



rijksuniversiteit
 groningen

faculteit wiskunde en
 natuurwetenschappen

OPWAARTS GESLOTEN DEELVERZAMELINGEN IN PARTIEEL GEORDENDE VERZAMELINGEN

BACHELOR SCRIPTIE INFORMATICA

6 JULI 2012

R. A. TRIESSCHEIJN

BEGELEIDER PROF. DR. G.R. RENARDEL DE LAVALETTE

TWEEDE LEZER DR. A. MEIJSTER

SAMENVATTING

Het bepalen van het aantal opwaarts gesloten deelverzamelingen in een partieel geordende verzameling levert een uniek kengetal op dat gebruikt kan worden om verzamelingen te vergelijken. Dit kengetal is als het ware een vingerafdruk van de verzameling. Tot nu toe is er, zover mij bekend, nog geen slim algoritme om het aantal opwaarts gesloten deelverzamelingen te bepalen. Daarom moest dit kengetal bepaald worden door met brute kracht alle mogelijke deelverzamelingen te controleren op opwaarts geslotenheid. Dit leidt zelfs voor triviale verzamelingen al snel tot problemen aangezien de complexiteit van dit proces altijd $O(2^n)$ bedraagt. Als het aantal opwaarts gesloten deelverzamelingen met de hand uitgerekend wordt kan eventuele symmetrie uitgebuit worden maar bij grotere verzamelingen, of als er geen symmetrie is, blijft dit een tijdrovend en foutgevoelig proces. In deze scriptie presenteer ik een computer algoritme dat exact en, in de meeste gevallen, sneller het aantal opwaarts gesloten deelverzamelingen kan bepalen.

INHOUDSOPGAVE

Introductie.....	1
Begrippen	1
Topologische ordening in partieel geordende verzamelingen.....	2
Een naïeve aanpak.....	4
Een algoritme om opwaarts gesloten deelverzamelingen te vinden en te tellen	5
Aanpak.....	5
Voorbeeld.....	6
Bewijs	9
Lemma's	9
Het Hoofdbewijs.....	10
Vergelijking met de naïeve aanpak	12
De uptrie datastructuur.....	14
De trie en uptrie	14
Zoeken in de uptrie	16
Opwaarts gesloten deelverzamelingen genereren met de uptrie	18
Snelheidsverbeteringen door het gebruik van de uptrie	19
Vergelijkende test	21
Kan het nog sneller?	23
Gerelateerd werk	23
Conclusie en verder onderzoek.....	24
Dankwoord	24
Bibliografie	25
Datasets.....	26
Broncode	30

INTRODUCTIE

In deze scriptie introduceren we een algoritme om de opwaarts gesloten deelverzamelingen, in een partieel geordende verzameling te vinden. Dit algoritme noemen we het ‘familiealgoritme’. We zoeken daarbij ook een efficiënte datastructuur om de opwaarts gesloten deelverzamelingen in op te slaan. Daarnaast vergelijken we twee implementaties van het familiealgoritme met een naïeve, brute force, aanpak om de kracht van het gevonden algoritme en de gevonden datastructuur aan te tonen.

Maar eerst zullen we de verwante begrippen introduceren.

BEGRIPPEN

Bij een opwaarts gesloten deelverzameling gaat het om ordening. Een opwaarts gesloten deelverzameling is dan ook altijd een deelverzameling van een partieel geordende verzameling.

Een partieel geordende verzameling is een paar (P, \leq) . Hierbij is \leq een binaire relatie op P die voldoet aan de volgende eigenschappen voor alle $x, y, z \in P$:

Reflexief: $x \leq x$ voor elke x

Transitief: $x \leq y \wedge y \leq z \rightarrow x \leq z$

Antisymmetrisch: $x \leq y \wedge y \leq x \rightarrow x = y$

Een totale ordening is transitief, antisymmetrisch en heeft de eigenschap totaliteit:

Totaliteit: $x \leq y \vee y \leq x$ voor elke x en y

Als we hier voor x en y hetzelfde element kiezen krijgen we:

$x \leq x$ voor elke x

Een totaal geordende verzameling is dus ook altijd een partieel geordende verzameling maar het omgekeerde is niet waar.

In een totaal geordende verzameling T geldt:

$\forall x, y \in T (x \neq y \rightarrow x > y \vee x < y)$

In een partieel geordende verzameling P kunnen, in het algemeen, niet altijd alle elementen met elkaar vergeleken worden, dat wil zeggen: er kunnen elementen $x, y \in P$ bestaan waarvoor zowel $x \leq y$ en $y \leq x$ niet gelden.

Een deelverzameling S van de partieel geordende verzameling P is opwaarts gesloten als de volgende eigenschap geldt:

$\forall x \in S \forall y \in P (x \leq y \rightarrow y \in S)$

We kunnen partieel geordende verzamelingen weergeven in een Hasse diagram [1]. In een Hasse diagram worden de elementen uit de verzameling afgebeeld als knopen. Een lijn van een lager gelegen knoop x naar een hoger gelegen knoop y geeft de relatie $x < y$ weer. Alleen de een-stap-ordening ($<_1$) wordt afgebeeld:

$$x <_1 y \equiv x < y \wedge \neg \exists z (x < z < y)$$

De oorspronkelijke relatie kunnen we uit het diagram afleiden door de transitieve eigenschap van partieel geordende verzamelingen.

In een Hasse diagram staan bovenin de maximale elementen. Een element x is maximaal als geldt: $\forall y \in P (x \leq y \rightarrow y = x)$. Onderin de tekening staan de minimale elementen. Een element x is minimaal als geldt: $\forall y \in P (y \leq x \rightarrow y = x)$. Een voorbeeld van een Hasse-diagram ziet u in Figuur A.

We definiëren de volgende functies over de partieel geordende verzameling P en element $x \in P$:

$parents(x)$: $\{y \mid y >_1 x\}$, de directe ouders van x

$children(x)$: $\{y \mid y <_1 x\}$, de directe kinderen van x

$max(P)$: $\{x \mid \forall y \in P (x \leq y \rightarrow y = x)\}$, alle maximale knopen in P

$min(P)$: $\{x \mid \forall y \in P (y \leq x \rightarrow y = x)\}$, alle minimale knopen in P

$uppersets(P)$: alle opwaarts gesloten deelverzameling in P

TOPOLOGISCHE ORDENING IN PARTIEEL GEORDENDE VERZAMELINGEN

Om een totale ordening te definiëren over de elementen uit een partieel geordende verzameling lenen we de topologische ordening [2] uit de grafentheorie.

In een gerichte acyclische graaf is de definitie van topologische ordening als volgt:

Laat \vec{G} een gerichte graaf zijn met n knopen. Een topologische ordening van \vec{G} is een nummering (v_1, v_2, \dots, v_n) van de knopen van \vec{G} zodat voor elke kant $(v_i, v_j) \in \vec{G}$ geldt dat $i < j$.

Bij het lopen over een gericht pad in \vec{G} komen we de knopen tegen in oplopende volgorde.

Een gerichte graaf heeft alleen een topologische ordening als deze acyclisch is.

Om deze topologische ordening te gebruiken zullen we een partieel geordende verzameling beschouwen als een gerichte acyclische graaf. Het eerder besproken Hasse-diagram is gericht, transitief en antisymmetrisch. Deze eigenschappen maken een Hasse-diagram ook acyclisch:

Als er een cykel bestaat met $x, y \in P$ op het pad van de cykel dan geldt door de transitieve eigenschap $x \leq y \wedge y \leq x$. De antisymmetrische eigenschap van P geeft ons dan dat $x = y$. Er kan dus geen cykel bestaan in een Hasse-diagram omdat alle elementen op het pad van een cykel dan het zelfde element zijn.

Een Hasse-diagram is dus een gerichte acyclische graaf. We kunnen dus de topologische ordening gebruiken voor partieel geordende verzamelingen.

We gebruiken het algoritme van Michael T. Goodrich en Roberto Tamassia [2] om een topologische ordening van een partieel geordende verzameling te bepalen, de pseudo code is als volgt:

Algorithm TopologicalOrdering(P)

Input partially ordered set P

Output array A with all the elements in P ordered such that

$$A[i] > A[j] \rightarrow i < j$$

$A \leftarrow \emptyset$

$Q \leftarrow \text{empty FIFO queue}$

index $\leftarrow 0$

foreach element **in** max(P) **do**

 Q.Enqueue(element)

while Q is not empty **do**

$z \leftarrow Q.Dequeue()$

$A[\text{index}] \leftarrow z$

 index $\leftarrow \text{index} + 1$

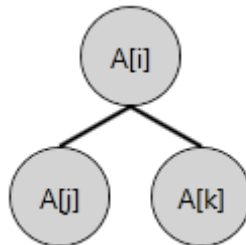
foreach child **in** children(z) **do**

if all elements in parents(child) are in A **then**

 Q.Enqueue(child)

return A

Als we dit algoritme los laten op de partieel geordende verzameling waarvan een deel is afgebeeld in figuur A geldt $A[i] > A[j] \rightarrow i < j$.



Figuur A: een Hasse diagram van een deel van een partieel geordende verzameling. De afgebeelde relaties zijn $A[i] > A[j]$ en $A[i] > A[k]$

We spreken nu af dat er een array $P[]$ is. Dit array heeft basis index 0 en is volgens bovenstaand algoritme geconstrueerd. Dit array bevat alle elementen uit de gelijknamige partieel geordende verzameling P en de elementen in dit array zijn geordend volgens een topologische ordening.

Ook spreken we af dat we op een array al de operator en kwantoren uit de verzamelingenleer kunnen toepassen. We beschouwen een array A dan als de verzameling $\{x \mid x = A[i], 0 \leq i \leq \#A\}$.

EEN NAÏEVE AANPAK

Als basis van vergelijking hebben we eerst een naïef algoritme geïmplementeerd. Dit naïeve algoritme genereert alle mogelijke deelverzamelingen van de ingevoerde partieel geordende verzameling P . Van elke van deze deelverzamelingen wordt gecontroleerd of deze opwaarts gesloten is. Om dit te controleren gaat het naïeve algoritme voor de deelverzameling S na of voor elk element $x \in S$ geldt $\text{parents}(x) \subseteq S$. Als dit voor alle elementen in S waar is dan is S een opwaarts gesloten deelverzameling.

In pseudocode ziet dit er als volgt uit:

Algorithm IsUpperSet(S)

Input set S which is a subset of a partially ordered set

Output true if S is an upper set, false otherwise

foreach element **in** S **do**

foreach parent **in** $\text{parents}(\text{element})$ **do**

if (parent $\notin S$) **then**

return false

return true

Omdat een verzameling met n elementen 2^n deelverzamelingen bevat heeft dit algoritme een tijdscomplexiteit van ten minste $O(2^n)$. Wel gaat dit naïeve algoritme erg efficiënt met het geheugen om, de geheugencomplexiteit van dit geheugen is van orde grote $O(n)$.

Toch is er veel ruimte voor verbetering omdat het naïeve algoritme voor alle partieel geordende verzameling dezelfde tijdscomplexiteit heeft van ten minste $O(2^n)$. De naïeve aanpak is dus zeer inefficiënt maar geeft ons wel een duidelijke basis om nieuwe algoritmen mee te vergelijken.

EEN ALGORITME OM OPWAARTS GESLOTEN DEELVERZAMELINGEN TE VINDEN EN TE TELLEN

AANPAK

Het grote probleem van de naïeve aanpak is dat het aantal mogelijke deelverzamelingen in een verzameling exponentieel groeit als de verzameling groter wordt terwijl het aantal opwaarts gesloten deelverzamelingen niet exponentieel hoeft te groeien. Ook de tijd die nodig is om het algoritme uit te voeren groeit exponentieel als de verzameling groter wordt en staat niet in verhouding tot het aantal opwaarts gesloten deelverzamelingen dat we vinden. In het ideale geval genereren we alleen maar opwaarts gesloten deelverzamelingen.

