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 3 zal het probleem van het vinden van een maximum flow en het gebruikte algoritme beschreven worden. De hoofdstukken 4, 5 & 6 beschrijven de algoritmen die gebruikt worden om een pad te vinden door het netwerk. De conclusies van dit onderzoek staan in hoofdstuk 7.

Hoofdstuk 2

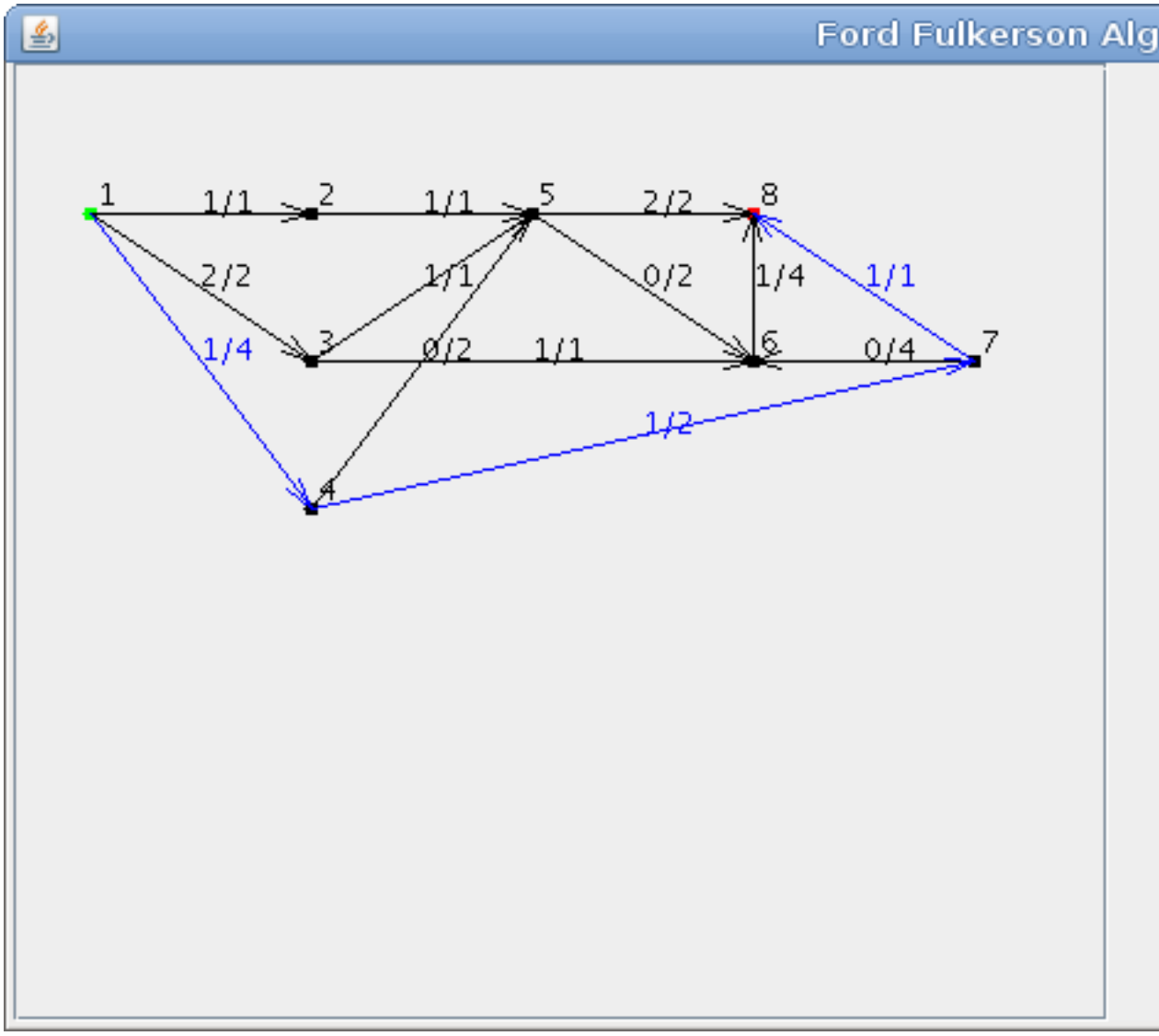
Software

Om de methodes te visualiseren is een GUI ontwikkeld, deze zal in dit hoofdstuk beschreven worden. In figuur 2.1 is de GUI te zien. De linkerkant laat de graaf zien, en aan de rechterkant zijn controllers aanwezig om de software te bedienen. Het startpunt en eindpunt worden weergegeven door respectievelijk de kleuren groen en de kleur rood. De paden waarover een flow wordt gestuurd zijn weergegeven met de kleur blauw. De knop 'Next flow' zal 1 augmented path zoeken met behulp van de methode die gekozen kan worden in de dropdown box. Wanneer een augmented path is gevonden zal de maximale flow voor dat pad er doorheen gestuurd worden. De knop 'Max flow' zal de actie die gedaan wordt voor de knop 'Next flow' uitvoeren tot er geen augmented path meer te vinden is. De dropdown box die zojuist genoemd is, is te vinden aan de rechterkant van de knop 'Max flow'. In deze dropdown box kan gekozen worden voor de methode die gebruikt dient te worden voor het zoeken naar een augmented path. Er kan gekozen worden tussen DFS (Depth-first search), BFS (Breadth-first search) en $DIJKSTRA$ (Dijkstra's algorime). De knop daarnaast, 'Reset', zal alle waardes voor de graaf