De decompositie van een gerichte graaf in zijn sterke componenten

De decompositie van een gerichte graaf in zijn sterke componenten. Een theoretisch meest efficient algorithme.

door

ir. J.A.G.M. Kerbosch

J.C. Wortmann

januari 1973

rapport KW 1

Abstract

This paper describes a theoretically most efficient algorithm, that decomposes a directed graph into its strongly-connected components. A formal proof of the algorithm is given. The computational time is shown to be proportional to the number of arcs in the graph.
A translation of this report in English is also available.

Samenvatting

Dit rapport geeft een beschrijving van een theoretisch meest efficient algorithme, dat een gerichte graaf splitst in sterk-samenhangende componenten. Er wordt een formeel bewijs van het algorithme gegeven. De rekentijd is evenredig met het aantal pijlen in de graaf.
Dit rapport is ook in het engels vertaald.

Voorwoord

Onze dank komt zeer velen toe. Het is echter weinig zinvol, hen allen te noemen, te meer daar vele invloeden indirect op dit project hun inwerking hadden.
Wij danken de heer H. van der Heyden en mej. Annelies Ummels voor de metamorfose, die zij tot stand brachten, en die een verzameling vodjes transformeerde tot dit rapport.

Inhoud

1. Inleiding

Het verkrijgen van inzicht in de structuur van een graaf, is een probleem bij veel toepassingen van de grafentheorie. Eén van de methoden, om dit probleem te lijf te gaan, is de decompositie van de graaf in een aantal componenten. Bij ongerichte grafen bijvoorbeeld, is decompositie in "samenhangende componenten", een voor de hand liggende en veel gebruikte methode. Bij gerichte grafen kan men nog een stap verder gaan, dankzij het begrip: sterke component.

Een *sterke component (SC)* van een graaf G, is een deelgraaf D⊂G, met de eigenschap, dat ieder knooppunt vanuit ieder ander knooppunt bereikbaar is; bovendien moet de deelgraaf *niet-uitbreidbaar* zijn, d.w.z. als men een knooppunt V aan D toevoegt, geldt voor de nieuwe deelgraaf niet meer, dat ieder knooppunt vanuit ieder ander knooppunt bereikbaar is.

Voorbeeld:

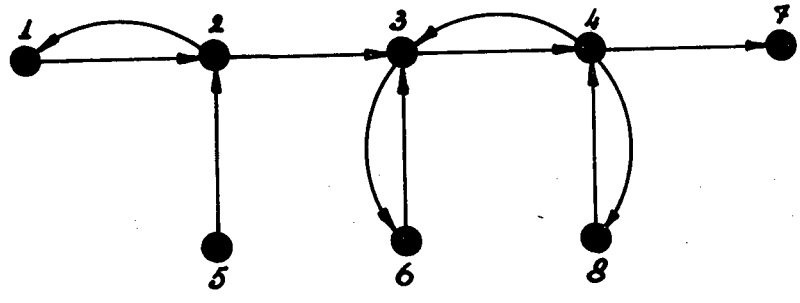


fig. 1

De sterke componenten van de graaf, getekend in fig. 1, zijn weergegeven in fig. 2. Merk op, dat de decompositie éénduidig bepaald is.

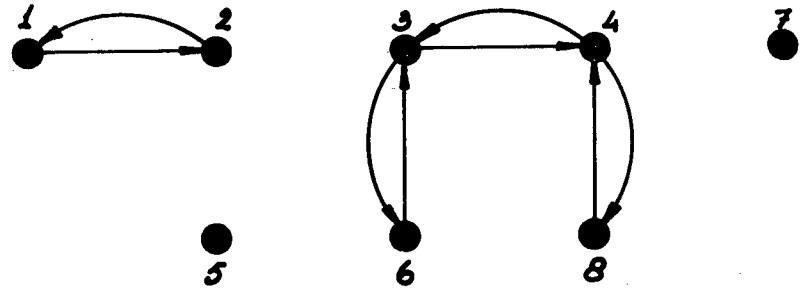


fig. 2

In dit rapport wordt een theoretisch meest efficient algorithme gegeven, om de sterke componenten van een gerichte graaf te bepalen. De rekentijd is evenredig met het aantal pijlen in de graaf.
In de terminologie van de grafentheorie heerst een babylonische spraakverwarring. Dit feit dwong ons, alle gebruikte begrippen te definieren. Deze definities kan men vinden in Appendix A.
In hoofdstuk 2 worden enkele begrippen en notaties nader toegelicht. In hoofdstuk 3 wordt behandeld, op welke manier de graaf in een computer wordt weergegeven (Datastructuur). Hoofdstuk 4 behandelt het algorithme SC-Worker. Paragraaf 4.1. geeft een handalgorithme dat bedoeld is om de lezer het algorithme duidelijk te maken; paragraaf 4.2. geeft een voorbeeld, waarin de stappen van het algorithme zijn aangegeven. Paragraaf 4.4. geeft een Algol-programma, waarin gebruik is gemaakt van recursie [1]. In paragraaf 4.3. wordt toelichting op dit programma gegeven. Appendix B geeft een bewijs van correctheid.
De efficiency van het algorithme komt ter sprake in hoofdstuk 5; het wordt vergeleken met enkele algorithmen, die ons uit de literatuur bekend zijn. Tenslotte geeft hoofdstuk 6 enkele toepassingsgebieden.

2. *Toelichting van enkele begrippen en notaties*

Als p een pijl is, wordt met b(p) het beginpunt van p bedoeld, en met e(p) het eindpunt.

