TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Een algorithme voor netwerkplanning volgens de (extended) metra-potentiaal methode

Citation for published version (APA):
Kerbosch, J. A. G. M., Schell, H. J., & Wortmann, J. C. (1973). *Een algorithme voor netwerkplanning volgens de (extended) metra-potentiaal methode*. (TH Eindhoven. ORS, Vakgr. operationele research : rapport; Vol. KS-2). Technische Hogeschool Eindhoven.

Document status and date:
Gepubliceerd: 01/01/1973

Document Version:
Uitgevers PDF, ook bekend als Version of Record

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

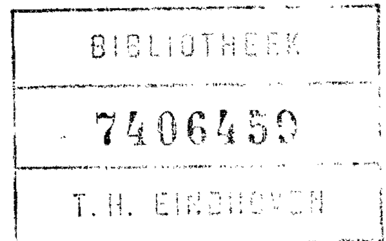
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 08. Feb. 2024


BIBLIOTHEEK
7406459
T.H. EINDHOVEN

EEN ALGORITHME VOOR NETWERKPLANNING VOLGENS DE (EXTENDED) METRA-POTENTIAAL METHODE

door
ir. J.A.G.M. Kerbosch
H.J. Schell
J.C. Wortmann

Rapport KS - 2
augustus 1973

Afdeling der Bedrijfskunde
Groep Operationele Research
Technische Hogeschool Eindhoven

Een beknopte weergave van dit rapport zal verschijnen in het maandblad informatie.

Samenvatting

Het algorithme, dat in dit rapport wordt beschreven, vraagt een MPM- of EMPM-planningsnetwerk als input. Het algorithme gaat na, of het aangeboden netwerk aan alle gestelde syntaktische korrektheids-eisen voldoet. Bij een korrekt netwerk wordt het volgende berekend:

- *de vroegst-mogelijke starttijden van de aktiviteiten;*
- *de laatst-toegestane starttijden van de aktiviteiten;*
- *het kritieke pad.*

Het algorithme, dat hier wordt gepresenteerd is aanzienlijk efficienter, dan de tot nog toe bekende algorithmen. Er wordt een bewijs van korrektheid gegeven.

Abstract

The algorithm, which is described in this paper, asks for an MPM- or EMPM-planning-network as input. The algorithm checks, whether the presented network satisfies all requirements of syntactical correctness. In case of a correct network, the following quantities are computed:

- *the earliest starts of the activities;*
- *the latest starts of the activities;*
- *the critical path.*

The algorithme, which is presented here, is far more efficient than the algorithms, known by now. A proof of correctness is given.

Voorwoord

Het ontwerpen van een algorithme wordt vaak gezien als een éénmalig projekt, dat op een gegeven moment gereed is. In werkelijkheid is dit gewoonlijk niet het geval. Veeleer is het ontwerpen van een algorithme een kontinu proces, waarin steeds nieuwe ideeën worden verwerkt. Wanneer men wetenschappelijk te werk wil gaan, dan wenst men de korrektheid van het geproduceerde algorithme te bewijzen. Het korrektheidsbewijs geeft dikwijls zoveel inzicht in het probleem dat ideeën voor een verbeterd algorithme ontstaan. Een dergelijk ontwerp-proces is iteratief.
Daarnaast zijn er vele eisen, die aan een programma worden gesteld vanuit gebruikers standpunt (foutmeldingen etc.). Een aantal van deze eisen worden eerst duidelijk, wanneer het programma inderdaad wordt gebruikt. Ook deze gebruikservaringen dragen ertoe bij, dat het ontwerpen van een algorithme een kontinu proces is.

Uit het bovenstaande wordt - naar wij hopen - duidelijk, waarom het verschijnen van dit rapport zo lang op zich heeft laten wachten. Immers, de vertragingstijd tussen een redelijk werkend algorithme, en de beschrijving daarvan bedraagt minstens vier maanden. Wanneer men dan het geschreven koncept-rapport vergelijkt met het algorithme, zoals het inmiddels is, zijn de verschillen zo groot, dat men slechts één mogelijkheid ziet: een kompleet nieuw rapport produceren.

Wij zijn dank verschuldigd aan ir. V. Swinkels, die in de beginfase van het ontwerpproces als student een belangrijke bijdrage aan de basisideeën leverde. Het eerste gedeelte van dit rapport werd getypt door mej. Annelies Ummels, tevens deed zij het korrektie-werk. Daarna werkten mevr. Lia Dietz-van Bergen en mevr. Dorothé Haak-Brandes er korte tijd aan. Mej. Marleen van Balen verzorgde de tweede helft van het rapport. De tekeningen, die de heer van der Heyden heeft verzorgd, vormen een essentieel gedeelte van het manuscript. Onze hartelijke dank voor dit excellente vakwerk. Daarnaast heeft de heer van der Heyden met grote welwillendheid en akkuratesse de Algol-teksten getypt.

INHOUD Blz.

1. Inleiding 6

2. Toelichting van enkele begrippen en notaties 8

3. Syntaktische korrektheid van het netwerk 10

4. Datastrukturen 12
   4.1. Inleiding 12
   4.2. Inverted list 12
   4.3. Multilist 15

5. Bestaande algorithmen 17
   5.1. Algorithmen voor cykelvrije netwerken 17
      5.1.1. Beschrijving 17
      5.1.2. Efficiency 19
   5.2. Een algorithme voor cykelvrije en niet-cykelvrije netwerken 19
      5.2.1. Beschrijving 19
      5.2.2. Efficiency 25
   5.3. Kritische beschouwing 26

6. Een verbeterd algorithme 27
   6.1. Het basis idee 27
   6.2. Een volgorde van sterke komponenten 28
   6.3. Splitsing van de verzameling pijlen binnen één SC 29

7. Hoofdprogramma van het algorithme 34
   7.1. Toelichting bij de algol-tekst 34
   7.2. Algol-tekst 35

8. Het deel-algorithme "Decompose, ordem SC and check" 36
   8.1. Beschrijving 36
   8.2. Voorbeeld 38
   8.3. Toelichting bij het algol-programma 39
   8.4. Algol-programma 40

9. Het deel-algorithme "Split arcs and orden nodes" 42
   9.1. Beschrijving 42
   9.2. Voorbeeld 46
   9.3. Toelichting bij het algol-programma 49
   9.4. Algol-programma 53

VERVOLG INHOUD Blz.

10. Het deel-algorithme "Compute es, check and find cp" 55
10.1. Beschrijving 55
10.2. Algol-programma 66

11. Het deel-algorithme "Compute ls" 70
11.1. Beschrijving 70
11.2. Algol-tekst 74

12. Kwaliteitsbeoordeling van het algorithme 76
12.1. Robuustheid 76
12.2. Overdraagbaarheid 76
12.3. Doorzichtigheid 76
12.4. Nauwkeurigheid 77
12.5. Betrouwbaarheid 77
12.6. Aanpasbaarheid 77
12.7. Geheugenbeslag en efficiency 78
12.7.1. Vooronderstellingen 78
12.7.2. Geheugenbeslag 78
12.7.3. Efficiency 79

Literatuur 84

Appendix A : Definities van fundamentele begrippen 86

Appendix B1: Kommentaar op het Gewald-Sauler algorithme 88

Appendix B2: De efficiency van het Gewald-Sauler algorithme 91

Appendix C : Bewijs van korrektheid van het algorithme 93
Inleiding 93
1. Korrektheid van het deel-algorithme "Decompose, orden SC and check" 94
2. Korrektheid van het deel-algorithme "Split arcs and orden nodes" 97
3. Korrektheid van het deel-algorithme "Compute es, check and find cp" 101
4. Korrektheid van het deel-algorithme "Compute ls" 106
5. Efficiency 109

1 *Inleiding*

In [7] is de netwerk-planningsmethode "Extended MPM" gepresenteerd. In het onderhavige rapport zal een efficient algorithme worden beschreven, dat de berekeningen aan een MPM- of EMPM-planningsnetwerk uitvoert.
Bij een planningsnetwerk behoort een gerichte 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 enige veel gebruikte begrippen en notaties nader toegelicht. In hoofdstuk 3 wordt gedefinieerd aan welke eisen een netwerk moet voldoen, om als planningsnetwerk zinvol te zijn. In het uiteindelijk te presenteren algorithme wordt deze z.g. syntaktische korrektheid, onderzocht.
In hoofdstuk 4 bespreken wij enige methoden, volgens welke het netwerk in de computer kan worden weergegeven (data-struktuur). In hoofdstuk 5 worden enige bestaande algorithmen besproken. Hierbij blijkt, dat deze algorithmen voor netwerken, waarin cykels mogen voorkomen aanzienlijk meer rekentijd nodig hebben, dan algorithmen voor cykelvrije netwerken.
In hoofdstuk 6 presenteren wij in grote lijnen een nieuw algorithme, geschikt voor cykelvrije en niet-cykelvrije netwerken.
In hoofdstukken 7 t/m 11 wordt het geschetste beeld nader uitgewerkt, en worden de deel-algorithmen gedetailleerd beschreven. Er is telkens een voorbeeld toegevoegd, en de komplete Algol-tekst wordt gegeven. In appendix C wordt de korrektheid van de deelalgorithmen bewezen.
In hoofdstuk 12 wordt de efficientievan het algorithme besproken.

Het algorithme is gedeeltelijk gebaseerd op het algorithme SC-Worker, zoals beschreven in [10]. Voor een volledig begrip van het eerste deel-algorithme is begrip van SC-Worker onontbeerlijk.

Bij verschillende deel-algorithmen wordt gebruik gemaakt van rekursieve programmeertechnieken [3]. In tegenstelling tot de beschrijving van SC Worker [10] hebben wij in dit rapport alleen de rekursieve beschrijvingswijze gepresenteerd. Bij een dergelijk groot programma wordt de iteratieve beschrijvingswijze snel onelegant, onleesbaar en volumineus. Wij hopen,

dat lezers die niet vertrouwd zijn met rekursie, na het lezen van de beschrijving van SC-worker, hiermede voldoende vertrouwd zullen zijn om dit rapport te kunnen volgen.

Bij de beschrijving van de algorithmen in dit rapport is gebruik gemaakt van THE-ALGOL-60, zoals gebruikt op de EL-X8, vanwege de duidelijke lay-out. Het in hoofdstuk 6 gepresenteerde algorithme maakt deel uit van het netwerkplanningsprogramma ANNETTE II, geschreven in Burroughs Extended Algol, en is uitvoerig getest op de B6700 komputer van de TH Eindhoven.

Terwille van de leesbaarheid van de programma's hebben wij de volgende konventie ingevoerd:
indien de deklaratie van een procedure op een bepaalde plaats programmatechnisch noodzakelijk is, maar de leesbaarheid van het geheel verstoort, vervangen wij de procedure-tekst op die plaats door:

< naam van de procedure >;

De procedure-tekst presenteren wij dan later.

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 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" zgn *transitief*, d.w.z.:

$$\left.\begin{array}{l}\text{z bereikbaar vanuit y}\\ \text{y bereikbaar vanuit x}\end{array}\right\} \Longrightarrow \text{z bereikbaar vanuit x}$$

Een *sterke komponent* van een graaf G is een deelgraaf $D \subset G$, met de eigenschap, dat ieder knooppunt van D vanuit ieder ander knooppunt bereikbaar is; bovendien moet de deelgraaf *niet-uitbreidbaar* zijn, d.w.z. als men één of meer knooppunten aan D toevoegt, geldt voor de nieuwe deelgraaf niet meer, dat ieder knooppunt vanuit ieder ander knooppunt bereikbaar is.
Een sterke komponent heet *enkelvoudig*, indien deze sterke komponent uit slechts één knooppunt bestaat; indien een sterke komponent uit meerdere knooppunten bestaat, heet hij *samengesteld*.
Wij spreken van een *netwerk*, als met iedere pijl van de graaf een lengte is geassocieerd. De *lengte van een pad* is de som van de lengten van de pijlen van dat pad. Een langste pad van het knooppunt START naar het knooppunt FINISH heet een *kritiek pad*.

De *vroegst - mogelijke start - tijd* (VMS-tijd) of *earliest start* (es) van een knooppunt V wordt als volgt gedefinieerd:
es [V] = een gegeven waarde , als V = START
es [V] = es [START] + lengte van het langste pad van START naar V,
als V $\neq$ START.

De *laatst - toegestane start - tijd* (lts - tijd) of *latest start* (ls) van een knooppunt V, wordt als volgt gedefinieerd:
ls [V] = es [V] , als V = FINISH
ls [V] = ls [FINISH] - lengte van het langste pad van V naar FINISH,
als V $\neq$ FINISH

In de beschrijving van de algorithmen zullen wij vaak spreken over *het in rekening brengen van een pijl P.* Hiermee wordt, bij de berekening van de vroegst-mogelijke-start tijden, het volgende bedoeld:

```
 if   earliest start [b(P)] + length [P]  > earliest start [e(P)]
then begin earliest start [e(P)]: =  earliest start [b(P)] + length[P];
           OVERIGE ADMINISTRATIE
     end;
```

Bij de berekening van de laatst toegestane start-tijden wordt hiermee bedoeld:

```
 if      latest start [b(P)] + length [P] > latest start [e(P)]
then begin latest start [b(P)] := latest start [e(P)] - length [P];
          OVERIGE ADMINISTRATIE
     end;
```

Een formele en meer uitgebreide behandeling van begrippen is gegeven in Appendix A.

3 *Syntaktische korrektheid van het netwerk*

We noemen een netwerk *syntaktisch korrekt*, indien het voldoet aan de volgende voorwaarden:

1. "uniek beginknooppunt"
   Er is één en slechts één (door de gebruiker van het algorithme aan te geven) beginknooppunt. We zullen dit knooppunt in het vervolg START noemen.

2. "uniek eindknooppunt"
   Er is één en slechts één (door de gebruiker aan te geven) eindknooppunt, dat we FINISH zullen noemen.

3. "bereikbaarheid"
   Ieder knooppunt ligt op een pad van START naar FINISH.
   Indien er geen pad bestaat van START naar een knooppunt K, dan is de vroegst mogelijke starttijd van dat knooppunt niet gedefinieerd. We noemen zo'n knooppunt K een *loze start*. Indien er geen pad bestaat van een knooppunt K naar FINISH, dan is de laatst toegestane starttijd van dit knooppunt niet gedefinieerd. We noemen zo'n knooppunt K een *loze finish*.

4. "afwezigheid van positieve cykels"
   Het netwerk mag cykli bevatten [7]. Deze dienen echter van niet-positieve lengte te zijn (≤ 0).

In het door ons gepresenteerde algorithme vindt een volledige kontrole op deze eisen plaats: kontrole op de voorwaarden 1. en 2. is triviaal. Kontrole op voorwaarde 3. vindt plaats in het deelalgorithme "decompose, orden SC and check". De voorwaarden 1. , 2. en 3. , noemen wij de bereikbaarheidsvoorwaarden. Kontrole op voorwaarde 4. vindt plaats in het deelalgorithme "compute es, check and find cp".

De bestaande algorithmen [5], [13], [15] , allen gebaseerd op [4], voeren geen kontrole uit op voorwaarde 3.
In sommige van deze algorithmen wordt ervoor zorg gedragen, dat aan voorwaarde 3 altijd is voldaan, bv. door het introduceren van fiktieve pijlen met lengte = 0 van START naar alle knooppunten, en van alle knooppunten naar FINISH. Soms worden deze fiktieve pijlen impliciet geintroduceerd, bv. door het initialiseren van alle VMS-tijden op 0. Hier zijn de volgende bezwaren tegen aan te voeren:

- de introduktie van fiktieve pijlen kan een netwerk kreëren, dat niet overeenstemt met de bedoeling van de gebruiker
- de introduktie van fiktieve pijlen kan fouten van de gebruiker (bv. inputfouten) verdoezelen.

Voor het in hst. 6 e.v. gepresenteerde algorithme gelden deze bezwaren niet, omdat een volledige kontrole op voorwaarde 3 wordt uitgevoerd. Indien een netwerk niet voldoet aan de voorwaarden 1. , 2. , of 3. , dan heeft berekening van vroegst mogelijke- en laatst toegestane starttijden geen zin. In zo'n geval wordt een lijst afgedrukt van alle knooppunten die niet een van deze voorwaarden voldoen, en worden de berekeningen vervolgens afgebroken.

4 *Datastrukturen*

4.1 *Inleiding*

Een netwerk kan op verschillende manieren in een komputer worden gerepresenteerd; iedere representatie zullen wij een *datastruktuur* noemen. De datastruktuur heeft een grote invloed op efficiency en elegantie van de algorithmen die gebruik maken van deze datastruktuur. De keuze van een datastructuur en het ontwerpen van een algorithme zijn dan ook niet twee verschillende zaken, maar vormen twee aspekten van hetzelfde ontwerp-proces.
De algorithmen voor netwerkplanning kunnen worden onderscheiden in:
- de reken-algorithmen;algorithmen voor het testen op de korrektheid van een netwerk, voor de berekening van de vms-tijden en lts-tijden, etc. Alle algorithmen uit dit rapport behoren tot deze klasse.
- de administratieve-algorithmen: dit zijn de in het kader van netwerkplanning onontbeerlijke algorithmen, zoals algorithmen voor het aanbrengen van wijzigingen in het netwerk (bestandsmanipulatie), voor de rapportage, etc.

Men dient zich te realiseren, dat beide soorten algorithmen hun eisen stellen aan de datastruktuur, en dat de uiteindelijke keuze vaak een kompromis is; ook zal men soms besluiten, de ene datastruktuur uit de andere af te leiden, wanneer dat voor een bepaald algorithme wenselijk is.
Als twee voor de hand liggende datastrukturen kunnen we noemen:
1. de "inverted list"
2. de "multilist"

4.2 *Inverted list*

De, in dit rapport gebruikte inverted-list datastruktuur veronderstelt, dat de knooppunten van de graaf zijn genummerd: 1,...,n. Er wordt gebruik gemaakt van de volgende 4 ééndimensionale arrays:
- suc (successors)
- fe (place of first outward edge in "suc")
- le (place of last outward edge in "suc")
- length

In het array suc zijn de direkte opvolgers van knooppunt "i" genoteerd op de plaatsen fe[i] tot en met le[i] . De lengten van de bijbehorende pijlen staan in het array length op de plaatsen fe[i] tot en met le[i]. Indien knooppunt i geen opvolgers heeft, geldt: le[i] < fe[i]. In het onderstaande schema is het array "length" weggelaten.

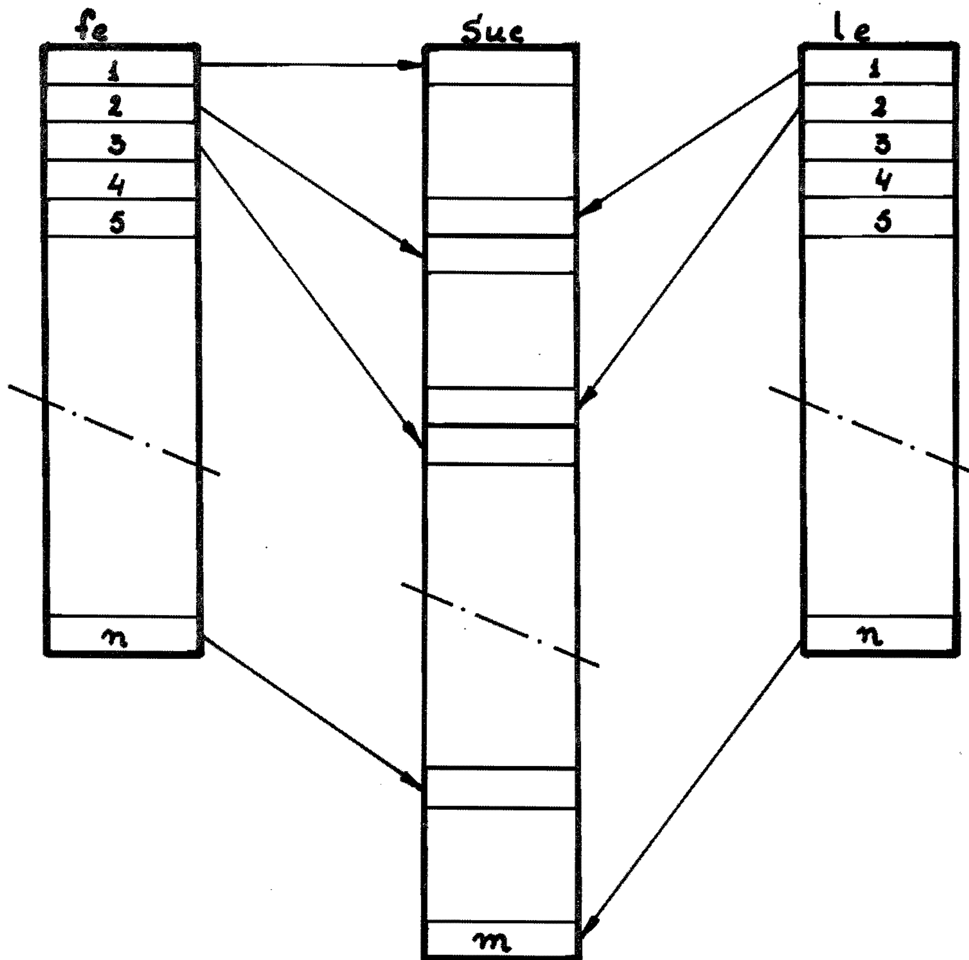
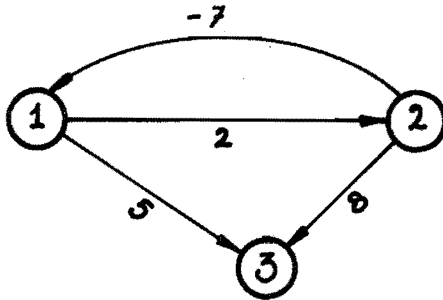


fig. 4.2.1

Als voorbeeld geven we nog de datastruktuur behorend bij het volgende netwerk:



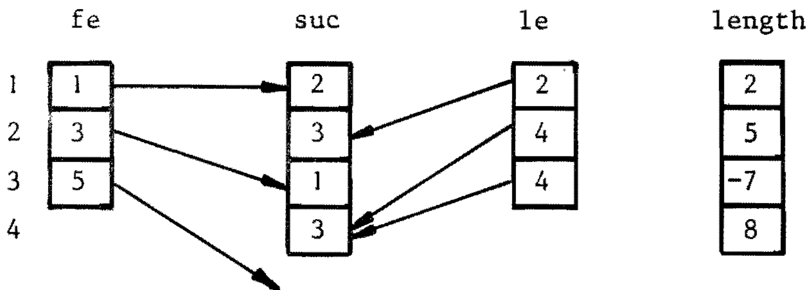


fig. 4.2.2

De algorithmen in dit rapport zijn alle gebaseerd op deze datastruktuur. Een aanpassing van de algorithmen aan de multilist-structuur, zoals die in 4.3 zal worden besproken, is eenvoudig te realiseren.

Gewald und Sauler [5] representeren een graaf in een drietal één dimensionale array's: I, J en PD .
I[K] bevat dan het beginpunt b(P) van een pijl P, J[K] het eindpunt e(P) en PD[K] de lengte van P.
Als voorbeeld, de graaf in fig. 4.2.2.:

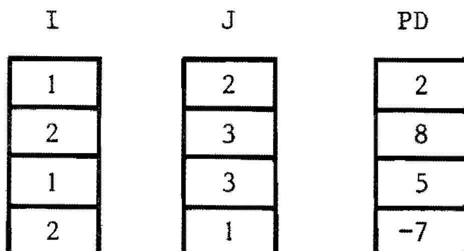


fig. 4.2.3.

De volgorde waarin de pijlen bij Gewald und Sauler worden opgeslagen en later in dit algorithme worden afgewerkt wordt bepaald door de volgorde waarin ze in de input voorkomen.

4.3 *Multilist*

Bij deze datastruktuur onderscheiden we in dit geval twee soorten rekords:
- rekords die informatie over een knooppunt bevatten
- rekords die informatie over een pijl bevatten.

Ieder pijlrekord P is onderdeel van 2 kettingen:
- een ketting gevormd door de pijlen die uitgaan van knooppunt b(P)
- een ketting gevormd door de pijlen die inkomen in knooppunt e(P)

Ieder knooppunt-rekord V vormt dan het uitgangspunt van 2 kettingen:
- de ketting van uitgaande pijlen van V
- de ketting van inkomende pijlen in V

Het illustreren van een dergelijke multilist m.b.v. een figuur, maakt i.h.a. weinig duidelijk; wij laten het hier derhalve achterwege. Zie ook [9].

