UNIVERSITEIT ANTWERPEN
Faculteit Wetenschappen
2015-2016

Design of fault-tolerant genetic algorithm for application placement in heterogeneous cloud environments

Master Thesis

Ruben Mennes
s0111307

Master in de Informatica
Computernetwerken en
Gedistribueerde Systemen

Promotor:
Prof. dr. Steven Latré
Bart Spinnewyn

Voorwoord

Deze thesis werd geschreven voor de opleiding "Master in de Informatica: Computernetwerken en Gedistribueerde Systemen" aan de Universiteit Antwerpen.

Toen ik in juni vorig jaar een onderwerp voor mijn masterproef zocht, was ik ervan overtuigd dat cloud computing en cloud management de topics waren waarmee ik aan de slag wou gaan. Mijn promotor, Prof. dr. Steven Latré, gaf mij de kans om mijn thesis en onderzoek te laten verder bouwen op het werk van Bart Spinnewyn, die mij voor de hele periode van mijn onderzoek mee begeleid heeft. Samen met Prof. dr. Steven Latré en Bart Spinnewyn hebben we het onderwerp van deze thesis gedefinieerd.

Gedurende de negen maanden waarin ik aan deze thesis gewerkt heb, heb ik mijn kennis over dit onderwerp zeer goed kunnen uitbreiden. Ik heb het genoegen gekregen om dit onderwerp uitgebreid en volgens mijn eigen stijl te kunnen bestuderen. Ook had ik het geluk gedurende vijf maanden deel te kunnen uitmaken van de MOSAIC onderzoeksgroep. Dit was een zeer leerrijke ervaring en diende mede als inspiratiebron voor deze thesis.

Ik wens u veel leesplezier toe.

Ruben Mennes

Antwerpen, 4 juni 2016

Dankwoord

Met het schrijven van dit dankwoord kan ik een periode van negen maanden intensief werken aan deze thesis afsluiten. Tijdens deze periode heb ik veel bijgeleerd, ervaring mogen opdoen en met mensen kunnen discussieren over dit onderwerp. Dit was echter niet mogelijk zonder de steun van verschillende mensen.

Eerst wil ik graag mijn promotor Prof. dr. Steven Latré en mijn begeleider Bart Spinnewyn willen bedanken. Het is dankzij hen dat ik aan dit werk kon beginnen. Zij gaven me de kans om dit te kunnen doen en hebben mij ook geholpen tijdens de onderzoeksfase en het schrijfproces. Ik wil hen bedanken voor de fijne samenwerking.

Ook de hele onderzoeksgroep MOSAIC mag niet ontbreken in dit dankwoord. Het was een fijne ervaring om mee te mogen draaien in deze onderzoeksgroep tijdens de onderzoeksfase van mijn thesis. Hierbij wil ik vooral ook Dr. Miguel Camelo en Dr. Bart Braem danken voor hun feedback en het delen van hun kennis. Naast de onderzoeksgroep kan ook Dr. Juan Felipe Botero niet ontbreken in dit dankwoord. Hij heeft mij enorm geholpen tijdens het schrijven van een paper over deze thesis.

Als laatste bedank ik mijn ouders en mijn vriendin voor de steun die ik kreeg tijdens het schrijven van deze thesis. Ze waren steeds een luisterend oor en hielpen mij bij het nalezen van deze thesis.

Ruben Mennes

Antwerpen, 4 juni 2016

Abstract

Applicatie-placement-algoritmes vormen de basis waarop de werking van een cloud berust. Er werden in het verleden al tal van zulke algoritmes beschreven. In heterogene clouds wordt het beheer ingewikkelder doordat de infrastructuur bestaat uit machines met diverse capaciteiten. Zo'n heterogene configuratie doet zich bijboorbeeld voor in Internet of Things toepassingen. Sensor nodes hebben typisch niet veel resources ter beschikking en zullen dus moeten beroepen op rekenkracht van anderen, meestal door gebruik te maken van (onbetrouwbare) draadloze netwerken. Om de beschikbaarheid en betrouwbaarheid te vergroten is het noodzakelijk om een applicatie-placement-algoritme ter beschikking te hebben dat verschillende CPU en RAM capaciteit, met verschillende bandbreedtes tussen verschillende devices en de kans dat sommige devices of links falen, mee in rekening kan brengen. Verschillende soorten applicaties kunnen op zo'n cloud geplaatst worden en sommige daarvan moeten een hoge availability hebben, bijvoorbeeld als ze gebruikt worden voor roboticatoepassingen. Het is dus noodzakelijk dat het placement algoritme rekening houdt dat bepaalde applicaties een hoge beschikbaarheid moeten hebben. Spinnewyn et al. [17] definieerde een geheeltallig programma (Integer Linear Program) dat deze probleemstelling beschreef. Er werd gebruikt gemaakt van verschillende services die samen een applicatie kunnen vormen door ze te verbinden met een virtuele link. Deze werden door het geheeltallig programma geplaatst op de beschreven cloud-omgeving. Een typisch probleem met een geheeltallig programma is dat het niet schaalbaar is. Dit blijkt ook uit deze thesis. Om het schaalbaarheids probleem aan te pakken beschrijf ik in deze thesis een heuristiek in de vorm van een gedistribueerd genetisch algoritme, zo'n genetisch algoritme zal typisch de schaalbaarheid sterk verbeteren. Meestal is dit ten koste van de nauwkeurigheid van de oplossing, maar uit de testen bleek dat het genetisch algoritme meestal de optimale oplossing nog steeds kan vinden en een performantie winst van een factor 1000 kan bereiken.

Inhoudsopgave

Voorwoord i

Dankwoord ii

Abstract iii

Inhoudsopgave iv

Lijst van Figuren vii

Lijst van Tabellen ix

Lijst van Afkortingen x

Inleiding 1

I Genetisch algoritme voor application placement 4

1 Genetisch Algoritme voor Application Placement 5

1.1 Probleemstelling 5

1.1.1 Cloud omgeving 6

1.1.2 Applicaties 7

1.1.3 Opgelegde beperkingen en doelstellingen 8
1.2 Genetische Algoritmes 10
1.3 Multi Objective 14
1.3.1 Elite non-dominant sorting 16
1.4 Genetische voorstelling 18
1.4.1 Beperken van het aantal ongeldige oplossingen 18
1.4.2 Decoding 20

2 Framework **25**
2.1 Pool model 25
2.2 Opslaan van individu's in een database 27
2.3 Worker Architectuur 30

3 Resultaten en Discussie **33**
3.1 Opstelling van de testen 33
3.2 Placement Ratio 36
3.3 Executietijd 39
3.4 Schaalbaarheidstest 41
3.5 Verbeteringen voor het genetisch algoritme 44
3.5.1 Netwerkallocatie 44
3.5.2 Chromosoom inkorten 45
3.5.3 Snelheid verbeteren 45

II Conclusie **48**

Besluit **49**

Bibliografie **52**

III Bijlagen 53

A Evacloud configuratie **54**

B Test resultaten **56**

Lijst van figuren

1.1 Chromosoomvoorstelling van een cilinder met diameter 8 en hoogte 10 [6] 11

1.2 Flowchart werking van een GA [6] 12

1.3 Reproductieoperator 'tournament selector' [6] 13

1.4 Crossover: single-point crossover [6] 13

1.5 Mutatoroperator [6] 13

1.6 Pareto front en object space van fictief probleem [6] 15

1.7 Procedure NSGA-II [6] 17

1.8 Vier non-dominant Pareto fronts [6] 17

1.9 Decoder vertaalt het individu met biased random key naar een voorstelling in de solution space waarop de fitness berekend kan worden [6] 19

1.10 Flowchart van decoding algoritme 24

2.1 Basis architectuur pool model 26

2.2 Basis architectuur Evacloud 27

2.3 CAP theorema met database voorbeelden 28

2.4 Individu voorstelling in de MongoDB database 30

2.5 Flowchart van een worker 32

3.1 Placement Ratio: $\mathbf{T}_{0\%}$ 36

3.2 Placement Ratio: $\mathbf{T}^{1}_{90\%}$ 37

3.3 Placement Ratio Genetisch algorithme 37

3.4 Generaties Genetisch algorithme 38

3.5 Executietijd: $\mathbf{T}_{0\%}$ 39

3.6 Executietijd: $\mathbf{T}^{1}_{90\%}$ 40

3.7 Executietijd Genetisch Algorithme 40

3.8 Resultaten schaalbaarheidstesten 42

3.9 Resultaten schaalbaarheidstest 50 nodes 43

A.1 Test configuratie evacloud 55

Lijst van tabellen

1.1 Lijst met symbolen gebruikt voor de cloud-omgeving 6

1.2 Lijst met symbolen gebruikt voor de applicaties 7

1.3 Tabel die aangeeft wanneer resources herbruikt kunnen worden 8

1.4 Overige lijst met symbolen gebruikt door ILP en doelstellingen 8

3.1 Symbolen gebruikt voor de testcases 34

3.2 Gemiddelde executietijden per operator 45

3.3 Gemiddelde executietijden per operator voor test met 50 PMs 46

B.1 Gemiddelde Placement Ratio: Availability 0% 57

B.2 Gemiddelde Placement Ratio: Availability 90% 58

B.3 Gemiddelde Placement Ratio: Availability 99% 59

B.4 Gemiddelde Executietijd: Availability 0% 60

B.5 Gemiddelde Executietijd: Availability 90% 61

B.6 Gemiddelde Executietijd: Availability 99% 62

B.7 Test resultaten test met 10 Physical Machine (PM)s 63

B.8 Test resultaten test met 20 PMs 63

B.9 Test resultaten test met 50 PMs 64

Lijst van afkortingen

CLF CPU Load Factor.

CPU Central Processing Unit.

ET ExecutieTijd.

GA Genetisch Algoritme.

HPC High Performance Cluster.

ILP Integer Linear Program.

IoT Internet of Things.

MOOP Multi-Objective OptimalisatieProbleem.

PL Physical Link.

PM Physical Machine.

PR Placement Ratio.

RAM Random-Access Memory.

SOA Service-Oriented Architecture.

VL Virtuele Link.

VMs Virtuele Machines.

Inleiding

De laatste jaren zien we dat meer en meer objecten verbonden zijn met allerlei soorten draadloze netwerken. Denk hierbij aan smartwatches die verbonden zijn met je smartphone, of draadloze verlichting die verbonden is met een Wi-Fi netwerk. De verzamelnaam voor allerlei soorten objecten die verbonden worden met netwerken wordt ook wel Internet of Things (IoT) genoemd. De meeste van deze IoT toepassingen vragen veel resources. Denk hierbij aan Central Processing Unit (CPU), memory en zelfs storage. Evenzeer zijn de devices meestal verbonden met het netwerk via low-power draadloze technologieën, die van nature al minder betrouwbaar of zelfs onbetrouwbaar zijn ten opzichte van traditionele bedrade netwerken [20].

De computerkracht die vereist is voor de IoT toepassingen wordt meestal voorzien in cloud-omgevingen. Deze cloud-omgevingen gebruiken virtuele machines om *on demand* rekencapaciteit aan te vragen en te gebruiken. Deze cloud-omgevingen introduceren typisch netwerk latency, omdat de infrastructuur van een cloud zich meestal in een afgelegen datacenter bevindt. Het resultaat van deze netwerk latency is dat de reactietijd van de applicaties ook zal toenemen. Edge-clouds en fog computing reduceren de latency in de meeste gevallen al. Maar ook hier kan de netwerk latency te hoog zijn. Sommige applicaties verwachten namelijk een ultrakleine responsetijd. We denken hierbij aan roboticatoepassingen die beslissingen moeten nemen binnen enkele milliseconden. Om deze ultrakleine responsetijd te bereiken, introduceerden Spinnewyn et al. [17] een heterogene cloud-omgeving. Dergelijke cloud-omgeving zal trachten gebruik te maken van verschillende soorten machines, die meestal in de buurt gelegen zijn en verbonden zijn met verschillende soorten netwerktechnologieën, om zo rekenkracht te voorzien.

In tegenstelling tot de traditionele clouds blijkt de infrastructuur in een heterogene cloud eerder onbetrouwbaar te zijn. De devices waarvan gebruik gemaakt worden,

kunnen gevoed worden door batterijen en/of verbonden zijn met low-powered wireless netwerktechnologieën. Fysieke verplaatsing van een device gedurende uitvoering van applicaties kan ook mogelijk zijn.

Clouds gebruiken Virtuele Machines (VMs) (of containers) om rekenkracht te voorzien. Deze VMs bevatten meestal een service van een grotere applicatie, dit soort architectuur noemen we een Service-Oriented Architecture (SOA). Een applicatie bestaat dus uit het combineren van verschillende services die in verschillende virtuele machines (of containers) draaien. Het algoritme dat verantwoordelijk is om deze virtuele machines (en services) te plaatsen is een van de belangrijkste algoritmes die een cloud provider nodig heeft om zijn cloud te starten. Er bestaan verschillende algoritmes die het plaatsen van applicaties kunnen doen. Jennings and Stadler [14] lijstten het grootste deel van de huidige statische en dynamische placement-algoritmes op. Desalniettemin voldoen deze algoritmes niet aan de eis die er is bij een heterogene cloud voor een IoT omgeving. Denk hierbij aan het maximaliseren van de beschikbaarheid (availability), rekening houden dat het netwerk eerder onbetrouwbaar is (zeker als het wireless is), of het reduceren van gebruikte energie (als de devices op batterijen werken). Dit is ook te lezen in het werk van Jennings and Stadler [14] waar beschreven staat dat het mobile cloud paradigm nog een grote uitdaging is.

Spinnewyn et al. [17] ontwierpen een geheeltallige programma, of Integer Linear Program (ILP), om de applicatie placement te berekenen voor de heterogene clouds. Een ILP is typisch NP-hard. Om de rekentijd van hiervan te beperken ontwierp ik een Genetisch Algoritme (GA) om de placement te benaderen. Dit model is geïmplementeerd in een eigen gedistribueerd Multi-Objective GA framework.

Het ontworpen framework moest gedistribueerd zijn om fault-tolerant te zijn. De meeste frameworks voor genetische algoritmes zijn ofwel niet geschikt voor multi-objectives, of zijn niet gebouwd voor gedistribueerde berekeningen. Het was dus nodig om er zelf een te implemeteren zodat we met het ontworpen GA testen konden doen.

Deze thesis is als volgt opgebouwd: in hoofdstuk 1.1 beschrijven we de probleemstelling. In hoofdstuk 1.2 en 1.3 beschrijven we de details van de gebruikte techniek, genetische algoritmes voor multiobjective optimalisatie, gebaseerd op het werk van Deb [6]. De chromosoomvoorstelling, wat het belangrijkste verwezelijking is van deze

thesis, wordt beschreven in hoofdstuk 1.4. Het ontwikkelde framework, waarmee tevens ook alle testen zijn gedaan, staat beschreven in hoofdstuk 2. Test resultaten en discussie staan genoteerd onder hoofdstuk 3.

DEEL I

Ontwerp van een robuust genetisch algoritme voor application placement in heterogene cloud-omgevingen

Hoofdstuk 1

Genetisch Algoritme voor Application Placement

Genetic algorithms are computer programs that 'evolve' in ways that resemble natural selection can solve complex problems even their creators do not fully understand

Holland 13

1.1 Probleemstelling

Zoals eerder vermeld is de enige studie voor applicatie-placement in heterogene cloud-omgevingen waarvan wij weet hebben, de studie van Spinnewyn et al. [17]. In dit werk wordt een ILP beschreven voor applicatie-placement in heterogene cloud-omgevingen. Deze paper en ILP vormen dan ook de basis van de thesis. Zo'n ILP is typisch niet schaalbaar. Daarom proberen we door gebruik te maken van een heuristiek, in de vorm van een GA om de snelheid voor het oplossen van een probleem te verbeteren. Om zo'n GA op te stellen beschrijven we in deze sectie het model dat we gebruikt hebben. Dat model is ook de basis voor het ILP en wordt hieronder uitvoerig besproken.

Symbool	Beschrijving
Ω_n	De CPU capaciteit van fysieke machine n
Γ_n	De RAM capaciteit van fysieke machine n
B_e	De bandbreedte van de fysieke link e
p_n^N	De kans op falen van fysieke machine n
p_e^E	De kans op falen van fysieke link e
$\hat{p}_n^N$	De beschikbaarheid van fysieke machine n
$\hat{p}_e^E$	De beschikbaarheid van fysieke link e

Tabel 1.1: Lijst met symbolen gebruikt voor de cloud-omgeving

1.1.1 Cloud omgeving

In de voorstelling van Spinnewyn et al. [17] stellen we de cloud voor als een verzameling van PM en Physical Link (PL). Fysieke nodes hebben een gegeven hoeveelheid CPU (Ω) en een gegeven hoeveelheid Random-Access Memory (RAM) (Γ) ter beschikking. Op deze machines kunnen services geplaatst worden op voorwaarde dat er CPU en RAM gealloceerd wordt (wordt besproken in sectie 1.1.2). Verschillende fysieke machines kunnen verbonden zijn met een PL. Deze link heeft een maximale hoeveelheid bandbreedte (B).

Elke PM en PL heeft een gegeven kans op falen (p^N en p^E respectievelijk). Er wordt ook verondersteld dat deze fouten onafhankelijk zijn van elkaar. We introduceren ook de kans dat de PM of PL beschikbaar zijn ($\bar{p}^N$ en $\bar{p}^E$ respectievelijk). Hiervoor geldt:

$$\bar{p}^N = 1 - p^N \tag{1.1}$$

$$\bar{p}^E = 1 - p^E \tag{1.2}$$

Alle symbolen die gebruikt worden voor de cloud omgeving te beschrijven zijn opgelijst in tabel 1.1.

