Project Algoritmen en datastructuren III

Gedistribueerde Genetische algoritmen

Robbert Gurdeep Singh

21 november 2014

Samenvatting

In dit verslag bespreken we de implementatie van een genetisch algoritmen voor het optimaal plaatsen van punten in een veelhoek. We onderzoeken wat de optimale parameters zijn en vergelijken enkele algoritmen. We zien dat Tournament Selection een goede selectiemethode voor zowel mate-selection als remove-selection. We merken ook dat random crossover voor dit probleem goed werkt. Verder zien we bij de gedistribueerde implementatie dat we ons moeten behoeden voor een te grote communicatieoverhead. We geven ook de resulatateen van enkele test op de HPC cluster.

Deel I

Genetisch algoritme

1 Inleiding

Het probleem dat we trachten op te lossen is het volgende: plaats in een convexe veelhoek een gegeven aantal punten zodat de functie f zoals weergegeven in sectie 1.1 gemaximaliseerd wordt. We maken zowel een serieel als gedistribueerde implementatie van een genetisch algoritme. Hierbij zullen we trachten de parameters van het algoritme te optimaliseren voor het gegeven probleem. Als we het vanaf nu hebben over een veelhoek dan bedoelen we een convexe veelhoek.

1.1 Gebruikte symbolen

Tenzij anders vermeld betekenen volgende symbolen het volgende:

$\boldsymbol{n}$: Het aantal te plaatsen punten

$\boldsymbol{X_i}$: Het i-de individu

$\boldsymbol{z}$: Het aantal zijden van de veelhoek

$\boldsymbol{N_p}$: Populatiegrootte

$\boldsymbol{N_l}$: Aantal geliefden

$\boldsymbol{P_M}$: Selectie druk

$\boldsymbol{f}$: De fitheidsfunctie

$$f(X_k) = \sum_{P_1 \in X_k} \sum_{P_2 \in X_k} \sqrt{d_2(P_1, P_2)}$$

2 Algortitmen

2.1 Punt in veelhoek

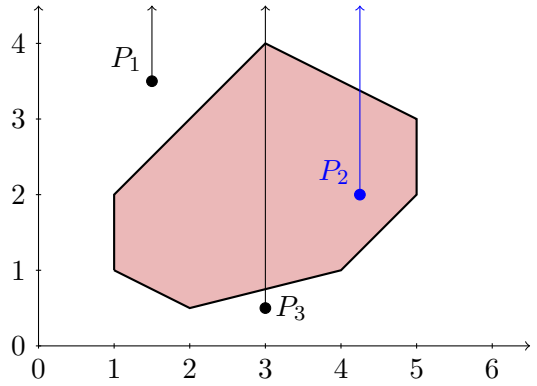
Eén van de voorwaarden waaraan een oplossing moet voldoen is dat alle punten **in** de veelhoek liggen. Het is dus noodzakelijk om een algoritme te hebben dat bepaalt of een punt zich al dan niet in de convexe veelhoek bevind.

2.1.1 Idee

Trekken we een lijn vanuit het punt "naar boven", dan kunnen we 3 gevallen onderscheiden:

- We snijden de veelhoek niet (P_1):
  We weten dat we ons niet binnen de veelhoek bevonden.
- We snijden de veelhoek juist 1 keer (P_2):
  Nu zijn we zeker in de veelhoek.
- We snijden de veelhoek juist[1] 2 keer (P_3):
  We liggen onder en dus ook buiten de veelhoek.

[1] Merk op dat indien we de veelhoek meer dan 2 keer snijden, we kunnen aantonen dat de veelhoek niet convex is.



Figuur 2.1: Punten binnen en buiten de figuur `soos.poly` met snijpunten.

We moeten dus nagaan dat een lijn "naar boven" vanuit het punt de veelhoek juist één keer snijd.

2.1.2 Algoritme

Om te tellen hoe vaak we de veelhoek snijden, gaan we als volgt te werk: Bij het inlezen van de veelhoek stellen we vergelijkingen op van de zijden van de veelhoek. Deze zijn van de vorm $y = a \cdot x + b$. Met $a = \infty$ als de rechte evenwijdig is met de y-as. Daarna gaan we te werk zoals algoritme 2.1 beschrijft.

In algoritme 2.1 zijn z.a en z.b de coëfficiënten zijn van de vergelijking $y = ax + b$ die de zijde voorstelt. De notatie van de intervallen houden geen rekening met de verhouding van de waarden ten opzichte van elkaar, wij bedoelen steeds de waarden tussen de gegeven waarden in of exclusief de grenzen.

2.1.3 Complexiteit

Kijken we naar algoritme 2.1 dan is het duidelijk dat de complexiteit $O(z)$ is met z het aantal zijden.

2.1.4 Implementatie

Deze methode is geïmplementeerd in `polygon.c` als `polygon_contains()`. We wijzen nog even op enkele details:

- We gebruiken $(P_x - x_1) \cdot (P_x - x_2) \leq 0$ om te bepalen of een waarde al dan niet tussen 2 waarden ligt. We doen dit zo omdat we niet weten hoe x_1 en x_2 zich onderling verhouden.

Algoritme 2.1 Bepalen of een punt in een veelhoek ligt

Require: zijden, een lijst van de zijden van een convexe veelhoek.
function INPOLYGON(P_x,P_y)
  count $\leftarrow 0$
  for each z $\in$ zijden **do**
    $(x_1, y_1) \leftarrow 1^{ste}$ gedefinieerde punt van z
    $(x_2, y_2) \leftarrow 2^{de}$ gedefinieerde punt van z
    if z.$a = \infty \wedge x_1 = P_x$ **then** ▷ z $\parallel$ y-as
      if $P_y \in [y_1, y_2[$ **then**
        return `true` ▷ Op lijn
      else if $y_1 > P_y$ **then**
        count $\leftarrow$ count+1
      end if
    else ▷ z $\not\parallel$ y-as
      $y_{inter} \leftarrow$ z.$a \cdot P_x$+z.b
      if $P_x \in [x_1, x_2[$ **then**
        if $y_{inter} = P_y$ **then**
          return `true` ▷ Op lijn
        else if $y_{inter} > P_y$ **then**
          count $\leftarrow$ count+1
        end if
      end if
    end if
  end for
  return count == 1 ? `true` : `false`
end function

- Om het herberekenen van a en b te vermijden, berekenen we deze eenmalig bij het inlezen van de veelhoek.

2.1.5 Genereren random punten

Om random individuen voor de populatie te creëren genereren we telkens een random x en y binnen het omsloten vierkant van de figuur. Vervolgens gaan we na of het ook wel degelijk in de veelhoek ligt. Telkens zo'n punt is gevonden wordt het toegevoegd aan het te genereren random individu tot er n punten zijn. Voor het genereren van N_p individuen heeft dit een complexiteit van $T(n) = \Theta(N_p \cdot n) = \Theta(n)$

2.2 Tournament Selection

Tijdens de uitvoering van het genetisch algoritme zullen er steeds meer individuen bijkomen. We willen er echter voor zorgen dat onze populatie een vaste grootte N_p behoud. Er zullen dus individuen vermoord[2] moeten worden.

2.2.1 Algoritme

Om te bepalen wie we vermoorden gebruiken we "tournament selection". Bij deze selectie word er telkens een vast aantal arbitraire individuen gekozen. Het individu met de slechtste fitheid uit deze groep wordt vervolgens vermoord. Dit wordt herhaald tot de populatie terug de juiste grootte heeft.

