Methodology

JGAP, uitgesproken als 'jee-gep', is een framework voor het implementeren van genetische algoritmen en het gebruik ervan in Java. Genetische algoritmen zijn voor velen onbekend terrein, maar een krachtig instrument voor het oplossen van bepaalde problemen. We geven een introductie tot genetische algoritmen en beschrijven de toepassingen daarvan. We laten zien hoe het JGAP-framework kan worden ingezet om problemen op te lossen met genetische algoritmen. Dit wordt gedaan aan de hand van een concreet voorbeeld.

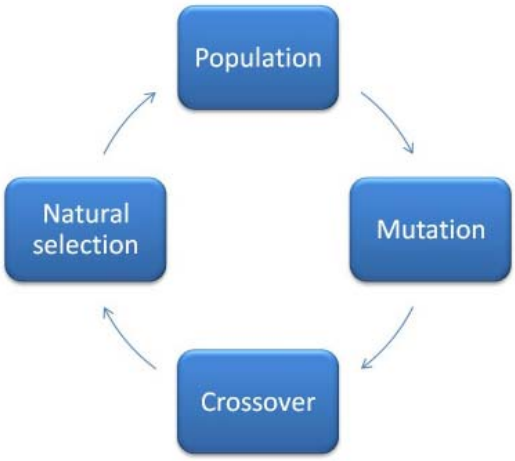
Genetische algoritmen met JGAP

Krachtige techniek om problemen op te lossen

Sommige problemen zijn zo complex en het aantal mogelijke oplossingen zo groot, dat het onmogelijk is om deze problemen met een wiskundig algoritme op te lossen. Met brute kracht simpelweg alle mogelijke oplossingen proberen kost zoveel tijd dat deze aanpak niet realistisch is. Voor dit soort problemen is een alternatieve aanpak nodig. Eén van deze alternatieven is het gebruik van genetische algoritmen.
Een genetisch algoritme is een zoekalgoritme welke uit een enorme hoeveelheid potentiële oplossingen, (in veel gevallen) een optimale oplossing weet te vinden. De verzameling van alle mogelijke oplossingen wordt zoekgebied genoemd. Een genetisch algoritme vindt de oplossing voor een bepaald probleem door iteratief een initiële, willekeurig gekozen, verzameling van oplossingen te muteren tot nieuwere, potentieel betere oplossingen. Genetische algoritmen zijn geïnspireerd op genetische operaties welke in de natuur voorkomen, zoals mutatie en crossover, en het Darwinian principe van natuurlijke selectie.

Een potentiële oplossing wordt een chromosoom genoemd. Een chromosoom bestaat uit een constant aantal genen. Een gen is een uniek, individueel, onderdeel van een chromosoom. Een aantal chromosomen bij elkaar wordt een populatie genoemd. De populatie kan dus worden gezien als een verzameling van mogelijke oplossingen. De chromosomen in de populatie worden met behulp van een iteratief proces omgevormd tot nieuwe chromosomen. Dit iteratieve proces wordt de evolutie genoemd. In elke iteratie van de evolutie wordt gebruik gemaakt van mutatie en crossover. Mutatie is het muteren van de individuele genen van een chromosoom. Crossover is het combineren van bestaande chromosomen tot nieuwe chromosomen. Mutatie en crossover leveren beide nieuwe chromosomen op.

Nadat mutatie en crossover zijn toegepast op de chromosomen in de populatie, wordt er bepaald welke chromosomen meedoen in de volgende iteratie van de evolutie. Om dit te bepalen wordt het mechanisme van natuurlijke selectie toegepast. Dit mechanisme is gebaseerd op de zogenaamde fitheid van een chromosoom. De fitheid geeft aan hoe goed



Figuur 1: Het evolutieproces.

Jamie Craane
is software architect en Java competence manager bij QNH Business Integration.