Symbool	Beschrijving
ω_s	CPU vereist voor service s
γ_s	RAM vereist voor service s
β_{s_1,s_2}	Vereiste bandbreedte van virtuele link tussen s_1 en s_2
R_a	Minimale beschikbaarheid van applicatie a
δ_a	Maximaal toegelaten duplicaten van applicatie a

Tabel 1.2: Lijst met symbolen gebruikt voor de applicaties

1.1.2 Applicaties

Het model beschrijft dat applicaties gebruik zullen maken van een SOA. Dit betekent dat er verschillende services zijn die door verschillende applicaties gebruikt kunnen worden. Services gebruiken een vaste hoeveelheid CPU (ω) en eisen een constante hoeveelheid memory (γ). Services binnen eenzelfde applicatie kunnen verbonden worden door een Virtuele Link (VL). Deze link zal er voor zorgen dat er communicatie mogelijk is tussen twee services. Deze virtuele link alloceert een vaste hoeveelheid bandbreedte (β). Elke applicatie eist ook een minimale beschikbaarheid (availability) (R).

Services kunnen enkel geplaatst worden op machines die voldoende memory en CPU ter beschikking hebben. Applicaties moeten minimaal aan de beschikbaarheidsvoorwaarde voldoen. Om dit te kunnen garanderen introduceerden Spinnewyn et al. [17] duplicaten. Een applicatie a mag maximaal δ_a duplicaten hebben. Verschillende duplicaten werken onafhankelijk van elkaar. Deze veronderstelling hebben we gehouden om de complexiteit van het probleem te reduceren. Het kan wel zijn dat een applicatie op eenzelfde machine dezelfde service meerdere keren geplaatst heeft van verschillende duplicaten.

Omdat services ook door andere duplicaten en applicaties op dezelfde node geplaatst kunnen worden, kunnen ze sommige resources delen. Zo kan bijvoorbeeld het RAM van de service herbruikt worden omdat in de praktijk de service maar exact 1 keer in geheugen geladen zal worden. Tabel 1.3 beschrijft welke resources er nog herbruikt kunnen worden als eenzelfde service op een eenzelfde node geplaatst wordt of eenzelfde VL op eenzelfde PL.

Resource	Gebruikt door zelfde app	Gebruikt door andere app
CPU	Gedeeld	Niet gedeeld
RAM	Gedeeld	Gedeeld
Bandbreedte	Gedeeld	Niet gedeeld

Tabel 1.3: Tabel die aangeeft wanneer resources herbruikt kunnen worden

Symbool	Beschrijving
$\mathbf{A}$	De verzameling van applicaties
$\mathbf{S}$	De verzameling van services
$\mathbf{E}$	De verzameling van fysieke links
$\mathbf{N}$	De verzameling van fysieke machines
$\mathbf{S}_a$	De verzameling van alle services gebruikt door applicatie a
O^a	Is 1 als de applicatie a correct geplaatst is. Anders 0.
$\Upsilon^a_{s_1,s_2}(e)$	Gebruik van fysieke link e door applicatie a voor virtuele link tussen s_1 en s_2
$\Pi^a_{s,n}$	1 als node n service s voor applicatie a host, anders 0.
$G^{d,a}$	1 als duplicaat d van applicatie a geplaatst is, anders 0.
$\mathbf{D}_a$	Verzameling van duplicaten voor applicatie a.
$U_{s,n}$	1 als service s gehost is op node n.

Tabel 1.4: Overige lijst met symbolen gebruikt door ILP en doelstellingen

De gebruikte symbolen voor de applicaties zijn terug te vinden in tabel 1.2.

1.1.3 Opgelegde beperkingen en doelstellingen

Net zoals Spinnewyn et al. [17] willen wij ook verschillende doelstellingen optimaliseren. Deze worden typisch beschreven in doelstellingfuncties. Omwille van consitentieredenen zullen we telkens streven om de doelstellingfuncties te minimaliseren. Hieronder lijsten we alvast de 5 doelstellingen die wij voor ogen hebben op. Extra symbolen die gebruikt worden, worden gedefiniëerd in tabel 1.4.

- Maximaliseer het aantal geplaatste applicaties.

$$f_1(\mathbf{A}) = -\sum_{a \in \mathbf{A}} O^a \tag{1.3}$$

- Minimaliseer de hoeveelheid gebruikte bandbreedte.

$$f_2(\mathbf{A}, \mathbf{E}, \mathbf{S}, \beta) = \sum_{a \in \mathbf{A}} \sum_{e \in \mathbf{E}} \sum_{s_1, s_2 \in \mathbf{S}} \Upsilon^a_{s_1,s_2}(e) \times \beta_{s_1,s_2} \tag{1.4}$$

- Minimaliseer de hoeveelheid gebruikte CPU

$$f_3(\mathbf{A}, \mathbf{N}, \mathbf{S}, \omega) = \sum_{n \in \mathbf{N}} \sum_{a \in \mathbf{A}} \sum_{s \in \mathbf{S}} \Pi^a_{s,n} \times \omega_s \tag{1.5}$$

- Minimaliseer het aantal gebruikte duplicaten

$$f_4(\mathbf{A}, \mathbf{D}) = \sum_{a \in \mathbf{A}} \sum_{d \in \mathbf{D}_a} G^{d,a} \tag{1.6}$$

- Minimaliseer het aantal gebruikte RAM

$$f_5(\mathbf{N}, \mathbf{S}, \gamma) = \sum_{n \in \mathbf{N}} \sum_{s \in \mathbf{S}} \Pi_{s,n} \times \gamma_s \tag{1.7}$$

De meeste van deze doelstellingen werden ook al gebruikt door Spinnewyn et al. [17]. Enkel het RAM gebruik werd niet geminimaliseerd in het werk van Spinnewyn et al. [17].

Zoals hierboven beschreven kan niet elke service zomaar op elke PM geplaatst worden. Er worden dus enkele beperkingen opgelegd. In de lijst hieronder worden alle beperkingen (constraints) opgesomd.

- Het aantal duplicaten die applicatie a kan plaatsen is beperkt door δ_a.

$$\forall a \in \mathbf{A} : |\mathbf{D}_a| \leq \delta_a \tag{1.8}$$

- Een applicatie is geplaatst als op zijn minst één van zijn duplicaten geplaatst werd.

$$\forall a \in \mathbf{A} : O^a = 1 \Leftrightarrow |\mathbf{D}_a| \geq 1 \tag{1.9}$$

- Een service kan enkel geplaatst worden op een PM waarvoor voldoende CPU (1.10) en RAM (1.11) beschikbaar is. Serviceses die herbruikt kunnen worden, gedragen zich zoals vermeld in tabel 1.3.

$$\forall n \in \mathbf{N} : \sum_{a \in \mathbf{A}} \sum_{s \in \mathbf{S}} \Pi^a_{s,n} \times \omega_s \leq \Omega_n \tag{1.10}$$

$$\forall n \in \mathbf{N} : \sum_{s \in \mathbf{S}} U_{s,n} \times \gamma_s \leq \Gamma_n \tag{1.11}$$

- Een service wordt enkel gehost op een PM als hij gebruikt wordt door minstens een duplicaat.

$$\forall s \in \mathbf{S} : \sum_{n \in \mathbf{N}} U_{s,n} > 0 \Leftrightarrow \exists a \in \mathbf{A} : O^a = \wedge s \in \mathbf{S}_a \tag{1.12}$$

- De totale bandbreedte van een PL mag niet overschreven worden door de totale bandbreedte van alle VL die gebruik maken van deze PL.

$$\forall e \in \mathbf{E} : \sum_{s_1 \in \mathbf{S}} \sum_{s_2 \in \mathbf{S}} \sum_{a \in \mathbf{A}} \Upsilon^a_{s_1,s_2}(e) \times \beta_{s_1,s_2} \leq B_e \tag{1.13}$$

- Een virtuele link kan enkel gebruik maken van een PL als er voldoende bandbreedte beschikbaar is. Tevens gedraagt het zich zoals vermeld in tabel 1.3.

$$\forall e \in \mathbf{E}, \forall a \in \mathbf{A} : \sum_{s_1,s_2 \in \mathbf{S}} \Upsilon^a_{s_1,s_2}(e) \times \beta_{s_1,s_2} \leq B_e \tag{1.14}$$

- Een applicatie wordt enkel geplaatst als de volledige applicatie de gevraagde beschikbaarheid kan garanderen [17].

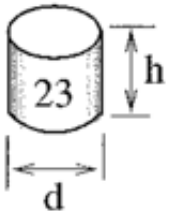
1.2 Genetische Algoritmes

In deze thesis onderzoeken we of een genetisch algoritme in staat is om een applicatieplaatsingsprobleem efficiënt op te lossen. In dit hoofdstuk worden de belangrijkste kenmerken van een GA uitgelegd en beschreven. We gebruiken de terminologie die Deb [6] gebruikte in zijn werk. Deze termen komen terug later in de thesis.

Genetisch Algoritme is een optimalisatietechniek waarbij er gebruik wordt gemaakt van natuurlijke selectie en *survival of the fittest* om zo tot een benaderende oplossing te komen. Het idee om de principes van evolutieleer te gebruiken in algoritmes werd voor het eerst beschreven door Turing [19]. Hij beschreef de basisprincipes van genetische algoritmes toen hij zijn *learning machine* beschreef. Genetische algoritmes legden een lange weg af en werden steeds meer en meer gebruikt. Onder andere Holland [13] beschreef dat je problemen kan voorstellen in een chromosoom dat wordt voorgesteld als een bitarray. Deze bitarray kan later geïnterpreteerd worden als oplossing zoals in figuur 1.1. In dit voorbeeld, [6], proberen we een cilinder

te zoeken waarbij we de kost minimaliseren maar de cilinder moet groter zijn dan 300ml. De kost de cilinder te maken is $c\left(\frac{\pi d^2}{2} + \pi dh\right)$. Omdat we de kost willen minimaliseren noemen we deze functie ook wel de fitness functie. We minimaliseren functie 1.15.

$$f(d,h) = c\left(\frac{\pi d^2}{2} + \pi dh\right) \tag{1.15}$$

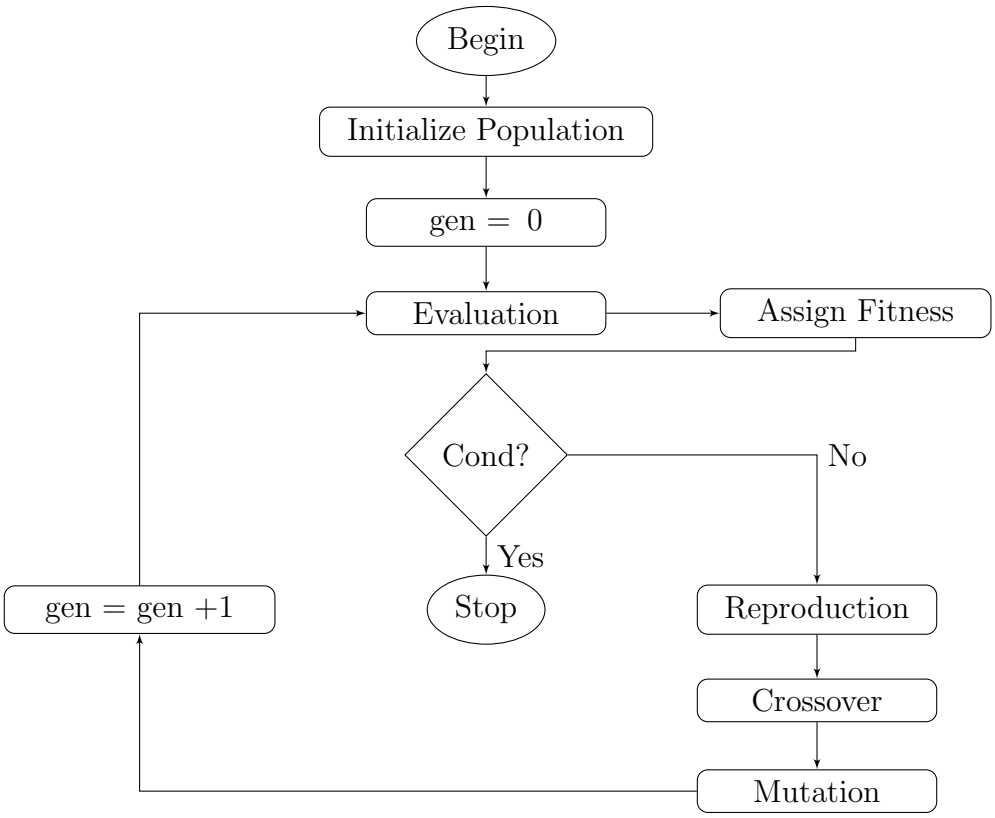


(d, h) = (8, 10) cm
(Chromosome) = 0 1 0 0 0 0 1 0 1 0

Figuur 1.1: Chromosoomvoorstelling van een cilinder met diameter 8 en hoogte 10 [6]

Zoals eerder vermeld lost een genetisch algoritme het probleem op door het nabootsten van natuurlijke selectie. We starten dus steeds met het aanmaken van een random startgeneratie. Daarna proberen we iteratief tot een oplossing te komen door het overlopen van 3 grote fases: reproductie (of selectie), crossover en mutatie (figuur 1.2).

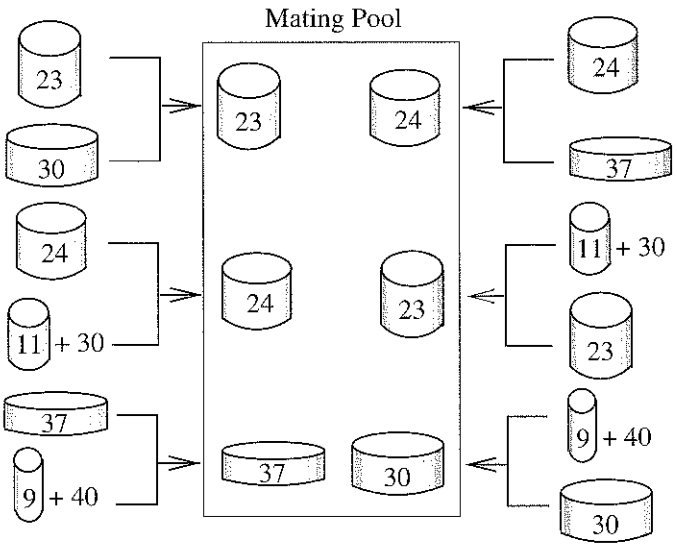
Reproductie De reproductieoperator, of ook wel selectieoperator genoemd, is verantwoordelijk voor het selecteren van de 'goede' individu's. Er zijn voor deze operator verschillende implementaties. Het meest eenvoudige algoritme is de 'Tournament selection'. Deze selector laat elk individu een andere random gekozen tegenstander kiezen. Hiervoor bekijken we telkens de twee fitness-waarden. De kleinste fitnesswaarde wint en wordt behouden. De verliezer voegen we niet toe aan de 'mating pool'. Enkel de individu's die het halen tot in de mating pool worden overgehouden om te beginnen aan de volgende stap (figuur 1.3). Zoals al eerder vermeld zijn er nog andere soorten selectieoperatoren. Deze gebruiken bijvoorbeeld een kansverdeling zodat ook het individu met een mindere fitness kans maakt om tot in de mating pool te geraken [6].



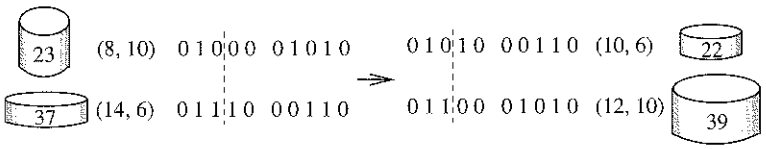
Figuur 1.2: Flowchart werking van een GA [6]

Crossover Crossover kan vergeleken worden met het aanmaken van nieuwe individu's. De meest typische manier om een crossover te doen is twee individu's met elkaar te laten paren en twee kinderen te maken (van een volgende generatie). Hierbij proberen we twee ouders te combineren om een beter individu te maken. Het spreekt voor zich dat dit niet altijd het geval is. Figuur 1.4 illustreert dat, als je twee individu's combineert, je tot een beter resultaat kan komen, maar dat het ook mogelijk is dat je een slechter individu kan bekomen. Ook van de crossover operator zijn er verschillende mogelijke implementaties. Figuur 1.4 illustreert een single-point crossover. Hierbij wordt er een punt bepaald. Alles voor dit punt komt van het eerste chromosoom en daarna komt alles van het tweede chromosoom. Bij het tweede kind is dit omgekeerd. Er kan er ook voor elke bit apart een keuze gemaakt worden met een kans van 50% om de eerste of de tweede bit te kiezen [6].

