TI2736-A Assignment 3: Path finding & TSP

David Akkerman - 4220390
Jan Pieter Waagmeester - 1222848

31 oktober 2014

3.1: Path finding through ACO

1. - Lange doodlopende straten
   - Straten met heel veel (korte) doodlopende zijstraten
   - Grote open gebieden

2. We, of de mieren, laten feromoon vallen om de voordelige paden aan te duiden. Ze laten het feromoon vallen 'op de trugweg', afhankelijk van hoe lang het pad was. Er is een waarde Q die ongeveer de geschatte routelengte is.

   $$\Delta\tau^k = Q \cdot \frac{1}{L^k}$$

   In deze formule staat $\Delta\tau^k$ voor de toe te voegen feromoon afkomstig van mier k, L^k de lengte van het pad. Als een pad korter is zullen de afzonderlijke verbindingen van dat pad meer worden versterkt dan wanneer het pad langer was.

3. Het verdampen van de feromoon is zinnig om niet in een lokaal optimum te blijven hangen. Doordat de waarde in elke cel met een bepaalde verdampingsfactor (ρ) afneemt is er kans dat een andere (wellicht betere) route wordt gekozen.

   $$\tau_{ij} = (1-\rho) \cdot \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k$$

   In deze formule is alleen de linker term $(1-\rho) \cdot \tau_{ij}$ de verdamping, de andere term beschrijft het toevoegen van feromoon op de door de mieren gebruikte paden.

4. Om te beginnen moeten we wat dingen initializeren: een stel mieren en een feromoon-matrix. Vervolgens laten we elke mier een pad zoeken in een matrix waarin de feromoon-waarden in elke cel gelijk zijn.

   Nadat alle mieren het einde gevonden hebben (of getermineerd zijn omdat ze er veel te lang over doen), kiezen we de beste mier uit. We laten een bebaald deel $(1-\rho) \cdot \tau_{xy}$ verdampen en tellen bij de cellen waar ons beste miertje geweest is een waarde $Q/|\text{ant}|$ op.

   Daarna gaat het weer opnieuw tot we het aantal iteraties hebben gehad.

```
ants = [ant0, ant1, ... antN]
pheromone[height][width] = 0.1 # initialize with small value

for i = 0 ... iterations
    foreach ant in ants
        ant.find_naive_solution(halt_after_steps=10000)

    best_ant = select_best(ants)
    pheromone = (1 - rho) * pheromone % evaporation

    for p in each best_ant.route
        pheromone (p) += Q / length(best-ant)
```

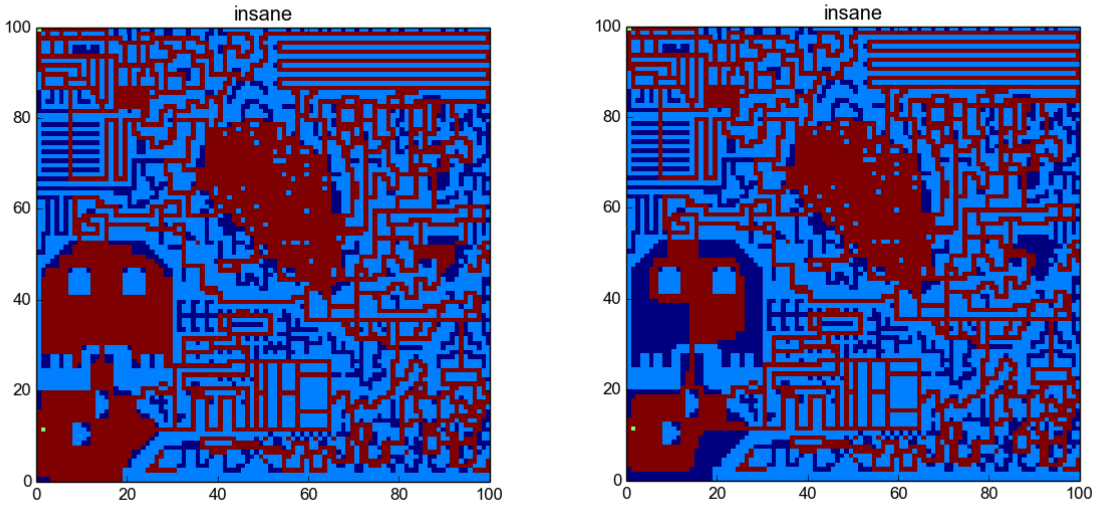
Het stappen van de afzonderlijke mieren kan als volgt worden weergegeven:

```
function find_naive_solution():
    position = maze.start
    trail = []
    while position != mazeend:
        moves = maze.get_valid_moves(position)
        move = choose_proportionally_to_tau(moves)

        trail.add(move)
        position = maze.move(move)

    return trail
```

In de volgende vraag bespreken we enkele uitbreidingen voor dit basisalgoritme.