2.2.2 Complexiteit

In algoritme 2.2 wordt er $|P| - N_p$ keer een slachtoffer gekozen door een toernooi uit te voeren. Tijdens een toernooi wordt er N_t keer een willekeurig individu gekozen en vergeleken. De complexiteit is dus $\Theta((|P|-N_p)\cdot N_t)$. Met N_t de toernooigrootte[3] en N_p de gewenste populatiegrootte. We zien dat N_t hier gelijk is aan $\left\lfloor \frac{P_M}{100} \cdot |P| \right\rfloor$. We bekomen dus:

$$\Theta\left((|P| - N_p) \cdot P_m \cdot |P|\right)$$

Nu merken we op dat dit allemaal constanten[4] zijn binnen ons programma dus hebben we dat de complexiteit $T(n) = \Theta(1)$.

[2] Dat is dus verwijderden uit de populatie
[3] Zie sectie 4.5
[4] Enkel aanpasbaar bij compilatie

Algoritme 2.2 Tournament-Select

Require:
P, te grote lijst van individuën
N_p, gewenste populatiegrootte
P_M, de selectiedruk
Ensure: returnt de index van verliezer
function TOURNAMENTSELECTION(P,N_t)
loser ← arbitraire waarde uit $\{0, \ldots, |P|\}$
for z = 2 → N_t **do**
j ← arbitraire waarde uit $\{0, \ldots, |P|\}$
if $f(X_j) < f(X_i)$ **then**
loser ← j
end if
end for
return *loser*
end function

while $|P| > N_p$ **do**
$N_t \leftarrow \left\lfloor \frac{P_M}{100} \cdot |P| \right\rfloor$
slachtoffer ← TOURNAMENTSELECT(P,N_t)
$P \leftarrow P \backslash$slachtoffer
end while

2.2.3 Opmerking

Onder de aanvaarde aanname dat er minstens twee verschillende arbitraire indexen worden gekozen kunnen we stellen dat het beste individu steeds in de populatie blijft.

Bewijs. Stel dat het beste individu door tournament selection geselecteerd wordt om te vermoorden. Dan betekent dit dat hij een slechtere fitheid heeft dan de andere geselecteerden. Waaruit volgt dat hij niet het beste individu was. ↯ □

2.2.4 Implementatie

Het bestand `genetic_base.c` bevat de implementatie van Algoritme 2.2 in de functies

- `do_deathmatch()` en
- `do_neg_tournament()`.

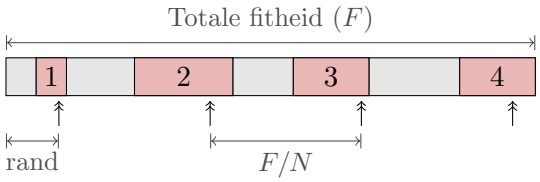
2.3 Selectie Paren

In het genetisch algoritme moeten er individuen worden geselecteerd die seks zullen hebben ter vorming van nieuwe individuen. Tijdens deze selectie moeten we er op letten dat dat we niet enkel de fitste individuen kiezen. De kans is namelijk reëel

dat de genetische informatie van een "slecht" individu aanleiding kan geven tot een zeer goed individu na paren. Natuurlijk willen we diegenen met een hogere fitheid wel een grotere kans geven zich voort te planten.

2.3.1 Idee

Stellen we onze individuen voor als blokjes met een grootte die evenredig is met hun fitheid, dan kunnen we ze na elkaar plaatsen en iets bekomen als in figuur 2.2. We kunnen dan ergens starten en met gelijke stappen vooruitgaan. Bij elke stap nemen we het stuk waar we bijstaan. Zo is de kans dat een stuk gekozen wordt recht evenredig met zijn fitness. Deze aanpak wordt Stochastic Universal Sampeling genoemd.



Figuur 2.2: Visualisatie Stochastic Universal Sampling

2.3.2 Algoritme

Het algoritme die het voorgaande idee implementeert is terug te vinden in algoritme 2.3.

2.3.3 Complexiteit

We moeten nagenoeg alle individuen in de populatie overopen. We kunnen dus stellen dat de complexiteit $\Theta(|P|)$ is, met $|P|$ de grootte van de populatie. Nu is $|P|$ geen argument van ons programma. Dus bekomen we een complexiteit van

$$T(n) = \Theta(1)$$

2.3.4 Alternatief

Een alternatief bestaat er in tournament selcetion te gebruiken zoals in Algoritme 2.2 uit sectie 2.2. Hierbij moeten we natuurlijk wel groter dan gebruiken in plaats van kleiner dan om meer kans te geven aan betere individuen. De complexiteit hiervan is

$$T(n) = \Theta(|P| \cdot P_M) = \Theta(1)$$

Algoritme 2.3 Stochastic Universal Sampeling

Require:
P, te grote lijst van individuën
N_l, gewenste aantal individuen
Ensure: returnt een lijst van gekozen individuen
function SUS(P,N_t)
Q $\leftarrow \emptyset$
total $\leftarrow \sum_{x \in P} f(x)$
size $\leftarrow \frac{\text{total}}{N_l}$
offset $\leftarrow$ arbitraire waarde uit $[0, \text{size}]$
$t \leftarrow$ offset
for each $p \in P$ **do**
$t = t - p$.fitness
if $t < 0$ **then**
$Q = Q \cup \{p\}$
$t = t$+offset
end if
end for
return i
end function

2.4 Mutation

We willen er voor zorgend dat de diversiteit in de samenleving niet verdwijnt. Daarom zullen we kleine afwijkingen introduceren. Mochten we dit niet doen, dan zouden we nooit andere punten kunnen krijgen dan deze die initieel gekozen zijn. En als gevolg zullen we het optimum nooit vinden.

2.4.1 Idee

Nadat er een kind gevormd is zullen we het met een bepaalde kans laten veranderen. Dat is dus dat er 1 punt lichtjes wordt verplaatst. Deze verplaatsing doen we door een nieuw punt te kiezen in de omgeving van het oorspronkelijk punt. Als dit gerandomiseerd punt buiten de figuur valt, proberen we het opnieuw met een kleinere omgeving.

2.4.2 Algoritme

Het algoritme die het voorgaande idee implementeert wordt beschreven in algoritme 2.4. Hierbij is de "diameter van poly" de maximale afstand tussen 2 punten van de convexe veelhoek.

2.4.3 Implementatie

De diagonaal die we in sectie 2.4.2 beschouwen, word bij de implementatie zeer ruw benaderd door de diagonaal van het omgestoten vierkant te bepalen.

Algoritme 2.4 Mutatie

Require:
I, een individu
poly de veelhoek
Ensure: I is mischien gemuteeerd
if `rand() % MUTATION_1_IN` $= 0$ **then**
$P \leftarrow$ arbitrair punt van I
$r \leftarrow$ diameter van poly / `MUTATION_DELTA`
repeat
kies $(x_{new}, y_{new}) \in \mathring{B}(P, r)$
$r \leftarrow r/2$
until $(x_{new}, y_{new}) \in poly$
end if

De implementatie is terug te vinden in `do_mutation()` van `genetic_base.c`

2.4.4 Complexiteit

Het kiezen van een nieuw punt is een operatie die in constante tijd kan gebeuren. Het nagaan dat het in de figuur ligt vraagt echter $\Theta(z)$ met z het aantal zijden in de veelhoek[5].

2.5 Crossover

2.5.1 Algoritme

Als 2 individuen geselecteerd zijn om te paren, dan moeten wij daaruit een kind creëren dat eigenschappen van beide ouders bevat. Dit doen we door het eerste deel van de ene ouder samen te stellen met het tweede deel van de andere ouder (en omgekeerd voor het genereren van een 2de kind). Deze aanpak staat beschreven in Algoritme 2.5. Alternatief kunnen we er ook voor kiezen om een willekeurig aantal keer te wissen (Algoritme 2.6).