Mutator De laatste operator is de mutator. Deze operator gebruikt een nieuw individu. Bij dit individu kan het zijn dat er een mutatie optreedt. Een bit wisselt

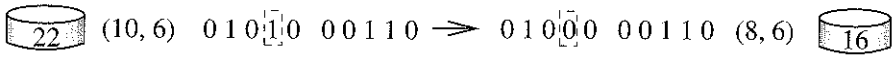


Figuur 1.3: Reproductieoperator 'tournament selector' [6]



Figuur 1.4: Crossover: single-point crossover [6]

van waarde. Je kan dit vergelijken met de biologie. In het menselijk chromosoom kan het ook zijn dat er een mutatie optreedt. In sommige gevallen geeft dit negatieve gevolgen, denk hierbij aan bepaalde syndromen of ziektes. Het kan echter ook zijn dat zich hierdoor een bepaalde gave ontwikkelt of zelfs een nieuwe diersoort. Als de mutatie een slechte invloed heeft op ons individu, zijn we er bijna zeker van dat na de volgende selectie dit individu 'gestorven' zal zijn. Een mutator zal meestal het aantal bits dat kan wisselen beperken (meestal maximaal 1 bit per chromosoom) en de kans dat een bit van waarde wisselt, is meestal ook klein. Een gemakkelijk voorbeeld is te zien in figuur 1.5.



Figuur 1.5: Mutatoroperator [6]

Eindconditie Zoals te zien in figuur 1.2 stopt het algoritme bij een bepaalde conditie. Deze conditie is afhankelijk van het probleem. Algemene voorbeelden van stopcondities is dat het GA stopt als het n generaties berekend heeft. Vaak wordt deze conditie altijd ook als extra conditie gebruikt zodat het GA niet oneindig blijft lopen maar stopt binnen een bepaalde tijd. Een andere eindconditie is een conditie die bijhoudt hoelang de beste individu's al in de huidige populatie zitten. Als een bepaald individu meer generaties in de database zit dan een drempel t zal het algoritme stoppen. In sommige specifieke gevallen kunnen ook een paar specifieke eigenschappen van de optimale oplossing gebruikt worden als stopconditie maar deze zijn probleemspecifiek.

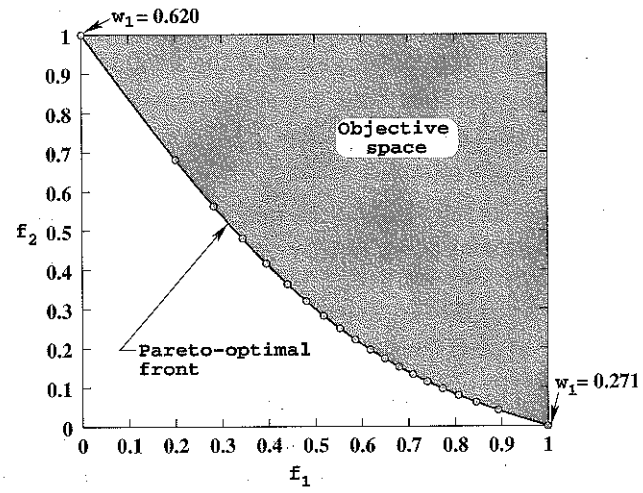
1.3 Multi Objective

Het beschreven ILP [17] probeerde meerder objectieven te minimaliseren/maximaliseren.

- Maximaliseer acceptatie
- Minimaliseer bandwidth
- Minimaliseer CPU
- Minimaliseer het aantal duplicaten

Om de bespreking overzichtelijk te houden gaan we er van uit dat we maar twee objectieven hebben (maximaliseren van acceptatie en minimaliseren van RAM). Als we hebben uitgelegd hoe een multi-objective genetisch algoritme voor twee objecten typisch werkt, is het introduceren van de overige objectieven triviaal.

Als we te maken hebben met een Multi-Objective OptimalisatieProbleem (MOOP) is er typisch geen enkelvoudige oplossing. We zijn op zoek naar een Pareto front. Een Pareto front is een curve waarop alle optimale oplossingen liggen. Het is logisch dat, als we bijvoorbeeld CPU willen minimaliseren, dat dit ten koste kan gaan van de acceptatie van de applicaties en omgekeerd. Het is aan de eindgebruiker om uit de oplossingen op dit Pareto front de beste te kiezen. In figuur 1.6 kan u een fictief voorbeeld zien waarbij verschillende oplossingen van de object space



Figuur 1.6: Pareto front en object space van fictief probleem [6]

op het Pareto front liggen. Welke oplossing hiervan de beste is, hangt af van het probleem en moet bepaald worden door de eindgebruiker. Een Pareto front is ook de verzameling van alle dominante resultaten [6]. In tegenstelling met single-objective optimalisatie, waarbij bijvoorbeeld de gewogen som van de verschillende objectieven gebruikt wordt, moet er een oplossing gekozen worden van het Pareto front. Terwijl een single-objective optimalisatie een unieke oplossing nastreeft.

Een oplossing $x^{(1)}$ is dominant ten opzichte van een andere oplossing $x^{(2)}$ als volgende condities gelden:

1. Oplossing $x^{(1)}$ is niet erger dan $x^{(2)}$.
   Dit betekent dat voor elk object i dat we willen minimaliseren $x_i^{(1)} \leq x_i^{(2)}$ en voor elk object j dat we willen maximaliseren $x_i^{(1)} \geq x_i^{(2)}$.

2. Oplossing $x^{(1)}$ is voor minstens een objective strict beter dan $x^{(2)}$.
   Dit betekent dat er minsten een object i dat we willen minimaliseren bestaat, waarvoor $x_i^{(1)} < x_j^{(1)}$ geldt, of een object j dat we willen maximaliseren bestaat waarvoor $x_j^{(1)} > x_i^{(2)}$ geldt.

Het spreekt voor zich dat de verzameling met alle dominante resultaten op het Pareto front moet liggen.

Als we een GA gebruiken om een MOOP op te lossen, is het belangrijk dat we onze oplossingen zo breed mogelijk in de oplossingsverzamelingen zoeken. Als we

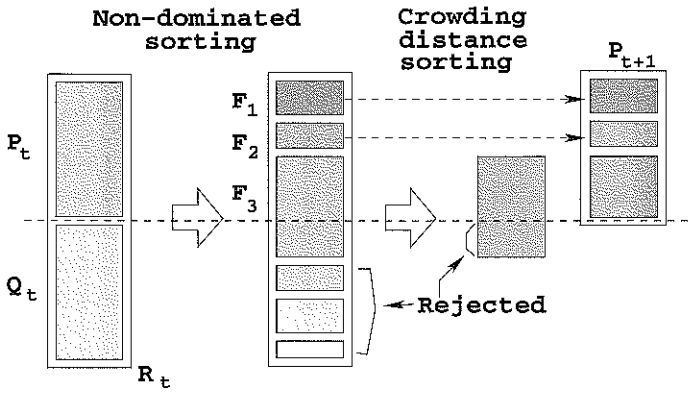
dit niet doen, is de kans groot dat we met ons GA evolueren naar een richting van het Pareto front terwijl we typisch op zoek gaan naar zoveel mogelijk verschillende oplossingen. De oplossingsverzameling zo breed mogelijk nemen, wordt ook wel *exploration* genoemd. Hierbij wordt geprobeerd de variateit zo groot mogelijk te houden. De tegenhanger hiervan wordt *exploitation* genoemd. Hier wordt er een selectie genomen om de zoekrichting te verkleinen. Dit kan bijvoorbeeld handig zijn als vooraf geweten is dat de fitness van een bepaald objectief typisch naar een goede oplossing leidt. In dat geval kan de zoektocht naar oplossingen verfijnd worden. Typisch zal het genetisch algoritme een deel van de exploitation op zich nemen.

Deb [6] beschreef dat er twee mogelijkheden zijn om een multi-objective genetisch algoritme op te lossen. Ofwel maken we gebruik van elitebehoud, ofwel niet. Het verschil in deze twee zit hem in het feit dat de algoritmes van het type niet-elitebehoud enkel zullen doorrekenen met de huidige nieuwgevormde generatie. De ouders zullen dus de volgende generatie niet overleven. Het kan echter zijn dat de ouders zelf enkel minder goede individu's maken, bijvoorbeeld als ze al dicht bij het absolute Parto front liggen. Algoritmes die gebruik maken van elitebehoud zullen zo'n ouder die al dicht bij het Pareto front ligt, bijhouden. Hier zullen namelijk enkel de n beste individu's (van zowel de nieuwe generatie als hun ouders) de volgende generatie overleven. Zoals we later zullen beschrijven, maak ik in onze voorstelling gebruik van een algoritme met elitebehoud.

Een andere methode is om meerdere objectives te vereenvoudigen naar een objective. Dit zal de schaalbaarheid van het probleem ten goede komen [4]. In ons geval zullen we de objective RAM, CPU en aantal duplicaten reduceren tot RAM omdat we er vanuit gaan dat deze grotendeels dezelfde acties zullen ondernemen: het reduceren van het aantal services.

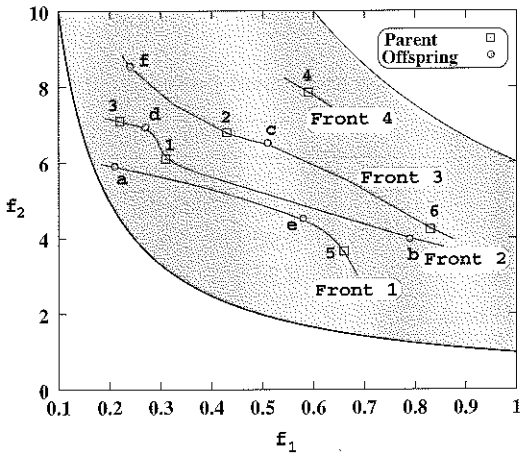
1.3.1 Elite non-dominant sorting

Omdat we een probleem hebben met enkele verschillende objectives is het noodzakelijk om aan elitebehoud te doen. Dit zorgt ervoor dat de beste oplossingen steeds aanwezig blijven in de database en niet verloren kunnen gaan. Deb et al. [7] suggereerde om elite non-dominant sorting te gebruiken. In deze thesis gebruiken en bespreken we NSGA-II als non-dominant sorteeralgorithme.



Figuur 1.7: Procedure NSGA-II [6]

Figuur 1.7 toont hoe het elitebehoud werkt. De eerste stap is om alle individu's, zowel van de huidige generatie als hun ouders, te rangschikken in de verschillende Pareto fronten, zoals weergegeven in figuur 1.8. De eerste fronten die volledig in de oplossingsverzameling passen worden direct toegevoegd aan de oplossingsverzameling. In figuur 1.7 weer gegeven als $\mathbf{F}_1$ en $\mathbf{F}_2$.



Figuur 1.8: Vier non-dominant Pareto fronts [6]

Voor de eerste verzameling die niet meer volledig in de oplossingsverzameling past, in figuur 1.7 weergegeven als $\mathbf{F}_3$, worden alle indivudu's gesorteerd via een crowding-sort algoritme. Dit algoritme zorgt er voor dat de individu's die overleven de individu's zijn die het meest gespreid zijn. Nadat deze individu's gesorteerd zijn, kiezen we de n eerste uit de gesorteerde lijst, waarbij n het aantal individu's is dat nog kan overleven. De set van nieuwe individu's moet altijd exact even groot zijn als de set

van de ouders.

De crowding-sort procedure $(\mathcal{F}, <_c)$ bestaat uit drie grote stappen.

1. Het aantal individu's in $\mathcal{F}$ is $l = |\mathcal{F}|$. Voor elk individu i in $\mathcal{F}$ geldt $d_i = 0$.
2. Voor elke objective functie $m = 1, 2, \cdots, M$ sorteer de individu's in dalende volgorde van f_m (het slechtste individu komt dus steeds bovenaan te staan). We noemen deze gekregen vector I^m.
3. Voor $m = 1, 2, \cdots M$ nemen we de 2 uiterste individu's en stellen deze gelijk aan oneindig, of $d_{I_1^m} = d_{I_l^m} = \infty$. Voor alle andere individu's $(j = 2, \cdots, (l - 1))$ geldt:
$$d_{I_j^m} = d_{I_j^m} + \frac{f_m^{(I_{j+1}^m)} - f_m^{(I_{j-1}^m)}}{f_m^{\max} - f_m^{\min}} \tag{1.16}$$
Sorteer daarna alle individu's dalend op basis van de d waarde.

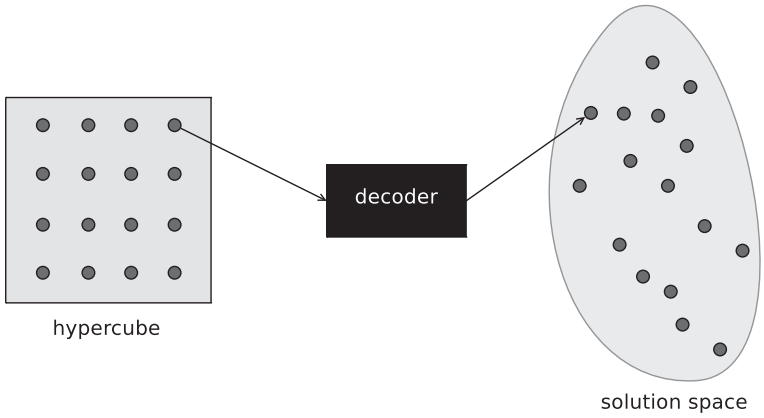
De crowding-sort procedure heeft een complexiteit van $\mathcal{O}(mN\log(N))$, met m het aantal objectives en N het aantal individu's. De complexiteit van de gehele elite non-dominant sorting is typisch $\mathcal{O}(mN^2)$ [7].

1.4 Genetische voorstelling

1.4.1 Beperken van het aantal ongeldige oplossingen

Een van de belangrijkste onderdelen van een GA is de genetische voorstelling van een bepaalde configuratie van de applicaties op het substraat netwerk. Belangrijk hierbij is dat we het chromosoom zodanig opstellen dat we het aantal ongeldige oplossingen dat we kunnen krijgen door een random chromosoom op te stellen, zo veel mogelijk beperken [6]. Een naïeve implementatie zou zijn dat we een binaire matrix als chromosoom zouden voorstellen (1.17). Het aantal ongeldige oplossingen kan zeer hoog oplopen omdat er met geen enkele beperking (beschreven in hoofdstuk 1.1.3) rekening wordt gehouden. Door gebruik te maken van zo'n chromosoom zal het algoritme een zeer trage vooruitgang boeken.

$$C = [\pi_{s_1,1}^{a_1}, \pi_{s_2,1}^{a_1}, \cdots, \pi_{s_{|\mathbf{S}|},1}^{a_1}, \pi_{s_1,2}^{a_1} \cdots, \pi_{s_{|\mathbf{S}|},|\mathbf{N}|}^{a_1}, \cdots \pi_{s_{|\mathbf{S}|},|\mathbf{N}|}^{a_{|\mathbf{A}|}}] \tag{1.17}$$



Figuur 1.9: Decoder vertaalt het individu met biased random key naar een voorstelling in de solution space waarop de fitness berekend kan worden [6]

Een binair chromosoom schrijven dat aan zo veel mogelijk, liefst aan alle, beperkingen voldoet, is moeilijk. Daarom hebben we ons laten inspireren door Gonçalves and Resende [10]. Zij toonden hoe we een random-key implementatie kunnen gebruiken om een genetische voorstellingen te maken. We noemen dit een biased-random key, omdat we een chromosoom maken dat bestaat uit verschillende genen die elk een random nummer tussen nul en een weergeven, dit door het gebruik van floating-point getallen (in onze voorstelling met dubbele precisie). We moeten op deze lijst van random getallen een decoding algoritme uitvoeren die de lijst omzet naar een bruikbare representatie (figuur 1.9). De term biased wordt gebruikt omdat deze voorstelling typisch in een richting werkt. Het is onmogelijk om van een werkbare voorstelling de correcte bijpassende random key te gaan berekenen. In de meeste gevallen zullen dat er dan ook verschillende zijn. Belangrijk met het decoding algoritme is dat het algoritme deterministisch moet zijn. Het spreekt voor zich dat een chromosoom bij elke decoding in dezelfde plaatsing configuratie moet resulteren.

Ik kies ervoor een chromosoon te beschrijven waarmee zoveel mogelijk geldige configuraties voorgesteld kunnen worden en ongeldige configuraties onmogelijk zijn. Hiervoor moet een chromosoomrepresentatie bedacht worden dat enkel geldige oplossingen kan genereren, maar waar er niet te hard gesnoeid wordt in de oplossingensverzameling. Ik wil echter ook zoveel mogelijk oplossingen waarbij zoveel mogelijk

applicaties geplaatst konden worden. Gebaseerd op het werk van Gonçalves et al. [11], die een Biased Random-Key algoritme gebruikten om een scheduling-probleem op te lossen, stel ik volgend chromosoom voor.

$$C = \Big[\underbrace{A_1, A_2, \cdots, A_{|\mathbf{A}|}}_{\text{Volgorde van de applicaties}}, \underbrace{S_1^{1,1}, S_2^{1,1}, \cdots, S_{|\mathbf{S}_1|}^{1,1}, S_1^{1,2}, \cdots, S_{|\mathbf{S}_1|}^{1,\delta_1}, \cdots, S_{|\mathbf{S_A}|}^{|\mathbf{A}|,\delta_{|A|}}}_{\text{Volgorde van de services voor elk duplicaat}}$$
$$\underbrace{N_1^{1,1}, N_2^{1,1}, \cdots, N_{|\mathbf{S}_1|}^{1,1}, N_1^{1,2}, \cdots, N_{|\mathbf{S}_1|}^{1,\delta_1}, \cdots, N_{|\mathbf{S_A}|}^{|\mathbf{A}|,\delta_{|A|}}}_{\text{PM nummer voor elke service van elke duplicaat}} \Big] \tag{1.18}$$

Zoals duidelijk geïllustreerd in formule 1.18 bestaat het chromosoom uit drie grote delen. Het eerste deel van het chromosoom bepaalt de volgorde van de applicaties. We gaan steeds de applicatie met het kleinste A-waarde eerst proberen te plaatsen. Hiermee proberen we een *online* placement na te bootsen in onze decoding. Het tweede deel van het chromosoom is verantwoordelijk voor de volgorde waarop services geplaatst zullen worden. Deze kan voor elke service van elke duplicaat van elke applicatie anders zijn. Het laatste deel van het chromosoom bepaalt voor elke service het PM waarop de service geplaatst zal worden.

