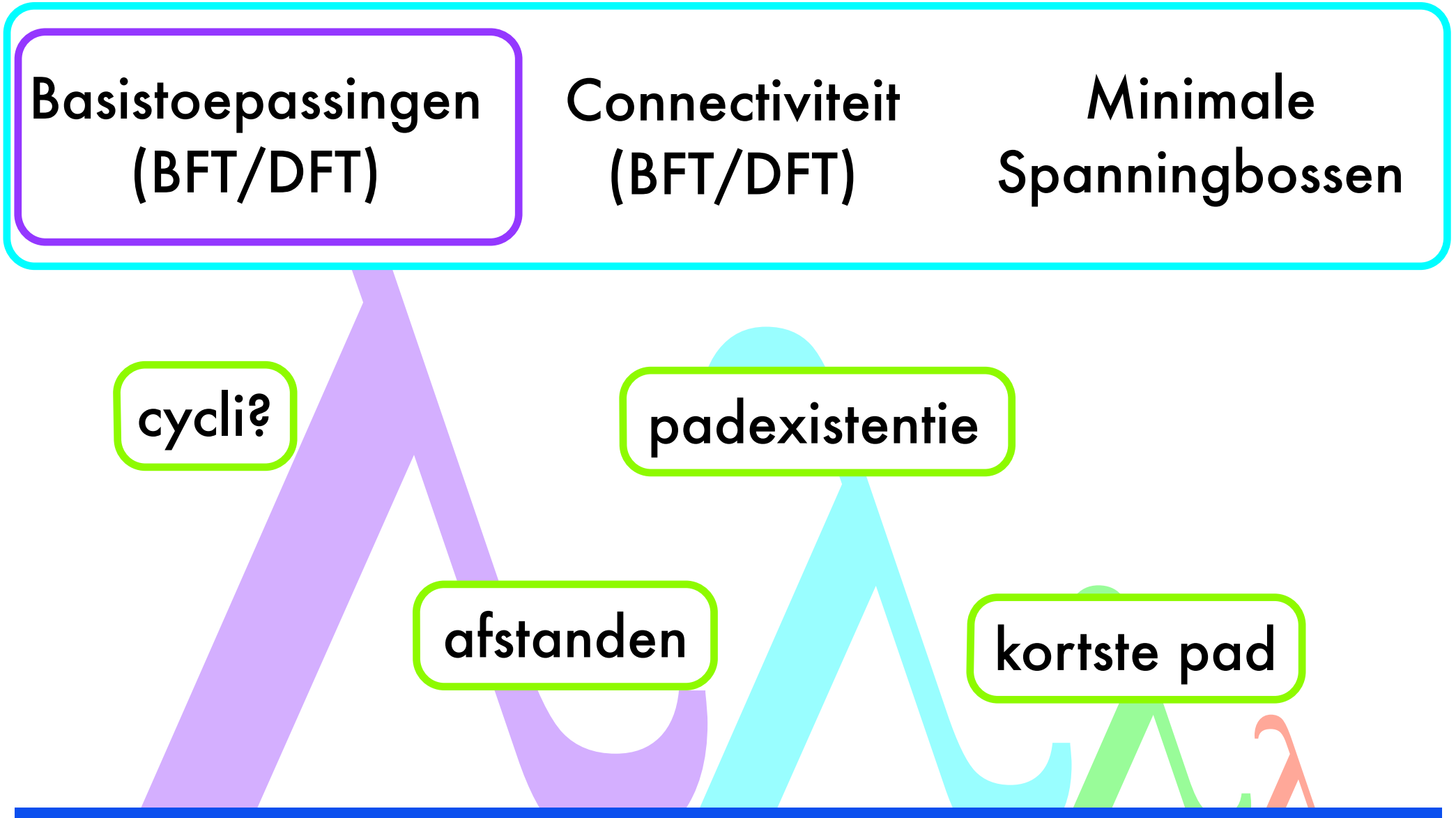
Hoofdstuk 15

Ongerichte Graafalgoritmen

Drie Groepen Algoritmen



Een Test op Cycli

```
(define (cyclic? g)
  (define tree (make-vector (order g) '()))
  (define cyclic #f)
  (dft g
       root-nop
       node-nop
       node-nop
       (lambda (from to)
         (vector-set! tree to from))
       edge-nop
       (lambda (from to)
         (if (not (eq? (vector-ref tree from) to))
             (set! cyclic #t))))
  cyclic)
```

Een graaf is cyclisch als en slechts als hij een terugboog heeft

De afstand tussen 2 knopen

Triviale toepassing van BFT (c.f. karakterisaties)

```
(define (distance g from to)
  (define distances (make-vector (order g) +inf.0))
  (vector-set! distances from 0)
  (bft g
       root-nop
       (lambda (node)
         (not (eq? node to)))
       (lambda (from to)
         (vector-set! distances to (+ (vector-ref distances from) 1)))
       edge-nop
       (list from))
  (vector-ref distances to))
```

Vertrekken bij from

Verschillende "Paden"-problemen

```
(define (shortest-path g from to)
  (define paths (make-vector (order g) '()))
  (vector-set! paths from (list from))
  (bft g
       root-nop
       (lambda (node)
         (not (eq? node to)))
       (lambda (from to)
         (vector-set! paths to (cons to (vector-ref paths from))))
       edge-nop
       (list from))
  (vector-ref paths to))
```

kortste: c.f. afstandsberekening

```
(define (exists-path? g from to)
  (define encountered #f)
  (dft g
       root-nop
       (lambda (node)
         (if (eq? node to)
             (set! encountered #t))
         (not encountered))
       node-nop
       edge-nop
       edge-nop
       edge-nop
       (list from))
  encountered)
```

Traversals vertrekken bij from

Drie Groepen Algoritmen

Basistoepassingen (BFT/DFT)

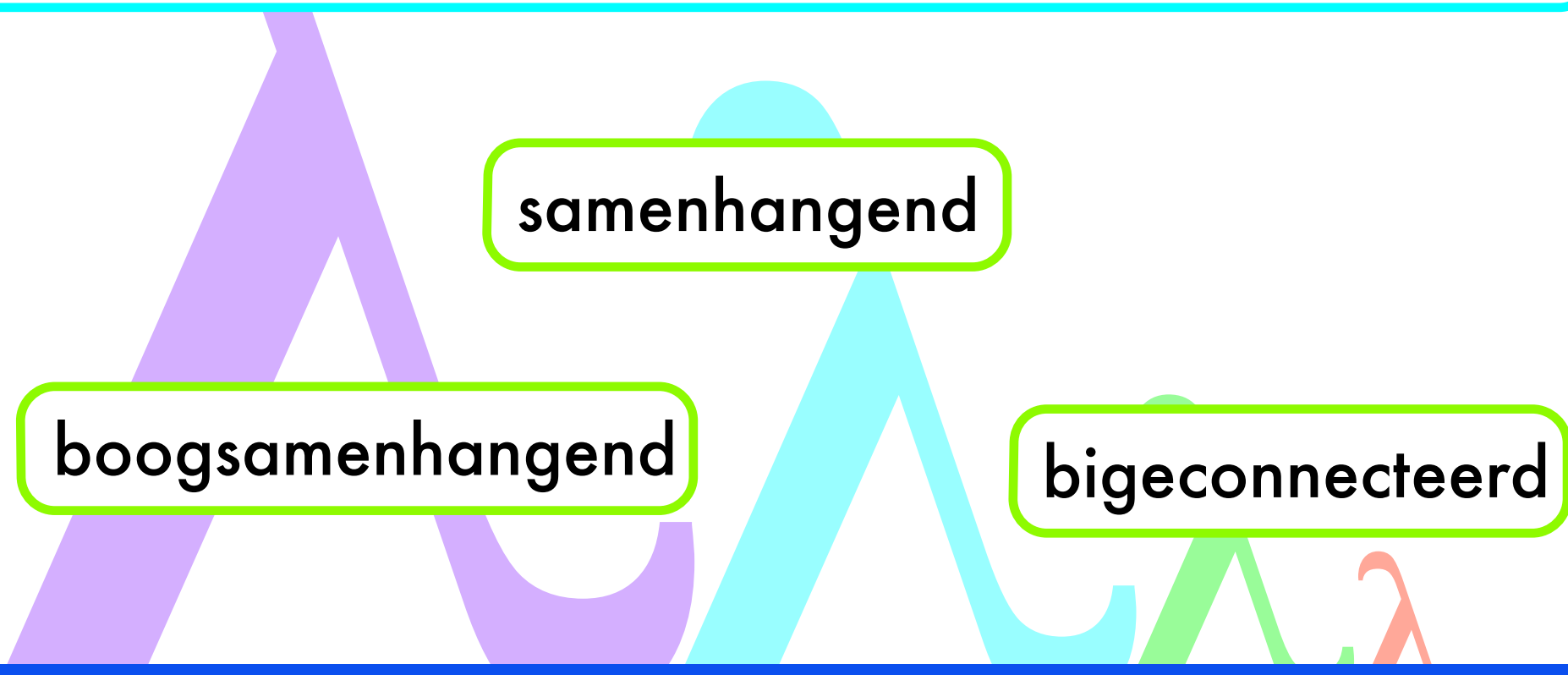
Connectiviteit (BFT/DFT)

