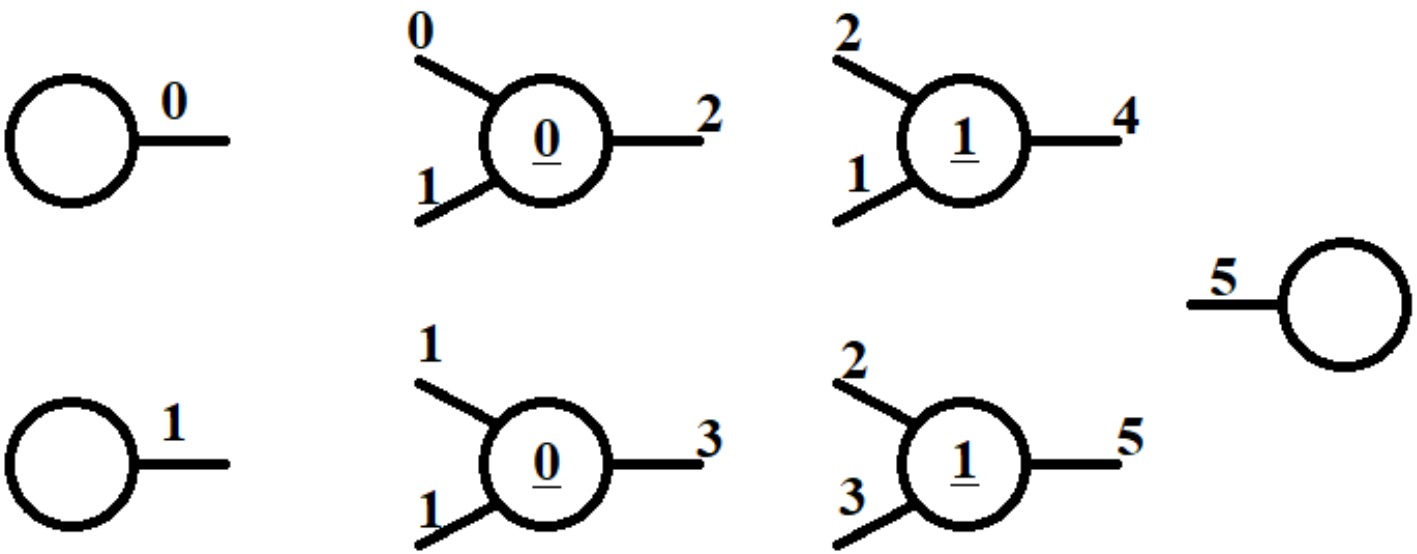
- CGP ist eine Form von evolutionären Algorithmen
- Von Julian F. Miller entwickelt
- Programme als azyklische, 2D-Graphen dargestellt (Individuen)
- Bestehen aus einer Reihe von Knoten
- Als Integer codiert
- Nutzen (1+4)-Evolutions-Algorithmus
- Fitness = Fähigkeit Problemstellung zu lösen
- Hier: Fitness = Anzahl der falsch klassifizierten Ziffern

(1+4)-Algorithmus

1. Erstelle eine Population aus 5 zufälligen Individuen
2. Generations-Nummer: g = 0
3. Berechne Fitness für jedes Individuum
4. Wähle das fitteste Individuum aus (in CGP: Neutrale Mutation)
5. Mutiere dieses um 4 Kinder zu generieren
6. Erstelle neue Generation mit diesen Individuen
7. g++
8. 3-7: bis akzeptable Fitness oder maximale Generations-Anzahl

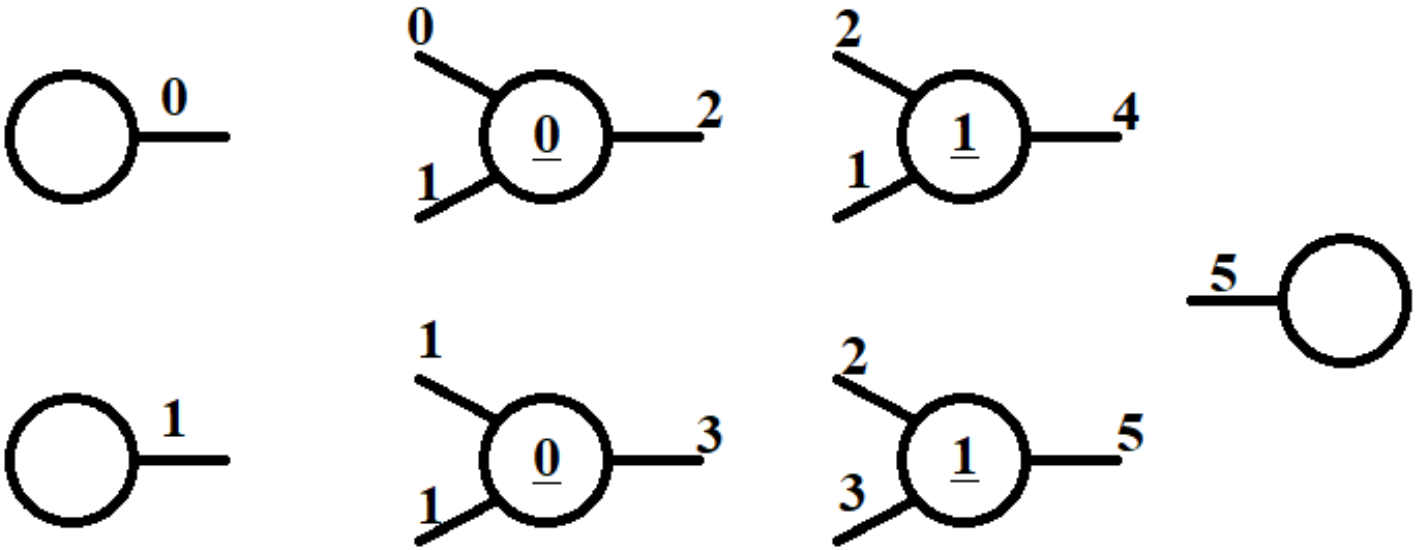
Allgemeinere Version: $(\mu + \lambda)$-Algorithmus

Aufbau eines CGP Individuums



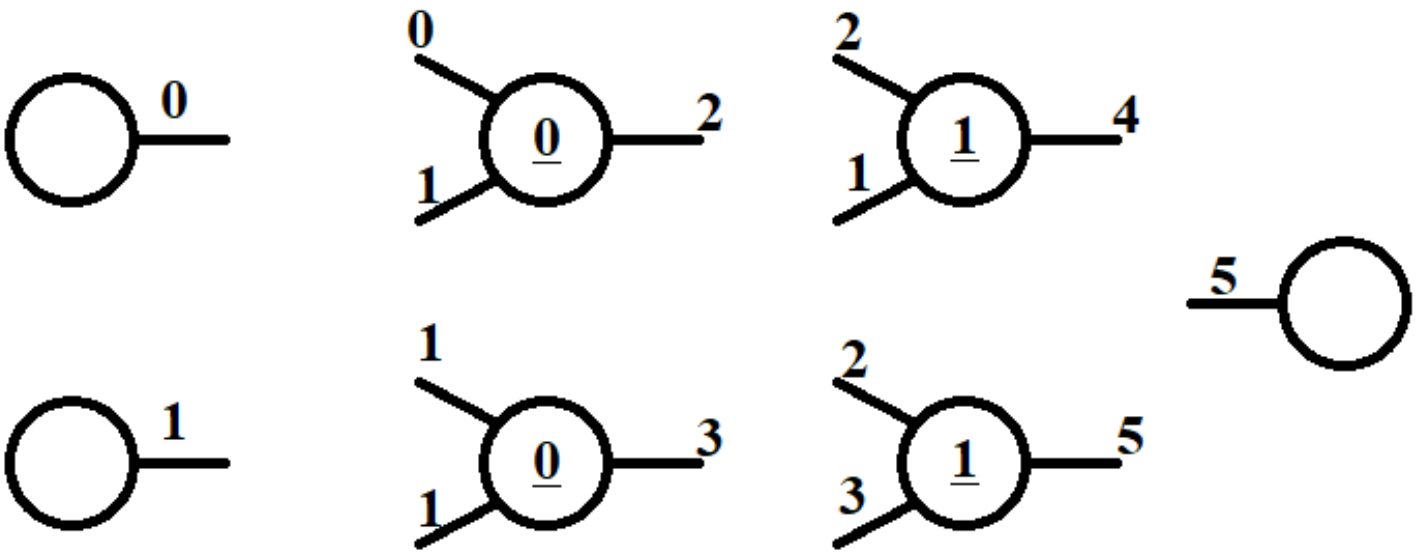
- Individuum besteht aus Knoten und Output-Adressen
- Ein Knoten besteht aus:
  - Funktions-Gen: Funktion des Knoten
  - Verbindungs-Genen: Eingaben für die Funktion
- Zu-Nutzende-Funktionen in Funktions-Tabelle
- Output-Adressen für die Ausgabe

