Universiteit van Groningen

Gevorderde Algoritmen en Datastructuren

door

Jos van der Til & Rene Zuidhof

21 januari 2011

Inhoudsopgave

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.

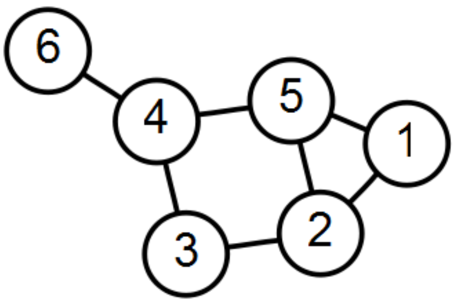
In hoofdstuk 2 zal het probleem van het vinden van een maximum flow en het gebruikte algoritme beschreven worden. De hoofdstukken 3, 4 & 5 beschrijven de algoritmen die gebruikt worden om een pad te vinden door het netwerk. De conclusies van dit onderzoek staan in hoofdstuk 6.

Hoofdstuk 2

Maximum flow probleem

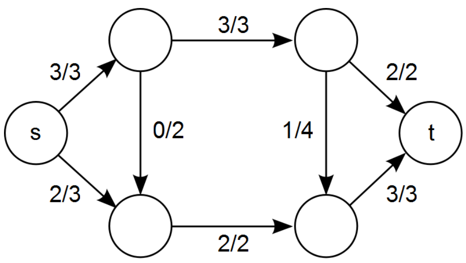
Om het probleem van het vinden van een maximum flow te kunnen begrijpen, volgt hier een korte introductie in de grafentheorie.

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 netwerk 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 de de capacity 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.

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

  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 ∈ N do
    flow(e) ← 0
  end for
  stop ← false
  repeat
    traverse G starting at s to find an augmenting path to t (π)
    if an augmenting path π exists then
      Δ ← +∞
      for all edge e ∈ π do
        if  residual capacity(e) ≤ Δ then
          Δ ← residual capacity(e)
        end if
      end for
      for all edge e ∈ π do
        if e is a forward edge then
          flow(e) ← flow(e) +Δ
        else
          flow(e) ← flow(e) −Δ
        end if
      end for
    else
      stop ← true
    end if
  until 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.

Tijdens het zoeken naar t worden de edges gelabeld met *discovery*, *unexplored* en *back*.

3.1 Pseudocode

De pseudocode waar de code op gebaseerd is, is te vinden in algoritme 2.

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 ∈ s.incidentEdges do
  if e is not labeled as UNEXPLORED ∧ s.residualCapacity(e) > 0 then
    w ← 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
```

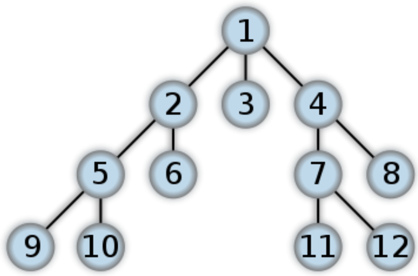
Wanneer het eindpunt t bereikt is kan het algoritme stoppen. Nu kan met behulp van de *parents* gezocht worden naar een pad van s naar t door te kijken wat de parent edge e is van t. Nu zal gekeken worden naar de parent edge van de overstaande van t via edge e. Door dit te doen tot er geen parent edge is zal s bereikt worden.

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 gedefineert 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

Algoritme 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

Hoofdstuk 6

Conclusie

Lijst van figuren

Lijst van tabellen

Bijlage A

Source Code

Source code HIER