Minimale Spanningbossen

samenhangend

boogsamenhangend

bigeconnecteerd

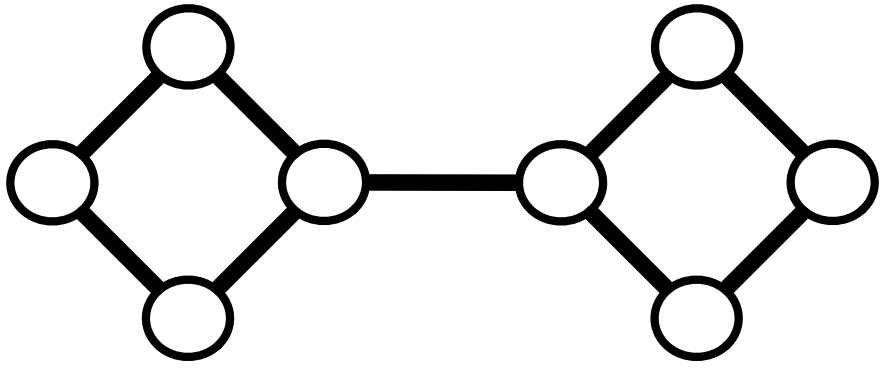


Ter herinnering

We noemen een ongerichte graaf samenhangend of geconnecteerd indien er een pad bestaat tussen elk koppel knopen. Een graaf die niet geconnecteerd is, bestaat uit geconnecteerde componenten. Dit zijn maximale deelgrafen die wél geconnecteerd zijn

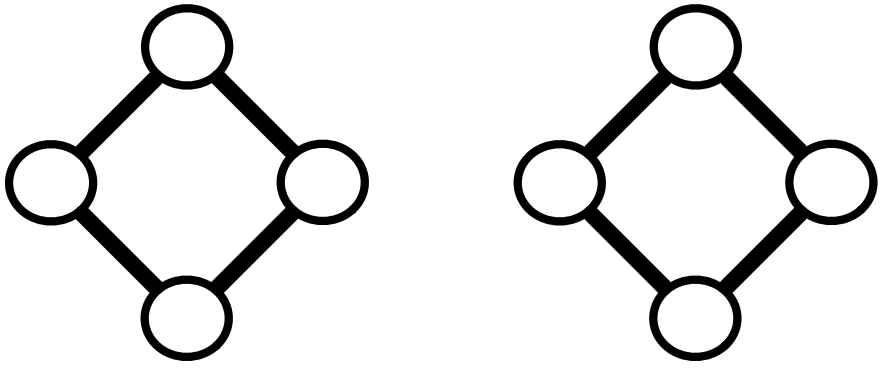
Voorbeelden

samenhangend



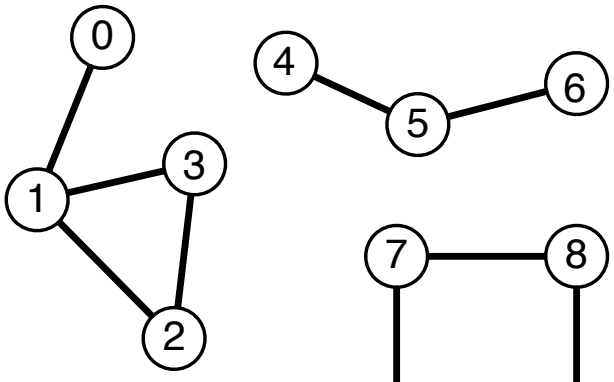
#geconnecteerde componenten = 1

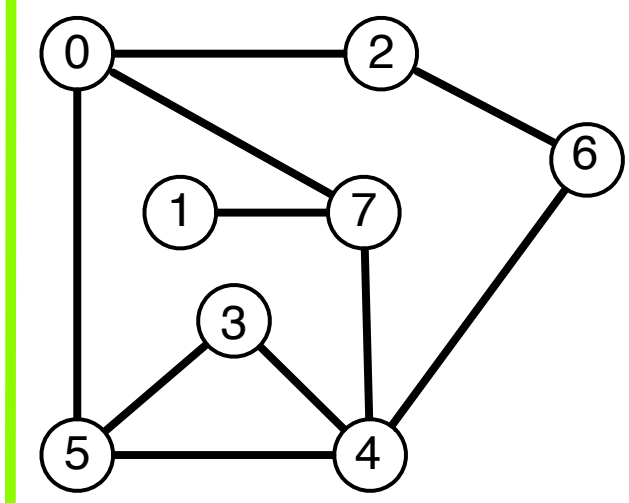
niet samenhangend

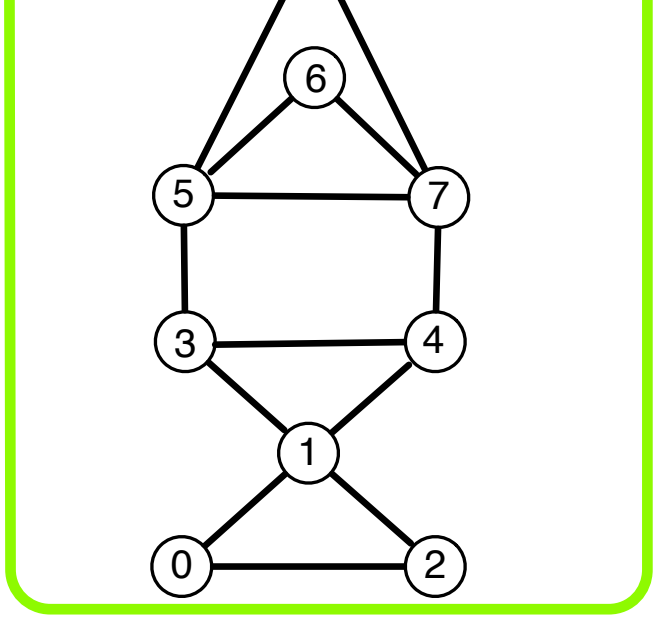


#geconnecteerde componenten = 2

Algoritmen: Voorbeelden








```
> (connected-components/dft three-cc)
{3 . #(1 1 1 1 2 2 2 3 3 3 3)}
```

```
> (connected-components/bft connected)
{1 . #(1 1 1 1 1 1 1 1 1)}
```

```
> (connected-components/dft kite)
{1 . #(1 1 1 1 1 1 1 1 1)}
```