Ideaal om problemen in de planning op te lossen

een bepaald chromosoom is ten opzichte van andere chromosomen. Chromosomen met een betere fitheid hebben een grotere kans om door te gaan naar een volgende iteratie in de evolutie dan chromosomen met een lagere fitheid. Door het proces van mutatie, crossover en selectie te herhalen op de chromosomen in de populatie, ontstaat er uiteindelijk een populatie met een (bijna) optimaal chromosoom (oplossing) voor het betreffende probleem. Dit chromosoom heeft dan de beste fitheid ten opzichte van alle andere chromosomen in de populatie. Het evolutieproces is weergegeven in figuur 1.

Doordat genetische algoritmen efficiënt zijn in het gericht zoeken naar oplossingen uit een enorme hoeveelheid potentiële oplossingen, zijn ze geschikt voor het oplossen van bepaalde soorten problemen. Een voorbeeld van een dergelijk probleem is een planningprobleem. In een fabriekshal vinden we een typisch voorbeeld van een planningprobleem. In de fabriekshal staan machines waarop taken in een bepaalde volgorde en tijd uitgevoerd worden. Deze taken draaien ook weer op specifieke machines welke door bepaalde personen worden bestuurd. Afhankelijk van het aantal te plannen taken kan het aantal potentiële oplossingen enorm zijn. In tegenstelling tot een heuristisch algoritme, welke zich op één enkele oplossing concentreert, opereren genetische algoritmen op een hele verzameling van oplossingen, de populatie. Doordat uit elke iteratie van de evolutie de beste oplossingen overblijven, die gebruikt worden voor het vormen van nieuwe oplossingen, worden er al snel acceptabele oplossingen gevonden.

Een ander voorbeeld van een probleem waarbij genetische algoritmen succesvol zijn, is het travelling salesman problem (TSP) en varianten daarop. Dit probleem luidt als volgt: gegeven een lijst van steden, wat is het kortst mogelijke pad om al deze steden maar één keer te bezoeken? Ook bij dit probleem geldt, hoe meer steden hoe groter het aantal mogelijke oplossingen.

Voorbereiding

Voordat er met het implementeren van het genetische algoritme wordt begonnen, moet er vooraf een aantal stappen worden doorlopen die de context van het probleem bepalen. Deze stappen zijn altijd noodzakelijk wanneer een genetisch algoritme wordt ingezet voor het oplossen van een bepaald probleem. Deze stappen zijn:

1. Het bepalen van de genetische representatie van de oplossing. In deze stap wordt bepaald hoe een chromosoom, dat bestaat uit individuele genen, is opgebouwd. In het concrete voorbeeld wordt hier dieper op ingegaan.
2. Het bepalen van de fitnessfunctie. De fitheid bepaalt hoe goed een bepaald chromosoom is ten opzichte van andere chromosomen. Het definiëren van een goede fitnessfunctie is het belangrijkste onderdeel van de voorbereiding. Wanneer de fitnessfunctie niet goed is gedefinieerd, is de kans groter dat er veel minder goede chromosomen worden gekozen voor de volgende iteratie in de evolutie. Op deze manier ontstaat er nooit een optimale oplossing voor het betreffende probleem.
3. Het bepalen van de run parameters van de evolutie. De run parameters bepalen het gedrag van de evolutie. Parameters waar aan gedacht moet worden zijn de initiële grootte van de populatie en het aantal iteraties in de evolutie die uitgevoerd worden.
4. Het bepalen van de succescriteria. De succescriteria bepalen wanneer de evolutie van het genetische algoritme eindigt. Dit kan bijvoorbeeld een vooraf vastgesteld aantal iteraties van de evolutie zijn. De run kan ook eindigen wanneer aan een bepaalde eis, die is gesteld aan de fitheid van een chromosoom, is voldaan. De run is bijvoorbeeld geslaagd wanneer de fitheid van een chromosoom groter is dan 50.

JGAP

Java Genetic Algorithms Package, kortweg JGAP, implementeert de concepten van genetische algoritmen als een Java-framework. De genetische concepten zijn in JGAP geïmplementeerd als afzonderlijke objecten. Mede hierdoor is de stap van de theorie naar de praktijk goed te begrijpen. De belangrijkste classes wanneer wordt gewerkt met genetische algoritmen zijn de volgende:

- **IChromosome**

De IChromosome interface representeert een potentiële oplossing en bestaat uit een vast aantal genen.