5. We hebben een aantal versnellingsmanieren bedacht en ook bijna allemaal geïmplementeerd.

   Doolhofeliminatie Door op een slimme manier te kijken of cellen in het doolhof wel nodig bekeken moeten worden, kunnen we een deel van de gangen afsluiten:



Figuur 1: Voorbeeld van eliminatie van doodlopende stukken. Links na de 1^e iteratie, rechts na de 30^e. Donkerblauwe vlakken zijn door de mieren 'uitgezet'.

Doodlopende gangen Als een mier vanaf een bepaalde positie nog maar één kant op kan (terug), zit hij aan het einde van een doodlopende gang. We markeren die plek in de maze dan als muur, zodat er in het vervolg geen mier meer probeert in te lopen.

Open ruimtes Hoekjes in een open ruimte kunnen worden weggestreept, als er iets bekend is over de aangrenzende cellen. Als een mier achtereenvolgens een hoekje en daarna twee cellen met voldoende ruimte tegenkomt waarbij de laatste op de diagonaal zit van het hoekje, kan het hoekje worden afgestreept. Een voorbeeld in de *insane* maze is te zien in figuur 1.
Merk op dat dit bij ons soms mis gaat doordat de mieren afzonderlijk te werk gaan in eigen threads. Het kan dus zijn dat twee of meer mieren er samen voor zorgen dat er een cruciale gang wordt afgesloten. Dit is wel op te lossen, maar gaat voorbij aan deze opdracht.

Probeer min mogelijk over eigen pad te lopen Het heeft niet veel zin om terug te keren waar we al geweest zijn. We geven daarom bij het kiezen van de volgende stap de voorkeur aan de posities waar we nog niet geweest zijn.

Alleen als het niet anders kan keren we weer terug op of over onze route.

Elitisme Gebruik het pad van de beste mier over alle iteraties ook bij de feromoon-update Door de beste mier in de run van het algoritme tot nu toe te onthouden en elke keer ook te gebruiken voor de update.

Optimalisatie van het gevonden pad Vaak is een door een mier gevonden pad niet bijzonder efficient, zeker aan het begin. We kunnen zorgen dat we niet op te veel plekken feromoon neerleggen door voor dat te doen het pad van elke mier nog te optimaliseren:

- Als de mier langs het einde loopt, direct naar het einde lopen en de rest van de route overslaan.
- Als de mier weer langs start loopt het het eerste stuk overslaan.
- Als een mier een rondje loopt dat stuk overslaan.

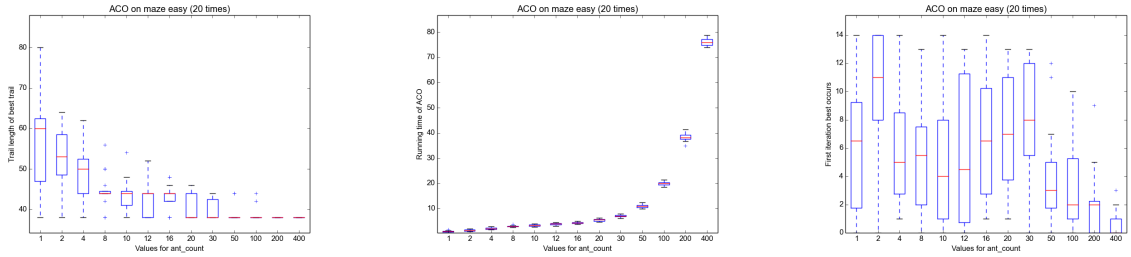
Overige ideeën Naast de voorgaande ideeën hebben we nog meer ideeën, waar voor de implementatie helaas de tijd ontbrak:

- Alle mogelijke stappen hebben lengte 1, omdat we in een grid werken. We zouden dit kunnen transformeren naar een graaf waarbij we de kruispunten zien als nodes. Dat scheelt een hele hoop mogelijke stappen, en er onstaat variatie in lengte.