Samenhangendheid

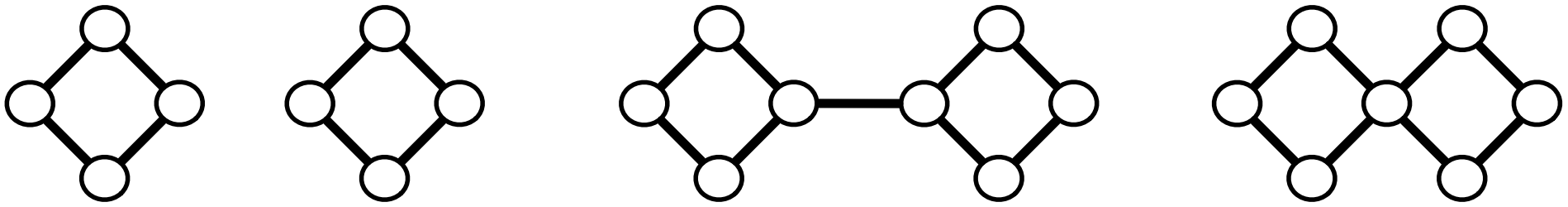
```
(define (connected-components/dft g)
  (define number-of-components 0)
  (define connected-components (make-vector (order g) '()))
  (dft g
       (lambda (root)
         (set! number-of-components (+ number-of-components 1)))
       (lambda (node)
         (vector-set! connected-components node number-of-components))
       node-nop
       edge-nop
       edge-nop
       edge-nop)
  (cons number-of-components connected-components))
```

recursief

```
(define (connected-components/bft g)
  (define number-of-components 0)
  (define connected-components (make-vector (order g) '()))
  (bft g
       (lambda (root)
         (set! number-of-components (+ number-of-components 1)))
       (lambda (node)
         (vector-set! connected-components node number-of-components))
       edge-nop
       edge-nop)
  (cons number-of-components connected-components))
```

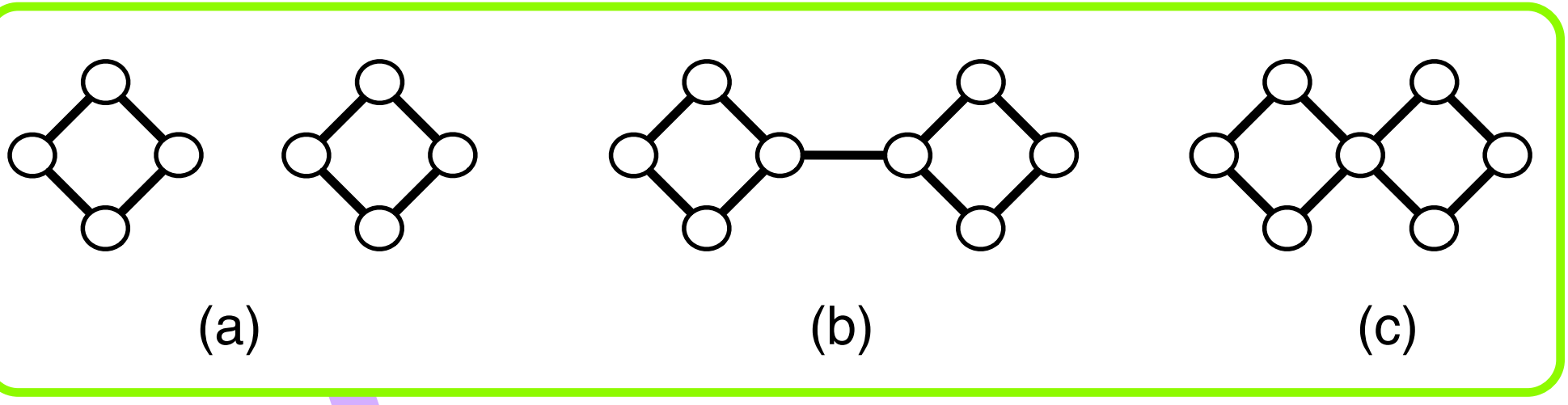
iteratief

Sterkere vormen van connectiviteit



(a) is niet geconnecteerd. (b) en (c) wel.
Voor sommige toepassingen zijn (b) en (c) echter niet "voldoende sterk" geconnecteerd.

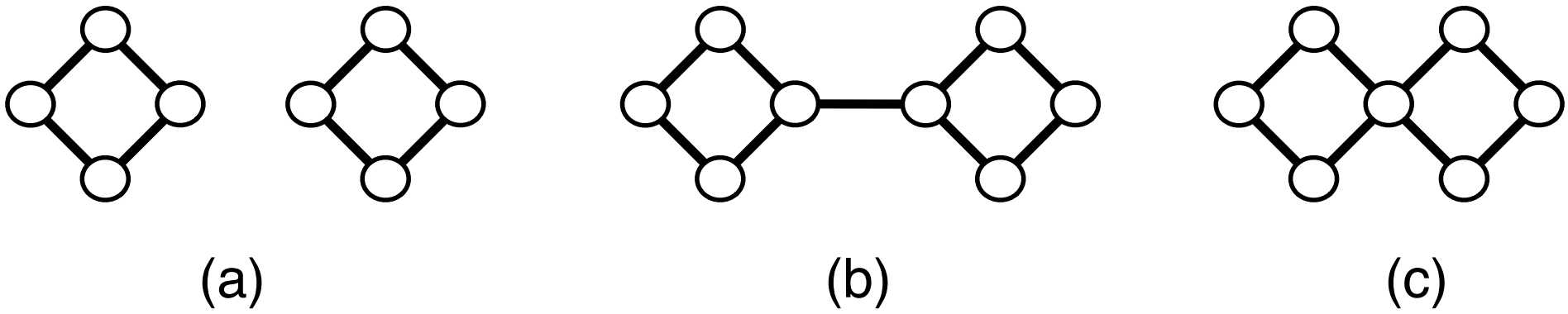
Sterkere vormen van connectiviteit



Een brug is een boog die een samenhangende ongerichte graaf in twee geconnecteerde componenten verdeelt indien hij verwijderd wordt uit E. Een graaf zonder bruggen heet boogsamenhangend (Eng: edge-connected)

Enkel (c) is boogsamenhangend

Sterkere vormen van connectiviteit



Een <u>scharnierpunt</u> of <u>articulatiepunt</u> is een knoop die een samenhangende graaf in twee of meerdere geconnecteerde componenten verdeelt indien hij verwijderd wordt (samen met bogen die eraan vast hangen). Een ongerichte graaf zonder articulatiepunten heet <u>bigeconnecteerd</u> (Eng: biconnected).

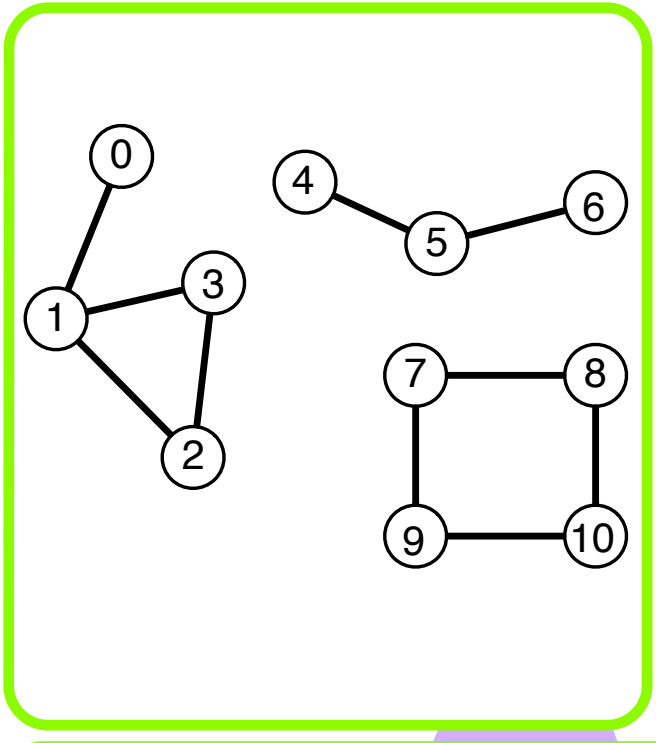
(a) noch (b) noch (c) is bigeconnecteerd

