Universiteit Gent
Faculteit Wetenschappen
Bachelor Informatica

Algoritmen en Datastructuren III

Genetische Algoritmen - Verslag

Maarten Desnouck

Verslag bij het project van
Algoritmen en Datastructuren III:
Genetische Algoritmen
Academiejaar 2015-2016

Table of Contents

1 Implementatie

1.1 Algemene structuur

We zullen beginnen met de algemene structuur van het programma en het verloop. Om te beginnen zitten alle hulpmethoden in aantal aparte c-bestanden. Zo zitten bijvoorbeeld alle functies die te maken hebben met de omzetting van graden naar cordinaten in graden.c en zitten alle functies die iets doen met het vermogen en het bereik van de zendmasten in vermogenEnBereik.c. Dit zorgt ervoor dat main.c mooi kort en overzichtelijk blijft en dat elke methode makkelijk te vinden is. Het maakt ook hergebruik van deze code voor een eventueel toekomstig project wat makkelijker.

Main.c bevat de main-methode en begint met het controleren van alle argumenten. Met name het aantal en de waarde. Op deze manier stopt het programma zeer vlug als de argumenten niet correct zijn en niet na bijvoorbeeld al alle zendmasten ingeladen te hebben.

Vervolgens worden dus de locaties van de zendmasten en gebiedpunten in graden ingeladen. Hiervoor wordt telkens een 'struct punt *' gebruikt. Een punt bestaat uit een x,y en z float. Dit zorgt voor een intutieve manier om de data op te slaan en op te halen. Wanneer deze beide zijn ingeladen overlopen we al deze punten en we slaan hiervan telkens de maxima en minima op. Deze hebben we nodig om een goede referentiepunten te kiezen voor de omzetting van graden naar cordinaten en om een grenzen te hebben waarin we bepaalde crossover-assen random mogen kiezen.

We weten nu dus ons referentiepunten en we zetten dan ook alle zendmasten en gebiedpunten van graden om in cordinaten. Vanaf hier worden dan ook enkel nog cordinaten gebruikt om alles te berekenen, zoals afstanden en posities ten opzichte van assen.

Nu is dus zo goed als alles opgezet om het eigenlijke genetische algoritme uit te voeren. De variabelen zoals populatiegrootte en aantal mutaties per generatie werden globaal en hardcoded opgeslaan aan het begin van main.c en veranderen niet gedurende de uitvoering van het programma. We initialiseren een populatie van zendmastconfiguraties met alle zendmasten op 0 dBm. Dit gebeurt in de vorm van een 'int **' met als dimensies het aantal zendmasten en de populatiegrootte.

1.2 Volgende generatie

Voor elke generatie maken we nu uiteraard een selectie uit de populatie, evalueren we de fitnessfunctie en creren we op basis hiervan de volgende generatie. Dit is het algemene idee voor een genetisch algoritme maar hoe is het hier precies gemplementeerd?

We kiezen drie random exemplaren uit de populatie en evalueren voor elk van hen de fitnessfunctie. We sorteren ze vervolgend op basis van die fitness. We hebben er nu voor gekozen om de beste gewoon over te nemen, de tweede te overschrijven met het 1ste kind door crossover tussen de beste en de tweede en de derde te overschrijven met het 2de kind van de crossover tussen de beste en de tweede.

Deze manier een nieuwe generatie genereren zorgt voor een aantal wenselijke en cruciale effecten. Zo blijft de grootte van de populatie mooi constant, wordt de beste nooit per ongeluk overschreven, en is normaal een vooruitgang in de populatiefitness aangezien de 'genen' van de slechtste van de drie volledig uit de genenpoel verdwijnen en vervangen worden door een combinatie van de beste en de tweede.

De random selectie van de drie exemplaren zorgt er ten slotte voor dat een niet zo goed maar veelbelovend exemplaar overlevingskansen heeft in de populatie als hij bijvoorbeeld samen met twee slechtere exemplaren gekozen wordt. Deze kans zal groter zijn in een grotere populatie. Dit zal ervoor zorgen dat er met een grotere populatie breder gezocht kan worden maar hier meer over in het hoofdstuk over optimalisatie van de variabelen.

