

UNIVERSITEIT VAN GRONINGEN

GEVORDERDE ALGORITMEN EN DATASTRUCTUREN

DOOR

JOS VAN DER TIL & RENE ZUIDHOF

21 JANUARI 2011

Inhoudsopgave

1	Introductie	2
2	Maximum flow probleem	3
2.1	Ford-Fulkerson algoritme	4
2.1.1	Pseudocode	4
3	Depth-first search	6
3.1	Pseudocode	6
3.1.1	Analyse	7
4	Breadth-first search	9
4.1	Pseudocode	9
4.2	Analyse	9
5	Priority First Search	11
5.0.1	Pseudocode	11
5.0.2	Analyse	11
6	Conclusie	13
	Lijst van figuren	14
	Lijst van tabellen	15
	Lijst van algoritmen	16
A	Source Code	17

Hoofdstuk 1

Introductie

Dit verslag maakt deel uit van de cursus Gevorderde Algoritmen en Datastructuren van de Rijksuniversiteit Groningen. In dit verslag zal de tweede practicum opdracht behandeld worden. Deze opdracht omvat het vinden van een maximum flow in een flow network en de verschillende manieren, om dit te doen, te analyseren.

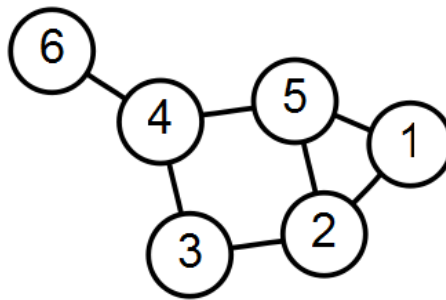
In hoofdstuk 2 zal het probleem van het vinden van een maximum flow en het gebruikte algoritme beschreven worden. De hoofdstukken 3, 4 & 5 beschrijven de algoritmen die gebruikt worden om een pad te vinden door het netwerk. De conclusies van dit onderzoek staan in hoofdstuk 6.

Hoofdstuk 2

Maximum flow probleem

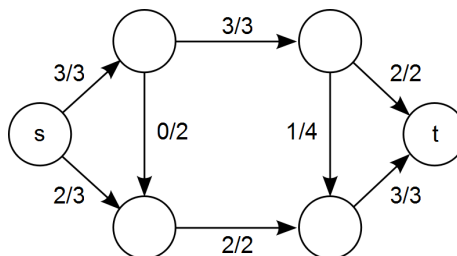
Om het probleem van het vinden van een maximum flow te kunnen begrijpen, volgt hier een korte introductie in de grafentheorie.

Een graaf is een verzameling punten (knopen) die verbonden zijn door lijnen (kanten). De kanten van een graaf kunnen een richting en/of een gewicht hebben. Een voorbeeld van een simpele graaf is te zien in figuur 2.1.



Figuur 2.1: Een ongerichte en ongewogen graaf met 6 nodes

In figuur 2.2 is een voorbeeld van een flow network te zien. De flow in deze afbeelding is maximaal, immers de capaciteit van de beide kanten die leiden naar t is volledig benut. Tevens is het duidelijk dat dit een gerichte (pijlen in plaats van lijnen als kanten) en gewogen (getallen bij de kanten) graaf is.



Figuur 2.2: Voorbeeld van een flow network met een maximum flow van s naar t . De getallen zijn flow / max capaciteit.

Het probleem is nu om een flow te vinden van s naar t die maximaal is. Om dit op te lossen is er het Ford-Fulkerson algoritme, vernoemd naar L.R. Ford en D.R. Fulkerson die dit algoritme publiceerden in 1956. Deze wordt nader toegelicht in paragraaf 2.1.

2.1 Ford-Fulkerson algoritme

Het algoritme van Ford & Fulkerson werkt eigenlijk volgens een heel simpel principe. Zolang er een pad is van s naar t met beschikbare capaciteit, dan wordt de flow daar langs gestuurd. Dit wordt herhaalt totdat er geen pad meer mogelijk is. Een pad van s naar t met beschikbare capaciteit wordt een 'augmenting path' genoemd.