In deze datastruktuur heeft men voor ieder knooppunt direkt de beschikking over:
- een lijst van alle direkte voorgangers;
- een lijst van alle direkte opvolgers.

Ter kontrole van de input worden dergelijke lijsten veelvuldig gevraagd bij netwerkplanning. Een voordeel van een multilist-struktuur boven een inverted-list is, dat het eenvoudig is, pijlen te verwijderen en/of, toe te voegen. Dit laatste is o.a. belangrijk bij konversationele-verwerking van een netwerk. Zie [8].

De bezwaren van een ketting-struktuur zijn:
- grotere ruimte-behoefte i.v.m. de ketting-administratie;
- een grotere adresseringsdiepte;
- grotere komplexiteit van elementaire handelingen, zoals: "Beschouw de volgende uitgaande pijl".

Het laatste bezwaar is te ondervangen door een passend procedure-pakket. Een procedure-pakket voor het manipuleren met multilists, is te vinden in [9].

In het programma ANNETTE II is de volgende strategie gevolgd: Het netwerk wordt d.m.v. een multilist weergegeven, omdat bij deze struktuur het leveren van kontrole-lijsten en het aanbrengen van wijzigingen, eenvoudig is. Onmiddellijk voorafgaand aan het rekengedeelte wordt deze multilist-struktuur omgezet in een inverted list-struktuur, zoals boven beschreven. Hierdoor wordt in het rekengedeelte, het volgende bereikt:

- de algorithmen blijven inzichtelijk, ook zonder kennis van het procedure pakket voor multi-lists [9];
- de geringere adresseringsdiepte draagt bij tot de efficiency;
- door de geringere ruimte-behoefte bestaat de mogelijkheid, de informatie in het kerngeheugen te houden, in dat gedeelte van het programma, waarin de gegevens in een willekeurige volgorde ter beschikking moeten staan.

5 *Bestaande algorithmen*

5.1 *Algorithmen voor cykelvrije netwerken*

5.1.1 *Beschrijving*

Bij de berekening van de vroegst-mogelijke start tijden, is het aantrekkelijk, een pijl P pas"in rekening te brengen" (zie hst.2) als de definitieve vms-tijd van knooppunt b(P) bekend is. Want in dat geval behoeft pijl P slechts éénmaal gedurende het algorithme in rekening te worden gebracht.
De meeste algorithmen voor cykelvrije netwerken gebruiken dit idee, door de uitgaande pijlen van knooppunt V pas in rekening te brengen, als de uitgaande pijlen van alle voorgangers van knooppunt V reeds in rekening gebracht zijn.
Bijvoorbeeld:

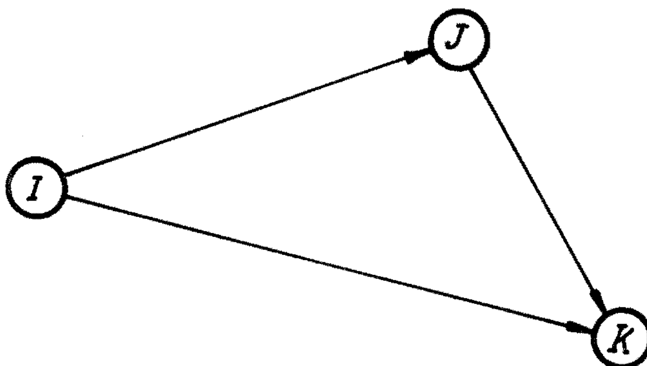


fig. 5.1 De pijl J,K wordt pas in rekening gebracht als de uitgaande pijlen van knooppunt I allemaal in rekening zijn gebracht.

Dit vereist een ordening van de knooppunten t.a.v. de relatie "bereikbaar vanuit"; d.w.z., alle voorgangers van knooppunt V hebben een lager rangnummer in de ordening dan V, en alle opvolgers hebben een hoger rangnummer. Bij een cykelvrij netwerk bestaan er i.h.a. een groot aantal van dergelijke ordeningen. Bij een niet-cykelvrij netwerk bestaat een dergelijke

ordening *niet*.
De algorithmen voor cykelvrije netwerken hebben daarom de volgende algemene struktuur:

deel-algorithme (a): orden de knooppunten m.b.t. de relatie: "bereikbaar vanuit";

deel-algorithme (b): werk de knooppunten in deze volgorde af, en breng hun uitgaande pijlen in rekening.

Voor het deel-algorithme (a) bestaan een aantal methoden, waarvan de *rangsbepalingsmethode* [15] de bekendste is.

Wij beschrijven hieronder een andere methode voor deel-algorithme(a), die ook verder in dit rapport van belang is. Wij gaan ervan uit, dat alle knooppunten vanuit start bereikbaar zijn. Bij deze methode wordt een array "list" met knooppunt-nummers gevuld. Tijdens het algorithme geeft de variabele "ploftop" (place of top) de plaats in "list" aan, waar het laatste knooppuntnummer is weggeschreven. Het volgende knooppuntnummer komt dan op plaats: ploftop + 1. In het boolean-array-element: inlist[V] wordt bijgehouden, of knooppunt V reeds tot "list" behoort.
De kern van het algorithme bestaat uit een rekursieve procedure "put into list (V)". Deze procedure gaat na of alle directe opvolgers van V reeds tot "list" behoren. Indien dat niet het geval is voor een opvolger, wordt "put into list (opvolger)" aangeroepen. Dit bewerkstelligt, dat knooppunt V pas in "list" komt, als alle opvolgers van V reeds tot "list" behoren.

```
procedure put into list (V); value V; integer V;
begin integer index, successor;
SCAN SUCCESSORS OF V:
      for index:=fe[V] step 1 until le[V] do
      begin successor:= suc[index];
            if not inlist[successor] then put into list (successor)
      end;
      ploftop:= ploftop + 1; list[ploftop]:= V; inlist[V]:= true
end;

MAIN PROGRAM: ploftop:= 0;  inlist[ i]:= false, i=1,...,n;
              put into list (START) ;
```

Wanneer wij ervan uitgaan, dat alle knooppunten vanuit START bereikbaar zijn, zijn nu alle knooppuntnummers in array "list" genoteerd. Hierbij is de volgorde zodanig dat:
knooppunt V is bereikbaar vanuit W →
rangnummer van V in "list" ≤ rangnummer van W in "list".

Het deelalgorithme (b) is als volgt:

```
Initialisering: es[i]:= -∞, i = 1,...,n;
                es[START]:= vroegste start van het projekt;
Berekening    : for i:= n step -1 until 1 do
                begin V:= list[i];
                     BRENG DE UITGAANDE PIJLEN VAN V IN REKENING
                end;
```

5.1.2 *Efficiency*

Het deelalgorithme (a) roept precies *n* maal de procedure "put into list (V)" aan; bovendien worden alle *m* pijlen precies éénmaal beschouwd.
Het deelalgorithme (b) bevat een for-statement met *n* repetitieslagen, ook hier worden alle pijlen precies éénmaal beschouwd.
De initialisering bevat een constant deel en een for-statement met *n* repetitieslagen.
Daarom wordt de rekentijd RT begrensd door de ongelijkheid:

$$RT \leq \alpha_0 + \alpha_1 \times n + \alpha_2 \times m$$

Ieder algorithme, waarvan de rekentijd aan bovenstaande ongelijkheid voldoet, noemen wij:
een *theoretisch meest efficient* algorithme.

5.2 *Een algorithme voor cykelvrije en niet-cykelvrije netwerken*

5.2.1 *Beschrijving*

Bij een niet-cykelvrij netwerk is een ordening, zoals hierboven onder (a) beschreven niet mogelijk. Immers, in een niet-cykelvrij netwerk komen altijd

twee knooppunten A en B voor, zodanig dat A bereikbaar is vanuit B en B bereikbaar is vanuit A.[1)]

De algorithmen, die ons uit de literatuur bekend zijn [5],[13],[15] zijn alle gebaseerd op de korste-pad-berekeningsmethode van Ford-Fulkerson [4]. Dit algorithme is iteratief. In iedere iteratie worden alle pijlen achtereenvolgens éénmaal in rekening gebracht. Het algorithme stopt, als gedurende een iteratie geen enkele verandering in de vms-tijden is opgetreden.

Algorithme:

```
Initialisering: comment wij gaan uit van een syntaktisch korrekte graaf;
es[i]:= -∞, i=1,...,n; es[start]:= vroegste start van project;
Berekening    : repeat change:= false;
                         for node:= 1 step 1 until n do
                         for index:= fe[node] step 1 until le[node] do
                         begin successor:= suc[index];
BRENG PIJL IN REKENING:         if es[successor] < es [node] + length[index] then
                                begin change:= true;
                                      es[successor]:= es [node] + length[index]
                                end
                         end
                until not change;
```

Indien het aangeboden netwerk een of meer positieve cykels bevat is, het bovenstaande algorithme niet eindig. Gewald und Sauler [5] voeren daarom na iedere iteratie een kontrole uit op positieve cykels. Wij zullen het Gewald-Sauler-algorithme hieronder presenteren in een gewijzigde vorm, welke aansluit bij het vervolg van dit rapport. In Appendix B geven wij een aantal bezwaren tegen de oorspronkelijke implementatie in [5], en een lijst met verschilpunten.
Het opsporen van positieve cykels gebeurt als volgt:

(1) Het gedeelte "BRENG PIJL IN REKENING" van het bovenstaande algorithme wordt gewijzigd tot:

---

1) Krafft [12] heeft dit in het geheel niet begrepen, zoals blijkt op blz. 83, 84.

```
if es [successor] < es[ node ] + length [index] then
begin change:= true; pusher[successor] := node;
      es [successor] := es [node]  + length [index]
end;
comment pusher[e(P)]= b(P) <->
        es[e(P)] is het laatst verhoogd tijdens het in rekening
        brengen van pijl P;
```

(2) In de initialisatie wordt toegevoegd:

```
stop:= 0; pusher[START]:= stop;
```

(3) Na iedere iteratie wordt de test op positieve cykels uitgevoerd; deze werkt als volgt: men vormt een keten van knooppunten:
FINISH, pusher[FINISH], pusher[pusher[FINISH]],...
Deze keten kan op twee manieren eindigen:
a) men komt bij het knooppunt START, terwijl nog steeds geldt:
pusher START = stop (=0). In dit geval is er geen cykel gevonden.
b) in de keten komt eenzelfde element tweemaal voor. In dat geval bevat de "pusher-graaf" een cykel.*) Men kan bewijzen, (zie appendix C) dat het oorspronkelijke netwerk nu een positieve cykel bevat, welke door dezelfde knooppunten gaat als de cykel in de "pusher-graaf". Het algorithme kan nagaan, of een nieuw element van de keten van pushers, al eerder in de keten voorkomt, m.b.v. het boolean array "incp" (in critical path). Aan het begin van het onderzoek krijgt incp[i] de waarde: false, i=1,...,n.
Wanneer een knooppunt V aan de keten wordt toegevoegd, krijgt incp[V] de waarde: true.

---

*) Het begrip "pusher-graaf" wordt gedefinieerd in appendix C3.

Algol tekst:

```
for node:= 1 step 1 until n do incp[node] := false;
node:= FINISH;
repeat if incp[node]
       then begin REPORT POSITIVE CYCLE;
                  node:= stop; change:= false ; comment prevents next iteration
            end
       else begin incp[node] := true;
                  node:= pusher[node]
            end
until node = stop;
```

Indien aan de bereikbaarheidsvoorwaarden voldaan is, zal het bestaan van een positieve cykel uiteindelijk door dit algorithme gedetekteerd worden, zie 5.2.2.

Gewald en Sauler pogen de efficiency van hun algorithme nog op de volgende wijze te verbeteren:
veronderstel, dat es[V] niet veranderd is in de vorige iteratie; dan worden in de onderhavige iteratie, de uitgaande pijlen van V niet in rekening gebracht. Bij de door Gewald en Sauler gebruikte datastruktuur, waarbij de pijlen als punten-paren in een random volgorde in een lijst genoteerd staan, werkt deze maatregel echter nauwelijks efficiency-verhogend; want bij het in beschouwing nemen van een pijl P moet getest worden, of b(P) in de vorige iteratie onveranderd is gebleven. Deze test is vrijwel even kostbaar als het in rekening brengen zelf. Bovendien moet de test uitgevoerd worden voor iedere pijl. Dit kan zelfs leiden tot efficiency-verlaging!
Bij de door ons gebruikte data-struktuur echter, is er per knooppunt slechts één test nodig, en worden op grond van deze test alle uitgaande pijlen of in rekening gebracht, of overgeslagen. Bovendien kan de test als volgt verscherpt worden:

- de uitgaande pijlen van V worden slechts in rekening gebracht, indien zij sinds de laatste verandering van es[V] niet reeds allemaal in rekening gebracht zijn.

Afgezien van de efficiency-winst levert dit een eenvoudiger programma.
Een en ander is door ons in het gewijzigde Gewald-Sauler algorithme geïmplementeerd m.b.v. het boolean array "changed [1:n]". Nu geldt:

changed[i] $\leftrightarrow$ de uitgaande pijlen van knooppunt i moeten opnieuw in rekening worden gebracht.

Het Gewald-Sauler algorithme, zoals gepresenteerd in [5], tracht positieve cykels weg te werken, met de bedoeling, meer dan één positieve cykel te detekteren (indien aanwezig). Dit is eenvoudigheidshalve in onze representatie weggelaten.

Een kritiek pad kan worden gevonden door de reeks knooppunten:
FINISH, pusher [FINISH] , pusher[pusher[FINISH]],..., START.

Het gewijzigde Gewald-Sauler algorithme:

```
Initialisering:
      comment Wij gaan uit van een graaf,die aan de bereikbaarheids-
              voorwaarden voldoet;
      es[i]:= - ∞ , i=1,....,n;
      es[START]:= earliest start of project;
      stop:= 0; pusher[START]:= stop;
      changed[i]:= true , i=1,....,n;
Berekening:
      repeat change:= false ;
      for node:= 1 step 1 until n do
      begin if changed[node] then
        begin
          for index:= fe[node] step 1 until le[node] do
          begin successor:= suc[index];
Breng pijl in rekening:
            if es[successor] < es[node] + length[index] then
            begin change:= true; pusher[successor]:= node;
              es[successor]:= es[node] + length[index];
              changed[successor]:= true
            end
          end;
          changed[node]:= false   { alle uitgaande pijlen zijn in rekening gebracht }
        end
      end;
Zoek positieve cycle:
      for node:=1 step 1 until n do incp[node]:= false;
      node:= FINISH;
      repeat if incp[node] then
                begin REPORT POSITIVE CYCLE;
                  node:= stop; change:= false
                end
              else
                begin incp[node]:= true;
                  node:= pusher[node]
                end
      until node = stop;
      until not change;
```

5.2.2. *Efficiency*

Veronderstel, dat het netwerk geen positieve cykels bevat.

Zij a_1 het aantal iteraties van het Gewald-Sauler algorithmen voor de berekening van de vms-tijden. Zij a_2 het aantal iteraties van het Gewald-Sauler algorithme voor de berekening van de lts-tijden. In appendix B wordt bewezen:

$a_1 + a_2 \geq$ het aantal knooppunten van het kritieke pad 1)

Het Gewald-Sauler algorithme is dus *sterk-iteratief*.

Veronderstel, dat het netwerk wel positieve cykels bevat.

In dit geval wordt een bovengrens voor het aantal iteraties niet meer bepaald door de struktuur van de graaf, maar hangt zij sterk af van de lengten der pijlen.

Voorbeeld:

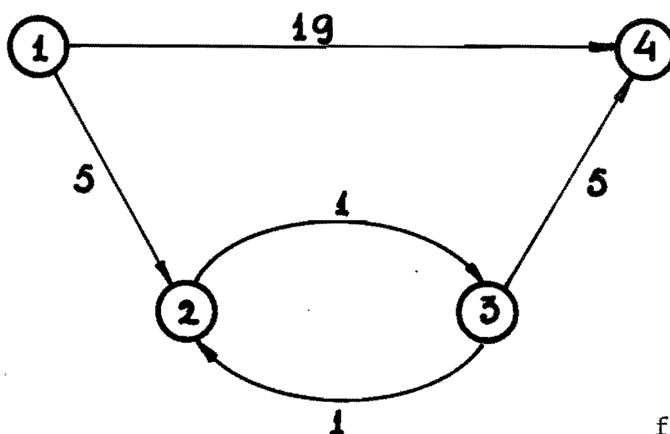


fig. 5.2.2.1

---

1) Dit wordt alleen bewezen, voor het geval er slechts één kritiek pad is. Het is redelijk, te verwachten dat bij aanwezigheid van meerdere kritieke paden het sterk-iteratieve karakter behouden blijft.

Er bevindt zich een positieve cykel in de graaf uit fig. 5.2.2.1. Er zijn 5 iteraties nodig, voordat deze cykel gaat behoren tot de "pusher-keten", welke begint bij FINISH. Het benodigde aantal iteraties voordat de positieve cykel ontdekt wordt verandert sterk, wanneer de lengtes van de pijlen veranderen.

Opmerking: Wij zijn er hierbij stilzwijgend vanuit gegaan, dat de graaf aan de bereikbaarheidsvoorwaarden voldoet. Indien dit niet het geval is, en er een positieve cykel bestaat, bereikbaar vanuit START maar waar vanuit geen weg naar FINISH bestaat, dan eindigt het Gewald-Sauler-algorithme niet.

5.3 *Kritische beschouwing.*

Het verschil in efficiency tussen algorithmen voor cykel-vrije netwerken en algorithmen voor niet-cykelvrije netwerken, is zeer markant. In de praktijk kan men nooit met zekerheid zeggen, of een netwerk cykel-vrij is; daarom is men gedwongen, altijd het algorithme voor niet-cykelvrije netwerken te gebruiken. Maar helaas is dit algorithme zeer ineffcient, zelfs als de aangeboden graaf cykelvrij is.
Van een goed algorithme zou men mogen verwachten, dat de rekentijd "evenredig" is met de "komplexiteit" van graaf. De rekentijd zou bij een cykelvrije graaf aan de in 5.1.2 genoemde ongelijkheid: $RT \leq \alpha_0 + \alpha_1 * n + \alpha_2 * m$ moeten voldoen. Planningsnetwerken uit de praktijk bevatten gewoonlijk wel cykels, maar hebben toch een zeer eenvoudige struktuur [zie 12.8]. De toename in de rekentijd t.o.v. het eerst genoemde geval zou dan ook gering moeten zijn.

6. *Een verbeterd algorithme*

6.1 *Het basis idee.*

In een verbeterd algorithme (voor cykelvrije en niet-cykelvrije) netwerken, zullen wij een *verwerkingsvolgorde der knooppunten* vastleggen, net zoals bij het algorithme voor cykelvrije netwerken. Deze verwerkingsvolgorde is bij een cykelvrije graaf identiek aan de volgorde, die het in 5.1 gepresenteerde algorithme voor cykelvrije grafen zou opleveren. Bij een niet-cykelvrije graaf zijn er bepaalde stukjes uit de volgorde, die iteratief doorlopen worden. Dit is in fig. 6.1.1 geschetst.

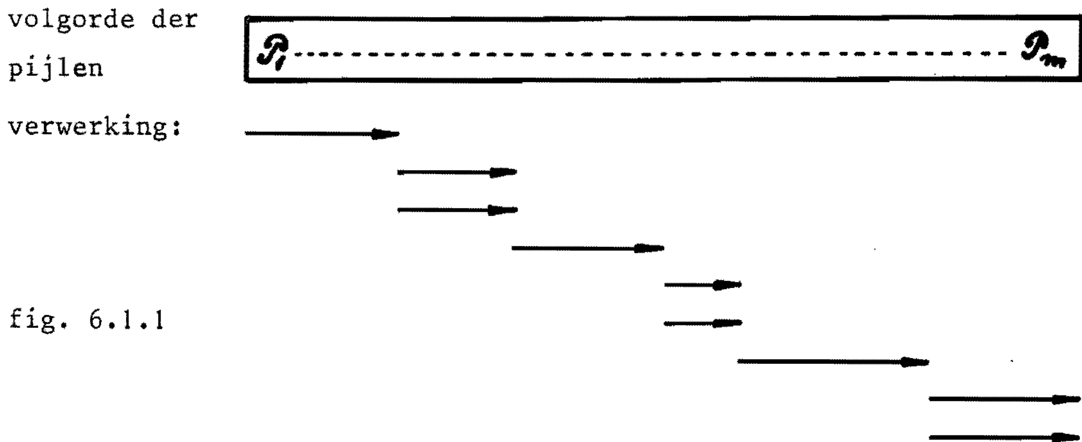


fig. 6.1.1

In fig. 6.2.2. is geschetst, hoe het Gewald-Sauler algorithme in een soortgelijk geval zou verlopen.

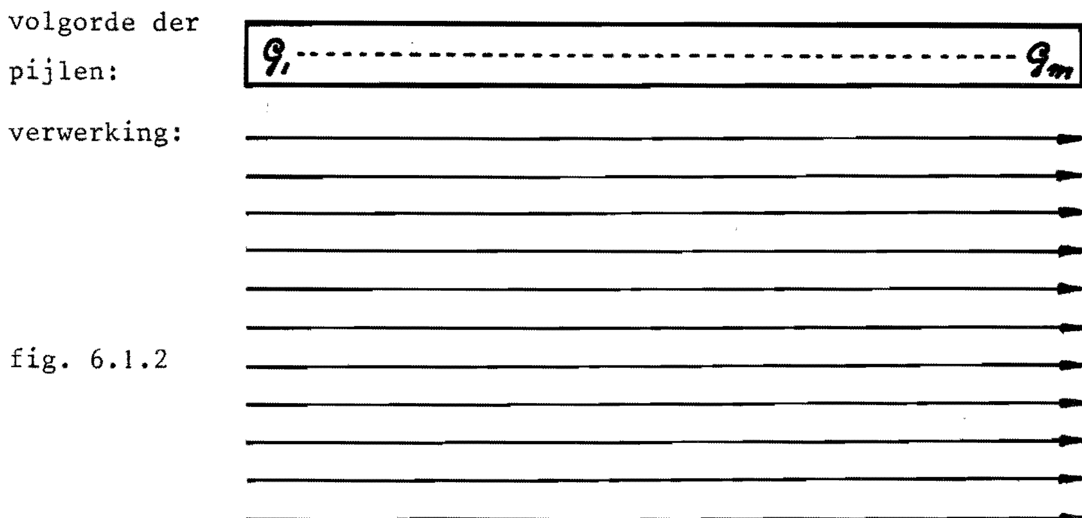


fig. 6.1.2

Een iteratie in ons algorithme betreft dus aanzienlijk minder pijlen dan een iteratie in het Gewald-Sauler algorithme. Mede daarom is onze werkwijze zeer effcient. Een nadere analyse van de effciency wordt gegeven in hoofdstuk 12.

6.2 *Een volgorde van sterke komponenten.*

Hierin wordt het deel-algorithme: "decompose, orden SC and check" beschreven. Dit deel-algorithme verricht *gelijktijdig* de volgende taken:

1) Het bepaalt de sterke komponenten van de graaf; dit is gebaseerd op het algorithme SC Worker [10].
2) Het onderzoekt, of de graaf aan de bereikbaarheidsvoorwaarden voldoet. Indien dit niet het geval is, stopt het algorithme, nadat een gedetailleerde foutmelding gegeven is.
3) Het bepaalt een volgorde van de sterke komponenten $SC_1, \ldots, SC_p$ en wel zodanig dat geldt:

$$SC_k \text{ bereikbaar vanuit } SC_h \rightarrow h < k$$

Deze volgorde kan men gebruiken bij de berekening van de vms-tijden: men berekent pas de vms-tijden van de knooppunten van SC_i, als de vms-tijden van de knooppunten van SC_{i-1} bekend zijn. Meer specifiek, verloopt deze berekening als volgt:

```
i:= 1; es[START]:= esproject;
repeat a) BEREKEN VMS-TIJDEN VAN KNOOPPUNTEN UIT SC_i;
          (bij deze berekening worden de pijlen P, waarvan b(P) en
          e(P) tot SC_i behoren, iteratief in rekening gebracht)
       b) BRENG PIJLEN IN REKENING VANUIT SC_i NAAR ANDERE SC's;
          (Deze berekening is niet-iteratief).
       c) i:= i + 1
until i > p;
```

Wanneer de graaf cykelvrij is, is (a) een loze statement, en verkrijgt men precies het algorithme uit 5.1 voor cykelvrije grafen.