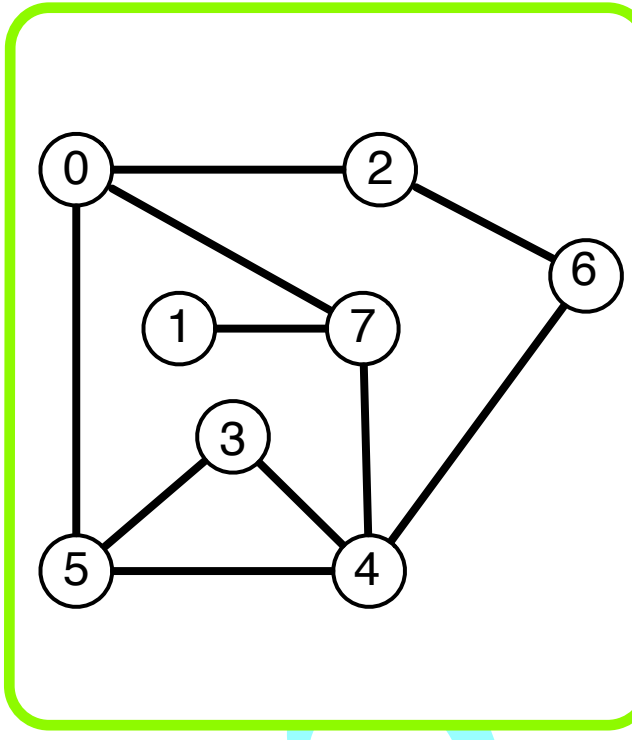
Toepassingen

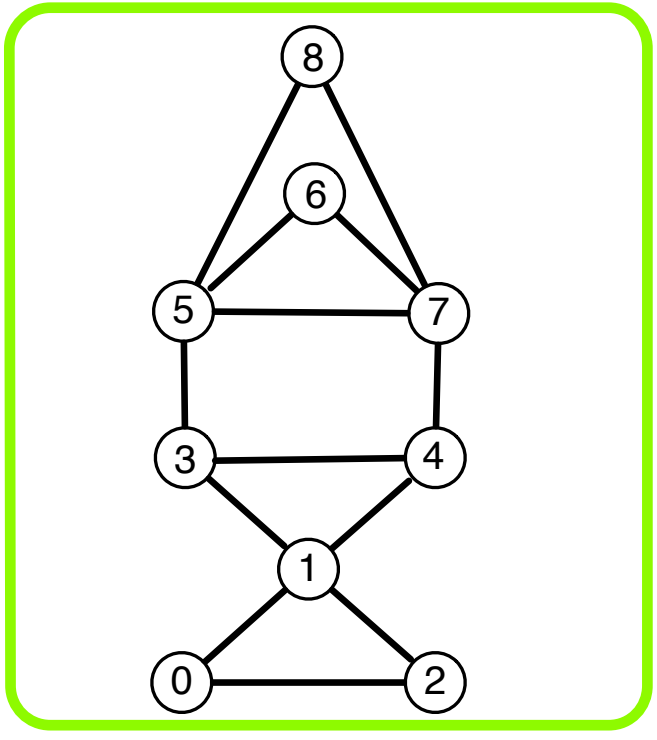
Luchtvaartmaatschappijen willen van sommige luchthavens geen articulatiepunten maken.

Vermijden van bruggen in het aanleggen van een skigebied of in het aanleggen van een (militair) communicatienetwerk.

Boogsamenhangend: Voorbeelden





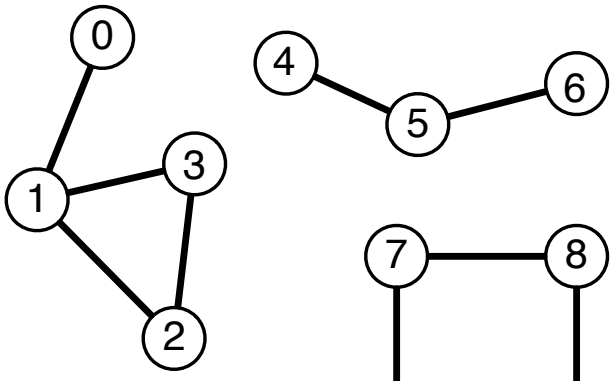


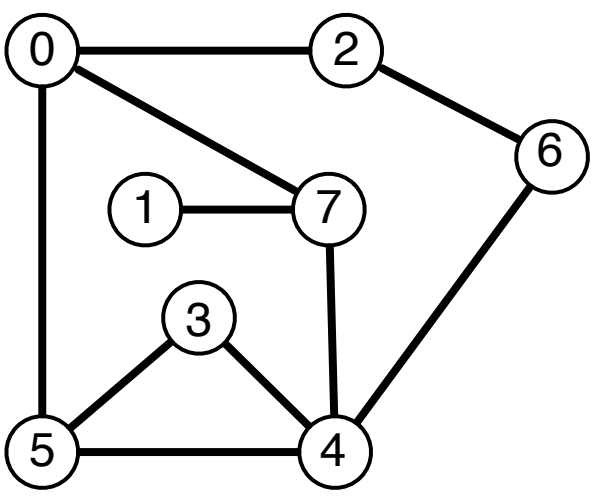

```
> (edge-connected-components three-cc)
{{4 . 5} {5 . 6} {0 . 1}}
```

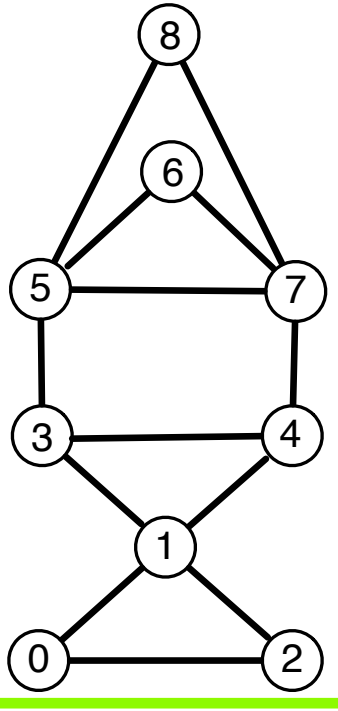
```
> (edge-connected-components connected)
{{7 . 1}}
```

```
> (edge-connected-components kite)
{}
```

Bigeconnecteerd: Voorbeelden








```
> (biconnected-components three-cc)
#(#f #t #f #f #f #t #f #f #f #f #f)
```

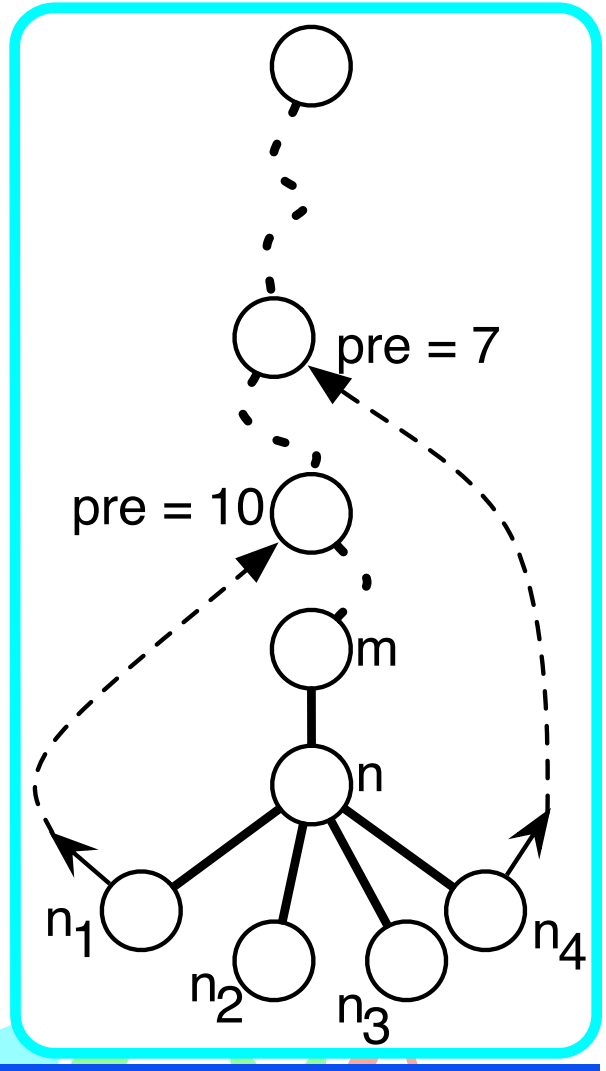
```
> (biconnected-components connected)
#(#f #f #f #f #f #f #f #t)
```

```
> (biconnected-components kite)
#(#f #t #f #f #f #f #f #f #f)
```

