Algoritmen en Datastructuren 2: Grafen

Youri Coppens
`Youri.Coppens@vub.be`

Opmerking

Al de code voor graaf-implementaties vind je terug in `a-d/graph`. We onderscheiden 3 deellibraries:

1. `unweighted`: grafen bestaande uit slechts bogen en knopen.
2. `weighted`: gewogen grafen, bogen bevatten een gewicht.
3. `labeled`: gelabelde grafen, zowel knopen als bogen kunnen een label toegewezen kregen.

Elke deellibrary is voorzien van een configuratie-bestand, `config.rkt`. In dit bestand kan je wisselen tussen de gewenste implementatie van de graaf in kwestie, nl. een *adjacency matrix* (`adjacency-matrix.rkt`) of *adjacency list* (`adjacency-list.rkt`). Het is de bedoeling dat als je gebruik wilt maken van de graaflibraries, je de config-file(s) importeert en de gewenste versie in de config-file uitcomment. Bij constructie van een graaf (`new`) dien je te bepalen of de graaf al dan niet gericht is d.m.v. een boolean mee te geven bij de parameters.

1 Gelabelde grafen als adjacency list

In de theorie werd het concept van gelabelde grafen behandeld. Dit zijn grafen waarbij er labels kunnen geplaatst worden op *zowel* nodes als edges. Een voorbeeldimplementatie kunnen jullie vinden in `a-d/graph/labeled/adjacency-matrix.rkt`. Deze implementatie gebruikt als onderliggende representatie voor de gelabelde graaf een adjacency matrix.

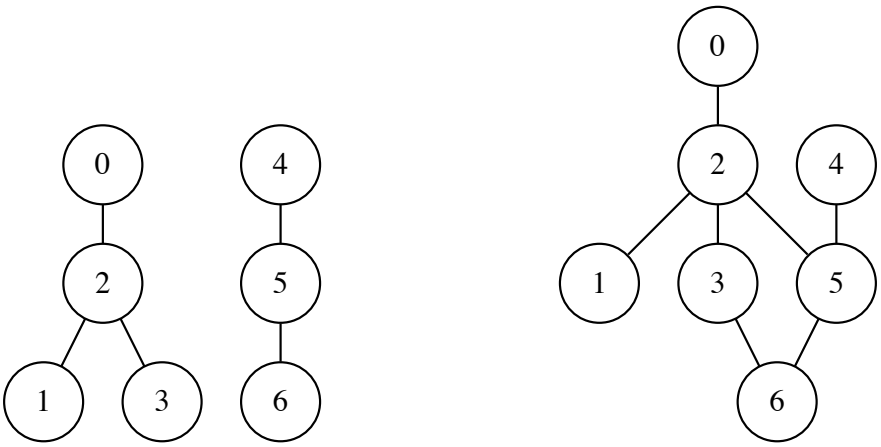
a) Implementeer nu zelf een ADT voor gelabelde grafen, maar met als onderliggende representatie de adjacency list. Je ADT moet minstens volgende operaties ondersteunen:

- Een edge toevoegen met zijn label (`add-edge!`)
- Een label van een node veranderen (`label!`)
- De label van een node opvragen (`label`)
- De label van een edge opvragen (`edge-label`)
- Een `for-each-node` waarvan de functie die je kan meegeven als argument een node en zijn label als parameters moet hebben
- en tenslotte, een `for-each-edge` waarvan de functie die je kan meegeven als argument een edge en zijn label als parameters moet hebben

Je kan hiervoor inspiratie opdoen door te kijken naar de implementatie van gewogen grafen met een adjacency list (`a-d/graph/weighted/adjacency-list.rkt`). Test je implementatie met behulp van `Vraag1-Tests.rkt` (meegeleverd met de opgave).

b) Wat zijn de voor- en nadelen ten opzichte van de implementatie met een adjacency matrix?

2 Connectiveit van grafen testen met disjoint sets



(a) Niet-geconnecteerde graaf (b) Geconnecteerde graaf

Figuur 1: Voorbeelden van (niet-)geconnecteerde grafen

Tijdens de theorie werd het principe van geconnecteerde en niet-geconnecteerde grafen besproken. Figuur 1 toont hiervan een voorbeeld. Je zal later in de theorie zien dat het mogelijk is om met "graph traversals" na te gaan of een graaf geconnecteerd is. Je kan dit echter ook doen met iets dat we al kennen, namelijk Disjoint Sets.

a) Hoe zou je aan de hand van Disjoint Sets kunnen nagaan of een graaf geconnecteerd is?

b) Voor welke type grafen zal dit echter niet werken?

c) Implementeer een predicaat `(connection? graph from to)` dat gebruik maakt van Disjoint Sets en nagaat of er een connectie bestaat tussen knoop `from` en knoop `to`. Maak hiervoor een kopie `a-d/graph/unweighted/adjacency-matrix.rkt` (dit is ook mogelijk met een adjacency list).

d) Implementeer ook in dit ADT een predicaat `(connected? graph)` dat in $O(1)$ nagaat of een graaf geconnecteerd. *Hint:* je mag hiervoor het Disjoint Sets ADT (`a-d/disjoint-sets/optimized.rkt`) aanpassen.

e) Welke graafoperatie is erg lastig te implementeren om deze implementatie volledig functioneel te krijgen? Je hoeft dit dan ook niet te doen, later zie je hiervoor betere algoritmen dan Disjoint Sets.

Om je programmeerwerk te testen kan je gebruikmaken van `Vraag2-Tests.rkt` (meegeleverd met de opgave).