Het idee voor het algoritme is simpel. Gegeven een partieel geordende verzameling P en een opwaarts gesloten deelverzameling G . Als we een element $x \in P$ kunnen vinden met

$$x \notin G \wedge \text{parents}(x) \subseteq G$$

dan is $\{x\} \cup G$ ook een opwaarts gesloten deelverzameling. Immers bevinden alle ouders van x zich al in G en omdat G een opwaarts gesloten deelverzameling is bevat G ook alle elementen die groter zijn dan de ouders van x , daarom is de toevoeging van x aan G ook een opwaarts gesloten deelverzameling.

We kunnen deze methode herhalen met andere elementen en met de nieuw gegenereerde opwaarts gesloten deelverzamelingen net zolang totdat er voor geen enkele opwaarts gesloten deelverzameling meer een element gevonden kan worden waardoor toevoegen van dit element een nog niet eerder gevonden opwaarts gesloten deelverzameling oplevert.

Het nadeel van deze aanpak is dat we zowel door een steeds groter wordende lijst van opwaarts gesloten deelverzamelingen moeten lopen als dat we voor elk van deze deelverzamelingen door alle kinderen van alle al toegevoegde elementen moeten lopen om te zien of we een nieuwe opwaarts gesloten deelverzamelingen kunnen construeren. Deze aanpak is dus nog steeds erg intensief.

We kunnen deze methode ook omdraaien. In plaats van per opwaarts gesloten deelverzameling te zoeken naar een nieuwe kandidaat om toe te voegen kunnen we ook één maal over alle elementen in de partieel geordende verzameling P lopen. Voor elk element $x \in P$ controleren we alle al gevonden opwaarts gesloten deelverzamelingen G om te zien of $\{x\} \cup G$ een nieuwe opwaarts gesloten deelverzameling oplevert. Om dit te kunnen doen moeten we wel pas x verwerken als $\text{parents}(x)$ al verwerkt is. Anders missen we opwaarts gesloten deelverzamelingen. We kunnen dit doen door een wachtrij bij te houden of door de elementen uit P te verwerken in volgorde van groot naar klein volgens de topologische ordening.

Deze aanpak is nog even intensief als de hiervoor beschreven aanpak maar we weten dat $\{x\} \cup G$ alleen een opwaarts gesloten deelverzameling is als alle ouders van x zich al in G bevinden. Als we, gegeven x , snel alle al gevonden opwaarts gesloten deelverzamelingen G kunnen opvragen waarvoor geldt: $\text{parents}(x) \subseteq G$ in plaats van door alle al gevonden opwaarts gesloten deelverzamelingen te zoeken is deze aanpak een stuk efficiënter. In een later hoofdstuk behandelen we de uptrie-datastructuur die deze vraag snel en efficiënt kan beantwoorden.

Zoals u ziet wordt er veel gebruik gemaakt van de ouder-kind relatie in dit algoritme. Daar komt ook de naam familiealgoritme voor dit algoritme vandaan.

Het familiealgoritme is samen te vatten in de volgende pseudocode:

```
Algorithm FamilyAlgorithm(P)
  Input   partially ordered set P
  Output  the set of upper sets in P

  Z ← {∅}
  i ← 0

  while i < #P do
    H ← {∅}

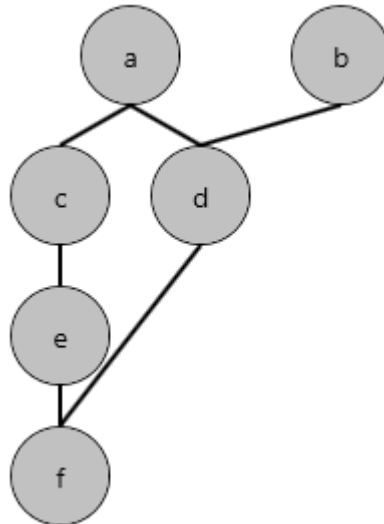
    foreach G in Z do
      if parents(P[i]) ⊆ G do
        H ← H ∪ {{P[i]} ∪ G}

    Z ← Z ∪ H
    i ← i + 1

  return Z
```

We zullen bewijzen dat dit algoritme correct is maar eerst zullen we het algoritme eenmaal stap voor stap demonstreren.

VOORBEELD



Figuur B: een Hasse diagram van een partieel geordende verzameling

Gegeven de in Figuur B afgebeelde partieel geordende verzameling P . We definiëren de volgende drie verzamelingen:

V : alle al door het algoritme verwerkte elementen

Z : $uppersets(V)$

Q : $\{x \mid x \in P \setminus V \wedge parents(x) \subseteq V\}$

Initieel geldt:

$V = \emptyset$

$Z = \{\emptyset\}$

$Q = \max(P)$

In elke iteratie inspecteert het algoritme één element uit Q . Initieel voldoen alleen de maximale elementen uit P aan de definitie van Q . We zullen dus als eerste een maximaal element a verwerken.

Er staat op dit moment één opwaarts gesloten deelverzameling in Z , namelijk de deelverzameling \emptyset . Deze bevat alle ouders van a dus we kunnen deze deelverzameling uitbreiden met a om weer een nieuwe opwaarts gesloten deelverzameling te construeren. Hierna is a verwerkt dus stoppen we deze in V . Dit betekent dat alle ouders van c nu verwerkt zijn en dat c in Q komt. De toestand van het algoritme is nu dus als volgt:

Z	Q	V
\emptyset	b	a
$\{a\}$	c	

Hierna verwerken we element b . Alle opwaarts gesloten deelverzamelingen in Z bevatten alle ouders van b , we kunnen dus elk van deze opwaarts gesloten deelverzamelingen uitbreiden met b om een nieuwe opwaarts gesloten deelverzameling te construeren. Als b eenmaal verwerkt is geldt ook dat alle ouders van d verwerkt zijn. De volgende toestand van het algoritme is dus:

Z	Q	V
\emptyset	c	a
$\{a\}$	d	b
$\{b\}$		
$\{a,b\}$		

Het volgende element in Q is c . Dit element heeft als ouder a , aan elke opwaarts gesloten deelverzameling waar a in voorkomt kunnen we dus ook c toevoegen om een nieuwe opwaarts gesloten deelverzameling te construeren. Verder kunnen we alle kinderen van c waarvan alle ouders verwerkt zijn toevoegen aan Q . De toestand van het algoritme ziet er hierna uit als volgt:

Z	Q	V
\emptyset	d	a
$\{a\}$	e	b
$\{b\}$		c
$\{a,b\}$		
$\{a,c\}$		
$\{a,b,c\}$		

Het volgende element dat we inspecteren is d . Dit element heeft als ouders a en b , aan elke opwaarts gesloten deelverzameling waarin deze beide ouders voorkomen kunnen we d toevoegen om een nieuwe opwaarts gesloten deelverzameling te construeren. Van het kind f van d zijn nog niet alle ouders verwerkt. De toestand van het algoritme ziet er nu als volgt uit:

Z	Q	V
\emptyset	e	a
{a}		b
{b}		c
{a,b}		d
{a,c}		
{a,b,c}		
{a,b,d}		
{a,b,c,d}		

We herhalen dezelfde stappen voor e , dit geeft:

Z	Q	V
\emptyset	f	a
{a}		b
{b}		c
{a,b}		d
{a,c}		e
{a,b,c}		
{a,b,d}		
{a,b,c,d}		
{a,c,e}		
{a,b,c,e}		
{a,b,c,d,e}		

Het laatste element in Q is f . De uiteindelijke toestand van het algoritme na het verwerken van f is:

Z	Q	V
\emptyset		a
{a}		b
{b}		c
{a,b}		d
{a,c}		e
{a,b,c}		f
{a,b,d}		
{a,b,c,d}		
{a,c,e}		
{a,b,c,e}		
{a,b,c,d,e}		
{a,b,c,d,e,f}		

Het algoritme heeft nu alle elementen uit P verwerkt en heeft zo alle 12 opwaarts gesloten deelverzameling gevonden.

In dit voorbeeld houden we ter verduidelijking van het algoritme expliciet de verzameling Q bij met alle elementen waarvan alle ouders al verwerkt zijn. Eerder is de topologische ordening geïntroduceerd, een eigenschap van de topologische ordening is dat een element pas na zijn ouders voorkomt. Als we elementen volgens de topologische ordening verwerken hoeven we deze verzameling dus niet expliciet bij te houden.

BEWIJS

Om te bewijzen dat het algoritme werkt hebben we eerst enkele lemma's nodig. Hiervoor definiëren we eerst:

$$P_i = \{P[0] \dots P[i-1]\}$$

Hierbij is P weer een partieel geordende verzameling.

LEMMA'S

Lemma 1 gegeven een partieel geordende verzameling P , een opwaarts gesloten deelverzameling G van P en element $x \in P$, dan geldt: $\{x\} \cup G$ is opwaarts gesloten dan en slechts dan als $\text{parents}(x) \subseteq G$ is.

Bewijs Dit lemma volgt uit de definitie van opwaarts geslotenheid, als alle ouders van x zich in een opwaarts gesloten deelverzameling bevinden, dan bevinden alle elementen die groter zijn dan x zich ook in de opwaarts gesloten deelverzameling. \square

Lemma 2: gegeven een partieel geordende verzameling P dan geldt

$$\forall i \in \mathbb{N} (\text{parents}(P[i]) \subseteq P_i).$$

Bewijs Dit lemma volgt uit de definitie van topologische ordening. In P_i komen alle elementen uit P tot, exclusief $P[i]$, zelf voor. De ouders van $P[i]$ zijn per definitie groter dan $P[i]$ en in een topologische ordening komen deze eerder voor dus zitten zij in P_i . \square

Lemma 3: gegeven een partieel geordende verzameling P dan is het zo dat

$$\text{uppersets}(P_{i+1}) - \text{uppersets}(P_i) = \{ \{P[i]\} \cup G \mid G \in \text{uppersets}(P_i), \text{parents}(P[i]) \subseteq G \}$$

Bewijs Volgens lemma 2 geldt door de topologische ordening:

$$\text{parents}(P[i]) \subseteq P_i$$

We gebruiken nu lemma 1 om de opwaarts gesloten deelverzamelingen in $\text{uppersets}(P_{i+1})$ te bepalen:

$$\text{uppersets}(P_{i+1})$$

$$=$$

$$\text{uppersets}(P_i \cup \{P[i]\})$$

$$=$$

$$\text{uppersets}(P_i) \cup \{ \{P[i]\} \cup G \mid G \in \text{uppersets}(P_i), \text{parents}(P[i]) \subseteq G \}$$

We weten $P[i] \notin P_i$ dus geldt:

$$\text{uppersets}(P_i) \cap \{ \{P[i]\} \cup G \mid G \in \text{uppersets}(P_i), \text{parents}(P[i]) \subseteq G \} = \emptyset$$

Daarom geldt:

$$\text{uppersets}(P_{i+1}) - \text{uppersets}(P_i) = \{ \{P[i]\} \cup G \mid G \in \text{uppersets}(P_i), \text{parents}(P[i]) \subseteq G \}$$

\square

HET HOOFDBEWIJS

We gebruiken deze lemma's samen met Hoare logica [3] [4] om de correctheid van het familiealgoritme te bewijzen.