Een knooppunt V heet *bereikbaar* vanuit een knooppunt W, als er een pad loopt van W naar V of als V = W. Dus V is per definitie altijd bereikbaar vanuit zichzelf.

Een knooppunt V is *sterk verbonden* met een knooppunt W, als V vanuit W *en* W vanuit V bereikbaar is.

De relaties: "bereikbaar vanuit" en "sterk verbonden met" zijn *transitief*, d.w.z.

z bereikbaar vanuit y
y bereikbaar vanuit x } → z bereikbaar vanuit x

Een meer formele behandeling van begrippen is te vinden in Appendix A.

3. Datastructuur

Wij veronderstellen, dat de knooppunten van de graaf G als volgt zijn genummerd:1,....,n. De graaf G wordt weergegeven in de computer door de volgende drie ééndimensionale arrays:

- suc (successors)
- fe (place of First outward Edge in "suc")
- le (place of Last outward Edge in "suc")

In het array suc zijn de directe opvolgers van knooppunt "i" genoteerd op de plaatsen fe[i] tot en met le[i]. Indien knooppunt i geen opvolgers heeft, geldt le[i] < fe[i].

Schema:

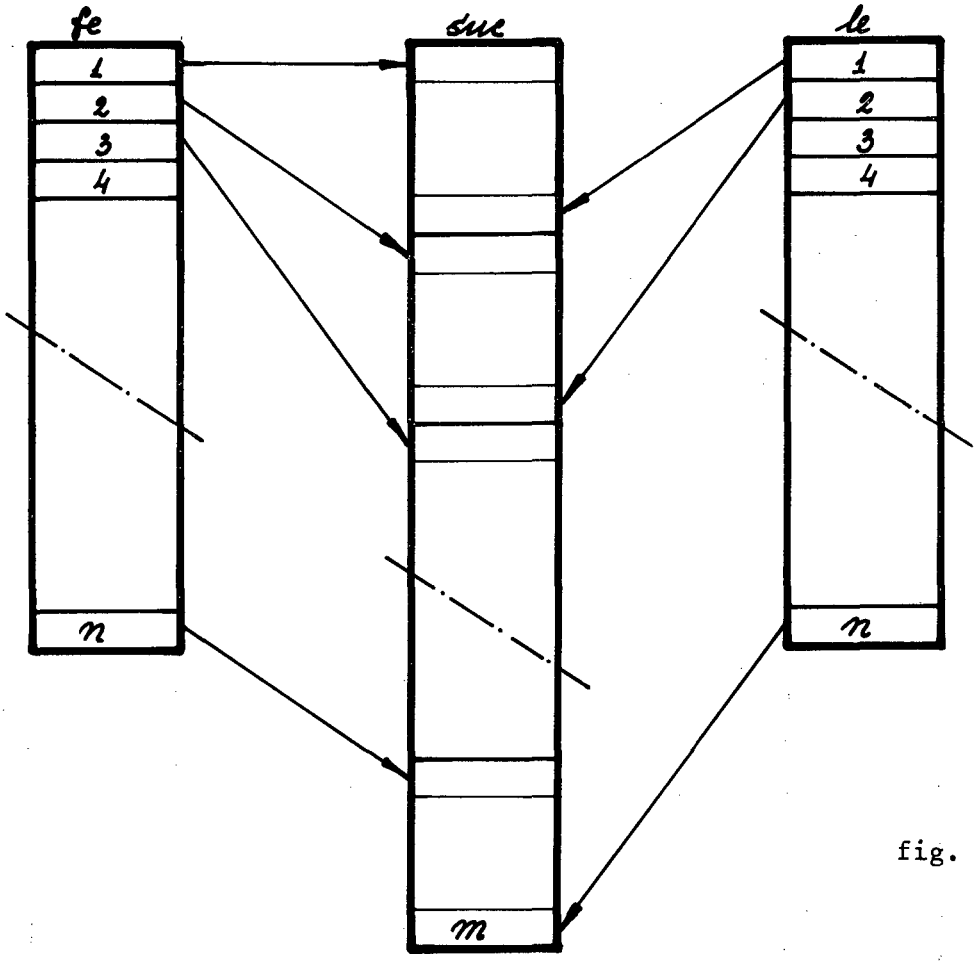


fig. 3

Een andere, veel gebruikte datastructuur is de zgn. "adjacency matrix", A. Dit is een n x n - matrix, waarin element A [i,j] gelijk is aan het aantal pijlen van knooppunt i naar knooppunt j.
Bij gebruik van de adjacency matrix zijn zelfs voor de eenvoudigste operaties, die op de gehele graaf wordt verricht, *tenminste* n^2 handelingen vereist. Daarentegen is het minimum aantal handelingen bij de bovenbeschreven datastructuur gelijk aan *m:* het aantal pijlen. Men kan nu tegenwerpen, dat *m* óók van orde n^2 is. Bij sommige toepassingen is dat inderdaad het geval; maar het is onjuist, dit als gemeen geldend aan te nemen. In de grafen, die ons bekend zijn uit OR-toepassingen, is het gemiddeld aantal uitgaande pijlen per knooppunt zelfs vrijwel nooit groter dan 6, zodat *m* van orde *n* is! In hoofdstuk 5 zal blijken, dat de rekentijd van het gepresenteerde algorithme evenredig is met *m*; in de ons bekende toepassingen betekent dit evenredig met *n*. Ook is het gebruik van geheugenruimte evenredig met *m*, en in onze toepassingen met *n*.
Anderszijds maakt dit het algorithme geenszins gecompliceerder.

