Wat zijn zelflerende computersystemen?

# Deelvraag 1 inleiding

Elk jaar boekt de mens grootschalige vorderingen op het gebied van computers, zowel hardware als software. Iets waar wij echter nog niet in geslaagd zijn te maken is een ware **Artificial Intelligence**, al lukt de schijn van denken creëren steeds beter. Voorbeelden zijn de ‘persoonlijke assistenten’ die inmiddels in elke smartphone geïntegreerd zijn. *Siri*, *Google Now* en *Cortana* maken gebruik van spraakherkenning om de gebruiker de gevraagde informatie te tonen, maar denken zoals mensen doen ze hierbij niet. Hoe een computersysteem toch beter kan worden in het imiteren van menselijk gedrag en van zijn eigen fouten kan leren onderzoeken we in deze deelvraag.

# Verschillende algoritmes

Computers hebben dus geen bewustzijn. Om deze reden kunnen ze niet zelf bepalen iets te doen. Waar computers wel in uitblinken, is het uitvoeren van taken die ze zijn opgelegd. Vaak komen deze taken in de vorm van code. Via code kan je computers opdrachten geven, bijvoorbeeld “laat een scherm zien”. De boodschap valt echter niet op deze manier over te brengen. Afhankelijk van de taal waarin je programmeert zijn er vaste commando’s waar de computer op zal reageren.  
Naarmate de opdracht die je een computer wil laten uitvoeren complexer worden, zal ook het gebruik in deze commando’s een verandering zien. Hier komen algoritmes in het spel. Een algoritme is een soort stappenplan voor de computer, waarin een complexere handeling in duidelijke opdrachten weergegeven wordt. De volgende definitie geeft een betekenis in de meest algemene zin: een algoritme is een eindige reeks instructies om vanaf een beginpunt een bepaald doel te bereiken.[a]  
Een toegankelijke vergelijking is koken. Er is een **input** van voedsel waar uiteindelijk een gerecht uit moet komen, de **output**. Voor het tot stand komen van dit gerecht gebruik je misschien een recept. Dit recept is als het ware het algoritme.  
Uit de gegeven definitie is af te leiden dat het aantal mogelijke algoritmes ontzettend groot is. Niet alleen is het een ruim begrip, ook kan het desbetreffende doel waarschijnlijk op meerdere manieren bereikt worden. In deze verschillende methodes kan de een echter beter zijn dan de andere, bijvoorbeeld door efficiënter te zijn.

Uiteraard zijn er ook vele algoritmes die gebruik maken van toepassingen, zoals een *queue* en een *stack*, die betrekking hebben tot ons onderwerp. Enkele hiervan zullen hier beschreven worden:

# Breadth-first search (BFS)

Dit algoritme, bedacht in de jaren vijftig van de vorige eeuw door E.F. Moore[b], een Amerikaans professor in de wiskunde en computer sciences en een voortrekker in kunstmatig leven, is een zoekalgoritme voor datasets in de vorm van grafieken of ‘boom’-structuren. In deze dataset wordt een **node** als oorsprong benoemd, de **root***.* Ook wordt een bepaalde uitkomst als doel gesteld. Vervolgens krijgt elke node drie waardes aangewezen:

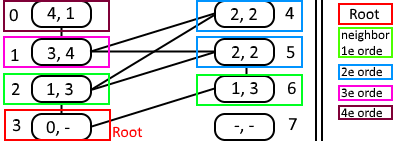
* De afstand van de huidige node naar de root. Dit is het aantal stappen dat gezet moet worden om bij de root te komen.
* De node die vóór de huidige node kwam, de **predecessor**. Anders gezegd: bij welke node je uitkomt als je een enkele stap terug zet.
* Een **state**. De state houdt bij of een node al gecontroleerd is.

Bij Breadth-first search wordt gebruik gemaakt van een queue. Dit is een lijst waar nodes aan toegevoegd en uitgehaald kunnen worden. Net zoals een daadwerkelijke wachtrij wordt het ‘als eerste erin, als eerste eruit’ principe toegepast.

Het algoritme ziet er als volt uit:

1. Maak een lege lijst S voor bezochte nodes.
2. Maak een lege lijst Q met de queue.
3. Benoem één node als root en voeg deze toe aan S.
4. Voeg de root toe aan Q.
5. Terwijl Q niet leeg is:
   1. Haal de voorste node uit de queue. Dit is de ‘current’ node.
   2. Als current het doel is:
      1. Return current.
   3. Voor elke node die grenst aan\* current:
      1. Als deze node nog niet bezocht is en dus niet in S zit:
         1. Voeg de node toe aan S.
         2. Zeg dat de predecessor van de node de current node is.
         3. Haal de node uit de queue.

