EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

MATH. - NATURWISS. FAKULTÄT
FACHBEREICH INFORMATIK
KOGNITIVE SYSTEME · PROF. A. ZELL

Evolutionäre Algorithmen

Übungsblatt 8, SS 2011

Abgabe: 14.06.11

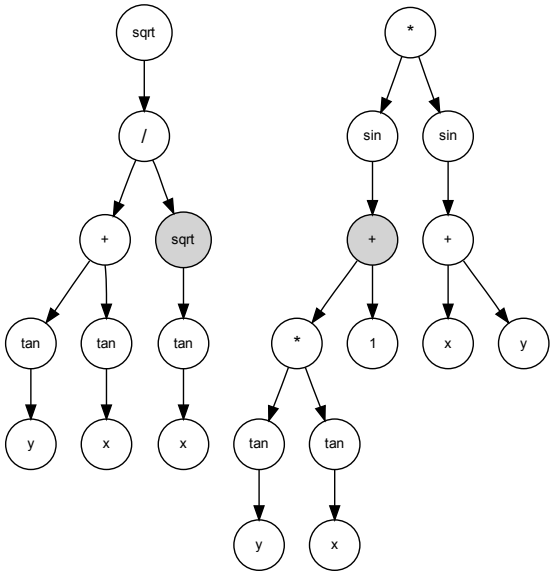
Aufgabe 21 Genetisches Programmieren: Baumstrukturen - (5 Punkte)

In einem gewurzelten Baum beschreibt die Tiefe k eines Knotens den Abstand zur Wurzel und die Tiefe d des Baumes die Länge des längsten Pfades zur Wurzel. Ein m-ärer Baum ($m \geqslant 2$) ist ein gewurzelter Baum, in dem jeder Knoten bis zu m Kindknoten besitzt. Ein vollständiger m-ärer Baum ist ein m-ärer Baum, der an jedem internen Knoten genau m Kinder hat und alle Blätter (Knoten ohne Kinder) dieselbe Tiefe haben.

(a) Zeigen Sie durch vollständige Induktion: Die maximale Zahl von Knoten mit Tiefe k in einem m-ären Baum ist m^k. (2 Punkte)

(b) Leiten Sie die minimale Zahl an Knoten in einem m-ären Baum der Tiefe d her.(2 Punkte)

(c) Gegegeben seien nebenstehende Bäume. Führen Sie an den grau hinterlegten Knoten ein Crossover durch. Geben Sie zu jedem Baum die zugehörige Infix-Notation an. (1 Punkt)



Aufgabe 22 GP-Bäume - (7 Punkte)

Es soll eine abstrakte Klasse `GPTree` entwickelt werden, die Programme als Baumstrukturen repräsentieren kann. Die Baumstruktur ist binär rekursiv angelegt, d. h. jeder `GPTree` enthält bis zu zwei Unterknoten, die wiederum Instanzen der Klasse `GPTree` sind. Jeder Knoten enthält insbesondere eine ganzzahlige Kennnummer (`type`), über die ihm eine spezifische Funktion zugeordnet werden kann, wobei `type` $= -1$ für den leeren Baum steht, der keine weiteren Informationen enthält.

Es gibt drei Arten von Knoten: Terminale sowie unäre und binäre Funktionen, welche entsprechend die Stelligkeit 0, 1 oder 2 haben. Beispielsweise kann der Term $+(\sin(-(x, y)), *(+(1, 1), x))$, der dem Ausdruck $\sin(x - y) + (1 + 1) * x$ entspricht, mit den Terminalen `term` $= \{1, x, y\}$, den unären Funktionen `unary` $= \{\sin, \cos\}$ und den binären Funktionen `binary` $= \{+, -, *, /\}$ durch `0(0(1(1,2)),2(0(0,0),1))` kodiert werden. Hierbei beginnt die Numerierung bei null.