4. Het algorithme SC-Worker

4.1. Beschrijving van het algorithme

Tijdens de uitvoering van het algorithme, kan men drie disjuncte deelverzamelingen van knooppunten onderscheiden:

(1) De verzameling NEW:

kenmerk : V ε NEW ↔ V is nog niet onderzocht;

implementatie: V ε NEW ↔ badge [V] = new = -1 ;

(2) De verzameling INSC:

kenmerk : V ε INSC ↔ V behoort tot een SC, die reeds is gerapporteerd;

implementatie: V ε INSC ↔ badge [V] = insc = n + 1;

(3) De geordende verzameling STACK:

kenmerk : V ε STACK ↔ V is nog in onderzoek;

implementatie: V ε STACK ↔ 1 ≤ badge [V] ≤ n ;

Het aantal elementen in STACK wordt weergegeven door de variabele "ploftop". (place of top). Het array "stack" bevat de elementen van STACK; zij staan op de plaatsen 1 t/m ploftop. Als V ε STACK, dan geeft plinstack [V] de plaats van V in STACK weer.

Wij geven nu een beschrijving van het algorithme in een 7-tal stappen. Hier en daar is commentaar gegeven, om het algorithme *plausibel* te maken; voor een *bewijs* zie men Appendix B.

Initialisering:

```
new:= -1 ; insc:= n + 1 ;
comment de grootheden "new" en "insc" zijn constanten;
badge [i] := new, i = 1,..., n ;
comment alle knooppunten behoren nu tot NEW ;
ploftop:= 0 ;
comment ploftop is het aantal elementen in STACK ;
```

Hoofdprogramma:

Stap 0: if NEW = ∅ do STOP;

Stap 1: kies een knooppunt V ε NEW;

Stap 2: breidt STACK uit met knooppunt V:

```
ploftop:= ploftop + 1;
plinstack [V] := ploftop;
comment de grootheid "plinstack[V]" blijft ongewijzigd;
badge [V] := ploftop;
comment V behoort nu tot STACK;
comment de grootheid "badge[V]" kan een lagere waarde krijgen,
        in het verdere verloop van het programma;
GEEF ALLE UITGAANDE PIJLEN VAN KNOOPPUNT V
DE MARKERING: "ONBEHANDELD";
```

Stap 3: werk de uitgaande pijlen van knooppunt V af:

```
if {ONBEHANDELDE UITGAANDE PIJLEN VAN KNOOPPUNT V} = ∅
do goto Stap 6;
KIES EEN ONBEHANDELDE UITGAANDE PIJL P VAN V;
GEEF P DE MARKERING: "IN BEHANDELING";
NOEM e(P) : "SUCCESSOR";
```

Stap 4: breidt eventueel uit:

```
if successor ε NEW do
begin V:= successor;
      comment successor vervult nu de rol van V;
      goto Stap 2
end;
```

comment Veronderstel, dat het algorithme vanuit Stap 4 inderdaad naar stap 2 is gegaan. Het knooppunt "successor", in de rol van V wordt op de stapel geplaatst. Tengevolge van vele, soortgelijke substituties in Stap 4 is het mogelijk dat nog vele opvolgers

van knooppunt successor de rol van V spelen. Maar door evenzovele omgekeerde substituties in Stap 7 krijgt knooppunt V weer de rol van V, en knooppunt successor de rol van successor, en het algorithme gaat verder bij stap 5;

Stap 5: verlaag eventueel badge[V]:

```
comment als successor ε INSC dan badge [successor] = nov + 1 > badge[V];
if badge [successor] < badge [V]
do badge [V] := badge [successor];
GEEF PIJL P DE MARKERING: "BEHANDELD";
goto Stap 3;
```

Stap 6: rapporteer eventueel een SC:

```
comment alle uitgaande pijlen van V hebben de markering: "behandeld";
if badge [V] = plinstack [V] do
begin RAPPORTEER EEN SC. DEZE BESTAAT UIT ALLE KNOOPPUNTEN ε STACK,
      VANAF PLINSTACK [V] TOT PLOFTOP;
comment voor een bewijs zie Appendix B;
HEVEL DEZE KNOOPPUNTEN OVER NAAR "INSC";
ploftop:= plinstack [V] -1
end;
```

Stap 7: Ga eventueel verder met het "vorige" knooppunt:

```
if ploftop = 0
then goto Stap 0
else begin ER IS EEN PIJL MET MARKERING: "IN BEHANDELING", WAAR-
           VAN V HET EINDPUNT IS: NOEM DEZE PIJL P;
           SUCCESSOR:= V;
           V:= b(P);
           comment Het beginpunt van pijl P vervult bij de volgende
                   stappen de rol van V;
           goto Stap 5
end;
```

4.2. Voorbeeld

Als voorbeeld gebruiken wij opnieuw
de graaf uit fig. 1, die opnieuw
is getekend in fig. 4.

De waarden, die de grootheden van het algorithme achtereenvolgens aannemen, zijn op blz. 10 weergegeven; het knooppunt, dat de rol van V speelt, is omcirkeld. Een accolade onder de knooppunten geeft aan dat deze knooppunten als SC worden gerapporteerd.