Voor dit bewijs gebruiken we de volgende verzamelingen:

- P : de partieel geordende verzameling
- Z : de verzameling van alle opwaarts gesloten deelverzamelingen $G \subseteq P_i$ met $i \in \mathbb{N}$

Het algoritme moet voldoen aan de volgende pre- en postcondities:

$$\{Pre: P \text{ is een partieel geordende verzameling}\}$$

$$\{Post: Z = \text{uppersets}(P)\}$$

We kiezen de invariant J en guard B als volgt:

$$J: Z = \text{uppersets}(P_i) \wedge 0 \leq i \leq \#P$$

$$B: i < \#P$$

Als $i = \#P$ geldt de guard niet meer maar geldt wel

$$J \wedge \neg B: Z = \text{uppersets}(P_{\#P})$$

Door de definitie van P_i weten we dat $P_{\#P} = P$ dus kunnen we $J \wedge \neg B$ vereenvoudigen tot we onze postconditie herkennen:

$$Z = \text{uppersets}(P)$$

We initialiseren Z met $Z = \{\emptyset\}$ en i met $i = 0$. Uit deze initialisatie volgt dat de invariant J waar is:

$$\{Pre: P \text{ is een partieel geordende verzameling}\}$$

$$i \leftarrow 0$$

$$\{i = 0\}$$

$$Z \leftarrow \{\emptyset\}$$

$$\{i = 0, Z = \{\emptyset\}\}$$

(* Volgens de definitie van P_i geldt $P_0 = \emptyset$, de enige opwaarts gesloten deelverzameling in P_0 is dan $\{\emptyset\}$ *)

$$\{i = 0, Z = \text{uppersets}(P_i)\}$$

$$\{Z = \text{uppersets}(P_i)\}$$

(* Herken definitie invariant J *)

$$\{\}$$

Met de gekozen invariant willen we met i van 0 naar $\#P$ lopen om zo telkens P_i te laten groeien totdat $P_i = P$. We vermoeden dat de variabele i elke iteratie zal veranderen, daarom kiezen we de volgende variante functie:

$$vf = \#P - i$$

De body van de loop is nu als volgt:

$\{J \wedge B \wedge vf = V\}$

(definitie J, B en vf *)*

$\{J: Z = \text{uppersets}(P_i) \wedge 0 \leq i \leq \#P \wedge i < \#P \wedge \#P - i = V\}$

(herschrijven *)*

$\{J: Z = \text{uppersets}(P_i) \wedge 0 \leq i < \#P \wedge \#P - i = V\}$

$H \leftarrow \emptyset$

$\{J: Z = \text{uppersets}(P_i) \wedge 0 \leq i < \#P \wedge \#P - i = V \wedge H = \emptyset\}$

foreach G **in** Z **do**

if $\text{parents}(P[i]) \subseteq G$ **then**

$H \leftarrow H \cup \{P[i]\} \cup G$

(Volgens lemma 3 geldt nu $H = \text{uppersets}(P_i \cup \{P[i]\}) - \text{uppersets}(P_i)$ *)*

$\{J: Z = \text{uppersets}(P_i) \wedge 0 \leq i < \#P \wedge \#P - i = V \wedge$

$H = \text{uppersets}(P_i \cup \{P[i]\}) - \text{uppersets}(P_i)\}$

$Z \leftarrow Z \cup H$

$\{J: Z = \text{uppersets}(P_i \cup \{P[i]\}) \wedge 0 \leq i < \#P \wedge \#P - i = V\}$

(Voorbereiden ophogen i *)*

$\{J: Z = \text{uppersets}(P_{i+1}) \wedge 0 \leq (i+1) \leq \#P \wedge \#P - (i+1) < V\}$

$i = i + 1$

$\{J: Z = \text{uppersets}(P_i) \wedge 0 \leq i \leq \#P \wedge \#P - i < V\}$

Als het algoritme stopt hebben we alle opwaarts gesloten deelverzamelingen in P_i gevonden en is P_i gelijk aan P . Het algoritme vindt dus correct alle opwaarts gesloten deelverzamelingen in P .

Zo komen we op het eerder gepresenteerde familiealgoritme.

De naïeve aanpak heeft een tijdscomplexiteit van tenminste $O(2^n)$, waar n het aantal elementen in de partieel geordende verzameling is. Om te bewijzen dat het familiealgoritme in veel gevallen efficiënter is moeten we de tijdscomplexiteit hiervan beredeneren.

Initialisatie

De eerste stap is het maken we een adjacency list representatie [2] voor de partieel geordende verzameling. We maken een lijst van elementen uit de partieel geordende verzameling en voorzien elk van deze elementen met een lijst met hun directe ouders, deze stap heeft een tijdscomplexiteit van $O(n^2)$.

Het familiealgoritme vereist dat we een topologische ordening maken en dat we alle elementen volgens deze topologische ordening sorteren. Bij het maken van een topologische ordening lopen we in de buitenste lus over alle n elementen in de partieel geordende verzameling P , dit zit in $O(n)$. In de binnenste lus lopen we voor elk van deze elementen over al hun directe kinderen. In een verzameling waar elk element een relatie heeft met elk ander element is dit ook $O(n)$. In een partieel geordende verzameling is dit door de antisymmetrisch eigenschap niet mogelijk maar we kunnen wel concluderen dat het maken van de topologische ordening ten hoogste in $O(n^2)$ zit.

De initialisatie heeft dus een tijdscomplexiteit van $O(n^2)$.

In de body van het familiealgoritme wordt elk element x uit de partieel geordende verzameling P eenmaal verwerkt. De buitenste lus wordt dus n keer uitgevoerd.

In deze lus vinden de volgende berekeningen plaats:

Zoeken naar uitbreidbare opwaarts gesloten deelverzamelingen

Hierna controleren we voor elke al gevonden opwaarts gesloten deelverzameling of alle ouders van x in deze deelverzameling zitten.

Het aantal ouders van element $x \in P$ kan niet groter zijn dan n . Immers, er kunnen niet meer dan alle elementen in P een ouder zijn van x . Het itereren over alle ouders zit dus in $O(n)$.

Het aantal reeds gevonden opwaarts gesloten deelverzamelingen noemen we b . Dit kan niet groter zijn dan het totaal aantal opwaarts gesloten deelverzamelingen in P . Uitzoeken of een element in een opwaarts gesloten deelverzameling zit is $O(n \log n)$. Deze test moeten we maximaal b keer uitvoeren in elke iteratie. Het zoeken naar gevonden opwaarts gesloten deelverzamelingen die alle ouders van x bevatten zit dus in $O((n \log n) * b)$.

Het toevoegen van nieuwe opwaarts gesloten deelverzamelingen

Als laatste voegen we alle nieuw gevonden opwaarts gesloten deelverzamelingen toe aan de lijst met gevonden opwaarts gesloten deelverzamelingen. We kunnen ten hoogste alle gevonden opwaarts gesloten deelverzamelingen uitbreiden met x . Het maken van een nieuwe opwaarts gesloten deelverzameling vereist het kopiëren van de data uit de oude opwaarts gesloten deelverzameling en het toevoegen van x . Dit is dus $O(n)$. In totaal kunnen we maximaal b nieuwe opwaarts gesloten deelverzamelingen maken dus deze actie is $O(n * b)$ en wordt in het algoritme n keer uitgevoerd.

Alles samen komen we tot een tijdscomplexiteit van $O(n^2)$ *initialisatie* + $O((n \log n) * n * b)$ *zoeken* + $O(n^2 * b)$ *toevoegen*. Dit kunnen we versimpelen tot $O(n^2 \log n * b)$.

Nu moeten we nog de term b wegwerken omdat anders alleen de tijdscomplexiteit van het familiealgoritme kunnen bepalen nadat we bepaald hebben hoeveel opwaarts gesloten deelverzamelingen er in P zitten.

In het slechtste geval zitten in een partieel geordende verzameling P 2^n opwaarts gesloten deelverzamelingen, bijvoorbeeld als er alleen maar maximale elementen in P zitten.

In het beste geval zitten er n partieel geordende verzamelingen in P . Dit gebeurt als P een totaal geordende verzameling is.

Het familiealgoritme heeft dus de volgende eigenschappen:

- Slechtste geval: $O(2^n)$
- Beste geval: $O(n^3 \log n)$

Het naïeve algoritme heeft in elk geval een tijdscomplexiteit van $O(2^n)$ waar het familiealgoritme, afhankelijk van de partieel geordende verzameling, een tijdscomplexiteit van $O(n^3)$ tot $O(2^n)$ heeft.

Qua geheugen gebruik is er een groot verschil tussen het familiealgoritme en het naïeve algoritme. Het naïeve algoritme hoeft geen lijst van opwaarts gesloten deelverzamelingen bij te houden om het aantal opwaarts gesloten deelverzamelingen te bepalen en heeft dus een geheugencomplexiteit van $O(n)$. Het familiealgoritme slaat alle opwaarts gesloten deelverzamelingen op en heeft dus ten hoogste een geheugen complexiteit van $O(2^n)$ en in het beste geval slechts een geheugen complexiteit van $O(n^2)$.

DE UPTRIE DATASTRUCTUUR

Het zoeken en opslaan van alle gevonden opwaarts gesloten deelverzamelingen heeft grote invloed op de tijds- en geheugencomplexiteit van het familiealgoritme. We proberen het algoritme sneller te maken en minder geheugen te laten gebruiken door gebruik te maken van een andere datastructuur voor het opslaan van de al gevonden opwaarts gesloten deelverzamelingen.

DE TRIE EN UPTRIE

De naam trie komt van het Engelse woord *retrieval* wat opzoeken betekent. Een trie heeft de volgende eigenschappen [5]. *We gebruiken hier enkele symbolen uit de taaltheorie* [6].

Laat S een set strings zijn uit alfabet Σ zodat er geen string $s \in S$ een prefix is voor een andere string. Een standaard trie voor S is een geordende boom T met de volgende eigenschappen:

- Elke knoop van T , behalve de wortel, is gelabeld met een karakter uit Σ
- De ordening van de kinderen van een interne knoop van T volgt uit de ordening van alfabet Σ
- Elk blad van T wordt geassocieerd met een string $s \in S$, s wordt geconstrueerd uit de samenvoeging van de labels van de knoop op het pad van de wortel naar dat blad.

We gebruiken de eigenschappen van partieel geordende verzamelingen en de topologische ordening om de trie naar onze wensen aan te passen we noemen deze aangepaste trie een uptrie. In een uptrie T van een partieel geordende verzameling P die geordend is volgens een topologische ordening geldt:

- P is het alfabet van de uptrie
- Elke knoop van T , behalve de wortel, is gelabeld met een element uit P .
- De strings $w \in T$ zijn de opwaarts gesloten deelverzamelingen in P . Deze zijn geordend volgens de topologische ordening van P . De ordening van de interne knopen van T volgt uit de ordening van het label van P .
- De strings die we kunnen aantreffen in de uptrie zijn deze in de taal $L_0 = \{w \in P^* \mid w \in \text{uppersets}(P) \text{ en } w \text{ topologische geordend}\}$

De samenvoeging van de labels op elke pad in de uptrie van wortel naar blad is een opwaarts gesloten deelverzameling en in dit pad zijn alle elementen geordend volgens de topologische ordening. Voor een pad R van wortel naar blad geldt dus:

$$\forall i \in \mathbb{N}(0 < i < \#R \rightarrow R[i] < R[i - 1])$$

Gegeven een verzameling V met $V = \{R[0] \dots R[i]\} \wedge 0 \leq i < \#R$ dan geldt:

$$\forall x \in V(\text{parents}(x) \subseteq V)$$

Dus in een uptrie is niet alleen elk pad van wortel naar blad een opwaarts gesloten deelverzameling, maar ook elk pad van wortel naar een interne knoop is een opwaarts gesloten deelverzameling. We hoeven dus niet voor elke opwaarts gesloten deelverzameling een apart blad element toe te voegen, er geldt dus dat er voor elke opwaarts gesloten deelverzameling in P precies één element x in de uptrie zit waarvan het pad van de wortel naar x deze opwaarts gesloten deelverzameling representeert.