weer op hun oorspronkelijke waarde zetten. Onder de zojuist genoemde knoppen is een knop en een veld te vinden waarmee een graaf bestand geladen kan worden, deze bestanden moeten geplaatst zijn in de hoofdmap van de software. De opbouw van dit bestand zal later besproken worden. Wanneer een bestandsnaam is ingevoerd en de knop 'Load' wordt ingedrukt zal de graaf geladen worden aan de linkerkant. Onder de knop 'Load' en het invoerveld is een veld te vinden waarin de status van de graaf weergegeven zal worden.

Hieronder is een voorbeeld te zien van de opbouw van het bestand voor graaf 1. Op de eerste regel staan respectievelijk het aantal knopen en het aantal kanten. Op de tweede regel staat respectievelijk het startpunt en het eindpunt. De volgende regels bevatten de kanten van de graaf door respectievelijk de beginknoop, de eindknoop en de capaciteit van de kant te benoemen.

```
6 8
1 6
1 2 3
1 3 3
2 3 2
2 4 3
```

2 6 2
3 5 2
5 6 3
4 6 2



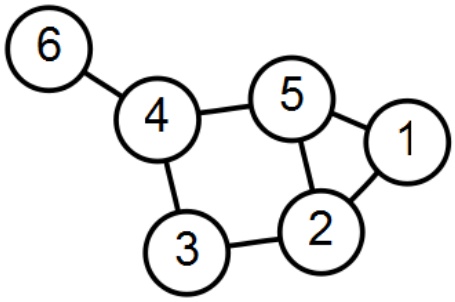
Figuur 2.1: Screenshot van de GUI

Hoofdstuk 3

Maximum flow probleem

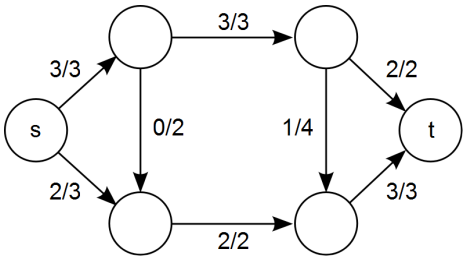
Om het probleem van het vinden van een maximum flow te kunnen begrijpen, volgt hier een korte introductie in de grafentheorie.

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 3.1.



Figuur 3.1: Een ongerichte en ongewogen graaf met 6 nodes

In figuur 3.2 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 3.2: 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 3.1.

3.1 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.