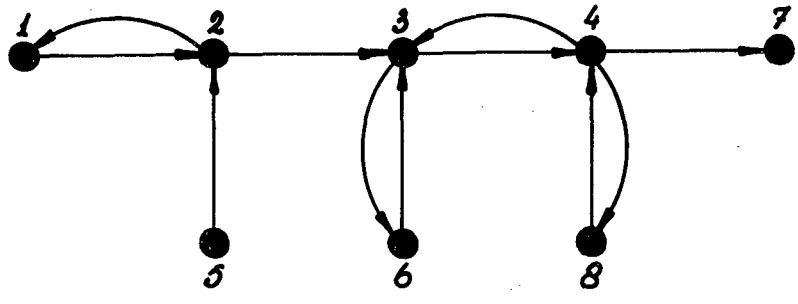


fig. 4

stap	pijl	stack [1][2][3][4][5][6][7][8]	badge [1][2][3][4][5][6][7][8]
0,1,2		(1)	1
3,4,2	1 → 2	1 (2)	1 2
3,4,5	2 → 1	1 (2)	1 1
3,4,2	2 → 3	1 2 (3)	1 1 3
3,4,2	3 → 4	1 2 3 (4)	1 1 3 4
3,4,2	4 → 8	1 2 3 4 (8)	1 1 3 4 5
3,4,5	8 → 4	1 2 3 4 (8)	1 1 3 4 4
3,6,7,5		1 2 3 (4) 8	1 1 3 4 4
3,4,2	4 → 7	1 2 3 4 8 (7)	1 1 3 4 6 4
3,6,7,5		1 2 3 (4) 8	1 1 3 4 insc 4
3,4,5	4 → 3	1 2 3 (4) 8	1 1 3 3 4
3,6,7,5		1 2 (3) 4 8	1 1 3 3 4
3,4,2	3 → 6	1 2 3 4 8 (6)	1 1 3 3 6 4
3,4,5	6 → 3	1 2 3 4 8 (6)	1 1 3 3 3 4
3,6,7,5		1 2 (3) 4 8 6	1 1 3 3 3 4
3,6,7,5		1 (2)	1 1 insc insc insc insc
3,6,7,5		(1) 2	1 1
3,6,7,0,1,2		(5)	insc insc 1
3,4,5	5 → 2	(5)	1
3,6,7,0			insc

4.3. Toelichting bij het Algolprogramma

In het programma (4.4.) speelt de recursieve procedure "extend (V)" een belangrijke rol. Een aanroep van extend (V) verzorgt de uitbreiding van STACK met V (Stap 2), het afgaan van de pijlen (Stap 3), het verlagen van badge [V] (Stap 5) en het eventueel rapporteren van een SC (Stap 6). Wanneer STACK verder moet worden uitgebreid met knooppunt "successor", (Stap 4) roept het programma "extend (successor)" aan; bij de terugkeer uit extend (successor) verzorgt het recursie-mechanisme automatisch Stap 7. De markering van de pijlen is geimplementeerd m.b.v. de lokale variabele "index", en wel als volgt:

-pijlen naar {suc[fe[V]],..., suc[index-1]} zijn "behandeld";
-de pijl naar suc[index] is "in behandeling";
-pijlen naar {suc[index+1],..., suc [le[V]]} zijn "onbehandeld".

Tenslotte is in het programma de grootheid "plinstack[V]" opgeslagen in een lokale variabele "plinstack".

4.4 Algol-programma.

```
procedure SC Worker (n , suc , fe , le); value n; integer n;
          integer array suc , fe , le;

begin integer i,new,insc,ploftop; integer array badge,stack[1:n];

      procedure extend(V); value V; integer V;
      begin integer index , successor , plinstack ;
PUT V ON STACK: badge[V]:=plinstack:=ploftop:=ploftop+1;stack[ploftop]:=V;
SCAN SUCCESSORS OF VERTEX V :
            for index:=fe[V] step 1 until le[V] do
            begin successor := suc[index] ;
                  if badge[successor] = new then extend(successor);
                  if badge[successor] < badge[V]
                  then badge[V] := badge[successor]
            end;
CHECK WHETHER SC IS FOUND :
            if badge[V] = plinstack then
            begin NEW PAGE ;
REPORT :          for index:=plinstack step 1 until ploftop do
                  begin badge[stack[index]]:= insc;
                        print(stack[index])
                  end;
                  ploftop:=plinstack - 1
            end
      end extend ;

INITIATE : new:= -1 ; insc:= n + 1 ; ploftop:= 0 ;
PUT ALL VERTICES INTO SET NEW :
      for i:=1 step 1 until n do badge[i]:=new;
RUN : for i:=1 step 1 until n do if badge[i]=new then extend(i)
end split graph into strongly-connected components ;
```

5. Efficiency

In het algorithme is het aantal aanroepen van de procedure "extend" gelijk aan n, het aantal knooppunten in de graaf. Binnen de procedure "extend (V)" wordt een for-statement uitgevoerd, waarvan het aantal repetitieslagen gelijk is aan het aantal uitgaande pijlen van V. De totale hoeveelheid werk, benodigd voor al deze for-statements is dus gelijk aan m, het aantal pijlen in de graaf. Daarom wordt een bovengrens van de rekentijd RT gegeven door:

$$RT \leq \alpha_1 \times n + \alpha_2 \times m \quad (\leq \alpha_3 \times n^2)$$

Ieder algorithme waarvan de rekentijd aan bovengenoemd criterium voldoet wordt beschouwd als een "theoretisch meest efficient algorithme".

Bij de gepubliceerde algorithmen [8] , [11] , is de bovengrens van de rekentijd evenredig met n^3. Deze algorithmen berekenen eerst de bereikbaarheidsmatrix;de rekentijd van deze berekening alleen reeds is evenredig met n^3.