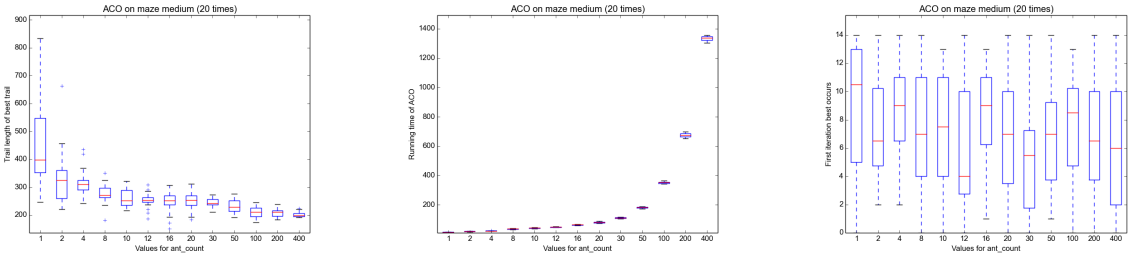
6. We hebben het algoritme laten lopen voor een aantal verschillende waarden van de parameters Q, `ant_count`, ρ and `optimize_ants`. Deze waarden hebben we 20 keer gevarieeerd voor elke doolhof terwijl de andere op een guestimated waarde stonden. Bij elke set van runs meten we drie dingen: run time, in welke iteratie de beste route voor het eerst voorkomt en de lengte van de beste route. Het was even rekenen, maar levert een hoop plaatjes op. Helaas zijn niet alle plaatjes hard maze gelukt en was er geen tijd meer om ze opnieuw door te rekenen[1].

   Excuus voor de wat kleine plaatjes, maar ze zijn goed zoombaar en anders werd het wel erg lang allemaal...[2]

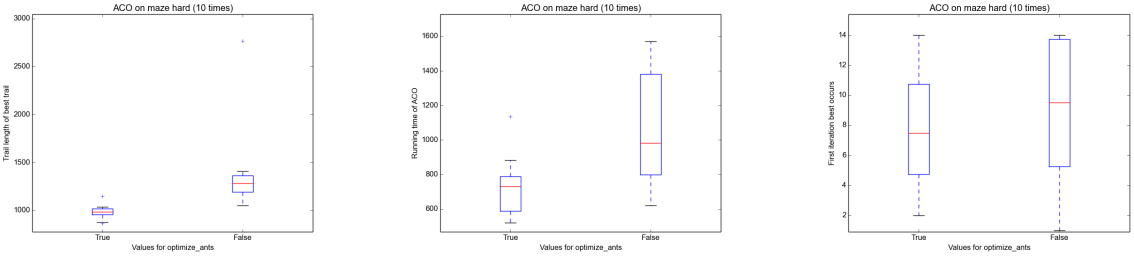
[1] Nog beter zou zijn meerdere parameters tegelijk te varieren, maar dat levert uiteraard nog veel meer rekenwerk op.
[2] Eventueel voor meer plaatjes: https://github.com/jieter/computational-intelligence/tree/master/assignment-3/code/compare-graphs



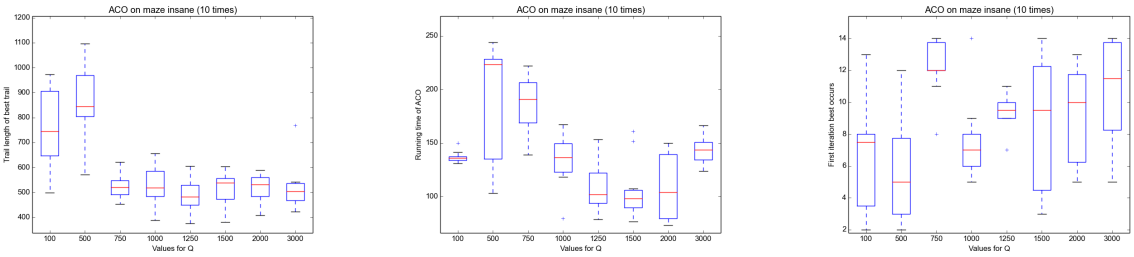
Figuur 2: Variatie van parameter **ant count** in de **easy** maze: respectievelijk lengte van beste trail, running time en de iteratie waarin de beste voor het eerst voorkomt.