Bij de berekening van de lts-tijden kan men op soortgelijke manier van de volgorde gebruik maken.

De volgorde van sterke komponenten berust op het concept van de *gereduceerde* graaf [10] *), die cykelvrij is.

Een gedetailleerde beschrijving wordt gegeven in hoofdstuk 8.

6.3 *Splitsing van de verzameling pijlen binnen één SC*

Men zou voor het berekenen van vms-tijden van knooppunten uit SC_i het algorithme van Gewald-Sauler kunnen gebruiken. Wij hebben echter gekozen voor een ander algorithme. Hiervoor is het nodig dat de verzameling pijlen gesplitst wordt in deelverzamelingen. Dit wordt verzorgd door het deel-algorithme "split arcs and orden nodes", dat hier globaal wordt beschreven. Dit deel-algorithme doet voor iedere sterke komponent SC_i het volgende:

- Er wordt een volgorde der knooppunten gekozen. In wezen kan deze volgorde willekeurig zijn; wij zullen later zien hoe deze volgorde de facto bepaald wordt. Wij zullen de knooppunten van SC_i in deze volgorde aanduiden met $V_1,\ldots,V_r$.
- Gebruik makend van deze volgorde wordt de verzameling uitgaande pijlen van de knooppunten van SC_i gesplitst in drie disjunkte deelverzamelingen:

1) FW_i (forward)

$$P \in FW_i \quad \leftrightarrow \quad \begin{cases} b(P) = V_h \\ e(P) = V_k \\ h < k \end{cases}$$

2) BW_i (backward)

$$P \in BW_i \quad \leftrightarrow \quad \begin{cases} b(P) = V_h \\ e(P) = V_k \\ h > k \end{cases}$$

3) ISC_i (inter SC)

$$P \in ISC_i \quad \leftrightarrow \quad \begin{cases} b(P) \in \{ V_1,\ldots,V_r \} \\ e(P) \notin \{ V_1,\ldots,V_r \} \end{cases}$$

*) Harrary C.S. spreken van een *gekondenseerde* graaf [6]

Merk op, dat de sterke komponent gesplist is in twee cykelvrije deelgrafen: de "FW-graaf" en de "BW-graaf".

Bij de enkelvoudige sterke komponent behoren alle uitgaande pijlen tot de verzameling ISC en behoeft dus geen expliciete splitsing meer plaats te vinden.

De merites van het splitsen van een sterke komponent in twee cykelvrije deelgrafen worden besproken in Appendix C. Hier wordt aangetoond, dat door een dergelijke splitsing, ongeacht hoe men deze ook doet , de bovengrens van de rekentijd een faktor '2' lager wordt. De merites van de door ons gekozen specifieke splitsing worden besproken in Hst 12. Dit gebeurt aan de hand van een aantal aannamen omtrent de struktuur van planningsnetwerken. Deze aannamen zijn gebaseerd op praktische ervaringen.

Een gedetailleerde beschrijving van dit deelalgorithme wordt gegeven in Hst 9.

6.4 *Het berekenen van de VMS tijden.*

Hierin wordt het deel-algorithme: "compute es, check and find cp" beschreven. Dit deel-algorithme verricht gelijktijdig de volgende taken:
- het bepaalt de vms-tijden;
- het kontroleert op positieve cykels;
- het bepaalt een kritiek pad.

Bij het bepalen van de vms-tijden en de kontrole op positieve cykels, wordt gebruik gemaakt van de in 6.2 en 6.3 genoemde volgorden. De sterke komponenten worden ieder éénmaal doorgerekend in de volgorde: $SC_1,\ldots,SC_p$.
Het doorrekenen van een sterke komponent houdt het volgende in:
- bij een enkelvoudige sterke komponent worden de uitgaande pijlen ieder éénmaal in rekening gebracht;
- bij een samengestelde sterke komponent, laat het algorithme zich als volgt globaal beschrijven:

<u>repeat</u> change := <u>false</u>;
SLAG 1: bereken VMS-tijden in FW-graaf
SLAG 2: bereken VMS-tijden in BW-graaf;
TEST : test op positieve cykels
<u>until</u> <u>not</u> change;
breng de ISC-pijlen eenmalig in rekening
van knooppunten binnen SCi

In SLAG1 en SLAG2 wordt gebruik gemaakt van de in 6.3 beschreven volgorde; deze deelstukken zijn beide volledig analoog aan de vms-berekeningen bij een cykelvrij-netwerk.
Wij zullen het doorrekenen van een cykelvrije deelgraaf een *slag* noemen. Twee opeenvolgende slagen brengen iedere pijl binnen de onderhavige SC éénmaal in rekening. Wij zullen twee opeenvolgende slagen een *iteratie* noemen, hoewel een iteratie in dit algorithme meestal aanzienlijk minder werk omvat dan een iteratie in het Gewald-Sauler algorithme. De bovengenoemde iteratie speelt zich immers af binnen één sterke komponent. E.e.a. wordt weergegeven door de korte en lange pijlen in resp. de figuren 6.1.1 en 6.1.2. De efficientie-winst zal groter zijn naarmate het netwerk bestaat uit een groter aantal sterke komponenten, m.a.w. naarmate de struktuur eenvoudiger is ("bijna cykelvrij"). Het is redelijk te veronderstellen, dat de struktuur van praktische planningsnetwerken meestal vrij eenvoudig zal zijn ("bijna, hoewel niet strikt cykelvrij"). Analoog aan het gewijzigde Gewald-Sauler algorithme, (zie 5.2) kan ook hier efficiency-winst verkregen worden, door het in rekening brengen van uitgaande pijlen van sommige knooppunten over te slaan. Wegens de splitsing in twee cykelvrije deelnetwerken, dient het kriterium een weinig anders te worden: het in rekening brengen van de pijlen uitgaande van een bepaald knooppunt kan worden overgeslagen, indien de VMS-tijd van dat knooppunt in de onderhavige en de vorige slag niet is gewijzigd.
De test op positieve cykels geschiedt op analoge wijze als in 5.2, door het aflopen van een keten van pushers: pusher[V], pusher[pusher[V]],... met dien verstande, dat dit geschiedt voor iedere $V \in SC_i$, voor zover:
1) het onderhavige pad ligt in SC_i
2) het onderhavige pad niet reeds eerder is onderzocht in dezelfde iteratie.
Het gevolg van deze werkwijze is, dat positieve cykels veel eerder ontdekt worden , dan wanneer men alleen de knooppunten van het kritieke pad onderzoekt.

De frequentie van testen op positieve cykels kan ook lager gekozen worden bv. om de 'q' iteraties (q>1). E.e.a. dient men te kiezen op grond van de te verwachten foutenfrequentie.
Indien een of meer positieve cykels zijn ontdekt wordt het programma na verloop van dit deel-algorithme afgebroken.
Een kritiek pad is te vinden door de reeks knooppunten:
FINISH, pusher[FINISH], pusher[pusher[FINISH]],...,START.
Een gedetailleerde beschrijving wordt gegeven in Hst 10.

6.5 *Het berekenen van lts tijden*

De berekening van de lts-tijden wordt verzorgd door het deel-algorithme "compute ls". Dit deelalgorithme wordt alleen uitgevoerd als het netwerk geen positieve cykels bevat. Het deel-algorithme "compute ls" is verder geheel analoog aan "compute es, check and find cp" met dienverstande dat:

- onder het in rekening brengen van een pijl P het volgende moet worden verstaan:
  <u>if</u> ls[e(P)]-length[P]<ls[b(P)]
  <u>then</u> ls[b(P)]:=ls[e(P)]-length[P]
- begonnen wordt met ls[FINISH] gelijk te stellen aan es[FINISH]
- de volgorde van de bewerkingen geheel omgekeerd is, dwz.:
  a) de sterke componenten worden in omgekeerde volgorde doorgerekend;
  b) bij een sterke komponent worden eerst de ISC-pijlen éénmalig in rekening gebracht; vervolgens worden bij een samengestelde sterke komponent, de pijlen van de cykel-vrije deelnetwerken in omgekeerde volgorde in rekening gebracht;
- het overslaan van het in rekening brengen van pijlen, is vanwege de gekozen datastruktuur niet efficient . Men zou dan namelijk per knooppunt rechtstreeks de beschikking moeten hebben over de inkomende pijlen. Hiertoe zou men op een andere datastruktuur moeten overgaan.

Per samengestelde sterke komponent laat het deelalgorithme zich als volgt beschrijven:

```
breng alle ISC-pijlen in rekening;
repeat change:= false;
SLAG 1: bereken lts-tijden in FW-graaf;
SLAG 2: bereken lts-tijden in BW-graaf;
until not change;
```

Een gedetailleerde beschrijving wordt gegeven in Hfst 11.

7. *Hoofdprogramma van het algorithme*

7.1 *Toelichting bij de algol-tekst*

Het hoofdprogramma wordt gevormd door de procedure: "EMPM Worker". Deze procedure maakt gebruik van de volgende, reeds eerder behandelde grootheden:

- de integers n en m, resp. het aantal knopen en pijlen van de graaf;
- de arrays fe, le, suc, length;
- de integers START en FINISH.

Daarnaast maakt het algorithme gebruik van de integer: "esproject" die de startdatum van het projekt representeert.

De output van de procedure wordt gevormd door de volgende grootheden:

- de boolean: "correct" geeft aan, of het netwerk aan de syntaktische korrektheidseisen voldoet. De overige uitvoergrootheden zijn alleen dan zinvol, als "correct" de waarde true heeft.
- de arrays "es" en "ls" geven per knooppunt de vms- resp. lts-tijden. Daarbij is "es[START] gelijk gemaakt aan "esprojekt", en is ls[FINISH] gelijk gemaakt aan es[FINISH]
- Het array: "pusher". Zij V een willekeurig knooppunt. Dan geeft "pusher [V]" de direkte voorganger van V aan op een langste pad van START naar V.
  Een kritieke pad wordt gevormd door de reeks knooppunten:
  FINISH, pusher[FINISH], pusher[pusher[FINISH]],..., START.
  De grootheid: "pusher[START]" heeft geen betekenis en krijgt in het programma de waarde '0'.
- Het array "holder". Zij V een willekeurig knooppunt. Dan geeft "holder [V]" de direkte opvolger van V aan op een langste pad van V naar FINISH. De grootheid holder [FINISH] heeft geen betekenis en heeft na het programma de waarde '0'.

De lokale arrays: lebw, lefw, size of SC, SClist, Vlist komen in de verschillende deel-algorithmen ter sprake. De integer "p" is gereserveeld voor het aantal sterke komponenten in de graaf; de integers: "i" en "V" zijn hulpvariabelen, en BIGM is een groot positief getal, groter dan de som van de absolute waarden van de lengtes van alle pijlen.

Het deel-algorithme: "split arcs and orden nodes" wordt alleen aangeroepen, als de graaf aan de bereikbaarheidsvoorwaarden voldoen, dwz. als "correct"

in "Decompose, orden SC and check" de waarde true heeft behouden. Het deelalgorithme "compute ls" wordt alleen aangeroepen, als het netwerk geen positieve cykels bevat. In de body van "compute ls" is de lts-tijd van ieder knooppunt V, dat op het kritieke pad ligt, gelijk gemaakt aan de vms-tijd. Hierbij wordt er voor gezorgd, dat de grootheid "holder [V]" de opvolger van V in het kritieke pad aangeeft.

7.2 *Algol-tekst*

```
procedure EMPM Worker( esproject,n,fe,le,m,suc,length,START,FINISH,
                       correct,es,ls,pusher,holder );
 value esproject,n,m,START,FINISH;
 integer esproject,n,m,START,FINISH;
 boolean correct;
 integer array fe,le,suc,length,es,ls,pusher,holder;
 begin
   integer array lebw,lefw,size of SC,SClist,Vlist[1:n];
   integer i,V,  p,BIGM;
   declaration page 40: procedure Decompose orden SC and check,
               page 53: procedure Split arcs and orden nodes,
               page 66: procedure Compute es check and find cp,
               page 74: procedure Compute ls;
   correct:= true; BIGM:= ⏨ 6;
   Decompose orden SC and check;
   if correct then
     begin Split arcs and orden nodes;
       for i:=1 step 1 until n do es[i]:= - BIGM;
       es[START]:= esproject;
       Compute es  check and find cp;
       if correct then
         begin for i:=1 step 1 until n do ls[i]:= BIGM;
           Compute ls
         end
     end
 end;
```

8. *Het deel-algorithme: "Decompose, orden SC and check"*

8.1 *Beschrijving.*

Dit deelalgorithme verricht gelijktijdig de volgende taken:

- het plaatst van iedere SC één knooppunt (de wortel van deze SC) in het array SClist;
- het onderzoekt of de graaf aan de bereikbaarheidsvoorwaarden voldoet. Dit zullen wij allereerst beschrijven.

De eis, dat alle knooppunten vanuit START bereikbaar moeten zijn, kan als volgt m.b.v. SC-Worker worden nagegaan: men begint het hoofdprogramma met "extend (start)"; als(en alleen als)na terugkeer uit "extend(start)" geldt :NEW = $\emptyset$,is aan de eis voldaan.

De eis, dat FINISH vanuit alle knooppunten bereikbaar moet zijn, kan als volgt m.b.v. SC Worker worden nagegaan:

a) Declareer het boolean array FR[1:n]   (Finish Reachable)
b) Initialiseer FR[i]:= <u>false</u>, i=1,...,n; FR[finish]:= <u>true</u>;
c) Voeg toe, na het afwerken van een pijl binnen "extend(V)":
```
   (stap 5)  if FR[successor] then FR[V]:= true
```
d) Voeg toe, bij het ontdekken van een SC binnen "extend(V)":
```
   (stap 6)  if not FR[V]
             then begin rapporteer, dat FINISH niet bereikbaar is vanuit de
                        knooppunten van de onderhavige SC:
                        for i:= plinstack step 1 until ploftop do print(STACK[i])
                        correct:= false
                  end
             else  for i:= plinstack step 1 until ploftop do
                   FR[i]:= true
```
Indien aan één van de bereikbaarheidsvoorwaarden niet is voldaan, krijgt de boolean variabele "correct" de waarde <u>false</u>.

Bij het ordenen van de sterke komponenten, wordt een sterke komponent SC_i gerepresenteerd door het eerste knooppunt van SC_i, dat in 'STACK' komt.

Wanneer een SC wordt ontdekt in "extend(V)", worden de volgende statements uitgevoerd:

```
if correct then
begin p:= p + 1;
  comment p telt het aantal sterke componenten,dat reeds is gevonden;
  SClist[p]:= V;
  size of SC[p]:= ploftop - plinstack + 1;
  comment size of SC[p] bevat het aantal knooppunten, dat tot de sterke
          komponent " p " behoort;
end;
```

Indien aan de bereikbaarheidsvoorwaarden voldaan is, geldt het volgende:

- de waarde van de variabele p geeft nu het aantal sterke komponenten van de gra
- de SC, waartoe FINISH behoort, SC_p, wordt het eerst ontdekt, en wordt genoteerd in SClist[1];
- de SC, waartoe START behoort, SC_1, wordt het laatst ontdekt, en wordt genoteerd in SClist[p];
- de sterke komponenten $SC_1,\ldots,SC_p$ worden in omgekeerde volgorde genoteerd in het array SClist.

Indien niet aan de bereikbaarheidsvoorwaarden voldaan is, kan men de knooppunten indelen in vier klassen:

- SFR: liggend op een pad van START naar FINISH;
- SR : bereikbaar vanuit START, geen pad naar FINISH;
- FR : niet bereikbaar vanuit START, wel een pad naar FINISH;
- R : niet bereikbaar vanuit START, geen pad naar FINISH.

Alle knooppunten van een sterke komponent behoren tot dezelfde klasse. Wij kunnen de sterke komponenten in soortgelijke klassen indelen. De foutenrapportage is als volgt: iedere sterke komponent, die tot een van de laatste drie klassen behoort, wordt gerapporteerd door:

- een aanduiding van de klasse, waartoe deze SC behoort;
- een aanduiding van de knooppunten, die tot deze SC behoren.

Een bewijs van korrektheid van het deel-algorithme wordt gegeven in Appendix C1.

8.2 *Voorbeeld*

Als voorbeeld gebruiken wij de graaf uit fig. 8.1.1; dezelfde graaf werd ook als voorbeeld gebruikt bij de beschrijving van het algorithme SC-Worker (zie [10], blz 9 en 10). Op blz.39 zijn de waarden weergegeven die de nieuwe grootheden van het algorithme aannemen; voor de waarden van stack[i] en badge[i] zie men [10] blz. 10. Eenvoudigheidshalve is het array "size of SC" niet weergegeven.

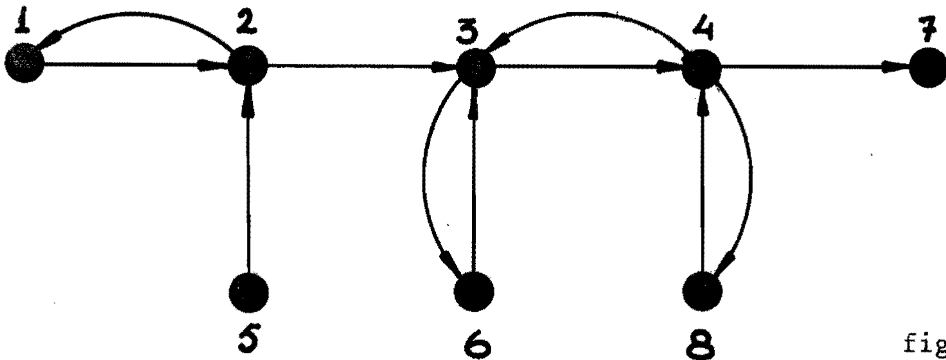


fig. 8.1.1

Wij nemen aan, dat de gebruiker heeft gespecificeerd:
START = 1 en FINISH = 7. Merk op, dat daardoor t.a.v. knooppunt 5 aan één van de bereikbaarheidsvoorwaarden niet is voldaan.

De tabel op pag. 39 is gebaseerd op de tabel op pag. 10 van [10]. De stapnummers refereren aan de stapnummers uit [10].
In de tabel geeft een plus-teken aan, dat een boolean de waarde true heeft, en een min-teken, dat deze de waarde false heeft.
De boolean "V is reachable from start" heeft de waarde true zolang de onderhavig sterke component bereikbaar is vanuit START.

stap	pijl	FR								p	SClist				correct	*)
		[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]		[1]	[2]	[3]	[4]....		
init.		-	-	-	-	-	-	+	-	0					+	+
0,1,2								+		0					+	+
3,4,2	1→2							+		0					+	+
3,4,5	2→1							+		0					+	+
3,4,2	2→3							+		0					+	+
3,4,2	3→4							+		0					+	+
3,4,2	4→8							+		0					+	+
3,4,5	8→4							+		0					+	+
3,6,7,5								+		0					+	+
3,4,2	4→7							+		0					+	+
3,6,7,5					+			+		1	7				+	+
3,4,5	4→3				+			+		1	7				+	+
3,6,7,5				+	+			+		1	7				+	+
3,4,2	3→6			+	+			+		1	7				+	+
3,4,5	6→3			+	+			+		1	7				+	+
3,6,7,5				+	+			+		1	7				+	+
3,6,7,5			+	+	+		+	+	+	2	7	3			+	+
3,6,7,5		+	+	+	+		+	+	+	2	7	3			+	+
3,6,7,0,1,2		+	+	+	+		+	+	+	3	7	3	1		+	-
3,4,5	5→2	+	+	+	+	+	+	+	+	3	7	3	1		+	-
3,6,7,0		+	+	+	+	+	+	+	+						-	-

*) V is reachable from start

8.3 *Toelichting bij het algol-programma*

De kern van het algol-programma bestaat uit het algorithme SC Worker, zoals gepresenteerd in [10]. Statements, die betrekking hebben op het onderzoeken van de bereikbaarheidsvoorwaarden, zijn gelabeled met CHECKn, waarin n een integer is. Statements, die betrekking hebben op het ordenen van de sterke componenten zijn gelabeled met ORDENn waarin n een integer is.

8.4 *Algol programma*

```
procedure Decompose orden SC and check;
  begin integer i,new,insc,ploftop;
    integer array stack,badge[1:n];
    boolean V is reachable from start;
    boolean array FR[1:n];

    procedure extend( V ); value V; integer V;
      begin integer index,successor,plinstack;
        declaration procedure REPORT;
    PUT V ON STACK:
        badge[V]:=plinstack:=ploftop:=ploftop + 1; stack[ploftop]:= V;
    SCAN SUCCESSOR 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];
    CHECK5:
          if FR[successor] then FR[V]:= true
        end;
    CHECK WHETHER SC IS FOUND:
        if badge[V] = plinstack then
          begin for index:= plinstack step 1 until ploftop do
                badge[stack[index]]:= insc;
    CHECK6:
            if FR[V] then
              for index:= plinstack step 1 until ploftop do
              FR[stack[index]]:= true;
    CHECK7:
            if not FR[V] or not V is reachable from start then
              begin correct:= false; REPORT end;
    ORDEN2:
            if correct then
              begin p:= p + 1; size of SC[p]:= ploftop - plinstack + 1;
                sclist[p]:= V
              end correct;
            ploftop:= plinstack - 1;
          end SC found
      end extend;
```

```
INITIATE:
    new:= - 1; insc:= n + 1; ploftop:= 0;
    for i:=1 step 1 until n do badge[i]:= new;
ORDEN1:
    p:= 0;
CHECK1:
    for i:=1 step 1 until n do FR[i]:= false;
CHECK2:
    FR[FINISH]:= true;
CHECK3:
    V is reachable from start := true;  extend( START );
CHECK4:
    V is reachable from start := false;
    for i:=1 step 1 until n do if badge[i] = new then extend( i );
  end Decompose orden SC and check;

procedure REPORT;
  begin integer index; NLCR;
    PRINTTEXT(≮ THE FOLLOWING VERTICES ARE ≯);
    if not FR[V] and not V is reachable from start then
      PRINTTEXT(≮ NEITHER REACHABLE FROM START;NOR IS FINISH REACHABLE FROM THEM
    else
      if FR[V] then
        PRINTTEXT(≮ NOT REACHABLE FROM START,BUT FINISH IS REACHABLE FROM THEM≯)
      else
        PRINTTEXT(≮ REACHABLE FROM START,BUT FINISH IS NOT REACHABLE FROM THEM≯)
    NLCR;
    for index:= plinstack step 1 until ploftop do
      PRINT( stack [ index ])
  end REPORT;
```

9. *Het deelalgorithme: "split arcs and orden nodes"*

9.1 *Beschrijving*

Het deel-algorithme: "split arcs and orden nodes" wordt alleen aangeroepen, als de graaf aan de bereikbaarheidsvoorwaarden voldoet. Het maakt gebruik van de volgende grootheden, die output zijn van "decompose, orden SC and check":

p : het aantal sterke komponenten in de graaf

SClist [1:n] : bevat van iedere SC één knooppunt, zodanig dat: SClist[i] bereikbaar vanuit SClist[j] $\rightarrow$ $i < j$

size of SC [1:n] : bevat op plaats i het aantal knooppunten van de SC, waartoe, knooppunt SClist[i] behoort, $1 \leq i \leq p$

Opmerking: de SC, waartoe knooppunt SClist[i] behoort, werd in hoofdstuk 8 gedefinieerd als SC_{p-i+1}

Zoals in hoofdstuk 6 is beschreven, bepaalt het deel-algorithme voor iedere samengestelde sterke komponent een ordening der knooppunten, en per sterke komponent SC_i een splitsing van de pijlen in drie verzamelingen: FW_i, BW_i en ISC_i.

