Meta-heuriestieken

Wim De Keyser - januari 2022

1. Inleiding

In de praktijk wordt iedereen dagelijks geconfronteerd met problemen die de dag van vandaag altijd maar complexer worden. Besluitnemers, ingenieurs, analisten, techniekers,… moeten keuzes maken waarbij gestreefd wordt naar het beste resultaat rekening houdend met de onvermijdelijk steeds aanwezige beperkingen (op vlak van geld, tijd, beschikbare mensen, grondstoffen, enz..).

Heel veel keuzeproblemen of beslissingsproblemen kunnen op een wiskundige manier als een **optimalisatieprobleem** worden geformuleerd en aangepakt. De bedoeling is om een wiskundig model te maken van het beschouwd beslissingsprobleem, op zoek te gaan met behulp van een wiskundige methode naar een goede of zelfs de beste oplossing en om tot slot de gevonden wiskundige oplossing terug te vertalen naar de realiteit om daar deze oplossing toe te passen. Optimalisatieproblemen en de aanpak ervan, maken deel uit van het *operations research* domein.

2. Optimalisatieproblemen

2.1. Algemeen

Een optimalisatieprobleem bestaat uit mogelijke **oplossingen**, een **doelfunctie** –of **kostfunctie**- en een lijst van beperkingen, **constraints** genoemd.

Een optimalisatieprobleem omvat in de praktijk een aantal **variabelen.** Een mogelijke oplossing bekomt men door aan de variabelen een concrete waarde toe te kennen.

De constraints bepalen de **oplossingsruimte**, deze bevat alle mogelijke oplossingen waartussen de besluitnemer kan kiezen die voldoen aan alle constraints.

De doelfunctie levert voor elke mogelijke oplossing een waarde op die bepaald of een gekozen oplossing een goede, op minder goede oplossing is. Een doelfunctie dient ofwel geminimaliseerd ofwel gemaximaliseerd te worden. De meeste optimalisatieproblemen hebben één doelfunctie, maar er zijn ook optimalisatieproblemen die twee of meerdere doelfuncties hebben die tegelijkertijd geoptimaliseerd dient te worden. In deze tekst concentreren we ons op optimalisatieproblemen bestaande uit één doelfunctie.

2.2. Combinatorisch optimalisatieprobleem.

Optimalisatieproblemen kunnen in twee hoofdcategorieën opgedeeld worden, afhankelijk of alle beschouwd variabelen discrete of continue waarden kunnen aannemen. Wanneer alle variabelen van een optimalisatieprobleem enkel discrete waarden kunnen aannemen, spreekt men van een combinatorisch optimalisatieprobleem.

Combinatorische optimalisatieproblemen zijn niet altijd gemakkelijk op te lossen. Op het eerste zicht kan men een eenvoudige oplossingsmethode (algoritme) voorstellen: som alle mogelijke oplossingen op, bepaal voor elke oplossing de waarde van de doelfunctie en selecteer die oplossing

met de beste waarde voor de doelfunctie. Dit eenvoudig algoritme wordt de *enumeration method* genoemd. Zolang het aantal mogelijke oplossingen beperkt is, kan dit algoritme worden toegepast. In de praktijk is het aantal mogelijke oplossingen (eindig en aftelbaar, maar) veel te groot waardoor de berekentijd nodig om dit algoritme toe te passen niet haalbaar is, zeker niet met de computers van vandaag (en zelfs niet met de computers van morgen!). Er bestaan naast de enumeration method dan ook heel wat andere algoritmes.

3. Algoritme versus heuristiek

3.1. Wat is een algoritme en wat is een heuristiek?

Het oplossen van een optimalisatieprobleem komt erop neer dat de *beste*, ook wel *optimale*, *oplossing* gezocht wordt tussen alle mogelijke oplossingen. Een wiskundige methode die toegepast op een optimalisatieprobleem gegarandeerd de 'optimale' oplossing bekomt, noemt met een **algoritme**. Naast algoritmen bestaan er ook heuristieken. Een **heuristiek** –ook wel benaderingsalgoritme genoemd- is een wiskundige methode die toegepast op een optimalisatieprobleem een 'goede' oplossing bekomt maar niet noodzakelijk de 'optimale' oplossing.

Om te begrijpen waarom heuristieken bestaan (Een algoritme geeft gegarandeerd de optimale oplossing, een heuristiek niet. Waarom zou ik dan een heuristiek gebruiken in plaats van een algoritme?) moeten we eerst ons eerst even verdiepen in de **computationele complexiteitstheorie**, een tak van de theoretische informatica.

3.2 Computationele complexiteit

Het doel van de **computationele complexiteitstheorie** is om problemen die door een computer kunnen worden opgelost, ook wel **computationele problemen** genoemd, te klasseren in een aantal moeilijkheidscategorieën. Voorbeelden van computationele problemen zijn het sorteren van een lijst van getallen, het bepalen of een gegeven getal een priemgetal is, het berekenen van het n-de fibonacci-getal, enz.

De **computationele complexiteit** van een probleem is het minimum van de middelen (geheugen, tijd,…) ingezet door gelijk welke machine/computer om het probleem op te lossen. Dit wordt uitgedrukt in functie van de lengte van de input (of grootte) van het probleem. Maar het gekozen algoritme om het probleem op te lossen bepaalt natuurlijk mee de computationele complexiteit. Voor verschillende beginsituaties van het probleem zal een algoritme meer of minder middelen nodig hebben op een machine/computer om tot de oplossing te komen.

Voor de **computationele complexiteit van een algoritme** wordt meestal uit gegaan van de worst-case startsituatie. Voor de **computationele complexiteit van een probleem** wordt uiteindelijk gekozen voor de computationele complexiteit van het algoritme dat het best (laagste computationele complexiteit) het probleem op lost.