*\*Aangrenzend zijn betekent hier ‘in directe verbinding staan met’.*



Figuur \*\*\*\*: de volgorde waarin de nodes bezocht worden is 3 – 2 – 6 – 4 – 5 – 1 – 0

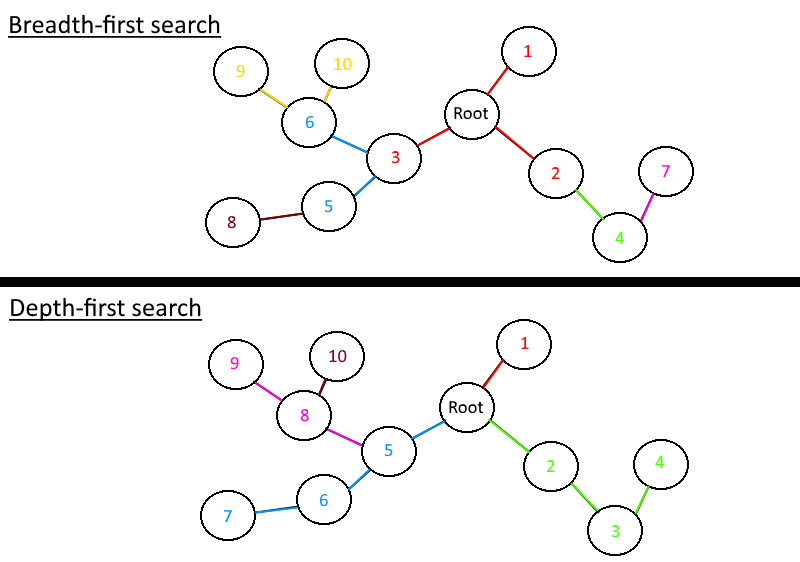
Hierboven is een voorbeeld van een simpele dataset weergegeven, genummerd van 0 tot en met 7. Node 3 is de root en 0 het doel. Voor het gemak staan in elke node twee getallen: de afstand tot 3 en het nummer van de predecessor. Met kleuren zijn aangegeven in welke volgorde de nodes bezocht worden.  
3 is de root. 3 wordt toegevoegd aan de lijsten Q en S. Hierdoor is de queue niet leeg. 3 wordt weer uit de queue gehaald en één voor één worden de aangrenzende nodes bekeken. Hierbij worden ze toegevoegd aan de stack. Omdat zowel 2 als 6 het niet het doel is, herhaald het algoritme zich. Nu wordt 2 gecheckt. Het doel is niet gevonden. Daarna 6, ook zonder succes. (Let hierbij op dat node 5 niet nogmaals gecheckt wordt, dit is namelijk als bij node 2 gedaan en is dus al aanwezig in lijst S) Intussen zij 4 en 5 toegevoegd aan de queue, ze zijn immers verbonden met 2. Ook hier wordt het proces herhaald, node 1 zit nu in de queue. Uiteindelijk wordt node 1 gecheckt en wordt het doel, node 0, gevonden.

Met BFS kan je de weg van de root naar het doel achterhalen. Dit is nuttig als je bijvoorbeeld een wegennetwerk hebt en wil weten wat de kortste weg van de ene naar de andere stad is.

# Depth-first search (DFS)

Evenals breadth-first search is depth-first search een algoritme voor het doorlopen van datasets in grafieken of trees. DFS verschilt echter op twee manieren van BFS:

* Depth-first search gebruikt een stack in plaats van een queue. Waar nodes in een BFS systeem in een wachtrij werden geplaats met een “Als eerst erin, als eerst eruit” principe, handhaaft een DFS systeem een wachtrij meer vergelijkbaar met een stapel papieren. Telkens pak je het bovenste element van de stapel om mee te werken, maar als je iets in de wachtrij stopt, komt dit ook weer bovenop de stapel te liggen. De meest recente toevoeging zal dus als eerste weer eruit gehaald worden.
* Breadth-first search begon bij een root. Vervolgens werd gekeken naar alle neighbors. Als de gewenste uitkomst niet tussen deze neighbors zit, worden de neighbors van deze neighbors gecontroleerd. Dit proces herhaalt zich totdat het doel gevonden is.  
  Depth-first search begint ook bij een root, maar kijkt direct naar een weg tot een node bereikt is die geen neighbors meer heeft. Als het doel dan niet bereikt is wordt een andere weg geprobeerd. Hiervoor wordt gebruik gemaakt van **recursive backtracking**.