Tevens vult het deel-algorithme het array "Vlist" met knooppuntnotities, en wel als volgt:

- allereerst worden de knooppunten van SC_p geordend, vervolgens de knooppunten van SC_{p-1}, etc. Wanneer wij uitgaan van de knooppunt-notities in SClist, betekent dit: allereerst de knooppunten van de SC, waartoe knooppunt SClist[1] behoort, en als laatste de knooppunten van de SC, waartoe SClist[p] behoort
- de knooppunten van SC_p worden, in een nader te bepalen volgorde, genoteerd op de plaatsen $n-r_p+1,\ldots,n$ (Hierin is r_p: het aantal knooppunten van SC_p, d.w.z. r_p=size of SC[1]). In het algemeen worden de knooppunten van SC_i genoteerd op de plaatsen:

$$n - \sum_{j=i}^{p} r_j + 1, \ldots\ldots\ldots\ldots, n - \sum_{j=i+1}^{p} r_j$$

Een en ander wordt schematisch weergegeven in fig. 2.1.1. Vanwege de overzichtelijkheid is het array SClist van rechts naar links opgeschreven.

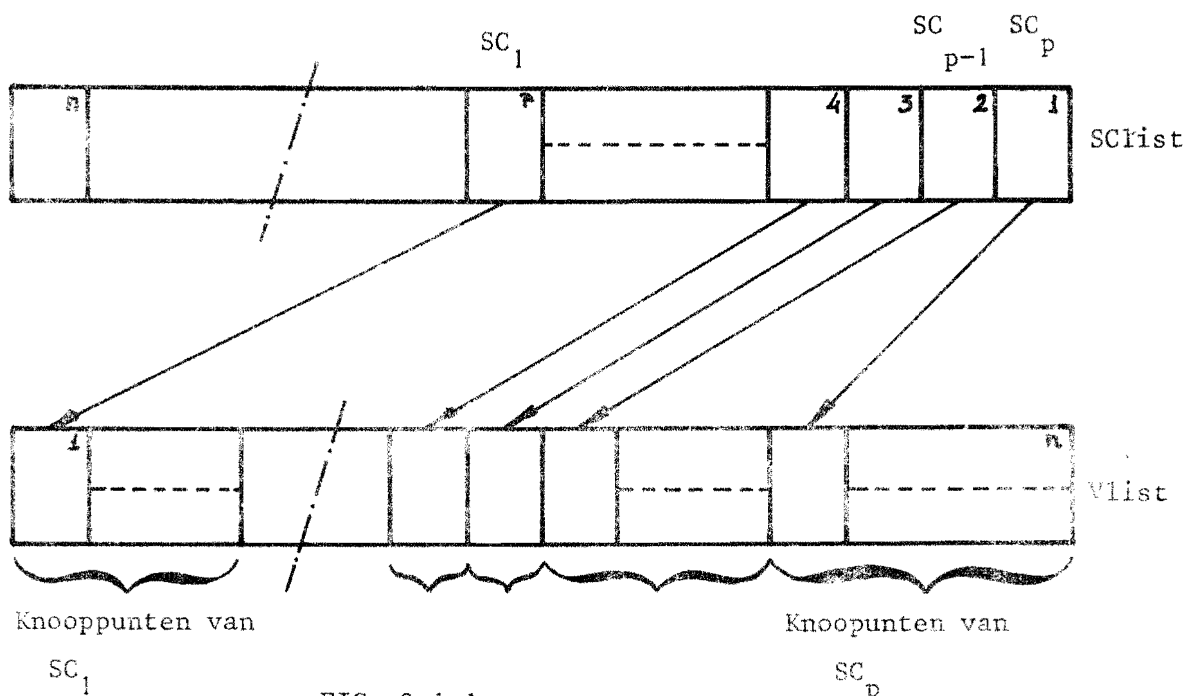


FIG. 9.1.1.

Simultaan met het ordenen van de knooppunten van een samengestelde sterke component SC_i, wordt de verzameling uitgaande pijlen van knooppunten van SC_i gesplitst in de deelverzamelingen FW_i, BW_i en ISC_i.

Wij zullen nu de procedure beschrijven, die het ordenen van de knooppunten van één samengestelde sterke komponent verzorgt, alsmede het splitsen van de verzameling uitgaande pijlen van die knooppunten. Van een sterke komponent SC_i staat één knooppunt genoteerd in het array SClist op plaats p-i+1. Wij zullen dit knooppunt de *wortel* van SC_i noemen, en aanduiden met W_i.

De procedure gaat ervan uit, dat:

- aan het begin van het deel-algorithme alle knooppunten de markering "UNSPLIT" hebben gekregen; de variabele "bottom" dient de waarde n+1 te hebben gekregen
- knooppunten, die wel bereikbaar zijn vanuit W_i, maar tot een andere sterke komponent behoren, de markering "READY" hebben.

- de verzameling knooppunten van SC_i derhalve gevormd wordt door de knooppunten,die vanuit W_i bereikbare zijn, *en niet* de markering READY hebben.

De procedure vormt, uitgaande van W_i, een simpel pad binnen SC_i; dit pad, het zgn. *"aktuele pad"*, wordt voortdurend uitgebreid en weer ingekrompen. Op ieder moment hebben de knooppunten van het aktuele pad de markering "SPLITTING", d.w.z. het splitsen van de verzameling uitgaande pijlen van die knooppunten is begonnen, maar nog niet voltooid. De knooppunten van SC_i, die reeds eerder tot het aktuele pad behoorden (maar er weer uit zijn verdwenen, ten gevolge van het inkrimpen) hebben de markering "SPLIT", d.w.z. het splitsen van de verzameling uitgaande pijlen is voltooid. Deze knooppunten zijn inmiddels genoteerd in het array "Vlist".

De volgorde, waarin de knooppunten van SC_i in het array Vlist komen, is omgekeerd aan de volgorde, waarin zij van het aktuele pad worden verwijderd, omdat het array Vlist "van boven af aan" wordt gevuld. Wanneer V een opvolger is van U in een aktueel pad, komt de knooppuntnotitie van V op een hogere plaats in Vlist dan de knooppuntnotitie van U. Een pijl (U, V) komt in FW_i en een pijl (V, U) in BW_i.

Zij *V* het laatste knooppunt van het aktuele pad. Achtereenvolgens worden nu de uitgaande pijlen van *V* afgewerkt:

a) een pijl P naar een knooppunt U met markering "SPLITTING", (dwz.: een voorganger in het aktuele pad) geeft aanleiding tot de statement:
   - VOEG P TOE AAN VERZAMELING "BW_i"

b) een pijl P naar een knooppunt U met markering "SPLIT", geeft aanleiding tot de statement:
   - VOEG P TOE AAN "FW_i"

c) een pijl P naar een knooppunt U met markering "UNSPLIT" geeft aanleiding tot de statements:
   - VOEG P TOE AAN "FW_i";
   - BREIDT HET AKTUELE PAD UIT MET KNOOPPUNT U;

d) een pijl P naar een knooppunt U met markering "READY" geeft aanleiding tot de statement:
   - VOEG P TOE AAN "ISC_i";

Nadat alle uitgaande pijlen van *V* zijn afgewerkt, worden de volgende statements uitgevoerd:

- GEEF *V* DE MARKERING: "SPLIT";

- bottom:= bottom-1; Vlist[bottom]:= *V*;
- VERWIJDER *V* UIT HET AKTUELE PAD;

De deelgraaf FW_i heeft de volgende eigenschap: alle knooppunten van SC_i zijn, binnen de deelgraaf FW_i bereikbaar vanuit knooppunt W_i. In hoofdstuk 12 zal aannemelijk gemaakt worden, dat deze eigenschap van de FW-deelgraaf gewoonlijk efficiency-verhogend werkt, terwijl er geen bezwaren aan zijn verbonden.

Het hierboven beschreven algorithme is geïmplementeerd m.b.v. de rekursieve procedure: "split and orden (*V*)". Door het aanroepen van "split and orden (*V*)", wordt het aktuele pad met *V* uitgebreid, door de terugkeer wordt *V* weer uit het aktuele pad verwijderd.
In het hoofdprogramma is de aanroep van "split and orden (W_i)" voldoende voor het ordenen van de knooppunten van SC_i en het splitsen van de verzameling uitgaande pijlen van die knooppunten. Zeer globaal ziet de procedure "split and orden (*V*)" er als volgt uit:

```
procedure Split and orden ( V ); value V; integer V;
  begin integer index,successor;
    GEEF V DE MARKERING "SPLITTING" ;
    index:= fe[V];
    while index ≤ le[V] do
    begin NOEM DE PIJL OP PLAATS index: "P";
      successor:= suc[index];
      if successor ε "UNSPLIT" then
        begin VOEG P TOE AAN FWi; Split and orden ( successor ) end
      else
        if successor ε "SPLITTING" then
          VOEG P TOE AAN BWi
        else
          if successor ε "SPLIT" then
            VOEG P TOE AAN FWi
          else
            VOEG P TOE AAN FSCi;
      index:= index + 1
    end;
    GEEF V DE MARKERING: "SPLIT";
    bottom:= bottom - 1; Vlist[bottom]:= V;
  end;
```

De globale variabele: "bottom" geeft de plaats in het array Vlist aan, waar de laatste knooppuntnotitie is weggeschreven. De implementatie van het geven van markeringen aan knooppunten, en van het splitsen van de verzameling pijlen, wordt beschreven onder 9.3.

Het hoofdprogramma van het deel-algorithme: "split and orden nodes" is als volgt:

```
GEEF ALLE KNOOPPUNTEN DE MARKERING "UNSPLIT" ;
bottom:= n + 1;
for i:=1 step 1 until p do
begin V:= SClist[i];
  if size of SC[i] = 1 then
    begin bottom:= bottom - 1; Vlist[bottom]:= V;
      GEEF V DE MARKERING: "READY";
      VOEG ALLE UITGAANDE PIJLEN VAN V TOE AAN ISCi
    end
  else
    begin Split and orden ( V );
      GEEF ALLE KNOOPPUNTEN MET MARKERING: "SPLIT" DE MARKERING: "READY"
    end
end;
```

9.2 *Voorbeeld*

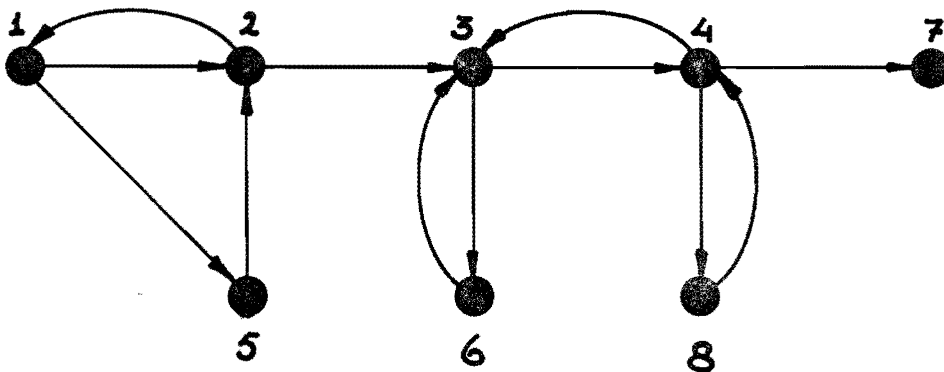


fig. 9.2.1.

Als voorbeeld gebruiken wij opnieuw de graaf uit fig. 8.1 waaraan echter de pijl (1,5) is toegevoegd, om de graaf aan de bereikbaarheidsvoorwaarden te laten voldoen. Na initialisatie van het deel-algorithme: "split arcs and orden nodes", hebben de verschillende grootheden de waarden, die hieronder zijn weergegeven:

	1	2	3	4	5	6	7	8
SClist[.]	7	3	1					
size of SC[.]	1	4	3					
status[.]	u	u	u	u	u	u	u	u

p = 3
bottom = n + 1 = 9

T.a.v. de status worden in de tabellen de volgende afkortingen gebruikt:

u = unsplit
sg = splitting
st = split
r = ready

In 9.3 wordt de werkelijke implementatie besproken.
Op blz. 48 is het verloop van het algorithme voor bovenstaande graaf geschetst.
De volgende momenten in het deelalgorithme worden aangegeven:

- direkt na de indeling van een pijl
- direkt na uitbreiding van het aktuele pad en het onderzoek van de eerste uitgaande pijl van het knooppunt waarmee het pad zojuist is uitgebreid
- direkt na inkrimping van het aktuele pad, aangegeven door $\not{i}$, waarbij i zojuist van het aktuele pad verwijderd is.

V	pijl	bot-tom	status[.]								Vlist[.]								Beschouwde uitgaande
			[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	pijl(en)van V gaan na
-	-	9	u	u	u	u	u	u	u	u	-	-	-	-	-	-	-	-	
7	-	8							r									7	
3	3-4				sg														FW
4	4-3				sg	sg													BW
	4-7																		ISC
	4-8																		FW
8	8-4									sg									BW
~~8~~		7								st							8	7	
~~4~~		6			sg	st			r	st						4	8	7	
3	3-6																		FW
6	6-3				sg	st		sg	r	st									BW
~~6~~		5			sg	st		st	r	st					6	4	8	7	
~~3~~		4			st	st		st	r	st				3	6	4	8	7	
3,6,4,8					r	r		r	r	r									
1	1-2		sg		r	r		r	r	r									FW
2	2-1		sg	sg	r	r		r	r	r									BW
	2-3																		ISC
~~2~~		3	sg	st	r	r		r	r	r			2	3	6	4	8	7	
1	1-5																		FW
5	5-2		sg	st	r	r	sg	r	r	r									FW
~~5~~		2	sg	st	r	r	st	r	r	r		5	2	3	6	4	8	7	
~~1~~		1	st	st	r	r	st	r	r	r	1	5	2	3	6	4	8	7	
1,2,5			r	r	r	r	r	r	r	r									

Het resultaat van het deelalgorithme kan m.b.v. fig. 9.2.2 geïllustreerd worden:

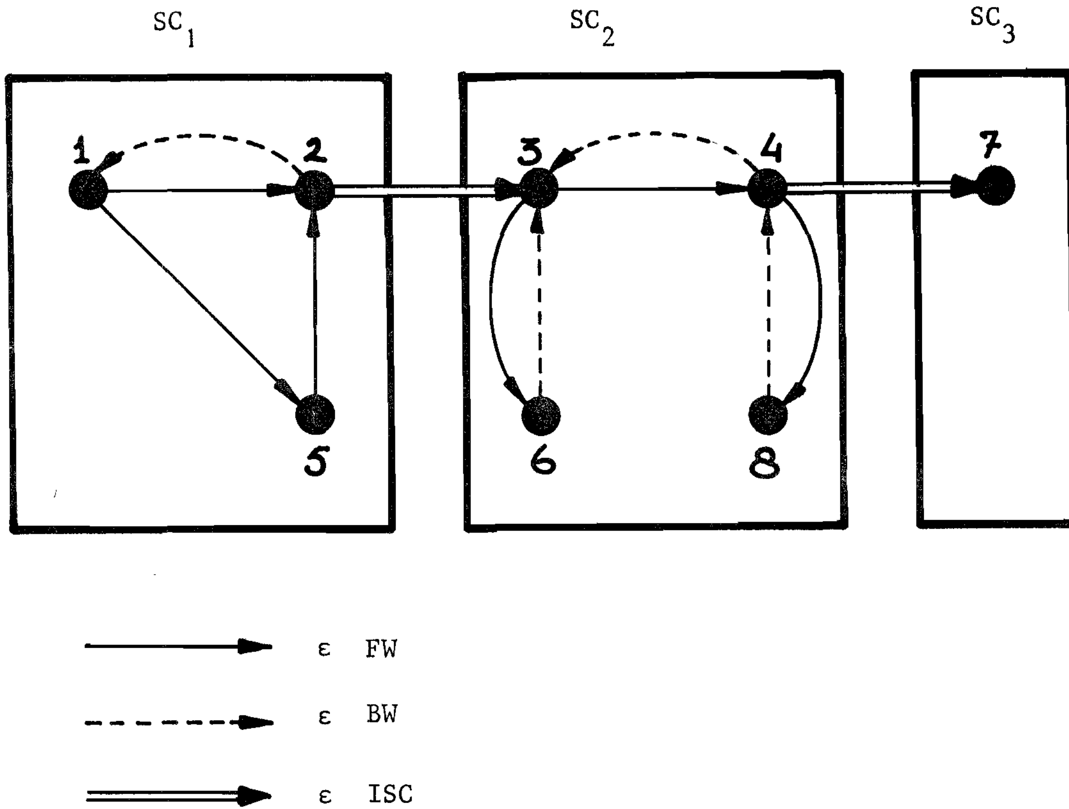


fig. 9.2.2.

9.3 *Toelichting bij het algol-programma*

Wij beschrijven eerst, hoe het splitsen van de verzameling uitgaande pijlen is geïmplementeerd; de uitgaande pijlen van knooppunt V worden zó herordend, dat:

- de pijlen uit het BW-netwerk in de arrays "suc" en "length" staan op de plaatsen: fe[V] tm lebw[V] (last edge backward)
- de pijlen van het FW-netwerk staan op de plaatsen: lebw[V]+1 tm lefw[V] (last edge forward arc)
- de pijlen "tussen" de sterke komponenten staan op de plaatsen: lefw[V]+ 1 tm le[V]

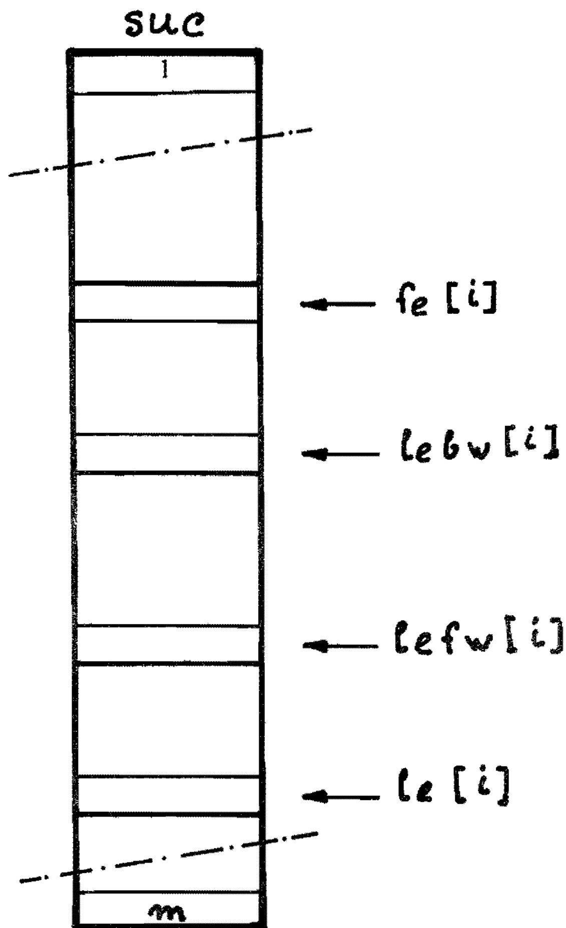


fig. 9.3.1.

T.b.v. dit hersorteren wordt gebruik gemaakt van de procedure"interchange arcs(p,q)", welke de pijl op plaats p en de pijl op plaats q verwisselt.
De statements: "VOEG PIJL P TOE AAN VERZAMELING X" worden nu:

Initialiseer: lebw[i]:= fe[i]-1, lefw[i]:= le[i], i = 1,...,n;

Pijl P gaat naar BW : lebw[V]:= lebw[V]+ 1;
interchange arcs (lebw[V], index);

Pijl P gaat naar FW : -

Pijl P gaat naar ISC: interchange arcs (lefw[V], index);
lefw[V]:= lefw[V]-1;
index:= index-1;

De laatste statement : "index := index -1" is nodig, omdat in de while-statement, waarin "index" de "lopende" variabele is, de waarde van "index" met één verhoogd wordt, waardoor de nieuwe, "tussengeschoven" pijl zou worden overgeslagen!

Index behoeft niet tot aan "le[V]" te lopen, maar slechts tot en met "lefw[V]", aangezien de pijlen op plaats "lefw[V] + 1,.... le[V]" reeds eerder zijn beschouwd.

Tenslotte beschrijven wij onze implementatie van het markeren van knooppunten:

V∈ " UNSPLIT" ↔ status [V] = unsplit (=1)
V∈ " SPLITTING" ↔ status [V] = splitting (=2)
V∈ " SPLIT" ↔ status [V] = split (=3)
V∈ " READY" ↔ status [V] = ready (=4)

Bovengenoemde implementatie is erg simpel. Het programma blijft hierdoor goed leesbaar. Er is echter ook een andere implementatie mogelijk, waarbij enige efficiency-winst geboekt kan worden. Deze implementatie is als volgt:

V∈ " UNSPLIT" ↔ status [V] = unsplit (=1)
V∈ "SPLITTING" ↔ status [V] = splitting (=2)
V∈ "SPLIT" ↔ status [V] = split (>3)
V∈ "READY" ↔ splitting < status [V] < split

Deze implementatie komt de lezer op het eerste gezicht wellicht merkwaardig voor. De reden, welke aan deze implementatie ten grondslag ligt, is dat de statement: "GEEF ALLE KNOOPPUNTEN MET MARKERING : "SPLIT" , NU DE MARKERING : "READY"; nu kan worden geïmplementeerd door: "split:= split + 1;" ! Tijdens het doorrekenen van een bepaalde SC verandert "split" niet van waarde.

Kijkend naar deze ene statement lijkt de efficiency-winst groot, namelijk evenredig met het aantal knooppunten van de sterke komponent. In het totaal van "split and order (V)" is de winst echter klein. De faktor die de efficiency-verhouding aangeeft, is een konstante, omdat de rekentijd van split and order (V) in beide gevallen evenredig is met het aantal knooppunten van de sterke komponent. Deze faktor ligt in de orde van grootte van 0.90.

9.4 *Algol programma*

```
procedure Split arcs and orden nodes;
  begin integer i,j,unsplit,splitting,split,ready,bottom;
    integer array status[1:n];
    declaration procedure Split and orden (V);
INITIATE:
    for i:=1 step 1 until n do
      begin lebw[i]:= fe[i] - 1; lefw[i]:= le[i] end ;
    unsplit:= 1; splitting:= 2; ready:= 3; split:= 4; bottom:= n + 1;
    for i:=1 step 1 until n do status[i]:= unsplit;
MAIN PROGRAM:
    for i:=1 step 1 until p do
      begin V:= SClist[i];
        if size of SC[i] = 1 then
          begin bottom:= bottom - 1; Vlist[bottom]:= V;
            lefw[V]:= fe[V] - 1
          end
        else Split and orden( V );
PUT ALL VERTICES OF SCi INTO SET READY:
        for j:= bottom step 1 until bottom + size of SC[i] - 1 do
          status[Vlist[j]]:= ready;
      end
  end Split arcs and orden nodes;
```

```
procedure Split and orden( V ); value V; integer V;
  begin integer index,successor;
    declaration procedure interchange arcs (p,q);
PUT V INTO ACTUAL PATH: status[V]:= splitting;
    index:= fe[V];
    while index ≤ lefw do
    begin successor:= suc[index];
      if SUCCESSOR BELONGS TO UNSPLIT: status[successor] = unsplit then
        comment arc must goto FW, i.e. do nothing;
        Split and orden( successor )
      else
ARC TO RIGHT SET:
        if SUCCESSOR BELONGS TO SPLITTING: status[successor] = splitting then
          begin
ARC TO BW:
            lebw[V]:= lebw[V] + 1;
            interchange arcs( lebw[V],index )
          end
        else
          if SUCCESSOR BELONGS TO READY: status[successor] = ready then
            begin
ARC TO ISC:
              interchange arcs( lefw[V],index );
              lefw[V]:= lefw[V] - 1; index:= index - 1
            end
          else
            SUCCESSOR BELONGS TO SPLIT:
            comment arc must go to FW, i.e. do nothing;
     index:= index + 1
  end while-statement ;
 PUT V INTO SPLIT: status[V]:= split;
     bottom:= bottom - 1;
     Vlist[bottom]:= V
  end Split and orden;

 procedure interchange arcs( p,q ); value p,q; integer p,q;
   begin integer help;
     help:= suc[p]; suc[p]:= suc[q]; suc[q]:= help;
     help:= length[p]; length[p]:= length[q]; length[q]:= help
   end;
```

10. *Het deel-algorithme: "compute es, check and find cp"*

10.1. *Beschrijving*

Het deel-algorithme "compute es, check and find cp" berekent de vms-tijden van de knopen, onderzoekt of het netwerk geen positieve cykels bevat, en berekent een kritiek pad.
Het maakt gebruik van de ordening van het array "Vlist" en van de inhoud van het array "size of SC". De relatie, die deze grootheden met elkaar hebben, is hieronder weergegeven.