Voorbeelden hiervan zijn o.a. terug te vinden in de TI cursus *Programmeren I – C*. In deze cursus werden o.a. verschillende sorteeralgoritmen behandeld. Voor elk van de sorteeralgoritmen werd voor een te sorteren lijst van n elementen berekend hoeveel onderlinge vergelijkingen nodig zijn om -in het slechtste geval- de lijst gesorteerd te krijgen:

	# vergelijkingen (worst case)
Selection sort	(n*(n-1))/2
Bubble sort	(n*(n-1))/2
Merge sort	≈n log n
Quick sort	≈n^2

Het symbool ≈ stond voor 'in de grootte orde van'. In computationele complexiteit wordt dit aangeduid door O(). De computationele tijdscomplexiteit van het quick sort algoritme bedraagt $O(n^2)$, die van het merge sort algoritme O(n log n). De computationele tijdscomplexiteit van het sorteerprobleem bedraagt O(n log n).

Merk op dat de computationele tijdscomplexiteit de manier weergeeft waarop het algoritme zich gedraagt qua benodigde tijd als de grootte van het op te lossen probleem toeneemt.

Voorbeelden van computationele tijdscomplexiteit:

Tijd	**Notatie**	**Voorbeeld**
Constante tijd	O(1)	Het opvragen van een element uit een array
Lineaire tijd	O(*n*)	Het minimum in een ongeordende reeks getallen zoeken
Logaritmische tijd	O(log *n*)	Binary search
Lineair logaritmische tijd	O(*n* log *n*)	Merge sort
Kwadratische tijd	$O(n^2)$	Selection sort, Bubble sort, Quick sort
Exponentiële tijd	$O(2^n)$	Recursief berekenen van het n-de fibonacci getal

Een algoritme wordt in **polynomiale tijd** uitgevoerd, indien de computationele tijdscomplexiteit begrensd wordt door een polynoom (veelterm). Polynomiale tijd wordt genoteerd als $O(n^k)$ waarbij *n* de grootte van de invoer van het probleem weergeeft en *k* een constante is. n^k is de grootste term van de polynoom die de exacte tijdscomplexiteit weergeeft ($a_0 + a_1 n + a_2 n^2 + \ldots + a_k n^k$). Welke zijn de polynomiale tijden in bovenstaande tabel?

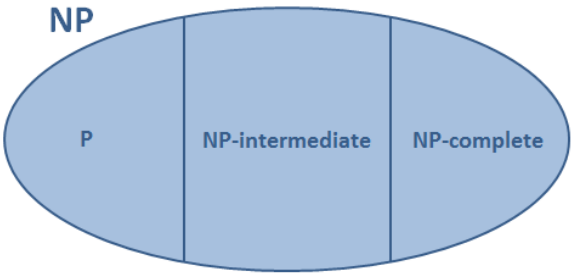
Hiervoor werd meerdere keren verwezen naar een machine/computer. In de computationele complexiteitstheorie maakt men gebruik van deterministische Turingmachines en niet-deterministische turingmachines. Een **Turingmachine** is een wiskundig model van een computer die gebruikmaakt van een strook papier om er symbolen op te schrijven en er vanaf te lezen. Vereenvoudigt gesteld, bevindt een Turingmachine zich steeds in één van de op voorhand vastgelegde toestanden. Afhankelijk van het symbool dat op de het strook papier staat, verandert de toestand, wordt eventueel een symbool weggeschreven en wordt de strook naar links of naar rechts opgeschoven. Dit 'gedrag' wordt omschreven met behulp van een transitiefunctie. Bij deterministische Turingmachines bevat de transitiefunctie voor elke toestand en elk symbool maar één mogelijkheid. Bij niet-deterministische Turingmachines bevat de transitiefunctie voor elke toestand en elk symbool meerdere mogelijkheden (er wordt op een random wijze één van de mogelijkheden gekozen). Turingmachines zijn niet bedoeld als echte computers, maar als gedachte-experiment waarop wiskundige analyses gemaakt kunnen worden.

In de computationele complexiteitstheorie beschouwd men meerdere complexiteitsklassen:

- De complexiteitsklasse **P**: hiertoe behoren problemen die in polynomiale tijd door een deterministische Turingmachine opgelost kunnen worden.
- De complexiteitsklasse **NP**: hiertoe behoren problemen die in polynomiale tijd door een niet-deterministische Turingmachine opgelost kunnen worden.
- De complexiteitsklasse **NP-complete**: dit is een subklasse van de klasse NP. Problemen die tot deze klasse behoren kunnen in polynomiale tijd op een deterministische Turingmachine omgezet worden naar een ander probleem in deze subklasse. Met ander woorden, indien voor één van de problemen behorende tot de klasse NP-complete een algoritme kan gevonden worden dat in polynomiale tijd op een deterministische Turingmachine kan worden opgelost, behoren in één klap alle problemen in de klasse NP-complete tot de klasse P.
- De complexiteitsklasse **NP-intermediate**: hiertoe behoren alle problemen die niet behoren tot P en niet behoren tot NP-complete.

Opmerkingen:

- Daar deterministische turingmachines beschouwd kunnen worden als speciale gevallen van een niet-deterministische Turingmachine, vormt P een deel verzameling van NP.
- Een polynomiale tijd op een niet-deterministische Turingmachine komt overeen met een exponentiële tijd op een deterministische Turingmachine.
- Voor een probleem behorende tot de complexiteitsklasse NP werd tot op heden nog geen algoritme gevonden dat het probleem in een polynomiale tijd op een deterministische Turingmachine oplost. Misschien bestaat zo'n algoritme niet, maar het kan ook zijn dat de mensheid dit algoritme nog niet heeft ontdekt/uitgevonden.
- Uit vorige opmerking volgt de vraag is P=NP of niet? Dit is een van de meest belangrijke openstaande vragen[1] in de theoretische informatica (als je even nadenkt over de gevolgen begrijp je direct waarom!)



3.3 Bestaansrecht heuristieken

Heel wat algoritmen voor optimalisatieproblemen hebben een exponentiële tijdscomplexiteit. Voor problemen van grote omvang zijn ze dus in de praktijk niet bruikbaar. Wie wenst er jaren of eeuwen te wachten op de optimale oplossing van het optimalisatieprobleem in zijn productieproces van vandaag?