Auf der Webseite der Übung ist ein Gerüst für `GPTree` vorgegeben. Zu implementieren sind ein Kon-

struktor, der einen zufälligen (nicht-leeren) Baum mit maximaler Tiefe d erzeugt, ein Copy-Konstruktor für die Replikation sowie Kreuzungs- und Mutationsoperatoren, welche die Konsistenz der Nachkommen beibehalten und die Tiefenbeschränkung beachten. Unvollständige Methoden sind mit dem Schlagwort `TODO` im Quellcode gekennzeichnet und mit Programmierhilfen versehen.

Da beim Initialisieren iterativ zufällige Knoten ausgewählt werden, muss entschieden werden, ob ein Terminal oder eine Funktion eingesetzt wird. Da die Funktionsbasis problemabhängig ist, wird diese nicht in `GPTree` spezifiziert, sondern über abstrakte Methoden verwendet.

Als Initialisierungs-Modi sind $full$ und $grow$ populär: Bei $full$ liegen alle Terminale auf Tiefe d, der Baum ist in diesem Sinne vollständig. Bei $grow$ wird an jedem Knoten gewürfelt, ob ein Terminal oder ein Nicht-Terminal eingesetzt wird, wodurch variable Strukturen entstehen.

Testen Sie Ihren Code – am besten mit dem `XORGPIndividual` aus der nächsten Aufgabe. Hinweis: Zu Testzwecken kann es sinnvoll sein, in der Klasse `EvolutionaryAlgorithm` nach einer `TODO`-Marke zu suchen.

Aufgabe 23 XOR-Funktion mit GP - (8 Punkte)

Gegeben sei die XOR-Funktion $f_{\mathrm{XOR}}(x) = x_1 \oplus x_2 \oplus \cdots \oplus x_n$ für $n \in \{2, 3, 4\}$ und $x_i \in \{0, 1\}$. Ein GP-Programm soll optimiert werden, so dass es die Funktion möglichst gut approximiert.

Gegeben ist die Klasse `XORGPIndividual`, die `GPIndividual` ableitet und `InterfaceIndividual` implementiert. `XORGPIndividual` verwendet die Darstellung von Bäumen aus Aufgabe 22, wobei `GPIndividual` die abstrakte Klasse `GPTree` erweitert. Als Terminale dienen $\{x_1, \ldots, x_n\}$, als unäre Funktionen $\{\mathrm{NOT}\}$ und als binäre Funktionen $\{\mathrm{IMP}, \mathrm{OR}\}$. IMP steht hier für die Implikation $\rightarrow$ mit $a \rightarrow b \iff (\neg a) \vee b$. Vervollständigen Sie die Klasse `XORGPIndividual`, indem Sie die Methoden `evalTreeAt(Object o)` und `evaluate()` implementieren (siehe Hinweise im Quellcode).

Sie können die Eingabe x als n-Bit-Binärzahl kodieren, um das Programmieren zu erleichtern. Zur Fitnessberechnung können Sie z. B. alle möglichen Eingaben auswerten und die Anzahl der Fehler gegenüber der Funktion f_{XOR} zählen. Ist Ihr Algorithmus mit einer solch einfachen Fitnessfunktion in der Lage, auch für $n = 4$ das Optimum zu finden? Versuchen Sie gegebenenfalls die Fitnessfunktion zu verbessern, um das Optimum zu erreichen. Um die Lösung zu finden, müssen Sie natürlich hinreichend große Baumtiefen zulassen. Bei n Eingaben reicht Baumtiefe $d = n + 2$. Testen Sie auch mit großen Populationen und kurzer Laufzeit, und variieren Sie das Verhältnis von Crossover- zu Mutationswahrscheinlichkeit.

Allgemeiner Hinweis: Abgabe per E-Mail (`andreas.jahn@uni-tuebingen.de`) ist erwünscht, besonders für die Programmieraufgaben. Bitte die Klassen vorher testen, da Syntaxfehler zu Punktabzug führen. Die Klassen sollten ohne irgendwelche Extras mit dem JDK 1.6 laufen. Bitte grundsätzlich keine grafischen Oberflächen programmieren, denn dies kostet meist deutlich mehr Zeit als geplant. Wer es dennoch für unerlässlich hält, möge ausschließlich das AWT und Swing verwenden.