Voor de uptrie definiëren we ook nog de volgende functies:

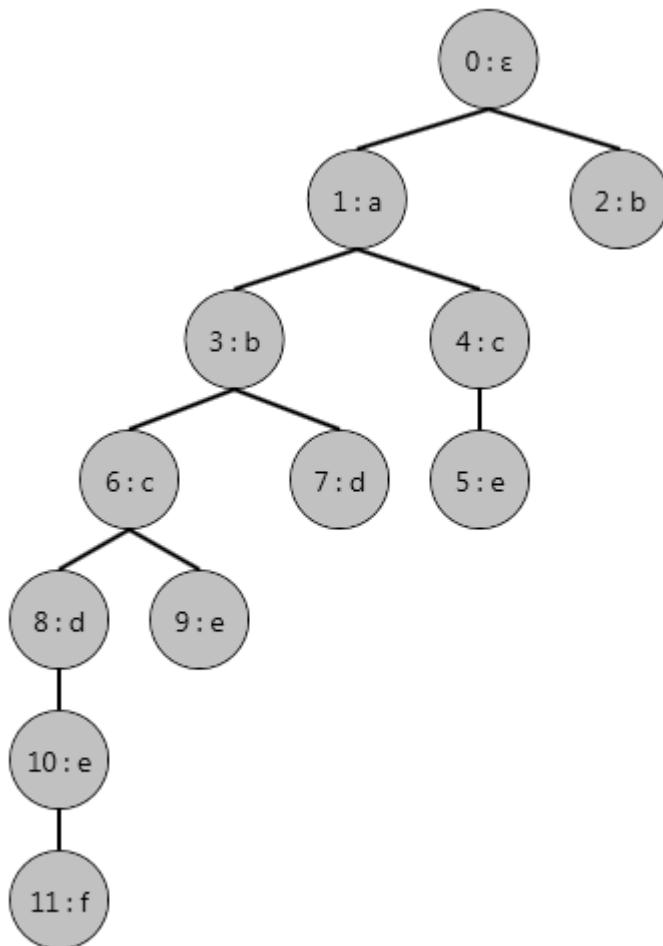
$label(x)$: gegeven een element $x \in T$ retourneert deze functie element $y \in P$ waarmee x gelabeld is.

$labels(S)$: gegeven een deelverzameling $S \subseteq T$ retourneert deze functie de verzameling van labels van de elementen in S .

$path(x, y)$: gegeven $x, y \in T$ retourneert deze functie de verzameling van elementen in T op het pad van x naar y inclusief x en y zelf.

$children(x)$: $\{y \mid y <_1 x\}$, de directe kinderen van $x \in T$

In een uptrie kunnen we veel compacter alle opwaarts gesloten deelverzamelingen uit P opslaan. Het opslaan in lijsten heeft een geheugencomplexiteit van $O(n^2)$ in het beste geval tot $O(2^n)$ in het slechtste geval. Het opslaan van alle opwaarts gesloten deelverzamelingen in de uptrie heeft een geheugencomplexiteit van $O(n)$ in het beste geval tot $O(2^n)$ in het slechtste geval.



Figuur C: een visualisatie van de uptrie van de opwaarts gesloten deelverzamelingen in de partieel geordende verzameling uit Figuur B. In elke node is gelabeld met één element uit P . Om makkelijk te kunnen verwijzen zijn alle nodes genummerd. De opwaarts gesloten deelverzameling $\{a, b, c\}$ wordt gerepresenteerd door het pad van node 0 naar 6.

Een van de belangrijkste stappen in het familiealgoritme is, gegeven $x \in P$, het vinden van opwaarts gesloten deelverzamelingen G met $parents(x) \subseteq G$. We weten dat in de uptrie een opwaarts gesloten deelverzameling gerepresenteerd wordt door de labels op het pad van de wortel naar een node x in de uptrie. Gegeven deelverzameling $S \subseteq P$, uptrie T en wortel van de uptrie z kunnen we een zoekfunctie construeren die de volgende verzameling retourneert:

$$H = \{x \mid labels(path(z, x)) \supseteq S\}$$

We hebben nu een verzameling $H \subseteq T$, voor elk element $x \in H$ wordt de opwaarts gesloten deelverzameling G die ten minste alle elementen in S bevat gegeven door $labels(path(z, x))$.

We implementeren deze zoekfunctie als een functie waarbij we recursief de uptrie aflopen. We gebruiken de deelverzameling S' om de elementen in S aan te geven welke we nog niet zijn tegen gekomen tijdens het aflopen van de uptrie. Tijdens het aflopen van de boom zijn er bij elke knoop $x \in T$ die we tegenkomen vier situaties mogelijk, we maken een beslissing op basis van element $y = label(x)$.

1. Als $S' \setminus \{y\} = \emptyset$ dan is het pad van wortel naar y een opwaarts gesloten deelverzameling G met $S \subseteq G$. Alle paden vanaf y representeren opwaarts gesloten deelverzamelingen G' met $S \subseteq G \subseteq G'$ dus we zoeken ook nog verder naar \emptyset in alle paden vanaf y .
2. Als y gelijk is aan het, volgens de topologische ordening, grootste element in S' dan zoeken we verder naar $S' \setminus \{y\}$ in de paden vanaf y .
3. Als $\exists z \in S' (y < z)$ dan zullen we volgens de topologische ordening op dit pad nooit z vinden en stoppen we.
4. Als $\forall z \in S' (z < y)$ dan zoeken we verder op dit pad omdat we nog alle elementen in S' kunnen tegen komen.

We gebruiken een paar keer de eigenschap dat de elementen in P geordend zijn volgens een topologische ordening en dat we twee elementen kunnen vergelijken op basis van hun ordening. De wortel van de uptrie bevat een speciaal element ε dat niet in P voorkomt. Toch willen we ook dit element vergelijken volgens de topologische ordening. Daarom spreken we af dat ε , volgens de topologische ordening, groter is dan alle elementen in P .

Deze aanpak vertaalt zich naar de volgende pseudo code

Algorithm FindUpperSetsWithElements(z, S')

Input a node z from the uptrie and a set of elements S'
Output the set H such that $H = \{x \mid \text{labels}(\text{path}(z, x)) \supseteq S'\}$

```

 $H \leftarrow \emptyset$ 
 $y \leftarrow \text{label}(z)$ 

if the largest element, by topological order, in  $S' = y$  then
     $S' \leftarrow S' \setminus \{y\}$ 

if  $\exists x \in S' (y < x)$  then
    return  $\emptyset$ 

if  $S' = \emptyset$  then
     $H \leftarrow H \cup \{y\}$ 

foreach child in children( $y$ ) do
     $H \leftarrow H \cup \text{FindUpperSetsWithElements}(\text{child}, S)$ 

return  $H$ 

```

Om dit algoritme te illustreren gebruiken we als voorbeeld de uptrie uit figuur C, en zoeken we naar alle opwaarts gesloten deelverzamelingen die a en b bevatten:

FindUpperSetsWithElements($\text{node } 0, \{a, b\}$)

Omdat de functie recursief is houden we een stack bij. Zo kunnen we zien welke aanroepen van de zoek functie, met welke argumenten, nog uitgevoerd moeten worden. Ook zetten we achter elk gevonden element, die we opslaan in H , de bijbehorende opwaarts gesloten deelverzameling.

Stack	H
$y = \text{node } 0, S' = \{a, b\}$	

In de eerste aanroep inspecteert het algoritme *node* 0, de wortel van de uptrie. Deze node bevat het element ε en representeert de opwaarts gesloten deelverzameling $\{\varepsilon\}$. We hebben afgesproken dat ε , volgens de topologische ordening, groter is dan alle elementen in P dus we gebruiken de vierde regel. We roepen de zoek functie aan op de twee direct kinderen van *node* 0, samen met de ongewijzigde verzameling S' .

Stack	H
$y = \text{node } 1, S' = \{a, b\}$	
$y = \text{node } 2, S' = \{a, b\}$	

In de volgende aanroep inspecteren we *node* 1, deze node bevat element a , welke gelijk is aan het, volgens de topologische ordening grootste element in S' . We gebruiken regel 2 en verwijderen a uit de verzameling S' en roepen de zoekfunctie weer aan op *node* 3 en *node* 4, de kinderen van *node* 1.

Stack	H
$y = \text{node } 3, S' = \{b\}$	
$y = \text{node } 4, S' = \{b\}$	
$y = \text{node } 2, S' = \{a, b\}$	

Nu inspecteren we *node* 3, deze node bevat element b , welke weer gelijk is aan het, nu enige, element in S' . We gebruik de eerste regel en voegen *node* 3, welke de opwaarts gesloten deelverzameling $\{a, b\}$ representeert, toe aan H . In de paden onder *node* 3 zoeken we verder naar $S' = \emptyset$. Het algoritme gebruikt hier telkens de eerste regel en voegt in de volgende iteraties dan ook nodes 6, 7, 8, 9, 10 en 11 toe. De toestand van het programma hierna is als volgt:

Stack	H
$y = \text{node } 4, S' = \{b\}$	<i>node</i> 3 : $\{a, b\}$
$y = \text{node } 2, S' = \{a, b\}$	<i>node</i> 6: $\{a, b, c\}$
	<i>node</i> 7: $\{a, b, d\}$
	<i>node</i> 8: $\{a, b, c, d\}$
	<i>node</i> 9: $\{a, b, c, e\}$
	<i>node</i> 10: $\{a, b, c, d, e\}$
	<i>node</i> 11: $\{a, b, c, d, e, f\}$

We gaan nu verder zoeken in een andere tak. We inspecteren *node* 4. Deze bevat het element c , welke volgens de topologische ordening kleiner is dan het element b uit S' . We gebruiken de derde regel en stoppen met verder zoeken in deze tak.

Stack	H
$y = \text{node } 2, S' = \{a, b\}$	$\{a, b\}$
	$\{a, b, d\}$
	$\{a, b, c\}$
	$\{a, b, c, d\}$
	$\{a, b, c, e\}$
	$\{a, b, c, d, e\}$
	$\{a, b, c, d, e, f\}$

Als laatste zoeken we in *node* 2 welke element b bevat. Ook hier geldt dat er een element in S' zit dat, volgens de topologische ordening, groter is dan het element in de huidige node. Dus we gebruiken weer de derde regel en we stoppen.

We hebben nu alle nodes in de uptrie gevonden die opwaarts gesloten deelverzamelingen representeren die de elementen a en b bevatten. Om van deze nodes de echte opwaarts gesloten deelverzamelingen te verkrijgen moet vanaf het element een pad gevolgd worden naar de wortel, de labels op dit pad zijn de elementen uit de opwaarts gesloten deelverzameling.

OPWAARTS GESLOTEN DEELVERZAMELINGEN GENEREREN MET DE UPTRIE

Laten we nogmaals het familiealgoritme beschouwen. Het familiealgoritme voert de volgende stappen uit voor elk element x in de partieel geordende verzameling P .

- Zoek alle opwaarts gesloten deelverzamelingen G die $\text{parents}(x)$ bevatten
- Voeg de nieuwe opwaarts gesloten deelverzameling toe te maken uit $G \cup \{x\}$.

Voor de eerste stap kunnen we de functie `FindUpperSetsWithElements` gebruiken. Voor de tweede stap zouden we een functie `Insert` kunnen definiëren die, net als bij een normale trie, zoekt naar een zo lang mogelijke prefix van P die zich al in de uptrie bevindt om daarachter

het laatste stuk van P toe te voegen aan de uptrie maar door de eigenschap dat zowel P als de opwaarts gesloten deelverzamelingen in de uptrie geordend zijn volgens dezelfde topologische ordening kunnen we de volgende eigenschappen gebruiken om nog makkelijker nieuwe opwaarts gesloten deelverzamelingen toe te voegen:

In elke iteratie i van het familiealgoritme is $P[i]$ het, volgens de topologische ordening, tot nu toe kleinste element dat we verwerken.

Het, volgens de topologische ordening, kleinste element in een opwaarts gesloten deelverzameling is in de uptrie altijd het laatste element op het pad dat die uptrie representeert.