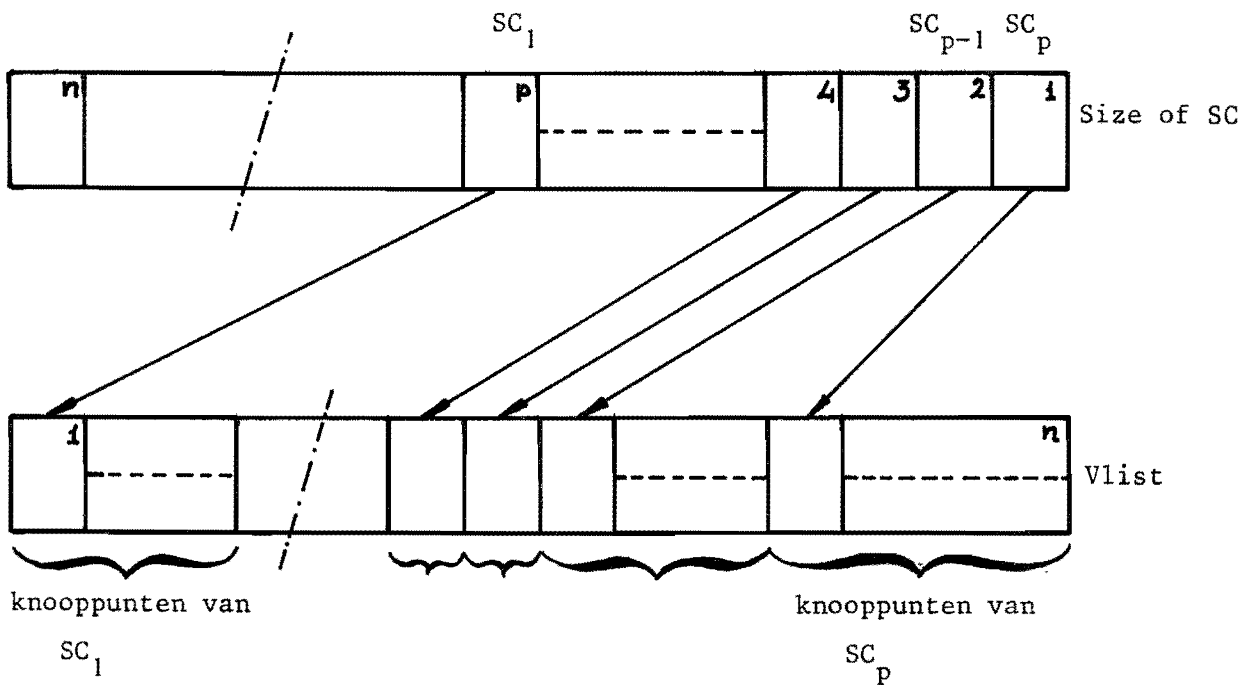


fig. 10.1.1.

Het algorithme werkt de sterke komponenten af in de volgorde SC_1, ... SC_p. Indien de sterke komponent slechts uit één knooppunt bestaat, worden de uitgaande pijlen van dit knooppunt éénmaal in rekening gebracht. Indien de sterke komponent samengesteld is, wordt eerst de procedure: "compute es of SCi" aangeroepen. Deze procedure brengt de pijlen van FWi en BWi beurtelings in rekening, totdat de vms-tijden van de knooppunten van SCi hun definitieve waarden hebben of positieve cykels zijn gevonden; deze procedure wordt later gedetailleerd besproken. Na terugkeer uit de procedure worden de pijlen van ISCi éénmaal in rekening gebracht.
Hieronder geven wij de globale struktuur van het deel-algorithme "compute es, check and find cp" weer. De procedure "take into account ISC (V, index)" verzorgt het in rekening brengen van een pijl tussen twee sterke komponenten. De knooppunten van de onderhavige SC staan in het array "V list" telkens op de plaatsen "low" van "up". Het aantal knooppunten van SCi wordt gegeven door: "size of SC[p-i+1]"

MAIN PROGRAM:

```
up:= 0;
for i:=1 step 1 until n do pusher[i]:= 0;
for i:=1 step 1 until p do
  begin low:= up + 1; up:= up + size of SC[p - i + 1];
    if size of SC[p-i+1] ≠ 1 then compute es of SCi( low,up );
    for j:=low step 1 until up do
      begin V:= Vlist[j]; FURTHER  REGISTRATIONS;
        for index:=lefw[V] + 1 step 1 until le[V] do
        take into account ISC( V,index )
      end;
  end;
```

De procedure take into account ISC (V,index) bestaat uit de volgende statements:

```
successor:= suc[index];
if es[V] + length[index] > es[successor] then
  begin es[successor]:= es[V] + length[index];
    pusher[successor]:= V;
  end;
```

Na afloop van het deel-algorithme: "compute es, check and find cp" wordt het kritieke pad gevormd door de reeks knooppunten:
FINISH, pusher[FINISH], pusher[pusher[FINISH]], ... START.
Aanvankelijk is pusher [i] ongedefiniëerd; dit wordt weergegeven door pusher [i] = 0 te maken. Bij een korrekt netwerk behoudt alleen pusher[START] de waarde: "0".

Wij bespreken nu de procedure "compute es of SCi".

Hieronder is de globale struktuur van het programma weergegeven (Zie Hoofdstuk 5):

```
repeat      change := false;
            COMPUTE ES FORWARD;
            COMPUTE ES BACKWARD;
            LOOK FOR POSITIVE CYCLES
until       not change or  TOO MANY CYCLES REPORTED;
```

Wij zullen de uitvoering van elk van beide procedures: "COMPUTE ES FORWARD" en "COMPUTE ES BACKWARD" een *slag* noemen. (Zie Hoofdstuk 6.3.)

In iedere slag behoeft men uitsluitend de uitgaande pijlen in beschouwing te nemen van die knooppunten, waarvan de vms-tijd in de vorige of in de huidige slag is gewijzigd. Dit is geïmplementeerd, door de slagen te nummeren m.b.v. de variabele "turnnumb". Het programma dient nu te beschikken over het nummer van de slag, waarin de vms-tijd van V het laatst werd gewijzigd; hiervoor wordt gebruik gemaakt van het array "turnnumb of last ch [1:n]".

De procedure-body van "compute es of SCi" wordt nu:

```
turnnumb := 0; NUMBER OF REPORTED POSITIVE CYCLES :=0;
for j := low step 1 until up do turnnumb of last ch[Vlist[j]]:= 1;
repeat   turnnumb := turnnumb + 1; change := false;
         COMPUTE ES FORWARD;
         if change or turnnumb = 1 then
         begin  turnnumb := turnnumb + 1;  change := false;
                COMPUTE ES BACKWARD;
         end;
         LOOK FOR POSITIVE CYCLES;
until    not  change  or  TOO MANY CYCLES REPORTED;
```

De test: "turnnumb = 1" in de vijfde regel is nodig, omdat het mogelijk is, dat de eerste aanroep van "COMPUTE ES FORWARD" geen verandering te weeg brengt, terwijl dan wel "COMPUTE ES BACKWARD" moet worden aangeroepen.
De procedure "COMPUTE ES FORWARD" neemt achtereenvolgende knooppunten uit het array Vlist van plaats "low" tot en met plaats "up", in beschouwing en brengt de pijlen van het FORWARD deelnetwerk in rekening:

```
procedure COMPUTE ES FORWARD;
  begin for j:=low step 1 until up do
    begin V:= Vlist[j];
      if turnnumb of last ch[V] ≥ turnnumb - 1 then
        for index:= lebw[V] + 1 step 1 until lefw[V] do
        take into account FBW( V,index )
    end
  end;
```

De procedure "COMPUTE ES BACKWARD" neemt achtereenvolgens de knooppunten uit het array Vlist van plaats "up" tot en met plaats "low" in beschouwing en brengt de pijlen van het backward deelnetwerk in rekening:

```
procedure COMPUTE ES BACKWARD;
  begin for j:=up step - 1 until low do
    begin V:= Vlist[j];
      if turnnumb of last ch[V] ≥ turnnumb - 1 then
        for index:= fe[V] step 1 until lebw[V] do
        take into account FBW( V,index )
    end;
  end;
```

De procedure: "take into account FBW(V, index)", bestaat uit de volgende statements:

```
successor:= suc[index];
if es[V] + length[index] > es[successor] then
  begin es[successor]:= es[V] + length[index];
    change:= true; pusher[successor]:= V;
    turn numb of last ch[successor]:= turnnumb;
  end;
```

Wij beschouwen nu het opsporen van positieve cykels. Binnen de procedure "compute es of SC_i" wordt na iedere twee slagen gestest, of de "pusher-graaf" (zie appendix C3, definitie 3.1.) een cykel bevat, m.b.v. de procedure "LOOK FOR POSITIVE CYCLE". In appendix C3 worden de volgende stellingen bewezen:

(a) Wanneer de "pusher-graaf" een cykel bevat, is er in de oorspronkelijke graaf een bijbehorende cykel van positieve lengte.

(b) Wanneer er in de sterke komponent een cykel van positieve lengte is, bevat de "pusher-graaf" na een gering aantal slagen een cykel.

Het aantal cykels, dat is gerapporteerd binnen SC_i wordt geteld m.b.v. de variabele "nofrepcyc". (number of reported cycles). Wanneer het aantal gerapporteerde cykels groter wordt dan een bepaalde waarde (in onze implementatie: groter dan het aantal knooppunten van SC_i), worden de berekeningen m.b.t. SC_i afgebroken.

T.b.v. de procedure: "LOOK FOR POSITIVE CYCLE" worden drie disjunkte verzamelingen knoopunten onderscheiden:

- De verzameling UNCHECKED; tot deze verzameling behoren:
  (a) alle knooppunten van SC_j , $i < j \leq p$
  (b) alle knooppunten van SC_i , op ieder moment buiten de procedure LOOK FOR POSITIVE CYCLE;
  (c) sommige knooppunten van SC_i , op momenten binnen de procedure LOOK FOR POSITIVE CYCLE;

  Implementatie: $V \in$ UNCHECKED $\leftrightarrow$ status [V] = unchecked (=1).
- De verzameling CHECKED; tot deze verzameling behoren:
  (a) alle knooppunten van SC_h , $1 \leq h < i$
  (b) sommige knoopunten van SC_i, op momenten binnen de procedure LOOK FOR POSITIVE CYCLE;
  (c) een fictief knooppunt: "0"; het nut hiervan,wordt hieronder verklaard.

  Implementatie: $V \in$ CHECKED: $\leftrightarrow$ status [V] = checked (=2).

- De verzameling CHECKING: tot deze verzameling behoren sommige knoopunten van SC_i, op momenten binnen de procedure LOOK FOR POSITIVE CYCLE.
  Implementatie: $V \in$ CHECKING:$\leftrightarrow$ status [V] = checking (=3).

Bij het binnengaan van de procedure "LOOK FOR POSITIVE CYCLE" hebben alle knooppunten van de SC de status "UNCHECKED".
De kern van het algorithme bestaat uit de volgende statements:

STAP 1 : Kies een knooppunt V ε UNCHECKED $\cap$ SCi;
VOEG V TOE AAN DE VERZAMELING CHECKING;

```
STAP 2 : while pusher [V] ε UNCHECKED do
         begin V:= pusher [V];
               VOEG V TOE AAN DE VERZAMELING CHECKING
         end;
```

Als na uitvoering van stap 2 geldt, dat knooppunt pusher [V]$\in$ CHECKING, is er een cykel gevonden; het algorithme gaat verder met de procedure: "REPORT AND REPAIR POSITIVE CYCLE", die later wordt besproken.
Als na uitvoering van stap 2 geldt, dat pusher [V]$\in$ CHECKED, terwijl knooppunt "pusher [V]" niet tot SC_i behoort, betekent dit dat de vms-tijd van V binnen de procedure "compute es of SC_i" niet is verhoogd, en dat dus géén positieve cykel is gevonden. Daarom krijgt pusher [i] , i=1,..,n,aan het begin van "compute es, check and find cp" de waarde = "0"; en is er een fiktief knooppunt **O** $\in$ CHECKED ingevoerd, opdat status[pusher[V]] altijd bestaat.
Als na uitvoering van stap 2 geldt, dat pusher [V]$\in$ CHECKED terwijl knooppunt:"pusher [V]" wel tot SC_i behoort, is dit knooppunt in dit onderzoek reeds tevoren beschouwd. Er is geen (nieuwe) positieve cykel gevonden.

Na eventuele uitvoering van "REPORT AND REPAIR POSITIVE CYCLE", moeten alle knooppunten, die de status "CHECKING" hebben, de status "CHECKED" krijgen, daar het onderzoek, dat deze knooppunten betreft, is afgesloten. Het algorithme gaat verder bij Stap 1, totdat geldt: UNCHECKED $\equiv \phi$.

Tenslotte worden alle knooppunten van de SC vanuit de verzameling CHECKED overgeheveld naar de verzameling UNCHECKED, opdat de knooppunten bij een eventuele volgende aanroep van "LOOK FOR POSITIVE CYCLE" de juiste status hebben.

De procedure "LOOK FOR POSITIVE CYCLE" is als volgt:

```
procedure  LOOK FOR POSITIVE CYCLE;
        declaration procedure  REPORT AND REPAIR POSITIVE CYCLE;
begin   for j: = low  step 1 until up do
        begin  V : = Vlist [j];
               if  V  ε UNCHECKED  then
               begin fix V:=V; status[V]:= checking;
               while pusher [V] ε UNCHECKED do
               begin V := pusher [V] ; status [V] := checking
               end;

               if pusher [V] ε CHECKING

               then begin PUT ALL NODES OF SET CHECKING INTO
                          SET CHECKED:  V : = fix V;
                   repeat status [V] := checked; V : = pusher [V]
                   until not  status [V] = checking;
                   REPORT AND REPAIR POSITIVE CYCLE (V);
                   correct : = false
                   end
               else begin  PUT ALLE NODES OF SET CHECKING INTO
                           SET CHECKED:  V : = fix V;
                   repeat status [V] :=checked;  V : = pusher[V]
                   until not  status [V] = checking
                   end
               end
        end;
```

```
PUT ALL NODES OF SET CHECKED INTO SET UNCHECKED:
for j : = low step 1 until up do  status [V list [j]]:= unchecked;
end;
```

De tests " V ε UNCHECKED" en "V ε CHECKING" moeten uiteraard vervangen worden door "echte" conditionele expressies, die onmiddellijk uit onze implementatie volgen. In het hoofdprogramma (d.w.z. in de procedure "compute es, check and find cp") wordt de status van alle knooppunten op "unchecked" geïnitialiseerd. Alleen het fiktieve knooppunt 0 krijgt als status: "checked". Zodra de knooppunten van SC_i hun definitieve vms-tijden hebben, krijgen zij de status: "checked".

Tenslotte bespreken wij de procedure "REPORT AND REPAIR POSITIVE CYCLE". Deze procedure spoort in het netwerk een positieve cykel op, aan de hand van de cykel die is gevonden in de "pushergraaf". De cykel wordt gerapporteerd. De lengte l van de positieve cykel wordt bepaald, en de lengte van één van de pijlen wordt met l verminderd. Deze strategie wordt gevolgd, om eventuele andere positieve cykels binnen dezelfde sterke komponent nog te kunnen rapporteren; wij pretenderen beslist niet, een zinvolle fout-korrektie te geven.
Dit opsporen van meerdere positieve cykels, en het aanpassen van de lengten der pijlen voor rekentechnische doeleinden, is nuttig bij een *batch-gewijze* verwerking; bij een *konversationele* verwerking zijn er betere mogelijkheden.
Ons algoritme spoort gewoonlijk niet alle positieve cykels op; wanneer men de lengte van één pijl verkort, kunnen vele

positieve cykels "verdwijnen". Het is vaak ook niet zinvol, te vragen om alle positieve cycles aangezien het aantal cykels zeer groot zijn. Ter illustratie beschouwe men fig.10.2.

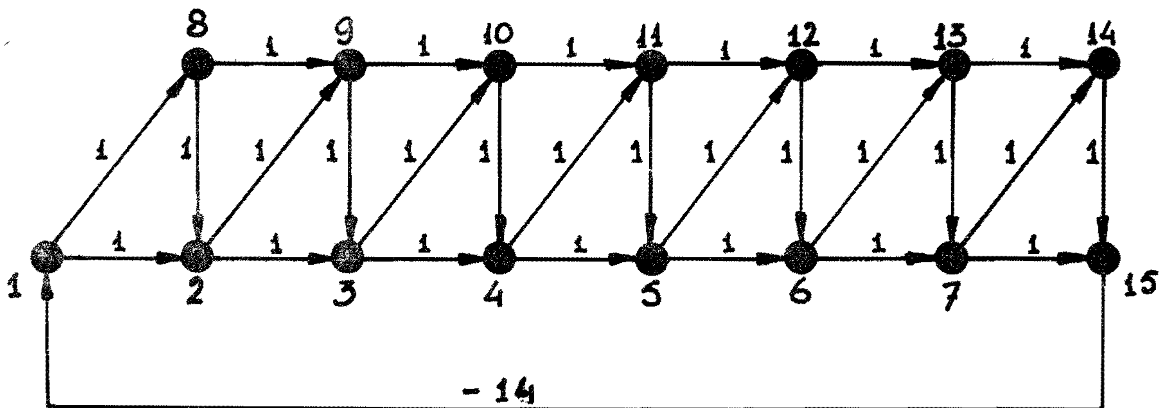


fig. 10.2.

Wanneer de pijl (7.1) door een ponsfout de lengte + 14 krijgt, bedraagt het aantal positieve cykels: 610.

Ons herstellingsmechanisme garandeert niet, dat er slechts een gering aantal cykels worden gerapporteerd. Daarom worden per sterke komponent het aantal gerapporteerde cykels geteld m.b.v. de variabele "nofrepcyc" (number of reported cycles); wanneer "nofrepcyc" groter wordt dan het aantal knooppunten van de onderhavige SC,wordt de berekening van deze SC afgebroken, en wordt de procedure "compute es of SC_i" verlaten. De variabele "nofrepcyc" wordt binnen de procedure "compute es of SC_i" geïnitialiseerd op de waarde nul.

Veronderstel dat zojuist een positieve cykel door het algorithme is gerapporteerd: indien de waarden van het array pusher in de volgende slagen niet gewijzigd worden, dan kan dezelfde cykel meerdere malen gerapporteerd worden. Dit kan als volgt worden

vermeden:
zij P de pijl, waarvan de lengte wordt verminderd met de lengte van de positieve cykel, dan krijgt pusher [e (P)] de waarde 0.

Hieronder volgt de procedure "REPORT AND REPAIR POSITIVE CYCLE (V)". Het knooppunt V behoort gegarandeerd tot een positieve cykel, die in "LOOK FOR POSITIVE CYCLE" is gevonden. Het knooppunt "av" (actual vertex) is telkens een knooppunt van de cykel, en "pav" is de pusher van "av". De variabele "index of maxl" geeft de plaats aan in het array "length", waar de langste[1] pijl van "pav" naar "av" genoteerd staat. Na afloop van de repeat-statement geeft "index of maxl" de plaats aan van de langste pijl van V naar de direkte opvolger van V in de gevonden positieve cykel.

```
procedure  REPORT AND REPAIR POSITIVE  CYCLE (V); value V; integer V;
begin    integer  av,  pav, index of maxl, index, cyclelength, maxl;

         av : = V ;   cycle length : = o;
         repeat   pav : = pusher [av]; maxl : = - BIGM;
FIND LONGEST ARC: for index : = fe [pav] step 1 until lefw [pav] do
                  if  suc [index] = av and length [index] > maxl
                  then   index of maxl : = index;
                         av : = pav; cycle length : = cycle length +
                                              length [index of maxl]
         until    av = V ;
         length [index of maxl] : = length [index of maxl]-cycle length;
         pusher [suc  [index of maxl]] : = 0;
         nofrepcyc : = nofrepcyc + 1
end;
```

1) In het hier gepresenteerde algorithme zijn meerdere pijlen van knooppunt i naar knooppunt j toegestaan. Het is dan essentieel de langste pijl te nemen. In ANNETTE II worden meerdere pijlen van knooppunt i naar knooppunt j geïnterpreteerd als een fout. Er wordt hierop in een eerder stadium gekontroleerd.

10.2 *Algol programma*

```
procedure compute es check and find cp;
  begin integer i,low,up,j,V,index,unchecked,checking,checked;
    integer array status[0:n];
    declaration procedure compute es of SCi, procedure take into account ISC;
PROGRAM:
    up:= 0; unchecked:= 1; checked:= 2; checking:= 3;
    for j:=1 step 1 until n do pusher[j]:= 0;
    for j:=1 step 1 until n do status[j]:= unchecked;
    status[0]:= checked;
    for i:=1 step 1 until p do
    begin low:= up + 1;
      up:= up + size of SC[p-i+1];
      if size of SC[p-i+1] ≠ 1 then compute es of SCi;
      for j:=low step 1 until up do
      begin V:= Vlist[j]; status[V] := checked;
        for index:= lefw[V] + 1 step 1 until le[V] do
        take into account ISC
      end
    end;
  end compute es  check and find cp;


procedure take into account ISC;
  begin integer successor;
    successor:= suc[index];
    if es[successor] < es[V] + length[index] then
      begin es[successor]:= es[V] + length[index];
        pusher[successor]:= V;
      end
  end take into account ISC;
```

```
procedure compute es of SCi;
  begin integer turnnumb,nofrepcyc,j;
    integer array turnnumb of last ch[1:n];
    boolean change;
    declaration procedure COMPUTE ES FORWARD,
                procedure COMPUTE ES BACKWARD,
                procedure LOOK FOR POSITIVE CYCLE,
                procedure take into account FBW( V,index );
    turnnumb:= 0; nofrepcyc:= 0;
    for j:=low step 1 until up do turnnumb of last ch[Vlist[j]]:= 1;
    repeat turnnumb:= turnnumb + 1; change:= false;
      COMPUTE ES FORWARD;
      if change or turnnumb = 1 then
        begin turnnumb:= turnnumb + 1; change:= false;
          COMPUTE ES BACKWARD
        end;
      if change then LOOK FOR POSITIVE CYCLE
    until not change or nofrepcyc ≥ up - low;
  end compute es of SCi;


procedure COMPUTE ES FORWARD;
  begin integer j,V,index;
    for j:=1 step 1 until up do
      begin V:= Vlist[j];
        if turnnumb of last ch[V] ≥turnnumb - 1 then
          for index:= lebw[V] + 1 step 1 until lefw[V] do
          take into account FBW( V,index );
      end
  end COMPUTE ES FORWARD;

procedure COMPUTE ES BACKWARD;
  begin integer j,V,index;
    for j:=up step - 1 until low do
      begin V:= Vlist[j];
        if turnnumb of last ch[V] ≥ turnnumb - 1 then
          for index:= fe[V] step 1 until lebw[V] do
          take into account FBW( V,index );
      end
  end COMPUTE ES BACKWARD;
```

```
procedure take into account FBW( V,index ); value V,index; integer V,index;
  begin integer successor;
    successor:= suc[index];
    if es[successor] < es[V] + length[index] then
      begin es[successor]:= es[V] + length[index];
        pusher[successor]:= V; change:= true;
        turnnumb of last ch[successor]:= turnnumb;
      end
  end take into account FBW;

procedure LOOK FOR POSITIVE CYCLE;
  begin integer j,V,fixV;
    declaration procedure REPORT AND REPAIR POSITIVE CYCLE( V );
    for j:= low step 1 until up do
    begin V:= Vlist[j];
      if V ε UNCHECKED: status[V]= unchecked then
        begin fixV:= V; status[V]:= checking;
          while status[pusher[V]]=unchecked do
          begin V := pusher[V]; status[V]:= checking
          end;
          if pusher[V] ε CHECKING: status[pusher[V]] = checking then
            begin ALL NODES OF SET CHECKING TO SET CHECKED:
              V:= fixV;
              repeat status[V]:= checked; V:= pusher[V];
              until not status[V] = checking;
              REPORT AND REPAIR POSITIVE CYCLE( V );
              correct:= false
            end
          else
            begin ALL NODES OF SET CHECKING TO SET CHECKED:
              V:= fixV;
              repeat status[V]:= checked; V:= pusher[V];
              until not status[V] = checking;
            end
        end
    end;
    PUT ALL NODES OF SET CHECKED INTO SET UNCHECKED:
    for j:= low step 1 until up do status[Vlist[j]]:= unchecked;
  end LOOK FOR POSITIVE CYCLE;
```

```
procedure REPORT AND REPAIR CYCLE( V ); value V; integer V;
  begin integer av,pav,index of maxl, index,
        cyclelength,maxl;
    av:= V; cyclelength:= 0;
    repeat pav:= pusher[av]; maxl:= - BIGM; PRINT( pav );
FIND LONGEST ARC:
      for index:= fe[pav] step 1 until lefw[pav] do
        if suc[index] = av and length[index] > maxl then
        index of maxl := index;
      av:= pav; cyclelength:= cyclelength + length[index of maxl];
    until av = V;
    length[index of maxl]:= length[index of maxl] - cyclelength;
    pusher[suc[index of maxl]]:= 0;
    nofrepcyc:= nofrepcyc + 1;
  end;
```

11. Het deel-algorithme: "Compute ls".

11.1. Beschrijving.

Het deelalgorithme: "compute ls" berekent de lts-tijden van de knooppunten. Het deel-algorithme wordt alleen dan uitgevoerd, als het netwerk géén positieve cykels bevat. In ons algorithme worden de lts-tijden van FINISH en van de overige knooppunten van het kritieke pad gelijk gemaakt om de vms-tijden van die knooppunten aan het begin van de procedure: "compute ls". Dit is een efficientie-maatregel.