- **Gene**

De Gene interface is een onderdeel van een chromosoom. Elke Gene heeft een waarde. De waarde van een Gene wordt allele genoemd welke met de getAllele() methode geretourneerd wordt. JGAP biedt een aantal implementaties van de Gene interface.
Voorbeelden van deze implementaties zijn:
a) IntegerGene. Dit is een implementatie van Gene waarbij de allele een Integer is.
b) BooleanGene. Dit is een implementatie van Gene waarbij de allele een Boolean is.
c) StringGene. Dit is een implementatie van Gene waarbij de allele een String is.
Verder is het mogelijk om zelf nieuwe implementaties van de Gene interface te maken. Dit wordt gedaan door de class BaseGene te extenden. Let er hierbij wel op dat de equals en hashCode() methode geïmplementeerd worden!

• **GenoType**
De GenoType is de populatie en bestaat uit een aantal chromosomen.

• **FitnessFunction**
De fitnessfunctie bepaalt hoe goed een bepaald chromosoom is ten opzichte van andere chromosomen. De baseclass FitnessFunction moet subclassed worden waarbij de evaluate(IChromosome a_subject) methode geïmplementeerd moet worden. Deze methode wordt voor elke chromosoom in de populatie aangeroepen en berekent de fitheid van de meegegeven IChromosome. De fitheid wordt geretourneerd als een double. De absolute waarde van de fitheid is hierbij niet van belang. Het gaat om het relatieve verschil tussen de fitheid van de chromosomen in de populatie.

• **GAConfiguration**
Dit object bevat de configuratiedata voor het genetische algoritme. Voorbeelden van configuratiedata zijn de grootte van de initiële populatie en de implementatie van de fitness function. Veel parameters hebben default waarden.

Concreet voorbeeld

De beste manier om te leren is door gebruik te maken van voorbeelden. Het voorbeeld dat hier is beschreven is een variant op het bin-packing probleem. Het probleem luidt als volgt: Een verhuisbedrijf is gespecialiseerd in het verplaatsen van spullen van de ene locatie naar de andere. Deze spullen worden in dozen verpakt. Deze dozen hebben allemaal een verschillend volume. Om kosten te besparen is het voor het verhuisbedrijf noodzakelijk om zijn verhuiswagens zo optimaal mogelijk in te zetten. Dit betekent dat er niet meer verhuiswagens mogen worden gebruikt voor het vervoeren van de dozen dan strikt noodzakelijk. De dozen worden in dit voorbeeld gerepresenteerd door een array van Box instanties. De Box class representeert een individuele doos met een volume. Codevoorbeeld 1a laat de initialisatie van de dozen zien:

```
private void createBoxes(int seed) {
     // De seed maakt het mogelijk de initialisatie te
reproduceren
    Random r = new Random(seed);
    this.boxes = new Box[125];
    for (int i = 0; i < 125; i++) {
          Box box = new Box(0.25 + (r.nextDouble() *
2.75));
         box.setId(i);
         this.boxes[i] = box;
    }
}
```

Codevoorbeeld 1a.

In codevoorbeeld 1a worden 125 dozen met een volume variërend tussen 0.25 en 3 kubieke meter gecreëerd. Deze dozen worden in busjes geplaatst met een inhoud van 4.33 kubieke meter. De busjes worden gerepresenteerd door de Van class. De volgorde van de dozen in de Box array, is de volgorde waarop de dozen in de busjes worden geplaatst. Het doel van het genetische algoritme is het vinden van de optimale volgorde waarop de dozen in de busjes worden geplaatst. Dit betekent dat het aantal benodigde busjes zo klein mogelijk moet zijn. De dozen uit codevoorbeeld 1a hebben een gezamenlijk volume van 210.25 kubieke meter, welke gegenereerd zijn met een seed van 37. Het minimum aantal benodigde busjes is 210.25 / 4.33 = 48.5. Aangezien er geen halve busjes gebruikt kunnen worden, wordt het resultaat naar boven afgerond op 49. Als we de dozen in de busjes plaatsen in de volgorde waarop de Box array initieel geïnitialiseerd is, zijn er 63 busjes nodig! Dit is beduidend meer dan het optimale aantal van 49. Het aantal mogelijke verschillende combinaties, het zoekgebied, is 125! (faculteit van het aantal dozen) = 1,88e+209. Een perfecte kandidaat voor een genetisch algoritme.