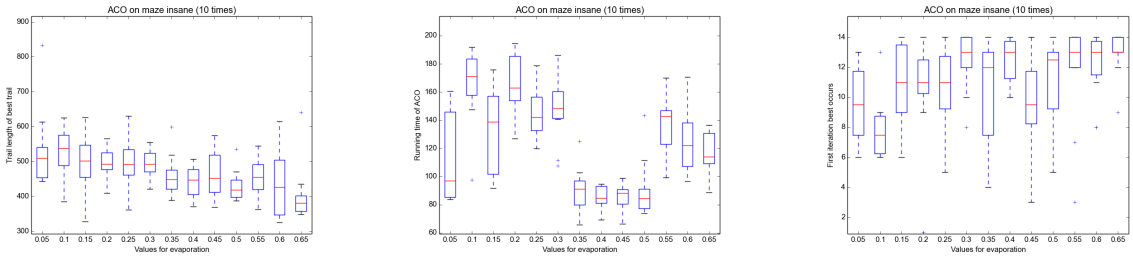
Figuur 3: Variatie van parameter **ant count** in de **medium** maze: respectievelijk lengte van beste trail, running time en de iteratie waarin de beste voor het eerst voorkomt.



Figuur 4: Variatie van parameter **ant trail optimisation** in de **hard** maze: respectievelijk lengte van beste trail, running time en de iteratie waarin de beste voor het eerst voorkomt.



Figuur 5: Variatie van parameter Q in de **insane** maze: respectievelijk lengte van beste trail, running time en de iteratie waarin de beste voor het eerst voorkomt.



Figuur 6: Variatie van parameter ρ **(evaporation)** in de **insane** maze: respectievelijk lengte van beste trail, running time en de iteratie waarin de beste voor het eerst voorkomt.

Easy maze Op grond van deze plaatjes maakt het niet zo veel uit wat we voor parameters gebruiken. Sowieso maakt het allemaal niet zoveel uit, want deze maze is zo klein dat de kortste route bijna altijd wel vrij snel gevonden wordt.

Met heel veel mieren is de kans zelfs vrij groot dat de beste oplossing al in de eerste iteratie gevonden wordt (zie figuur 2).

Medium maze We zien hier, bijvoorbeeld in figuur 3 dat het vanaf 12 mieren niet zoveel zin meer heeft om meer mieren toe te voegen.

Hard maze Bij de hard maze is goed zichtbaar dat ons padoptimalisatiealgoritme voor het pad van afzonderlijke mieren een zinnige verbetering is: het halveert de run time van het algoritme (figuur 4). Bovendien levert het ook nog eens kortere eindoplossingen op.

Insane maze De insane maze laat een ander beeld zien (figuren 5 en 6). Waardes voor ρ rond de 0.4 lijken hier vrij goed te werken om snel klaar te zijen, maar lage waardes leveren weer gemiddeld betere oplossingen op. Ook hier heeft het weinig zin om meer mieren toe te voegen dan ongeveer 15.

Dat brengt ons uiteindelijk bij de volgende waarden:

	easy	medium	hard	insane
# iterations	10	15	30	20
Q	50	300	4000	1000
`ant_count`	10	15	40	50
ρ (evaporation)	0.15	0.4	0.4	0.5
`optimize_ants`	`True`	`True`	`True`	`True`
Kortste gevonden route	38	141	837	295

Merk op dat door gebruik van elitisme een hoge waarde van evaporatie als eenmaal een korte overall route gevonden is niet zoveel meer doet omdat die route elke keer versterkt wordt. We zien dus ook dat veel iteraties met dit algoritme minder zin heeft.

7. Zoals aan het einde van de vorige vraag al opgemerkt is het waarschijnlijk dat doordat we globaal en iteratie-elitisme gebruiken (alleen de global best en iteration best doen mee met de feromoon-update), de waarden van Q en ρ minder relevant zijn.

   In het algemeen lijkt het wel zo te zijn dat als een maze meer open ruimtes heeft, een hogere waarde van ρ beter werkt.