Daarnaast levert de procedure "compute ls" een array: "holder" af, dat een soortgelijke rol speelt bij de lts-tijden als het array: "pusher" bij de vms-tijden. Na afloop van het deel-algorithme geldt namelijk:

$$\text{holder } [V] = W \rightarrow \exists \text{ pijl } P,\ b(P) = V,\ e(P) = W : \text{ls } [V] + \text{length } [P] = \text{ls } [W]$$

De reeks knooppunten: V, holder [V], holder [holder [V]] ,....., FINISH vormt een kritiek pad van V naar FINISH. Bij ieder knooppunt V, behalve bij FINISH, geeft holder [V] aan welk knooppunt "verantwoordelijk" is voor de lts-tijd van V. Bij de initialisatie wordt voor alle i: holder [i] op "0" gesteld, d.w.z. holder [i] is ongedefinieerd. Vervolgens wordt voor alle knooppunten V, V$\in$ het kritiek pad, V $\neq$ START:
holder [pusher [V]] = V gesteld.
Dit laatste is noodzakelijk, als men, zoals hier, voor knooppunten van het kritiek pad, de lts-tijden tevoren initieert op de waarde, die ze anders uiteindelijk toch zouden krijgen. Na afloop van het algorithme geldt:

$$i = \text{FINISH} \leftrightarrow \text{holder } [i] = 0$$

Het deel-algorithme maakt gebruik van het array "Vlist" en van de inhoud van het array "size of SC". De relatie, die deze grootheden met elkaar hebben, is hieronder weergegeven.

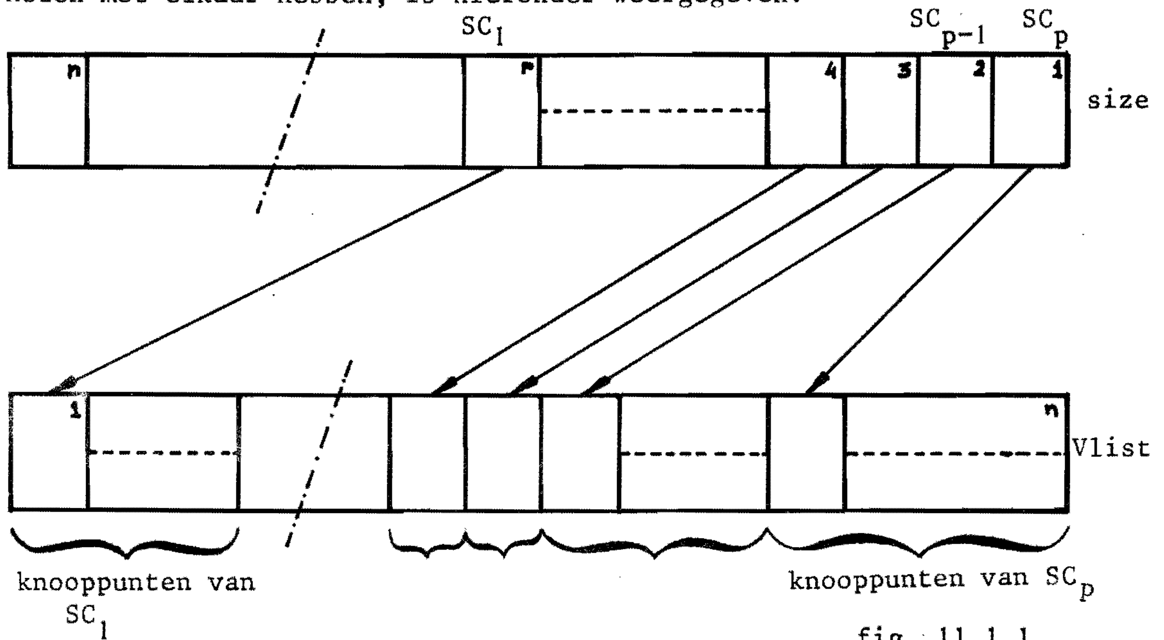


fig. 11.1.1.

Het algorithme werkt de sterke komponenten af in de volgorde SC_p, ..., SC_1. Indien de sterke komponent uit slechts één knooppunt bestaat, worden de uitgaande pijlen van dit knooppunt éénmaal in rekening gebracht. Indien de sterke komponent SC_i samengesteld is, worden eerst de pijlen uit de verzameling ISC_i éénmaal in rekening gebracht, en wordt vervolgens de procedure: compute ls of SC_i" aangeroepen. Hieronder volgt de globale struktuur van het deel-algorithme "compute ls":

```
MAIN PROGRAM:
  for j:=1 step 1 until n do holder[j]:= 0;
  j:= FINISH; ls[FINISH]:= es[FINISH];
  while j ≠ START do
    begin holder[pusher[j]]:= j; j:=pusher[j]; ls[j]= es[j] end;
  low:= n + 1;
  for i:=p step - 1 until 1 do
    begin up:= low - 1;
      low := low - size of SC[p-i+1];
      for j:= low step 1 until up do
      begin V:= Vlist[j];
        for index:= lefw[V] + 1 step 1 until le[V] do
        take into account ISC( V,index );
      end;
      if size of SC[p-i+1] ≠ 1 then compute ls of SCi( low,up );
    end;
```

De procedure "take into account ISC(V,index)" bestaat uit de volgende statements:

```
successor:= suc[index];
if ls[V] + length[index] > ls[successor] then
  begin ls[V]:= ls[successor] - length[index];
    holder[V]:= successor;
  end;
```

Wij bespreken nu de procedure "compute ls of SCi(low,up)".
De globale struktuur van het programma is hieronder weergegeven:

```
COMPUTE LS FORWARD;
repeat change:= false; COMPUTE LS BACKWARD;
  if change then
    begin change:= false; COMPUTE LS FORWARD end
until not change;
```

Bij het berekenen van de vms-tijden werd een pijl P slechts in rekening gebracht, als de vms-tijd van b(P) is gewijzigd sinds de vorige keer, dat P in rekening werd gebracht. Dit was efficient te implementeren, omdat de pijlen in onze datastruktuur naar beginknooppunt gesorteerd staan. Om bij het berekenen van de lts-tijden een soortgelijke efficiency-maatregel te nemen, dienen de pijlen echter naar eindknooppunt gesorteerd te staan. Immers, dan kan men de strategie volgen, om de binnenkomende pijlen van knooppunt V slechts in rekening te brengen, als de vms-tijd van knooppunt V in de vorige of huidige slag is gewijzigd. Men kan dus bij het berekenen van de lts-tijden op twee manieren te werk gaan:

(a) De datastruktuur wijzigen, en de efficiency-maatregel nemen.
(b) De datastruktuur niet wijzigen en de efficiency-maatregel weglaten.

Het wijzigen van de datastruktuur vergt een tamelijk ingewikkeld algorithme, waarvan de rekentijd evenredig is met m, het aantal pijlen in de graaf.

Wij menen, dat de efficiency-winst van (a) t.o.v. (b) miniem is. Wij hebben daarom en vanwege de eenvoud gekozen voor (b). Hierbij zijn het array "turnnumb of last ch" en de variabele "turnnumb" vervallen.

Voor het overige spreekt de algol-tekst van "compute ls" voor zichzelf.

11.2 *Algol - tekst*

```
procedure compute ls;
  begin integer i,low,up,j,V,index;
    declaration procedure compute ls of SCi, take into account ISC;
PROGRAM:
    for j:=1 step 1 until n do holder[j]:= 0;
    j:= FINISH; ls[FINISH]:= es[FINISH];
    while j ≠ START do
      begin holder[pusher[j]]:= j; j:= pusher[j]; ls[j]:= es[j] end;
    low:= n + 1;
    for i:= p step - 1 until 1 do
      begin up:= low - 1; low:= low - size of SC[p-i+1];
        for j:= low step 1 until up do
          begin V:= Vlist[j];
            for index:= lefw[V] + 1 step 1 until le[V] do
            take into account ISC
          end;
        if size of SC[p-i+1] ≠ 1 then compute ls of SCi
      end;
  end;


procedure take into account ISC;
  begin integer successor;
    successor:= suc[index];
    if ls[V] + length[index] > ls[successor] then
    begin ls[V]:= ls[successor] - length[index];
      holder[V]:= successor;
    end
  end take into account ISC in compute ls;
```

```
procedure compute ls of SCi;
  begin boolean change;
    declaration procedure COMPUTE LS FORWARD,
                procedure COMPUTE LS BACKWARD,
                procedure take into account FBW( V,index );
    COMPUTE LS FORWARD;
    repeat change:= false;
      COMPUTE LS BACKWARD;
      if change then
        begin change:= false; COMPUTE LS FORWARD end;
    until not change;
  end;

procedure COMPUTE LS FORWARD;
  begin integer j,V,index;
    for j:= up step - 1 until low do
      begin V:= Vlist[j];
        for index:= lebw[V] + 1 step 1 until lefw[V] do
          take into account FBW( V,index );
      end;
  end COMPUTE LS FORWARD;

procedure COMPUTE LS BACKWARD;
  begin integer j,V,index;
    for j:= low step 1 until up do
      begin V:= Vlist[j];
        for index:= fe[V] step 1 until lebw[V] do
          take into account FBW( V,index );
      end
  end COMPUTE LS BACKWARD;

procedure take into account FBW( V,index ); value V,index; integer V,index;
  begin integer successor;
    successor:= suc[index];
    if ls[V] + length[index] > ls[successor]then
      begin ls[V]:= ls[successor] - length[index];
        change:= true; holder[V]:= successor;
      end
  end take into account FBW;
```

12. *Kwaliteitsbeoordeling van het Algorithme*

Beoordeling zal hier plaatsvinden volgens de door Alanen [1] genoemde kriteria. Behalve de gebruikelijke kriteria "geheugenbeslag" en "efficientie" onderscheidt Alanen de volgende beoordelingskriteria:

- robuustheid: hoe gedraagt het programma zich bij verschillende soorten input.
- overdraagbaarheid: is het met weinig moeite ook geschikt voor andere machines.
- doorzichtigheid: helderheid van representatie.
- nauwkeurigheid: de nauwkeurigheid van het eindresultaat.
- betrouwbaarheid: mate van bescherming tegen fouten.
- aanpasbaarheid: is het eenvoudig te wijzigen voor een iets andere probleemstelling.
- geheugenbeslag: gebruik van ruimte.
- efficientie: gebruik van rekentijd. In onze beschouwingen wordt de efficientie beoordeeld aan de hand van de evenredigheid met de grootte en komplexiteit van het probleem. Een detaillering naar aantal vermenigvuldigen, optellingen, procedure-aanroepen etc. heeft alleen zin voor een specifieke machine en leek ons hier niet zinvol.

12.1. *Robuustheid.*

Het programma gedraagt zich bij verschillende problemen goed, d.w.z. de rekentijd neemt evenredig toe met de grootte van het probleem en neemt langzaam toe met stijgende komplexiteit. (zie 12.8.)

12.2. *Overdraagbaarheid.*

Behoudens kleine details (while) is het programma direkt geschikt voor iedere machine met een Algolvertaler. Er is nergens gebruik gemaakt van specifieke machine-eigenschappen als woordlengte, getalrepresentatie etc.

12.3. *Doorzichtigheid*

Wij zijn van mening, dat de doorzichtigheid van het programma sterk bevorderd is door:

- het gebruik van rekursie
- het gebruik van repeat en while-statements
- het ontbreken van goto-statements
- een modulaire opbouw uit procedures met een duidelijke werking waardoor de lezer "laagsgewijs" een verdergaande detaillering kan verkrijgen.

12.4. *Nauwkeurigheid.*

Doordat uitsluitend met integers gerekend wordt, is de nauwkeurigheid absoluut.

12.5. *Betrouwbaarheid.*

In de appendices worden van de niet-triviale gedeelten van het algorithme bewijzen van korrektheid geleverd. Het algorithme is beveiligd tegen gebruikers-fouten (positieve cykels). Wel wordt er van uitgegaan, dat de aangeleverde datastruktuur een graaf representeert. Verder wordt t.a.v. de input uitgegaan van de volgende praemissen:

- $n \geq 1$
- lussen ontbreken.

12.6. *Aanpasbaarheid.*

Uiteraard is dit algorithme alleen geschikt voor netwerkplanning. Binnen deze klassen van problemen is het eenvoudig om kleine wijzigingen aan te brengen zoals:

- kontrole op positieve cykels om de k-slagen.
- op konversationele wijze het herstel van positieve cykels aan de gebruiker over te laten.
- het berekenen en afdrukken van de langste paden van "start" naar ieder knooppunt "i" of van "i" naar "finish". Het berekenen en afdrukken van bepaalde delen van deze paden op aanwijzing van de gebruiker.
  de array's "pusher" en "holder" bevatten de hiervoor benodigde gegevens.
- het is mogelijk bepaalde faciliteiten te laten vervallen, teneinde het ruimte beslag te verminderen. Zo kan men het array "le" laten vervallen waarbij: le [i] vervangen moet worden door "fe [i+1]-1", mits: fe[n+1] = m+1.

Het array "holder" kan men laten vervallen, wanneer men alleen geïnteresseerd is in het kritieke pad, etc.

12.7. *Geheugenbeslag en efficientie.*

12.7.1. *Vooronderstellingen.*

Bij de beschouwingen over geheugenbeslag en efficientie zijn wij hier uitgegaan van het "normale" of "gemiddelde" geval. Beschouwingen over het "gunstigste" en het "ongunstigste" geval zijn te vinden in de appendices.
Het door ons gekozen "gemiddelde" geval laat zich als volgt karakteriseren:

- het netwerk heeft een langgerekte struktuur met een verhoudingsgewijs gering aantal pijlen.
- het netwerk is te splitsen in een groot aantal sterke komponenten waarvan de samengestelde meestal 2 en hoogstens 8 knooppunten bevatten.
- er is één en precies een langste weg van "start" naar "finish". Deze voorwaarde maakt de efficientieberekeningen veel eenvoudiger, terwijl anderzijds de eindkonklusies bij afwijking van deze voorwaarde meestal slechts in geringe mate anders zullen zijn. (zie appendix B)
- er bevinden zich $\pm \bar{n}/8$ knooppunten op "het" (zie boven) kritieke pad. Batersby [2] geeft op, dat bij grotere netwerken 10 - 15 % van de aktiviteiten kritiek zijn. Dit sluit aan bij onze eigen praktische ervaring.

12.7.2. *Geheugenbeslag.*

Het geheugenbeslag is in de diverse deelalgorithmen uitgedrukt in de volgende parameters:

n = het aantal knooppunten van de graaf

m = het aantal pijlen van de graaf.

r = de maximale rekursie-diepte.

n_i= het aantal knooppunten van SC_i

r_i= de maximale rekursie diepte bij SPLIT ARCS AND ORDEN NODES van SC_i $(r_i \leq n_i)$

Bij het schatten van het gemiddelde geval zijn wij uitgegaan van de volgende veronderstellingen (zie 12.7.1.)

1. $m \approx 4\ n$ (maximaal $m = n^2$)
2. $r \approx n/8$ (maximaal $r = n$)
3. $n_i \leq 8$
4. $r_i = 8$ (maximaal $r_i = n_i = n$)

Voor iedere procedure-aanroep hebben wij 2 woorden extra ruimte beslag aangenomen. Het ruimte beslag voor lokalen van niet-rekursie procedures is konstant en klein en is hier niet apart vermeld.
Wij komen zo tot de volgende schatting van het maximale geheugengebruik in het door ons gekozen gemiddelde geval:

	als funktie van n, m, r_i, t	als funktie van n
Altijd binnen EMPM-Worker:	$11\ n + 2\ m$	$19\ n$
Binnen Decompose, orden SC and check:	$14\ n + 2\ m + 6\ r$	$23\ n$
Binnen Split arcs and orden nodes:	$12\ n + 2\ m + 5\ r_i$	$20\ n + 40$
Binnen compute es, check and find up:	$13\ n + 2\ m$	$21\ n$
Binnen compute ls:	$11\ n + 2\ m$	$19\ n$
Binnen het oorspronkelijke Gewald-Sauler algorithme:	$4\ n + 4\ m$	$16\ n$

Het is uiteraard mogelijk, door het plaatsen van meerdere gegevens in een machinewoord aanzienlijk te besparen op geheugenruimte. Zo zijn een aantal arrays mogelijkerwijs te kombineren zoals "fe", "le", lebw en lefw. Mogelijkheden hiertoe zijn bij diverse Algol-dialekten sterk verschillend. In deze representatie zou deze werkwijze de overdraagbaarheid en de doorzichtigheid geschaad hebben. Wanneer het wegens ruimte gebrek op een bepaalde machine nodig zou zijn, kan het programma alsnog door iedere geroutineerde programmeur in dier voege gewijzigd worden. Uiteraard zal de rekentijd dan iets toenemen.

12.7.3. *Efficientie.*

Doel van deze beschouwing is het geven van een vergelijking tussen het Gewald-Sauler algorithme en EMPM-Worker. De verhouding van de werkelijke rekentijd zal per machine verschillend zijn.
Wij gaan uit van een machine waarbij er geen grote verschillen in rekentijd zijn tussen: optelling, aanroep van een procedure, aanroep van een array element etc.

We kunnen de rekentijd dan uitdrukken in de volgende eenheid: de eenheid nodig voor een (soms komplexe) standaard-handeling. Als standaard-handeling beschouwen we:
Het in beschouwing nemen (en eventueel in rekening brengen) van een pijl. Uiteraard maken wij een fout in de berekening van de benodigde hoeveelheid werk, door slechts naar één, tamelijk grof gedefinieerde standaard-handeling te kijken.
Verdere detaillering en precisering geeft echter weinig nieuwe informatie.

Bij de efficientie-berekeningen maken wij onderscheid tussen het *voorbereidend gedeelte* en het *reken-gedeelte*. In EMPM-Worker wordt het voorbereidend gedeelte gevormd door de procedures: "Decompose, orden SC and check" en "Split arcs and orden nodes". De eerste procedure vraagt m standaard-handelingen; de tweede procedure vraagt in het gunstigste geval, bij een cykel-vrije graaf, geen enkele standaard handeling, en in het ongunstigste geval m standaard-handelingen.
Bij het Gewald-Sauler algorithme ontbreekt het voorbereidend gedeelte. Wanneer evenwel niet aan de bereikbaarheidsvoorwaarden is voldaan, is de vms-tijd of lts-tijd van sommige knooppunten niet gedefinieerd. Bovendien is het algorithme niet eindig , wanneer er zich een positieve cykel bevindt in een sterke komponent, van waaruit FINISH niet bereikbaar is. Daarom zou men ook hier een test op de bereikbaarheidsvoorwaarden moeten uitvoeren, wat altijd m standaard-handelingen kost.

In het reken-gedeelte is een ondergrens voor het aantal standaard-handelingen voor het Gewald-Sauler-algorithme(zie Appendix B2) $(r + 3) \times m$, waarin r het aantal pijlen van het kritieke pad is. Voor EMPM-Worker maken wij onderscheid tussen het gunstigste geval, het ongunstigste geval en het "normale" geval. Wij nemen aan, dat het netwerk geen positieve cykels bevat.
In het gunstigste geval, bij een cykel-vrije graaf, is het aantal standaard-handelingen in het rekengedeelte gelijk aan 2 m.
In het ongunstigste geval, dat echter bijzonder onwaarschijnlijk is, is een bovengrens voor het aantal standaard handelingen: $(r+2)\cdot m$, onder zwakke vooronderstellingen. Dit wordt afgeleid in Appendix C5.

Voor het "normale geval" is het redelijk te veronderstellen dat de sterke komponenten niet alleen klein ($n_i \leq 8$) maar ook vrij eenvoudig van struktuur zijn. Bij de keuze van de volgorde zoals deze door SPLIT wordt bepaald, stonden ons dan ook sterke komponenten voor ogen van een zeer simpele struktuur: "de school van vissen".

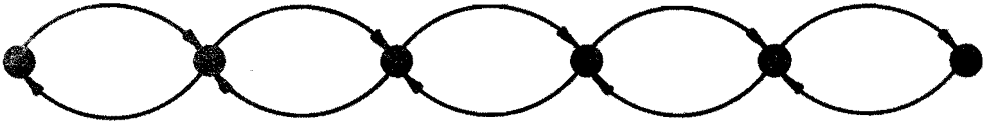


fig. 12.1.

Wij verwachten, dat dit soort zeer simpele strukturen bij de samengestelde komponenten frequent voor zullen komen. Het aantal slagen, benodigd voor het doorrekenen van een "school vissen" is ten hoogste 4, onafhankelijk van het aantal "vissen". Wanneer wij de sterke komponent niet splitsen in twee cykelvrije deelnetwerken, maar de pijlen P tussen twee knooppunten van de sterke component iteratief in rekening brengen (Gewald-Sauler-strategie binnen een sterke komponent), dan zullen tenminste twee iteraties nodig zijn, meestal echter meer dan twee. Deze twee iteraties komen dan overeen met 4 slagen:
de door ons voorgestelde splitsing bewijst zijn nut.
De pijlen van het FW-netwerk en het BW-netwerk worden dus tweemaal in rekening gebracht, de pijlen van het ISC-netwerk eenmaal, bij de berekening van de lts-tijden geldt hetzelfde. Als wij aannemen, dat de helft van de pijlen tot het ISC-netwerk behoort, is het aantal standaard handelingen in het reken-gedeelte: 3 m.

Wanneer wij m standaard-handelingen definiëren als een iteratie, komen wij tot het volgende overzicht:
Aantal iteraties: (r is het aantal pijlen van het kritieke pad.)

	ondergrens Gewald-Sauler	EMPM-Worker normale geval	EMPM-Worker gunstigste geval	EMPM-Worker ongunstigste geval
voorbereidend-gedeelte:	1	2	1	2
reken-gedeelte:	r+3	3	2	r+2
Totaal:	r+4	5	3	r+4

fig. 12.7.2.

Het algorithme EMPM-Worker is dus universeel efficienter, onder zeer zwakke aannamen. (Appendix C5). De winst-faktor bedraagt in het normale geval $\frac{r+4}{5}$. Voor $r = \frac{n}{8}$ is dit: $\frac{n+32}{40}$.
Voor praktijk-netwerken met 300 tot 2000 knooppunten is de winst een faktor 8 tot 50.
Wanneer het netwerk positieve cykels bevat, veranderen voor EMPM-Worker bovenstaande schattingen nauwelijks. Voor het Gewald-Sauler algorithme kan dan geen bovengrens van de rekentijd worden gegeven in termen van de graaf alleen; de lengten van de pijlen moeten erbij betrokken worden. (zie hoofdstuk 5.1).