Aufbau eines CGP Individuums



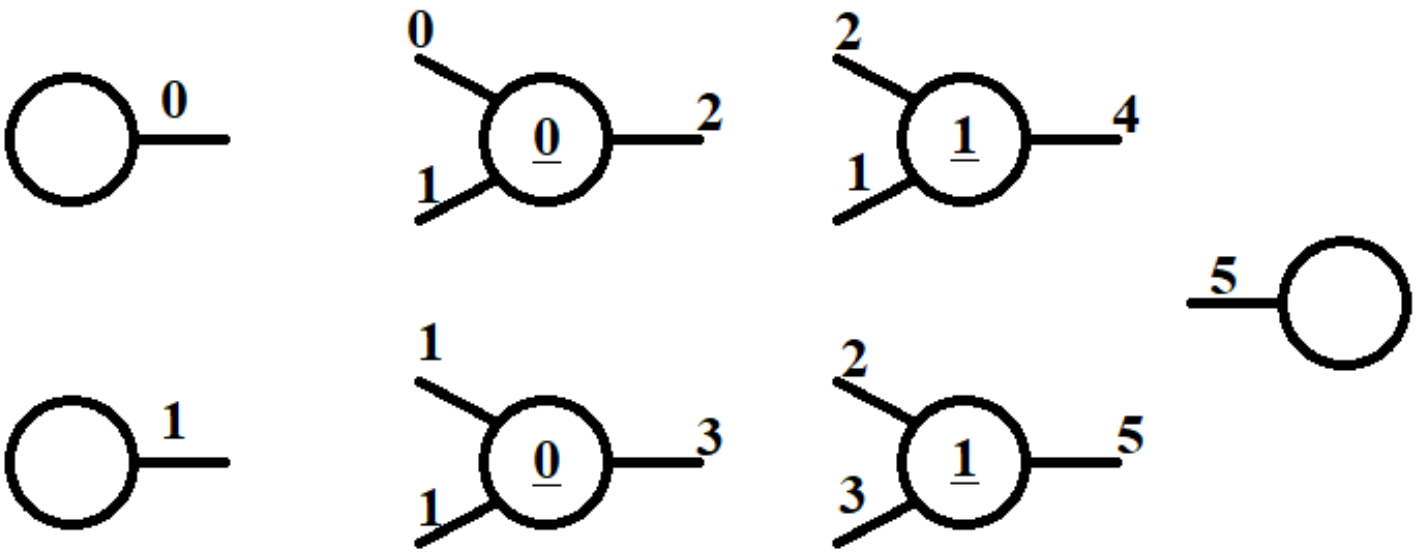
- Jeder Knoten besitzt eine Adresse
- Inputs nehmen die ersten Adressen ein
- Über Adressen werden Inputs und Knoten referenziert
- In diesem Beispiel:
  - Input-Adressen: 0, 1
  - Knoten-Adressen: 2, 3, 4, 5

Output-Berechnung



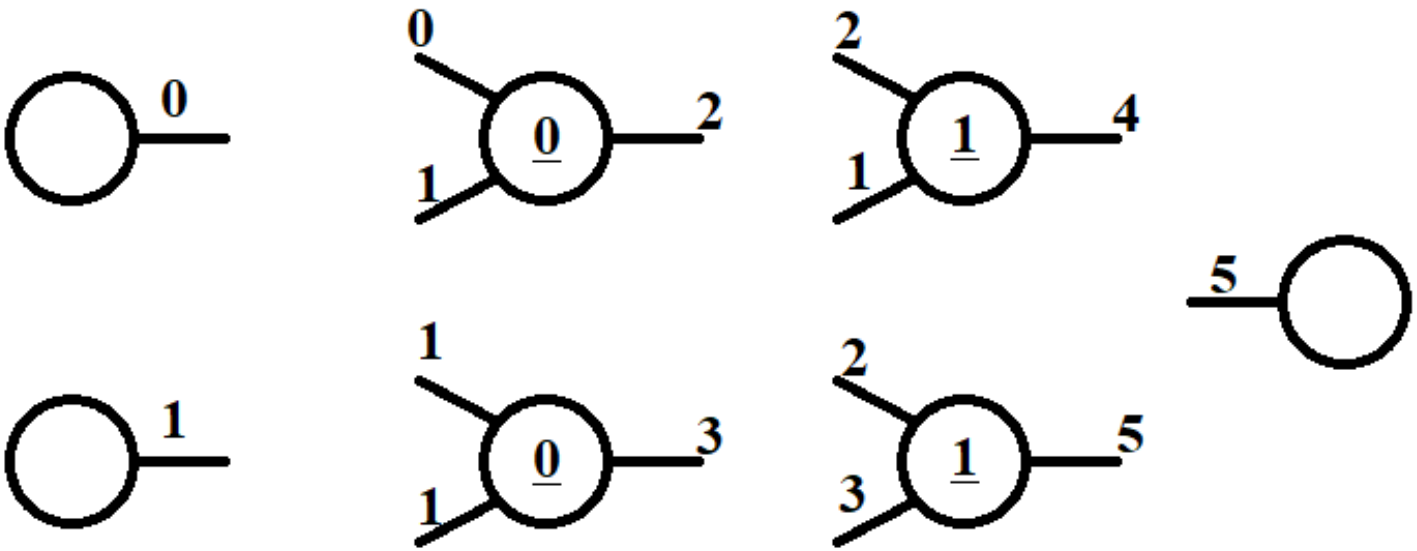
- Rekursiv vom Output ausgehend
- alle referenzierten Knoten markieren (aktive Knoten)
- Für diese Knoten Output berechnen
- In diesem Beispiel:
  - Funktions-Set = { $\underline{0}$:+, $\underline{1}$: * }
  - codierte Funktion = $(x_0 + x_1) - (x_1 + x_1)$

Verbundenheit



- Outputs und Input sind mit Knoten oder Inputs verbunden
- Graph ist feed-forward
- Beeinflussbar mit levels-back-Parameter
- Bei einem levels-back-Wert von 2:
  - Knoten kann Input als Verbindung wählen oder
  - einen der zwei Knoten vor ihm

Genotyp



- Genotyp beschreibt das Individuum
- Nur Knoten-Gene und Output-Adressen werden codiert
- Genotyp in diesem Beispiel:
  - 0 0 1 0 1 1 1 2 3 1 2 1 5.
- Anzahl der Inputs muss bekannt sein

Mutation

- Ziel: Individuum verändern
- Gegeben: Individuum und Mutationsrate

Vorgehen

So oft wie nötig:

- zufälliges Gen wählen
- Dieses Gen zu einem zufälligen Wert ändern
  - Funktions-Gen zu zufälliger Funktions-Adresse
  - Verbindungs-Gen zu zufälliger Verbindungs-Adresse (gilt auch für Output-Gene)

(levels-back-Parameter beachten!)

Parameter für CGP

- Mutations-Rate
- Knoten-Anzahl
- maximale Generationsanzahl
- Funktions-Set
- Output-Anzahl
- levels-back-Parameter
- μ und λ

Gegebene Werte:

- Input-Anzahl: 784
- Gewünschter Output

Implementierungen

- In Java nur ein Framework für CGP (CGP for ECJ)
- kohärente Probleme zwischen CGP for ECJ und ECJ
- Deswegen: eigenes Framework entwickelt