Als eerste moeten we de genetische representatie van de oplossing bepalen. De oplossing bestaat in dit voorbeeld uit een lijst van array indices, de volgorde waarop de dozen in de busjes worden geplaatst. De oplossing kunnen we in JGAP representeren als een chromosoom met 125 IntegerGene's. Elke IntegerGene krijgt een waarde uit de range 0 t/m 124, de indices uit de Box array. Elke IntegerGene representeert dus een Box in de Box array.
Nu we de genetische representatie hebben bepaald, moeten we de fitnessfunctie implementeren. De fitnessfunctie voor dit probleem is weergegeven in codevoorbeeld 1b.

```
protected double evaluate(IChromosome a_subject) {
    double wastedVolume = 0.0D;
    double volumeInCurrentVan = 0.0D;
    int numberOfVansNeeded = 1;

    Gene[] genes = a_subject.getGenes();
    for (Gene gene : genes) {
        int index = (Integer) gene.getAllele();
       if (enoughSpaceAvailable(volumeInCurrentVan, this.
boxes[index])) {
                volumeInCurrentVan += this.boxes[index].
getVolume();
        } else {
            numberOfVansNeeded++;
                wastedVolume += Math.abs(vanCapacity -
volumeInCurrentVan);
         // Make sure we put the box which did not fit in
this van in the next van
                volumeInCurrentVan = this.boxes[index].
getVolume();
        }
    }

    return wastedVolume * numberOfVansNeeded;
}
```

Codevoorbeeld 1b.

De fitnessfunctie in codevoorbeeld 1b berekent hoeveel busjes er nodig zijn voor de dozen waarvan de volgorde wordt gerepresenteerd door het meegegeven chromosoom. Van elke Gene in het chromosoom

wordt de waarde opgevraagd die een index in de Box array representeert. Er wordt bepaald hoeveel dozen er in een busje passen. Wanneer er geen dozen meer in het busje passen, wordt het overgebleven volume opgeteld bij het totaal aan overgebleven volume. Hoe meer volume verspild wordt, hoe hoger de fitness waarde. Als laatste wordt de fitness waarde vermenigvuldigd met het aantal busjes dat nodig is voor dit chromosoom. Hoe meer busjes hoe hoger de fitness waarde. In dit voorbeeld geldt: hoe lager de fitness waarde hoe beter de oplossing.

Nu we de genetische representatie en de fitnessfunctie hebben bepaald, is het tijd om de run parameters en de succescriteria te bepalen. Het aantal evolutie iteraties dat we uitvoeren stellen we vast op 2500. Dit aantal is gekozen op basis van een aantal experimentele runs van het algoritme. De initiële grootte van de populatie zetten we op 50. Dit betekent dat er initieel 50 chromosomen gecreëerd worden. De evolutie eindigt wanneer er 2500 iteraties zijn uitgevoerd óf wanneer het optimale aantal benodigde busjes van 49 is bereikt. Codevoorbeeld 1c toont de evolutie:

```
private void evolve(Genotype a_genotype) {
    // 4.33 is the volume of an individual van
      int optimalNumberOfVans = (int) Math.ceil(this.
totalVolumeOfBoxes / 4.33);

            double previousFittest = a_genotype.
getFittestChromosome().getFitnessValue();
    int numberOfVansNeeded = Integer.MAX_VALUE;
    for (int i = 0; i < 2500; i++) {
        a_genotype.evolve();
     double fittness = a_genotype.getFittestChromosome().
getFitnessValue();
      int vansNeeded = this.numberOfVansNeeded(a_genotype.
getFittestChromosome().getGenes()).size();
         if (fittness < previousFittest && vansNeeded <
numberOfVansNeeded) {
                        this.printSolution(a_genotype.
getFittestChromosome());
             previousFittest = fittness;
             numberOfVansNeeded = vansNeeded;
        }

        // Optimal solution is reached
         if (numberOfVansNeeded == optimalNumberOfVans)
{
             break;
        }
    }
             IChromosome fittest = a_genotype.
getFittestChromosome();
    // Code omitted to print solution
}
```