[1] Dit is één van de zeven Millenium Prize Problems opgesteld in het jaar 2000 door Clay Mathematics Institute. Wie, als eerste, één van deze zeven problemen oplost krijgt een beloning van $ 1.000.000 (zie http://www.claymath.org/millennium-problems/millennium-prize-problems)

Heuristieken kunnen snel –en dan spreken we van seconden tot minuten- 'goede' oplossingen bekomen. Wie over een beperkte tijd beschikt kan dus genoegen nemen met een qua 'kwaliteit' mindere oplossing. Het is een goede oplossing maar is niet gegarandeerd de optimale oplossing. In de praktijk is in zeer veel gevallen goed goed genoeg. Voor bepaalde problemen kan men zelfs aantonen dat de oplossing gevonden met specifieke heuristieken zich dicht in de buurt van de optimale oplossing bevindt (soms een afwijking van slecht 5% of minder).

Om een idee te krijgen van wat exponentiële tijdscomplexiteit betekent voor problemen met een grote omvang, kunnen we eventjes naar het **handelsreizigersprobleem** kijken, een probleem dat behoort tot de complexiteitsklasse NP:

> *Een handelsreiziger dient n steden te bezoeken. Hij kent de afstanden tussen ieder paar van de te bezoeken steden. Hij wenst elke stad één keer te bezoeken en te eindigen in zijn vertrekstad (waar hij woont). Welke is de kortste weg die de handelsreiziger kan volgen?*

Er zijn in totaal n! aantal mogelijke routes, dit zijn het aantal mogelijke permutaties van de n steden. Dus de *enumeration method* heeft een tijdscomplexiteit van O(n!). Met behulp van *dynamic programming* werd een algoritme opgebouwd met een tijdscomplexiteit van $O(n^2\ 2^n)$. Een eenvoudige heuristiek voor het handelsreizigersprobleem is de *nearest neighbour* benadering. De handelsreiziger kiest steeds de stad waar hij nog niet is geweest dat het dichtst ligt bij de stad waar hij zich bevindt. De tijdcomplexiteit is polynomiaal: $O(n^2)$.

n	**n!**	**$n^2\ 2^n$**	**n^2**
5	120	800	25
10	3.628.800	102.400	100
15	1.307.674.368.000	7.372.800	225
20	2.432.902.008.176.640.000	419.430.400	400

Opmerking: het handelsreizigersprobleem komt ook om de hoek kijken bij het ontwerpen van printplaten, bij het productieproces waarbij meerdere gaten of uitsparingen in een plaat dienen te worden geboord,…

Heuristieken kunnen ook nuttig zijn, zelfs wanneer er voor het beschouwde probleem een algoritme bestaat dat de oplossing in een polynomiale tijd weergeeft. Een heuristiek met als tijdscomplexiteit $O(n^2)$ kan voor grote waarden van n veel interessanter zijn dan een algoritme met als tijdscomplexiteit $O(n^6)$.

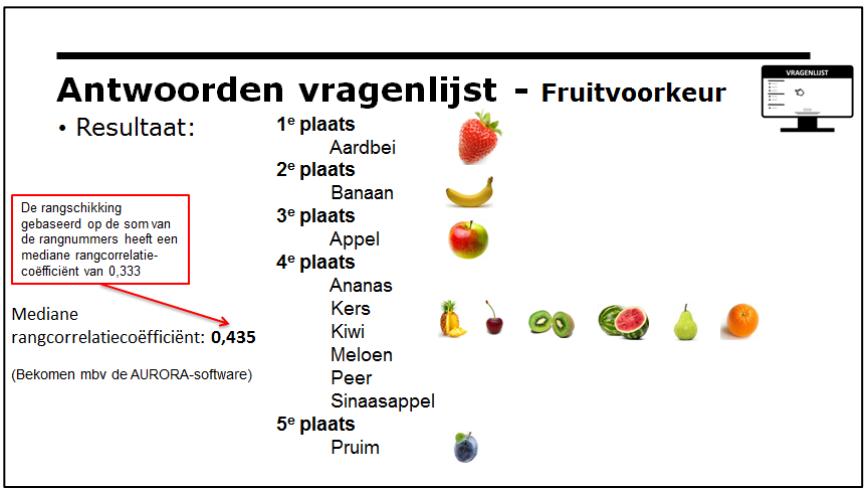
Sommige optimalisatieproblemen kunnen eenvoudig geformuleerd worden, maar zijn moeilijk om wiskundig op te lossen. Er wordt dan eerder naar heuristieken dan naar algoritmen gegrepen. Beschouw het volgende optimalisatieprobleem:

> *Een groep van m besluitnemers hebben elk n alternatieven gerangschikt. Er zijn dus m individuele rangschikkingen beschikbaar. De groepsrangschikking wordt gezocht die de mediane rangcorrelatiecoëfficiënt (Kendall) van de groepsrankschikking met de m individuele rangschikkingen maximaliseert.*

Dit is een combinatorisch optimalisatieprobleem waarvan de basisberekeningscomponenten namelijk de rangcorrelatiecoëfficiënt van Kendall en de mediaan, eenvoudige begrippen zijn. Het

samenbrengen van deze twee berekeningscomponenten zorgde echter voor een dusdanig moeilijke berekening dat de onderzoekers, die voor het eerst met dit optimalisatieprobleem werden geconfronteerd, zelfs niet op zoek gingen naar een algoritme, maar direct de piste van heuristieken bewandelden. Eén van de heuristieken die werden uitwerkt was op basis van simulated annealing (zie verder). Tijdens hun zoektocht naar betere heuristieken verkregen de onderzoekers gaande weg een beter inzicht in de berekeningsproblematiek en werd uiteindelijk – na 3 jaar- toch een ('branch and bound') algoritme gevonden. Het werd nog niet bewezen, maar er is een sterk vermoeden (daar het met een 'branch and bound'-algoritme kan worden opgelost) dat het probleem tot de complexiteitsklasse NP-complete behoort.

Bovenstaand optimalisatieprobleem kan je herkennen in de vraag om een door de groep gedragen groepsrangschikking op te stellen voor de voorkeuren voor 10 soorten fruit opgegeven door de studenten in de enquête ingevuld tijdens de lessen van het vak *Data Science I*.