PCGP - Piepenbrink Cartesian Genetic Programming

- Implementierung von CGP in Java für MNIST-Klassifizierung

Ergebnisse CGP

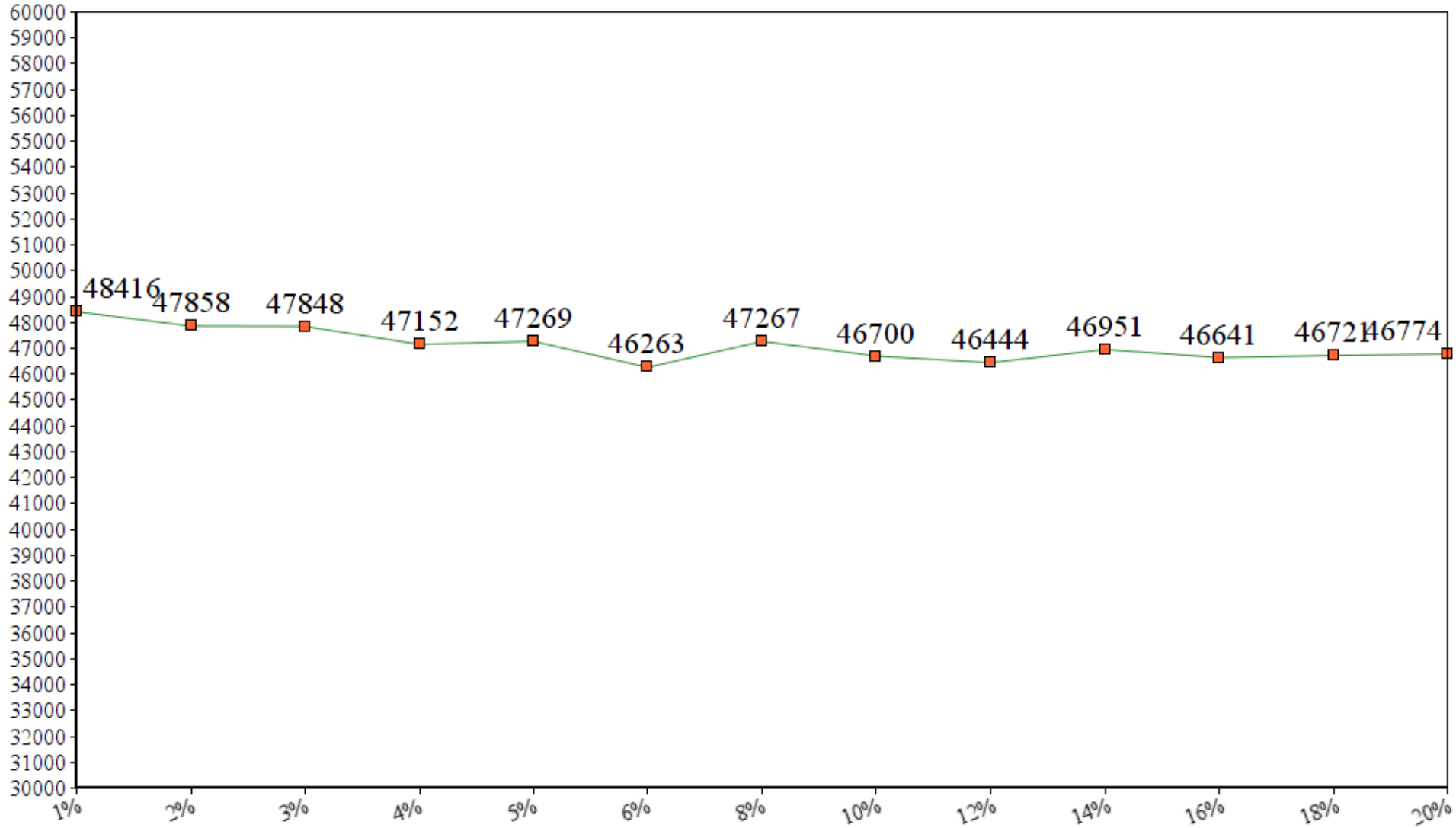
Nutzen der vorgeschlagenen Werte
Versuchen bessere Werte zu ermitteln

- Mutations-Rate = 3%
- Knoten-Anzahl = 1000
- maximale Generationsanzahl = 2000
- Funktions-Set = von Miller vorgeschlagenes (getestet)
- Output-Anzahl = 1
- levels-back-Parameter = -1
- (1+4)-Strategie