Dit geeft ons dat voor elke opwaarts gesloten deelverzameling die we willen toevoegen in iteratie i het element $P[i]$ het kleinste element in die opwaarts gesloten deelverzameling is.

De functie `FindUpperSetsWithElements` geeft ons de laatste node y op een pad dat een opwaarts gesloten deelverzameling representeert die $parents(P[i])$ bevat. Het uitbreiden van deze opwaarts gesloten deelverzameling met $P[i]$ kunnen we dan simpelweg doen door $P[i]$ aan de uptrie toe te voegen als een kind van node y . Dit levert ons de volgende pseudo code op:

Algorithm `GenerateUpperSets(P)`

Input partially ordered set P

Output an uptrie representing all uppersets in P .

`uptrieRoot` $\leftarrow \varepsilon$

for $i \leftarrow 0$ to $\#P-1$ **do**

$H \leftarrow \text{FindUpperSetsWithElements}(\text{uptrieRoot}, \text{parents}(P[i]))$

foreach e **in** H **do**

 create a new child node for e and label it with $P[i]$

return `uptrieRoot`;

SNELHEIDSVERBETERINGEN DOOR HET GEBRUIK VAN DE UPTRIE

Zoals eerder genoemd heeft de naïeve aanpak een tijdscomplexiteit van ten minste $O(2^n)$. Het familiealgoritme, zonder gebruik te maken van de uptrie, heeft een tijdscomplexiteit van $O(n^3 \log n)$ tot $O(2^n)$ en een geheugencomplexiteit van $O(n^2)$ tot $O(2^n)$.

Als we bij het familiealgoritme gebruik maken van de uptrie is de geheugencomplexiteit $O(n)$ tot $O(2^n)$, immers bevat de uptrie slechts één element voor elke opwaarts gesloten deelverzameling. Er is bij het beste geval dus een grote winst.

Ook de tijdscomplexiteit van het familiealgoritme verandert door het gebruik van de uptrie. Gegeven een partieel geordende verzameling P met n elementen berekenen we de tijdscomplexiteit als volgt:

Initialisatie

De initialisatie stap voor het familiealgoritme blijft gelijk, we maken een adjacency list voor de partieel geordende verzameling, daarna moet er een topologische ordening gemaakt worden en

de elementen in de partieel geordende verzameling moeten nog steeds gesorteerd worden volgens de topologische ordening. De initialisatie heeft dus een tijdscomplexiteit van $O(n^2)$.

Zoeken naar uitbreidbare opwaarts gesloten deelverzamelingen

Hierna wordt voor elk element $x \in P$ gezocht naar al gevonden opwaarts gesloten deelverzamelingen van P waar de ouders van x in zitten. De uptrie bevat 1 element voor elke opgeslagen opwaarts gesloten deelverzameling. We noemen het aantal opwaarts gesloten deelverzamelingen in P weer b . In het slechtste geval moet alle elementen van de uptrie doorzocht worden, deze stap heeft dus een tijdscomplexiteit van $O(b)$.

De opwaarts gesloten deelverzamelingen die gevonden zijn worden hierna uitgebreid met x en toegevoegd aan de uptrie. Dit doen we door in de uptrie element x toe te voegen als blad aan alle gevonden opwaarts gesloten deelverzamelingen, het toevoegen kunnen we doen in constante tijd en er zijn maximaal b gevonden opwaarts gesloten deelverzamelingen, deze stap heeft dus een tijdscomplexiteit van $O(b)$ en wordt n keer uitgevoerd in het algoritme.

Als we bij het familiealgoritme de uptrie gebruiken is de tijdscomplexiteit geven door $O(n^2)$ *initialisatie* + $O(n * b)$ *zoeken* + $O(n * b)$ *toevoegen*. Dit is gelijk aan een tijdscomplexiteit van $O(n * b)$. We hebben al bepaald dat b kunnen herschrijven in termen van n voor het slechtste tot 2^n en in het beste geval tot n .

Als het familiealgoritme gebruik maakt van de uptrie heeft het de volgende eigenschappen:

- Slechtste geval $O(2^n)$
- Beste geval $O(n^2)$

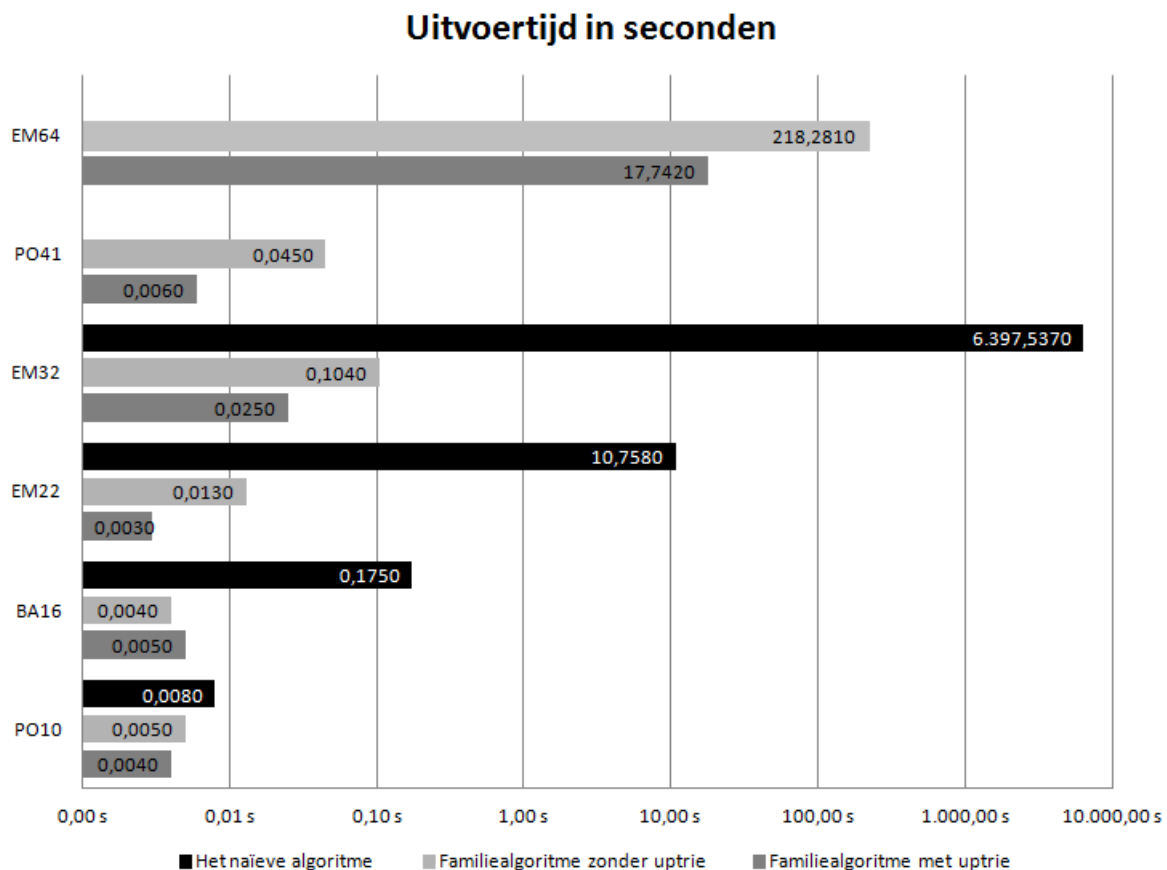
Dat de verbeteringen in de geheugencomplexiteit en tijdscomplexiteit ook in de praktijk uitmaken zullen we aantonen in het volgende hoofdstuk.

VERGELIJKENDE TEST

Naast de tijds- en geheugencomplexiteit zijn we ook geïnteresseerd in de prestaties van de algoritmen als zij toegepast worden op echte partieel geordende verzamelingen. Daarom hebben wij het naïeve algoritme, het familiealgoritme en het familiealgoritme met uptrie geïmplementeerd in C# 4.0 en deze laten werken aan zes verschillende datasets. Deze datasets zijn te vinden in de bijlage datasets en de code voor de algoritmen kunt u vinden de sectie broncode, listings 1, 2 en 3.

Elk algoritme heeft elke dataset vijf keer doorgerekend, van deze vijf resultaten werd het minimum genomen als eind score voor die dataset. Elk algoritme kreeg 2 uur de tijd om een dataset door te rekenen. De tests zijn uitgevoerd op een Intel® Core™ i5-2500K Processor met 8GB werkgeheugen, de C# byte code werd uitgevoerd door de Microsoft.NET 4.0 64bits CLR.

De resultaten kunt u zien in de figuur D. De grafiek gebruikt een logaritmische schaal om de uiteenlopende resultaten in één figuur te kunnen tonen. Van het naïeve algoritme zijn geen resultaten voor testsets PO41 en EM64 omdat dit algoritme hiervoor teveel tijd nodig had.



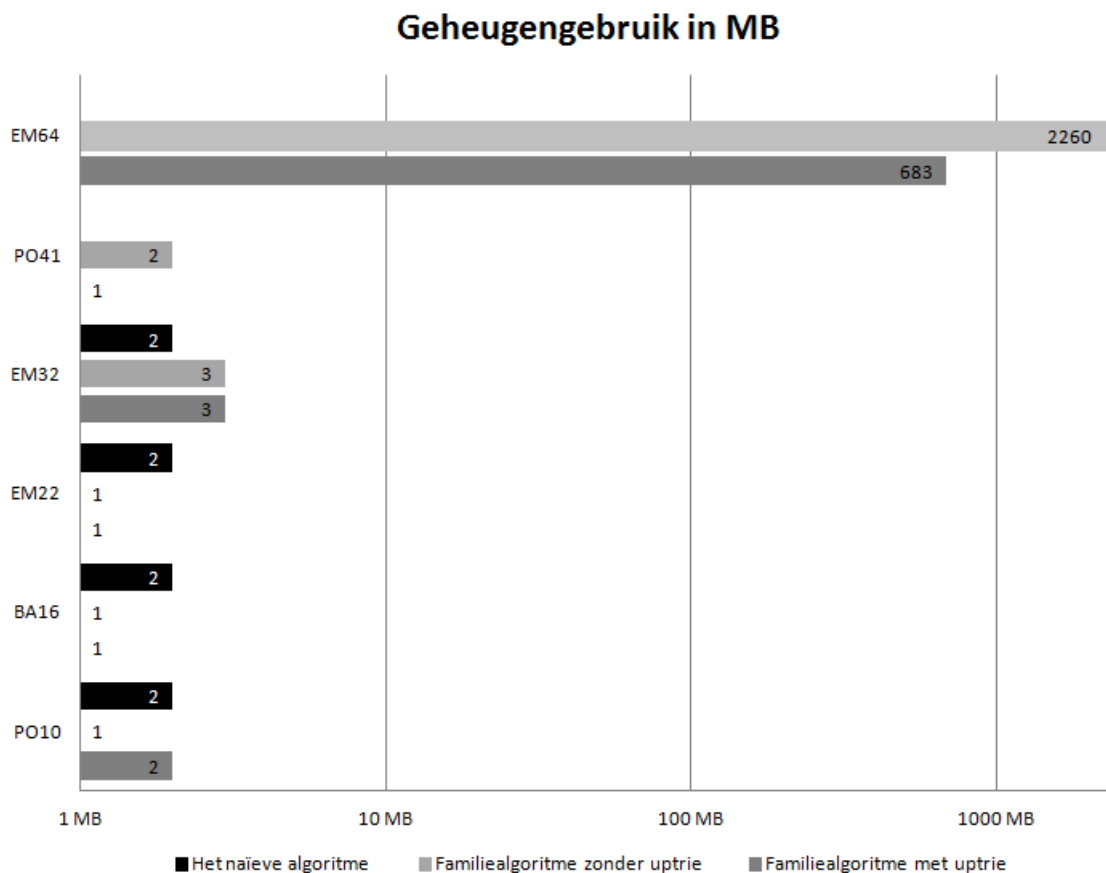
Figuur D, uitvoersnelheden van verschillende algoritmen

In Figuur D is duidelijk te zien dat bij grotere datasets het familiealgoritme met uptrie het snelste is, bij de dataset EM64 is het familiealgoritme met uptrie zelfs meer dan 10x zo snel als het familiealgoritme zonder uptrie.