1.3 Fitnessfunctie

De fitnessfunctie bepaalt wat we als een goede oplossing zien en wat niet. Hier waren door de opgave niet zo veel keuzemogelijkheden opengelaten. Ik heb er

voor gekozen om 'de dekkingsgraad' als resultaat te nemen wanneer deze lager is dan de 'minimaleDekking'. Wanneer de dekkingsgraad tussen de 'minimaleDekking' en de 'overbodigeDekking' ligt is het resultaat 'dekking - minimaleDekking + overbodigeDekking)*factor/verbruik' en wanneer de dekking groter dan de 'overbodigeDekking' is geeft de functie 'factor/verbruik' terug. Dit ziet er op het eerste zicht niet zo intutief uit maar als de factor voldoende groot gekozen wordt zorgt het ervoor dat alle configuraties met een dekking groter dan 'minimaleDekking' sowieso beter zijn dan die die deze dekking niet hebben. Ook sluiten de twee andere gevallen mooi aan elkaar aan. Zo behalen configuraties die niet aan 'overbodigeDekking' raken maar een stuk minder energieverbruiken toch een goede score. Er kan nog gekozen worden om dekking en verbruik met verschillende factoren mee te laten tellen maar daar heb ik hier niet voor gekozen.

1.4 Soorten mutaties en crossovers

Een genetisch algoritme bootst de natuur na en ook in de productie van een nieuwe generatie doe het het dat. Zo is er crossover, DNA van beide ouders krijgen, en mutatie, kleine foutjes die in het DNA sluipen. Beide zijn volgens de vakliteratuur nodig om een gezonde populatie en een goed producerend genetisch algoritme te hebben.

1.4.1 Mutatie

Om mutatie na te bootsen zijn twee verschillende functies gemplementeerd; mutate en mutate2. Mutate kiest een random aantalMutaties in de opgegeven rangen en voort dan telkens een een mutatie uit met als grootte opnieuw in een opgegeven range. Het bereik van deze ranges is vrij belangrijk en zullen we in het volgend hoofdstuk proberen te optimaliseren. Mutate2 zet een aantal masten op maximaal vermogen of volledig af. Dit ook een mogelijke soort mutatie en lijkt vooral voor dit probleem goed te werken aangezien in de optimale oplossing bijna alle masten op maximaal vermogen of volledig af blijken te staan. Deze functie zullen we niet optimaliseren. Ook de soort verdeling van het aantal mutaties en de grootte van die mutatie is volgens vakliteratuur belangrijk voor de prestatie van het algoritme maar dit zullen we ook niet optimaliseren en we zullen het bij de uniforme verdeling binnen de opgegeven intervallen houden.

1.4.2 Crossover

Ook crossover proberen we met dit algoritme na te bootsen. Hiervoor zijn ook opnieuw verschillende opties waarvan er 2 gemplementeerd zijn; singleAxisCrossover en singleBlockCrossover. SingleAxisCrossover kiest een random as in het opgegeven

gebied en wisselt dan n van de twee kanten tussen de twee ouders. SingleBlock-Crossover kiest 2 horizontale en 2 verticale assen in het gebied. Dit bakent 9 gebieden af en er wordt vervolgens 1 random block gekozen en van dit block worden de vermogens van de masten in dit block tussen beide ouders gewisseld. Hier zijn opnieuw vele mogelijkheden om dit principe te implementeren en we hebben er opnieuw voor gekozen om dit niet te optimaleren. Ten slotte wil ik nog opmerken dat de aanzet tot percentageBlockCrossover ook in de code zit. Volgens de vakliteratuur is een crossover-functie met veel assen en blocks maar waar elk kind ongeveer 50 percent van DNA van elke ouders krijgt een zeer goede keuze maar ik ben er niet toe gekomen die te implementeren of te testen.

Bij deze implementaties van crossover wil ik nog opmerken dat het volgens mij belangrijk is om de 2D structuur van het probleem bij crossover te behouden. Je wil namelijk dat masten die dicht bij elkaar staan en eventueel een deel van het gebied efficint bedekken ook een grote kans hebben samen gewisseld of samen behouden te worden, iets wat bijna nooit zal gebeuren als crossover implementeert op een lijst of array.