Jeden Versuch 60 mal ausgeführt

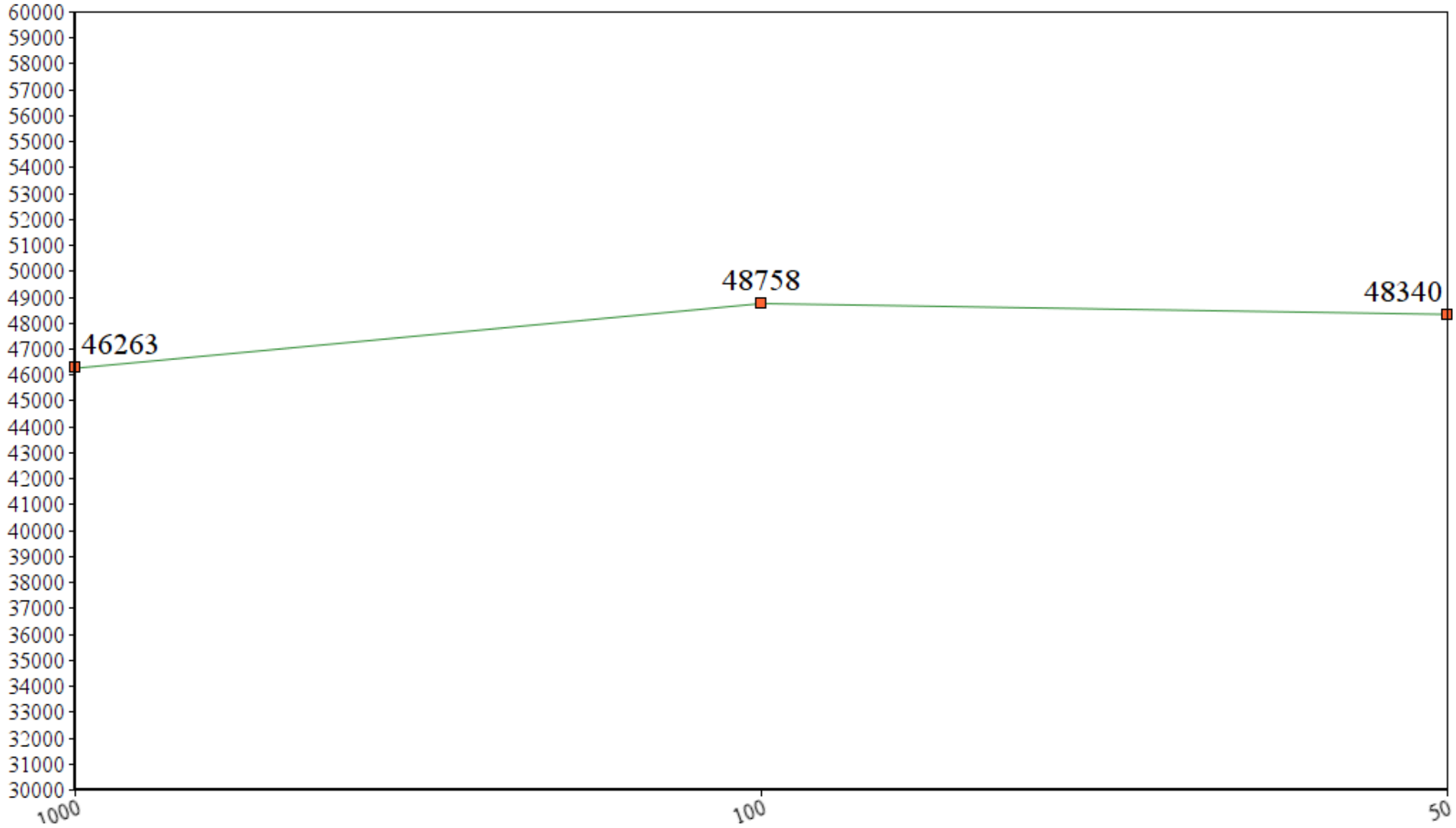
CGP - Mutationsrate

Knoten-Anzahl = 1000, Max.Generationen = 2000, levels-back = -1



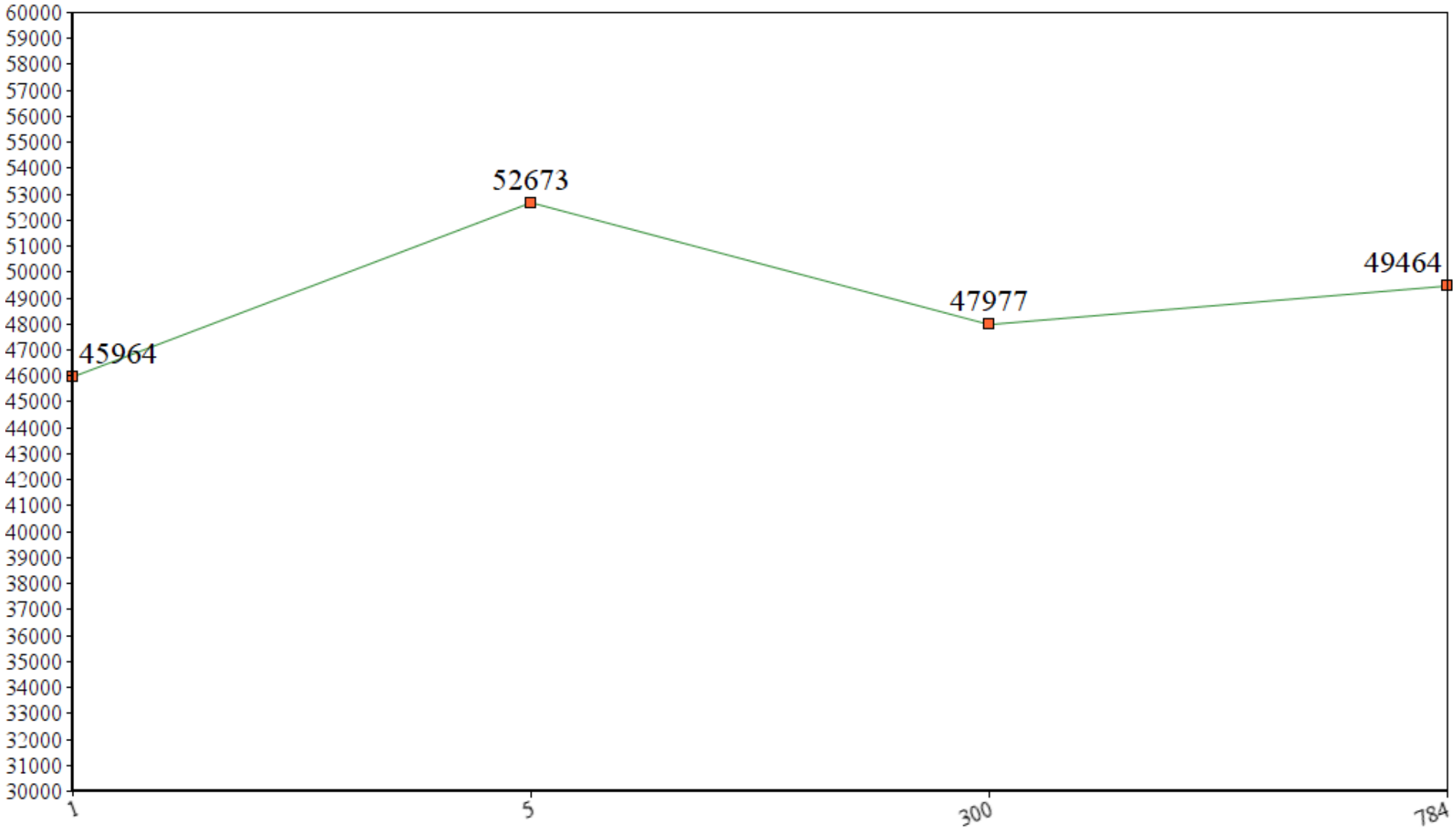
CGP - Knoten-Anzahl

MR = 6%, Max.Gen. = 2000, levels-back = -1



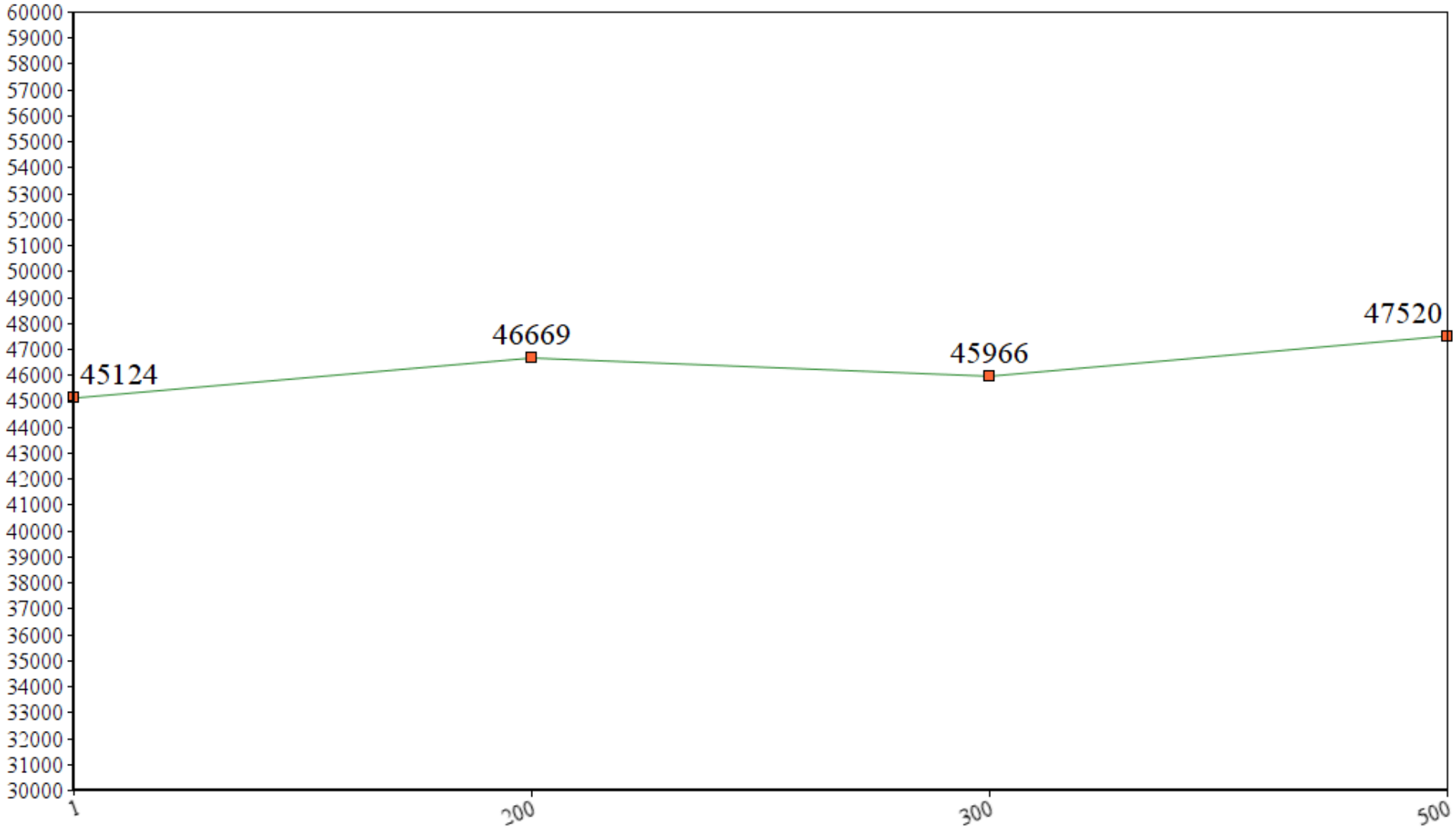
CGP - Levels-Back

MR = 6%, Max.Gen. = 2000, Knoten = 1000



CGP - Levels-Back