Dataset EM32 is de grootste dataset die het naïeve algoritme binnen twee uur wist te verwerken, dit kostte meer dan 100 minuten. We zien hier duidelijk de voordelen van beide implementaties van het familiealgoritme, beide konden deze dataset in enkele tienden van seconde uitrekenen.

In de kleinere datasets wordt het verschil tussen de beide implementaties van het familiealgoritme minder groot. In de dataset BA16 is de implementatie van het familiealgoritme zonder uptrie zelfs iets sneller dan de implementatie met uptrie. Dit kan meerdere oorzaken hebben. Zo is een nadeel van de grote O-notatie dat constante factoren wegvallen, het zou kunnen dat bij kleine datasets de hogere constante factor van het familiealgoritme met uptrie niet opweegt tegen de gemiddeld lagere tijdscomplexiteit van dit algoritme. Een andere mogelijkheid heeft te maken met feit dat de versie zonder uptrie lijsten gebruikt en de versie met uptrie een boomstructuur. Een boomstructuur heeft een slechtere datalokaliteit dan een lijst waardoor het mogelijk is dat het familiealgoritme zonder uptrie sneller werkt zolang het niet veel meer operaties hoeft uit te voeren dan het familiealgoritme met uptrie [7].

Het familiealgoritme met uptrie is duidelijk het snelste algoritme.



Figuur E, geheugengebruik van verschillende algoritmen

Qua geheugengebruik scoort het familiealgoritme met uptrie beter dan de implementatie zonder uptrie, zoals te zien is in figuur E.

KAN HET NOG SNELLER?

Hoewel het familiealgoritme in combinatie met de uptrie in het beste geval een tijdscomplexiteit heeft van $O(n^2)$ heeft het in het slechtste geval nog steeds een tijdscomplexiteit van $O(2^n)$, het zelfde als het naïve algoritme. Een algoritme zoals het familiealgoritme dat alle opwaarts gesloten deelverzamelingen in P moet genereren om ze te kunnen tellen en zal altijd een worst case hebben van ten minste $O(2^n)$ omdat een partieel geordende verzameling 2^n opwaarts gesloten deelverzamelingen kan bevatten. Maar zelfs algoritmen die niet alle opwaarts gesloten deelverzamelingen hoeven te generen om te ze te kunnen tellen komen waarschijnlijk niet van deze worst case van $O(2^n)$ af. J. Scott Provan en Michael O. Ball hebben in een publicatie in het SIAM Journal on Computing bewezen dat het tellen van alle opwaarts gesloten deelverzamelingen in een partieel geordende verzameling zich in de complexiteitsklasse $\#P$ -volledig bevindt [8]. Problemen in deze complexiteitsklasse houden zich bezig met het bepalen van het aantal oplossingen voor beslissingsproblemen in de complexiteitsklasse NP -volledig. Deze complexiteitsklasse bestaat uit beslissingsproblemen die in polynomiale tijd geverifieerd kunnen worden op een deterministische Turingmachine en, anders gezegd, op te lossen zijn in polynomiale tijd op een niet-deterministische Turingmachine.

Logischerwijs is een probleem in $\#P$ -volledig ten minste net zo moeilijk op te lossen als een probleem in NP -volledig, als we makkelijk het aantal oplossingen zouden kunnen tellen zouden we immers ook makkelijk kunnen vertellen of er oplossingen zijn. Er wordt geloofd dat de problemen in de complexiteitsklasse NP -volledig, en dus ook $\#P$ -volledig, niet op te lossen zijn in polynomiale tijd op een deterministische Turingmachine [9] en daarom denken wij dat er geen algoritme bestaat dat in het slechtste geval een betere tijdscomplexiteit heeft dan het familiealgoritme.

GERELATEERD WERK

Het probleem van Dedekind is een bekend probleem waarbij het bepalen van het aantal opwaarts gesloten deelverzamelingen van belang is. Hierbij wordt er gezocht naar het aantal zwakstijgende functies met als invoer n Boolse variabelen (*variabelen die alleen de waarde 0 of 1 kunnen hebben*) die als uitvoer een Boolse variabele produceert. Het aantal van deze functies bij n invoer variabelen wordt ook wel het n -de Dedekind nummer genoemd [10].

Dit probleem kan ook opgevat worden als het bepalen van het aantal opwaarts gesloten deelverzamelingen in een vrije gedistribueerde tralie met n generatoren. Datasets EM32 en EM64 zijn deze tralies voor het 5^e en 6^e Dedekind nummer.

Onder andere Daniel Kleitman heeft onderzoek gedaan naar een bovengrens voor het aantal opwaarts gesloten deelverzamelingen in zo'n tralie [11] en Doug Wiedemann heeft in 1991 het 8^e Dedekind nummer bepaald [12].

In 2011 hebben Yong Chan Kim en Young Sun Kim een algemene studie gedaan naar de eigenschappen van opwaarts gesloten deelverzamelingen en partieel geordende verzamelingen [13].

En Giorgio Delzanno & Jean-Francois Raskin stellen in een rapport een symbolische representatie van verzamelingen van opwaarts gesloten deelverzamelingen voor die enigszins lijkt op de door ons voorgestelde uptrie [14].

CONCLUSIE EN VERDER ONDERZOEK

Het voorgestelde familiealgoritme, in combinatie met de uptrie datastructuur, laat een grote snelheidswinst zien t.o.v. het naïeve algoritme. Toch is alleen het familiealgoritme niet genoeg om onderzoekers te helpen bij het bepalen van de opwaarts gesloten deelverzamelingen van grote partieel geordende verzamelingen. Onderzoekers zullen het familiealgoritme wel kunnen gebruiken op kleinere partieel geordende verzamelingen. Ook kunnen onderzoekers het familiealgoritme toepassen op delen van grotere partieel geordende verzamelingen en zo door slim gebruik te maken van symmetrie toch het aantal opwaarts gesloten deelverzamelingen snel bepalen.

Verder onderzoek zou kunnen zoeken naar een algoritme om partieel geordende verzamelingen op te delen in symmetrische eenheden. Via het familiealgoritme zou dan het aantal opwaarts gesloten deelverzamelingen in deze eenheden bepaald kunnen worden waarna uit deze deelresultaten het totaal aantal opwaarts gesloten deelverzamelingen gedistilleerd zou kunnen worden.

DANKWOORD

Ik wil graag prof. dr. G.R. Renardel de Lavalette, bedanken voor het begeleiden van deze scriptie en het voordragen van dit onderwerp. Dankzij hem heb ik in dit onderzoek een compleet nieuwe kant van het vakgebied informatica ontdekt en zijn enthousiasme en heldere uitleg zijn de kwaliteit van deze scriptie zeker ten goede gekomen.

Ook wil ik dr. A Meijster bedanken voor zijn werk als tweede lezer.

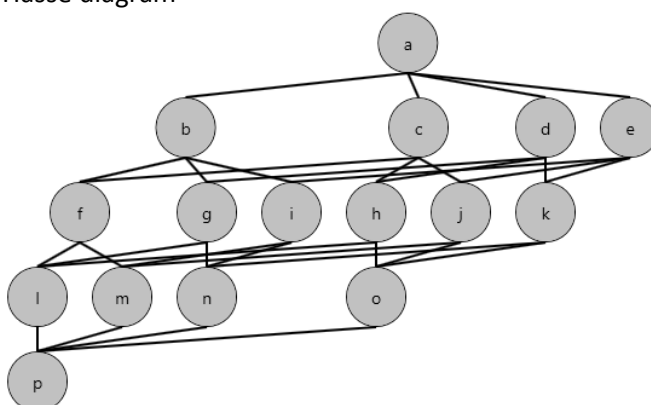
BIBLIOGRAFIE

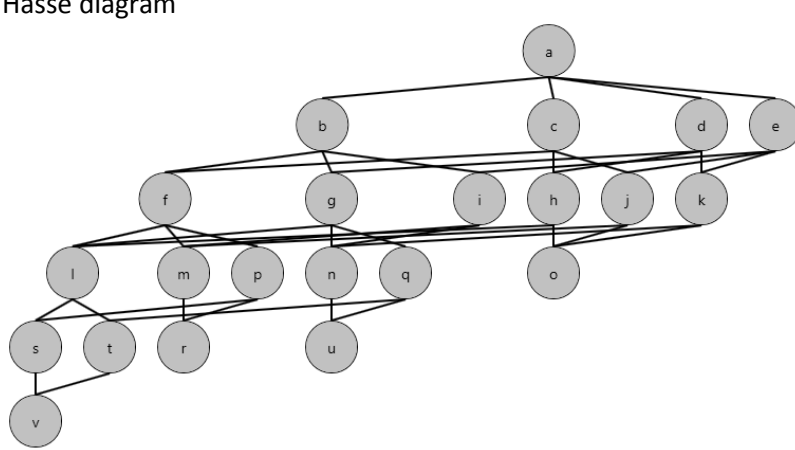
- [1] Giuseppe Di Battista and Roberto Tamassia, "Algorithms for plane representations of acyclic digraphs," *Theoretical Computer Science volume 61*, pp. 175-198, 1988.
- [2] Michael Goodrich and Roberto Tamassia, *Algorithm Design: Foundations, Analysis and Internet Examples*, 1st ed.: John Wiley & Sons, 2002.
- [3] Charles Anthony Richard Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576-583, Oktober 1969.
- [4] Wim Hesselink and Gerard Renardel de Lavalette, "Programmacorrectheid," Rijksuniversiteit Groningen, Groningen, Syllabus 2008.
- [5] Edward Fredkin, "Trie memory," *Communcations of the ACM*, vol. 3, no. 9, pp. 490-499, September 1960.
- [6] Wim Hesselink, "Talen en Automaten," Rijksuniversiteit Groningen, Groningen, Syllabus 2010.
- [7] Trishul Chilimbi, Mark Hill, and James Larus, "Cache-Conscious Structure Layout," in *ACM SIGPLAN 1999 conference on Programming language design and implementation*, New York, Mei 1999.
- [8] J. Scott Provan and Michael Ball, "The Complexity of Counting Cuts and of Computing the Probability that a Graph is Connected," *SIAM Journal on Computing*, vol. 12, no. 4, pp. 777-788, Februari 1983.
- [9] Christopher Granade. (2012, April) Stanford Complexity Zoo. [Online]. http://qwiki.stanford.edu/index.php/Complexity_Zoo:N#np
- [10] Wikipedia. (2012, April) Dedekind Number. [Online]. http://en.wikipedia.org/wiki/Dedekind_Number
- [11] Daniel Kleitman, "On Dedekind's Problem: The Number of Monotone Boolean Functions," *Proceedings of the American Mathematical Society*, vol. 21, no. 3, pp. 677-682, Juni 1969.
- [12] D Wiedermann, "A computation of the eighth Dedekind number," *Order*, vol. 8, no. 1, pp. 5-6, 1991.
- [13] Yong Chan Kim and Young Sun Kim, "Relations and Upper Sets on Partially Ordered Sets," *International Mathematical Forum*, vol. 6, pp. 899-908, 2011.
- [14] Giorgio Delzanno and Jean-Francois Raskin, "Symbolic Representation of Upwards Closed Sets," EECS Department University of California Berkeley, Berkeley, Technial Report 1999.