Tot slot dient opgemerkt te worden, dat de rekentijd van het netwerk programma "ANNETTE II" niet alleen bepaald wordt door de rekentijd van het in dit rapport beschreven rekenstuk. Sterker nog: er is veel aandacht besteed aan de efficientie van het rekenstuk, terwijl aan de efficientie van het input/output nog weinig aandacht is besteed. Dit laatste is met opzet gebeurd. Speciaal t.a.v. het opzoeken van namen en het leveren van een balkendiagram, (Gantt-chart) zijn er stukken aan te wijzen waar nog een flinke efficientie-winst te behalen is. Reeds bij de opzet van het programma was ons dit bekend. Het is een kwestie van gebrek aan tijd die ons t.a.v. dit stuk heeft doen kiezen voor een eenvoudig en minder efficient programma. Uiteraard zal een van de eerstvolgende programmeer-activiteiten zijn het verbeteren van dit gedeelte.
Dit gedeelte is als volgt te karakteriseren:

- de moeilijkheden zijn niet van mathematische, maar meer van administratieve en organisatorische aard (bestandsmanipulatie, stringhandling, searching techniques etc.).

- er is intensief gebruik gemaakt van de specifieke mogelijkheden van Burroughs Extended Algol (BEA).
- zeer veel regels programma tekst.

Een en ander kan geïllustreerd worden door het volgende:

	CPU tijd	Aantal regels Algol
rekenstuk	1 sec.	500
overig gedeelte	10 sec.	4500

De rekentijden behoren bij een voorbeeld van 150 aktiviteiten (= 300 knooppunten: [7]: methode 2). Dit voorbeeld is verwerkt op de B 6700 van de THE te Eindhoven.

Literatuur

[1] Alanen, J.
Scientific program analysis techniques, Amsterdam, Mathematisch Centrum, 1972.

[2] Battersby, A.
Network Analysis for Planning and Scheduling, Macmillan, London 1970.

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

[4] Ford, L.R. and D.R. Fulkerson.
Flows in Networks, Princeton, New Jersey (1962).

[5] Gewald, K. and D. Sauler.
Ein ALGOL-programm für die METRA-potential methode, Ablauf, Planung und Forschung 7 (1966), hfdst. 4, pag. 199 ev.

[6] Harary, F., R. Norman and D. Cartwright.
Structural Models, New York, Wiley, 1965.

[7] Kerbosch, ir. J.A.G.M. en H.J. Schell.
Netwerkplanning volgens de methode Extended MPM, rapport KS 1, febr.'72, Technische Hogeschool Eindhoven.
Verschenen in: Informatie 14 (1972) no. 4, pag. 199 ev.

[8] Kerbosch, J.A.G.M. en H.J. Schell.
Gebruikershandleiding bij het netwerkprogramma ANNETTE II. Technisch Rapport KS 3, Technische Hogeschool Eindhoven.

[9] Kerbosch, J.A.G.M. en H.J. Schell.
Een procedure-pakket voor het hanteren van multilists in Burroughs Extended Algol. Technisch Rapport KS 4. Technische Hogeschool Eindhoven.
Verschijnt binnenkort

[10] Kerbosch, ir. J.A.G.M. en J.C. Wortmann.
De decompositie van een gerichte graaf in zijn sterke componenten, rapport KW 1, januari 1973, Technische Hogeschool Eindhoven.

[11] Kerbosch, ir. J.A.G.M. and 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, Technological University Eindhoven.
Verschijnt binnenkort

[12] Krafft, A.
Netzplantechnik, Dissertation Universität Basel.
Die Reprografie - Stuttgart 1969

[13] Roy, B.
Graphes et ordonnancements, Revue Francaise de recherche operationelle, nr. 25, 6 (1962, 10), pag. 323.

[14] Nederkoorn, J.
A PERT-program in Algol-60, MR56, Amsterdam, Mathematisch Centrum. (1963)

[15] Wille, H., K. Gewald und H.W. Weber.
Netzplantechnik, Methoden zur Planung und Ueberwachung von Projekten, Band I, Zeitplanung. Oldenbourgh, München (1966).

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 ε K). Twee knooppunten V en W heten *sterk verbonden*, d.e.s.d. als V bereikbaar vanuit W <u>en</u> W bereikbaar vanuit V. Pijl P heeft een lus, als b (p) = e (P). Het tripel <K', P', ϕ'> heet een *deelgraaf* van <K, P, ϕ> als $K' \subset K$, $P' \subset P$, en $\phi'(p') = \phi(p')$ $\forall$ $p' \varepsilon P'$. De deelgraaf heet *vol* als $P' = \{p \mid p \varepsilon P$ en $\phi(p) \varepsilon K' \times K'\}$. Kortheidshalve schrijven wij "deelgraaf" i.p.v. "volle deelgraaf". Zij D een deelgraaf van G. Als ieder tweetal knooppunten D_1 en D_2 ε 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 ε D, dan heet D een *sterk-samenhangende komponent* van de graaf G, ook wel *"sterke komponent"* genoemd; afkorting: *SC*. Merk op, dat de relatie "sterk-verbonden" een transitieve, symmetrische en reflexieve relatie is; oftewel, een equivalentie-relatie . Dit heeft tot gevolg, dat de verzameling knooppunten K éénduidig ontleedbaar is in equivalentie klassen, t.a.v. de relatie "sterk verbonden" . Deze equivalentie klassen van knooppunten vormen de sterke komponenten.

Indien een sterke komponent uit slechts één knooppunt bestaat, noemen wij deze *enkelvoudig*, anders *samengesteld*.
Een *netwerk* is een graaf, waarbij aan iedere pijl een getal is toegevoegd. Bij netwerk-planning noemen wij dit getal de *lengte* van de pijl.
De *lengte van een pad* is de som van de lengten van de pijlen van dat pad.
Een *kritiek pad* van een knooppunt V naar een knooppunt W is een pad, met de eigenschap dat er geen pad van V naar W loopt met een grotere lengte. Als in de netwerkplanning de knooppunten V en W niet nader wordt gespecificeerd, wordt een kritiek pad van knooppunt START naar knooppunt FINISH bedoeld.
In de netwerkplanning behoort de grootheid "vroegste start van projekt" te zijn gegeven; wij definiëren de *vroegst mogelijke starttijd* van een knooppunt V als volgt:

vms [V] = vroegste start van projekt +
lengte van het langste pad van START naar V.

Wij definiëren de *laatst-toegestane starttijd* van een knooppunt V als volgt:

lts [FINISH] = een gegeven waarde; in onze implementatie is deze gelijk gemaakt aan vms [FINISH]

lts [V] = lts [FINISH] -
lengte van het langste pad van V naar FINISH.

Appendix B1: Commentaar op het Gewald-Sauler-algorithme.

Opmerking: Appendix B1 is o.i. alleen interessant voor die lezers, die willen pogen, het artikel van Gewald und Sauler [5], te lezen.

Teneinde de lezer in staat te stellen, bovengenoemd artikel te lezen, verklaren wij allereerst een aantal zaken omtrent de programma-tekst, die o.i. niet geheel triviaal zijn.

ad blz. 200: - In de gewijzigde versie hebben wij aangenomen, dat de hernummering der knooppunten tot $\{1, .., n\}$ reeds is gebeurt. Anderzijds eisen wij niet, dat START = 1 of FINISH = n.

Het array "L", waarin de originele knooppuntsnummering van de gebruiker staat genoteerd, dient tevens als een soort boolean array.

ad blz. 201: - Het eerst worden de vms-tijden berekend, (case: S=1) met de kontrole op positieve cykels. De vms-tijden worden echter eerst weggeschreven in het array TS, en pas na afloop van de vms-berekening weggeschreven in het array TF.

- Ondertussen wordt het array TF gebruikt als "pusher".
- L [T] > 0 ↔ vms [T] is in de vorige iteratie gewijzigd. (In de gewijzigde versie: itnumb of lastch [T] ≥ itnumb -1)
- TS [T] < 0 ↔ vms [T] is in de huidige iteratie gewijzigd merk op, dat een pijl N niet wordt behandeld, als vms [b(N)] de vorige iteratie niet is gewijzigd, onafhankelijk van wat er in de huidige situatie is gebeurt.

ad blz. 202: - Case S=2 betreft lts-tijden.

- Tijdens het aflopen van het "pusher-pad" worden in het begin een stap van 3 pushers genomen, waarschijnlijk uit efficientie overwegingen.
- TF [V] < 0 ↔ (in onze gewijzigde versie): incp [V]
- Na de label WEITER: De variabele K geeft aan, hoeveel knooppunten tijdens de afgelopen iteratie zijn gewijzigd:
- Het algorithme tussen de label "WEITER" en "L3" voert de volgende transformatie uit:

  $(L[T] < 0) \wedge (TS[T] < 0) \rightarrow (L[T] > 0) \wedge (TS[T] > 0)$

  $(L[T] < 0) \wedge (TS[T] > 0) \rightarrow$ verandert niet

  $(L[T] > 0) \wedge (TS[T] > 0) \rightarrow (L[T] < 0) \wedge (TS[T] > 0)$

  $(L[T] > 0) \wedge (TS[T] < 0) \rightarrow (L[T] > 0) \wedge (TS[T] > 0)$

ad blz. 203: - Als een positieve cykel is ontdekt, STOP;

- Als geen positieve cykel is ontdekt, vul het array TF met de waarde van TS en vul het array TS met de grootste vms-tijd. (dit overhevelen werd waarschijnlijk nodig geoordeeld, om tweemaal met dezelfde programma-tekst te kunnen werken !)

ad blz. 204: - Alle pijlen van de cykel krijgen de lengte "o". De bedoeling hiervan is, dat men de positieve cykel laat verdwijnen, opdat men met het programma verder kan gaan en eventueel méér positieve cykels kan vinden. In ons gewijzigd algorithme (hoofdstuk 5) hebben wij dit voor de eenvoud weggelaten. In de uiteindelijke versie (hoofdstuk 10) is wel iets dergelijks geïmplementeerd. Men zie ook onder "bezwaar c."

- Merk op dat onze datastruktuur dit zoeken naar een pijl van HILF naar A met een factor n versnelt.
- De statements die zoeken naar een pijl van A naar PRUEF, midden op de bladzijde, begrijpen wij niet. Men zie onder "bezwaar c".
- De statement "<u>if</u> PRUEF = 1 <u>them</u> <u>goto</u> PROEND, begrijpen wij niet. Bovendien kan bewezen worden, dat PRUEF nooit de waarde 1 kan hebben. Dit bewijs is langdradig en vervelend, en wordt daarom hier weggelaten.
- Het "<u>else</u>"-stuk onderaan wordt binnengegaan, zolang HILF $\neq$ PRUEF;

Bezwaren: Onze bezwaren zijn, afgezien van de bijzonder slechte presentatie van het geheel, de volgende

a) Het streven van Gewald and Sauler, om minieme efficientie voordelen te behalen, ten kosten van de begrijpelijkheid en de bewijsbaarheid van het geheel; wij achten het streven naar efficientie hier alleen gerechtvaardigd als:
   1. De begrijpelijkheid en bewijsbaarheid niet achteruitgaat
      of
   2. De winst in efficientie *een faktor αn of meer* oplevert. ($\alpha > 0$)

b) Gewald and Sauler stellen niet expliciet, dat de aangeboden graaf aan de bereikbaarheidsvoorwaarden moet voldoen. Deze eis is zeer belangrijk, daar het algorithme anders niet eindig is; de positieve cykel hoeft dan nl. niet op het kritieke pad te komen.

c) Het korrigeren van de lengten der pijlen gebeurt niet korrekt. Het is immers goed mogelijk, dat een negatieve pijl die de lengte o krijgt, een nieuwe positieve cykel genereert! Daarom hebben wij er in hoofdstuk 10 voor gekozen, de lengte van één pijl te verminderen met de totale cykellengte.

d) Tijdens de vms-berekening wordt in het array TF het minteken gebruikt als boolean. Deze truc veroorzaakt een fout, als de gebruiker een korrect netwerk heeft aangeboden, waarin één van de vms tijden negatief is!

Appendix B 2: De efficientie van het Gewald-Sauler algorithme.

Stelling 1: *veronderstel, dat*

a) het netwerk syntaktisch korrekt is;

b) het netwerk een éénduidig bepaald kritiek pad bevat met r pijlen.

c) de berekening van de vms-tijden a iteraties vraagt en de berekening van de lts-tijden b iteraties;

dan geldt: $a + b \geq r + 3$

Bewijs: Zij het kritieke pad: Vo, p_1, V_1, ..., V_{r-1}, P_r, V_r. (Hierbij is Vo = START en V_r = FINISH). Bij de berekening van de vms-tijden krijgen de knooppunten in de volgorde V_0, V_1, ..., V_r hun definitieve vms-tijden, omdat het kritieke pad éénduidig bepaald is, dus kan es $[V_i]$ pas zijn definitieve waarde krijgen, *nadat* es $[V_{i-1}]$ zijn definitieve waarde heeft gekregen, doordat pijp P_i in rekening wordt gebracht. Evenzo krijgen de knooppunten bij de berekening van de lts-tijden in de volgorde V_r, ..., V_0 hun definitieve lts-tijden.

In fig. B1 staan de pijlen van graaf afgebeeld in de volgorde, waarin zij zowel bij de berekening van de vms-tijden als bij de berekening van de lts-tijden worden afgewerkt.

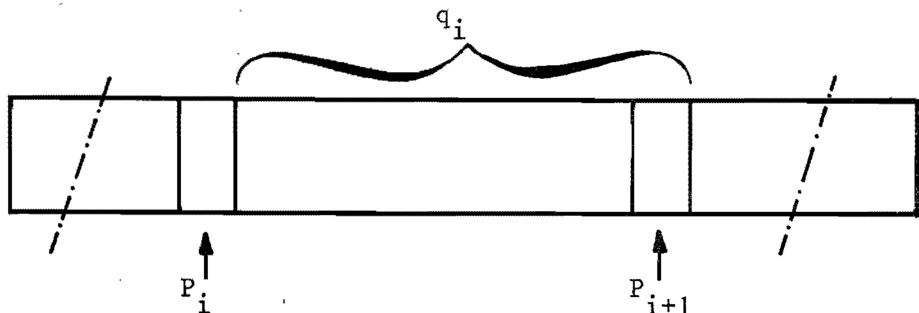


fig.B2.1

De pijlen P_i en P_{i+1} zijn apart weergegeven. Wanneer es$[V_i]$ zijn definitieve waarde heeft, moeten q_i pijlen in beschouwing zijn genomen, vóórdat es $[V_{i+1}]$ zijn definitieve waarde heeft.

Wanneer ls $[V_i]$ zijn definitieve waarde heeft, moeten $m-q_i$ pijlen in beschouwing zijn genomen, vóórdat ls $[V_{i-1}]$ zijn definitieve waarde heeft. Definieer q_i als het aantal pijlen van P_i tot P_{i+1} ($1 \leq i < r$) en q_0 en q_r als resp., het aantal pijlen van het begin van de lijst tot P_1 en het aantal pijlen van P_r tot aan het eind van de lijst. Dan is het aantal pijlen, dat tijdens de berekening van de vms-tijden in beschouwing wordt genomen, tenminste: $\sum_{i=0}^{r} q_i$

en het aantal pijlen dat tijdens de berekening van de lts-tijden in beschouwing wordt genomen: $\sum_{i=0}^{r} (m - q_i)$

Dus het totale aantal pijlen dat in beschouwing wordt genomen is minstens: $(r + 1)\, m$. Er zijn dus tenminst $r + 1$ iteraties plus nog 2 iteraties om te konstateren dat geen verandering meer optreedt. q.e.d.

Wij zijn hierbij uitgegaan van het originele Gewald-Sauler algorithme, waarbij iedere pijl altijd "in beschouwing" (Hfst. 5) genomen wordt. In de door ons gewijzigde versie kunnen door de door ons gekozen datastruktuur een aantal pijlen worden overgeslagen en hoeft bovenstaande ondergrens niet te gelden.
Het is interessant te vermelden, dat de poging tot efficientie-verbetering van Gewald-Sauler:
- indien de vms-tijd van een knooppunt in de vorige iteratie niet is gewijzigd, breng dan zijn uitgaande pijlen niet in rekening (Appendix B1),

  averechts kan werken.

Wanneer b.v.:
- Pijl P_i staat op plaats j in de lijst
- Pijl P_{i+1} staat op plaats k in de lijst
- $j < k$ staat
- in de vorige iteratie is de vms-tijd van V_{i+1} niet gewijzigd
- in de huidige iteratie is de vms-tijd van V_{i+1} wel gewijzigd

dan wordt P_{i+1} ten onrechte niet in rekening gebracht.

Appendix C: Bewijs van correctheid van het algorithme.

Inleiding.

Het bewijs van korrektheid van het onderhavige algorithme zal niet altijd even gedetailleerd worden gegeven als het bewijs van het algorithme SC-Wolker in [10]. Hiervoor zijn een aantal praktische overwegingen:

- het geheel zou te uitgebreid worden om nog door iemand te worden gelezen.
- het algorithme SC-Worker is te beschouwen als een basis-algorithme dat in verschillende varianten in een groot aantal toepassingen gebruikt kan worden. Het is verantwoord aan het bewijs van korrektheid van basis-algorithmen bijzondere zorg te besteden.
- EMPM-Worker bestaat voor een groot deel uit vrij simpele administratie. Het formaliseren van al deze administratieve handelingen zou het bewijs langdradig maken. Anderzijds is het voor de lezer vrij simpel te verifieren (b.v. of de boolean variabele "correct" de juiste waarde heeft).

Het onderhavige algorithme heeft als basis-aanname, dat de datastruktuur, bestaande uit de arrays: *fe*, *le*, *suc*, *length* en uit de integer *n* inderdaad een graaf representeert. I.h.a. zullen deze data worden gegenereerd door een programma, dat wij de *netwerk generator* zullen noemen. De programmeur van deze netwerk generator dient derhalve te bewijzen, dat zijn programma *altijd* iets genereert, dat een graaf representeerd.

In de volgende stellingen en bewijzen gebruiken wij de term "bereikbaar" ook m.b.t. sterke komponenten. "Vanuit SC_i is FINISH bereikbaar" betekent: "vanuit ieder knooppunt behorende tot SC_i is FINISH bereikbaar."

1. Korrektheid van het deel-algorithme: "Decompose, orden SC and Check"

Het deel-algorithme gaat ervan uit, dat de globale boolean: "correct" in het hoofdprogramma de waarde: true heeft gekregen.
Wij beschouwen het programma, zoals afgedrukt op pagina 40.
Merk op, dat de statements die gelabeled zijn met CHECKn en ORDENn de grootheden: stack [i], badge [i], ploftop, niet wijzigen.

Stelling 1.0: In het algorithme SC-Worker geldt:
SC_j is bereikbaar vanuit SC_i , $j \leq i$
De index slaat op de volgorde van het vinden van de sterke komponenten.

Bewijs: Het bewijs kan geleverd worden door deze stelling toe te voegen aan PROP F,[10].

Stelling 1.1: Het deel algorithme: "Decompose, orden SC and check", vindt de sterke komponenten op analoge wijze als SC Worker.

Bewijs: Wanneer men de statements, die gelabeled zijn met CHECKn en ORDENn weglaat, verkrijgt men vrijwel de tekst van SC-Worker. Het enige verschil bestaat hierin, dat "Decompose orden SC and Check" begint met een aanroep van "extend (start)". Wanneer men zou hernummeren, zodat "start" de waarde 1 krijgt, zou de procedure SC-Worker exact hetzelfde uitvoeren als "Decompose, orden SC and Check".

Stelling 1.2: Als een SC niet bereikbaar is vanuit START, krijgt boolean "correct" de waarde false, en wordt deze SC gerapporteerd.

Bewijs: Triviaal.

Stelling 1.3: Op het moment, dat SC_k gevonden wordt, geldt direkt na de statement CHECK 6:
$\forall V , V \in SC_i , 1 \leq i \leq k$;
FR [V] $\leftrightarrow$ FINISH is bereikbaar vanuit V

Bewijs: Wij zullen het bewijs leveren m.b.t. induktie.

Wij onderscheiden de momenten CHECK 6_1 ... CHECK 6_p
en de wortels van de SC's: W_1 ... W_p
Aan het begin van het algorithme geldt:
FR [FINISH] (initialisatie).
Beschouw het moment CHECK 6_1:

Twee mogelijkheden:
- W_1 = FINISH $\rightarrow$ direct na CHECK 6_1 geldt de stelling; triviaal.
- $W_1 \neq$ FINISH: twee mogelijkheden:
  - FINISH is bereikbaar vanuit W_1
    $\rightarrow$ FINISH ε SC_1 (stelling 1.0)
    $\rightarrow$ er is een "aktueel pad" geweest van W_1 naar FINISH
    $\rightarrow$ FR [W_1] wegens CHECK 5 en initialisatie
    $\rightarrow$ na CHECK 6_1: $\forall$ V, V εSC_1: FR [V]
  - $\neg$ FINISH bereikbaar vanuit W_1
  $\leftrightarrow$ $\forall$ V, V ε SC_1 : $\neg$FINISH bereikbaar vanuit V
  $\rightarrow$ er is niet een aktueel pad geweest van enige V naar FINISH
  $\rightarrow$ $\forall$ V, V ε SC_1: $\neg$ FR [V]

De stelling geldt op het moment direct na CHECK 6_1.

Stel de stelling geldt voor alle i : $1 \leq i \leq k < p$
op de momenten direkt na CHECK 6_i.
Beschouw het moment CHECK 6_{k+1}. Merk op dat na uitvoering van CHECK 6_k de waarden van FR [V]:
V ε SC_i, $i \leq k$ niet meer veranderen.
Wij moeten dus nog bewijzen, dat de stelling geldt direkt na uitvoering van CHECK 6_{k+1} voor alle V : Vε SC_{k+1}; twee mogelijkheden:
- W_{k+1} = FINISH triviaal, zie bij CHECK 6_1
- $W_{k+1} \neq$ FINISH, twee mogelijkheden
  - FINISH ε SC_{k+1} bewijs analoog aan bewijs bij CHECK 6_1
  - FINISH $\notin$ SC_{k+1} $\rightarrow$ bij initialisatie geldt: $\neg$ FR [V] voor alle V : Vε SC_{k+1}.

Twee mogelijkheden :

- FINISH is bereikbaar vanuit W_{k+1}

  → $\exists$ pad $W_{k+1}, p_1, V_1, \ldots, V_r, p_{r+1}$, FINISH.
  Stel p_l is de eerste ISC-pijl in dit pad van SC_{k+1}, naar een andere SC : SC_m.

  → "CHECK 5" voor pijl p_l vindt plaats na CHECK 6_m ;
  uit induktie-voorwaarde volgt:

  → FR $[V_l]$ op moment CHECK 5 (p_l)

  → FR$[V_{l-1}]$ op moment CHECK 5 (p_l)

  → FR $[W_{k+1}]$ direct voor CHECK 6, wegens doorgeven langs aktueel pad

  → $\forall$ V ,V ε SC_{k+1} :FR[V] na afloop van CHECK 6.

- $\neg$ FINISH bereikbaar vanuit W_{k+1}

  → $\forall$ V , V $\in$ SC_{k+1} : $\neg$ FINISH bereikbaar vanuit V

  → $\nexists$ pijl p , b(p) ε SC_{k+1}, e (p) $\notin$ SC_{k+1} :
  FR [e(p)] op moment CHECK 5 (p)

  → $\forall$ V , V ε SC_{k+1} : $\neg$ FR [V].
  m.a.w.FR[V] is sinds de initialisatie niet veranderd.

Stelling 1.4: Als FINISH niet vanuit SC_i bereikbaar is, krijgt de boolean "correct" de waarde <u>false</u> en wordt deze SC gerapporteerd.

Bewijs: triviaal.

Stelling 1.5: Indien het netwerk aan de bereikbaarheidsvoorwaarden voldoet geldt:
Van iedere SC wordt één knooppunt in het array SClist geplaatst; het aantal knooppunten van de SC wordt genoteerd op de bijbehorende plaats in het array "size of SC". Het aantal SC's van de graaf wordt uiteindelijk weergegeven door de waarde van de globale variabele: p.

Bewijs: triviaal.

C2. Korrectheid van het deel-algorithme: Split arcs and orden nodes.

Het deel-algorithme wordt alleen uitgevoerd, als de graaf aan de bereikbaarheidsvoorwaarden voldoet; d.w.z., wanneer de boolean "correct" na uitvoering van "decompose, orden SC and check" de waarde true heeft. Daarom wordt bij de volgende bewijzen aangenomen, dat de graaf aan de bereikbaarheidsvoorwaarden voldoet.