MR = 6%, Max.Gen. = 8000, Knoten = 1000



Die meisten aktiven Knoten immer bei keinem levels-back-Parameter

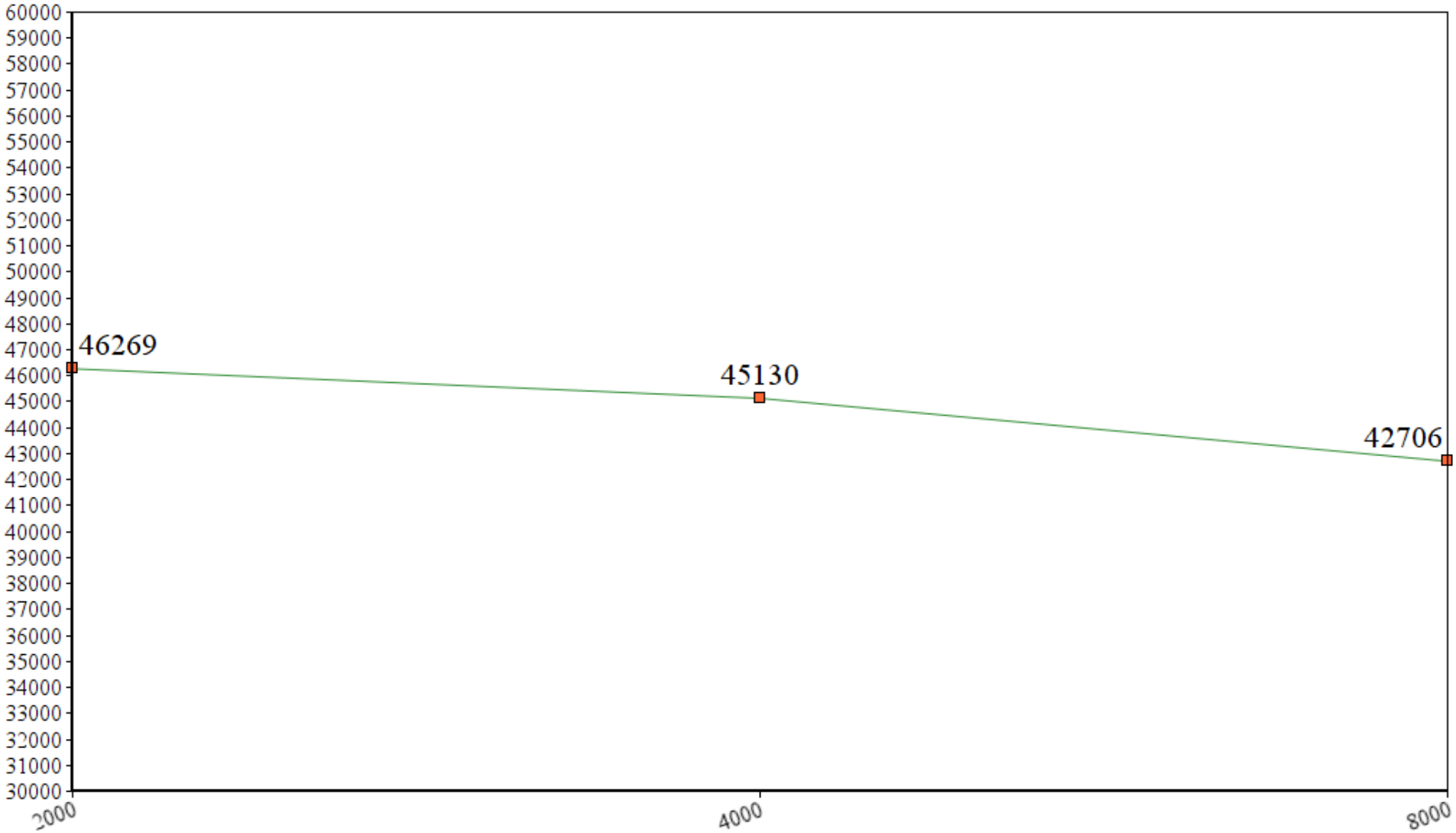
vorherige Bildbearbeitung

2 Verfahren:

- Verkleinern
  - Für 4 Pixel arithm. Mittel ausrechnen
  - Dies dann neuen Pixel zuweisen
  - 25%ige Größe
- in Schwarz-Weiß überführen
  - Treshold festlegen
  - Ist Pixel-Wert darüber dann 1
  - sonst ist der Pixel 0