2.5.2 Complexiteit

Een crossover kopieert n waarden. Het is dus

$$T(n) = \Theta(n)$$

2.5.3 Implementatie

Het bestand `genetic_base.c` bevat de implementatie van Algoritme 2.5 en 2.6 in de functie `do_crossover()`. Het bestand bevat beide implementaties, bij compilatie kan er tussen beide gekozen worden door we waarde van

[5]Zie Subsectie 2.1

Algoritme 2.5 1-point Crossover

Require:
mama, *papa* de ouders
n het aantal te plaatsen punten
Ensure: *kind* is een nieuw kind dat genetisch gelijkend is met de ouders
$r \leftarrow$ arbitrair getal in $\{1, \ldots, n-2\}$
for i **from** 0 **to** r **do**
kind.punten[i] $\leftarrow$ *mama*.punten[i]
end for
for i **from** $r+1$ **to** $n-1$ **do**
kind.punten[i] $\leftarrow$ *papa*.punten[i]
end for

Algoritme 2.6 random Crossover

Require:
mama, *papa* de ouders
n het aantal te plaatsen punten
Ensure: *kind* is een nieuw kind dat genetisch gelijkend is met de ouders
$r \leftarrow$ arbitrair getal in $\{1, \ldots, n-2\}$
for i **from** 0 **to** $n-1$ **do**
oud $\leftarrow$ arbitrare waarde uit $\{mama, papa\}$
kind.punten[i] $\leftarrow$ *oud*.punten[i]
end for

`RANDOM_CROSSOVER` in te stellen met de `-D` vlag van de compiler. Standaard staat deze waarde ingesteld op random, de reden hiervoor vind u in sectie 3.2.

2.6 Stopvoorwaarde

We moeten nu nog een voorwaarde opstellen om te stoppen met itereren. We willen pas stoppen als we er nagenoeg zeker van zijn dat er geen verbetering meer zal zijn.

2.6.1 Idee

We willen een waarde hebben die een maat is voor de grootte van recente verandering van de maximale fitheid in de populatie. Hiervoor zouden we een exponentieel lopend gemiddelde kunnen bijhouden van de verandering van de maximale fitheid per iteratie. Als we de maximale fitheid na de i-de iteratie $\bar{f}(i)$ noemen,dan kunnen we zo'n exponenteel lopend gemiddelde als volgt defineeren:

$$\Delta_i = \bar{f}(i) - \bar{f}(i-1)$$

$$g(i) = \Delta_i \cdot (1-\alpha) + g(i-1) \cdot \alpha$$

met α een weegfactor[6] die in $]0,1[$ ligt.

Als we dit in expliciete vorm herschrijven zien we dat het verleden steeds minder waarde heeft.

$$g(i) = (1-\alpha)\sum_{j=0}^{i} \Delta_j \cdot \alpha^j + g(0) \cdot \alpha^{i+1}$$

We kunnen nu verwachten dat het niet meer veel zal verbeteren eens $g(i)$ nagenoeg nul[7] is. Door $g(0)$ een hoge waarde te geven zorgen we er kunstmatig voor dat we een minimum aantal iteraties hebben.

2.6.2 Algoritme & Implementatie

Voor het beginnen van de iteraties houden we twee variabelen bij. Deze variabelen stellen de vorige maximale fitheid en de huidige waarde van $g(i)$ voor. Na elke iteratie herberekenen we die waarden. Eens $g(i)$ onder de treshhold valt stoppen we met itereren. Om het eindigen van het algoritme te verzekeren gebruiken we een bovengrens voor het aantal iteraties.

2.7 Het genetisch algoritme

Nu we alle onderdelen van het genetisch algoritme besproken hebben kunnen we het samenbrengen tot een geheel.

2.7.1 Algoritme

Merk op dat we in algoritme 2.7 tournament selection gebruikt voor zowel het selecteren van wie sterft als voor het selecteren wie seks heeft. We hebben de verklaring hiervoor geformuleerd in Sectie 3.1.

2.7.2 Complexiteisanalyse

We overlopen de stappen in Algoritme 2.7 om de complexiteit te bepalen:

1. **Random generatie**: Sectie 2.1.5 geeft ons:
   $T_{gen}(n) = \Theta(N_p \cdot n) = \Theta(n)$
2. **Iteraties**:
   (a) **Lover selection**:
   $T(n) = \cdot P_M \cdot N_p = \Theta(1)$

[6] In ons project is dit `WEIGHTING_DECREASE`; Zie Sectie 4.6 voor een bespreking van de optimale waarde.

[7] geimplementeerd als "minder dan 10^{11}"

Algoritme 2.7 Het genetisch algoritme

Require: n, het aantal te plaatsen punten
Ensure: een goede benadeering van een oplossing die aan de voorwaarden van een oplossing voldoet wordt eteeruggegeven
function DARWINSPOINTS
  $P \leftarrow N_p$ random individuen ▷ Sec 2.1.5
  while niet stopconditie **do** ▷ Sec 2.6
    $L \leftarrow$ SELECTLOVERS(N_l) ▷ Sec 2.3.4
    for $i \leftarrow 0 \dots \lfloor |L|/2 \rfloor$ **do**
      Maak 2 kinderen met ouders $L[2i]$ en $L[2i+1]$ en voeg ze toe aan P ▷ Sec 2.5.1
      Bereken fitheid kinden
    end for
    TOURNAMENT(P,N_p) ▷ Sec 2.2
  end while
  return $opl \in \{x \in P \mid f(x) = \max_{y \in P} f(y)\}$
end function

   (b) **Crossover**: Uit sectie 2.5.2
   $T_1(n) = \Theta(N_l \cdot n) = \Theta(n)$
   (c) **Fitheid kinderen berekenen**:
   $T_1(n) = \Theta(n^2)$
   (d) **Tournament Selection**:
   $T(n) = \Theta(N_p \cdot P_M) = \Theta(1)$
3. **Beste teruggeven**:
   Alles overlopen dus $T(n) = \Theta(N_p) = \Theta(1)$

In de 2de grafiek van figuur 3.1 op pagina 7 zien we dat het aantal iteraties $\Theta(n)$ is. We komen dus uit op een totale complexiteit van

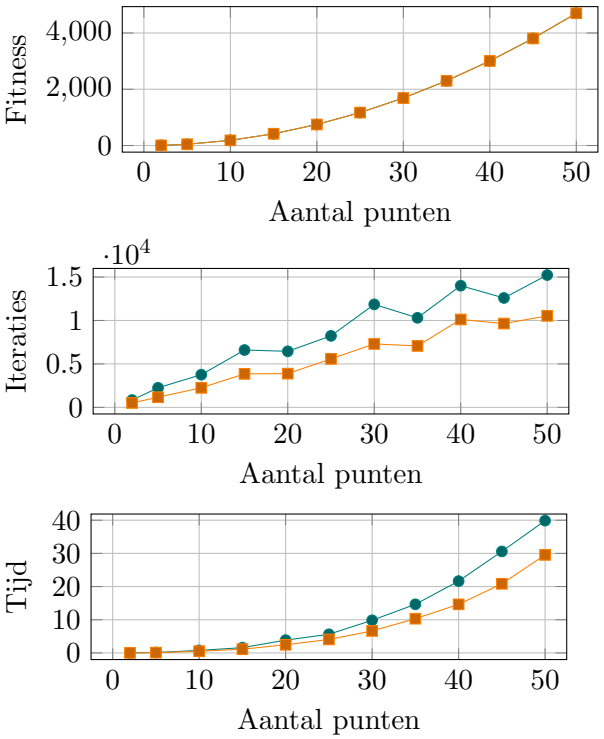
$$T(n) = \Theta\left(n^3\right)$$

Er dient opgemerkt te worden dat het aantal iteraties begrensd is in onze implementatie. Dus als n zo groot is dat het maximaal aantal iteraties wordt overschreden, dan in de complexiteit $\Theta(n^2)$.