1.4.2 Decoding

Figuur 1.10 en Algorithme 1 tonen hoe de decoding van het chromosoom exact in zijn werk gaat. De eerste fase bestaat uit het rangschikken van de applicaties in dalende volgorde op basis van het eerste deel van het chromosoom. De complexiteit van het sorteren van de applicaties kan gebeuren in $\mathcal{O}\big(|\mathbf{A}| \log(|\mathbf{A}|)\big)$.

Vervolgens proberen we in deze volgorde elke applicatie te plaatsen. Een applicatie is geplaatst als de totale availability van de applicatie groter is dan de gevraagde availability. Zolang de minimale availability niet verzekerd kan worden, proberen we een volgende duplicaat te plaatsen. Als het maximaal aantal duplicaten bereikt is en de availability kan nog steeds niet verzekerd worden, wordt alles wat geplaatst of gereserveerd is tijdens het plaatsen van deze applicatie verwijderd. De availability kan eenvoudig berekend worden door het principe van inclusie en exclusie (1.19). We moeten dus enkel de toegevoegde availability berekenen. De complexiteit van

deze berekening is voor duplicaat d gelijk aan $\mathcal{O}(2^{d-1})$. Hierbij merken we op dat d meestal klein blijft.

$$|\cup_{i=1}^{n} A_i| = \sum_{\emptyset \neq J \subseteq \{1,2,\cdots,n\}} (-1)^{|J|-1} \, |\cap_{j \in J} A_j| \tag{1.19}$$

Als we een duplicaat d van een applicatie a proberen te plaatsen, sorteren we eerst alle services van het gegeven duplicaat oplopend op basis van het chromosoom $(S_1^{a,d}, \cdots S_{|\mathbf{S}_a|}^{a,d})$. Vervolgens overlopen we elke service en lijsten we elke PM op waarop een bepaalde service s, voor een bepaalde applicatie a kan op draaien. Hierbij houden we rekening met alle constraints (behalve availability) die opgelijst zijn in 1.1.3. Kortom, we selecteren enkel fysieke machines waarbij de beschikbare RAM en CPU groter is dan de gevraagde RAM en CPU voor deze service. Hierbij houden we rekening met resource hergebruik zoals verduidelijkt in tabel 1.3. Van de machines die dan nog overblijven berekenen we voor elke machine of er een unsplittable path beschikbaar is tussen alle al eerder geplaatste services van dat duplicaat die een virtuele link hebben. Hierbij wordt wederom rekening gehouden met eventueel resource hergebruik. (We gebruiken een enkel path om de complexiteit van het probleem te reduceren.) Als er een path gealloceerd kan worden voor alle al eerder geplaatste services komt deze PM in aanmerking om geselecteerd te worden. Het selecteren van alle mogelijke fysieke machines waarop s kan draaien, heeft een worst-case complexiteit $\mathcal{O}(|\mathbf{S}_a||\mathbf{E}|)$. Hieruit volgt dat het plaatsen van een duplicaat een worst-case complexiteit heeft van $\mathcal{O}(|\mathbf{S}_a|^2|\mathbf{E}|)$.

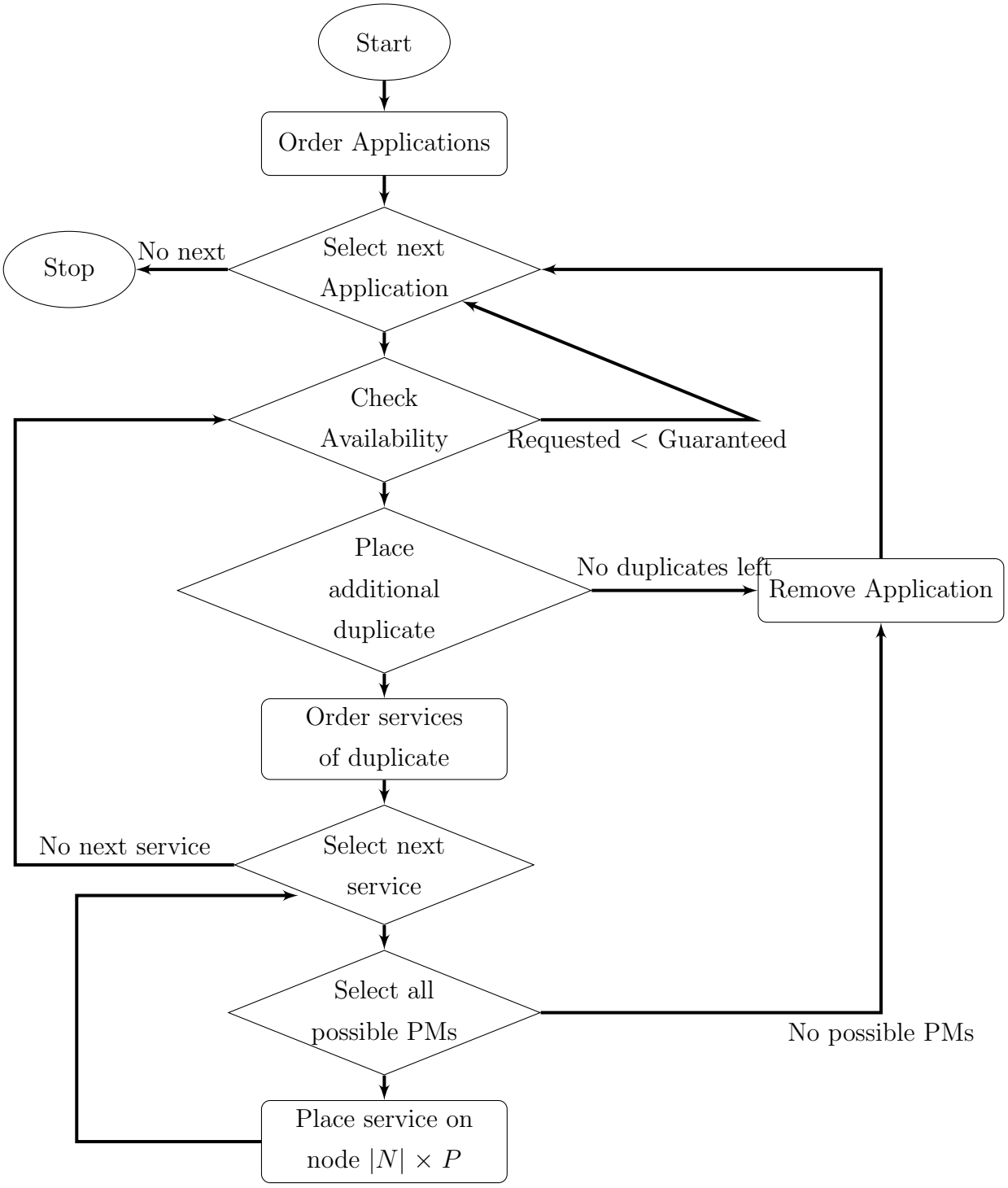
Als de lijst met mogelijke fysieke machines leeg is, betekent dit dat geen enkele machine de gevraagde service nog kan plaatsen. In dat geval stoppen we met het plaatsen van de applicatie en verwijderen we de applicatie volledig van het systeem. De gevraagde availability kan namelijk niet gegarandeerd worden. Als de lijst niet leeg is, gebruiken we het laatste deel van het chromosoom om een PM te selecteren. Dit doen we door eerst de lijst van geslecteerde fysieke nodes te ordenen (bijvoorbeeld op PM-id). Als er n fysieke machines in de lijst staan, vermenigvuldigen n met $N_s^{a,d}$. Deze vermenigvuldiging bepaalt welke PM we uit de lijst gaan selecteren, op dat PM wordt de service geplaatst. Ook worden vanaf dan alle virtuele links van alle services die al geplaatst zijn voor dat duplicaat mee gealloceerd. Deze selectie kan typisch in een complexiteit $\mathcal{O}(n \log n)$, dit omdat de berekeningen die gebeurd zijn tijdens de selectie fase herbruikt kunnen worden.

Als we de plaatsing gedecodeerd hebben, kunnen we van deze representatie eenvoudig alle objectives berekenen.

We moeten hierbij wel opmerken dat het gebruik van de biased-random-key en de decoding die hierboven beschreven staat een surjectieve functie is. Kortom er kunnen verschillende individu's zijn die toch dezelfde plaatsing hebben, na de decoding. Dit kan nadelig zijn voor het GA, dit is echter nog steeds beter dan een voorstelling die oplossingen genereert buiten de oplossingsverzameling. Belangrijk is om er tijdens de selectie- en eliteoperatoren in het GA de individu's in een zo breed mogelijke oplossingsverzameling te kiezen. Zo proberen we te vermijden dat dezelfde individu's de bovenhand zouden nemen. Dit kan eenvoudig door bijvoorbeeld NSGA-II operator te gebruiken [6].

Algoritme 1 Decoding algoritme

1: **procedure** DECODING($A_1 \cdots A_{|\mathbf{A}|}, S_1^{1,1} \cdots S_{|\mathbf{S}_{|\mathbf{A}|}|}^{|\mathbf{A}|,\delta_{|\mathbf{A}|}}, N_1^{1,1} \cdots N_{|\mathbf{S}_{|\mathbf{A}|}|}^{|\mathbf{A}|,\delta_{|\mathbf{A}|}}$)
2: sortByChromosome($\mathbf{A}$, $A_1..A_{|\mathbf{A}|}$)
3: **for each** $a \in \mathbf{A}$ **do**
4: $\delta = 0$
5: $r = 0$
6: **while** $\delta < \delta_a \wedge r < R_a$ **do**
7: $\delta += 1$
8: sortByChromosome($\mathbf{S}_a, S_1^{a,\delta}..S_{|\mathbf{S}_a|}^{a,\delta}$)
9: **for each** $s \in \mathbf{S}_a$ **do**
10: $L =$ allPossiblePM(s)
11: **if** $|L| == 0$ **then**
12: **break**
13: **end if**
14: sortByID(L)
15: $n = N_s^{a,\delta}|L|$
16: placeService(s, n)
17: **end for**
18: $r =$ calculateAvailability(a)
19: **end while**
20: **if** $r < R_a$ **then** removeApplication(a)
21: **end if**
22: **end for**
23: **end procedure**



Figuur 1.10: Flowchart van decoding algoritme

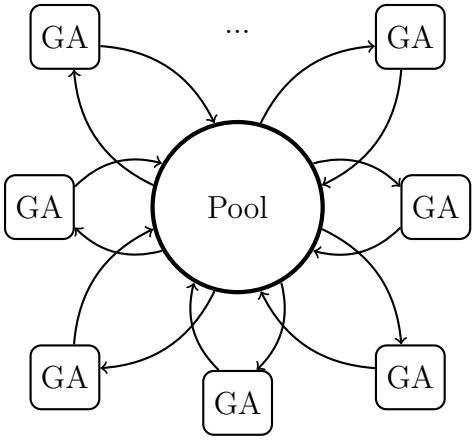
Hoofdstuk 2

Framework

Het framework dat ik geschreven heb voor deze masterproef is een framework voor het oplossen van Multi-Objective genetische algoritmes geschreven in Java 1.8 [15]. Het maakt gebruik van een MongoDB 3.2 database [1]. Hieronder worden de belangrijkste kenmerken en keuzes overlopen. Een manual van het framework is te genereren via de javadoc documentatiegenerator. De readme met alle details voor het opzetten en gebruiken van het framework is ook te vinden bij de sourcecode van het framework (`https://bitbucket.org/rubne/evacloud`).

2.1 Pool model

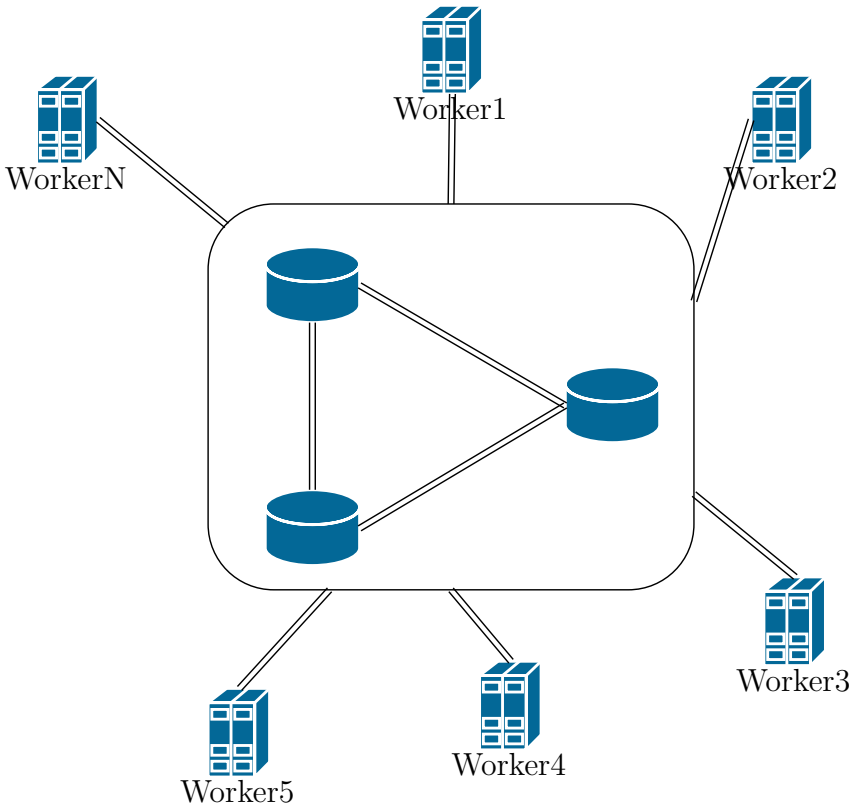
Om het ontwikkelde algoritme te testen en te verbeteren hadden we nood aan een framework waarin we eenvoudig een multi-objective algoritme konden beschrijven en verschillende operatoren en individu's konden testen. We willen echter dat dit algoritme zelf al gedistribueerd kan werken. Tevens willen we een framework dat fault-tolerant is. Gong et al. [12] beschreven de *state-of the-art* gedistribueerde evolutionaire algoritmes. We kozen we er voor om het pool model te gebruiken. Gong et al. [12] beschreven het pool model als *mega schaalbaar en fault-tolerant*. Bij een pool model maken we gebruik van een centrale database. Hierop werken verschillende *workers* die elk een deel van de populatie voor een generatie laten evolueren op hun 'eiland'. Een visuele weergave van het pool model is geïllustreerd in figuur 2.1. Het nadeel aan het pool model is dat de schaalbaarheid gelimiteerd is door de



Figuur 2.1: Basis architectuur pool model

centrale datapool. Als de datapool zich op een gecentraliseerde node bevindt, kan dit ook een single-point of failure vormen. Van het pool model zijn er wel nog verschillende varianten mogelijk. Wij kozen ervoor om te werken met een barrier-based benadering. Deze benadering zegt dat er niet generatie-overschreidend gewerkt kan worden. De oudste generatie moet eerst volledig geëvolueerd zijn alvorens er aan de volgende generatie begonnen kan worden. Dit komt de schaalbaarheid zeker niet ten goede omdat er mogelijks gewacht moet worden tot alle workers klaar zijn met de oudste generatie. Meestal bezorgt deze aanpak ons wel sneller een resultaat omdat de betere individu's doorgaans sneller hun goede genen zullen verspreiden.

Roy et al. [16] en García-Valdez et al. [9] maakten gebruik van het pool model. García-Valdez et al. [9] hebben een python framework geschreven dat gebruik maakt van CouchDB [18] en een reïnsertion manager om de gedeelde pool te beheren. De reïnsertion manager is zo verantwoordelijk om individu's terug beschikbaar te maken als een worker niet meer beschikbaar is (door welke fout dan ook). We konden echter dit framework niet (of moeilijk) gebruiken omdat het framework niet multi-objective is en de gedistribueerde versie was open-source niet te vinden. Dit leidde er toe dat ik een eigen framework geschreven heb met de architectuur zoals gevisualiseerd in figuur 2.2.