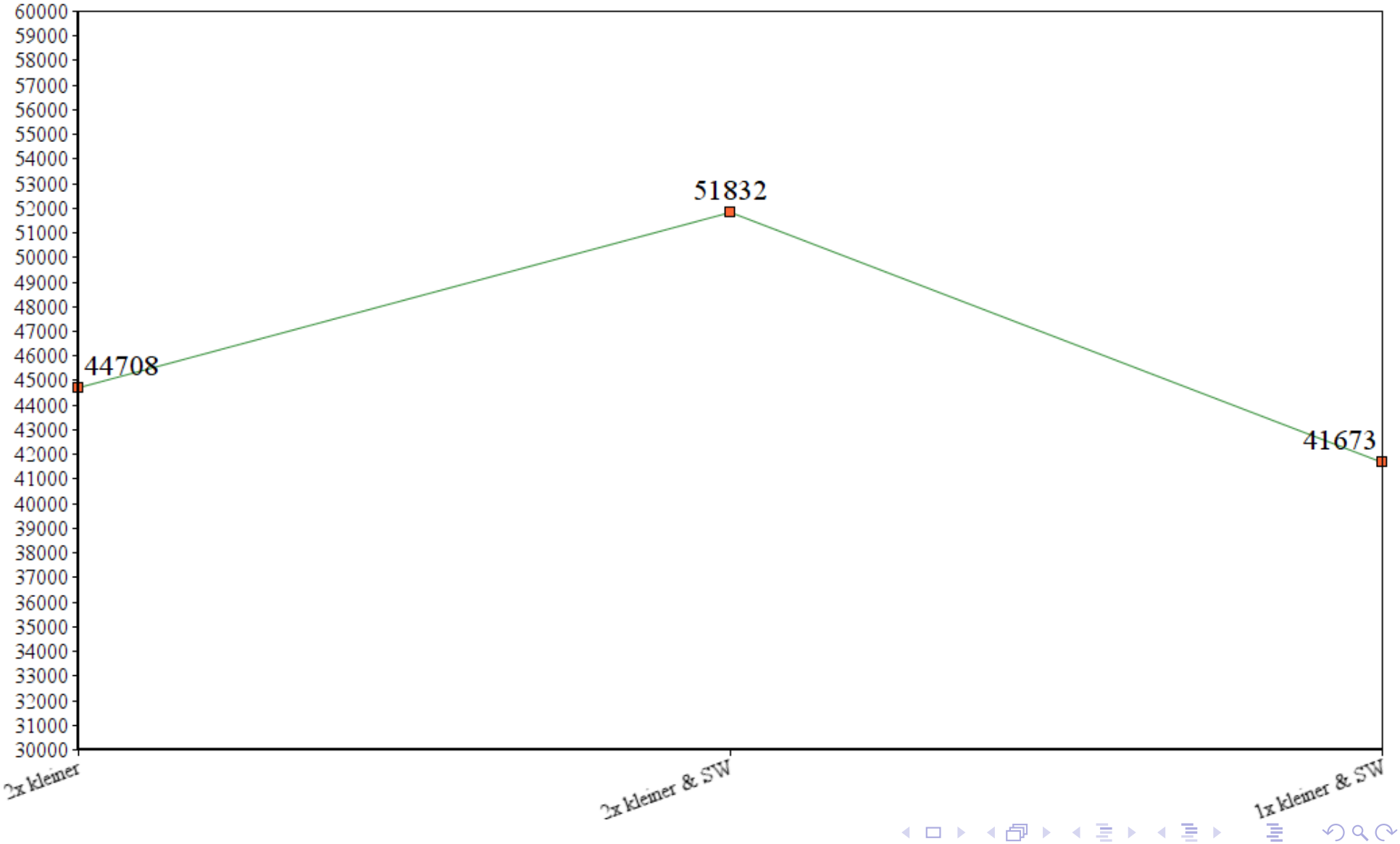
CGP - 1x Bild-Verkleinerung

MR = 10%, Knoten = 1000, levels-back = -1



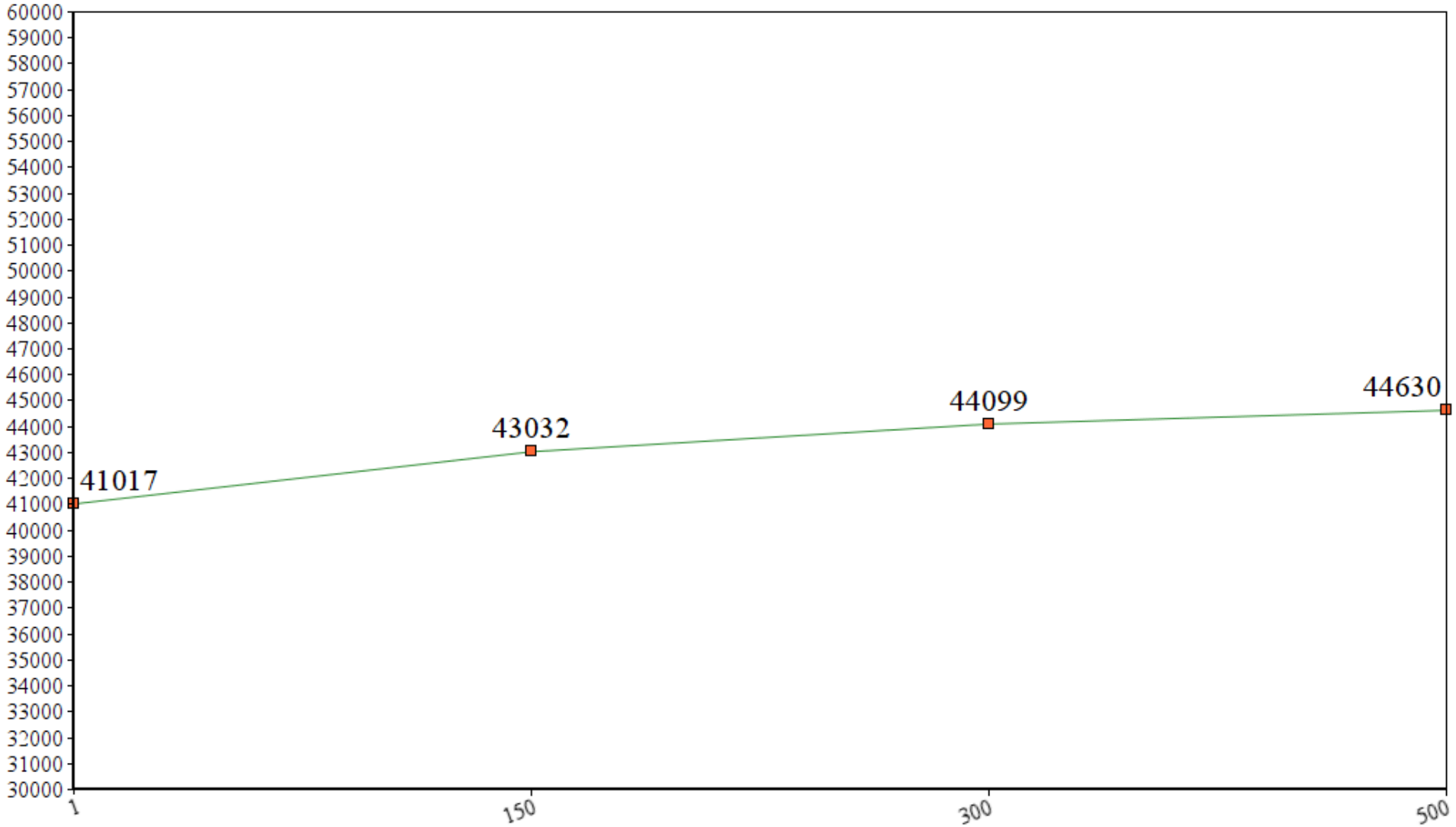
CGP - mehr Bild-Bearbeitung

MR = 10%, Max.Gen. = 4000, Knoten = 1000, levels-back = -1



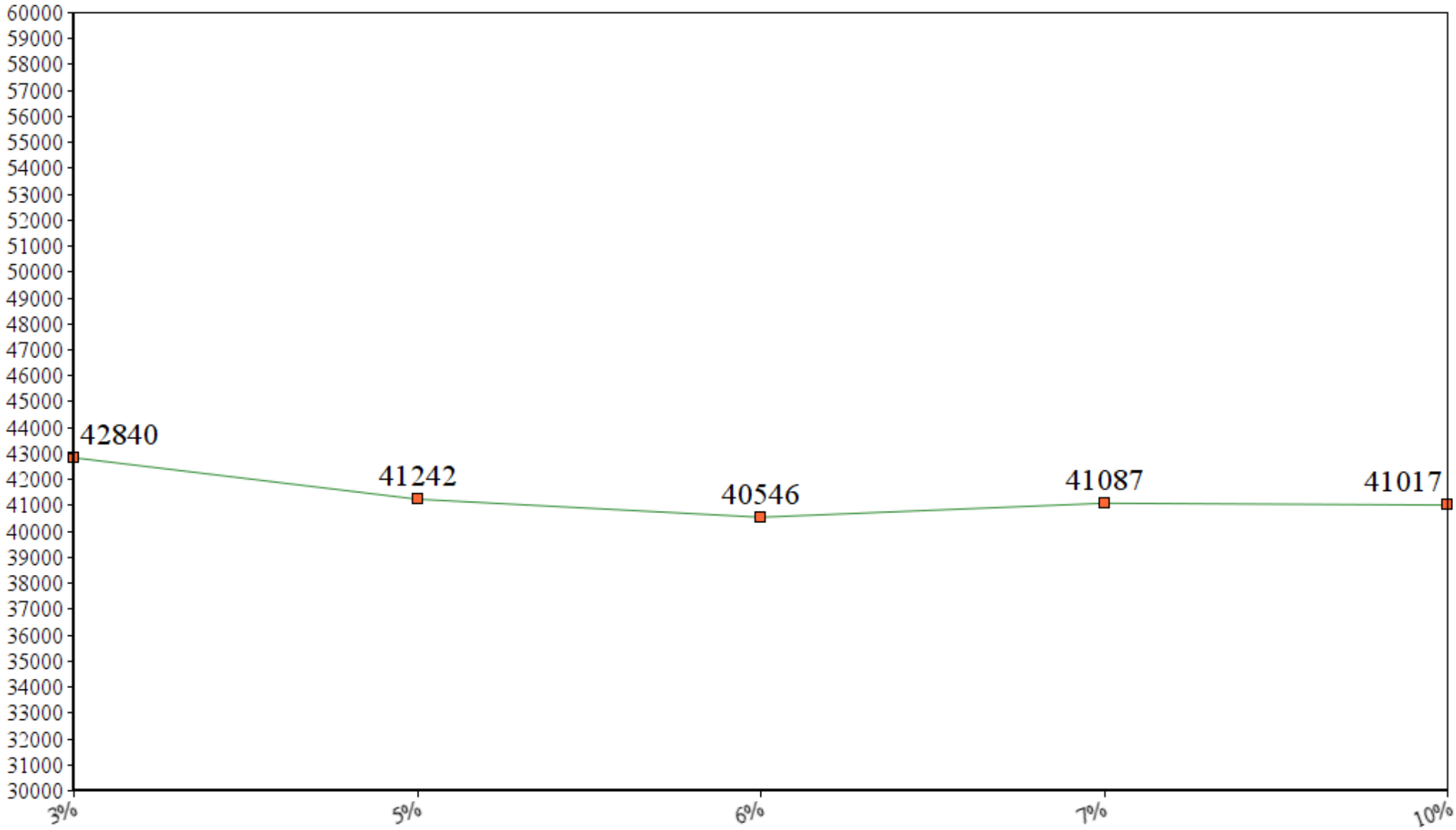
CGP - Levels-Back

MR = 10%, Max.Gen. = 10000, Knoten = 1000



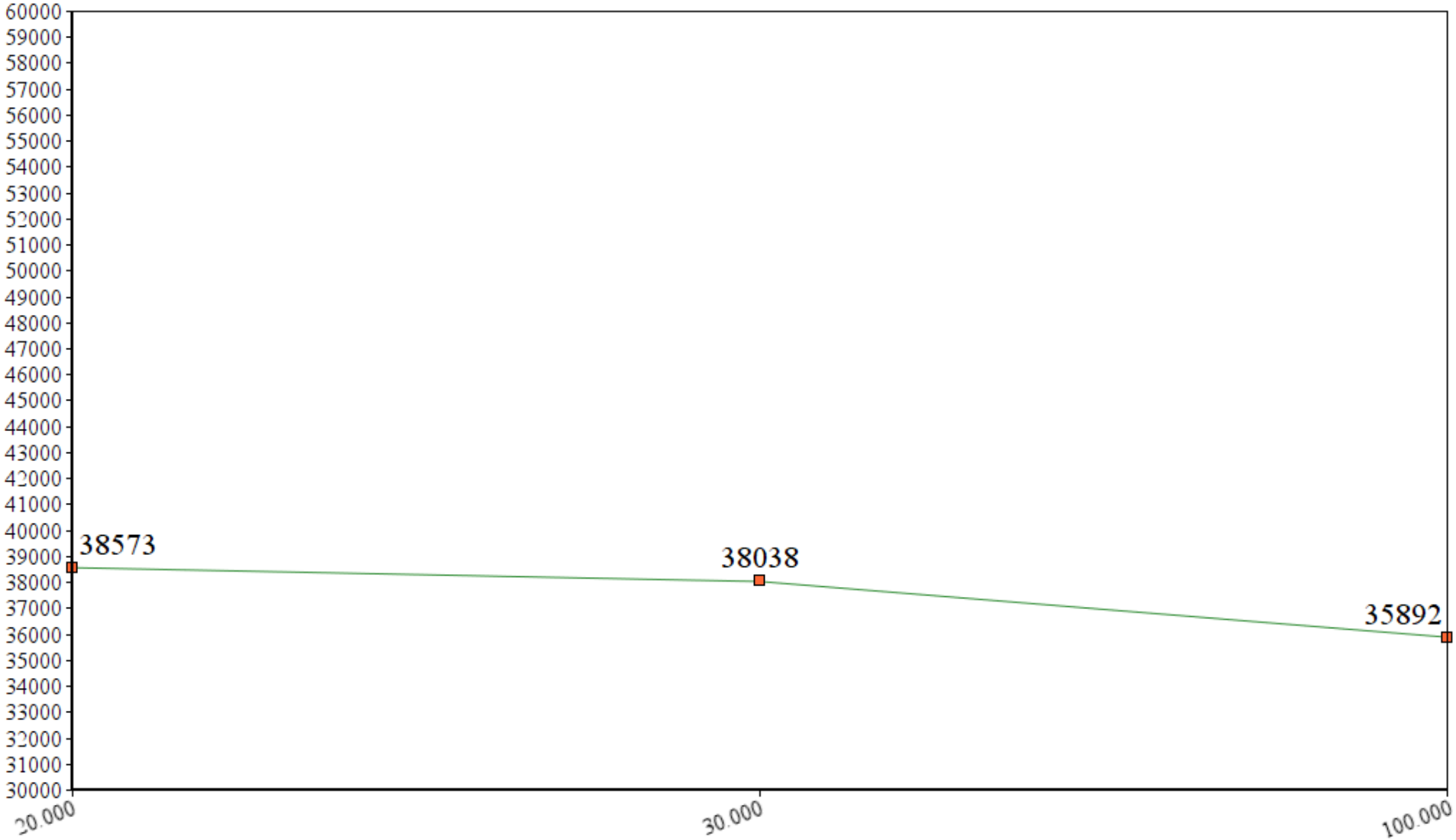
CGP - Mutations-Rate

Max.Gen. = 10000, Knoten = 1000, levels-back = -1



CGP - mehr Generationen

MR = 6%, Knoten = 1000, levels-back = -1



Bester Lauf hat Fitness von 29965

Klassifikator-Generierung

Generierung eines Klassifikators mithilfe von JavaPoet

- Erstellt aus einem Individuum eine ausführbare Klasse
- Für jeden Knoten eine Methode
- Speichert Ausgaben in Array ab
- gibt schnell Ergebnis für bestimme Eingabe aus