3 Keuzes alogoritmen

Tijdens het optimaliseren van ons project hebben we hier en daar verschillende algoritmen uitgeprobeert om te zien welk resultaat ze opleveren. Hierna bespreken we welke dat zijn en welke we gekozen hebben.

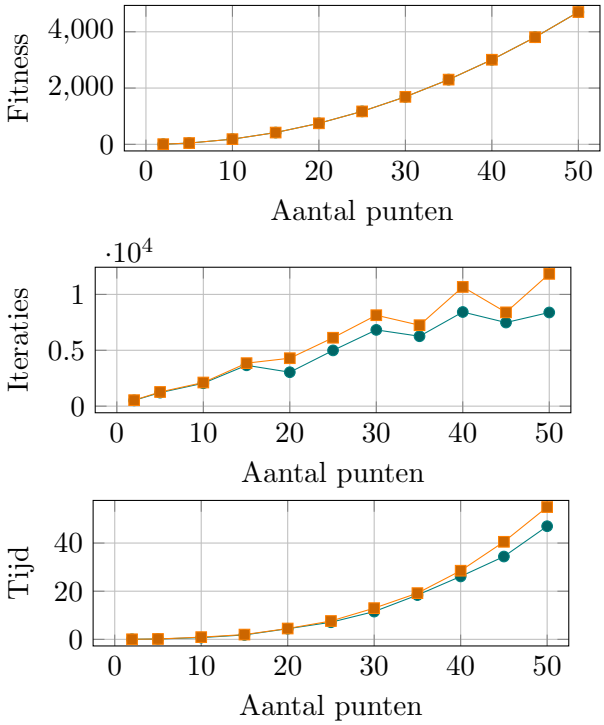
3.1 Selectie methode geliefden



Figuur 3.1: Vergelijking van Stochastic Universal Sampling (blauw) en Tournament Selection (oranje) voor selectie van individu's om voort te planten bij het plaatsen van een variabel aantal punten in `vierkant`.

Kijken we naar de grafieken in figuur 3.1 dan zien we dat Stochastic Universal Sampeling steeds een factor slechter trager is terwijl het resultaat even goed is. We merken ook dat het tijdsverschil ontstaat door het hoger aantal iteraties bij Sotochastic Universal Sampling. Met andere woorden, hier hebben we getoond dat Tournament de betere keuze is in tegenstelling tot wat enkele bronnen ons trachten te doen geloven. Dit kan uiteraard liggen aan het soort probleem.

3.2 Crossover



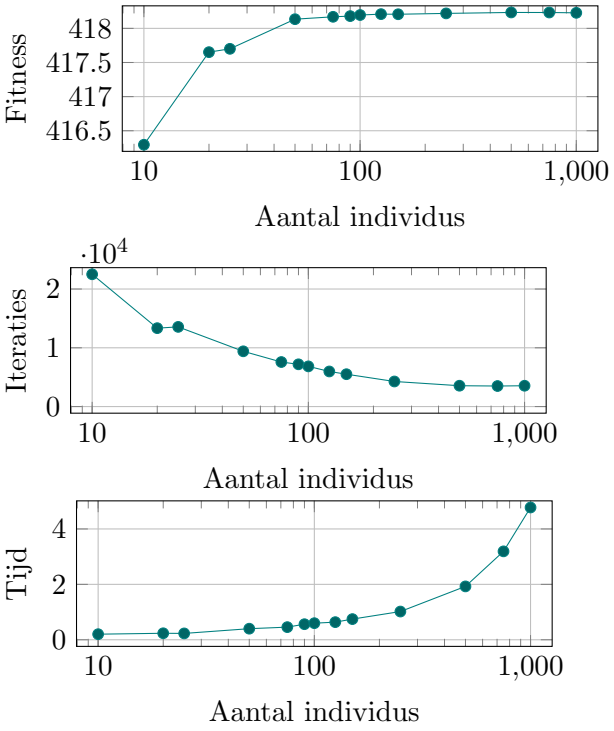
Figuur 3.2: Vergelijking van random crossover (blauw) en 1-point crossover (oranje) bij het plaatsen van een variabel aantal punten in `vierkant`.

De grafieken in figuur 3.2 geven een resultaat dat we niet verwachten. Random crossover presteert beter dan 1-Point crossover. We vermoeden dat dit komt doordat het random wisselen van punten er voor zorgt dat de punten aan de uiteinden minder gekopelt zijn aan elkaar waardoor er meer genetische diversiteit is wat bijdraagt aan een snellere convergentie.

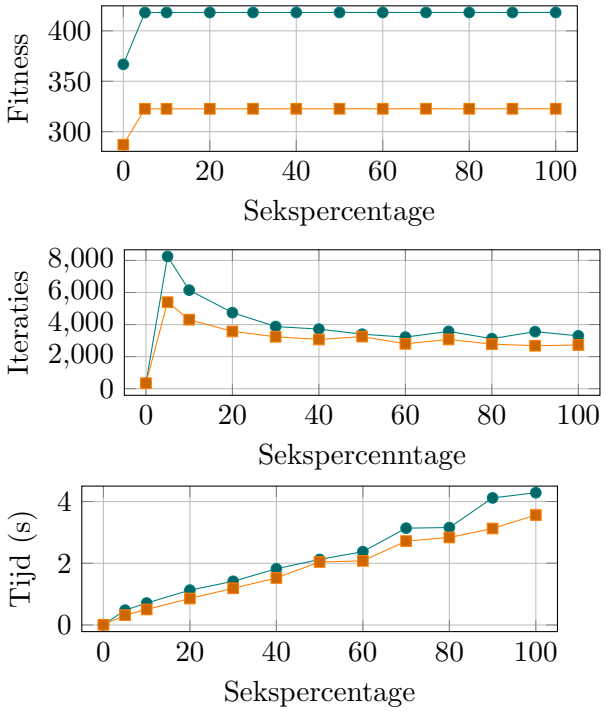
4 Parameter Optimalisatie

4.1 Aantal individuen

Een eerste parameter die we zullen onderzoeken is `NUM_INDIVIDUS`. Deze parameter stelt het aantal individuen in de populatie voor. In dit verslag wordt deze waarde ook N_p genoemd.



Figuur 4.1: Correlatie tussen het aantal individuen op convergentie snelheid en kwaliteit. Reslutaten voor het plaatsen van 15 punten in `vierkant.poly`.



Figuur 4.2: Correlatie tussen de hoeveelheid seks en de convergentie snelheid en kwaliteit. Reslutaten voor het plaatsen van 15 punten in `vierkant.poly` (blauw) en `soos.poly` (oranje).

Kijken ze naar de grafieken uit figuur 4.1, dan zien we dat het fitheid stijgt naarmate het het aantal individuen stijgt. Initieel is er een sterke stijging die afvlakt naarmate we honderd individuen bereiken. Helaas groeit de uitvoertijd ook lineair[8] met het aantal individuen. We kiezen voor een populatie van **100 individuen**.