Edge/Bi-connected

Bepaal de kleinste preordenummers die je vanuit iedere knoop kan bereiken

```
(define preorder-time 0)
(define preorder-numbers (make-vector (order g) 0))
(define highest-back-edge (make-vector (order g) 0))
(dft g
     ...)
```



Neem het minimum van de highest-back-edge van alle processed nakomelingen

Neem het minimum met preorde nummers van alle bumped bogen

Hopcroft&Tarjan (1973)

Basisprincipe Hopcroft&Tarjan

Neem het minimum van de highest-back-edge van alle processed bogen

```
(dft g
     root-nop
     (lambda (node)
       (vector-set! preorder-numbers node preorder-time)
       (vector-set! highest-back-edge node preorder-time)
       (set! preorder-time (+ preorder-time 1)))
     node-nop
     (lambda (from to)
       (vector-set! parents to from))
     (lambda (from to)
       (vector-set! highest-back-edge
                    from (min (vector-ref highest-back-edge from)
                              (vector-ref highest-back-edge to)))
     (lambda (from to)
       (if (not (eq? (vector-ref parents from) to))
           (vector-set! highest-back-edge
                        from (min (vector-ref highest-back-edge from)
                                  (vector-ref preorder-numbers to))))))
```

En het minimum met preorde nummers van alle bumped bogen

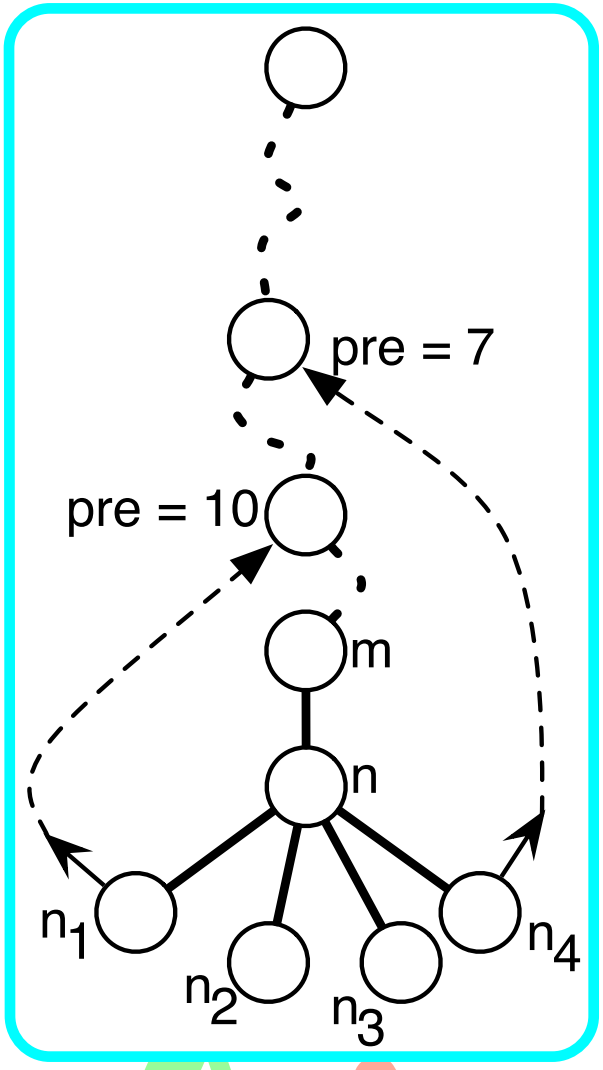
∀ moment: ∀n:

highest-back-edge[n] ≤ preorder-numbers[n]

Boogsamenhangendheid

```
(define (edge-connected-components g)
  (define preorder-time 0)
  (define preorder-numbers (make-vector (order g) 0))
  (define parents (make-vector (order g) '()))
  (define highest-back-edge (make-vector (order g) 0))
  (define bridges '())
  (dft g
       ...)
  bridges)
```

Als je van knoop n niet naar een knoop met kleiner preordenummer kunt geraken dan dat van n, dan is de boog van m naar n een brug.



Boogsamenhangendheid

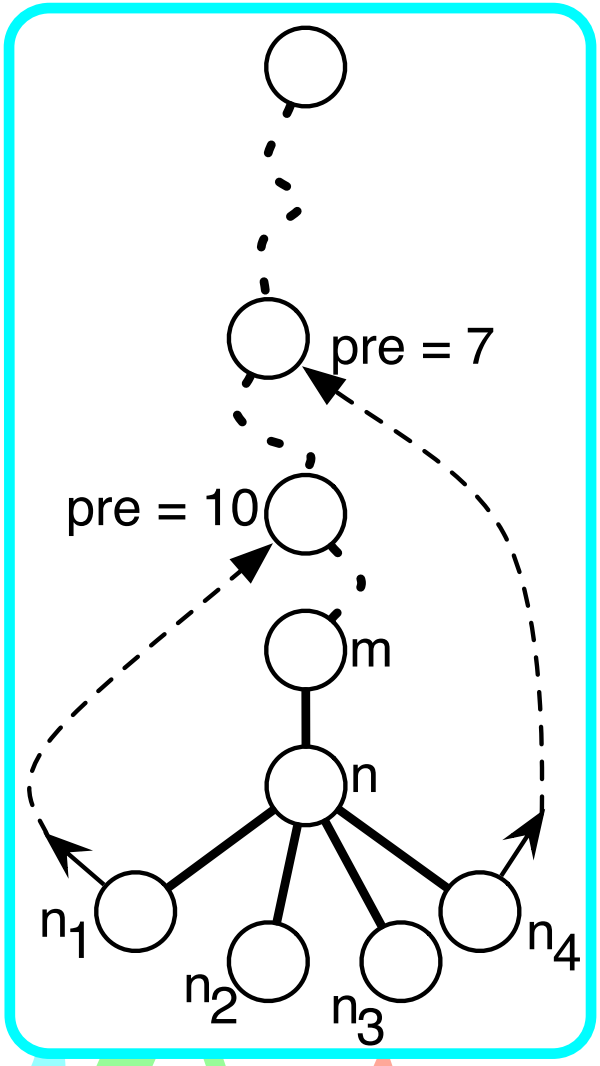
```
(dft g
     root-nop
     (lambda (node)
       (vector-set! preorder-numbers node preorder-time)
       (vector-set! highest-back-edge node preorder-time)
       (set! preorder-time (+ preorder-time 1)))
     node-nop
     (lambda (from to)
       (vector-set! parents to from))
     (lambda (from to)
       (vector-set! highest-back-edge
                    from (min (vector-ref highest-back-edge from)
                              (vector-ref highest-back-edge to)))
       (when (= (vector-ref preorder-numbers to)
                (vector-ref highest-back-edge to))
         (set! bridges (cons (cons from to) bridges))))
     (lambda (from to)
       (if (not (eq? (vector-ref parents from) to))
           (vector-set! highest-back-edge
                        from (min (vector-ref highest-back-edge from)
                                  (vector-ref preorder-numbers to))))))
```