Figuur \*\*\*\*

In figuur \*\*\*\* is de werking van BFS en DFS weergegeven. Het getal in elke node geeft aan als hoeveelste het bereikt wordt. De kleuren representeren de gebieden die het algoritme per stap doorloopt.

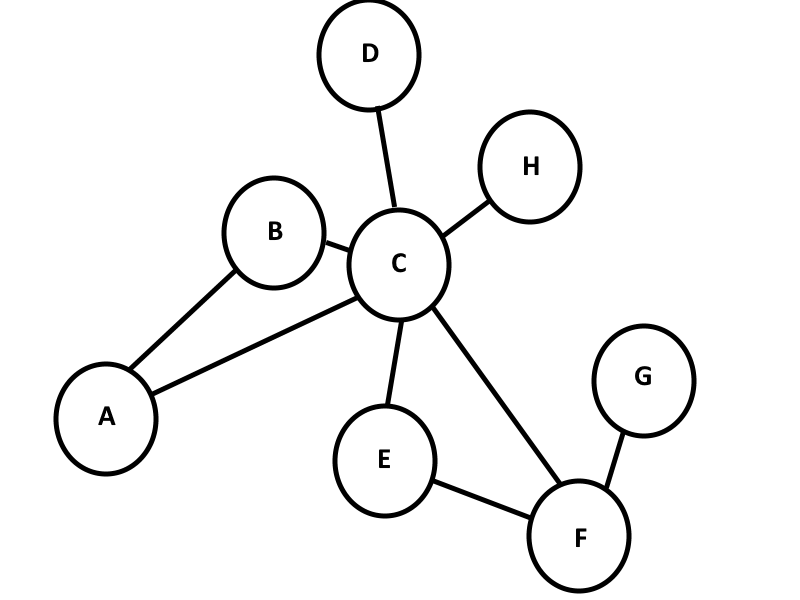
Ook bij DFS hebben de nodes een state: bezocht of niet bezocht.  
Ten eerste wordt de root gekozen en wordt deze als bezocht opgeslagen. Zoals te zien wordt er vanaf de root één (willekeurige) neighbor gekozen om te onderzoeken. Elke bezochte neighbor wordt als bezocht genoteerd. De root wordt in de stack geplaats. Als de gekozen neighbor niet het doel is, wordt de meest recentelijk toegevoegde node, de root, gehaald, de eerst bezochte had immers geen aanliggende nodes om te onderzoeken. Deze tweede neighbor, cirkel 2 in de figuur, heeft wel een neighbor. Deze wordt gecontroleerd, evenals diens neighbors. Telkens wordt de huidige node toegevoegd aan de stack.  
Als het programma de laatste in een reeks nodes bereikt heeft, wordt de bovenste node uit de stack gepakt en gekeken of daar nog niet bezochte nodes aan grenzen. Dit wordt backtracking genoemd.  
Dit proces wordt herhaald totdat het doel gevonden is of totdat alle nodes geweest zijn.

Als vuistregel kan het volgende gehanteerd worden: depth-first search wordt gebruikt als je weet dat er maar één uitkomst is, breadth-first search als je de kortste weg wil weten.

# Voorbeelden algoritmes

Algoritmes hebben meestal vele toepassingen. Hier zijn enkele voorbeelden van de eerder genoemde algoritmes.