   Wat deze observaties ingewikkeld maakt is het maze-eliminatie-algoritme wat we hebben toegevoegd, wat er voor zorgt dat gedurende de loop van het algoritme het doolhof verandert: er worden doodlopende gangen afgestoten en open ruimtes dichtgebouwd.

   Het aantal mieren is vooral afhankelijk van de grootte van de maze: meer mieren bij grotere doolhoven.

   Voor de waarde van Q voldoet het prima om het algoritme eerst een keer te laten lopen met een semiwillekeurige waarde en dan de kortste gevonden lengte als waarde voor Q te gebruiken.

Het optimaliseren van het pad (`optimize_ants`) behoeft niet veel discussie. Dat is gewoon altijd beter om aan te hebben.

3.2: The Traveling Robot Problem

8. Het TSP (Reizende Verkoper Probleem) gaat over een verkoper die een aantal, bijvoorbeeld 20, steden moet bezoeken. Natuurlijk vindt hij het prettig om dit in een zo efficiënt mogelijke volgorde te doen. Het probleem is dat dit erg moeilijk is om uit te rekenen, want het aantal mogelijke volgordes is $20! = 2.4 \cdot 10^{18}$, wat hij logischerwijs niet allemaal door kan rekenen.

9. Het verschil tussen een gebruikelijk TSP en het Reizende Robot Probleem is dat de afstanden tussen de steden van het TSP alleen afhankelijk zijn van hun positie, terwijl het zich voor ons probleem in een doolhof afspeelt, waarbij verschillende routes tussen twee punten genomen kunnen worden, die niet allemaal zo makkelijk te berekenen zijn. Het kan ook voorkomen dat twee punten praktisch naast elkaar liggen, maar de route ertussen toch erg lang is.

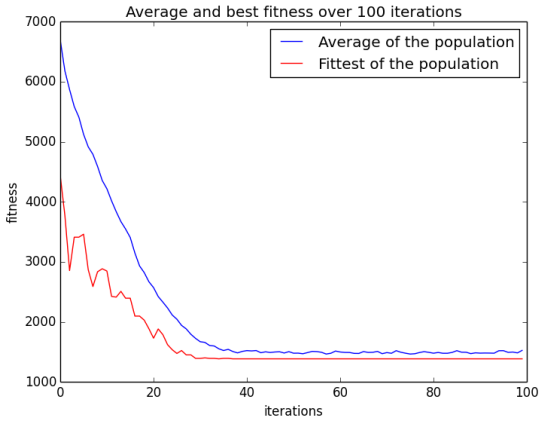
10. CI-technieken zijn erg handig om dergelijke problemen op te lossen, doordat ze de rekenkracht van een computer op een slimme manier gebruiken, namelijk door slim naar oplossingen te zoeken en niet in het wilde weg alles te proberen, waardoor (bij benadering) het optimum veel sneller gevonden kan worden dan met enkel brute rekenkracht. Door de complexiteit van het doolhof zou het een mens veel langer kosten dan een computer om een goede route te vinden.

11. De chromosomen bestaan uit 18 genen. Elk gen representeert een ander item in het doolhof. De volgorde van deze genen bepaalt in welke volgorde de producten opgehaald worden.

12. De fitness functie van het TSP is één gedeeld door de totale afstand die de robot van begin tot eind aflegt als hij alle producten oppakt. Wij gebruiken voor de berekengingen in plaats daarvan de totale afstand. Dit is dus eigenlijk niet de fitness, maar de inverse van de fitness.

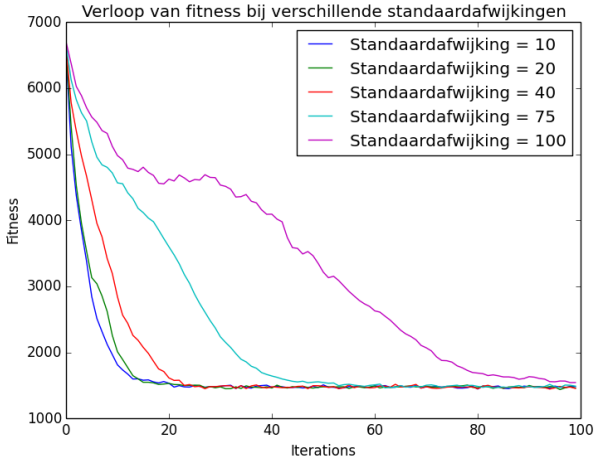
$$F = s_{b,1} + \sum_{i}^{N-1} s_{i,i+1} + s_{N,e}$$