Test

Bigeconnecteerdheid

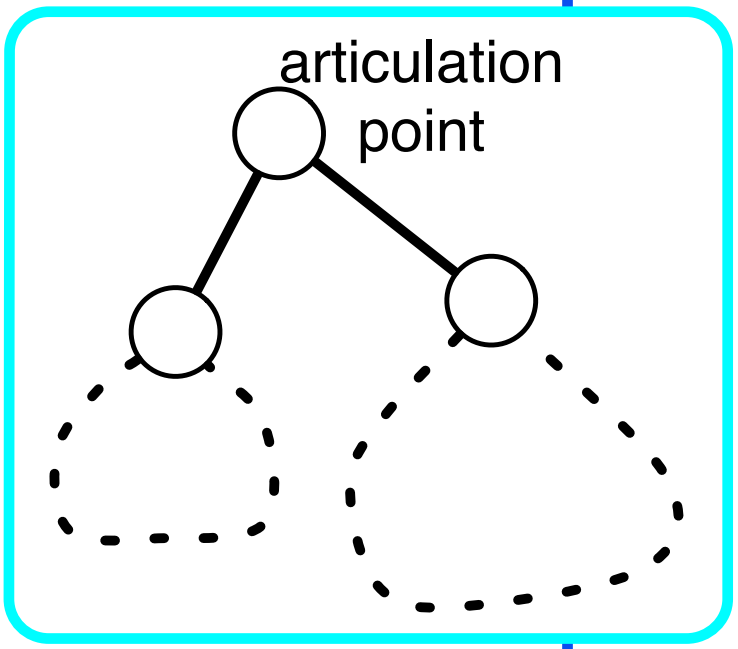
```
(define (biconnected-components g)
  (define preorder-time 0)
  (define preorder-numbers (make-vector (order g) 0))
  (define parents (make-vector (order g) '()))
  (define highest-back-edge (make-vector (order g) -1))
  (define articulation-points (make-vector (order g) #f))
  ...
  (dft g
       ...)
  articulation-points)
```

Als je vanuit n niet hoger geraakt dan n zélf, is n een scharnierpunt



Pathologaisch Geval

```
(define (biconnected-components g)
  ...
  (define current-root '())
  (define branch-count 0)
  (define (root-branch? node)
    (if (= node current-root)
        (set! branch-count (+ 1 branch-count))))
  (define (root-node! node)
    (set! branch-count 0)
    (set! current-root node))
  (define (root-node? node)
    (= node current-root))
  (dft g
       root-node!
       (lambda (node)
         ...)
       (lambda (node)
         (if (root-node? node)
             (vector-set! articulation-points node (>= branch-count 2))))
       (lambda (from to)
         (root-branch? from)
         (vector-set! parents to from))
       (lambda (from to)
         ...)
       (lambda (from to)
         ...))
  articulation-points)
```



branching factor ≥ 2

Bigeconnecteerdheid

```
(dft g
     root-node!
     (lambda (node)
       (vector-set! preorder-numbers node preorder-time)
       (vector-set! highest-back-edge node preorder-time)
       (set! preorder-time (+ preorder-time 1)))
     (lambda (node)
       (if (root-node? node)
           (vector-set! articulation-points node (>= branch-count 2))))
     (lambda (from to)
       (root-branch? from)
       (vector-set! parents to from))
     (lambda (from to)
       (vector-set! highest-back-edge
                    from (min (vector-ref highest-back-edge from)
                              (vector-ref highest-back-edge to)))
       (if (>= (vector-ref highest-back-edge to)
               (vector-ref preorder-numbers from))
           (vector-set! articulation-points from #t)))
     (lambda (from to)
       (if (not (eq? (vector-ref parents from) to))
           (vector-set! highest-back-edge
                        from (min (vector-ref highest-back-edge from)
                                  (vector-ref preorder-numbers to))))))
articulation-points)
```

Tests

Drie Groepen Algoritmen

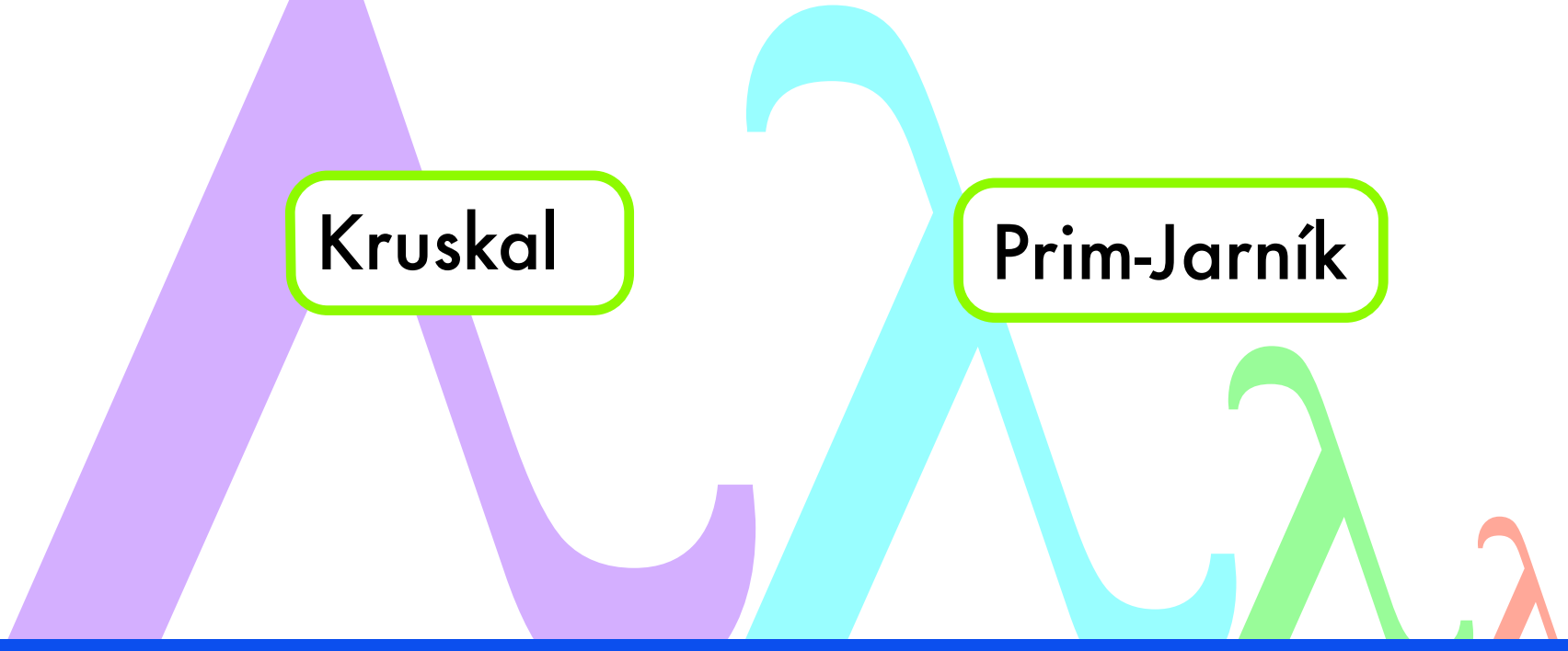
Basistoepassingen (BFT/DFT)

Connectiviteit (BFT/DFT)

Minimale Spanningbossen

Kruskal

Prim-Jarník



Ter Herinnering

Een graaf G = (V,E) heet gewogen indien er een functie $w_G : E \rightarrow \mathbb{R}$ bestaat die met iedere boog een gewicht associeert.

Zij G = (V,E) een gewogen graaf met gewichtsfunctie w_G. Het gewicht van de graaf $w(G) = \sum_{e \in E} w_G(e)$

Minimale Spanningbossen/bomen

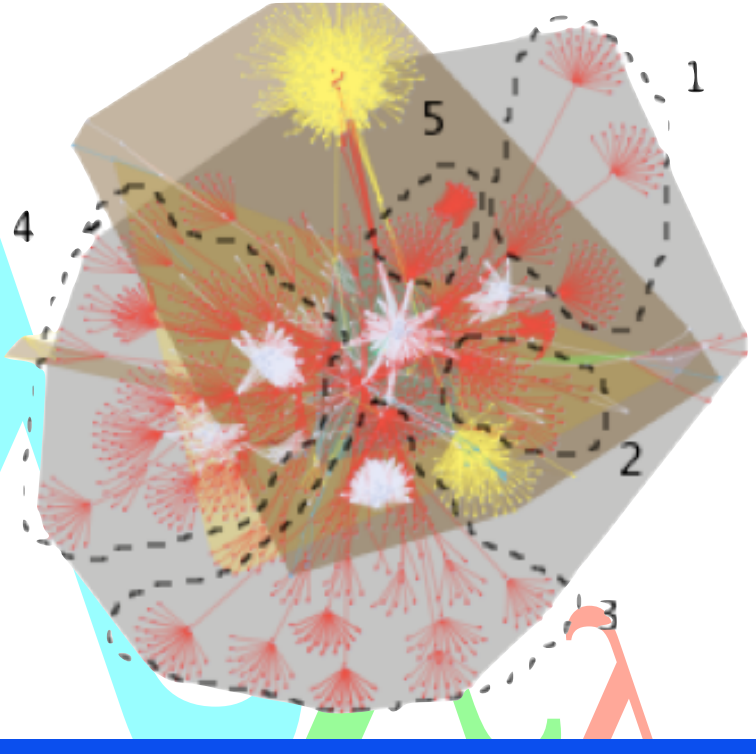
Zij $G = (V,E)$ een ongerichte gewogen graaf met gewichtsfunctie w_G. Een minimaal spanningbos voor G is een spanningbos $B = (V,E_B)$ zodat voor elk ander spanningbos $B' = (V,E'_B)$ van G geldt: $w(B) \leq w(B')$.