4. Soorten heuristieken

4.1 'Custum made'-heuristieken

Vele heuristieken zijn 'custom made': ze worden ontwikkeld voor een specifiek optimalisatieprobleem en kunnen dus niet hergebruikt worden voor een ander optimalisatieprobleem. Dit komt doordat 'custom made'-heuristieken specifieke aspecten en eigenschappen van het optimalisatieprobleem gebruiken/exploiteren om snel tot een goede oplossing te komen. Deze specifieke aspecten en eigenschappen zijn echter niet noodzakelijk aanwezig in andere optimalisatieproblemen.

4.2 Eenvoudige heuristieken

Eenvoudige heuristieken berekenen aan de hand van een eenvoudig toe te passen criterium snel een goede oplossing. Denk hierbij als voorbeeld aan de *nearest neighbour* benadering van het handelsreizigersprobleem. Een oplossing bekomen met een eenvoudige heuristiek kan relatief ver afliggen van de 'optimale' oplossing maar moet gezien worden als een 'quick and dirty' manier om tot een oplossing te komen en is bijna altijd beter dan gewoonweg at random een oplossing te selecteren.

Men maakt het onderscheid tussen 'constructie'-heuristieken en 'twee-fasen'-heuristieken.

Bij een **'constructie'-heuristiek** wordt stapsgewijs een oplossing opgebouwd. Denk hierbij als voorbeeld opnieuw aan de *nearest neighbour* benadering van het handelsreizigersprobleem waarbij men start met één stad, de vertrekplaats van de handelsreiziger, en er stap voor stap een stad aan toevoegt om uiteindelijk te eindigen met een oplossing waarin alle steden zitten.

Zoals de naam weergeeft, wordt bij **een 'twee-fasen'-heuristiek** in twee fasen gewerkt. Ofwel construeert men in een eerste fase een initiële oplossing die men in een tweede fase verbetert, ofwel splits men in de eerste fase het probleem op in een aantal deelproblemen waarbij men voor elk deelprobleem een initiële deeloplossing opbouwt om daarna in de tweede fase de deeloplossingen te integreren in één oplossing. Voor het handelsreizigers probleem kan men de volgende 'twee-fasen'-heuristiek als voorbeeld beschouwen: in een eerste fase worden de te bezoeken steden geclusterd en wordt voor elke cluster een route opgebouwd tussen de steden behorende tot de cluster en in de tweede fase worden er routes gekozen om de routes in de clusters met elkaar te verbinden.

Opmerking: wanneer er meerdere eenvoudige heuristieken bestaan, kan men de **'best of'-benadering** toepassen. Hierbij worden de meerdere eenvoudige heuristieken toegepast op het probleem en de beste van de gevonden oplossingen wordt weerhouden.

4.3. 'Lokale zoek'-heuristieken

Er bestaat wel een globaal principe dat steeds als basis kan dienen voor een heuristiek en aldus op gelijk welk optimalisatieprobleem kan toegepast worden.

Dit principe is de **iteratieve verbetering**, ook wel **verbeteringsprincipe** of **'lokaal zoeken'** genoemd. Men start met het nemen van een startoplossing x_0 waarvan de doelfunctie wordt berekend: $f(x_0)$. Deze startoplossing kan ofwel willekeurig worden geselecteerd of kan het resultaat zijn van het toepassen van een eenvoudige heuristiek op het probleem. Vervolgens wordt er een kleine aanpassing gemaakt in de oplossing x_0 om te komen tot een aangepaste oplossing en wordt berekend of de doelfunctie in deze aangepaste oplossing beter is dan $f(x_0)$. Indien dit zo is wordt deze aangepaste oplossing het nieuwe vertrekpunt nl. x_1 met $f(x_1)$ voor een volgende iteratie. Indien de aanpassing geen verbetering is, dan blijft men bij de huidige oplossing en probeert men een andere mogelijke aanpassing. Wanneer meerdere (tot alle) aanpassingen geen andere oplossing oplevert met een betere waarde in de doelfunctie, dan stopt de heuristiek en wordt de huidige oplossing beschouwd als de uiteindelijke oplossing.

Om een aanpassing te maken in de huidige oplossing om zo eventueel een beter oplossing te vinden zijn er meerdere benaderingen mogelijk. Men moet er bovendien over waken dat de aanpassing een oplossing oplevert die nog steeds voldoen aan de gestelde beperkingen.

Het nadeel van 'lokale zoek'-heuristieken is dan men riskeert in een lokaal minimum terecht te komen. Door enkel (kleine) wijzigingen in de huidige oplossing te beschouwen, zoekt men in feite een nieuwe betere oplossing in de buurt van de huidige oplossing. Wanneer in de buurt geen betere oplossingen zijn, stopt de heuristiek in de veronderstelling dat er nergens betere oplossingen zijn. Misschien zijn die er wel, maar niet in de nabijheid van de huidige oplossing.

Om aan dit euvel tegemoet te komen wordt soms de iteratieve verbetering meerdere keren toegepast met telkens een andere beginoplossing. De beste van de gevonden oplossingen is dan de uiteindelijke oplossing.

4.4. Meta-heuristieken

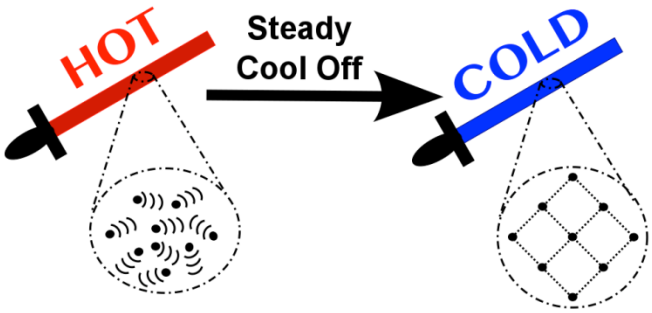
Een meta-heuristiek is een high level procedure dat op gelijk welk optimalisatieprobleem kan worden toegepast. Het bevat de volgende elementen:

- Er wordt een initiële oplossing gezocht (deze kan at random bepaald worden of met behulp van een eenvoudige heuristiek);
- Er wordt van de ene oplossing naar een andere oplossing in de buurt overgestapt (zoals in een 'lokaal zoek'-heuristiek) ;
- Nu en dan wordt toch een verslechtering van de waarde van de doelfunctie toegelaten wanneer men van een oplossing naar een andere oplossing in de buurt overstapt (een manier om aan een lokaal minimum te ontsnappen);
- Is gebaseerd op een analogie uit de fysica, de biologie of de ethologie -de leer die het gedrag van dieren in hun natuurlijke biotoop onderzoekt-;
- Er worden parameters gebruikt die de duur van de heuristiek en de kwaliteit van de gevonden oplossing beïnvloeden (het is meestal niet evident om de juiste waarden voor deze parameters te bepalen).