Bij het algorithme, gepubliceerd in [12] blz 263, is de bovengrens van de rekentijd evenredig met m x n, bij geschikte datastructuur.

6. Enige toepassingsgebieden

Algemeen

In de inleiding werd het probleem, inzicht te verkrijgen in de structuur van een graaf, reeds aangesneden. Door de decompositie van een graaf in sterke componenten, is dit probleem ontleed in twee deelproblemen. Men kan eerst de verschillende sterke componenten elk apart analyseren; vervolgens kan men de samenhang tussen de verschillende sterke componenten bestuderen. Men kan namelijk de SC's laten "ineenschrompelen" tot knooppunten; men verkrijgt zó, uitgaande van de graaf G, een gereduceerd graaf, G^1 die cycle-vrij is [12] pag. 264 - 268.

Markov-ketens

Bij een graaf, die een Markov-keten voorstelt, zijn o.m. de "fuiken" van het markov-proces sterke componenten van de graaf, en wel precies die sterke componenten, die in de gereduceerd graaf geen uitgaande pijlen hebben.

Circuits

Beschouw het probleem, alle circuits te vinden van een gerichte graaf. Omdat een circuit altijd tot slechts één SC behoort, kan men het algorithme "SC-Worker" benutten bij een algorithme, dat alle circuits vindt [7].

Operationele Research

Bij de netwerkplanningsmethode EMPM [5], [6] wordt van de gebruiker een planningsnetwerk gevraagd, dat o.m. aan de volgende eisen moet voldoen:

(1) alle knooppunten moeten bereikbaar zijn vanuit het knooppunt START en
(2) vanuit alle knooppunten moet het knooppunt FINISH bereikbaar zijn.

Ramamoorthy [11] stelt dezelfde eisen aan grafen, die discrete sequentiele processen voorstellen, zoals computerprogramma's.
Ook bij de berekening van stromen in netwerken [4] worden soortgelijke eisen impliciet gesteld.
Hoe kan men nagaan of een gegeven graaf aan deze eisen voldoet?
Dit probleem kan m.b.v. het algorithme SC-Worker als volgt worden opgelost:
Men brengt in de graaf tijdelijk een pijl aan een FINISH naar START. Het aangeboden netwerk voldoet dan en slechts dan aan de eisen, als de nieuwe graaf uit slechts één SC bestaat.
Bij het algorithme voor EMPM [6] wordt het algorithme SC-Worker gebruikt voor twee doeleinden, nl.:

a) Een controle op de eisen (1) en (2);
b) Het bepalen van een volgorde van de sterke componenten. Deze volgorde kan gebruikt worden om de efficiency te verbeteren van het algorithme ter bepaling van vroegst mogelijke en laatst toegestane start.

7. Literatuur

[1] Bron, C.
Het nut van resursieve programmeertechnieken
Informatie 12 (1970) no. 12.

[2] Dijkstra, E.W.
A constructive approach to the problem of program correctness.
BIT 8 (1968), 174 - 180

[3] Daniels, M.J.M. en A.C.J. de Leeuw
Inleiding Meten
Dictaat nr. 1.102
Januari 1972 - Technische Hogeschool Eindhoven.

[4] Ford, L.R. and D.R. Fulkerson
Flows in networks
Princeton, N.J. (1962)

[5] Kerbosch, J.A.G.M. en H.J. Schell
Netwerkplanning volgens de methode extended MPM.
Technisch Rapport KS-1, Technische Hogeschool Eindhoven,
februari 1972.
Verschenen in:
Informatie 14 (1972) no. 4

[6] Kerbosch, J.A.G.M. en H.J. Schell
Een algorithme voor netwerkplanning volgens de Metra-potentiaal methode.
Technisch Rapport KS-2, Technische Hogeschool Eindhoven,
verschijnt binnenkort.

[7] Kerbosch, J.A.G.M. en J.C. Wortmann,
Some search-algorithms to find all simple cycles in a finite directed
graph: a case-study in efficiency.
Technical Report KW-2, Technical University Eindhoven,
verschijnt binnenkort.

[8] Moyles, Dennis M. and Gerald L. Thompson
An algorithm for finding a minimum equivalent graph of a digraph
Journal of the A.C.M. 16 (1969), 455 - 460.

[9] Naur, P.
Proof of algorithms by General Snapshots
BIT 6 (1966), 310 - 316.

[10] Naur, P.
Programming by Action Clusters
BIT 9 (1969), 250 - 258.

[11] Ramamoorthy, C.V.
Analysis of Graphs by Connectivity Considerations.
Journal of the A.C.M. 13 (1966), 211 - 222.

[12] Roy, B.
Algèbre moderne et théorie des graphes, Dunod, Paris, 1969.