Figuur 2.2: Basis architectuur Evacloud

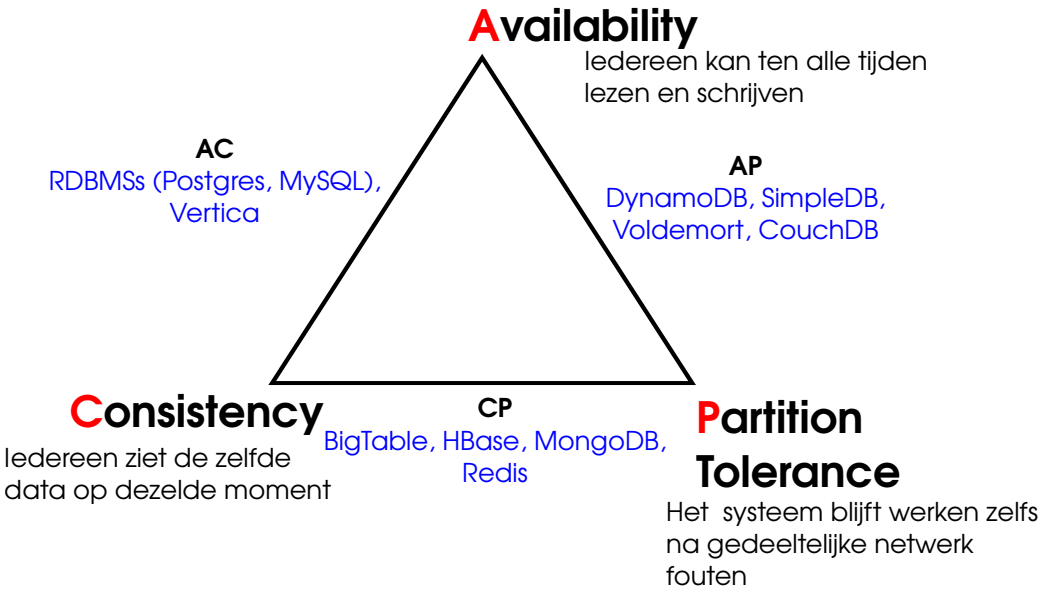
2.2 Opslaan van individu's in een database

Omdat het poolmodel gebaseerd is op een centrale datapool is het belangrijk om een goede centrale database te kiezen. Afhankelijk van het type database is er een extra instantie nodig, net zoals er bij García-Valdez et al. [9] de reïnsertion manager gedefinieerd werd. Deze extra instantie kan extra complexiteit met zich meebrengen zoals het oplossen van een gedistribueerd consensusprobleem.

Eckerstorfer [8] lijstte de performance van verschillende No-SQL databases op. De meeste van deze No-SQL databases zijn eenvoudig gedistribueerd te beheren en te gebruiken. De keuze om een document-based database te gebruiken was snel gemaakt omwille van het makkelijk wegschrijven van het chromosoom en het makkelijk indexeren van individu's op verschillende parameters. Denk hierbij aan het opzoeken van individu's op fitness-waarde of generatie.

Er restte ons de keuze tussen twee databases, MongoDB en CouchDB. De twee andere documentgebaseerde databases die beschreven staan in het werk van Ecker-

storfer [8] zijn ofwel bedoeld voor big data toepassingen, Terrastore, ofwel worden deze niet meer ondersteund, SimpleDB. Het verschil tussen CouchDB en MongoDB is dat CouchDB [18] typisch available en partition-tolerant is in het CAP theorema van Browne [3], terwijl MongoDB [1] partition-tolerant en consistent is in het CAP theorema. Dit is ook geïllustreerd in figuur 2.3.



Figuur 2.3: CAP theorema met database voorbeelden

García-Valdez et al. [9] maakten gebruik van een CouchDB database en voegden er een reïnsertion manager aan toe. Het is zo dat er getracht wordt om de consistentie te verhogen. Deze reïnsertion manager is in dit geval ook noodzakelijk om het gehele framework partition-tolerant te houden. Anders zou het kunnen gebeuren dat, als een gegeven worker uitvalt, de individu's die door die worker behandeld worden, verloren zijn. Het voordeel van het gebruiken van deze setup is dat het zeer eenvoudig is om een random set van n individu's te selecteren uit de database. En als een worker uitvalt, kan er direct een actie ondernomen worden om de individu's terug ter beschikking te maken voor andere workers.

Ik koos er echter voor om een MongoDB database te gebruiken en geen reinsertion

manager te gebruiken. Dit is mogelijk omdat MongoDB een consistente master-slave database is. Als we geen reïnsertion manager beschrijven, moeten we een ander mechanisme beschrijven om het framework partition-tolerant te houden. Wij kozen ervoor om gebruik te maken van een timeout methode. Dit betekent dat als een worker individu's uit de database haalt ze een tijdslot zal alloceren tot wanneer de individu's niet beschikbaar zijn. We zorgen er ook voor dat de implementatie van de worker zodanig is dat een worker enkel oude aan hem gealloceerde individu's kan overschrijven. Zo houden we het systeem consistent en zijn gedeeltelijke falingen mogelijk. Het nadeel van deze veronderstelling is dat het kan gebeuren dat we moeten wachten omdat een worker is uitgevallen en de timeout voor de individu's die hij heeft opgevraagd nog niet vervallen is. Het zou wel uitgebreid kunnen worden, zodat op het moment er nog maar op enkele workers gewacht moet worden andere workers ook starten aan dezelfde set. De worker die dan eerst klaar is met deze set kan de individu's wegschrijven. De andere workers die dezelfde set behandelen worden gestopt van zodra iemand deze set gedaan heeft. Deze redenering wordt ook gebruikt bij Hadoop MapReduce [5].

Om een random selectie van individu's uit de database te halen, maken we gebruik van buckets. Vooraf moet beschreven worden hoeveel buckets er gebruikt zullen worden. Elk individu krijgt op het moment dat het wordt weggeschreven in de database een willekeurig bucketnumber mee. Op het moment dat een worker individu's wil selecteren uit de database zal het een willekeurige bucket (van de oudste generatie) selecteren en alloceren (via het timeout mechanisme). Een voorbeeld van hoe een individu wordt opgelagen kan u vinden in figuur 2.4.

Het spreekt voor zich dat het aantal buckets groter of gelijk aan het aantal workers moet zijn voor een zo performant mogelijke executie. In het beste geval is het aantal buckets gelijk aan het aantal workers in het systeem. Zo kan elke worker exact een bucket op zich nemen en is het aantal individuen per bucket nog groot genoeg. Als het aantal buckets te groot wordt zullen er minder individu's in elke bucket zitten waardoor cross-over en selectie (of elite behoud) soms betere individu's zullen elimineren.

Figuur 2.4 stelt voor hoe een individu wordt opgeslagen in de database. Zoals u kan zien wordt ook de fitness-waarde van het individu mee opgeslagen. Deze operatie voorkomt dat de fitness van eenzelfde individu meerdere malen geëvalueerd wordt.

```
{
  "chromoson": [0, 1, 0, 0, 0, 0, 1, 0, 1, 0],
  "class": "com.uantwerpen.evacloud.worker.individual.
     IndividualClass",
  "gen": 5,
  "fitness": { "0": 23},
  "_bucket": 1,
  "_timeout": 1452653572986,
  "_mongo": 3419,
  "_id": "5695b1fcbabd946a4600d26d"
}
```


Figuur 2.4: Individu voorstelling in de MongoDB database

Deze vorm van caching resulteert in een significante prestatiewinst wanneer het evalueren van de fitness-functie een CPU intensieve taak is.

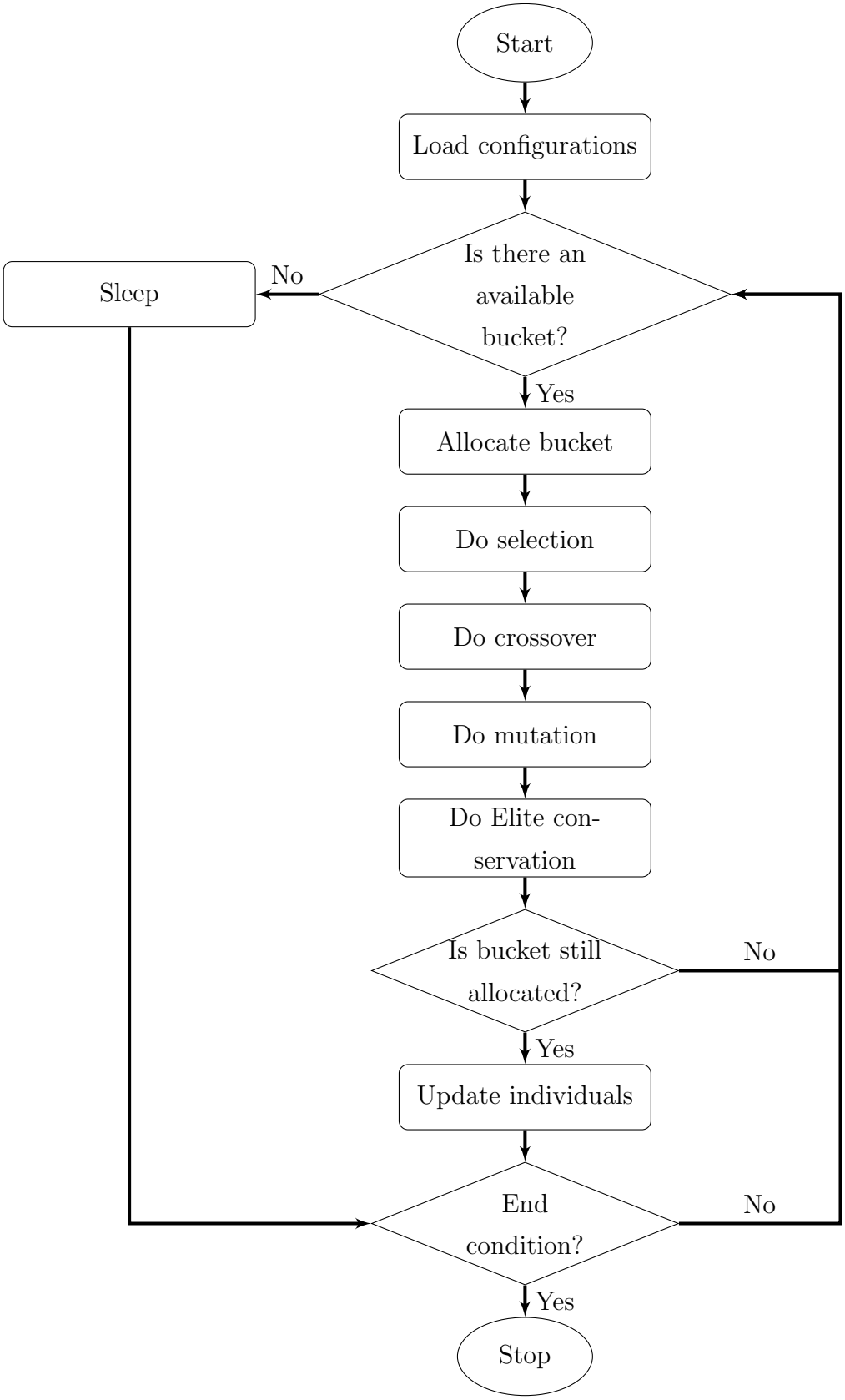
2.3 Worker Architectuur

Het belangrijkste onderdeel van het framework is de worker. Het is de worker die in staat is om individu's te laten evolueren tot een zo goed mogelijke oplossing. De werking van een worker is op zich zeer eenvoudig en visueel weergegeven in figuur 2.5. Op het moment dat een worker wordt opgestart, zal hij verbinding proberen te maken met de database. De parameters van de database (host, port en collection) haalt de worker uit een meegeleverde configuratie file.

In de database wordt verwacht dat een eerste random verzameling individu's aanwezig is samen met de gewenste configuraties. De worker zal de gewenste configuraties uitlezen en al de gewenste configuraties laden. De configuraties bevatten informatie over de operatoren (welke mutator, crossover, selector, fitnessfunctie en eindconditiefunctie), maar ook de klasse van het individu moet beschreven worden.

Vanaf het moment dat de configuraties ingeladen zijn start de worker met het meewerken aan het huidige probleem. De worker zal een volledige niet-gealloceerde bucket (van de oudste generatie) opvragen uit de database en deze alloceren (voor

een bepaalde timeout). De worker wil hierop de geconfigureerde selector, crossover, mutator, eventuele elitemethode en eventuele extra tussentijdse stap uitvoeren. Als dit eiland geëvolueerd is, zal hij de nieuwe individu's wegschrijven naar de database. Zoals eerder vermeld kan een worker enkel individu's overschrijven die eerder gealloceerd werden voor hem. Als de timeout is verlopen zal hij de individu's niet kunnen wegschrijven. De worker zal in dat geval de berekende individu's niet opslaan maar verwijderen. De worker zal terug proberen om een bucket te alloceren die nog niet gealloceerd is en opnieuw een berekening doen. Dit kan dezelfde bucket zijn als geen enkele andere worker deze gealloceerd heeft. Zolang de eindconditie niet geldt, zal hij blijven verder werken.



Figuur 2.5: Flowchart van een worker

Hoofdstuk 3

Resultaten en Discussie

In dit hoofdstuk vergelijken we het ontwikkelde genetisch algoritme met het werk van Spinnewyn et al. [17]. We doen dit door het algoritme, samen met het ontwikkelde framework, uitgebreid te testen op verschillende criteria. Dit hoofdstuk bevat ook de discussie die mogelijk gepaard zal gaan met de testrestultaten.

3.1 Opstelling van de testen

Om het ontwikkelde algroritme te evalueren, vergelijken we het algoritme met de resultaten van Spinnewyn et al. [17]. De testopstelling die we daarom in deze thesis hanteren is ook gebaseerd op de testen beschreven in het werk van Spinnewyn et al. [17].

We maken in deze testopstelling gebruik van de CPU Load Factor (CLF) waarde. Deze waarde is de verhouding van de totale CPU vraag van alle applicaties ten op zichte van de totale beschikbare CPU van de cloud. Zoals weergegeven in vergelijking 3.1.

$$\text{CLF} = \frac{\sum_{a \in \mathbf{A}} \sum_{s \in \mathbf{S}_a} \omega_s}{\sum_{n \in \mathbf{N}} \Omega_n} \tag{3.1}$$

Een enkele run van een test heeft drie variabelen die we meegeven. Het aantal duplicaten, dit kan een, twee of drie zijn, is voor alle applicaties gelijk. De availability

Symbool	Beschrijving
$\mathbf{T}_R$	De verzameling testen, waarbij de applicaties een minimum availability hebben van R.
$\mathbf{T}^\delta$	De verzameling testen waarbij elke applicaties maximaal δ duplicaten mag plaatsen.
$\mathbf{T}_R^\delta$	De verzameling testen, waarbij de applicaties een minimum availability hebben van R en er voor alle applicaties δ duplicaten zijn toegelaten.
$\mathbf{T}_{R,c}^\delta$	Alle testen uit de verzameling $\mathbf{T}_R^\delta$ waarbij de CLF waarde gelijk is aan c

Tabel 3.1: Symbolen gebruikt voor de testcases

die alle applicaties moeten garanderen, deze is 0%, 90% of 99%. En de CLF-waarde die in stapgrote van 0.1 in het interval $[0.1; 1]$ ligt, waarbij we CLF afronden op een significant cijfer. Voor elke CLF waarde hebben we 100 willikeurige testopstellingen gegenereerd die voldoen aan de gevraagde CLF waarde. Deze opstelling wordt dan aangevuld met de availability variable en duplicaten variable. Om de beschrijving van de testen te vergemakkelijken introduceren we een notatie om testen te beschrijven in tabel 3.1.

De verzameling applicaties bestaat telkens uit 10 applicaties. Deze 10 applicaties worden opgebouwd uit minstens 1 service. De services worden gekozen uit een lijst met 3 eerder gegenereerde services waarbij voor elke service s uit de lijst geldt dat $\omega_s \in \{0.2; 1\}$ en $\gamma_s \in \{0.75; 1\}$. Elke service heeft een kans van 60% dat de service behoort tot de gegeven applicatie. Voor elke applicatie a zijn alle services verbonden met elkaar door minstens $|\mathbf{S}_a| - 1$ en maximaal $\frac{1}{2}|\mathbf{S}_a|(|\mathbf{S}_a| - 1)$ bidirectionele links zodat alle services van de applicatie met elkaar verbonden zijn. De bandbreedte van elke virtuele links is uniform verdeeld in het interval $[0.02; 0.04]$.

De cloud-omgeving bestaat uit vijf fysieke machines en acht (ongerichte) fysieke links. De links worden zodanig gekozen dat er eerst een minimale spanningsboom wordt opgezet [2], daarna worden er willekeurig links toegevoegd tot we aan acht links komen. Elke fysieke link heeft een vaste bandbreedte van 1. Voor elke PM n geldt dat $\Omega_n \in \{0.5; 2; 10; 50\}$ en $\Gamma_n \in \{1; 1.5; 2\}$. Voor elke PM en PL geldt dat de kans op falen willekeurig gekozen is uit volgende verzameling: $\{0.0; 0.025; 0.05\}$.

Als instelling voor het framework maken we gebruik van elitebehoud. We gebruiken geen selector, een uniforme crossover waarbij $p = 0.5$ en een standaard mutator die maximaal één gen aanpast met een kans $p = 0.05$. Om elitebehoud te garanderen maken we gebruik van NSGA-II zoals verduidelijkt in hoofdstuk 1.3.1. Als eindconditie geldt dat we stoppen als alle applicaties geplaatst kunnen worden, als de placement ratio van de oplossing voor 20 generaties dezelfde is gebleven of we 101 generaties berekend hebben. De hele configuratie zoals deze in de database geplaatst is voor de testen is te lezen in figuur A.1.