We omschrijven hieronder kort de principes van een aantal veel gebruikte meta-heuristieken.

4.4.1 Simulated annealing

Simulated annealing is gebaseerd op het gedrag van atomen bij verandering van temperatuur. Bij het produceren van metalen onderdelen (vb. tandwielen) wordt metaal verhit tot het vloeibaar is. Het vloeibare metaal wordt in een vorm gegoten waarna het wordt afgekoeld. Wanneer de afkoeling te snel gebeurt, ontstaan er onzuiverheden (amorfe structuren) en is het eindproduct van mindere kwaliteit (kan gemakkelijker breken). Wanneer de afkoeling geleidelijk gebeurt, ontstaan er sterkere kristalstructuren. Het afkoelen gebeurt aan de hand van een annealing-proces dat bepaald in welke stappen de temperatuur verlaagd wordt en hoelang men op een temperatuur blijft vooraleer de volgende temperatuurverlaging wordt doorgevoerd. Men zorgt dus eerst voor een thermodynamisch evenwicht vooraleer de temperatuur mag dalen.



Simulated annealing bootst dit proces na door toe te laten dat een oplossing overstapt naar een 'slechtere' oplossing in de buurt afhankelijk van een kansfunctie die o.a. bepaald wordt door de

temperatuur. Naarmate de gesimuleerde temperatuur daalt, verlaagt de kansfunctie en wordt het minder waarschijnlijk om over te stappen naar een nabij gelegen oplossing dat een 'slechtere' waarde heeft in de doelfunctie. Op elke gesimuleerde temperatuur worden een aantal overstappen van oplossingen toegelaten vooraleer de gesimuleerde temperatuur wordt verlaagd. De initiële gesimuleerde temperatuur is hoog genoeg om zo goed als elke overstap van een oplossing naar een andere oplossing in de buurt toe te laten.

Pseudo code:

```
InitializeParameters (Temperature t,  TemperatureReduction α)
initialSolution (Solution s)
s* = s   //best found solution
while t > TMIN
     temperatureIteration = 0
     While temperatureIteration < maxIterations
          s'=SelectNeighbour(s)
          Δ = objectiveFunction(s') – objectiveFunction(s)
                              // objectiveFunction must be minimized
          if (Δ < 0)
          then s = s'
               if objectiveFunction(s') < objectiveFunction(s*)
               then s* = s'
          else if atRandom > probability(exp(-Δ/T))
               then s = s'
     end while
     t = α*t
end while
return s*
```

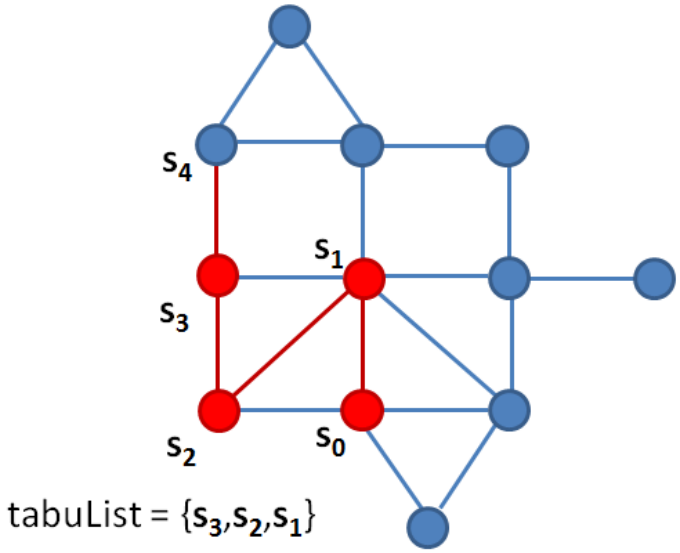
Op de website https://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/ krijg je een (versnelde) animatie van het toepassen van simulated annealing op het handelsreizigersprobleem (bezoeken van de 48 hoofdsteden van de staten van de USA).

4.4.2. Tabu search

Tabu search boots het gedrag na wanneer een persoon iets zoekt: de persoon kijkt niet twee keer (of meerdere keren) op dezelfde plaats. Dit doet een persoon door gebruik te maken van zijn geheugen (hij onthoudt waar hij al gekeken heeft).

Bij tabu search wordt voor een oplossing gekeken welke de beste oplossing is in zijn buurt. Er wordt overgestapt naar deze 'beste buur' ook indien deze 'beste buur' een slechtere waarde heeft in de doelfunctie. Er wordt een lijst bijgehouden van de vorige oplossingen. Deze lijst wordt de tabu lijst genoemd, daar het verboden is om bij het kiezen van de 'beste buur' terug te keren naar één van de vorige oplossingen die in de tabu lijst vermeld staan. Via een parameter wordt bepaald hoe lang deze tabu lijst is. De lijst kan beschouwd worden als het korte termijn geheugen daar niet alle vorige oplossingen worden opgeslagen, maar enkel de meest recente. Een stopcriterium bepaalt wanneer

de tabu search wordt beëindigd. De beste oplossing die tijdens het zoekproces werd tegengekomen vormt het resultaat van de tabu search.



Voor grotere optimalisatieproblemen vergt het bijhouden van oplossingen in een tabu lijst heel wat geheugen. Dan wordt eerder bijgehouden uit welke 'richting' men kwam en bevat de tabu lijst 'richtingen' die men niet uit mag gaan.

Het is ook mogelijk om een soort van lange termijn geheugen te implementeren binnen tabu search met de bedoeling om gebieden die veel belovend zijn van naderbij te onderzoeken en om gebieden die nog niet of te weinig onderzocht werden te bezoeken.

