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.

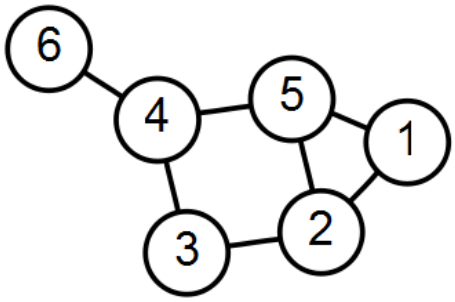
In hoofdstuk 2 zal het probleem van het vinden van een maximum flow en het gebruikte algoritme beschreven worden. Hoofdstuk 3 beschrijft hoe de software gebruikt kan worden met behulp van de GUI. De hoofdstukken 4, 5 & 6 beschrijven de algoritmen die gebruikt worden om een pad te vinden door het netwerk. De conclusies van dit onderzoek staan in hoofdstuk 7.

# Hoofdstuk 2

# Maximum flow probleem

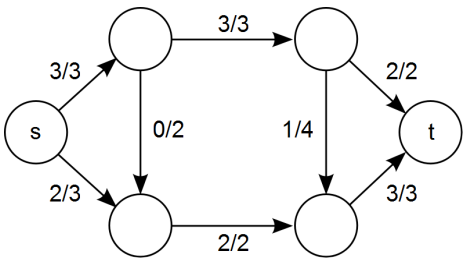
Om het probleem van het vinden van een maximum flow te kunnen begrijpen, volgt hier een korte introductie in de grafentheorie.

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 te zien 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 Ford-Fulkerson algoritme is te vinden in algoritme 1.

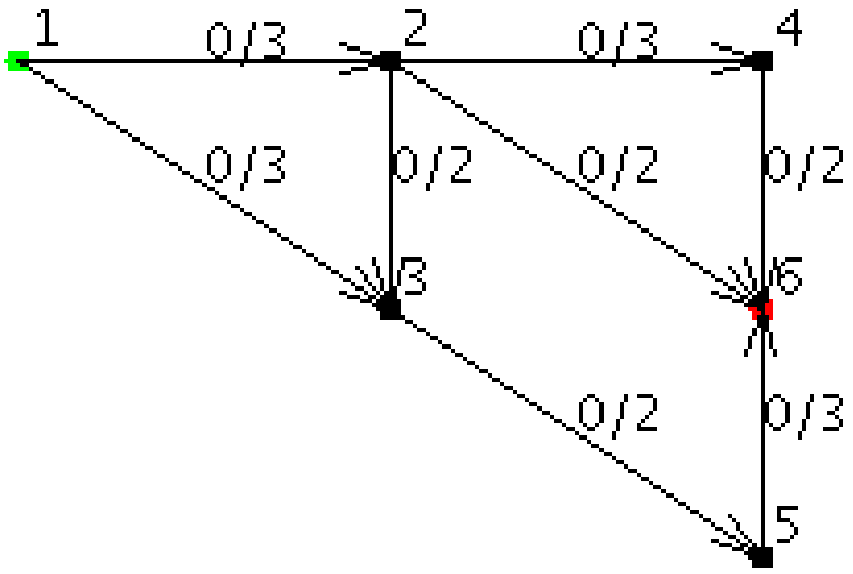
## 2.2 Analyse

Voor de analyse van de zoekmethodes worden de grafen uit de figuren 2.3 en 2.4 gebruikt.

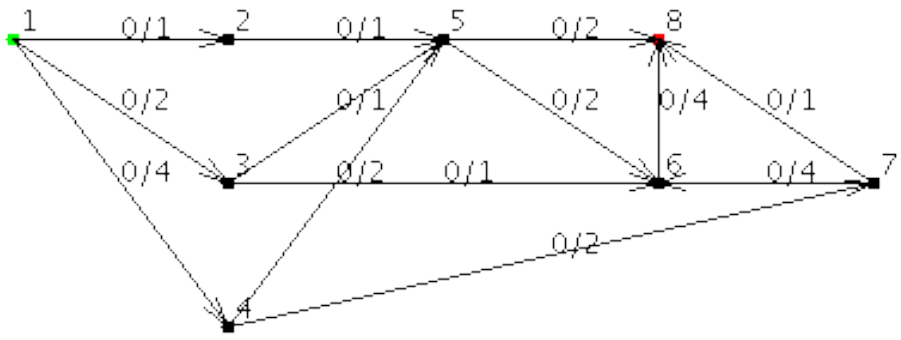
**Algoritme 1** Ford-Fulkerson Algorithm