De eisen die gesteld worden aan een geldige flow zijn:

- De flow mag nooit groter zijn dan de capaciteit van een kant. $0 \leq flow(u, v) \leq capacity(u, v)$
- De netto flow van een node is gelijk aan 0. Dit geldt niet voor s of t .

$$\sum_{e \in E^-} flow(e) - \sum_{v \in E^+} flow(v) = 0$$

Waar E^- de verzameling van uitgaande kanten is en E^+ de verzameling inkomende kanten van knoop E is.

Omdat het Ford-Fulkerson algoritme niet aangeeft op welke manier er een 'augmenting path' gevonden dient te worden, zijn er meerdere methodes beschikbaar. De methodes die onderzocht zullen worden in dit document zijn:

1. Depth-first search;
2. Breadth-first search;
3. Priority-first search.

2.1.1 Pseudocode

De pseudocode van het algoritme is te vinden in algoritme 1.

Algoritme 1 Ford-Fulkerson Algorithm

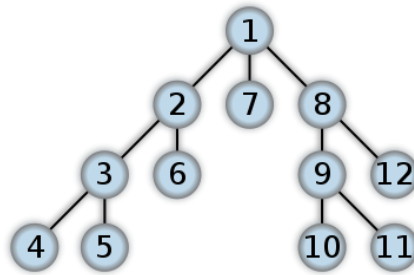
Require: Input: Flow network N containing graph G

```
for all edge  $e \in N$  do
     $\text{flow}(e) \leftarrow 0$ 
end for
 $\text{stop} \leftarrow \text{false}$ 
repeat
    traverse  $G$  starting at  $s$  to find an augmenting path to  $t$  ( $\pi$ )
    if an augmenting path  $\pi$  exists then
         $\Delta \leftarrow +\infty$ 
        for all edge  $e \in \pi$  do
            if  $\text{residual capacity}(e) \leq \Delta$  then
                 $\Delta \leftarrow \text{residual capacity}(e)$ 
            end if
        end for
        for all edge  $e \in \pi$  do
            if  $e$  is a forward edge then
                 $\text{flow}(e) \leftarrow \text{flow}(e) + \Delta$ 
            else
                 $\text{flow}(e) \leftarrow \text{flow}(e) - \Delta$ 
            end if
        end for
    else
         $\text{stop} \leftarrow \text{true}$ 
    end if
until  $\text{stop}$ 
```

Hoofdstuk 3

Depth-first search

De eerste methode die onderzocht is voor het zoeken naar een augmented path is de Depth-first search methode. Deze methode zal, zoals de naam suggereert, de diepte in gaan op zoek naar t . Dit is geïllustreerd in figuur 3.1, de getallen op de knopen geven aan in welke volgorde ze doorzocht zijn.



Figuur 3.1: Depth-first doorzoeken van een boom

De recursieve methode DFS verwacht als invoer een graaf g , een startknoop s , een eindknoop t en een map $parents$. Deze map wordt later gebruikt om de weg van t naar s te vinden. Bij elke aanroep zal s gelabeld worden als *EXPLORED* \wedge . Hierna zullen alle aanliggende kanten van s bijlangs gegaan worden om te kijken of hier nog een eventueel pad mogelijk is. Dit wordt gedaan door te kijken naar de overstaande knoop w via e . Wanneer w gelabeld is als *UNEXPLORED* en de kant e nog een capaciteit heeft, is hier een pad mogelijk. e zal nu gezet worden als de parent van w en daarnaast ook nog gelabeld worden als *DISCOVERY* \wedge . Nu zal een recursieve aanroep gedaan worden met de parameters respectievelijk g , w , t en $parents$. Als w niet gelabeld is als *UNEXPLORED* zal e gelabeld worden als *BACK* \wedge .

3.1 Pseudocode

De pseudocode waar de code op gebaseerd is, is te vinden in algoritme 2. De werkelijke code heeft een paar toevoegingen zodat deze stopt wanneer het eindpunt t bereikt is. Wanneer dit het geval is kan met behulp van de $parents$

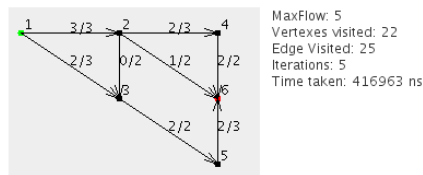
gezocht worden naar een pad van s naar t door te kijken wat de parent edge e is van t . Nu zal gekeken worden naar de parent edge van de overstaande van t via edge e . Door dit te doen tot er geen parent edge is zal s bereikt worden.