4.2 Hoeveelheid seks

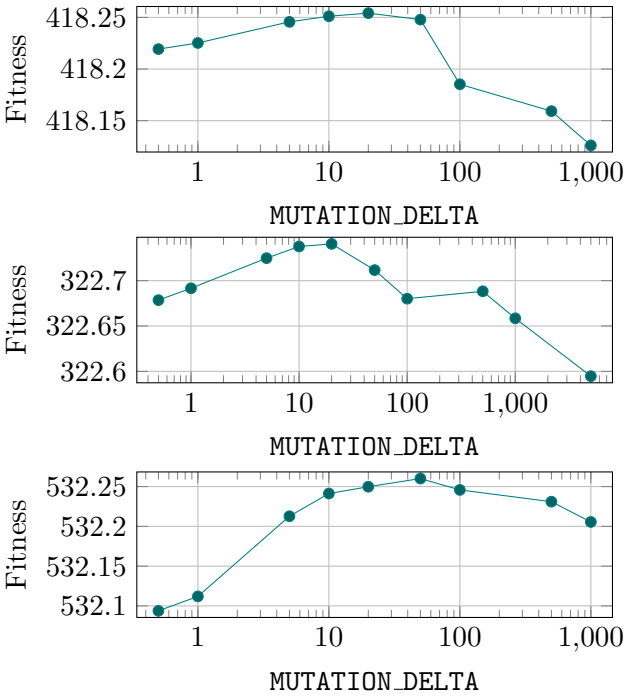
Nu onderzoeken we de invloed van `LOVER_PERCENT`. Deze parameter geeft aan welk percentage van de populatie zich voortplant per iteratie.

[8] Merk de logaritmische schaal van de x-as op.

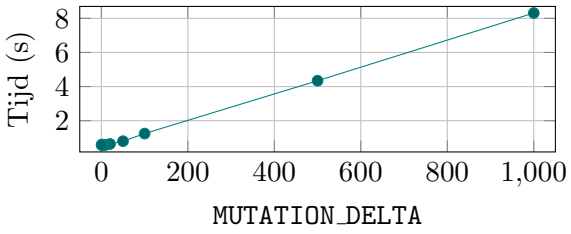
In de eerste grafiek van figuur 4.2 zien we dat de aanwezigheid van seks noodzakelijk is voor het behalen van een optimale fitheid. Daarnaast zien we dat het niet echt uit maakt hoeveel seks er is, als het er maar is. De stijging vlakt snel af, na 5% is de stijging nog maar miniem. De andere grafieken uit dezelfde figuur tonen ons dat hoe meer seks er is hoe langer het duurt om te convergeren. We kiezen ervoor om tijdens elke iteratie **30%** van de populatie seks te laten hebben. Merk op dat we niet 5% kiezen hoewel de grafiek dit suggereert. Als we de data grondiger bestuderen zien we dat de waarde van de fitheid monotoon blijft stijgen. We kiezen er voor om een wat tijd en ruimte op te offeren voor een iets beter resultaat.

4.3 Hoeveelheid mutatie

Als kinderen geboren worden krijgen ze door ons genetisch algoritme een mutatie. Het is duidelijk dat de grootte van deze mutatie aangepast moet zijn aan de grootte van de figuur. Daarom kiezen we telkens een punt in een cirkel die als diameter een fractie van de diameter van de figuur heeft. Deze fractie wordt voorgesteld door de parameter `MUTATION_DELTA`.



Figuur 4.3: Corelatie `MUTATION_DELTA` en de bekomen fitheid voor het plaatsen van 15 punten in `vierkant`, `soos` en `icosagon` (van boven naar onder)



Figuur 4.4: Corelatie `MUTATION_DELTA` en uitvoertijd voor 15 punten in `vierkant.poly`

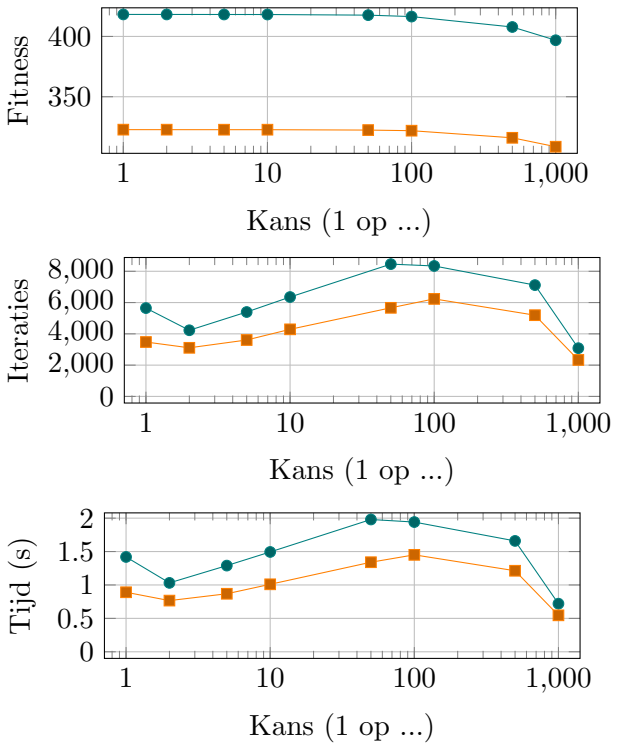
In de grafieken van figuur 4.3 zien we dat er telkens een waarde is voor `MUTATION_DELTA` die de fitheid optimaliseert. Helaas zien we ook dat deze sterk afhankelijk is van de figuur. Na veel testen kunnen we besluiten dat de waarde tussen 10 en 100 moet liggen.

We merken op dat als `MUTATION_DELTA` te groot is, dat betekend dus heel kleine stappen, de punten zich trager verplaatsen. Waardoor ze trager tot een optimum komen. Deze observatie zien we gereflecteerd in figuur 4.4.

We kiezen de waarde `MUTATION_DELTA = 20`. Dit wil zeggen dat het nieuwe punt zal gekozen worden in een cirkel met diameter die een twintigste is van de totale diameter.

4.4 Mutatie kans

Niet alle kinderen worden gemuteerd. De kans dat een kind muteert wordt voorgesteld door de parameter `MUTATION_1_IN`. De waarde van deze parameter geeft aan per hoeveel beschouwde kinderen er gemiddeld één muteert.



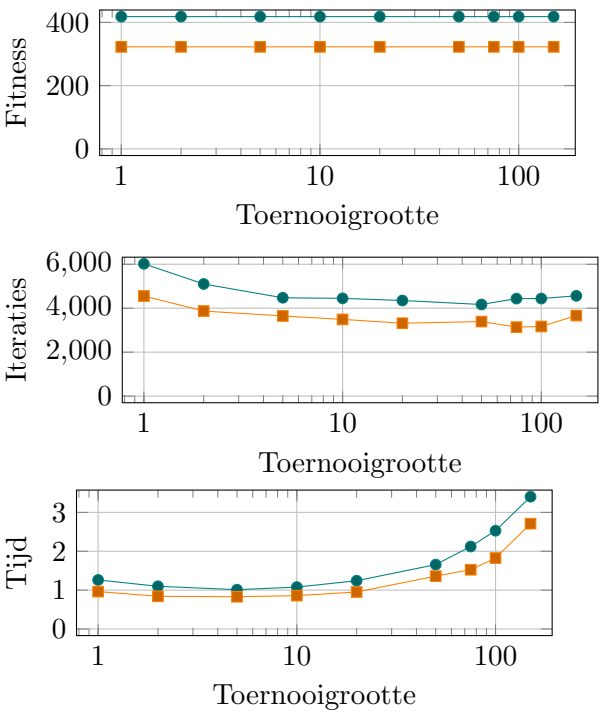
Figuur 4.5: Corelatie mutatiekans en de bekomen fitheid voor het plaatsen van 15 punten in `vierkant` (blauw) en `soos` (oranje)

De grafieken in Fig. 4.5 tonen ons dat hoe kleiner de kans om te muteren is, hoe slechter de fitheid. Om de fitheid te optimaliseren kiezen we dus best een grote kans. We kiezen: **1 mutatie per 3 kinderen**.