Toepassing op "wiring problemen". Voorbeeld: alle computers verbinden met een bus-netwerk met zo weinig mogelijk kabel.

Voor het eerst geformuleerd door Borůvka voor het ontwerpen van electriciteits-netwerk in Moravië

Toepassing: Clusteranalyse

Vertrek met objecten o_1, o_2, ... o_n en met een afstandsrelatie tussen die objecten. Gegeven k. Gevraagd de k sterclusters van dichtsbijzijnde objecten.



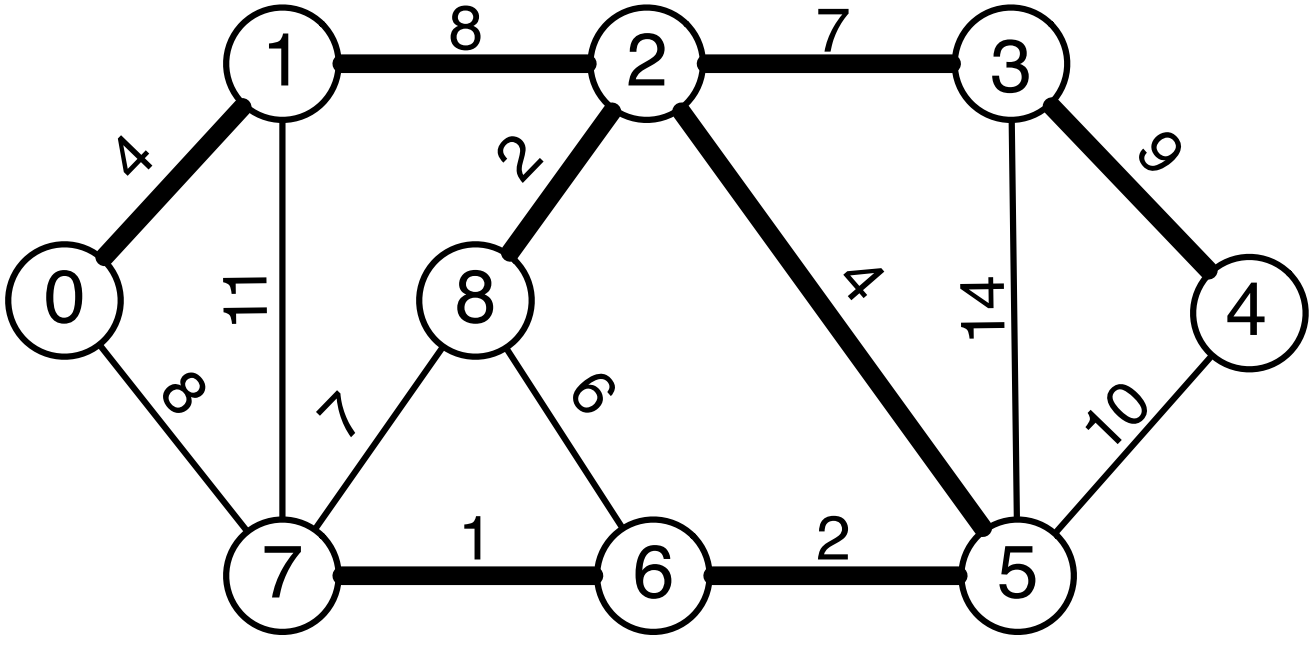
We bestuderen 2 algoritmen

Kruskal (1956)

Prim (1957) (Jarník 1930)

<u>Basisidee</u>: vertrek met een leeg bos en voeg in een iteratie telkens een (of meerdere) “veilige” bo(o)g(en) toe.

Het Kruskal Algoritme: Voorbeeld




```
> (mst-kruskal cormen571)
{{3 9 4} {1 8 2} {2 7 3} {0 4 1} {2 4 5} {2 2 8} {5 2 6} {6 1 7}}
```

Een Hulpstuk

```
(define (all-edges g)
  (define n-e (nr-of-edges g))
  (define edges (make-vector n-e))
  (define edge-count 0)
  (for-each-node
   g
   (lambda (from)
     (for-each-edge
      g
      from
      (lambda (weight to)
        (when (<= from to)
          (vector-set! edges edge-count (cons from (cons weight to)))
          (set! edge-count (+ 1 edge-count))))))))
  edges)
```

Let op: een boog wordt twee keer bezocht

Het Algoritme van Kruskal

Sorteer alle bogen volgens gewicht.
Vertrek van een leeg bos.

Beschouwen alle bogen van klein naar groot gewicht.

Indien een boog een cyclus veroorzaakt gooien we hem weg. Anders voegen we hem toe aan het bos.

union! find

Het Algoritme van Kruskal

```
(define (mst-kruskal g)
  (define edges (all-edges g))
  (define n-e (vector-length edges))
  (define forest '())
  (define node-sets (dset:new (order g)))
  (quicksort edges (lambda (edge1 edge2) (< (cadr edge1) (cadr edge2))))
  (do ((edge-idx 0 (+ edge-idx 1)))
    ((= edge-idx n-e) forest)
    (let* ((edge (vector-ref edges edge-idx))
           (from (car edge))
           (weight (cadr edge))
           (to (cddr edge))
           (from-set (dset:find node-sets from))
           (to-set (dset:find node-sets to)))
      (when (not (dset:same-set? from-set to-set))
        (set! forest (cons edge forest))
        (dset:union! node-sets from-set to-set)))))
```

$O(f_{\texttt{all-edges}}+|E|.\log(|E|)+|E|.\alpha(|V|)) = O(|E|.\log(|V|))$

met Radix Sort: $|E|.\alpha(|V|))$

Het Algoritme van Prim-Jarník

Vertrek van een leeg "bos tot nu toe". Alle knopen van de graaf zijn +∞ verwijderd van dat bos.

Hou een PQ van "nog niet opgespannen knopen" bij, samen met hun afstand tot het "bos tot nu toe".

Serve een knoop uit die PQ en voeg hem toe aan het "bos tot nu toe". Beschouw al zijn buren. Enqueue buren die nog niet in de PQ zitten en niet in het "bos tot nu toe" zitten. Overschrijf de prioriteit van een buur in de PQ als die zich dichter bij het "bos tot nu toe" bevindt.

Ter Herinnering

```
ADT priority-queue< P >

new
   ( ( P P → boolean ) → priority-queue< P > )
priority-queue?
   ( any → boolean )
enqueue!
   ( priority-queue<P> any P → priority-queue< P > )
peek
   ( priority-queue< P > → any )
serve!
   ( priority-queue< P > → any )
full?
   ( priority-queue< P > → boolean )
empty?
   ( priority-queue< P > → boolean )
```

Algoritmen & Datastructuren 1

Maar...

Vertrek van een leeg "bos tot nu toe". Alle knopen van de graf zijn 'infinity verwijderd van dat bos.

Hou een PQ van "nog ... opgespannen knopen" bij, samen met hun afstand tot het "bos

O(1) ?

O(log(n)) ?

Serve een knoop uit die PQ en voeg hem toe aan het "bos tot nu toe". Beschouw al zijn buren. Enqueue buren die nog niet in de PQ zitten en niet in het "bos tot nu toe" zitten. Overschrijf de prioriteit van een buur in de PQ als die zich dichter bij het "bos tot nu toe" bevindt.