Algoritme 2 Depth-first search Algorithm

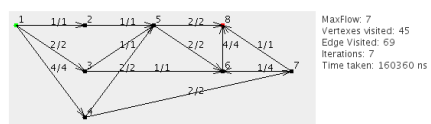
Require: Input: Graph g , Start vertex s , End vertex t , HashMap parents with vertexes and its parent edges
 Label s as *EXPLORED*
for all edge $e \in s.incidentEdges$ **do**
 if e is not labeled as *UNEXPLORED* \wedge $s.residualCapacity(e) > 0$ **then**
 $w \leftarrow g.opposite(s, e)$
 if w is labeled as *UNEXPLORED* **then**
 label e as *DISCOVERY* edge
 set e as parent of w in the hashmap parents
 recursive call with g , w , t and parents
 else
 label e as *BACK* edge
 end if
end if
end for

3.1.1 Analyse

Figuur 3.2 en 3.3 laten de uitkomst zien van de analyse.



Figuur 3.2: Analyse van de eerste graaf



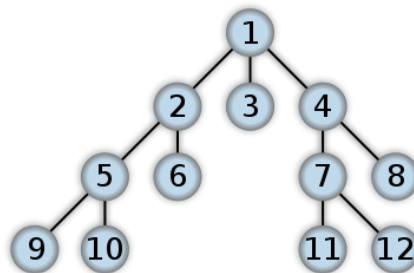
Figuur 3.3: Analyse van de tweede graaf

Hoofdstuk 4

Breadth-first search

De tweede manier om een augmenting path te vinden in een graaf is de breadth-first search. Deze methode, die ook gebruikt wordt in het Edmonds-Karp algoritme, vindt het kortste pad van s naar t . Het kortste pad is in dit geval gedefinieert als het pad met het laagste aantal kanten.

Het breadth-first doorlopen van een graaf is niet anders dan dat dat bij een tree gebeurt, elk niveau wordt volledig doorzocht, voordat het algoritme naar het niveau daar onder gaat. Dit is te zien in figuur 4, de getallen op de knopen geven aan in welke volgorde de boom doorzocht wordt.



Figuur 4.1: Breadth-first doorlopen van een boom

Om vervolgens het pad van s naar t te vinden in een graaf, wordt eerst de graaf doorlopen volgens het breadth-first principe totdat t gevonden is. Terwijl de graaf doorlopen wordt, wordt bijgehouden welke kant leid naar welke knoop. Hierdoor is het gemakkelijk om het pad van t naar s terug te vinden. De pseudocode voor dit algoritme is te vinden in algoritme 3.

4.1 Pseudocode

4.2 Analyse

Algorithme 3 Breadth-first search path finding

Require: **Input:** Graph G , Node s , Node t **Output:** An augmenting path, or an empty path if none found. $Q \leftarrow$ new queue $M \leftarrow$ new hashmap $s.state \leftarrow EXPLORED$ $Q.enqueue(s)$ **while** $\neg Q.isEmpty()$ **do** $v \leftarrow Q.dequeue()$ **for all** edge $e \in G.incidentEdges(v)$ **do****if** $e.state = UNEXPLORED \wedge e.residualCapacity() > 0$ **then** $w \leftarrow G.opposite(v, e)$ **if** $\neg w.state = EXPLORED$ **then** $Q.enqueue(w)$ $Q.state = EXPLORED$ $M.put(w, e)$ $\{w$ discovered through edge $e\}$ $e.state = DISCOVERY$ **if** $e.start = w$ **then**Mark e as forward**else**Mark e as backward**end if****if** $w = t$ **then**pathFound \leftarrow **false** $p \leftarrow w$ $path \leftarrow$ new list**while** $\neg pathFound$ **do** $c \leftarrow M.get(p)$ $\{Retreive$ edge c that led to $p\}$ $path.add(c)$ $\{Add$ edge c to the path $\}$ $p \leftarrow G.opposite(p, c)$ $\{Go$ back another step in the graph $\}$ **if** $p = s$ **then**pathFound \leftarrow **true** $\{We$ found the start node, we are done. $\}$ **end if****end while****return** path**end if****end if****else** $e.state \leftarrow BACKWARD$ **end if****end for****end while****return** empty list $\{No$ path found $\}$

Hoofdstuk 5

Priority First Search

De laatste methode die onderzocht is, is Dijkstra's algoritme. Dit algoritme geeft prioriteiten aan de knopen en gaat aan de hand hiervan de knopen bijlans.