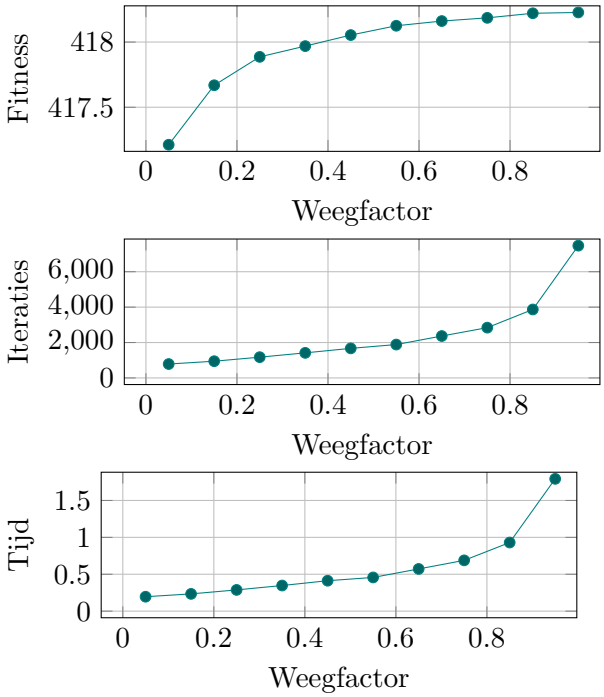
4.5 Selectie druk

De selectie druk is een maat voor de waarschijnlijkheid dat enkel de beste individuen zich kunnen voortplanten en lang kunnen leven. In ons programma is deze waarde voorgesteld door de parameter `SELECTION_PRESSURE`. Deze parameter wordt gebruikt om de toernooigrootte te bepalen bij tournament selection[9].

[9] Zie sectie 2.2



Figuur 4.6: Inpact van de selectiedruk op de convergentie snelheid en kwaliteit voor het plaatsen van 15 punten in `vierkant` en `soos`



Figuur 4.7: Correlatie weegfactor en convergentie snelheid en kwaliteit voor het plaatsen van 15 punten in `vierkant`

Kijken we naar de grafieken in figuur 4.6, dan zien we dat de bekomen fitheid nagenoeg constant blijft ongeacht de selectiedruk. Wat wel sterk afhangt van de selectie druk is de uitvoeringstijd en het aantal iteraties. We zien dat de uitvoertijd stijgt als de druk te hoog wordt. Hiervoor kunnen we tweee verklaringen geven.

- de uitvoertijd ligt hoger omdat de toernooien langer duren
- en de uitvoertijd ligt hoger omdat enkel de besten gekozen worden waardoor de genetische diversiteit vernietigd wordt en er dus enkel met mutaties tot een goede oplossing moet worden gekomen.

Wij hebben gekozen voor een selectiedruk van 5. Wat wil zeggen dat de toernooigrootte 5% is van de populatiegrootte.

4.6 Weegfactor exponentieel lopend gemidelde

Om te weten of het programma klaar is met optimaliseren, houden we een gewogen exponentieel gemiddelde bij[10]. De weegfactor die we kiezen is dus ook een parrameter die wel willen optimaliseren.

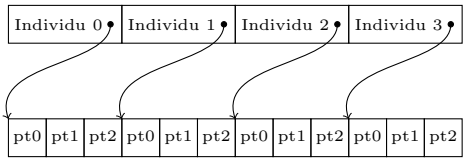
[10] zie ook sectie 2.6

We zien wat we verwachten te zien. Hoe hoger de weegfactor hoe beter het resultaat maar ook hoe langer het zal duren om tot dat resultaat te komen. We kiezen voor een weegfactor van **0.95** omdat we een optimaal antwoord willen. Merk op dat we niet kiezen voor 100 omdat het algoritme dat niet meer de waarde zou updaten.

5 Toelichting Code

In deze sectie gaan we wat dieper in op enkele programeerkeuzes die niet tot het algoritmische horen, maar die we wel graag vermelden.

5.1 Gebruik heap



Figuur 5.1: De individu array en de punten array op de heap

Om de individuen bij te houden hebben we er voor gekozen ze in 1 lange rij op de heap te plaatsen. Dit zorgt ervoor dat we de populatiegrootte at runtime kunnen bepalen, wat handig kan zijn bij

de parallelle uitvoering. We houden ook 1 lange array van punten bij op de heap. Deze array bevat $n \cdot N_p$ punten. Elk van de individuen heeft een pointer die naar een plaats in deze array wijst. Het in één keer alloceren zorgt ervoor dat we tijdens de uitvoering geen overhead hebben door allocaties. Het zorgt er ook voor dat we minder gemakkelijk memory leaks enzo hebben.

5.2 Plaats kideren

Na het paren zijn er meer individuen in de populatie. Om deze een plaats in het geheugen te geven maken tijdens de initialisatie de arrays waarin we het in Sectie 5.1 hadden extra "lege" plaatsen. De kinderen worden dan gestokeerd vanaf index N_p.

Door deze grotere array te nemen wordt het ook gemakelijker om tournament selection to te passen op heel de populatie.

5.3 Tests

Om te testen of bepaalde stukken code wel werkten hebben we testen geschreven. Deze zijn te vinden in `main.c` en kunnen worden geactiveerd met de `-D` vlag van de compiler.

5.4 Debuging

Tijdens de ontwikkeling wilden we graag zien hoe de fitheid verliep in de tijd. Hiervoor hebben we een `log_dbg` macro geintroduceerd die enkel print als er gecompileerd werdt met `-DDEBUG`.

Om de performantie gemakkelijk te testen met Python hebben we een *Performance Print* mode toegevoegd. Deze wordt geactiveerd door te compilen met `-DPERFORMANCE_PRINT`. In deze mode wordt enkel het aantal iteraties en de bekomen fitheid geprint.

Deel II
Gedistribueerde implementatie

Om ons algoritme te parallelliseren gaan we parallel meerdere keren naast elkaar het algoritme uitvoeren zoals beschreven in sectie 2.7. Als we echt gebruik willen maken van parallelle rekenkracht, dan moeten we die processen natuurlijk met elkaar laten communiceren. In de volgende secties bespreken we wat en hoe we zullen communiceren.

1 Idee

In de grafieken van figuur 4.2 zien we dat hoe meer individuen we hebben, hoe minder iteraties er nodig zijn maar ook hoe groter de uitvoeringstijd. Nu kunnen we een grotere populatie bestaande uit verschillende deelpopulaties nabootsen. Elke deelpopulatie geven we zijn eigen processor.

We merken op dat individuen gekozen door een positieve Touranament Selection[11] ideaal zijn om gedeeld te worden over processen heen. We zullen op zo'n manier individuen selecteren uit de populatie van het huidige proces en die verplaatsen naar de populaties van de andere processen.