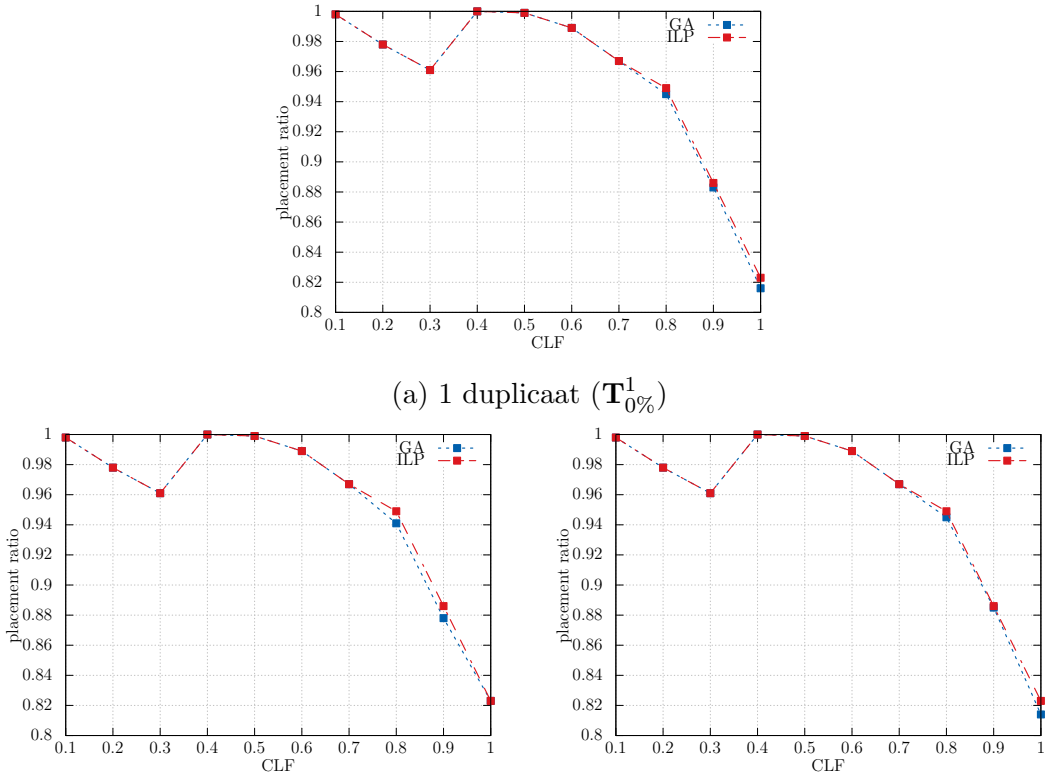
De infrastructuur is dezelfde als in het werk van Spinnewyn et al. [17]. We maken gebruik van de High Performance Cluster (HPC) CalcUA van de universiteit van Antwerpen. Enkel de database kon niet draaien op de HPC. Deze is op een virtuele machine ter beschikking gesteld op een server van de MOSAIC onderzoeksgroep. Dit zorgt er wel voor dat we rekening moeten houden met een mogelijke netwerk delay tussen de database en de workers. Gemiddeld is er een latency van 1.740 milliseconden meetbaar tussen de database server en een node van de HPC.

Naast het genetisch algoritme dat hierboven beschreven staat, hebben we voor elke test ook de optimale oplossing laten berekenen door het ILP beschreven door Spinnewyn et al. [17]. Dit was noodzakelijk om een eerlijke vergelijking van de Placement Ratio (PR) en uitvoeringstijd te maken. Het ILP is aangepast aan de defenitie van clouds en applicaties die in deze thesis beschreven staat. Het bleek dat als de test te complex werd (te veel mogelijkheden in de oplossingsverzameling) het ILP te veel geheugen begon te gebruiken op de machines (meer dan 186 GB). Voor sommige testen ($\{\mathbf{T}^2_{90\%}, \mathbf{T}^3_{90\%}, \mathbf{T}_{99\%}\}$) is het dus onmogelijk om een goede vergelijking te maken.

Een individuele test die na twaalf uren nog steeds geen resultaat had werd gestopt.

We kunnen natuurlijk wel zeggen dat het genetisch algoritme hierdoor veel efficiënter is. Of het genetisch algoritme ook daadwerkelijk de beste oplossing vond is niet te zeggen. We vermoeden dat de oplossingen grotendeels optimaal zullen zijn vermits er vaak nog een volledige plaatsing gevonden wordt, zoals beschreven in de sectie rond placement ratio (3.2).

3.2 Placement Ratio



(a) 1 duplicaat ($\mathbf{T}^1_{0\%}$)

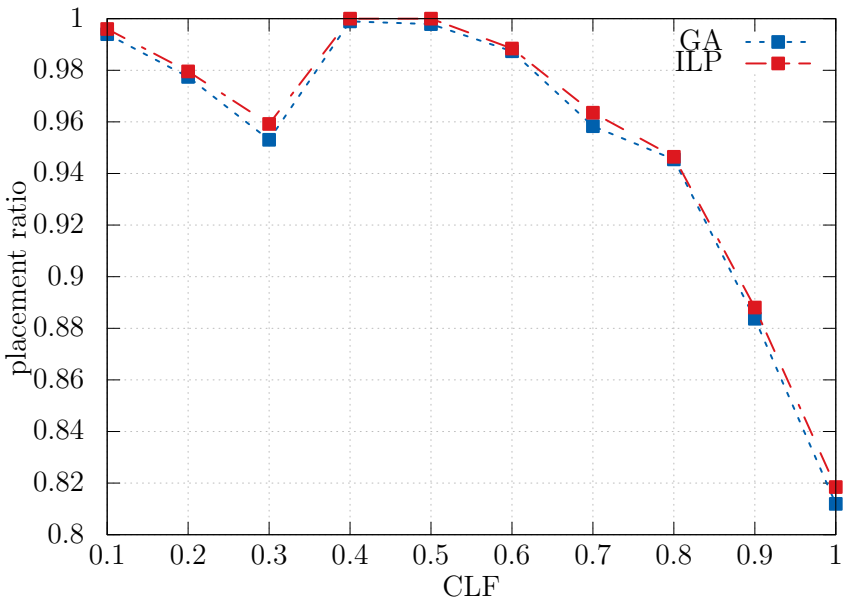
(b) 2 duplicaten ($\mathbf{T}^2_{0\%}$)

(c) 3 duplicaten ($\mathbf{T}^3_{0\%}$)

Figuur 3.1: Placement Ratio: $\mathbf{T}_{0\%}$

Vergelijking van de PR van respectievelijk GA en ILP staat ons toe de optimaliteit van de GA te quantificeren. Als de PR voor het GA en ILP gelijk zijn heeft het GA voor alle testen van die reeks de optimale oplossing gevonden. Merk op dat we voor de reeks $\mathbf{T}^1_{90\%}$ er 45 ILP-testen waren die niet uitvoerbaar waren binnen de 12 uren. Deze worden voor de PR resultaten weggelaten voor zowel het GA als het PR.

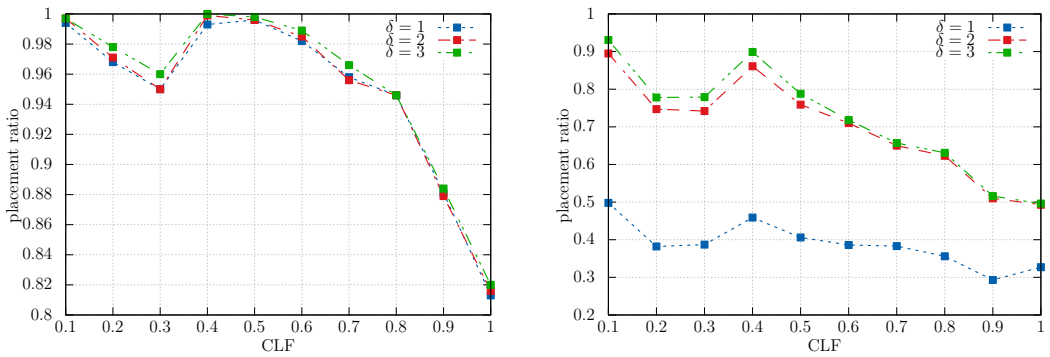
Als we ons focussen op de testen waarbij we geen availability-beperking opstellen ($\mathbf{T}_{0\%}$), zien we in afbeelding 3.1 en tabel B.1 dat de PR voor alle duplicaten van het GA zo goed als gelijk loopt met de PR bepaald door het ILP. We merken enkel een klein verschil bij de hogere CLF waarden. Dit is logisch omdat de oplossingensverzameling groter wordt en het chromosoom langer. Het gemiddeld verschil is nog steeds zeer klein. Dat de PR voor elk duplicaat gelijk zijn is logisch vermits er geen availability-beperking is en we in geen enkel geval meer dan één duplicaat moeten plaatsen.



Figuur 3.2: Placement Ratio: $\mathbf{T}^1_{90\%}$

Als we de vervolgens de minimum availability optrekken naar 90% ($\mathbf{T}_{90\%}$) kunnen we enkel nog de testen vergelijken waarbij er maar één duplicaat geplaatst wordt. In afbeelding 3.2 en tabel B.2 zien we dat de gemiddelde PR niet exact bereikt kan worden. Het verschil is echter zeer klein. Er zijn dus maar enkele testen waarvoor we geen optimale oplossing kunnen vinden. Als we meer duplicaten gaan toevoegen benaderen we vermoedelijk ook nog een optimale oplossing als we dit vergelijken met de resultaten van Spinnewyn et al. [17].

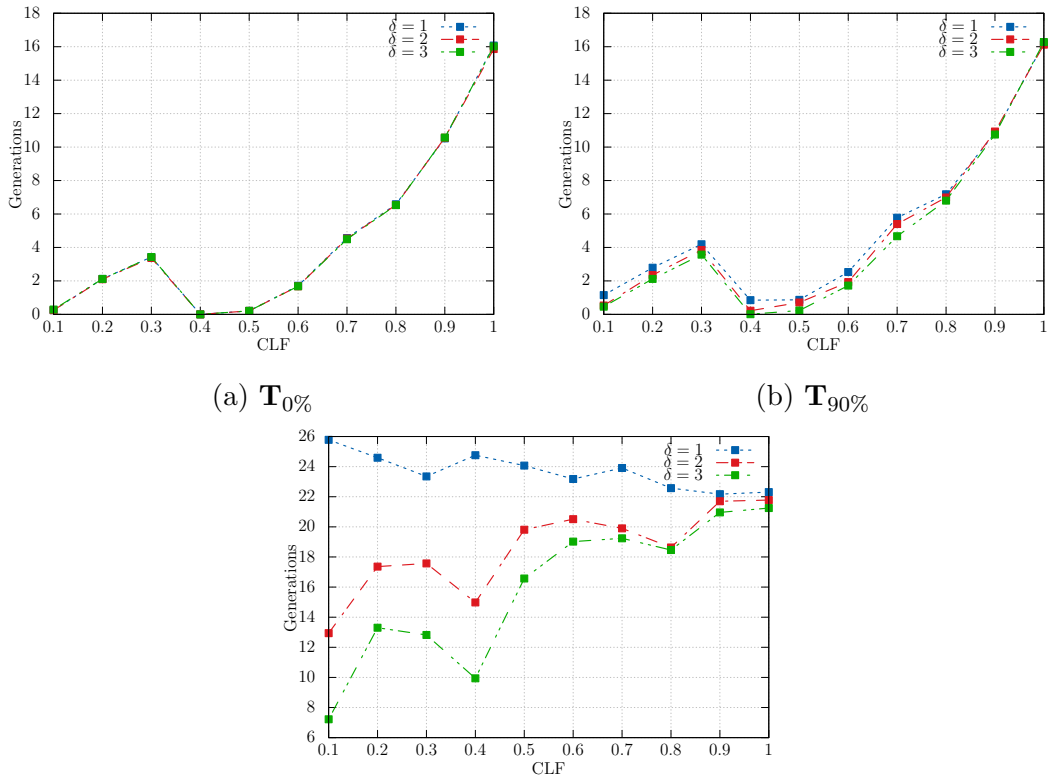
Voor de testen $\mathbf{T}_{99\%}$ was het onmogelijk om het ILP toe te passen op deze problemen.



(a) $\mathbf{T}_{90\%}$ (b) $\mathbf{T}_{99\%}$

Figuur 3.3: Placement Ratio Genetisch algorithme

Het is dus niet met zekerheid te zeggen of het genetisch algoritme wel degelijk dicht bij de optimale oplossing ligt. We zien wel voor de eerste keer een groot verschil tussen de testen waarbij we één duplicaat of meerdere duplicaten gebruiken. Dit is logisch omdat met een availability van 99% voor de eerste keer grotere problemen zijn met availability van applicaties. Hierdoor zal het noodzakelijk zijn om voor sommige applicaties meerdere duplicaten te plaatsen. Dit is onmogelijk als we maar één duplicaat ter beschikking hebben. Ook is het logisch dat hoe meer duplicaten er zijn, hoe makkelijker het is om een applicatie de gevraagde availability te kunnen garanderen. Uiteraard moet er wel voldoende resources zijn om deze applicatie te plaatsen het is dus logisch dat de PR daalt als de CLF stijgt. Dit alles is duidelijk te zien in figuur 3.3.



(a) $\mathbf{T}_{0\%}$ (b) $\mathbf{T}_{90\%}$

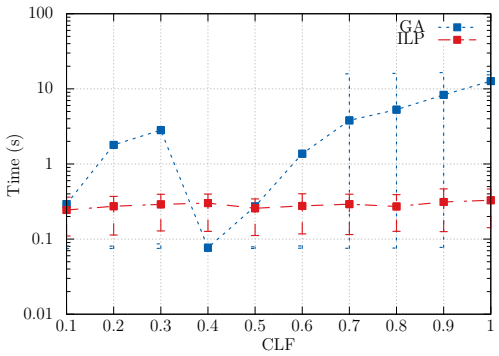
(c) $\mathbf{T}_{99\%}$

Figuur 3.4: Generaties Genetisch algorithme

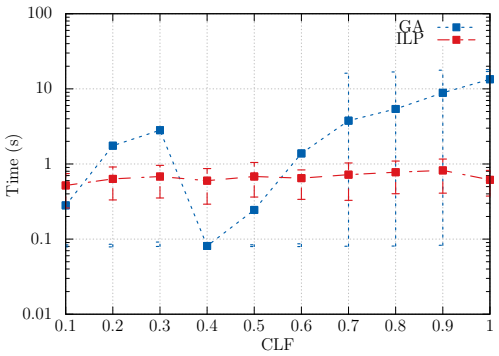
Figuur 3.4 toont hoeveel generaties er gemiddeld nodig zijn voordat het GA stopt. We merken hierbij op dat er maximaal 50 generaties nodig waren alvoren het stop-criterium, vermeld in sectie 3.1, geldig was. We merken op dat voor test $\mathbf{T}_{99\%,1}$ het aantal generaties iets groter is dan 21. Dit duidt er op dat de optimale oplossing al

in een van de eerste generaties aanwezig is.

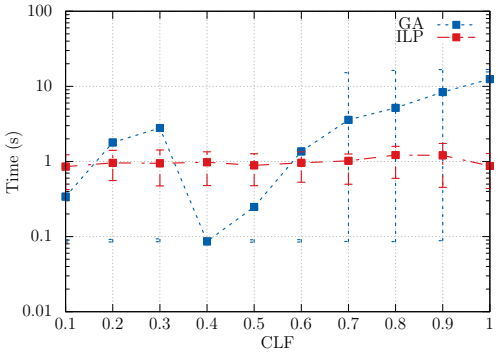
3.3 Executietijd



(a) 1 duplicaat ($\mathbf{T}^1_{0\%}$)



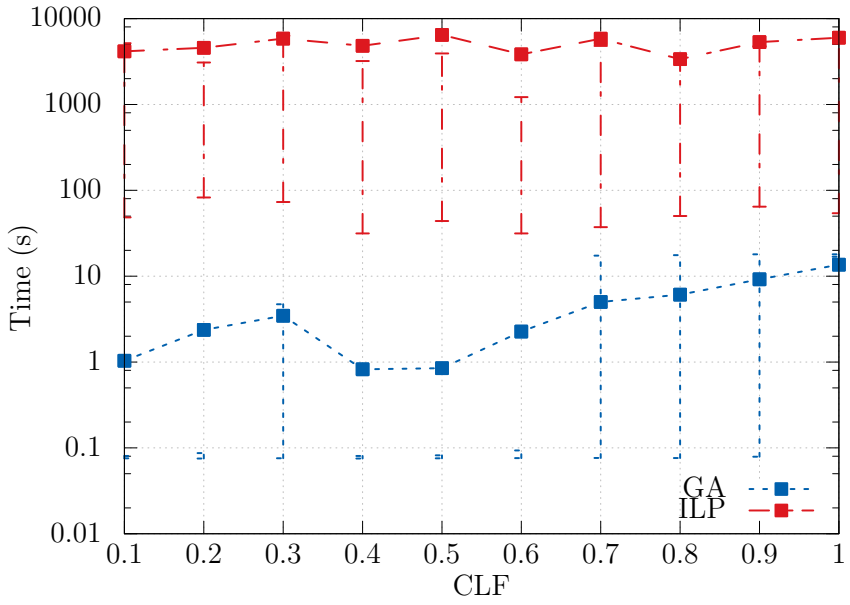
(b) 2 duplicaten ($\mathbf{T}^2_{0\%}$)



(c) 3 duplicaten ($\mathbf{T}^3_{0\%}$)

Figuur 3.5: Executietijd: $\mathbf{T}_{0\%}$

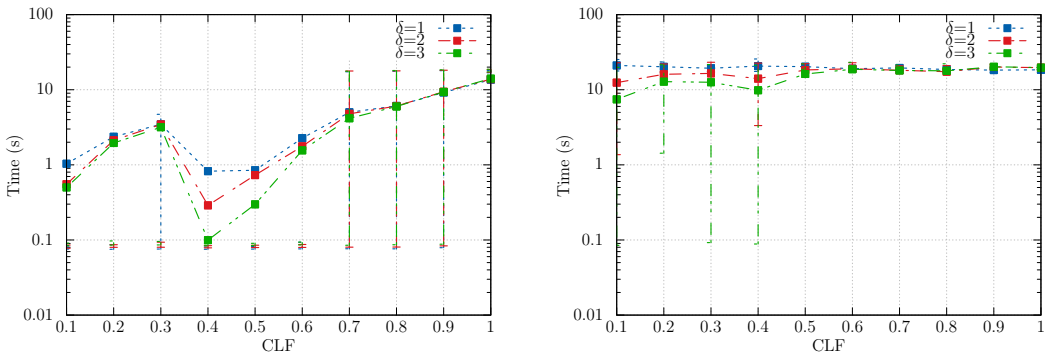
Vermits het doel van het GA is om de ExecutieTijd (ET) voor het verkrijgen van een semi-optimale oplossing te verkleinen worden in deze sectie de resultaten van de executietijden besproken. De ET wordt steeds uitgedrukt in seconden (tenzij er anders wordt vermeld). Voor de ILP-testen die niet binnen de 12 uur een oplossing hadden, rekenen we dat de executietijd 12 uur is, ondanks dat deze waarschijnlijk veel hoger zal liggen. Het ILP is typisch niet oplosbaar in polynomiale tijd en bijgevolg dus niet schaalbaar. Dit is waarom een GA typisch beter is. Vermits er gerichter gezocht wordt naar de oplossing in de oplossingverzameling schaalt het probleem typisch beter. Daarentegen is het wel mogelijk dat de optimale oplossing met een GA nooit gevonden zal worden. Er moet dus gezocht worden naar de beste



Figuur 3.6: Executietijd: $\mathbf{T}^1_{90\%}$

benadering van de optimale oplossing. De resultaten daarvan zijn te vinden in de vorige sectie (3.2). De figuren in dit hoofdstuk hebben ook error bars, deze duiden eerste en derde kwartiel aan.