DATASETS

PO 10		
Elementen	10	
Relaties	12	
Opwaarts gesloten deelverzamelingen	41	
Hasse diagram		Relaties
	(0,1)	
	(0,2)	
	(0,3)	
	(1,4)	
	(1,5)	
	(2,4)	
	(2,6)	
	(3,5)	
	(3,6)	
	(4,7)	
	(5,8)	
	(6,9)	

PO 41		
Elementen	41	
Relaties	79	
Opwaarts gesloten deelverzamelingen	1891	
Hasse diagram		Relaties
	(E,0)	
	(0,1)	
	(0,2)	
	(0,3)	
	(1,12)	
	(1,13)	
	(2,12)	
	(2,23)	
	(3,13)	
	(3,23)	
	(12,4)	
	(12,123)	
	(13,5)	
	(13,123)	
	(23,6)	
	(23,123)	
	(4,7)	
	(4,34)	
	(5,8)	
	(5,25)	
	(6,9)	
	(6,16)	
	(123,34)	
	(123,25)	
	(123,16)	
	(7,37)	
	(8,28)	
	(9,19)	
	(34,37)	
	(34,45)	
	(34,46)	
	(25,45)	
	(25,28)	
	(25,56)	
	(16,46)	
	(16,19)	
	(16,56)	
	(37,57)	
	(37,67)	
	(28,48)	

BA 16																																		
Elementen	16																																	
Relaties	32																																	
Opwaarts gesloten deelverzamelingen	168																																	
Hasse diagram																																		
Relaties	<table><tr><td>(a,b)</td><td>(f,l)</td></tr><tr><td>(a,c)</td><td>(f,m)</td></tr><tr><td>(a,d)</td><td>(g,l)</td></tr><tr><td>(a,e)</td><td>(g,n)</td></tr><tr><td>(b,f)</td><td>(h,l)</td></tr><tr><td>(b,g)</td><td>(h,o)</td></tr><tr><td>(c,h)</td><td>(i,m)</td></tr><tr><td>(b,i)</td><td>(i,n)</td></tr><tr><td>(c,f)</td><td>(j,m)</td></tr><tr><td>(c,j)</td><td>(j,o)</td></tr><tr><td>(d,g)</td><td>(k,n)</td></tr><tr><td>(d,h)</td><td>(k,o)</td></tr><tr><td>(d,k)</td><td>(l,p)</td></tr><tr><td>(e,i)</td><td>(m,p)</td></tr><tr><td>(e,j)</td><td>(n,p)</td></tr><tr><td>(e,k)</td><td>(o,p)</td></tr></table>		(a,b)	(f,l)	(a,c)	(f,m)	(a,d)	(g,l)	(a,e)	(g,n)	(b,f)	(h,l)	(b,g)	(h,o)	(c,h)	(i,m)	(b,i)	(i,n)	(c,f)	(j,m)	(c,j)	(j,o)	(d,g)	(k,n)	(d,h)	(k,o)	(d,k)	(l,p)	(e,i)	(m,p)	(e,j)	(n,p)	(e,k)	(o,p)
(a,b)	(f,l)																																	
(a,c)	(f,m)																																	
(a,d)	(g,l)																																	
(a,e)	(g,n)																																	
(b,f)	(h,l)																																	
(b,g)	(h,o)																																	
(c,h)	(i,m)																																	
(b,i)	(i,n)																																	
(c,f)	(j,m)																																	
(c,j)	(j,o)																																	
(d,g)	(k,n)																																	
(d,h)	(k,o)																																	
(d,k)	(l,p)																																	
(e,i)	(m,p)																																	
(e,j)	(n,p)																																	
(e,k)	(o,p)																																	

EM 22																																										
Elementen	22																																									
Relaties	40																																									
Opwaarts gesloten deelverzamelingen	626																																									
Hasse diagram																																										
Relaties	<table><tr><td>(a,b)</td><td>(h,l)</td></tr><tr><td>(a,c)</td><td>(h,o)</td></tr><tr><td>(a,d)</td><td>(i,m)</td></tr><tr><td>(a,e)</td><td>(i,n)</td></tr><tr><td>(b,f)</td><td>(j,m)</td></tr><tr><td>(b,g)</td><td>(j,o)</td></tr><tr><td>(b,i)</td><td>(k,n)</td></tr><tr><td>(c,f)</td><td>(k,o)</td></tr><tr><td>(c,h)</td><td>(f,p)</td></tr><tr><td>(c,j)</td><td>(g,q)</td></tr><tr><td>(d,g)</td><td>(p,r)</td></tr><tr><td>(d,h)</td><td>(m,r)</td></tr><tr><td>(d,k)</td><td>(p,s)</td></tr><tr><td>(e,i)</td><td>(l,s)</td></tr><tr><td>(e,j)</td><td>(l,t)</td></tr><tr><td>(e,k)</td><td>(q,t)</td></tr><tr><td>(f,l)</td><td>(q,u)</td></tr><tr><td>(f,m)</td><td>(n,u)</td></tr><tr><td>(g,l)</td><td>(s,v)</td></tr><tr><td>(g,n)</td><td>(t,v)</td></tr></table>		(a,b)	(h,l)	(a,c)	(h,o)	(a,d)	(i,m)	(a,e)	(i,n)	(b,f)	(j,m)	(b,g)	(j,o)	(b,i)	(k,n)	(c,f)	(k,o)	(c,h)	(f,p)	(c,j)	(g,q)	(d,g)	(p,r)	(d,h)	(m,r)	(d,k)	(p,s)	(e,i)	(l,s)	(e,j)	(l,t)	(e,k)	(q,t)	(f,l)	(q,u)	(f,m)	(n,u)	(g,l)	(s,v)	(g,n)	(t,v)
(a,b)	(h,l)																																									
(a,c)	(h,o)																																									
(a,d)	(i,m)																																									
(a,e)	(i,n)																																									
(b,f)	(j,m)																																									
(b,g)	(j,o)																																									
(b,i)	(k,n)																																									
(c,f)	(k,o)																																									
(c,h)	(f,p)																																									
(c,j)	(g,q)																																									
(d,g)	(p,r)																																									
(d,h)	(m,r)																																									
(d,k)	(p,s)																																									
(e,i)	(l,s)																																									
(e,j)	(l,t)																																									
(e,k)	(q,t)																																									
(f,l)	(q,u)																																									
(f,m)	(n,u)																																									
(g,l)	(s,v)																																									
(g,n)	(t,v)																																									

EM 32

Elementen

32

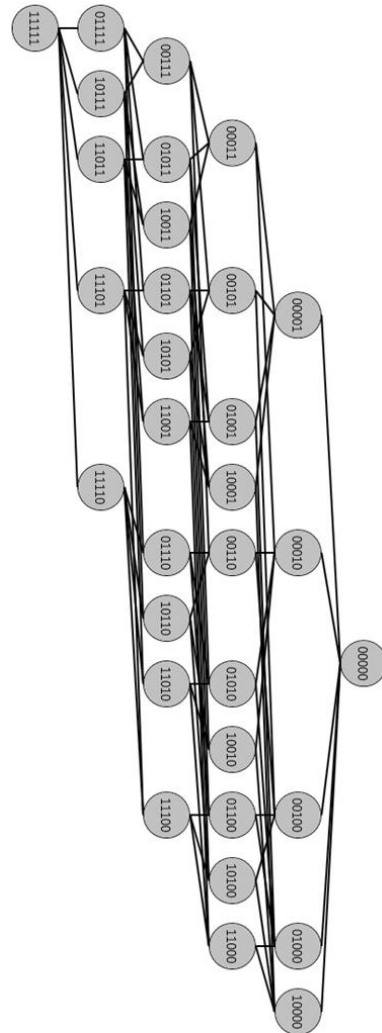
Relaties

80

Opwaarts gesloten deelverzamelingen

7581 (het 5^e Dedekind nummer)

Hasse diagram



Relaties

(00000,00001)	(01100,01101)
(00000,00010)	(01100,01110)
(00000,00100)	(01100,11100)
(00000,01000)	(10001,10011)
(00000,10000)	(10001,10101)
(00001,00011)	(10001,11001)
(00001,00101)	(10010,10011)
(00001,01001)	(10010,10110)
(00001,10001)	(10010,11010)
(00010,00011)	(10100,10101)
(00010,00110)	(10100,10110)
(00010,01010)	(10100,11100)
(00010,10010)	(11000,11001)
(00100,00101)	(11000,11010)
(00100,00110)	(11000,11100)
(00100,01100)	(00111,01111)
(00100,10100)	(00111,10111)
(01000,01001)	(01011,01111)
(01000,01010)	(01011,11011)
(01000,01100)	(01101,01111)
(01000,11000)	(01101,11101)
(10000,10001)	(01110,01111)
(10000,10010)	(01110,11110)
(10000,10100)	(10011,10111)
(10000,11000)	(10011,11011)
(00011,00111)	(10101,10111)
(00011,01011)	(10101,11101)
(00011,10011)	(10110,10111)
(00101,00111)	(10110,11110)
(00101,01101)	(11001,11011)
(00101,10101)	(11001,11101)
(00110,00111)	(11010,11011)
(00110,01110)	(11010,11110)
(00110,10110)	(11100,11101)
(01001,01011)	(11100,11110)
(01001,01101)	(01111,11111)
(01001,11001)	(10111,11111)
(01010,01011)	(11011,11111)
(01010,01110)	(11101,11111)
(01010,11010)	(11110,11111)

EM 64

Elementen

64

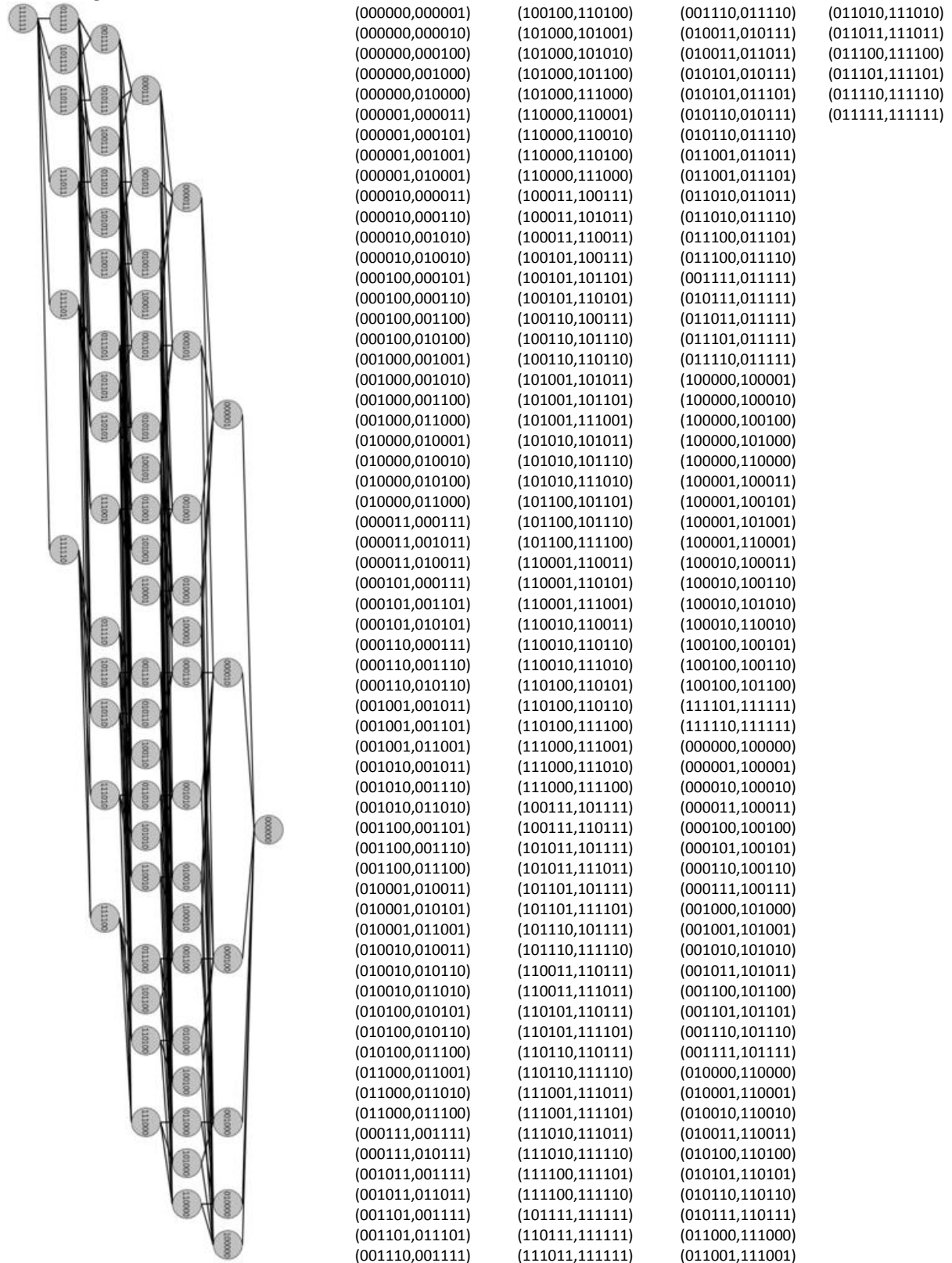
Relaties

192

Opwaarts gesloten deelverzamelingen 7828354 (het 6^e Dedekind nummer)