Appendix A: Definities van fundamentele begrippen

Een *gerichte graaf G* is een tripel <K, P, ϕ> waarin K en P disjuncte verzamelingen zijn, en ϕ: P $\rightarrow$ K x K een afbeelding is. Elementen van K heten *knooppunten* (vertices), elementen van P heten *pijlen* (arcs). Het is toegestaan, dat $\phi(p_1) = \phi(p_2)$, d.w.z. dat tussen twee knooppunten meerdere pijlen lopen. Knooppunt V is een *directe voorganger* van knooppunt W, d.e.s.d. als een pijl p bestaat, zodat $\phi(p) = (V,W)$. In dat geval is W een *directe opvolger* van V. Bovendien heet V=b(p) het *beginpunt* van p en W=e(p) het *eindpunt* van p. Een rij van afwisselend knooppunten en pijlen, $K_0, p_1, K_1, \ldots, p_n, K_n$, $n \geq 0$, $K_{i-1} = b(p_i)$, $K_i = e(p_i)$, heet een *pad*. Als $K_i \neq K_j$, $i \neq j$ spreken wij van een *simpelpad*. K_0 heet het *beginpunt* van het pad, K_n het *eindpunt*. Als er in een graaf een pad is, dat een knooppunt V tot beginpunt, en een knooppunt W tot eindpunt heeft, dan is W *bereikbaar* vanuit V. Merk op, dat de relatie "bereikbaar vanuit" transitief is (V bereikbaar vanuit U, W bereikbaar vanuit V $\rightarrow$ W bereikbaar vanuit U), en reflexief (V bereikbaar vanuit V voor $\forall$ V $\in$ K). Twee knooppunten V en W heten *sterk verbonden*, d.e.s.d. als V bereikbaar vanuit W en W bereikbaar vanuit V.

Het tripel <K', P', ϕ'> heet een *deelgraaf* van <K, P, ϕ> als K'$\subset$K, P'$\subset$P, en $\phi'(p') = \phi(p')$ $\forall$ p' $\in$ P' . De deelgraaf heet *vol* als P' = {p| p $\in$ P en $\phi(p) \in K' \times K'$}. Kortheidshalve schrijven wij "deelgraaf" i.p.v. "volle deelgraaf". Zij D een deelgraaf van G. Als ieder tweetal knooppunten D_1 en $D_2 \in D$, sterk verbonden is, heet D een *sterk-verbonden deelgraaf*. Als D niet uitbreidbaar is, d.w.z. $\nexists$ knooppunt W $\notin$ D zodat W sterk verbonden met een knooppunt $D_1 \in D$, dan heet D een *sterk-samenhangende component* van de graaf G, ook wel *"sterke component"* genoemd; afkorting: *SC*. Merk op, dat de relatie "sterk-verbonden" een transitieve, symmetrische en reflexieve relatie is; oftewel, een equivalentie-relatie [3]. Dit heeft tot gevolg, dat de verzameling knooppunten K éénduidig ontleedbaar is in equivalentie klassen, t.a.v. de relatie "sterk verbonden" [3]. Deze equivalentie klassen van knooppunten vormen de sterke componenten.

Appendix B: bewijs van correctheid van het algorithme SC Worker

Binnen één bepaalde incarnatie van extend, zeg extend (U), beschouwen wij het recursie-niveau, waarop U de rol van V speelt. Dan zijn de lokale variabelen "index" en "plinstack" bekend. Wij zullen deze variabelen, indien dat nodig is, noteren als: "index[U]" resp. "plinstack[U]". Beschouw het moment "D (U, index)", waarop de pijlen, behorend bij "fe[U]" tm "index[U]", zijn "afgewerkt", terwijl de pijl, behorend bij index[U]+ 1 nog niet in behandeling is genomen. De waarde van "badge[U]" op dat moment noteren wij soms als "badge[U, index]".

Definitie: De "Until Now Earliest Attainable" van een knooppunt U bij gegeven waarde van index [U] is:

$$UNEA\ (U, index[U]) = \min_{r} \{plinstack[V_r] \mid (1)\}$$

(1) Vr is bereikbaar vanuit U via pad $Vr_0, \ldots Vr_k$,

waarbij: (a) $Vr_0 = U$, $Vr_k = V_r$, $k \geq 0$

(b) $plinstack[Vr_i] > plinstack[U]$, $1 \leq i \leq k-1$

(c) $Vr_1 \in \{suc[fe[U]], \ldots, suc[index[U]]\}$

Merk op, dat UNEA (U,p| p < fe[U]) = plinstack[U]

Voorbeeld:

STACK = {1,2,3,4}
De pijl (4,5) in onbehandeld,
de pijl (4,3) is behandeld
de pijl (3.1) is behandeld.
Dan is UNEA (4, index[4]) = 3
UNEA (3, le[3]) = 1
en UNEA (4, le[4]) = 2

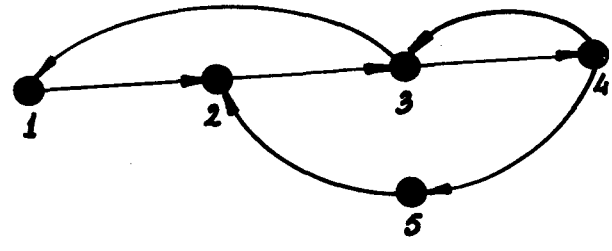


fig. 5

Definitie: De earliest Attainable EA (U) = UNEA(U, le[U])

Algorithme met moment-opnamen:

