Algoritmen en Datastructuren 2: Gerichte Graafalgoritmen

Youri Coppens
Youri.Coppens@vub.be

Opmerking

Tijdens dit WPO zullen we niet alle gerichte graafalgoritmen uit dit hoofdstuk bespreken. Deze hebben echter wel allemaal een implementatie in Scheme. Al deze algoritmen zijn terug te vinden in de folder `a-d/graph-algorithms/directed/`. Hier volgt een overzicht van waar je alle algoritmen terug kan vinden:

- De file `basic.rkt` bevat simpele hulpprocedures `in/out-degrees`, `copy` en `transpose`, om respectielijk de in- en uitgraden van knopen te berekenen, een gerichte graaf te kopiëren of een getransponeerde gerichte graaf op te stellen (de richting van bogen werd omgekeerd). Deze komen van pas in de overige implementaties van de gezien algoritmes in dit hoofdstuk.
- De file `topological-sorting.rkt` bevat de geziene algoritmes om (gewogen) gerichte acyclische grafen topologisch te sorteren.
- De file `connectivity.rkt` bevat de algoritmes van Kosaraju en Tarjan voor sterksamenhangendheid na te gaan.
- De file `single-source-shortest-path.rkt` bevat de algoritmes van Bellman-Ford, Dijkstra en Lawler.
- De file `traclo-unweighted.rkt` bevat algoritmes om de transitieve sluiting van een gerichte graaf te berekenen, bv. Warshall.
- De file `traclo-weighted.rkt` bevat het algoritme van Floyd-Warshall om het All-Pair-Shortest-Path-probleem uit te rekenen.

1 Cycliciteit in gerichte grafen

Voor ongerichte grafen hebben we gezien dat een simpele toepassing van DFT ons toelaat om na te gaan of een graaf al dan niet een cyclus bevat. Dit geziene predikaat `cyclic?` werkt echter niet voor **gerichte grafen**.

- Waarom is dat zo? In de file `Vraag1-Tests.rkt` staan enkele grafen om je te helpen.
- Implementeer een nieuw predikaat `cyclic*?` dat werkt voor **zowel** gerichte als ongerichte grafen. Je kan hiervoor vertrekken van het bestaande predikaat `cyclic?`.

Je kan deze oefening maken in de file `Vraag1-Tests.rkt`. Hier wordt je oplossing meteen getest op gerichte en ongerichte grafen.

2 Sterk-samenhangendheid

In de theorie zagen we twee algoritmen om de sterk-samenhangendheid van gerichte grafen te onderzoeken. Een daarvan is het algoritme van Tarjan. Dit algoritme maakt gebruik van een stack om de ontdekte knopen bij te houden. Wanneer een nieuwe sterk-geconnecteerde component gevonden wordt (in `node-processed`) zal de stack deels afgebouwd worden en zullen de knopen die eraf gehaald worden aan de nieuwe component toegewezen worden. Het feit dat een dergelijke knoop nu tot een component behoort wordt onthouden met behulp van een knoop-geïndexeerde vector van booleans (`included`).

We kunnen deze implementatie echter efficiënter maken op vlak van geheugengebruik. Het is namelijk mogelijk om de stack te elimineren door bovenstaande knoop-geïndexeerde vector eveneens hiervoor te gebruiken. Omdat de stack nooit groter kan worden dan het aantal knopen in de graf is dit perfect mogelijk. De bestaande functie van deze vector mag natuurlijk niet in het gedrang komen.

Ga hiervoor als volgt te werk:

- Vervang het stack ADT door een stack opgeslagen in een **nieuwe** knoop-geïndexeerde vector + een top variabele als pointer naar de knoop bovenaan van de stack. Vervang dus alle `push!` en `pop!` operaties door code die je stack-vector en de top variabele gebruikt i.p.v. het stack ADT. Laat de `included` vector voorlopig ongemoeid.
- Breng de functies van de stack-vector en de included-vector samen in 1 vector (de stack-vector dus). Hoe zal je nu aangeven wanneer een knoop *included* is?

Je kan deze oefening maken in de file `Vraag2-Tests.rkt`. Je kan jouw oplossing vergelijken met de uitkomst van het originele algoritme van Tarjan.

3 All Pair Shortest Path Problem

In de theorie hebben we twee algoritmen gezien om het all pair shortest path probleem op te lossen: het algoritme van Floyd-Warshall en het algoritme van Johnson.

De implementatie van het algoritme van Floyd-Warshall levert een matrix af met de afstanden tussen alle paren van knopen in de graf. De overeenkomstige paden worden echter niet teruggegeven. We zullen nu een aanpassing implementeren zodat ook deze informatie teruggegeven wordt. Gebruik hiervoor opnieuw een matrix. Belangrijk is dat

je deze matrix correct interpreteert. De waarde die staat op index (i, j), bv. k, is de knoop op het kortste pad van i naar j waarna je knoop j uiteindelijk zal bereiken. Om de vorige knoop van het pad te vinden moet je dan gaan kijken op index (i, k) en zo verder. Op die manier bouw je het kortste pad van i naar j achterstevoren terug op.

Je kan deze oefening maken in de file `Vraag3-Tests.rkt`.