Een niet-triviale variant

PQ-operaties registreren verplaatsingen van elementen

```
ADT priority-queue< P >

new
 ( ( P P → boolean ) → priority-queue< P > )
priority-queue?
 ( any → boolean )
enqueue!
 ( priority-queue<P> any P (number any P → ∅) → priority-queue< P > )
peek
 ( priority-queue< P > → any )
serve!
 ( priority-queue< P > (number any P → ∅) → ( any P ) )
full?
 ( priority-queue< P > → boolean )
empty?
 ( priority-queue< P > → boolean )
priority-of
 ( priority-queue< P > number → P )
reschedule!
 ( priority-queue<P> number P (number any P → ∅) → priority-queue< P> )
```

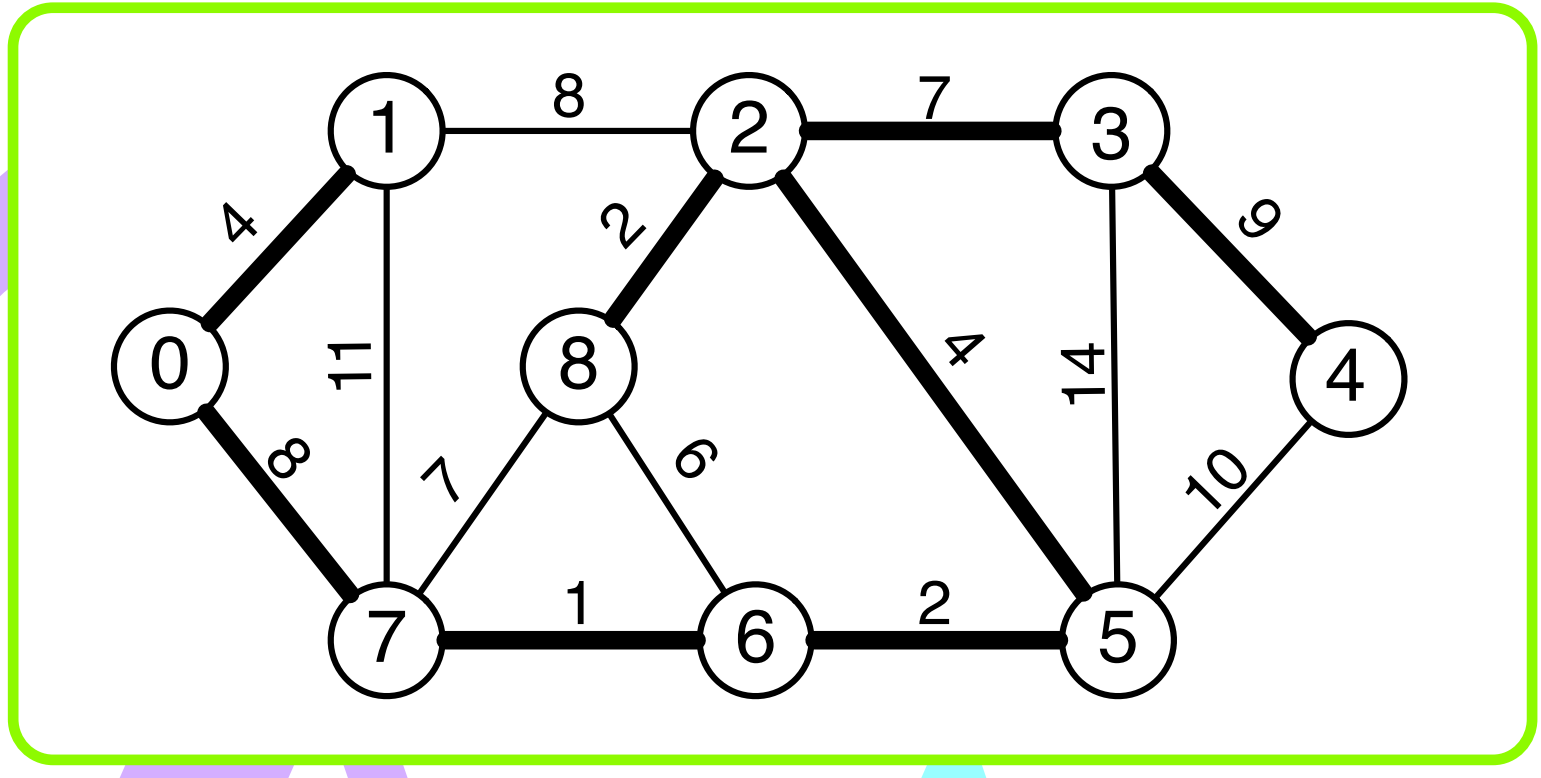
Implementatie: Smaakmakertje

```
(define (sift-up heap idx notify)
  (let
      ((vector-ref
        (lambda (v i)
          (vector-ref v (- i 1))))
       (vector-set!
        (lambda (v i a)
          (vector-set! v (- i 1) a)
          (notify i a)))
       (vector (storage heap))
       (size (length heap))
       (<<? (lesser heap)))
    (let sift-iter
      ((child idx)
       (element (vector-ref vector idx)))
      (let ((parent (div child 2)))
        (cond ((= parent 0)
               (vector-set! vector child element))
              ((<<? element (vector-ref vector parent))
               (vector-set! vector child (vector-ref vector parent))
               (sift-iter parent element))
              (else
               (vector-set
```

Registratie

c.f. Algoritmen en Datastructuren 1

Prim-Jarník: Voorbeeld




```
> (mst-prim cormen571)
#({() . +inf.0} {0 . 4} {5 . 4} {2 . 7}
  {3 . 9} {6 . 2} {7 . 1} {0 . 8} {2 . 2})
```

Het Algoritme van Prim-Jarník

```
(define (mst-prim-jarnik g)
  (define tree (make-vector (order g) '()))
  (define pq-ids (make-vector (order g) '()))
  (define (track-ids id node weight)
    (vector-set! pq-ids node id))
  (define (id-of node)
    (vector-ref pq-ids node))
  (define (pty< edge1 edge2)
    (< (cdr edge1) (cdr edge2)))
  (define pq (pq:new (order g) pty<))
  (define (prim-jarnik-iter closest-node&edge)
    ...
    (unless (pq:empty? pq)
      (prim-jarnik-iter (pq:serve! pq track-ids))))
  (for-each-node g (lambda (node)
                     (when (null? (vector-ref tree node))
                       (pq:enqueue! pq node (cons '() +inf.0) track-ids)
                       (prim-jarnik-iter (pq:serve! pq track-ids)))))
  tree)
```

Het Algoritme van Prim-Jarník

$O((|E|+|V|).\log(|V|))$

$= O(|E|.\log(|V|)$

```
(define (prim-jarnik-iter closest-node&edge)
  (define closest-node (car closest-node&edge))
  (define closest-edge (cdr closest-node&edge))
  (vector-set! tree closest-node closest-edge)
  (for-each-edge
   g
   closest-node
   (lambda (weight to)
     (define edge-to-to (cons closest-node weight))
     (if (null? (id-of to))
         (pq:enqueue! pq to edge-to-to track-ids)
         (if (and (null? (vector-ref tree to))
                  (pty< edge-to-to (pq:priority-of pq (id-of to))))
             (pq:reschedule! pq (id-of to) edge-to-to track-ids)))))
  (unless (pq:empty? pq)
    (prim-jarnik-iter (pq:serve! pq track-ids))))
```

Samenvatting

Kruskal	O(|E|.log(|V|) of: O(|E|.α(|V|)))
Prim-Jarník	O(|E|.log(|V|)

Hoofdstuk 15

15.1 Basisalgoritmen
- 15.1.1 Cycliciteitstest
- 15.1.2 Paden
- 15.1.3 Afstand tussen knopen

15.2 Connectiviteit
- 15.2.1 Samenhangendheid
- 15.2.2 Boogsamenhangendheid
- 15.2.3 Bigeconnecteerdheid
- 15.2.4 Bipartiteheid

15.3 Minimum spanningsbossen
- 15.3.1 Kruskal
- 15.3.2 Prim-Jarník