Voor de testen uit de verzameling $\mathbf{T}_{0\%}$ zien we dat het ILP snel resultaten kan berekenen. Dit omdat het ILP geöptimaliseerd is en de availability-stap in dit geval volledig kan overslaan. Bij het GA wordt wel steeds de availability berekend om te kijken of deze groot genoeg is (wat in het geval van een availability-beperking van 0% altijd zal zijn). Zoals te zien is in afbeelding 3.5 en tabel B.4 merken we op



(a) $\mathbf{T}_{90\%}$ (b) $\mathbf{T}_{99\%}$

Figuur 3.7: Executietijd Genetisch Algorithme

dat de gemiddelde ET voor het GA voor elk duplicaat nagenoeg gelijk blijft. Dit is logisch te verklaren omdat het GA typisch op hetzelfde moment decoding kan stoppen. In alle gevallen is er na één duplicaat te plaatsen voldoende availability beschikbaar. Voor het ILP zien we dat de executietijd stijgt naarmate er meer duplicaten gebruikt kunnen worden. Het ILP exploiteert ook de mogelijkheden om die duplicaten te plaatsen. Hiervan zien we al een lichte stijging van de executietijd.

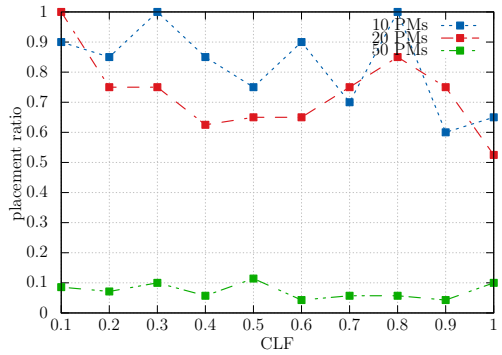
Als we de minimale beschikbaarheid van de applicaties optrekken tot 90% kunnen we enkel nog maar vergelijken met de testen waarbij maar één duplicaat gebruikt wordt ($\mathbf{T}^1_{90\%}$). In afbeelding 3.6 en tabel B.5 zien we dat de ET van het ILP fors gestegen is tot een gemiddelde van 5023 seconden, of wel 1 uur en 24 minuten. Dit alles terwijl voor het GA er maar een kleine stijging van de ET meetbaar is. De gemiddelde ET van $\mathbf{T}^1_{90\%}$ voor het GA is 4.5 seconden. Zoals we vaststelden in sectie 3.2 zagen we dat voor $\mathbf{T}^1_{90\%}$ er maar een kleine afwijking van het optimale PR was. De executie van het GA is echter 1000 keer sneller dan die van het ILP. Hiermee kunnen we vaststellen dat de schaalbaarheid van het GA een enorme verbetering is t.o.v. het ILP.

Zoals eerder aangehaald is een ILP typisch een NP-hard probleem. De schaalbaarheid is dan ook navenant. Het spreekt dan ook voor zich dat als we de minimale availability of het aantal duplicaten optrekken het ILP zeer traag gaat worden. Het GA daarintegen schaalt veel beter. Voor $\mathbf{T}_{99\%}$ zien we in figuur 3.7b en tabel B.6 dat de gemiddelde ET onder de 20 seconden blijft. Zoals doet vermoeden uit de resultaten van de $\mathbf{T}^1_{90\%}$ testen, zal het ILP er nog veel langer over doen.

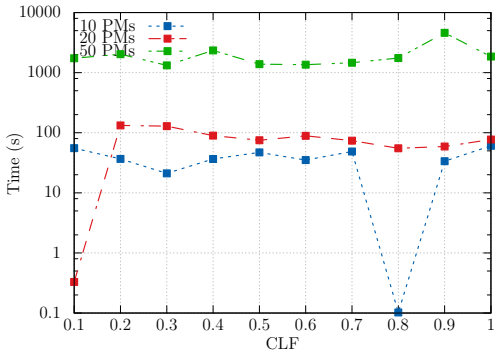
3.4 Schaalbaarheidstest

Om de schaalbaarheid van het GA te testen hebben we naast de vorige testen ook nog drie grotere simulaties opgesteld, gelijkaardig aan de vorige testen. Er werden drie testcases opgesteld, voor elke test case werd één enkele test run uitgevoerd voor een CLF waarde uit de verzameling $\{0,1; 0,2; 0,3; 0,4; 0,5; 0,6; 0,7; 0,8; 0,9; 1,0\}$. Let hierbij op dat we in de resultaten maar één geval per CLF waarde behandelen. We zullen ons vooral focussen op de gemiddelden van de hele test case.

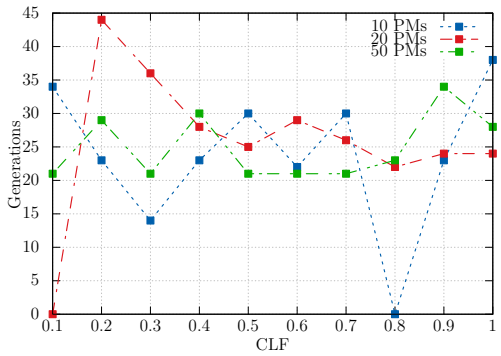
In de eerste testcase zijn er 10 PMs die worden verbonden met 15 PLs. Er moeten 20



(a) Placement Ratio



(b) Executie tijd



(c) Generaties

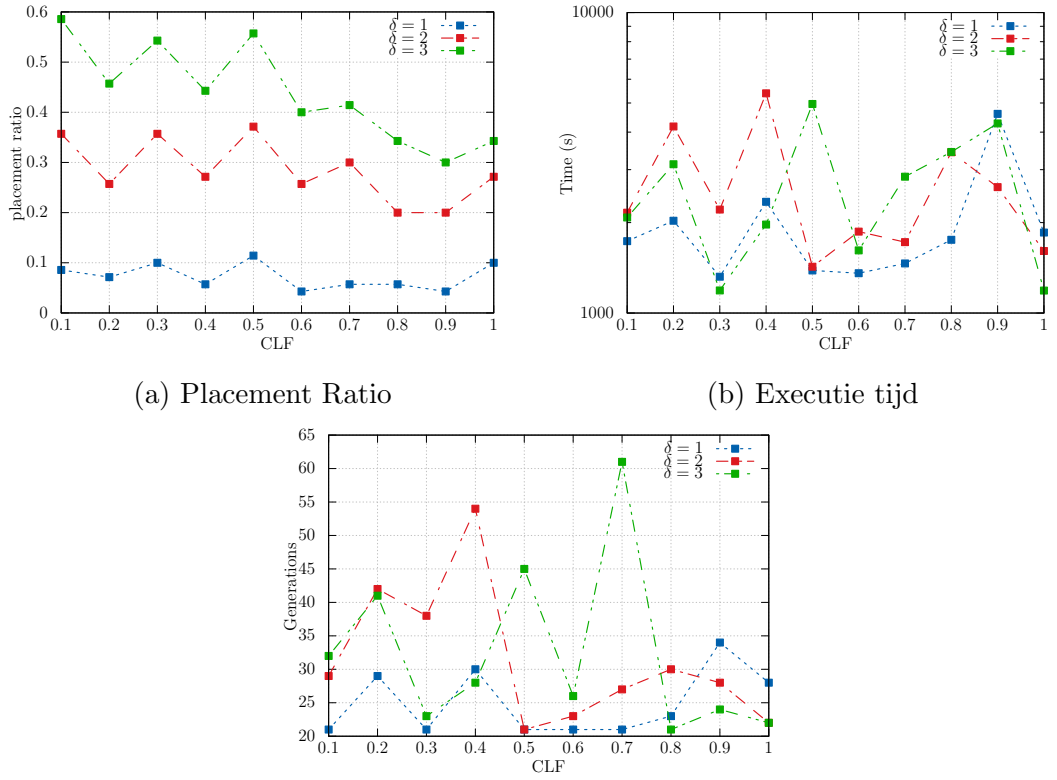
Figuur 3.8: Resultaten schaalbaarheidstesten

applicaties geplaatst worden. Er zijn 6 verschillende services en elke service heeft een kans van 60% om te behoren tot een gegeven applicatie. Voor de applicaties staan we maximaal één duplicaat toe en er moet een minimum availability van 90% gehaald worden. Alle parameters voor de services (ω_s, γ_s), VLs (β_{s_1,s_2}), PMs $(\Omega_n, \Gamma_n, p_n^N)$ en PLs (B_e, p_e^E) blijven gelijk aan de waarden die ze hadden in de vorige testen.

In de tweede testcase wordt het aantal applicaties opgetrokken tot 40, er zijn 20 PMs die verbonden worden met 30 PLs.

En in de laatste testcase zijn er 70 applicaties, 16 services, 50 PMs die verbonden worden met 75 PLs.

Als we de resultaten van deze testen bekijken in figuur 3.9, valt er ons direct op dat het PR van de test met 50 PMs enorm laag is. Dit is eenvoudig te verklaren doordat het aantal services per applicatie is toegenomen, terwijl het aantal CPU op de PMs niet is toegenomen. Daardoor zal de availability van de applicatie fors dalen. Vermits er maar één duplicaat per applicatie is toegelaten zal het zeer moeilijk



(a) Placement Ratio

(b) Executie tijd

(c) Generaties

Figuur 3.9: Resultaten schaalbaarheidstest 50 nodes

worden om applicaties correct te plaatsen.

Eveneens kunnen we vaststellen dat de executietijd voor het probleem met 50 PMs (gemiddelde van 1984 seconden of 33 minuten) 25 keren langer is dan de executie van de test met 10 en 20 PMs (gemiddelde van respectievelijk 37.5 en 78 seconden). Dit is het gevolg van het aantal PMs, applicaties en services dat fors stijgt. Voor elk service s van elk duplicaat dat geplaatst wordt, moeten alle PMs nagekeken of ze service s kunnen plaatsen. Het spreekt voor zich dat dit een negatieve invloed heeft op de performantie.

Als we naar het aantal berekende generaties kijken, stellen we vast dat het aantal generaties die nodig zijn om een gegeven probleem op te lossen ongeveer gelijk blijft bij alle verschillende problemen, dit ondanks dat het chromosoom langer wordt.

Als we bij de test met 50 PMs het aantal duplicaten verhogen, kunnen we vaststellen dat het PR ook toeneemt, zoals te zien in figuur 3.9a. Het valt wel op dat de gemiddelde executietijd niet zo fors toenoemt. Wanneer we één duplicaat hebben,

is de gemiddelde executietijd ongeveer 33 minuten, wanneer we 2 of 3 duplicaten hebben is de gemiddelde executietijd ongeveer 44 minuten.

Het voorgestelde GA is schaalbaar in functie van de gevraagde availability per applicatie en het maximale aantal duplicaten. Als het aantal applicaties stijgt, stijgt bij gevolg ook het aantal services. Dit heeft als gevolg dat het aantal services die er moeten geplaatst worden niet meer lineaire is, waardoor het algoritme minder schaalt en de executietijden kunnen oplopen.

3.5 Verbeteringen voor het genetisch algoritme

Ondanks het feit dat we hebben aangetoond dat het genetisch algoritme veel schaalbaarder is dan het ILP, zijn er mogelijkheden om het algoritme te verbeteren.

3.5.1 Netwerkallocatie

Op dit moment wordt tijdens het decoden van een individu gebruik gemaakt van een shortest-path-algoritme om de VL te plaatsen. Uiteraard is dit een grote vereenvoudiging van de werkelijkheid. In de werkelijkheid is het namelijk mogelijk om een multi-path-verbinding te hebben of kunnen andere routering-algoritmes gebruikt worden. Een uitbreiding van het GA zou kunnen zijn om deze andere routering-algoritmes aan te nemen en multi-path-routering te ondersteunen. Dit gaat typisch de complexiteit enorm doen stijgen omdat deze berekeningen voor alle nodes waarop al services van een applicatie geplaatst zijn, uitgevoerd moeten worden. Tevens zou er voor het alloceren van het netwerk ook gebruik gemaakt kunnen worden van een heuristiek om de allocatie van het netwerk te verbeteren. Hierbij moeten we wel opletten dat de heuristiek deterministisch is, vanwege dat de eigenschap dat een decoding van een individu deterministisch moet zijn. Het zou mogelijk zijn om het chromosoom uit te breiden zodat de keuzes die gemaakt worden bij de netwerkheuristiek mee in het individu zitten waardoor de keuzes deterministisch worden.

3.5.2 Chromosoom inkorten

Het is algemeen geweten dat de lengte van een chromosoom invloed heeft op de schaalbaarheid van het probleem [6]. Op dit moment wordt het chromosoom snel lang als er meerdere duplicaten gebruikt worden en als het aantal applicaties stijgt. Verder onderzoek is noodzakelijk om te kunnen zeggen of het inkorten van het chromosoom mogelijk is en toch de oplossingsverzameling zo groot mogelijk te houden.

3.5.3 Snelheid verbeteren

Onderdeel	Gemiddelde executietijd (s)	Ratio executietijd (%)
Initialisatie	0.090135	1.067119
Individu's laden	0.226543	2.682058
Selectieoperator	0.001154	0.013662
Crossover-operator	0.024633	0.291636
Mutatieoperator	0.007824	0.092625
Decoding	1.191827	14.110116
Elite stap	0.210262	2.489303
Individu's wegschrijven	6.694233	79.253481
Totaal	8.446611	100

Tabel 3.2: Gemiddelde executietijden per operator

In tabel 3.2 is weergegeven hoeveel tijd (absoluut en relatief) er naar welke operator gaat voor alle 9000 testen van het GA in het Evacloud framework. In deze resultaten merken we al snel op dat bijna 80% van de executietijd van het GA naar het opslaan van de individu's gaat. Dit is eenvoudig te verklaren door het feit dat we gebruik maken van een consistente partitietolerante database zoals mongoDB [1]. Naast de netwerk-delay die geïntrudoceerd wordt, moet een worker ook wachten totdat de individu's correct zijn opgeslagen op disk bij de master van de database cluster. Ook is het Evacloud framework zo gemaakt dat een worker zijn eigen gealloceerde individu's gaat updaten, wat maakt dat deze eerst gealloceerd worden in een critische sectie door de database. Deze twee feiten zorgen ervoor dat de operatie om

Onderdeel	Gemiddelde executietijd (s)	Ratio executietijd (%)
Initialisatie	0.384235	0.019364
Individu's laden	23.247930	1.171588
Selectie operator	0.002973	0.000150
Crossover operator	0.453290	0.022844
Mutatie operator	0.195387	0.009847
Decoding	1704.961242	85.922154
Elitestap	0.501488	0.025273
Individu's wegschrijven	254.562699	12.828781
Totaal	1984.309245	100

Tabel 3.3: Gemiddelde executietijden per operator voor test met 50 PMs

individu's weg te schrijven lang duurt. Vooral omdat de worker dan in *bussy-wait*-staat verkeert. Mijns inziens bestaat er geen consistente gedistribueerde database die gericht is op het opslaan van kleine datasets (zoals de individu's in ons geval). Meer onderzoek naar andere databases (en het gevolg dat deze hebben) is vereist om de performantie van het GA te verbeteren.

Naast de dure database-operaties kunnen we ook vaststellen dat de decoding een intensieve operatie is. In tegenstelling tot de database-operaties verkeert de decoding fase niet in een *bussy-wait*-staat, maar zal er enorm veel CPU vereist worden. De reden hiervoor is dat er voor elke service die we willen plaatsen er verschillende vergelijkingen zijn met alle andere nodes. Er wordt namelijk elke keer voor alle nodes gekeken of het mogelijk is om de service op deze node te plaatsen, waarbij er gekeken wordt naar CPU, geheugen en de bandbreedte met al eerder geplaatste services (van dezelfde applicatie). Een mogelijke verbetering voor het intensieve rekenwerk tijdens de decoding is om het aantal mogelijke nodes waarop services van een applicatie geplaatst kunnen worden te beperken. Dit kan door middel van een heuristiek, of door enkel PMs te testen die in het netwerk dichtbij al eerder gebruikte PMs liggen. Belangrijk is om na te gaan of de decoding deterministisch blijft.