2 Optimalisatie

Om elk van de hardcoded variabelen wat te optimaliseren heb ik een aantal experimenten gedaan door het verloop van de fitnesswaarde van de eerste te plotten in functie van de generatie, dit telkens over een verloop van 1000 generaties. Er is wel een kleine wijziging gebeurd in verschil met de gemplementeerde functie. Zo zijn de waarden waarbij de dekking kleiner dan de minimale dekking waren achteraf met een factor 10 vermenigvuldigd om ze duidelijker zichtbaar te maken op de grafiek. Dit heeft geen invloed op de interpretatie van de resultaten. Aangezien elk datapunt telkens de beste is van drie random gekozen exemplaren zit hier wat variatie op, maar de trend is wel duidelijk zichtbaar. Voor elke configuratie van variabelen heb ik het drie keer laten lopen en dan de gemiddelde waarde berekent. De drie runs zijn bij elke waarde over elk experiment vrij consistent en dus kunnen er vrij geloofwaardige conclusies aan verbinden. De ruwe data werd bij dit verslag gevoegd als Excel-bestand.

2.1 Invloed van de populatiegrootte op de fitnessevolutie

We beginnen met de invloed van de grootte van de populatie op het verloop van de fitness. Op Figure 2.1 zien we hoe verschillende populatiegroottes presteren. We zien dat een populatie van 5 exemplaren over de hele lijn beter presteert dan een populatie van 10 exemplaren en dat die beide beter presteren dan een populatie van 50. De conclusie is duidelijk; voor deze implementatie van dit prob-

leem is en kleinere populatie beter. Dit is hoogstwaarschijnlijk omdat een brede zoekruimte voor dit probleem niet opweegt tegen de veel tragere evolutie van de volledige populatie omdat het steeds 3 voortplantingen per generatie blijven. Het zou waarschijnlijk interessant zijn om eens te zien hoe de evolutie verloopt als het aantal voortplantingen per generatie mee zou schalen met de populatie (zoals in de natuur). Maar de metriek is hier processortijd dus de conclusie blijft dat een kleinere populatie hier beter presteert. Voor de volgende experimenten hebben we de populatie telkens op 10 gehouden.

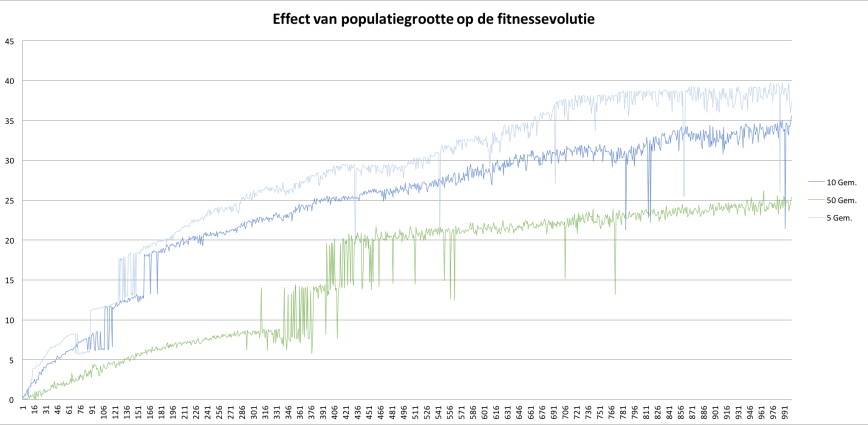

Figure 2.1: Fitness in functie van generatie voor verschillende populatiegroottes

2.2 Invloed van het aantal mutaties per generatie op de fitnessevolutie

Een tweede factor die we kunnen bekijken is hoe het aantal mutaties per generatie invloed heeft op de evolutie van de fitness. Als we naar Figure 2.2 kijken zien we een zeer interessant verloop. We zien dat 80 tot 100 mutaties per generatie het zeer slecht doet. Het lijkt bijna op random in de zoekruimte kiezen en er is dan ook zo goed als geen verbetering. Ook 10 tot 20 mutaties per generatie doet het vrij slecht maar deze kan blijkbaar net het random kiezen met evolutie overwinnen en er is dan ook een kleine maar merkbare verbetering. Zoals in de vakliteratuur voorspelt levert een klein aantal mutaties per generatie het beste resultaat op. Misschien moet de range van 1 tot 5 zelfs nog wat kleiner gemaakt worden voor een optimaal resultaat.