3.1.1 Pseudocode

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

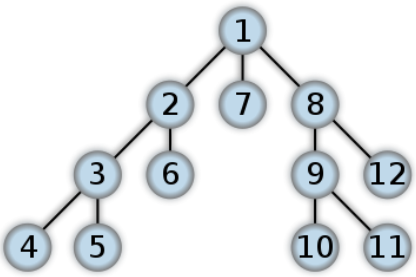
Algoritme 1 Ford-Fulkerson Algorithm

Require: Input: Flow network N containing graph G
  for all edge $e \in N$ **do**
    flow(e) $\leftarrow 0$
  end for
  stop $\leftarrow$ **false**
  repeat
    traverse G starting at s to find an augmenting path to t (π)
    if an augmenting path π exists **then**
      $\Delta \leftarrow +\infty$
      for all edge $e \in \pi$ **do**
        if residual capacity(e) $\leq \Delta$ **then**
          $\Delta \leftarrow$ residual capacity(e)
        end if
      end for
      for all edge $e \in \pi$ **do**
        if e is a forward edge **then**
          flow(e) $\leftarrow$ flow(e) $+\Delta$
        else
          flow(e) $\leftarrow$ flow(e) $-\Delta$
        end if
      end for
    else
      stop $\leftarrow$ **true**
    end if
  until *stop*

Hoofdstuk 4

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 4.1, de getallen op de knopen geven aan in welke volgorde ze doorzocht zijn.



Figuur 4.1: 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* $\wedge$.

4.1 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

4.2 Analyse

De grafen in 4.2 en 4.3 laten de uitkomst zien van het Ford-Fulkerson algoritme. De tabellen 4.1 en 4.2 geven de benchmarks weer voor de respectievelijke grafen.

Maximale flow	5
Knopen bezocht	22
Kanten bezicht	25
Iteraties	5
Benodigde tijd	416.963 ns

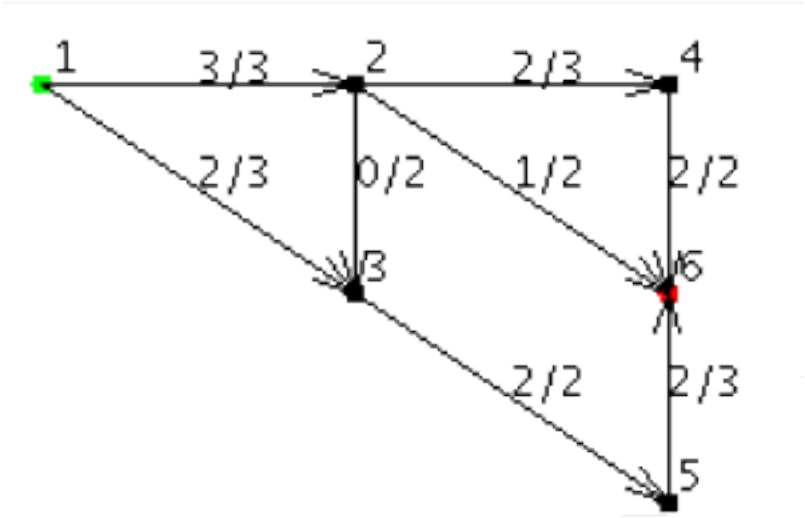
Tabel 4.1: Resultaten voor graaf in figuur 4.2

.

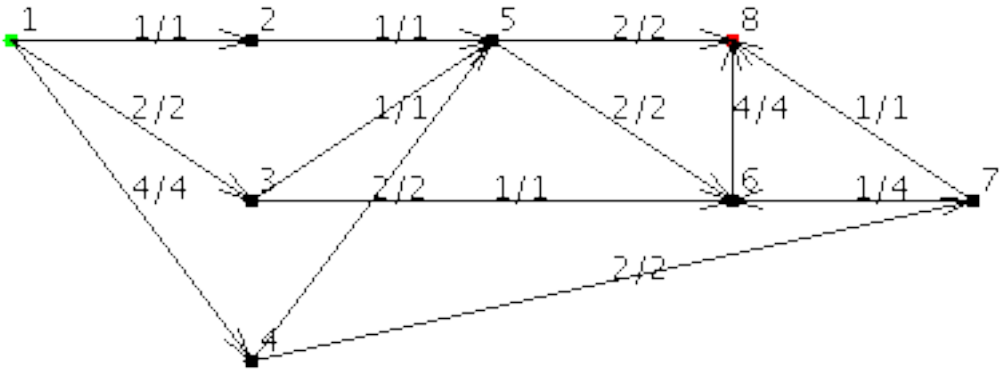
Maximale flow	7
Knopen bezocht	43
Kanten bezicht	1234
Iteraties	6
Benodigde tijd	350.261 ns

Tabel 4.2: Resultaten voor graaf in figuur 4.3

.