Pseudo code:

```
InitializeParameters (maxLenghtTabuList, stopCriterionParameter)
initialSolution(Solution s)
tabuList = {}
s* = s   //best found solution
repeat
     s'=SelectNeighbour(s)
     if not inList(s', tabuList)
     then if length(tabuList) = maxLenghtTabuList
          then removeOldestFromList(tabuList)
          addToList(s', TabuList)
          s = s'
          if objectiveFunction(s') < objectiveFunction(s*)
                                  // objectiveFunction must be minimized
          then s* = s'
     Update(stopCriterionParameter)
until stopCriterion(stopCriterionParameter)
return s*
```

4.4.3 Genetische algoritmen

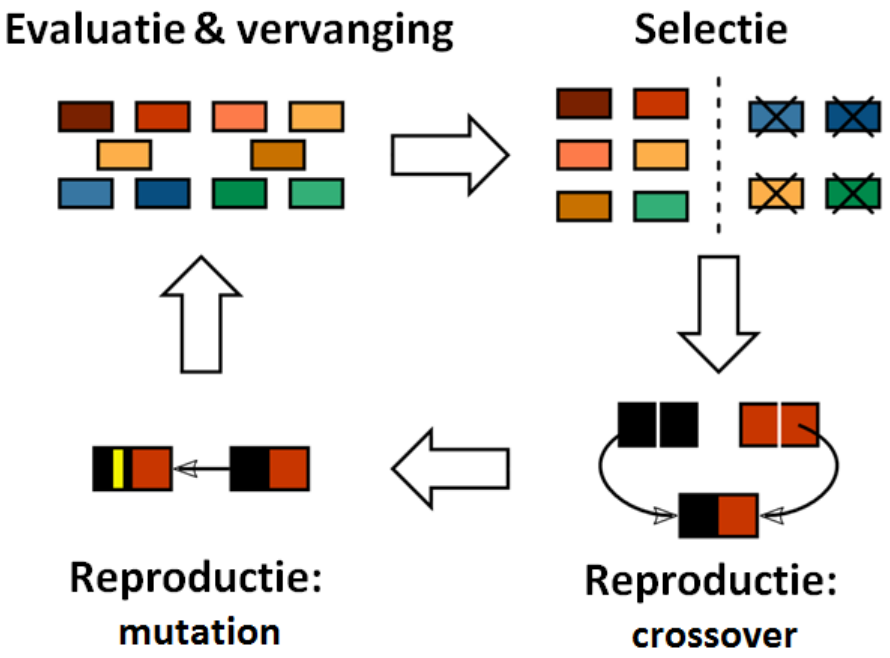
Genetische algoritmen zijn geïnspireerd op de evolutie van soorten (biologie).

Er wordt gestart met het at random selecteren van een groep van oplossingen. Dit is de initiële populatie. Voor elke individuele oplossing uit de initiële populatie wordt de waarde van de doelfunctie berekend en deze bepaald de mate waarin een individuele oplossing aangepast is aan zijn omgeving (indien de doelfunctie dient te worden geminimaliseerd, dan is hoe lager zijn waarde in de doelfunctie, hoe sterker de individuele oplossing). Er worden opeenvolgende generaties bepaald waarbij de populatie grootte constant blijft. Van generatie op generatie wenst men te evolueren naar sterkere individuele oplossingen. Om dit te bereiken, wordt er aan 'natuurlijke selectie' –survival of the fittest- en aan 'reproductie' – mengen, hersamenstellen en variëren van overerfbare eigenschappen van de ouders om afstammelingen te bekomen-.

De overstap van een populatie naar de volgende generatie verloopt in 4 fasen:

- Selectie-fase: bepalen welke individuele oplossingen gebruikt zullen worden in de reproductie-fase. Sterke oplossingen hebben een grote kans om geselecteerd te worden.
- Reproductie-fase: er worden nieuwe oplossingen bepaald aan de hand van *crossover* –of *recombination*- (op basis van twee van de geselecteerde oplossing uit de vorige fase) en aan de hand van *mutation* (op basis van één van de geselecteerde oplossing uit de vorige fase).
- Evaluatie-fase: de nieuwe individuele oplossingen worden geëvalueerd –wat is hun sterke-
- Vervangingsfase: daar de populatie qua grootte constant moet blijven en er door de reproductie nieuwe individuele oplossingen zijn bijgekomen, dienen een aantal individuele oplossingen uit de populatie te worden verwijderd. Hierbij worden de zwakste individuele oplossing in de populatie vervangen door de sterkste nieuwe individuele oplossingen bekomen in de reproductie-fase.

De heuristiek stopt na een aantal generaties in combinatie met een stopcriterium.



Pseudo code:

```
InitializeParameters (maxLenghtPopulation, maxReproductions,
                      maxMutations, stopCriterionParameter)
initialPopulation(PopulationListOfSolutions populationList,
                  maxLenghtPopulation)
s* = bestSolutionInPopulation(populationList)  //best found solution
repeat
    offSpringList = {}
    for i = 1 to maxReproductions
        SelectTwoParents(populationList, s1, s2)
        offSpringi = createOffSpring(s1, s2)    //using crossover
        addToOffSpringList(offSpringi)
    end for
    for j = 1 to maxMutations
        offSpringj = selectAnOffSpring(offSpringList)
        offSpringj = mutate(offSpringj)
        replaceInOffSpringList(offSpringj)
    end for
    for j = 1 to lenght(offSpringList)
        offSpringj = selectAndRemovefirst(offSpringList)
        weakest = weakestIndividualInPopulation(populationList)
        if objectiveFunction(offSpringj) <
                                        objectiveFunction(weakest)
                            // objectiveFunction must be minimized
        then replaceInPopulation(populationList, weakest,
                                    offSpringj)
    end for
    s' = bestSolutionInPopulation(populationList)
    if objectiveFunction(s') < objectiveFunction(s*)
    then s* = s'
    Update(stopCriterionParameter)
until stopCriterion(stopCriterionParameter)
return s*
```

4.4.4 Ander meta-heuristieken

Hierbij een niet exhaustieve lijst van andere meta-heuristieken:

- ant colony optimisation algorithm
- artificial immune systems
- bacterial foraging optimization algorithm
- bat algorithm
- biogeography-based optimization
- coevolutionary algorithms
- cultural algorithms
- differential evolution algorithm
- greedy randomized adaptive search procedure
- particle swarm optimization
- scatter search