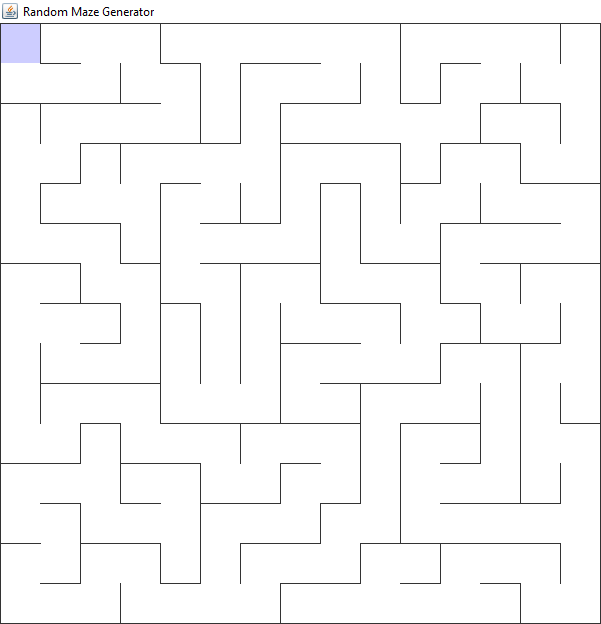
## Breadth-first search



Figuur \*\*\*\*

In figuur \*\*\*\* is een dataset te zien, bijvoorbeeld een telefoonboom. Elke cirkel representeert een persoon. Zo kan persoon A de personen B en C bellen, maar A bezit geen andere telefoonnummers. Toch zou hij een boodschap naar H kunnen sturen: via C.   
Stel, persoon A wil nu iets tegen F zeggen. In een kleine dataset als deze is makkelijk met het oog te zien dat de snelste manier hiervoor A – C – F is en dat A – B – C – E – F veel langer is. Bij grotere datasets is dit echter al snel moeilijk met zekerheid te zeggen. Hiervoor kan breadth-first search ingezet worden.

## Depth-first search

****

Figuur \*\*\*\*

Depth-first search kan gebruikt worden voor zowel het maken als oplossen van doolhoven. In figuur \*\*\*\* is een doolhof te zien dat gemaakt is met behulp van DFS. (\*\*\*\*programma gemaakt in kader van deze deelvraag\*\*\*\*). Het algoritme in de vorm van een stappenplan is als volgt:

1. Maak de start cel current en markeer deze als bezocht.
2. Terwijl er nog niet bezochte cellen aanwezig zijn:
   1. Als current neighbors heeft die nog niet bezocht zijn:
      1. Kies willekeurig een van de neighbors
      2. Voeg current toe aan de stack
      3. Verwijder de muur tussen de huidige cel en de gekozen cel
      4. Benoem de gekozen cel als current en zet de state op bezocht
   2. Anders, als de stack niet leeg is:
      1. Haal de laatst toegevoegde cel uit de stack en verwijder deze hieruit
      2. Maak deze cel current

# Zelflerend?

Breadth-first search en Depth-first search zijn beide algoritmes met vele toepassingen. Toch kunnen beide algoritmes niet als zelflerend worden beschouwd, ze verbeteren hun manier van zoeken namelijk niet. Hoe zit een zelflerend systeem dan wel in elkaar?

**Gebruikte bronnen (temporary)**

[a] Link: <http://www.woorden.org/woord/algoritme>   
Geraadpleegd op: 22- 05 – 2017   
Laatst gewijzigd op: niet bekend

Link: <https://beebom.com/examples-of-artificial-intelligence/>   
Geraadpleegd op: 22- 05 – 2017   
Laatst gewijzigd op: 26 – 09 – 2017

Link: <http://study.com/academy/lesson/what-is-an-algorithm-in-programming-definition-examples-analysis.html>  
Geraadpleegd op: 22- 05 – 2017   
Laatst gewijzigd op: 2017

[b] Link: <http://www.secfac.wisc.edu/senate/2003/0929/1727(mem_res).pdf>   
Geraadpleegd op: 23- 05 – 2017   
Laatst gewijzigd op: 23 – 09 – 2003

Link: <https://www.khanacademy.org/computing/computer-science/algorithms/breadth-first-search/a/the-breadth-first-search-algorithm>   
Geraadpleegd op: 22- 05 – 2017   
Laatst gewijzigd op: 26 – 09 – 2017

Link: <http://www.graph500.org/specifications#sec-5>   
Geraadpleegd op: 22- 05 – 2017   
Laatst gewijzigd op: 26 – 09 – 2017

Link: <https://www.kirupa.com/developer/actionscript/depth_breadth_search.htm>   
Geraadpleegd op: 27- 05 – 2017   
Laatst gewijzigd op: 13 – 01 – 2006

Link: <http://www.algolist.net/Algorithms/Graph/Undirected/Depth-first_search>   
Geraadpleegd op: 27- 05 – 2017   
Laatst gewijzigd op: niet bekend

Link: <https://en.wikipedia.org/wiki/Maze_generation_algorithm#Recursive_backtracker>   
Geraadpleegd op: 27- 05 – 2017   
Laatst gewijzigd op: 19 – 05 – 2017

Link: <https://www.khanacademy.org/computer-programming/depth-first-traversals-of-binary-trees/934024358>   
Geraadpleegd op: 27- 05 – 2017   
Laatst gewijzigd op: 2012