2 Algoritme

Het algoritme dat wij gebruiken staat in Algoritme 2.1 beschreven. Voor het plaatsen van de individuen in de populatie hebben we 2 mogelijkheden. Deze worden besproken in section 4. Op welke manier de gegevens verdeeld worden is terug te vinden in sectie 3

Algoritme 2.1

Require:
N_p de populatiegrootte
N_i iteraties voor synchronisatie
N_t aantal te versuren en ontvanegn individuen
Ensure: Geeft een goede benadering voor een de beste oplossing terug
function ParralelDarwin
$P \leftarrow N_p$ random individuen
while not stopvoorwaarde **do**
Darwin(P,N_i)
$T \leftarrow$ PositiveTournament(N_t)
Verdeel T over de andere processen.
Plaats N_t ontvangen individuen in P
end while
return $opl \in \{x \in P \mid f(x) = \max_{y \in P} f(y)\}$
end function

De stopvoorwaarde die wij hebben gekozen is dat 10 keer na elkaar meer dan 30% van de processen hun stopvoorwaarde hebben bereikt.

3 Communicatie

We willen een vaste hoeveelheid individuen laten versturen door elk proces ongeacht het aantal lo-

[11] zie sectie 2.3.4 en 2.2

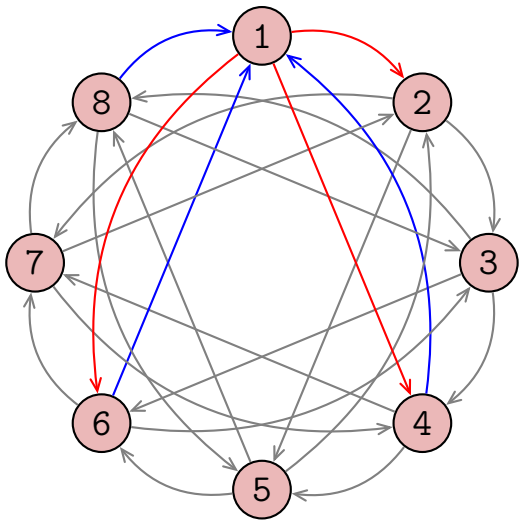
pende processen. Er kunnen zich 2 gevallen voordoen. Ofwel is de hoeveelheid individuen die we willen versturen per iteratie groter dan het aantal processen. Ofwel is dat niet zo.

3.1 Meer processen dan aantal te versturen

We willen er voor zorgen dat de individuen gelijkmatig worden doorgegeven en bij elk proces kunnen terechtkomen. We moeten er dus voor zorgen dat het nooit zo kan zijn dat de communicatie graaf meer dan 1 component bevat. Om te voorkomen dat er meerdere componenten zijn gaan we steeds naar het proces dat op ons volgt sturen[12]. Verder moeten we ervoor zorgen dat data van onze populatie zo goed mogelijk wordt verspreid over de andere processen. We kunnen dit doen door vanuit het i de te versuren naar de processen uit de verzameling

$$\left\{ \left\lfloor i + 1 + \frac{k \cdot p}{b} \right\rfloor \mod p \;\middle|\; k \in \{0, \ldots, b-1\} \right\}$$

met p het aantal processen en b het aantal te versturen individuen. Als we kijken naar figuur 3.1 zien we dat de de uitgaande en binnenkomende pijlen min of meer gelijkmatig verdeeld worden over de andere processen.



Figuur 3.1: De overdracht tussen 8 processen met een buffergrootte van 3. Uitgaande individuen van 1 zijn in het rood aangeduid. Binnenkomende individuen zijn in het blauw aangeduid.

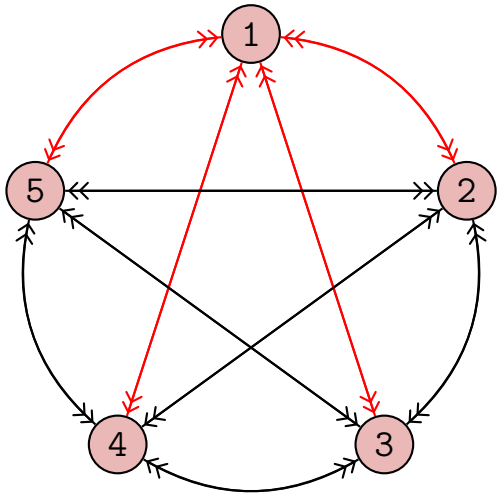
[12] formeel is dat dus process $i+1 \mod n$ voor i de huidige id en n het aantal processen

3.2 Minder processen dan aantal te versturen

Als er minder procesen zijn dan te verzenden individu's dan zullen alle processen van alle andere processen minstens 1 individu krijgen. We sturen

$$\#(\text{per proces}) = \left\lfloor \frac{b}{p-1} \right\rfloor$$

individuen door naar elk ander proces. Figuur 3.2 toont hoe de communicatie verloopt bij 5 processen en een buffergrootte van 8. We zien dat elk proces 2 individuen krijgt.



Figuur 3.2: De overdracht tussen 5 processen met 8 te versturen individuen. Uitgaande en binnenkomende individuen van 1 zijn in het rood aangeduid.

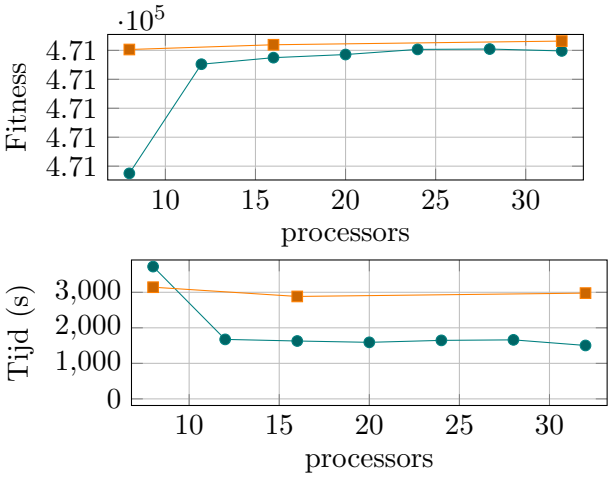
4 Kopiëren of verplaatsen

Elk process krigt nu evenveel individuen binnen als hij verstuurd. Wij kunnen nu 2 dingen doen:

1. De verstuurde individuen overschrijven met de nieuwe individuen.

2. De ontvangen individuen gewoon toevoegen aan de populatie en dan de populatie terug naar zijn oorspronkelijke grootte brengen zoals beschreven in sectie 2.2.

Figuur 4.1 toont ons dat kopiëren nefast is voor de performantie. We kunnen dit verklaren door op te merken dat bij het kopiëren de populatie als het ware veel duplicaten zal bevatten van verstuurde individuen. Dit zorgt er voor dat de ‚genetische diversiteit afneemt. Wat op zijn beurt de daling in prestatie verklaart.



Figuur 4.1: Uitvoeringstijd en fitheid in functie van het aantal processoren voor het plaatsen van 500 punten in `vierkant.poly` met (geel) en zonder (groen) kopiëren

5 Complexiteit

We bepalen nu de complexiteit van het algoritme rekening houdend met het aantal processoren p.

Als we kijken naar de grafieken uit 4.1 zien we dat zien we dat het aantal nodige iteraties exponentieel afneemt met stijgende populatiegrootte en dan stabiliseert. We merken op dat de sterke stijging in uitvoeringstijd er net toe doet daar we het parallel uitvoeren met kleine populaties. Hoe meer processoren, hoe minder tijd er nodig zal zijn om te convergeren tot op het moment van stabilisatie. We kunnen schrijven dat, tot er stabilisatie is, het aantal iteraties $\Theta\left(\frac{n}{\log p}\right)$ is.