**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
```




Figuur 2.3: Analyse graaf 1




Figuur 2.4: Analyse graaf 2

# Hoofdstuk 3

# Software

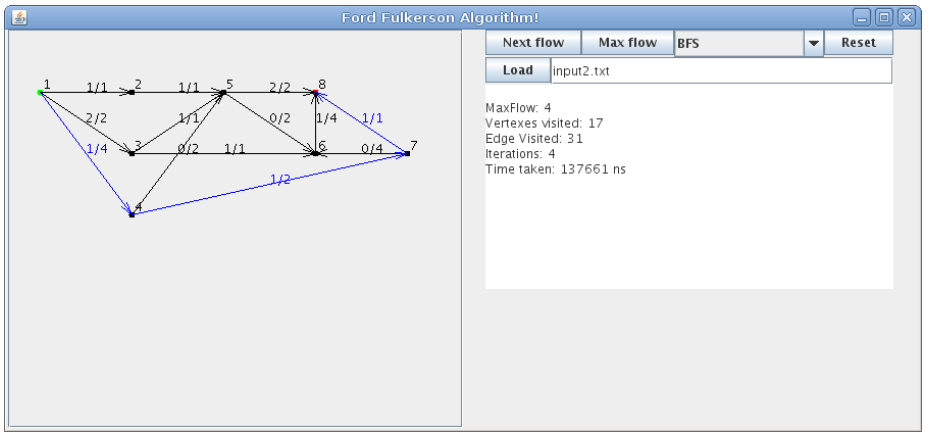
Om de methodes te visualiseren is een GUI ontwikkeld, deze zal in dit hoofdstuk beschreven worden. In figuur 3.1 is de GUI te zien. De linkerkant laat de graaf zien, en aan de rechterkant zijn controllers aanwezig om de software te bedienen. De start- en eindknoop hebben de kleur groen en rood, respectievelijk. De paden waarover een flow wordt gestuurd zijn blauw gekleurd.

De knop 'Next flow' zal één 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. 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 en het algoritme weergegeven wordt.

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 staan de getallen die de start- en eindknoop aangeven ($s$ en $t$). De overige regels bevatten de kanten van de graaf bestaande uit: de beginknoop, de eindknoop en de capaciteit van de kant.




Figuur 3.1: Screenshot van de GUI

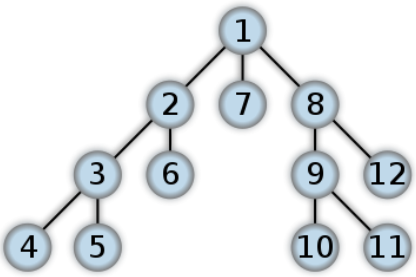
| | | |
|---|---|---|
| 6 | 8 | |
| 1 | 6 | |
| 1 | 2 | 3 |
| 1 | 3 | 3 |
| 2 | 3 | 2 |
| 2 | 4 | 3 |
| 2 | 6 | 2 |
| 3 | 5 | 2 |
| 5 | 6 | 3 |
| 4 | 6 | 2 |

Tabel 3.1: Inhoud van een graaf bestand (input1.txt)

# 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 $t$ terug naar $s$ te vinden. Bij elke aanroep zal $s$ gelabeld worden als *EXPLORED*. 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 gelabeld worden als *DISCOVERY*. 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.

## 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 | 17 |
| Kanten bezocht | 18 |
| Iteraties | 5 |
| Benodigde tijd | 70.261 ns |

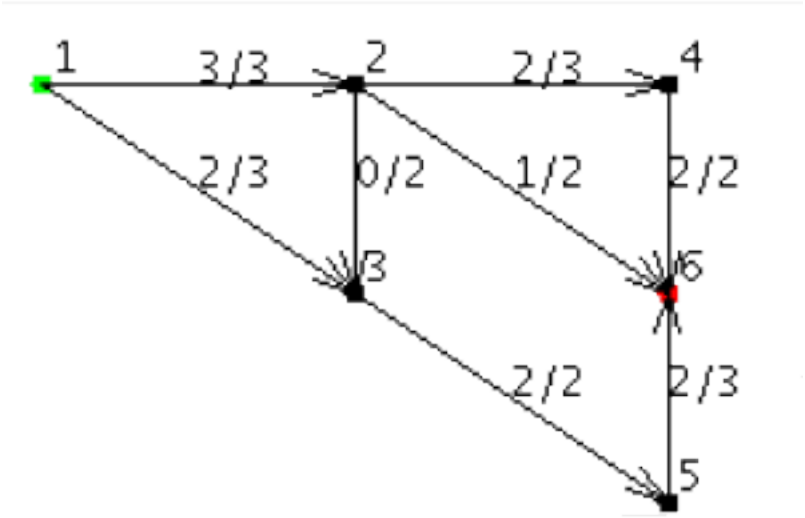
Tabel 4.1: Resultaten voor graaf in figuur 4.2

.

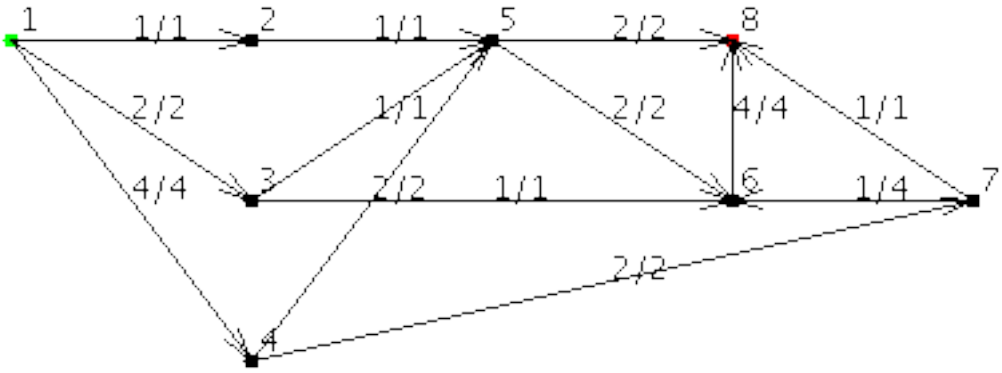
| | |
|---|---|
| Maximale flow | 5 |
| Knopen bezocht | 31 |
| Kanten bezocht | 42 |
| Iteraties | 7 |
| Benodigde tijd | 124.250 ns |

Tabel 4.2: Resultaten voor graaf in figuur 4.3

.




Figuur 4.2: De eerste graaf



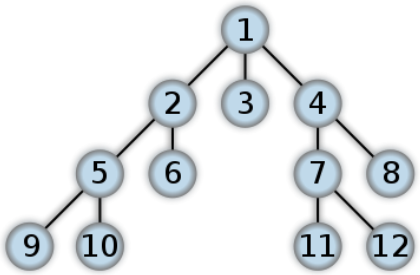

Figuur 4.3: De tweede graaf

# Hoofdstuk 5

# Breadth-first search

De tweede manier om een augmenting path te vinden in een graaf is de breadth-first search. Deze methode wordt ook gebruikt in het Edmonds-Karp algoritme, dit algoritme is een specifieke variant van het Ford-Fulkerson algoritme. Deze methode 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 ∧v.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}
```