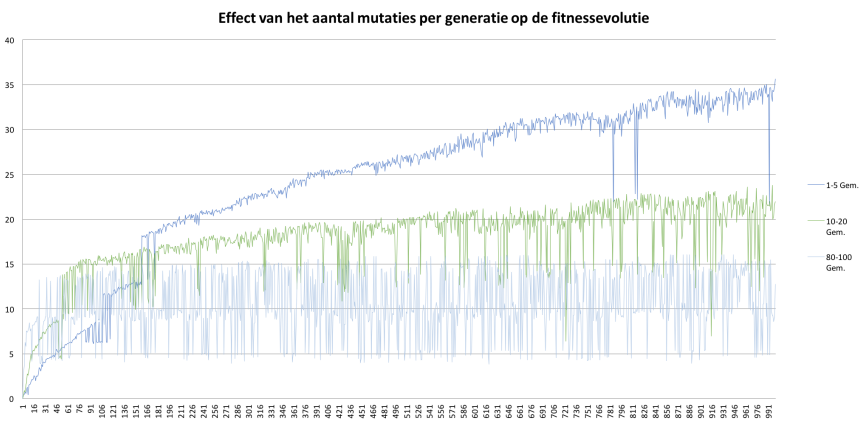

Figure 2.2: Fitness in functie van generatie voor verschillende aantal mutaties per generatie

2.3 Invloed van de grootte van een mutatie op de fitnessevolutie

De laatste variabele die we ten slotte zullen proberen te optimaliseren is de grootte van een enkele mutatie. We hebben deze experimenten uitgevoerd met telkens 1 tot 5 mutaties per generatie maar we hebben de grootte van die mutaties laten variren van 1 tot 43. We zien dat 1 en 5 in 1000 generaties zelfs de minimale dekkingsgraad niet halen. We zien dat 43 het best presteert. Als we er over nadenken komt dit eigenlijk neer op het vermogen op een random waarde initialiseren. Een zeer interessante conclusie.

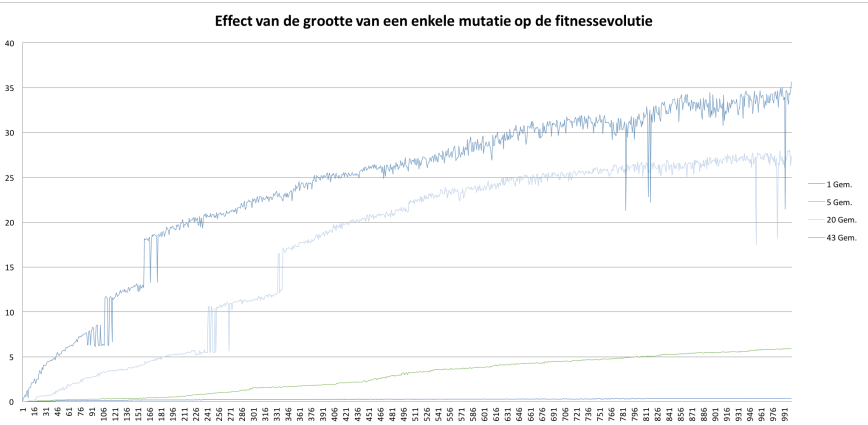

Figure 2.3: Fitness in functie van generatie voor verschillende mutatiegroottes

2.4 Conclusie

Er zijn nog een boel variabelen die we konden testen maar daar was geen tijd meer voor. Zo konden we eens kijken hoe mutate2 presteert, want ik denk dat dit zeer goed zou kunnen zijn aangezien een optimale oplossing zeer vaak masten op 0 of 43 lijkt te zetten. We konden als extra experiment ook eens kijken hoe mutate zonder crossover zou presteren.

Maar toch kunnen we concluderen dat voor deze implementatie van dit probleem een kleine populatie, een klein aantal mutaties per generatie en een grote mutatiegrootte optimale variabelen zijn om snel (binnen de geteste 1000 generaties) tot een relatief goede oplossing te komen.