Codevoorbeeld 1c.

Codevoorbeeld 1c berekent als eerste het optimaal aantal benodigde busjes. Wanneer het genetische algoritme dit aantal bereikt, stopt de evolutie. Er worden maximaal 2500 evolutie iteraties uitgevoerd. Tijdens elke iteratie wordt bepaald of het beste chromosoom in de populatie beter is dan de tot dan toe gevonden chromosoom. Indien dit zo is dan tonen we de betreffende oplossing. De output van het genetische algoritme is te zien in codevoorbeeld 1d.

```
The total volume of the [125] boxes is [210.25989987666645]
cubic metres.
The optimal number of vans needed is [49]
Fitness value [4123.992085987977]
The total number of vans needed is [63]
Fitness value [3458.197333300851]
The total number of vans needed is [61]
...Some output omitted
Fitness value [774.5352329142294]
The total number of vans needed is [51]
Fitness value [535.1696373214758]
The total number of vans needed is [50]
Fitness value [307.8713731063958]
The total number of vans needed is [49]
computation time = 23922
Van [1] has contents with a total volume of
[4.204196540671411] and contains the following boxes:
Box:0, volume [2.2510465109421443] cubic metres.
Box:117, volume [1.9531500297292665] cubic metres.
Van [2] has contents with a total volume of
[4.185233471369987] and contains the following boxes:
Box:17, volume [1.0595047801111055] cubic metres.
Box:110, volume [0.5031165156303853] cubic metres.
Box:26, volume [2.6226121756284955] cubic metres.
...Further output omitted
```

Codevoorbeeld 1d.

Zoals te zien in codefragment 1d bereikt het genetische algoritme de optimale verdeling van de dozen zodat deze in 49 busjes passen. De fitnesswaarde van deze oplossing is 307.87. De gemiddelde tijd die het algoritme nodig heeft om de optimale oplossing te vinden ligt gemiddeld tussen de 20 en 25 seconden. Ter vergelijking: bij een aantal van 20 dozen heeft een brute-force algoritme circa 30 seconden nodig om tot de optimale oplossing te komen. Een genetisch algoritme heeft, op dezelfde systeemconfiguratie, maar 31 milliseconden nodig om tot de optimale oplossing te komen bij hetzelfde aantal dozen!

De volledige sourcecode van dit voorbeeld is te vinden op Google code, zie hiervoor de referenties.

Conclusie

Genetische algoritmen is een krachtige techniek voor het oplossen van bepaalde soorten problemen. Problemen die op een efficiënte manier door genetische algoritmen worden opgelost zijn o.a. complexe planningproblemen of problemen waarvoor geen wiskundige analyse beschikbaar is. JGAP maakt het mogelijk om op een eenduidige manier genetische algoritmen binnen Java te gebruiken. JGAP implementeert de concepten uit de theorie waardoor de brug van theorie naar praktijk eenvoudig is te maken.

Voor wie verder geïnteresseerd is in de interne werking van genetische algoritmen, zoals bijvoorbeeld de werking van crossover en mutatie en het samenstellen van een nieuwe populatie, wordt verwezen naar bronnen uit de referenties. «

Referenties

http://jgap.sourceforge.net
http://en.wikipedia.org/wiki/Bin_packing_problem
http://code.google.com/p/jc-examples
http://www.tjhsst.edu/~ai/AI2001/GA.HTM
http://www.obitko.com/tutorials/genetic-algorithms/index.php