Daar we telkens even veel individuen versturen om kunnen we er van uitgaan dat het aantal het aantal uitwisselingen lineair is in het aantal te verzenden punten[13]. We bekomen dus een complexiteit van $T(n) = \Theta\left(\frac{p \cdot n}{\log p}\right)$ voor de communicatie.

Zoals besproken in sectie 2.7 heeft het genetisch algoritme zelf een complexiteit van $T(n) = \Theta(n^2)$ per iteratie.

Zolang de winst door het vergroten van de populatiegrootte niet stagneert is de complexiteit

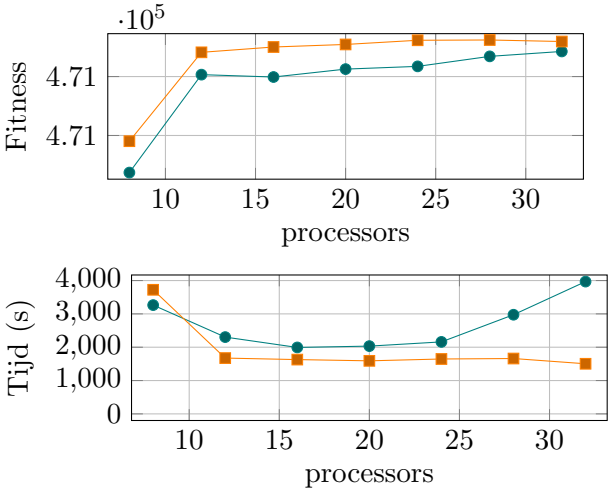
$$T(n) = \Theta\left(\frac{p \cdot n + n^2}{\log p}\right) = \Theta\left(\frac{p \cdot n + n^3}{\log p}\right)$$

. Nadat de winst door populatievergroting stagneert, wordt de noemer gewoon een constante. In figuur 6.1 kan je dit fenomeen zien in de groene grafiek. Deze grafiek is geproduceerd door te weinig communicatie te voeren tussen processen waardoor de winst door populatievergroting verkleint. Bij ongeveer 24 processoren wordt de overhead door communicatie ($\Theta(p \cdot n)$) groter dan de winst door populatievergroting.

[13] wat evenredig is met n, het aantal te plaatsen punten

6 Uitvoeren op HPC

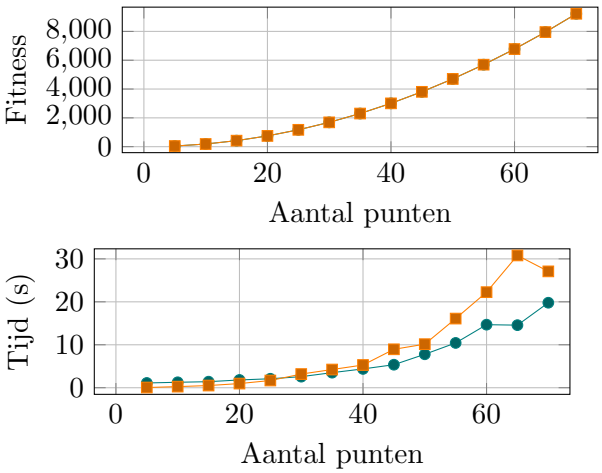
Om onze gedistribueerde code te testen maken we gebruik van de Pokémon[14] Delcatty. In figuur 6.1 zien we op de oranje grafiek bij het vergoten van het aantal processoren de fitheid stijgt en de uitvoeringstijd daalt. We merken echter op dat het dalen verbeteren initieel sterk is en daarna stagneert dit komt omdat er geen winst meer gehaald wordt uit het vergroten van de totale populatie. De groene grafiek geeft aan wat er gebeurt als er te men te veel tijd laat tussen synchronisaties (500 iteraties van het gewone algoritme in plaats van 50). Als we niet vaak genoeg synchroniseren emuleren we geen grotere populatie waardoor de winst door grotere populatie sterk wordt verminderd. We zien ook dat het eens de communicatie de overhand neemt de grafiek lineair stijgt zoals uitgelegd in sectie 5.



Figuur 6.1: Uitvoeringstijd en fitheid in functie van het aantal processoren voor het plaatsen van 500 punten in `vierkant.poly` bij goed gekozen communicatiehoeveelheid (oranje) en te weinig communicatie (groen)

[14] Dat is dus een supercomputer van het HPC

7 Vergelijking serieel



Figuur 7.1: Uitvoeringstijd en fitheid in functie van het aantal te plaatsen punten in `vierkant.poly` serieel (oranje) en parallel (groen)

Figuur 7.1 toont ons dat het gedistribueerde algoritme sneller is dan het serieel algoritme voor een groter aantal punten. Als er weinig punten te plaatsen zijn zorgt de communicatie voor te veel overhead. In de data die tot de grafieken leid zien we ook dat het parallel algoritme ook over het algemeen een fitheid geeft die net iets hoger ($< 0.1\%$) is dan bij serieel.

8 Implementatie

We bespreken nog hoe we de distributie hebben geïmplementeerd met MPI. Allereerst hebben we een functie gemaakt die het serieel algoritme voor een bepaald aantal iteraties uitvoert. Eens dit gedaan is, wordt een function pointer oproepen met pointer de huidge populatie en het aantal uitvoerde iteraties. Als deze oproep `GENETIC_CONTINUE` teruggeeft, dan wordt er verder gegaan met itereren anders wordt er gestopt. De functie waar de functionpointer naar wijst is verschillend voor het master proces en de slave processen.

Het master proces ontvangt telkens de fitheid van het beste individu en het percentage van het aantal gevraagde iteraties dat is uitvoert voor de stopvoorwaarde van het serieel algoritme werd bereikt[15] van de slaves met een `Gather`. Uit deze informatie leid het master proces af of er al dan niet gestopt moet worden. Als dat zo is stuurt hij een bericht naar alle slaves met `Send`. In dit bericht wordt het processID van het beste proces geplaatst. Het beste proces stuurt zijn beste individu terug naar de master die ze dan afbeeld op het scherm. Om zinloos wachten te vermijden doet het master proces, net als de slaves, ook iteraties van het genetisch algoritme.

De slave processen sturen de gegevens gevraagd door de master en stopen met itereren als de master dit vraagt. Er wordt gecontroleerd of de master een bericht heeft gestuurd met `IProbe`. Als het master proces een stop signaal geeft wordt `GENETIC_BEST` of `GENETIC_NOT_BEST` teruggegeven waardoor het itereren zal stoppen. Anders wordt er `GENETIC_CONTINUE` teruggegeven.

Om individuen door te sturen hebben we ze geserialiseerd tot een array van `floats`.

[15] Als deze niet berei werd, word 100% doorgestuurd

9 Bronnen

Er moet vermeld worden dat het `icosagon.poly` bestand afkomstig is van Jonathan Peck. We hebben dit bestand uitgewisseld om een andere figuur dan het gegeven vierkant te hebben samen met een notie van de maximale fitheid voor 50 punten in deze figuur. Code is er natuurlijk niet uitgewisseld.

Referenties

[1] Haupt, Randy L., and Sue Ellen Haupt. “Practical genetic algorithms.” (2004).

[2] Baker, James Edward. ”Adaptive selection methods for genetic algorithms.”Proceedings of an International Conference on Genetic Algorithms and their applications. 1985.

[3] Pit, Laurens Jan. ”Parallel genetic algorithms.”MS (Computer Sci.) Dissertation (1995).

[4] Brinkmann, Gunnar. ”Datastructuren en Algoritmen III, 2014.”Cursus (2014)

[5] University of Tennessee, ”MPI: A Message-Passing Interface Standard”Online PDF. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (2012)