Hieronder volgt de tekst van het algorithme. Deze is doorspekt met comment-statements, die stellingen bevatten, en *momenten* aanwijzen [9].
De stellingen 5(V) en 6(V) gelden alleen op een vast recursieniveau; zij hebben betrekking op één bepaalde incarnatie van extend.
De stellingen (1), (2), (3) en (4) gelden bij iedere moment-opname, maar niet noodzakelijkerwijs ook tijdens de uitvoering van de tussenliggende "action clusters" [10].

```
procedure SC Worker(n , suc , fe , le); value n; integer n;
      integer array suc , fe , le;

begin integer i,new,insc,ploftop; integer array badge,stack[1:n];

PROP: comment Bij iedere momentopname gelden de volgende stellingen :
              (1) ploftop = aantal knooppunten in de verzameling STACK .
              (2) NEW + STACK + INSC = verzameling knooppunten van de graaf.
              (3) V ∈ INSC <=> V behoort tot een verzameling knooppunten X,
                                welke eerder als SC is gerapporteerd.
              (4) Iedere verzameling knooppunten X , die eerder als SC is
                  gerapporteerd , vormt inderdaad een SC ;

      procedure extend(V); value V; integer V;
      BODY OF EXTEND : SEE NEXT PAGE ;

INITIATE : new:= -1 ; insc:= n + 1 ; ploftop:= 0 ;
PUT ALL VERTICES INTO SET NEW :
      for i:=1 step 1 until n do badge[i]:=new ;
RUN : for i:=1 step 1 until n do if badge[i] = new then
PROP A: begin comment momentopname A(i): 0 < j < i  => knooppunt j ∈ INSC;
              extend(i);
PROP H:       comment momentopname H(i): 0 < j ≤ i  => knooppunt j ∈ INSC
        end
end split graph into strongly connected components ;
```

```
      procedure extend(V); value V; integer V;
      begin integer index , successor , plinstack;
            comment Bij iedere momentopname op dit recursie-niveau geldt:
PROP 5:           (5(V))  plinstack ≤ i ≤ ploftop  =>
                          stack[i] bereikbaar vanuit V ,
PROP 6:           (6(V))        1 ≤ i ≤ ploftop  =>
                          V bereikbaar vanuit stack[i] ;
PUT V ON STACK: badge[V]:=plinstack:=ploftop:=ploftop+1;stack[ploftop]:=V;
PROP B:     comment momentopname B(V): badge[V]=plinstack ,
                                       stack[badge[V]] bereikbaar vanuit V ;
SCAN SUCCESSORS OF VERTEX V :
            for index:=fe[V] step 1 until le[V] do
            begin successor := suc[index] ;
PROP C:           comment momentopname C(V, index): badge[V]≤UNEA(V,index-1) ,
                                       stack[badge[V]] bereikbaar vanuit V   ;
                  if badge[successor] = new then extend(successor) ;
                  if badge[successor] < badge[V]
                  then badge[V]:=badge[successor];
PROP D:           comment momentopname D(V, index): badge[V]≤UNEA(V, index) ,
                                       stack[badge[V]] bereikbaar vanuit V
            end for-statement ;
PROP E:     comment momentopname E(V): badge[V] ≤ EA(V) .
                                       stack[badge[V]] bereikbaar vanuit V.
                                       ¬ ∃ directe opvolger van V ∈ NEW ;
CHECK WHETHER SC IS FOUND :
            if badge[V] = plinstack[V] then
PROP F:     begin comment momentopname F(V): De verzameling knooppunten :
                        X = (x |x ∈ STACK ∧ plinstack[x] ≥ plinstack[V])
                        vormt een SC;
REPORT :          NEW PAGE ; for index:= plinstack[V] step 1 until ploftop
                        do begin print(stack[index]);
                                 badge[stack[index]] := insc
                           end;
                  ploftop := plinstack[V] - 1
            end;
PROP G:     comment momentopname  G(V):
                  if EA(V) = plinstack[V] then badge[V] = insc else
                 (badge[V] ≤ EA(V)  ∧  stack[badge[V]] bereikbaar vanuit V)
      end extend ;
```

Bewijs:

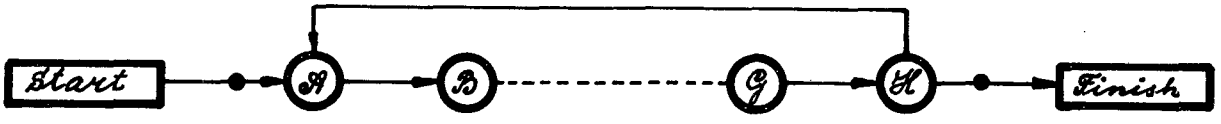
Wij zullen het bewijs met inductie leveren, als volgt:

- Allereerst wordt in een schema aangegeven, welke overgangen van moment i naar moment j mogelijk zijn, *zonder dat ondertussen nog een andere moment-opname plaats heeft gevonden.*
- Vervolgens wordt voor iedere overgang bewezen dat de stellingen blijven gelden, *mits* zij op alle voorafgaande momenten golden.
- Tenslotte wordt een eindigheidsbewijs gegeven.

De mogelijke overgangen:

In het schema geeft een pijl met een stip aan, dat de overgang expliciet wordt behandeld. Een pijl zonder stip duidt een overgang aan, waarvan het bewijs van correctheid triviaal is.