Embedded Cartesian Genetic Programming

Erweiterung von CGP

- Nutzt erstellbare Module
- Bestehend aus Knoten des Genotypen
- Können erstellt und zerstört werden
- Komplexer als CGP
- qualitativ bessere Ergebnisse mit weniger Generationen
- auch selber in Java implementiert (**PECGP**)

Parameter ECGP

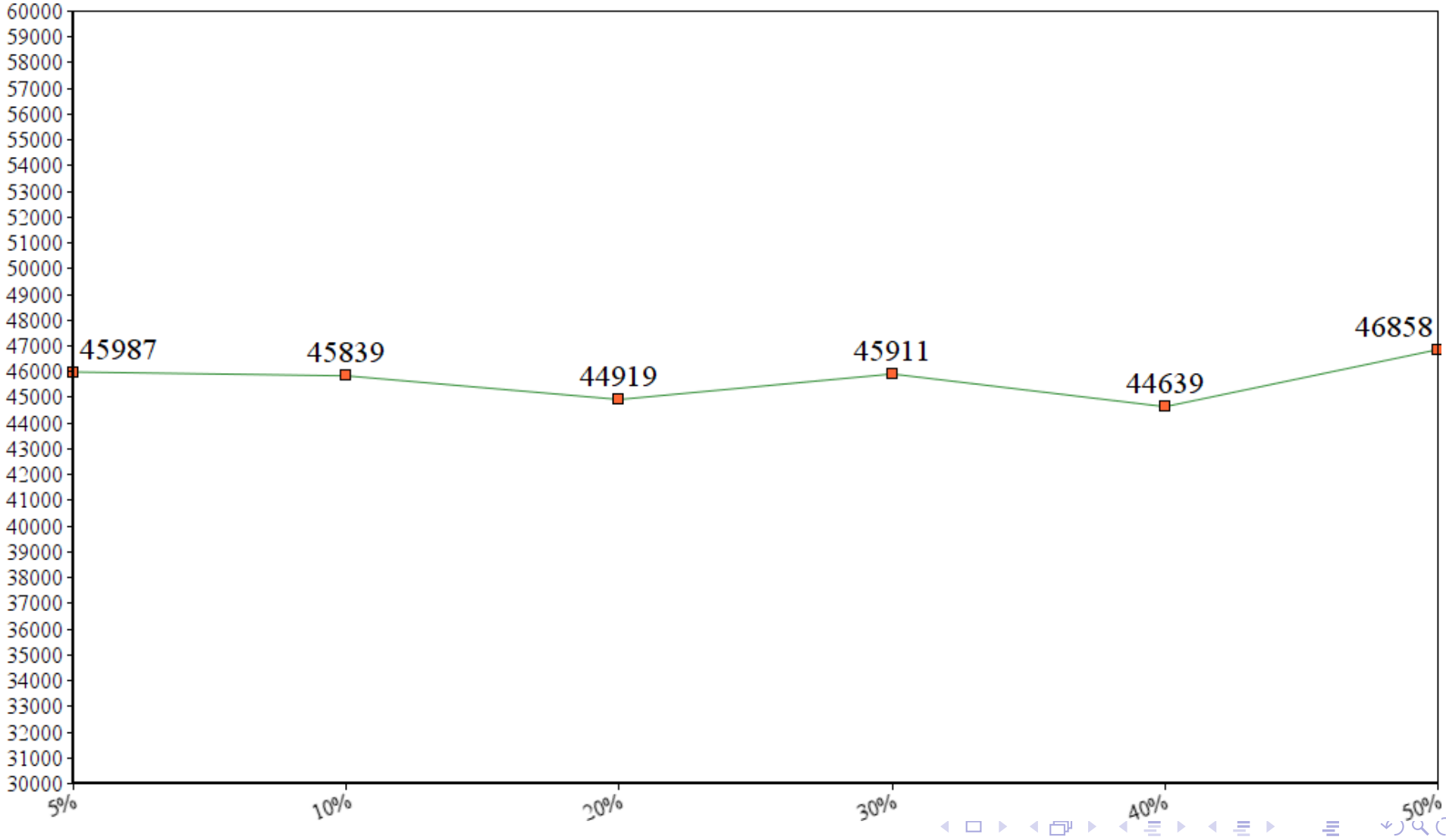
Zusätzlich zu denen von CGP:

- maximale Modul-Größe
- Compress-Wahrscheinlichkeit
- Modul-Punkt-Mutations-Wahrscheinlichkeit
- Add-Input-Wahrscheinlichkeit
- Add-Output-Wahrscheinlichkeit