Figuur 4.2: Analyse van de eerste graaf



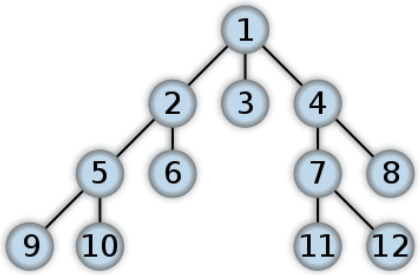
Figuur 4.3: Analyse van de tweede graaf

Hoofdstuk 5

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 5, de getallen op de knopen geven aan in welke volgorde de boom doorzocht wordt.



Figuur 5.1: 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.

5.1 Pseudocode

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 ← new queue
  M ← new hashmap
  s.state ← EXPLORED
  Q.enqueue(s)
  while ¬Q.isEmpty() do
    v ← Q.dequeue()
    for all edge e ∈ G.incidentEdges(v) do
      if e.state = UNEXPLORED ∧e.residualCapacity() > 0 then
        w ← G.opposite(v, e)
        if ¬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 ← false
            p ← w
            path ← new list
            while ¬pathFound do
              c ← M.get(p) {Retreive edge c that led to p}
              path.add(c) {Add edge c to the path}
              p ← G.opposite(p, c) { Go back another step in the graph}
              if p = s then
                pathFound ← true {We found the start node, we are done.}
              end if
            end while
            return path
          end if
        end if
      else
        e.state ← BACKWARD
      end if
    end for
  end while
  return empty list {No path found}
```

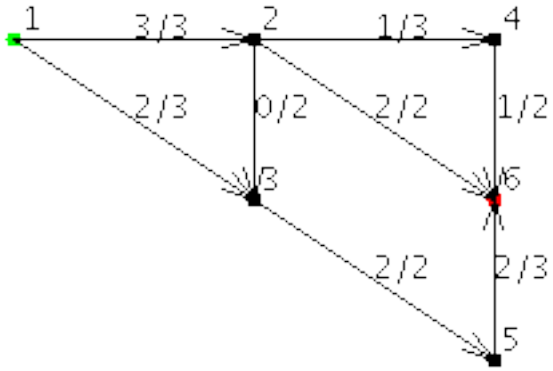
5.2 Analyse

De grafen in 5.2 en 5.3 laten de uitkomst zien van het Ford-Fulkerson algoritme. De tabellen 5.1 en 5.2 geven de benchmarks weer voor de respectievelijke grafen.