…

4.4.5. Finale beschouwingen

Meta-heuristieken zijn handig wanneer men geconfronteerd wordt met een optimalisatieprobleem waarvan

- men op voorhand geen idee heeft van hoe de optimale oplossing er zou kunnen uitzien,
- men niet direct een idee heeft van hoe men de optimale oplossing zou kunnen vinden (geen gekende of bruikbare algoritmen),
- men niet direct informatie heeft over mogelijke 'custom made'-heuristieken,
- het aantal mogelijke oplossingen zeer groot is en enumeratie –gebruik van 'brute force'- daardoor is uitgesloten
- men wel van een kandidaat-oplossing kan nagaan of hij voldoet aan de constraints en of dit een goede oplossing zou kunnen zijn ("*you know a good one when you see it*")

De meeste meta-heuristieken bestaan al zeer lang, maar krijgen pas het laatste decennia meer aandacht. Dit komt doordat meta-heuristieken veel rekenkracht kunnen gebruiken. Rekenkracht dat door de snellere processoren maar vooral door de mogelijkheid van parallellisme van processoren de laatste jaren beschikbaarder is geworden.

5. Meta-heuristieken in Python

Een van de grote voordelen van een Meta-heuristiek is de generieke benadering (de "meta" in de naam Meta-heuristiek) die erachter zit. Codegewijs is er de mogelijkheid om probleemspecifieke elementen te scheiden van de meta-heuristische elementen. Met andere woorden, de programmacode voert de Meta-heuristiek uit en vereist alleen van de gebruiker de implementaties van een paar methoden (met vooraf bepaalde naamgeving, parameters,..) waar alle aspecten van het specifieke probleem dat men met de Meta-heuristiek wil oplossen, worden afgehandeld.

In Python kan je verschillende implementaties van Meta-heuristieken vinden. Helaas zijn ze niet altijd van een acceptabele en/of betrouwbare kwaliteit:

- sommige interessante implementaties zijn geschreven in oudere versies van Python, maar niet onderhouden en bevatten nu syntaxisfouten
- sommige implementaties scheiden de probleemspecifieke aspecten niet van de algemene procedure van de Metaheuristiek
- sommige implementaties kunnen slechts één specifiek probleem met de Metaheuristiek aan/ oplossen.

In deze tekst wordt gebruik gemaakt van een specifieke implementatie voor Simulated Annealing en een specifieke implementatie voor Genetic Algorithms. Voor Tabu Search werd geen bruikbare implementatie gevonden. Maar een programmeur laat zich hierdoor niet: ga de uitdaging aan en implementeer Tabu search zelf in Python!

5.1. Simulated annealing

Een van de Python libraries die Simulated Annealing ondersteunen, is het pakket *simanneal*.
Zie **GitHub - perrygeo/simanneal: Python module for Simulated Annealing optimization** voor meer informatie met betrekking tot deze library.
Vergeet niet om voor het eerste gebruik van de functies in de library de library te installeren:

```
>>> pip install simanneal      # only the first time
```

et doel van de *simanneal* library is om probleem-specifieke berekeningen te scheiden van meta-heuristiek-specifieke berekeningen:

- ⇨ Probleem-specifieke componenten:
  - ↳ de functie **move**: hoe van een oplossing naar een buuroplossing te gaan
  - ↳ de functie **energy**: berekent de waarde van de objectieve functie van een oplossing
- ⇨ Meta-heuristiek-specifieke componenten:
  - ↳ stel de annealing parameters in, zo niet worden standaardwaarden gebruikt
  - ↳ voer simulated annealing uit.

Om Simulated Annealing met *simanneal* uit te voeren, moet een subklasse van de klasse `Annealer` worden gemaakt waarin beide functies **move** en **energy** moet worden geïmplementeerd (ze worden in de klasse `Annealer` namelijk als abstracte functies gedeclareerd).

Laten we het gebruik van *simanneal* voor Simulated Annealing illustreren aan de hand van een voorbeeld. Een klassieke case om optimalisatie algoritmen en heuristieken te testen is de **Rastrigin functie** (https://en.wikipedia.org/wiki/Rastrigin_function). De hierbij gebruikte objectieve functie creëert heel wat lokale optima en omspant een grote 'oplossingsruimte' (= verzameling van alle mogelijke oplossingen). In dit optimalisatieprobleem dient een objectieve functie te worden geminimaliseerd voor continue variabelen (en hier kan men het aantal variabelen zelf kiezen) die tussen eenzelfde onder- en bovengrens liggen. Stel dat we 30 continue variabelen (x_i met i = 1,..,30) beschouwen die een waarde kunnen aannemen in het interval [-5.12, 5.12]. De optimale oplossing is hier gekend: alle variabelen hebben de waarde 0. De helpt om de kwaliteit van de gevonden oplossing door een heuristiek te kunnen beoordelen.

De objectieve functie ziet er als volgt uit:

$$f(x) = 10.n + \sum_{i=1}^{n} (x^2 - 10.\cos(2.\pi.x_i))$$

In Python implementeren we de objectieve functie aan de hand van de functie **energy**:

```
>>> from simanneal import Annealer
>>> class RastriginProblem(Annealer):
    """Test Annealer from package simanneal with the Rastrigin problem."""
        def move(self):
            # not yet implemented

        def energy(self):
            sum = 10 * len(self.state)
                for i in range(0,len(self.state)):
                    sum = sum+(self.state[i]**2-10*
                               math.cos(2*math.pi*self.state[i]))
            return sum
```

We kunnen een initiële startoplossing, bestaande uit 30 random gegenereerde reële getallen tussen de gestelde onder- en bovengrens, als volgt in Python bekomen:

```
>>> init_sol =  np.random.uniform(-5.12,5.12, size=30)
```

De overgang van een oplossing naar een buuroplossing kan worden gedaan door willekeurig een van de 30 dimensies te selecteren en voor die dimensie een nieuwe waarde te genereren (tussen de grenzen van de oplossingsruimte). Deze code moet worden geplaatst in de **move** functie:

```
def move(self):
            i = np.random.randint(0,len(self.state))
            self.state[i]=np.random.uniform(-5.12,5.12)
```