Beschouw het moment, waarop het algorithme begint aan de verwerking van de samengestelde sterke component SC_i. Veronderstel dat de knooppunten die deel uitmaken van SC_h, $h < i$ behoren tot de verzameling READY; veronderstel verder, dat de knooppunten van SC_k, $k \geq i$, behoren tot de verzameling UNSPLIT.
Onder deze voorwaarden bewijzen wij stelling 2.1 t/m 2.5; in stelling 2.6 zullen wij bewijzen, dat aan bovenstaande voorwaarden *altijd* is voldaan.

Stelling 2.1: Zij W_i de wortel van SC_i, dan is de aanroep van "split and orden (W_i)"eindig.

Bewijs: Bij aanroep van:"split and orden (W_i)" hebben alle knooppunten van SC_i de status: UNSPLIT. De procedure "split and orden (successor)" wordt alleen aangeroepen als successor ε UNSPLIT, en direct na aanroep krijgt knooppunt successor de status "SPLITTING". Geen enkel knooppunt krijgt ooit opnieuw de status UNSPLIT. Dus het aantal aanroepen van "split and orden (V)"is eindig. De while-statement binnen de procedure "split and orden (V)" is eindig, omdat na iedere slag óf de variabele"index" met 1 wordt opgehoogd, óf de variabele: "index" dezelfde waarde behoudt, maar de bovengrens met 1 wordt verlaagd.

Stelling 2.2: In de while-statement binnen de procedure: "split and orden (V)" worden alle uitgaande pijlen éénmaal beschouwd.

De compound -statement, gelabeled:ARC TO BW, voegt de onderhavige uitgaande pijl van V toe aan het BW-deelnetwerk.

De compound -statement gelabeled: ARC TO ISC, voegt de onderhavige uitgaande pijl van V toe aan de verzameling pijlen ISC.

De empty-statement, gelabeled: ARC to FW, I.E. DO NOTHING, voegt de onderhavige pijl van V toe aan de verzameling FW.

Bewijs: Het bewijs kan gemakkelijk worden geleverd met induktie naar de waarde van "index", en met induktie-bewering:

aan het begin van ieder slag van de while-statement geldt:

- de nog niet beschouwde pijlen staan op de plaatsen index, ..., lefw [V];
- de pijlen, die reeds beschouwd zijn, en "backward" zijn bevonden, staan op de plaatsen: fe [V],..., lebw [V].
- de pijlen, die reeds beschouwd zijn, en tot het "forward" deelnetwerk bleken te behoren staan op de plaatsen: lebw [V] + 1, ..., index - 1;
- de pijlen, die reeds beschouwd zijn, en tot de verzameling "ISC" bleken te behoren, staan op de plaatsen: lefw [V] + 1, ..., fe [V] .

Stelling 2.3: Tussen de aanroep van "split and orden (W_i)" en de terugkeer, komen alle knooppunten van SC_i precies éénmaal in het aktuele pad. Knooppunten uit andere sterke komponenten komen niet in het aktuele pad.

Bewijs: "Tenminste eenmaal": ieder knooppunt van SC_i is bereikbaar vanuit W_i en behoort bij aanroep van "split and orden (W_i) tot de verzameling UNSPLIT.

"Slechts eenmaal": zie bewijs van stelling 2.1.

Knooppunten uit andere sterke komponenten, die vanuit W_i bereikbaar zijn, behoren volgens stelling 1.0 tot sterke komponenten SC_h, $h < i$, en behoren volgens de aanname tot de verzameling READY.

Stelling 2.4: Na terugkeer uit "split and orden (W_i)" behoren alle knooppunten van SC_i tot de verzameling SPLIT en staan als ononderbroken reeks in het array Vlist.

Bewijs: Dit volgt uit de laatste 3 statements van "split and orden (V)", in kombinatie met stelling 2.3.

Stelling 2.5: Nummer de knooppunten van SC_i in de volgorde, waarin zij, na terugkeer uit "split and orden (W_i)" in het array Vlist staan: $V_1, \ldots, V_r$ (zie fig. 9.1.1): als P een pijl is, zodanig dat $b(P) = V_k$, geldt:

(a) $P \in BW_i$
$\quad \rightarrow \quad e(P) \in \{V_1, \ldots, V_{k-1}\}$

(b) $P \in FW_i$
$\quad \rightarrow \quad e(P) \in \{V_{k+1}, \ldots, V_r\}$

(c) $P \in ISC_i$
$\quad \rightarrow \quad e(P)$ behoort tot SC_h, $h < i$.

Bewijs: ad (a): Op het moment, waarop P in beschouwing wordt genomen, is b (P) het eind van het aktuele pad, en zit e (p) vóór b (P) in het aktuele pad. Dus b (P) wordt eerder dan e (P) uit het aktuele pad verwijderd; dus e (P) heeft een lager rangnummer dan "k" in Vlist.

ad (b): Op het moment, dat het algorithme *begint* met P in beschouwing te nemen, behoort e (P), òf tot de verzameling UNSPLIT òf tot de verzameling SPLIT. Op het moment, dat het algorithme *ophoudt* met het in beschouwing houden van P, behoort e (P) tot SPLIT. Dus e (P) heeft een hoger rangnummer dan "k" in Vlist.

ad (c): zie stelling 1.0

Stelling 2.6: De stellingen 2.1 t/m 2.5 gelden *altijd*.

Bewijs: Wij zullen bewijzen, dat aan de gestelde voorwaarden altijd is voldaan d.m.v. induktie naar sterke komponenten.

Induktie voorwaarde: Op het moment, dat het algorithme begint aan de verwerking van SC_i, geldt:

1) $V \varepsilon SC_h$, $h<i \rightarrow V \varepsilon$ READY

2) $V \varepsilon SC_h$, $h \geq i \rightarrow V \varepsilon$ UNSPLIT

De bewering is waar voor i = 1.

Stel, de bewering is waar voor i $i \leq k$.

Dan is de bewering waar voor i = k + 1 op grond van de statement in het hoofdprogramma:

```
for j:= bottom step 1 until  bottom + size of SC [i] -1
   do status [V list [j] ] : = ready;
```

C3. *Korrektheid van het deel-algorithme: compute es, check and find cp.*

Definitie 3.1: De *pusher-graaf* is een deelgraaf van het netwerk zodanig dat:

(1) alle knooppunten van het oorspronkelijk netwerk behoren tot de pusher-graaf;

(2) W = pusher [V] $\neq$ o $\leftrightarrow$
$\exists$ pijl P in de pusher-graaf ; b (P) = W, e(P) = V;

De pusher-graaf verandert voortdurend tijdens de uitvoering van "Compute es, check and find cp". Aan het begin van het deel-algorithme bevat de pusher-graaf geen enkele pijl; aan het einde heeft ieder knooppunt behalve START precies één binnenkomende pijl, althans als het netwerk syntaktisch korrekt is.

In de volgende stellingen, in het bijzonder 3.1, 3.2 en 3.5, wordt ervan uitgegaan, dat gedurende het algorithme de volgende eigenschap geldt:
$\forall$ V : pusher [V]=W$\neq$O $\rightarrow$ $\exists$ pijl P, b (P) = W, e (P) = V:
es [W] + length [P] $\geq$ es [V]
Bij het initialiseren van het array pusher geldt de eigenschap.
Bij het, in rekening brengen van pijlen wordt deze eigenschap niet verstoord. Bij "REPORT AND REPAIR POSITIVE CYCLE" wordt er zorg voor gedragen dat de eigenschap behouden blijft: zie bewijs van stelling 3.6.

Stelling 3.1: Wanneer de pusher-graaf op enig moment een cykel bevat, bevat het netwerk een positieve cykel door dezelfde knooppunten.

Bewijs: Laat de cykel in de pusher-graaf gevormd worden door de knooppunten V_o, ..., V_k. Zij V_o het knooppunt, waarvan de vms-tijd als laatste is opgehoogd. Zij P_i de langste pijl tussen knooppunt V_i en knooppunt V_{i+1} ($o \leq i \leq k-1$) en P_k de langste pijl van V_k naar V_o. Merk op, dat geldt:
V_i = pusher $[V_{i+1}]$ $\rightarrow$ es $[V_i]$ $\neq$ length $[P_i]$ $\geq$ es $[V_{i+1}]$
Op het laatste moment, vóór dat vms $[V_o]$ werd verhoogd, gold:

$$\sum_{i=o}^{k-1} (\text{es } V_i + \text{length}[P_i]) \geq \sum_{i=o}^{k-1} \text{es } [V_{i+1}] \quad \ldots\ldots (1)$$

Bovendien gold, omdat V_o naderhand inderdaad werd opgehoogd:

$$\text{es}[V_k] = \text{length}[P_k] > \text{es}[V_o] \quad \ldots\ldots (2)$$

(1) en (2) gesommeerd levert $\sum_{i=o}^{k} \text{length}[P_i] > o$, q.e.d.

Stelling 3.2: Wanneer het netwerk syntaktisch korrekt is, wordt het array: "es" gevuld met de juiste waarden. Een kritiek pad wordt gevormd door de reeks knooppunten: FINISH, pusher [FINISH] , pusher [pusher [FINISH]],...,START

Bewijs: Wij schetsen een globaal bewijs met induktie. Bij de gekozen volgorde van de SC's geldt:

- bij het iteratief doorrekenen van de FW_i en BW_i pijlen en bij het eenmalig in rekening brengen van de ISC_i-pijlen veranderen de vms-tijden van de knooppunten van SC_j, $j < i$ niet, Als deze knooppunten dus eenmaal de juiste vms-tijden hebben, blijft de juistheid behouden.
  (analogie algorithme voor cykelvrije netwerken)
- na het iteratief doorekenen van FW_i en BW_i zijn de vms-tijden van de knooppunten van SC_i korrekt.
  (analogie Gewald-Sauler en Ford-Fulkerson algorithme)

Resulterend geldt, dat na afloop de vms-tijden van alle knooppunten korrekt zijn d.w.z.:

- $\forall$ j: es [b (P_j)] + length [P_j] $\leq$ es [e (P_j)], $1 \leq j \leq m$
  (er wordt voldaan aan de ongelijkheden)
- $\forall_i : \exists j$: e (P_j) = i, b (P_j) = pusher [i],
  es [pusher [i]] + length [P_j] = es [i],
  $1 \leq i \leq n$, i $\neq$ START
  $\rightarrow$ vms-tijden zijn minimaal, ieder pad in de pushergraaf is een kritiek pad.

Omdat de pusher-graaf geen cykels bevat (stelling 3.1) en ieder knooppunten V_i behalve START, de eigenschap: pusher [V] $\neq$ o heeft, eindigt iedere reeks knooppunten V, pusher [V] , pusher [pusher [V]], ..., bij START

Stelling 3.3: Wanneer een sterke komponent, bestaande uit q knooppunten geen positieve cykel bevat, heeft het algorithme ten hoogste q + 1 slagen nodig voor de verwerking van deze sterke komponent.

Bewijs: Beschouw de verzameling van paden, die van START naar een knooppunt V_Π in de sterke komponent lopen. Een deelverzameling hiervan is de verzameling van *langste* paden, Π_1, ..., Π_k, ... die van start naar V_π lopen.

Wij beschouwen de deelverzameling $\hat{\pi}_1$, ... $\hat{\pi}_k$,.... van paden; hierin is $\hat{\pi}_i$ dat gedeelte van π_i dat binnen de SC ligt.
Neem een willekeurig cykelvrij pad $\hat{\pi}_i$
Noteer *het aantal pijlen van* $\hat{\pi}_i$ als $|\hat{\pi}_i|$.
Wij kleuren in gedachte de pijlen die tot FW_i behoren rood, en de pijlen die tot BW_i behoren blauw. $\hat{\pi}_i$ bestaat dus afwisselend uit rode en blauwe stukken.
Indien $\hat{\pi}_i$ rood begint, krijgen de knooppunten van het eerste rode stuk hun definitieve vms-tijden; anders doet de eerst slag m.b.t. pad $\hat{\pi}_i$ niets. Uiterlijk in de tweede slag krijgen de knooppunten van het eerste blauwe stuk hun definitieve vms-tijden; uiterlijk in de derde slag krijgen de knooppunten uit het tweede rode stuk hun definitieve vms-tijden, etc. In het ongunstigste geval, nl. wanneer de pijlen van $\hat{\pi}_i$ afwisselen blauw en rood zijn en de eerste pijl blauw is, krijgt V_π na ten hoogste $|\hat{\pi}_i|$ +1 slagen zijn definitieve vms-tijd.
Dit geldt voor ieder knooppunt V_π . Dus na ten hoogste q slagen hebben alle knooppunten hun definitieve vms-tijd. Er is nog één slag nodig, om te konstateren dat geen verandering meer optreedt.

Definitie 3.2: de reeks knooppunten: pusher [V] , pusher [pusher [V]] ,... noemen wij de *pusher-reeks van V*.

Indien de pusher-graaf cykels bevat, behoeft bovenstaande reeks niet eindig te zijn. Indien de pusher-graaf geen cykels bevat is de reeks eindig en is het laatste element = o.

Stelling 3.4: Wanneer een sterke komponent SC_i bestaande uit q knooppunten, positieve cykels bezit, bevat de pusher-graaf na ten hoogste q + 1 slagen een cykel.
Indien men deze cykels niet verbreekt, blijft de pusher-graaf cykels bevatten.

Bewijs: Veronderstel, dat de pusher-graaf na q+k slagen geen cykel bevat ($k \geq 1$). Dan is er een knooppunt V, $V \in SC_i$, behorend tot een positieve cykel, zodanig dat na q+k slagen geldt:
- de eerste t ($t \geq o$) knooppunten van de pusher-reeks van V: V_1, V_2, ... behoren wel tot SC_i, maar behoren niet tot een positieve cykel, terwijl V_{t+1} òf =0 is, òf niet tot SC_i behoort.

Dan heeft de grootheid: "pusher[V]" na ten hoogste t+1 slagen de waarde, die deze grootheid na q+k slagen heeft. (Zie bewijs van stelling 3) Wanneer de cykel c knooppunten heeft, verandert na ten hoogste c volgende slagen de waarde van pusher [V]. Dus: $t + 1 + c > q + k$. Tevens geldt $t + c \leq q$, het totaal aantal knooppunten van SC_i. Kontradiktie wegen $k \geq 1$.

Stelling 3.5: Een positieve cykel wordt slechts éénmaal gerapporteerd.

Bewijs: In de procedure "REPORT AND REPAIR POSITIVE CYCLE" wordt telkens de lengte van de langste pijl tussen "pusher[V]" en "V" bij de variabel "cyclelength" opgesteld om de cykel-lengte te verkrijgen. Deze cykellengte wordt afgetrokken van de laatste langste pijl tussen V en de direkte opvolger W van V in de cykel. De grootheid: "pusher [W]" wordt =0 gemaakt. Hierdoor is wederom voldaan aan de eigenschap:

$\forall$ V: pusher [V] = W $\rightarrow \exists$ pijl P, b(P) = W, e (P) = V:
es [W] + length [P] $\geq$ es [V].

De positieve cykel is uit de pushergraaf en uit het netwerk verwijderd. Wegens stelling 3.1 en 3.4 zal dezelfde cykel niet opnieuw in de pusher-graaf komen.

Als "parallel" aan de ingekorte pijl nog een tweede pijl loopt, kan deze eveneens tot een cykel-rapport aanleiding geven - maar dan bevat het oorspronkelijke netwerk twee verschillende positieve cykels door dezelfde knooppunten.

Stelling 3.6: Door het veranderen van de lengten der pijlen ontstaan geen nieuwe positieve cykels.

Bewijs: Triviaal. De stelling wordt slechts vermeld, omdat het Gewald-Sauler algorithme *alle* pijlen van een positieve cykel de lengte nul geeft - ook de negatieve pijlen. Hierdoor kunnen nieuwe positieve cykels gegenereerd worden.

Stelling 2.7: Na: "Compute es, check and find Op" geldt:

correct $\leftrightarrow$ het netwerk is **syntaktisch korrekt.**

Bewijs: Triviaal.

C4. Correctheid van het deel-algorithme: Compute ls.

Wij gaan er bij deze bewijzen van uit, dat het deel-algorithme alleen wordt aangeroepen als het netwerk géén positieve cykels bevat. In ons hoofdprogramma (hoofdstuk 7) wordt het deel-algorithme "compute ls" dan ook alleen aangeroepen als de boolean "correct" de waarde true heeft.

Definitie 4.1: De *holder-graaf* is een deelgraaf van het netwerk met de eigenschap :

(1) alle knooppunten van het oorspronkelijke netwerk behoren tot de holder-graaf.

(2) $W = \text{holder}\ [V] \neq 0 \rightarrow$
$\exists$ pijl P in de holder-graaf, $b(P) = V$, $e(P) = W$;

De holder-graaf verandert voortdurend tijdens de uitvoering van "compute ls"; aan het begin van het deel-algorithme bevat de holder-graaf geen enkele pijl; aan het einde heeft ieder knooppunt behalve FINISH precies één uitgaande pijl.

In de volgende stellingen wordt ervan uitgegaan, dat gedurende het algorithme de volgende eigenschap geldt:

$\forall W : \text{holder}\ [W] = V \neq 0 \rightarrow \exists \text{pijl } P,\ b(P) = W,\ e(P) = V:$
$\text{ls}\ [W] + \text{length}\ [P] \geq \text{ls}\ [V].$

Bij het initialiseren van het array holder geldt de eigenschap.
Bij het in rekening brengen van de pijlen blijft deze behouden.

Stelling 4.1: Het array "le" wordt gevuld met de juiste waarden.
Een kritiek pad wordt gevormd door de knooppunten: START, holder [START], holder [holder [START]], ... FINISH.

Bewijs: Wij schetsen een globaal bewijs met induktie. Bij de gekozen volgorde van SC's geldt:

- bij het eenmalig in rekening brengen van de pijlen uit ISC_i, en bij het iteratief doorrekenen van de pijlen uit FW_i en BW_i veranderen de lts-tijden van de knooppunten van SC_j, $j \neq i$ niet.

Als de knooppunten behorend tot SC_j, $j>i$ eenmaal de juiste lts-tijden hebben, blijft de juistheid behouden. (analoog aan het algorithme voor cykel-vrije netwerken)

- Na het iteratief doorrekenen van FW_i en BW_i zijn de lts-tijden van de knooppunten van SC_i korrekt. (analoog aan het Gewald-Sauler en Ford-Fulkerson-algorithme)

Resulterend geldt, dat na afloopr de lts-tijden van alle knooppunten korrekt zijn, d.w.z.:

- $\forall$ j: ls [b (P_j)] + length [P_j] $\leq$ ls [e (P_j)], $1 \leq j \leq m$
  (er wordt voldaan aan de ongelijkheden)
- $\forall_i : \exists j$: b (P_j) = i, e (P_j) = holder [i],
  ls [i] + length [P_j] = ls [holder [i]],
  $1 \leq i \leq n$, $i \neq$ FINISH
  De lts-tijden zijn maximaal, ieder pad in de holder-graaf is een kritiek pad.

Omdat de "holder-graaf" geen cykels bevat (bewijs analoog aan stelling 3.1), en ieder knooppunt V, behalve FINISH, de eigenschap: holder [V] $\neq$ o heeft, eindigt iedere reeks knooppunten:
V, holder [V], holder [holder [V]], ..., bij FINISH.

Stelling 4.2: Voor een sterke komponent, bestaande uit q knooppunten heeft het algorithme ten hoogste q+1 slagen nodig.
Bewijs: dit gaat volledig analoog aan het bewijs van stelling 3.3.

Zoals in hoofdstuk 12 is uiteengezet, kan men vanweg de data-struktuur bij de berekening der lts-tijden moeilijk pijlen P overslaan, als ls [e (P)] ongewijzigd blijft. Daarom zal de hoeveelheid werk *per slag* bij de berekening der vms-tijden meestal kleiner zijn dan bij de berekening van de lts-tijden.
Anderzijds werkt het efficientie verbeterend, dat wij bij alle knooppunten van het kritieke pad, de lts-tijd en de vms-tijd aan elkaar gelijk stellen.

Het zou ook mogelijk zijn een soortgelijke maatregel te nemen bij ieder knooppunt, en wel als volgt:

stel, dat de lts-tijd van een knooppunt V verandert; stel, na deze verandering geldt: delta = ls [V] - es [V]. Men loopt nu de pusher-reeks af, V_1, V_2, ...en voert het volgende uit:

```
V: = pusher [V];
while ls [V] + es [V] > delta do
begin ls [V] : = es [V] + delta; V: = pusher [V] end;
```

Appendix C5: efficientie

In Appendix B2 is bewezen, dat een ondergrens voor het aantal iteraties bij het Gewald-Sauler algorithme gegeven wordt door r+3. Hierbij is r het aantal pijlen van het kritieke pad, en wordt een iteratie gevormd door m standaard-handelingen. Deze ondergrens werd afgeleid onder de voorwaarde, dat er één kritiek pad van START naar FINISH is. (Voorwaarde (a)).
Om een vergelijking met EMPM-Worker mogelijk te maken, stellen wij nog een extra konditie: het aantal pijlen van het kritieke pad van START naar FINISH $\geq$ het aantal pijlen van een kritiek pad van START naar een willekeurig knooppunt i, of van een willekeurig knooppunt naar FINISH. (Voorwaarde (b)). Het nut van de voorwaarden (a) en (b) is:

- voorwaarde (a) is nodig om een eenvoudige formule te geven als een ondergrens voor het aantal iteraties van het Gewald-Sauler-algorithme;
- wanneer men een bovengrens wil afleiden voor EMPM-Worker, uitgedrukt in de parameter r, moet er een nevenkonditie aan r worden opgelegd. Immers men kan bij een gegeven waarde van r een willekeurige ingewikkelde graaf konstrueren, en zodoende het aantal iteraties willekeurig groot krijgen.
- wanneer men wenst, dat de ondergrens voor het Gewald-Sauler algorithme enigermate scherp is, moet men eveneens nevenkondities aan r opleggen, om dezelfde reden als boven.
- Zelfs bij ernstige afwijking van deze voorwaarden,zoals die in de praktische netwerk-planning zeker niet voor zullen komen, veranderen de eindkonklusies weinig. Zelfs bij geringe afwijking van de voorwaarden (a) en (b), is het nodig een groot aantal nieuwe definities in te voeren en worden de formules onhanteerbaar.

Stelling 5.1: Onder de voorwaarden (a) en (b) wordt een bovengrens van het aantal standaardbehandelingen van het rekengedeelte van EMPM-Worker gegeven door: (r+3)m. Dit komt overeen met r+3 iteraties uit het Gewald-Sauler-algorithme.

Bewijs: Wij maken gebruik van het bewijs van stelling 3.3.
Daarin werd aangetoond, dat binnen de procedure: "Compute es of SC_i" het knooppunt V_π na ten hoogste $|\hat{\pi}_i| + 1$ slagen zijn definitieve vms-tijd heeft.
Een extra slag is nodig om te konstateren dat er geen verandering optreedt.
Volgens voorwaarde (b) geldt: $\forall V_\pi, \forall_i: |\hat{\pi}_i| \leq r$.
Dus de procedure: "Compute es of SC_i" heeft ten hoogste r+2 slagen nodig. Dus iedere pijl van FW_i en BW_i wordt $\frac{1}{2}$ (r+2) maal in beschouwing genomen. Binnen de procedure "Compute es, check and find up" wordt dus maximaal $\frac{1}{2}$ (r+2) m standaard-handelingen verricht. Hetzelfde geldt voor "Compute ls", q.e.d.

Het is bijzonder waarschijnlijk, dat bij praktische netwerkplanning de rekentijd ver onder de bovengrens zal blijven. Opdat de rekentijd de bovengrens bereikt, moet simultaan aan de volgende voorwaarden zijn voldaan:

- het netwerk moet sterk verbonden zijn.
- er moet een kritiek pad zijn van START naar een knooppunt i ter lengte r, met de pijlen afwisselend behorend tot het FW- en BW-netwerk;
- er moet een kritiek pad zijn van enig knooppunt j naar FINISH, ter lengte r met de pijlen afwisselend behorend tot het FW en het BW netwerk; dit pad moet tevens geen enkel knooppunt behalve FINISH gemeên hebben met "het" kritieke pad, omdat de lts-tijden van de knooppunten van "het" kritieke pad immers gelijk gemaakt worden aan hun vms-tijden.
- het overslaan van pijlen tijdens "Compute es of SC_i" moet niet effektief zijn.