Waarbij F de totaal afgelegde afstand is, N het aantal genen in een chromosoom en $s_{i,i+1}$ de afstand tussen het i^{ste} item en het $i + 1^{ste}$ item. Hierbij is b het start- en e het eindpunt. In figuur 7 is zichtbaar hoe fit het gemiddelde van de populatie en de fitste van de populatie zijn.



Figuur 7: Verloop van de gemiddelde fitness en de beste fitness

13. De ouders worden geselecteerd met een kans die evenredig is met de fitness. Alle oorspronkelijke chromosomen worden gesorteerd op fitness. Vervolgens wordt er een normale kansverdeling gebruikt met verwachtingswaarde 0 en standaardafwijking 75. De absolute waarde hiervan geeft aan welke chromosoom uit de gesorteerde reeks genomen zal worden. In figuur 8 wordt geïllustreerd wat er gebeurt als de standaardafwijking veranderd wordt. Een lage standaardafwijking zorgt voor snellere convergentie naar een lokaal optimum. Als de standaardafwijking wordt verhoogt, vertraagt het proces weliswaar, maar hierdoor wordt de kans op een gelukkige afwijking ook groter, waardoor het makkelijker is voorbij een lokaal optimum te springen.



Figuur 8: Verloop van fitness bij verschillende waarden van de standaarddeviatie

14. De nieuwe chromosomen worden gecreëerd door zowel cross-over als mutatie toe te passen. Cross-over houdt in dat van een ouderchromosoom een aantal genen gekopiëerd wordt, wat vervolgens de basis is van het kindchromosoom. De missende genen worden aangevuld door deze bij het kind in te plakken in de volgorde waarop ze voorkomen bij een tweede ouderchromosoom.
Mutatie betekent dat van een ouderchromosoom een willekeurig aantal genen geknipt, omgedraaid en teruggeplakt wordt. Dit wordt gedaan omdat op deze manier stukjes van dicht bij elkaar liggende items niet uit elkaar worden gescheurd, maar alleen in de omgekeerde volgorde bezocht worden, wat kan zorgen voor een verbetering.
In een kwart van de gevallen wordt er mutatie toegepast en in driekwart van de gevallen cross-over. Dit wordt gedaan omdat cross-over in minder gevallen effectief is, maar wel nodig om naar een goede oplossing te komen. Hierdoor moet er in meer gevallen cross-over gebruikt worden om het nuttig te maken.

15. De oorspronkelijke chromosomen worden zó gecreëerd dat ze precies alle items eenmaal bevatten. De manieren waarop gereproduceerd wordt, zijn zo ontwikkeld dat het niet kan voorkomen dat een kindchromosoom wel tweemaal hetzelfde getal bevat en een ander niet. Bij het mutatie- en cross-overproces wordt namelijk enkel de volgorde van de items veranderd, niet de items zelf.

16. Doordat er een normale verdeling gebruikt wordt om de ouders te selecteren, bestaat er altijd een kans dat er een slechter presterend chromosoom gepakt wordt. Dit chromosoom kan in een enkel geval wel betere kinderen geven, waardoor er voorbij een lokaal minimum gegaan kan worden. Om de optimale oplossing te vinden, moet echter een oneindig aantal iteraties plaatsvinden, want je weet niet wanneer de optimale oplossing gevonden gaat worden, maar enkel dát hij ooit gevonden wordt.

17. Elitisme houdt in dat een aantal van de fitste chromosomen niet sterven maar ook weer meedoen in de volgende generatie. Op deze manier voorkom je dat er alleen maar minder fitte chromosomen komen. Een gevaar is echter dat er op deze manier te vroeg een lokaal minimum gevonden wordt en niet verder wordt gezocht. Wij hebben elitisme gebruikt in zoverre dat het mogelijk is dat er bij mutatie slechts een rij van één getal wort geïnverteerd, wat geen verandering in het chromosoom aanbrengt, waardoor de ouder dus ongeschonden naar de volgende generatie doorgaat.