Werte von CGP werden übernommen

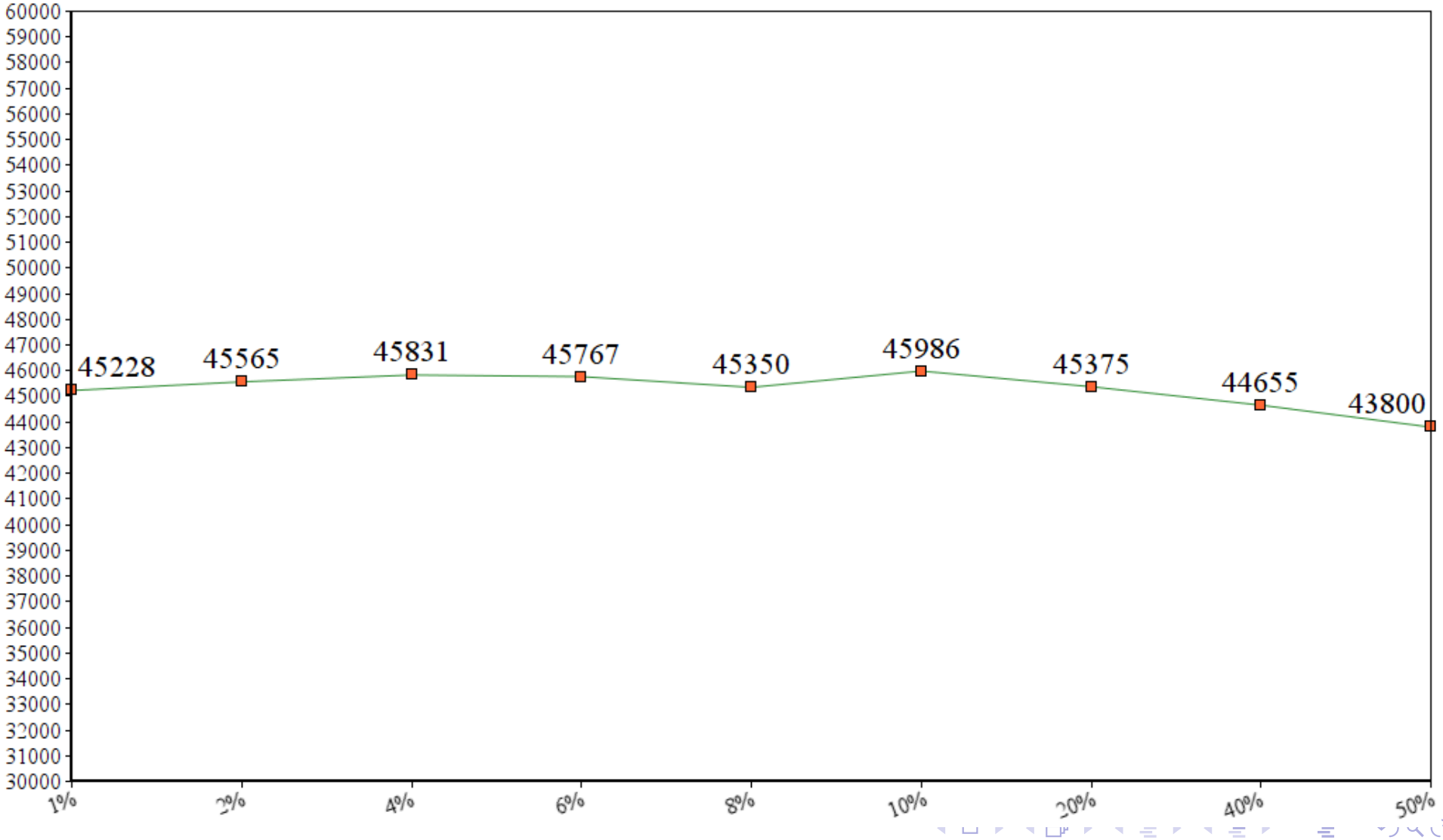
ECGP - Compress

Max.Gen.=2000, Mod.Punkt = 4%, AddIn = 1%, AddOut = 1%, max.Mod.Größe = 10



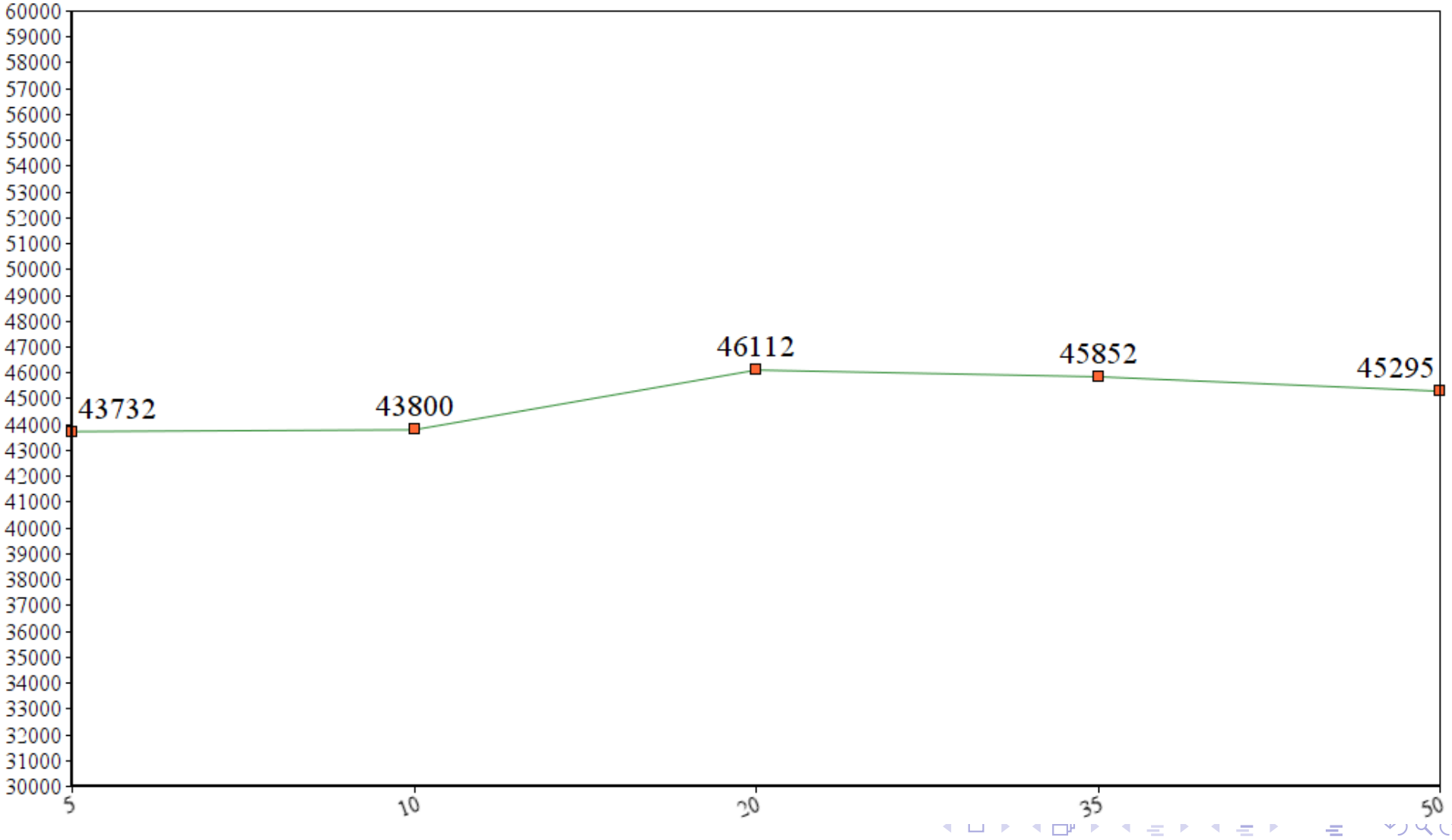
ECGP - Modul-Punkt-Mutation

Max.Gen.=2000, Compress = 20%, AddIn = 1%, AddOut = 1%, max.Mod.Größe = 10



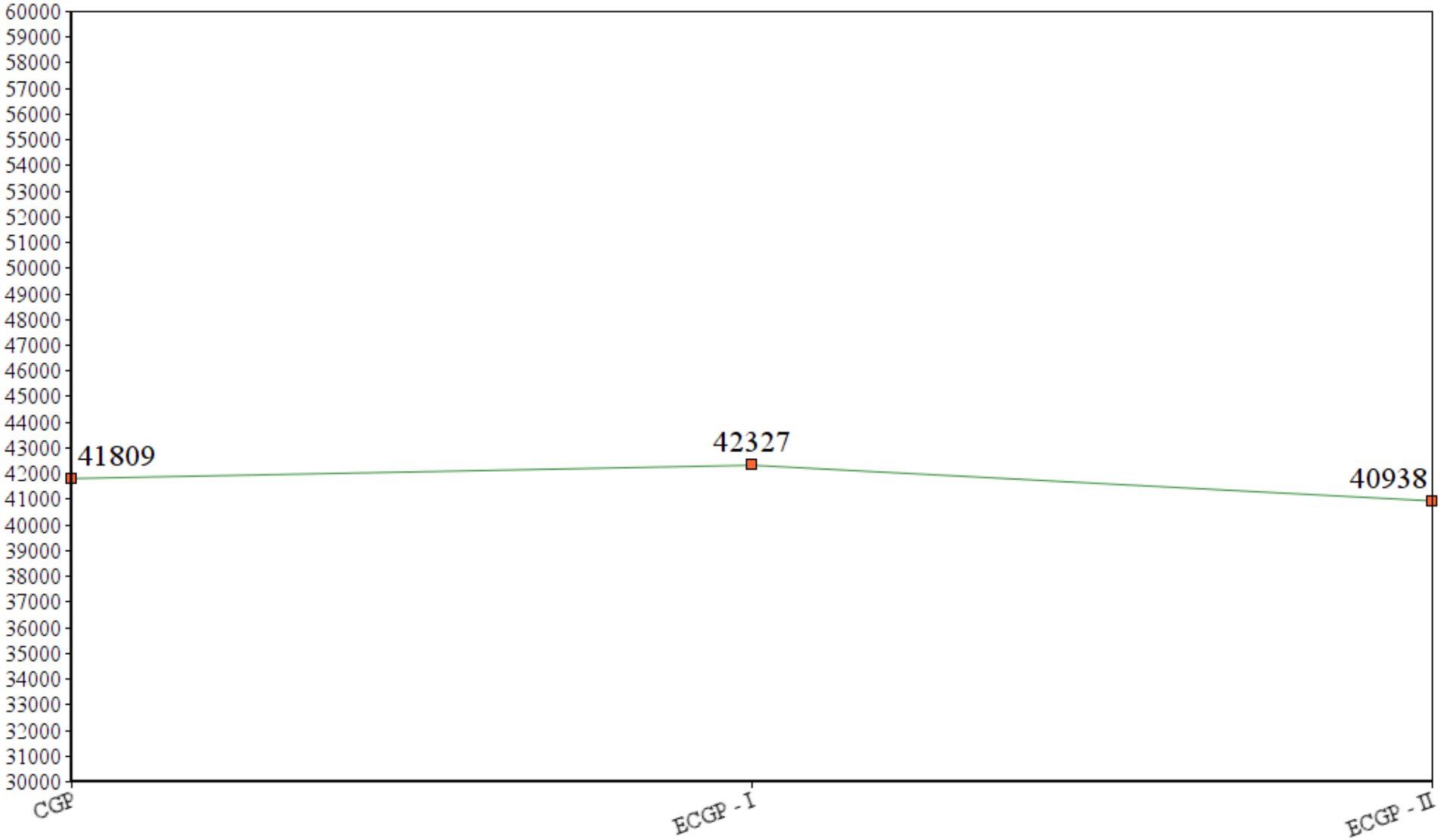
ECGP - max. Modul-Größe

Max.Gen.=2000, Compress = 20%,Mod.Punkt = 40%, AddIn = 1%, AddOut = 1%



Vergleich ECGP zu CGP

Max.Gen.=8000



Eher CGP als ECGP aufgrund Laufzeit

Fazit

- CGP kann gut genutzt werden um Ziffern zu klassifizieren
- Mehr als die Hälfte des Datensatzes konnte korrekt klassifiziert werden
- Fitness-Senkung von 47848 zu 35892
- ECGP qualitativ besser, aber längere Laufzeiten
- weitere Senkung möglich

Ausblick

Weitere Fitness-Senkung möglich durch:

- weitere Optimierung der Parameter
- speziellere Bildbearbeitung
- Entwicklung eines speziellen Funktions-Sets

Mit PCGP und PECGP möglich!

Quellen

- Miller, J.-F. (Herausgeber): Cartesian Genetic Programming. Springer-Verlag, Berlin Heidelberg, 2011.
- www.diagrammerstellen.de
- PCGP: bit.ly/2oRKU2j
- PECGP: bit.ly/2NYPl6s

Vielen Dank für Ihre Aufmerksamkeit