Maximale flow	5
Knopen bezocht	11
Kanten bezicht	15
Iteraties	4
Benodigde tijd	60.625 ns

Tabel 5.1: Resultaten voor graaf in figuur 5.2

.

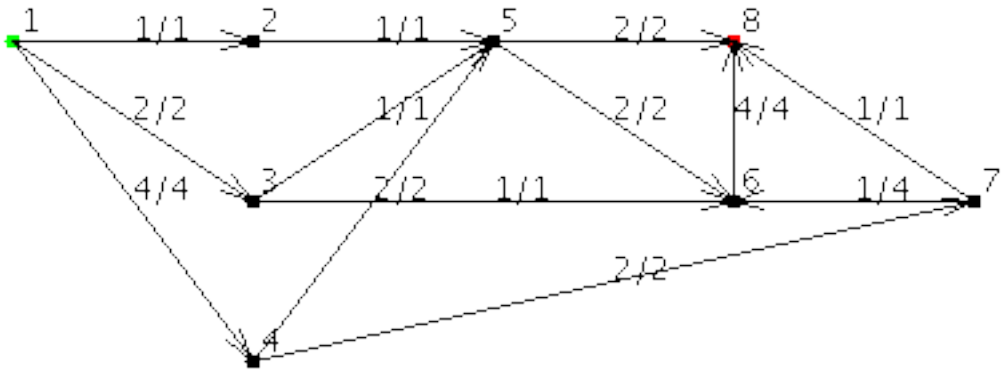


Figuur 5.2: Analyse van de eerste graaf

Maximale flow	7
Knopen bezocht	29
Kanten bezicht	48
Iteraties	7
Benodigde tijd	441.687 ns

Tabel 5.2: Resultaten voor graaf in figuur 5.3

.



Figuur 5.3: Analyse van de tweede graaf

Hoofdstuk 6

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 bijlangs.

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.

6.0.1 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.

6.0.2 Analyse

Figuur 6.1 en 6.2 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 ← 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 $r < z.maxFlow \wedge e$ not parent of z **then**
        $z.maxFlow \leftarrow r$
        set e as parent of z
        update z in Q
      end if
    end for
  end while

Maximale flow	5
Knopen bezocht	18
Kanten bezicht	36
Iteraties	4
Benodigde tijd	922.904 ns

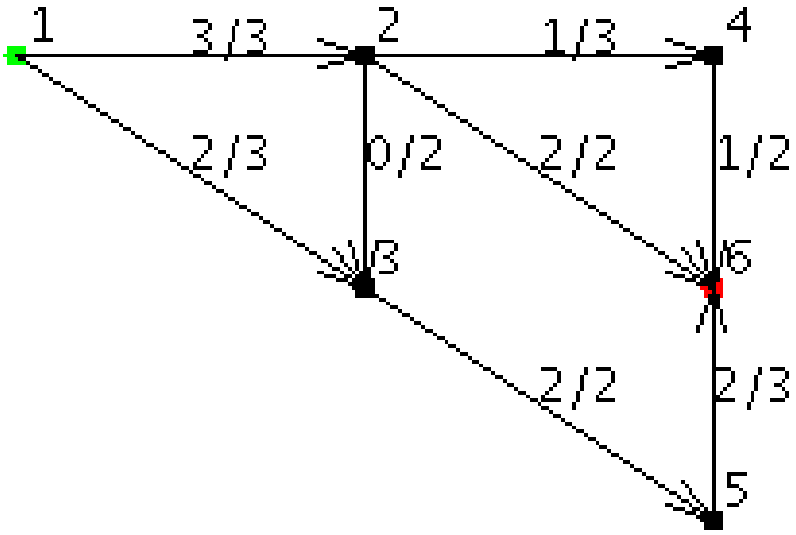
Tabel 6.1: Resultaten voor graaf in figuur 6.2

.

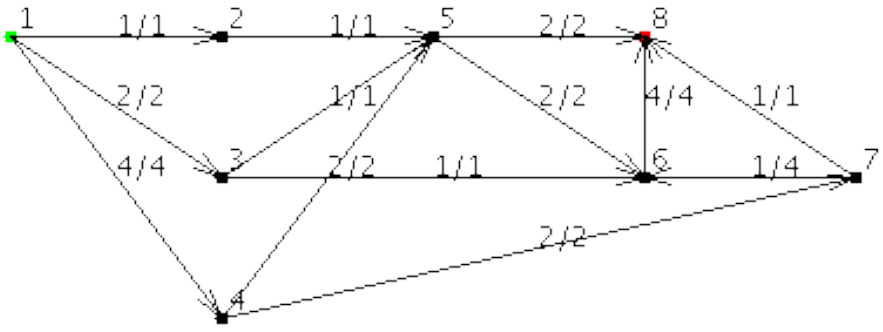
Maximale flow	7
Knopen bezocht	43
Kanten bezicht	123
Iteraties	6
Benodigde tijd	350.261 ns

Tabel 6.2: Resultaten voor graaf in figuur 6.2

.



Figuur 6.1: Analyse van de eerste graaf



Figuur 6.2: Analyse van de tweede graaf

Hoofdstuk 7

Conclusie

Lijst van figuren

Lijst van tabellen

Lijst van algoritmen

Bijlage A

Source Code

Source code HIER