Hoofdflow:



procedure extend:

recursie-niveau: k-1

recursie-niveau: k

recursie-niveau: k+1

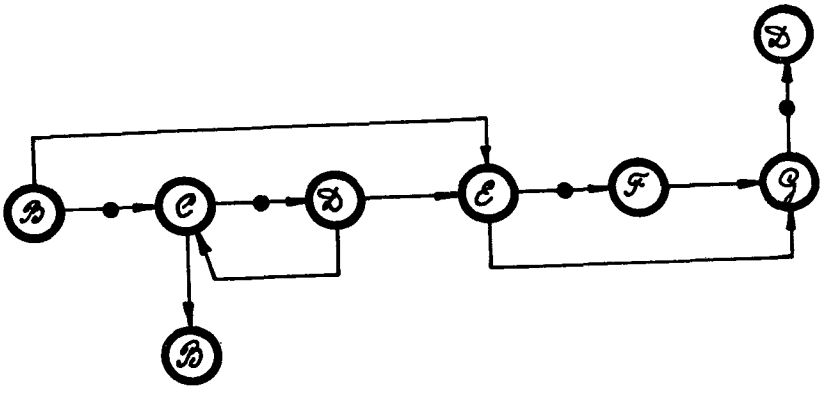


fig. 6

Moment: *Plaats in het programma:*

A : In het hoofdprogramma, vóór de aanroep van extend
B : Na de aanroep van extend
C : Begin van een repititie-slag van de for-statement binnen extend
D : Eind van een repititie-slag van de for-statement binnen extend
E : Na de for-statement binnen extend
F : Nadat een SC ontdekt is
G : Voor de terugkeer uit extend
H : In het hoofdprogramma, na de terugkeer uit extend

De niet-triviale overgangen:

START → A(1):

alleen de stellingen (1), (2), (3) en (4) zijn van toepassing. Aangezien alle knooppunten tot de verzameling NEW behoren, is de correctheid duidelijk.

B(V)→C(V,fe[V]):

De grootheden in de stellingen 1 t/m 6 veranderen niet, dus deze stellingen blijven gelden. Op moment C(V,fe[V]) geldt:
index = fe[V] → UNEA(V,index-1) = plinstack[V], per definitie;
badge[V] = plinstack[V], want badge[V] is ongewijzigd sinds B(V).

C(V,index) → D(V,index):

Deze overgang heeft alleen plaats, als successor ∉ NEW op C(V,index).
- Veronderstel: successor ε INSC → UNEA(V,index-1) = UNEA(V,index)
  → badge(V) is terecht onveranderd gebleven.
- Veronderstel: successor ε STACK en plinstack [successor] < plinstack[V]
Dan geldt op het moment C (V,index):

$$\begin{aligned} badge[V,index] &= \min\{badge[successor], badge[V,index-1]\} \\ &\leq \min\{plinstack[successor], UNEA(V,index-1)\} \\ &= UNEA(V,index) \end{aligned}$$

Stack[badge[V,index]] is bereikbaar vanuit V, omdat stack[badge[successor]] bereikbaar is vanuit V (transitiviteit) en stack[badge[V,index-1]] vanuit V bereikbaar is.
- Veronderstel: successor ε STACK en plinstack[successor] ≥ plinstack[V]
  dan geldt: UNEA(V,index-1) = UNEA(V,index)

De grootheden van de stellingen 1 t/m 6 veranderen niet tussen C(V,index) en D(V,index). De stellingen blijven gelden.

E(V) → F(V):

Volgens stellingen 5(V) en 6(V) geldt:

X:= {x|x ε STACK en plinstack[x] ≥ plinstack[V]}

vormt een sterk-verbonden deelgraaf. Wij moeten aantonen, dat X niet uitbreidbaar is.

- Veronderstel: X is uitbreidbaar met knooppunt U ε INSC; dit is in tegenspraak met stelling (4)
- Veronderstel: X is uitbreidbaar met knooppunt U ε STACK, plinstack [U]< plinstack[V]. Dan geldt: EA(V) < plinstack[V]; volgens stelling E(V) geldt: badge[V] ≤ EA(V). Dus badge [V] < plinstack[V]. Maar dan vindt de overgang E(V) → F(V) niet plaats. Contradictie.
- Veronderstel: X is uitbreidbaar met knooppunt U ε NEW.
  Dan bestaat een pijl p met b(p) ε X en e(p) ε NEW. Dit is in strijd met eigenschap E(b(P)).

G(successor) → D(V,index):

De grootheden van de stellingen 1 t/m 4 blijven onveranderd bij deze overgang. Hun correctheid blijft dus behouden. Stelling 5(V) is triviaal. Wij bewijzen nu eerst stelling 6(V):

Voor 1 ≤ plinstack[i] ≤ plinstack[V] is deze triviaal. Voor plinstack[i] > plinstack[V] geldt, op grond van de eigenschap G(i), dat vanuit i het knooppunt stack[badge[i]] bereikbaar is, met badge[i] ≤ EA(i) < plinstack[i]. Op grond van de transitiviteit geldt dat knooppunt V bereikbaar is vanuit knooppunt i, q.e.d.

Wij bewijzen vervolgens, dat badge[V] ≤ UNEA(V,index) op D(V,index):

UNEA(V,index) = min{UNEA(V,index-1), EA(successor)}
≥ min{badge[V,index-1] , badge[successor]}
= badge[V,index]

De bereikbaarheid van stack[badge[V]] kan op analoge wijze bewezen worden als bij de overgang C(V,index) → D(V,index).

H → FINISH:

Alle knooppunten behoren nu tot de verzameling INSC. De stellingen (3) en (4) garanderen de correctheid van het algorithme.

Eindigheidsbewijs:

Het aantal aanroepen van de procedure "extend" is gelijk aan n, daar de procedure "extend(V)" alleen wordt aangeroepen als V ∈ NEW; knooppunt V wordt direct na de aanroep van "extend(V)" overgeheveld naar de verzameling STACK, en kan daarna niet meer in de verzameling NEW komen.
De enige gebruikte repeatable statements, zijn for-statements die een eindige bovengrens van de lopende variabele hebben. De lopende variabele wordt tijdens de for-loop niet gewijzigd. Het algorithme is daarom eindig.