In tabel 3.3 kunnen we vaststellen dat voor de grotere test met 50 PMs, zoals beschreven in sectie 3.4, de decoding het meeste tijd in beslag neemt, gemiddeld zo'n 86%. Zoals eerder beschreven is dit resultaat intuitief makkelijk te begrijpen,

vermits de decoding voor elke service van elk duplicaat dat we proberen te plaatsen van elke applicatie alle mogelijk PMs moet overlopen. In complexiteit kunnen we dit als volgt uitdrukken:

$$\mathcal{O}(|\mathbf{N}| \sum_{a \in \mathbf{A}} |\mathbf{S}_a| \delta_a)$$

Een methode gebruiken die er voorzorgt dat niet elke keer alle nodes overlopen moeten worden zal de schaalbaarheid van het GA dus sterk ten goede komen.

DEEL II

Conclusie

Besluit

Om applicaties te plaatsen in een mobile heterogene cloud is een plaatsingsalgoritme dat rekening houdt met availability, CPU, geheugen en bandbreedte noodzakelijk. Een oplossing moet ook te bepalen zijn in een kleine tijdspanne. Het werk van Spinnewyn et al. [17] leverde een ILP en een model waarop we ons konden baseren om een schaalbare heuristiek te maken in de vorm van een GA. In deze thesis hebben we een chromosoom beschreven dat een deterministische plaatsing van gegeven applicaties in een gegeven cloudomgeving in de vorm van een biased-random-key chromosoom [10]. Het chromosoom dat we beschreven is op een determinisitsche manier te decoden via een decoding algoritme dat beschreven staat in sectie 1.4.2.

Om het voorgestelde chromosoom te testen, ontwikkelde we het framework Evacloud. Dit framework is gebasseerd op een gedistribueerd pool model [12]. De testresultaten toonden aan dat we met het gegeven GA voor de meeste problemen de optimale oplossing vinden. We merkten ook op dat als we complexere problemen opstelden, het ILP-model snel grote executietijden kende, of zelfs vastliep, terwijl de executietijd van het GA langzamer steeg. Voor de meest complexe testcase die we konden uitvoeren met het ILP ($\mathbf{T}^1_{90\%}$) kwam het GA wel 1000 keer sneller tot een resultaat. Hieruit konden we concluderen dat het GA veel beter schaalbaar is dan het ILP in verhouding met de minimale availability per applicatie en het aantal duplicaten. Als de problemen echter groter worden, zagen we dat de executietijd van het GA ook fors kon stijgen omdat het aantal services, applicaties en nodes steeg waardoor de decoding veel tijd in beslag nam.

Meer onderzoek is echter noodzakelijk. Als eerste gebeurt de allocatie van de PLs door middel van een shortest-path-algoritme. In de werkelijkheid is het perfect mogelijk om andere routeringen te ondersteunen. Heuristieken voor de netwerkallocatie zijn een mogelijke oplossing. Als tweede is er verder onderzoek nodig naar de mo-

gelijkheid om het chromosoom in te korten. Vermits het chormosoom een lengte heeft van $|\mathbf{A}| + 2\sum_{a\in\mathbf{A}} |D_a||S_a|$ wordt het chromosoom snel lang. Dit kan als gevolg hebben dat verschillende chromosonen dezelfde applicatieplaatsing voorstellen. Zo'n surjectie is typisch nadelig voor een GA. Verder onderzoek moet ook gedaan worden naar de beste manier om individu's op te slaan in een consistente gedistribueerde database. Het gebruik van MongoDB gaf ons correcte resultaten maar omdat elke worker moest wachten tot de database de nieuwe individu's op schijf had weggeschreven is dit een dure operatie die lang kan duren.

Bibliografie

[1] 10gen Inc. Mongodb, sep 2015. URL `http://www.mongodb.org`.

[2] A. Broder. Generating random spanning trees. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 442–447. IEEE, 1989.

[3] J. Browne. Brewer's cap theorem. *J. Browne blog*, 2009.

[4] M. Caramia and P. Dell'Olmo. *Multi-objective Management in Freight Logistics*. Springer-Verlag London, first edition, 2008.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] K. Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[7] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. *Lecture notes in computer science*, 1917:849–858, 2000.

[8] F. Eckerstorfer. Performance of nosql databases, 2011.

[9] M. García-Valdez, L. Trujillo, J.-J. Merelo, F. F. de Vega, and G. Olague. The evospace model for pool-based evolutionary algorithms. *Journal of Grid Computing*, pages 1–21, 2014.

[10] J. F. Gonçalves and M. G. Resende. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17(5):487–525, 2011.

[11] J. F. Gonçalves, J. J. de Magalhães Mendes, and M. G. Resende. A hybrid genetic algorithm for the job shop scheduling problem. *European journal of operational research*, 167(1):77–95, 2005.

[12] Y.-J. Gong, W.-N. Chen, Z.-H. Zhan, J. Zhang, Y. Li, and Q. Zhang. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing*, 2015.

[13] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[14] B. Jennings and R. Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, pages 1–53, 2014.

[15] Oracle. Oracle technology network for java developers, sep 2015. URL `http://www.oracle.com/technetwork/java/index.html`.

[16] G. Roy, H. Lee, J. L. Welch, Y. Zhao, V. Pandey, and D. Thurston. A distributed pool architecture for genetic algorithms. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 1177–1184. IEEE, 2009.

[17] B. Spinnewyn, B. Braem, and S. Latré. Fault-tolerant application placement in heterogeneous cloud environments. 2015.

[18] The Apache Software Foundation. Apache couchdb: The apache couchdb project., sep 2015. URL `http://couchdb.apache.org`.

[19] A. M. Turing. I.computing machinery and intelligence. *Mind*, LIX(236): 433–460, 1950. doi: 10.1093/mind/LIX.236.433. URL `http://mind.oxfordjournals.org/content/LIX/236/433.short`.

[20] R. Yu and T. Watteren. Reliable, low power wireless sensor networks for the internet of things: Making wireless sensors as accessible as web servers. Technical report, Dust Networks Product Group, Linear Technology Corp., 2013.

DEEL III

Bijlagen

Bijlage A

Evacloud configuratie

```
{
    "_id" : "worker",
    "individuals" : "com.uantwerpen.evacloud.worker.
       individual.CloudIndividual",
    "fitness" : "com.uantwerpen.evacloud.worker.fitness.
       CloudFitness",
    "selector" : "com.uantwerpen.evacloud.worker.selector
       .NoSelector",
    "crossover" : "com.uantwerpen.evacloud.worker.
       crossover.DefaultCrossover",
    "mutator" : "com.uantwerpen.evacloud.worker.mutator.
       DefaultMutator",
    "elitist" : "com.uantwerpen.evacloud.worker.elitist.
       CloudNdsIIElitist",
    "endCondition" : "com.uantwerpen.evacloud.worker.
       endconditions.OrCondition",
    "buckets" : NumberInt(1),
    "onIsland" : {
        "runs" : NumberInt(1),
        "individuals" : NumberInt(30)
    }
}
{
    "_id" : "Crossover.DefaultCrossover",
    "p" : 0.5
}
{
    "_id" : "mutator.DefaultMutator",
    "pm" : 0.05,
    "n" : NumberInt(1)
}
{
    "_id" : "elitist.CloudNdsIIElitist",
    "p" : 0.75
}
{
    "_id" : "EndCondition.OrCondition",
    "first" : "com.uantwerpen.evacloud.worker.
       endconditions.AllPlacedEndCondition",
    "second" : "com.uantwerpen.evacloud.worker.
       endconditions.StableCondition"
}
{
    "_id" : "EndCondition.AllPlacedEndCondition",
    "n" : NumberInt(100)
}
{
    "_id" : "EndCondition.StableCondition",
    "n" : NumberInt(20)
}
```

Figuur A.1: Test configuratie evacloud

Bijlage B

Test resultaten

Duplicaten	CLF	Gemiddelde PR GA	Gemiddelde PR GA	Verschil
1	0.1	0.998	0.998	0.000
	0.2	0.978	0.978	0.000
	0.3	0.961	0.961	0.000
	0.4	1.0	1.0	0.000
	0.5	0.999	0.999	0.000
	0.6	0.989	0.989	0.000
	0.7	0.967	0.967	0.000
	0.8	0.945	0.947	0.002
	0.9	0.883	0.886	0.006
	1.0	0.816	0.823	0.007
2	0.1	0.998	0.998	0.000
	0.2	0.978	0.978	0.000
	0.3	0.961	0.961	0.000
	0.4	1.0	1.0	0.000
	0.5	0.999	0.999	0.000
	0.6	0.989	0.989	0.000
	0.7	0.967	0.967	0.000
	0.8	0.941	0.949	0.008
	0.9	0.878	0.886	0.008
	1.0	0.823	0.823	0.000
3	0.1	0.998	0.998	0.000
	0.2	0.978	0.978	0.000
	0.3	0.961	0.961	0.000
	0.4	1.0	1.0	0.000
	0.5	0.999	0.999	0.000
	0.6	0.989	0.989	0.000
	0.7	0.967	0.967	0.000
	0.8	0.945	0.949	0.003
	0.9	0.885	0.886	0.001
	1.0	0.814	0.823	0.009

Tabel B.1: Gemiddelde Placement Ratio: Availability 0%

Duplicaten	CLF	Gemiddelde PR GA	Gemiddelde PR GA	Verschil
1	0.1	0.994	0.996	0.002
	0.2	0.977	0.980	0.003
	0.3	0.953	0.959	0.006
	0.4	0.999	1.0	0.001
	0.5	0.998	1.0	0.002
	0.6	0.987	0.988	0.001
	0.7	0.958	0.964	0.006
	0.8	0.945	0.946	0.001
	0.9	0.884	0.888	0.004
	1.0	0.811	0.818	0.007
2	0.1	0.997	?	?
	0.2	0.971	?	?
	0.3	0.950	?	?
	0.4	0.999	?	?
	0.5	0.996	?	?
	0.6	0.985	?	?
	0.7	0.956	?	?
	0.8	0.946	?	?
	0.9	0.879	?	?
	1.0	0.816	?	?
3	0.1	0.997	?	?
	0.2	0.978	?	?
	0.3	0.960	?	?
	0.4	1.0	?	?
	0.5	0.998	?	?
	0.6	0.989	?	?
	0.7	0.966	?	?
	0.8	0.946	?	?
	0.9	0.884	?	?
	1.0	0.820	?	?

Tabel B.2: Gemiddelde Placement Ratio: Availability 90%

Duplicaten	CLF	Gemiddelde PR GA
1	0.1	0.498
	0.2	0.382
	0.3	0.387
	0.4	0.459
	0.5	0.406
	0.6	0.386
	0.7	0.383
	0.8	0.356
	0.9	0.293
	1.0	0.327
2	0.1	0.895
	0.2	0.747
	0.3	0.742
	0.4	0.861
	0.5	0.759
	0.6	0.710
	0.7	0.650
	0.8	0.623
	0.9	0.510
	1.0	0.493
3	0.1	0.931
	0.2	0.778
	0.3	0.779
	0.4	0.899
	0.5	0.788
	0.6	0.718
	0.7	0.657
	0.8	0.631
	0.9	0.516
	1.0	0.496

Tabel B.3: Gemiddelde Placement Ratio: Availability 99%

Duplicaten	CLF	Gemiddelde ET (s) GA	Gemiddelde ET ILP (s)	Verschil (s)
1	0.1	0.2927	0.2440	0.0487
	0.2	1.7954	0.2735	1.5219
	0.3	2.8213	0.2902	2.5311
	0.4	0.0768	0.3015	-0.2247
	0.5	0.2727	0.2575	0.0152
	0.6	1.3681	0.2767	1.0914
	0.7	3.8046	0.2918	3.5128
	0.8	5.2839	0.2719	5.012
	0.9	8.3201	0.3123	8.0078
	1.0	12.6776	0.3289	12.3487
2	0.1	0.2810	0.5183	-0.2373
	0.2	1.7506	0.6330	1.1176
	0.3	2.8142	0.6806	2.1336
	0.4	0.0810	0.5991	-0.5181
	0.5	0.2445	0.6818	-0.4373
	0.6	1.3831	0.6474	0.7357
	0.7	3.7710	0.7207	3.0503
	0.8	5.3960	0.7763	4.6197
	0.9	8.8583	0.8219	8.0364
	1.0	13.4199	0.6154	12.8045
3	0.1	0.3406	0.8548	-0.5142
	0.2	1.7907	0.9587	0.832
	0.3	2.7898	0.9457	1.8441
	0.4	0.0863	0.9764	-0.8901
	0.5	0.2483	0.8881	-0.6398
	0.6	1.3597	0.9601	0.3996
	0.7	3.5795	1.0201	2.5594
	0.8	5.1715	1.2180	3.9535
	0.9	8.4046	1.2067	7.1979
	1.0	12.4740	0.8723	11.6017

Tabel B.4: Gemiddelde Executietijd: Availability 0%

Duplicaten	CLF	Gemiddelde ET (s) GA	Gemiddelde ET ILP (s)	Verschil (s)
1	0.1	1.0352	4154.6194	-4153,5842
	0.2	2.3669	4574.5309	-4572.1640
	0.3	3.4598	5850.6294	-5847.1696
	0.4	0.8236	4828.8767	-4828.0531
	0.5	0.8473	6455.4704	-6454.6231
	0.6	2.2669	3833.9388	-3831.6719
	0.7	5.0094	5839.2345	-5834.2251
	0.8	6.0972	3369.3313	-3363.2341
	0.9	9.1902	5323.8511	-5314.6609
	1.0	13.6143	5995.6838	-5982.0695
2	0.1	0.5523	?	?
	0.2	2.1235	?	?
	0.3	3.4486	?	?
	0.4	0.2890	?	?
	0.5	0.7299	?	?
	0.6	1.7648	?	?
	0.7	4.7772	?	?
	0.8	6.0647	?	?
	0.9	9.4093	?	?
	1.0	13.8237	?	?
3	0.1	0.5006	?	?
	0.2	1.9517	?	?
	0.3	3.1681	?	?
	0.4	0.0995	?	?
	0.5	0.2980	?	?
	0.6	1.5586	?	?
	0.7	4.1676	?	?
	0.8	5.9798	?	?
	0.9	9.3499	?	?
	1.0	14.0586	?	?

Tabel B.5: Gemiddelde Executietijd: Availability 90%

Duplicaten	CLF	Gemiddelde ET (s) GA
1	0.1	21.0481
	0.2	20.2178
	0.3	19.3159
	0.4	20.6349
	0.5	20.2872
	0.6	19.0599
	0.7	19.5112
	0.8	18.5866
	0.9	18.2532
	1.0	18.4013
2	0.1	12.4316
	0.2	16.0437
	0.3	16.5071
	0.4	14.0517
	0.5	18.3376
	0.6	19.0226
	0.7	18.1494
	0.8	17.3937
	0.9	20.0622
	1.0	19.6950
3	0.1	7.4401
	0.2	12.8050
	0.3	12.5708
	0.4	9.8355
	0.5	16.1821
	0.6	18.7423
	0.7	18.0548
	0.8	17.8593
	0.9	20.2314
	1.0	19.6785

Tabel B.6: Gemiddelde Executietijd: Availability 99%

CLF	PR	ET (s)
0.1	0.90	55.4142
0.2	0.85	36.6934
0.3	1.00	21.0847
0.4	0.85	36.5866
0.5	0.75	47.1134
0.6	0.90	35.1194
0.7	0.70	48.7644
0.8	1.00	0.1023
0.9	0.60	33.5933
1.0	0.65	60.6860

Tabel B.7: Test resultaten test met 10 PMs

CLF	PR	ET (s)
0.1	1.000	0.3290
0.2	0.750	132.4528
0.3	0.750	128.8403
0.4	0.625	89.5649
0.5	0.650	75.0350
0.6	0.650	88.9111
0.7	0.750	73.8430
0.8	0.850	55.0889
0.9	0.750	59.0743
1.0	0.525	77.1787

Tabel B.8: Test resultaten test met 20 PMs

Duplicaten	CLF	PR	ET (s)
1	0.1	0.086	1734.7092
	0.2	0.071	2028.7449
	0.3	0.100	1320.8388
	0.4	0.057	2345.3089
	0.5	0.114	1385.8344
	0.6	0.043	1356.9570
	0.7	0.057	1462.6067
	0.8	0.057	1752.9975
	0.9	0.043	4602.1654
	1.0	0.100	1852.9295
2	0.1	0.3571	2157.6218
	0.2	0.257	4177.5659
	0.3	0.357	2209.6379
	0.4	0.271	5384.8681
	0.5	0.371	1426.1838
	0.6	0.257	1866.2784
	0.7	0.300	1721.4633
	0.8	0.200	3427.4805
	0.9	0.200	2624.2755
	1.0	0.271	1608.5121
3	0.1	0.586	2079.6150
	0.2	0.457	3127.0865
	0.3	0.543	1188.9467
	0.4	0.443	1969.8573
	0.5	0.557	4963.6411
	0.6	0.400	1616.0662
	0.7	0.443	2842.4344
	0.8	0.343	3436.4754
	0.9	0.300	4275.1313
	1.0	0.343	1187.8310

Tabel B.9: Test resultaten test met 50 PMs