Hasse diagram

Relaties



BRONCODE

```
public class BruteForceCalculator : ICalculator
{
    public int Calculate(IEnumerable<Node> poSet)
    {
        int uppersets = 0;
        Node[] nodes = poSet.ToArray();
        BigInteger subsets = BigInteger.Pow(2, poSet.Count());
        BigInteger combination = new BigInteger(0);

        for (BigInteger i = 0; i < subsets; i++)
        {
            List<Node> subset = new List<Node>();
            BitArray bits = new BitArray(combination.ToByteArray());

            for (int y = 0; y < bits.Count; y++)
            {
                if (bits[y] == true) // if this bit has been set
                {
                    subset.Add(nodes[y]);
                }
            }

            Node[] subset_a = subset.ToArray();
            if (IsUpperSet(subset_a))
                ++uppersets;

            combination += 1;
        }
        return uppersets;
    }

    private bool IsUpperSet(Node[] subset)
    {
        foreach (Node n in subset)
        {
            foreach (Node parent in n.Parents)
            {
                if (!subset.Contains(parent))
                    return false;
            }
        }

        return true;
    }
}
```

Listing 1: Het naïve algoritme

```

public class FamilyCalculator : ICalculator
{
    public int Calculate(IEnumerable<Node> poSet)
    {
        return Generate(poSet).Count;
    }

    public List<Node[]> Generate(IEnumerable<Node> poSet)
    {
        List<Node[]> uppersets = new List<Node[]>();
        uppersets.Add(new Node[] { }); // add the empty set

        Dictionary<Node, int> ordering =
            SetUtilities.CreateTopologicalOrdering(poSet);
        List<Node> sorted = new List<Node>(from item in ordering orderby
            item.Value descending select item.Key);

        foreach (Node current in sorted)
        {
            // if all parents of current are in uppersets[i]
            // create and add a new upperset: {current} union uppersets[i]
            List<Node[]> found = FindUpperSetsWithElements(uppersets,
                current.Parents);

            foreach(Node[] upperSet in found)
            {
                Node[] upperset = new Node[upperSet.Length + 1];
                upperSet.CopyTo(upperset, 0);
                upperset[upperset.Length - 1] = current;
                uppersets.Add(upperset);
            }
        }

        return uppersets;
    }

    private static List<Node[]> FindUpperSetsWithElements(List<Node[]> uppersets,
        IList<Node> toSearch)
    {
        int uppersetCount = uppersets.Count;

        List<Node[]> found = new List<Node[]>();
        for (int i = 0; i < uppersetCount; i++)
        {
            bool containsAllParents = true;
            foreach (Node parent in toSearch)
            {
                if (!uppersets[i].Contains(parent))
                {
                    containsAllParents = false;
                    break;
                }
            }

            if (containsAllParents)
                found.Add(uppersets[i]);
        }
        return found;
    }
}

```

Listing 2: Het familiealgoritme zonder uptrie

```

public class UpTrieCalculator : ICalculator
{
    public int Calculate(IEnumerable<Node> poSet)
    {
        int count;
        UpTrie trie = FillTrie(poSet, out count);
        return count;
    }

    public List<Node[]> Generate(IEnumerable<Node> poSet)
    {
        int count;
        UpTrie trie = FillTrie(poSet, out count);
        return trie.GetUpperSets();
    }

    private static UpTrie FillTrie(IEnumerable<Node> poSet, out int count)
    {
        Dictionary<Node, int> ordering =
            SetUtilities.CreateTopologicalOrdering(poSet);
        List<Node> sorted = new List<Node>(from item in ordering orderby item.Value
                                         descending select item.Key);
        UpTrie trie = new UpTrie(ordering);

        count = 1; // for empty set
        foreach (Node current in sorted)
        {
            List<Node> parents = new List<Node>(from item in current.Parents orderby
                                                ordering[item] descending select item);
            List<UpTrieNode> found = trie.FindUpperSetsWithElements(parents);
            foreach (UpTrieNode end in found)
            {
                UpTrieNode child = new UpTrieNode(current);
                end.Children.Add(child);
                ++count;
            }
        }
        return trie;
    }
}

```

Listing 3: Het familiealgoritme met uptrie

```

public class UpTrie
{
    private UpTrieNode treeRoot;
    private Dictionary<Node, int> ordering;
    private static Node EmptyHeadNode = new Node("\u03B5"); // empty symbol

    public UpTrie(Dictionary<Node, int> ordering)
    {
        this.ordering = ordering;
        ordering.Add(EmptyHeadNode, Int32.MaxValue);
        treeRoot = new UpTrieNode(EmptyHeadNode);
    }

    public List<UpTrieNode> FindUpperSetsWithElements(IList<Node> toSearch)
    {
        List<UpTrieNode> ends = new List<UpTrieNode>();
        if (toSearch.Count == 0)
        {
            ends.Add(treeRoot);
            ends.AddRange(GetAllLowerNodes(treeRoot));
            return ends;
        }

        Stack<Frame> stack = new Stack<Frame>();
        stack.Push(new Frame(treeRoot, 0));

        while (stack.Count > 0)
        {
            Frame frame = stack.Pop();
            int index = frame.index;

            int o_search = ordering[toSearch[frame.index]];
            int o_root = ordering[frame.item.Content];

            if (o_search == o_root)
            {
                ++index;

                if (index == toSearch.Count)
                {
                    ends.Add(frame.item);
                    ends.AddRange(GetAllLowerNodes(frame.item));
                    continue;
                }
            }
            else if (o_search > o_root)
            {
                continue;
            }

            foreach (UpTrieNode child in frame.item.Children)
            {
                stack.Push(new Frame(child, index));
            }
        }

        return ends;
    }
}

```

Listing 4: De uptrie

```

public List<Node[]> GetUpperSets()
{
    List<Node[]> uppersets = new List<Node[]>();
    Stack<LabelFrame> stack = new Stack<LabelFrame>();
    stack.Push(new LabelFrame(new Node[] {}, treeRoot));

    while (stack.Count > 0)
    {
        LabelFrame frame = stack.Pop();
        uppersets.Add(frame.UpperSet);

        foreach (UpTrieNode child in frame.Head.Children)
            stack.Push(new LabelFrame(frame.UpperSet, child));
    }

    return uppersets;
}

private IEnumerable<UpTrieNode> GetAllLowerNodes(UpTrieNode root)
{
    List<UpTrieNode> nodes = new List<UpTrieNode>();
    Stack<UpTrieNode> stack = new Stack<UpTrieNode>();
    stack.Push(root);

    while (stack.Count > 0)
    {
        UpTrieNode current = stack.Pop();
        foreach (UpTrieNode child in current.Children)
        {
            stack.Push(child);
            nodes.Add(child);
        }
    }

    return nodes;
}

```

Listing 4: De uptrie (vervolg)

```

private class LabelFrame
{
    public Node[] UpperSet { get; private set; }
    public UpTrieNode Head { get; private set; }
    public LabelFrame(Node[] prevUpperset, UpTrieNode head)
    {
        this.Head = head;
        if (head.Content != null && !head.Content.Equals(EmptyHeadNode))
        {
            this.UpperSet = new Node[prevUpperset.Length + 1];
            Array.Copy(prevUpperset, this.UpperSet, prevUpperset.Length);
            this.UpperSet[prevUpperset.Length] = head.Content;
        }
        else
        {
            UpperSet = new Node[] { };
        }
    }
}

private class Frame
{
    public UpTrieNode item;
    public int index;

    public Frame(UpTrieNode item, int index)
    {
        this.item = item;
        this.index = index;
    }

    public override string ToString()
    {
        return item.Content.ToString() + " : " + index.ToString();
    }
}
}

```

Listing 4: De uptrie (vervolg)

```

public static class SetUtilities
{
    public static List<Node[]> GenerateSubsets(IEnumerable<Node> set)
    {
        List<Node[]> subsets = new List<Node[]>((int)Math.Pow(set.Count(), 2));
        foreach (Node node in set)
        {
            Node[] single = new Node[] { node };
            int subsetCount = subsets.Count;

            for (int i = 0; i < subsetCount; i++)
            {
                Node[] newSubset = new Node[subsets[i].Length + 1];
                subsets[i].CopyTo(newSubset, 0);
                newSubset[newSubset.Length - 1] = node;
                subsets.Add(newSubset);
            }
            subsets.Add(single);
        }

        subsets.Add(new Node[] { }); // add the empty set
        return subsets;
    }

    public static Dictionary<Node, int> CreateTopologicalOrdering(IEnumerable<Node>
                                                                    poSet)
    {
        Dictionary<Node, int> parentsProcessed = new Dictionary<Node, int>();
        Dictionary<Node, int> ordering = new Dictionary<Node, int>();
        int rank = poSet.Count();
        Queue<Node> queue = new Queue<Node>();

        foreach (Node n in poSet)
        {
            parentsProcessed.Add(n, 0);

            if (parentsProcessed[n] == n.Parents.Count)
                queue.Enqueue(n);
        }

        while (queue.Count > 0)
        {
            Node current = queue.Dequeue();
            ordering.Add(current, rank--);

            foreach (Node child in current.Children)
            {
                parentsProcessed[child]++;
                if (parentsProcessed[child] == child.Parents.Count)
                {
                    queue.Enqueue(child);
                }
            }
        }

        return ordering;
    }
}

```

Listing 5: Utilities


```

public class UpTrieNode
{
    public MinimalList<UpTrieNode> Children { get; set; }
    public Node Content { get; set; }

    public UpTrieNode(Node content)
    {
        this.Content = content;
        Children = new MinimalList<UpTrieNode>();
    }

    public override string ToString()
    {
        return Content.ToString() + ", c: " + Children.Count.ToString();
    }
}

```

Listing 6: De klasse voor de elementen in de uptrie

```

public class Node
{
    public MinimalList<Node> Children { get; set; }
    public MinimalList<Node> Parents { get; set; }
    public string Decoration { get; set; } // should be unique

    public Node(string decoration)
    {
        Children = new MinimalList<Node>();
        Parents = new MinimalList<Node>();

        this.Decoration = decoration;
    }

    public override bool Equals(object obj)
    {
        if (obj is Node)
        {
            Node other = (Node)obj;
            return other.Decoration.Equals(this.Decoration);
        }
        else
        {
            return false;
        }
    }

    public override int GetHashCode()
    {
        return Decoration.GetHashCode();
    }

    public override string ToString()
    {
        return Decoration;
    }
}

```

Listing 7: De klasse voor elementen uit partieel geordende verzamelingen