Nu kunnen we een object aanmaken van de subklasse `RastriginProblem` :

```
>>> rastrigin=RastriginProblem(init_sol)
```

Om een resultaat/oplossing te bekomen moeten we het simulated annealing proces laten uitvoeren:

```
>>> rastrigin.anneal()
```

We hebben hier geen parameters gespecifieerd, dus worden de default waarden voor deze parameters bij de uitvoering van het simuleted annealing gebruikt:

- Tmax = 25000.0 # start temperatuur
- Tmin = 2.5 # stop temperatuur
- temperature steps = 50000 # aantal iteraties
- updates = 100 # aantal updates

We kunnen echter deze parameters een waarde geven vooraleer we het annealing proces starten. Deze parameters zijn namelijk attributen van het rasterin-object waardoor we als volgt tewerk kunnen gaan (bijvoorbeeld de parameter Tmin):

```
>>> rastrigin.Tmin = 25.0
```

Opmerkingen:

- De energy-functie wordt per default gemaximaliseerd, dus vermenigvuldig het resultaat van de objectieve functie met -1 als de objectieve functie moet worden geminimaliseerd
- Andere beperkingen dan grenzen aan de waarden van de oplossing moeten worden opgenomen in de energy-functie (= objectieve functie), niet in de move-functie.

Maak wat aanpassingen in de parameters en kijk wat het effect is op het verloop van het simulated annealing proces.

Uitdaging: wat moet er veranderen om een oplossing te vinden van bovenstaand optimalisatieprobleem maar met de bijkomende constraint dat de oplossing voor geen enkele variabele in het interval [-1.0, 1.0] mag liggen?

5.2. Genetische algoritmen

Om genetische algorithmen in Python uit te voeren kunnen we gebruik maken van *ec* (Evolutionary Computation) objecten en functies uit het *inspyred* package.

De volgende stappen zijn nodig om goede oplossingen te vinden:

1. Creëer de initiële populatie met behulp van de opgegeven kandidaat seeds en de **GENERATOR**
2. Evalueer de initiële populatie met behulp van de **EVALUATOR**
3. Stel het aantal evaluaties in op de grootte van de initiële populatie
4. Stel het aantal generaties in op 0
5. Roep de **OBSERVER** op met de eerste populatie
6. Zolang de **TERMINATOR** niet waar is, voer de volgende lus uit:
    - Kies 'ouders' via de **SELECTOR**
    - Bepaal de nakomelingen van de ouders met behulp van de **VARIATOR** (o.a. toepassen van crossover & mutation)
    - Evalueer nakomelingen met behulp van de **EVALUATOR**
    - Update het aantal evaluaties
    - Vervang individuen in de huidige populatie met behulp van de **REPLACER**
    - Migreer individuen in de huidige populatie met behulp van de **MIGRATOR**
    - Archiveer individuen in de huidige populatie met behulp van de **ARCHIVER**
    - Verhoog het aantal generaties
    - roep de **OBSERVER** op met de huidige bevolking

Het doel van de *inspyre* library is de probleem-specififieke berekeningen te scheiden van de meta-heuristiek-specifieke berekeningen:

- ⇨ Probleem-specifieke componenten:
    - Een **GENERATOR** die bepaalt hoe oplossingen worden gecreerd
    - Een **EVALUATOR** die bepaalt de fitness (objectieve functie) waarden van de oplossingen
- ⇨ Meta-heuristiek-specifieke componenten:
    - Een **OBSERVER** die bepaalt hoe de gebruiker de toestand van de evolutie kan monitoren
    - Een **TERMINATOR** die bepaalt wanner de evolutie moet stoppen
    - Een **SELECTOR** die bepaalt welke individuel 'ouders' worden
    - Een **VARIATOR** die bepaalt hoe nakommemlingen worden gecreëerd op basis van bestaande individuen
    - Een **REPLACER** die bepaalt welke individuelen overleven en deel uitmaken van de volgende generatie
    - Een **MIGRATOR** die bepaalt hoe oplossingen getransfereerd worden tussen verschillende generaties
    - Eenn **ARCHIVER** die bepaalt hoe bestaande oplossingen opgeslagen worden buiten de huidige populatie

Ter illustratie van het gebruik van *ec* voor Genetische Algoritmen, hernemen we het voorbeeld dat we gebruikt hebben bij Simulated Annealing, namelijk de klassieke case om optimalisatie algoritmen en heuristieken te testen: de **Rastrigin functie** (https://en.wikipedia.org/wiki/Rastrigin_function).

```
#Problem specific components
def obj_func(solution):
    sum = 10 * len(solution)
    for i in range(0, len(solution)):
        sum = sum+(solution[i]**2-10*math.cos(2*math.pi*solution[i]))
    return sum

def generate(random = None, args =  None) -> []:
     size = args.get('num_inputs',10)
     return np.random.uniform(low=-5.12, high=5.12, size=size)

def evaluate(candidates, args = {}):
    fitness = []
    for candidate in candidates:
       fitness.append(obj_func(candidate))
    return fitness

#Metaheuristic specific components
import inspyred
from inspyred import ec #ec stands for Evolutionary computation
From random import Random
rand = Random()
ga = ec.GA(rand)
population: [ec.Individual] = ga.evolve(
    generator=generate,
    evaluator=evaluate,
    selector = ec.selectors.tournament_selection, #optional
    pop_size=100,
    maximize=False,
    bounder=ec.Bounder(-5.12, 5.12),
    max_evaluations=10000,
    mutation_rate=0.25,
    num_inputs=30)
population.sort(reverse=True)
print(population[0])
```

Maak wat aanpassingen in de parameters en kijk wat het effect is op het verloop van het genetisch algoritme.

6. Bronnen

Ding-Zhu Du and Ker-I Ko (2014): Theory of Computational Complexity, second edition, John Wiley & Sons Inc.

Labadie, N., Prins, C. and Prodhon, C. (2016): Metaheuristics for Vehicle Routing Problems, John Wiley & Sons Inc.

Sean L. (2013): Essentials of Metaheuristics, second edition, Lulu

Siarry, P (ed.)(2016): Metaheuristics, Springer International Publishing