De methode verwacht als invoer een graaf g , een startknoop s en een eindknoop t . Als eerste wordt de maxflow van alle knopen in g op 0 gezet, behalve de maxflow van s , deze krijgt de waarde ∞ . Al deze knopen worden toegevoegt aan een priorityqueue Q , de maxflow van de knopen wordt gebruikt als de prioriteit. Nu zal het algoritme doorgaan tot Q leeg is. Elke keer zal de hoogste waarde in Q verwijderd worden, deze waarde krijgt hier de naam u . De eerste keer zal dit s zijn. Elke keer wanneer een knoop u uit Q gehaald wordt zullen alle kanten van u bezocht worden. Voor elke kant zal de overstaande knoop (z) van u via kant e opgezocht worden. De flow die naar z kan is de maxflow van u of de residual capacity van e als deze lager is dan de maxflow van u . Wanneer de flow van u naar z hoger is dan de huidige maxflow van z , zal deze bijgewerkt worden. Nu zal ook de waarde van z in Q geupdate worden.

5.0.1 Pseudocode

De pseudocode waar de code op gebaseerd is is te vinden in algoritme 4. De werkelijke code heeft een paar toevoegingen zodat deze stopt wanneer het eindpunt t bereikt is. Wanneer dit het geval is kan met behulp van de *parents* gezocht worden naar een pad van s naar t door te kijken wat de parent edge e is van t . Nu zal gekeken worden naar de parent edge van de overstaande van t via edge e . Door dit te doen tot er geen parent edge is zal s bereikt worden.

5.0.2 Analyse

Figuur 5.1 en 5.2 laten de uitkomst zien van de analyse.

Algoritme 4 Dijkstra's Algorithm

Require: Input: Graph g , Start vertex s , End vertex t

HashMap parents with vertexes and edges

$Q \leftarrow$ new PriorityQueue

for all vertex $v \in g.vertexes$ **do**

if $v = s$ **then**

$v.maxFlow \leftarrow \infty$

else

$v.maxFlow \leftarrow 0$

end if

$Q.add(maxFlow, v)$

 set parent to \emptyset

end for

while Q is not empty **do**

$u \leftarrow Q.removeMax()$

for all edge $e \in u.incidentEdges$ **do**

$z \leftarrow g.opposite(u, e)$

$r \leftarrow \min(u.getResidualCapacity(e), u.maxFlow)$

if $r < z.maxFlow \wedge e \neq parentofz$ **then**

$z.maxFlow \leftarrow r$

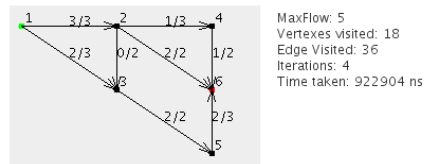
 set e as parent of z

 update z in Q

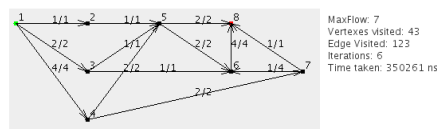
end if

end for

end while



Figuur 5.1: Analyse van de eerste graaf



Figuur 5.2: Analyse van de tweede graaf

Hoofdstuk 6

Conclusie

Lijst van figuren

2.1	Een ongerichte en ongewogen graaf met 6 nodes	3
2.2	Voorbeeld van een flow netwerk met een maximum flow van s naar t . De getallen zijn flow / max capaciteit.	3
3.1	Depth-first doorzoeken van een boom	6
3.2	Analyse van de eerste graaf	7
3.3	Analyse van de tweede graaf	8
4.1	Breadth-first doorlopen van een boom	9
5.1	Analyse van de eerste graaf	12
5.2	Analyse van de tweede graaf	12

Lijst van tabellen

Lijst van algoritmen

1	Ford-Fulkerson Algorithm	5
2	Depth-first search Algorithm	7
3	Breadth-first search path finding	10
4	Dijkstra's Algorithm	12

Bijlage A

Source Code

Source code HIER