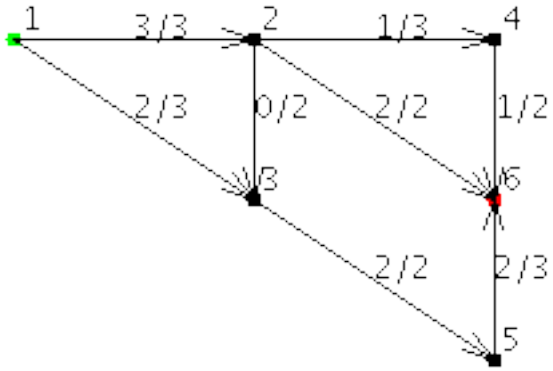
## 5.2 Analyse

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 bezocht | 15 |
| Iteraties | 4 |
| Benodigde tijd | 57.061 ns |

Tabel 5.1: Resultaten voor graaf in figuur 5.2

.

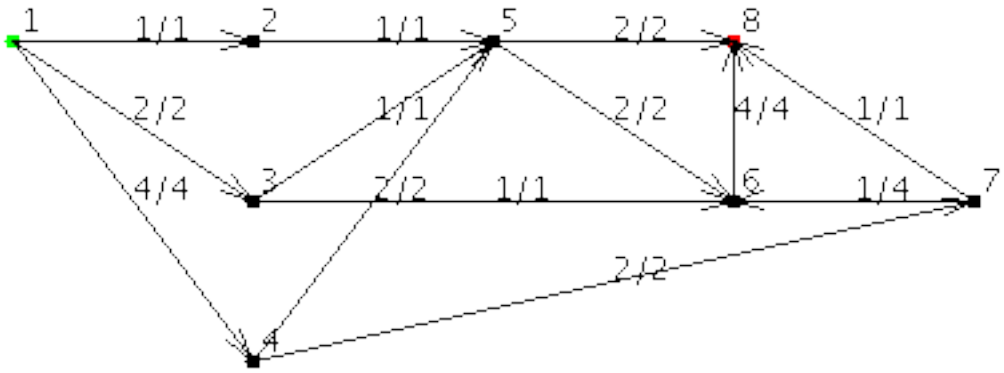



Figuur 5.2: Analyse van de eerste graaf

| | |
|---|---|
| Maximale flow | 7 |
| Knopen bezocht | 29 |
| Kanten bezocht | 48 |
| Iteraties | 7 |
| Benodigde tijd | 149.954 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 $\infty$. Al deze knopen worden toegevoegd 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. 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.

## 6.1 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.

## 6.2 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 bezocht | 36 |
| Iteraties | 4 |
| Benodigde tijd | 151.140 ns |

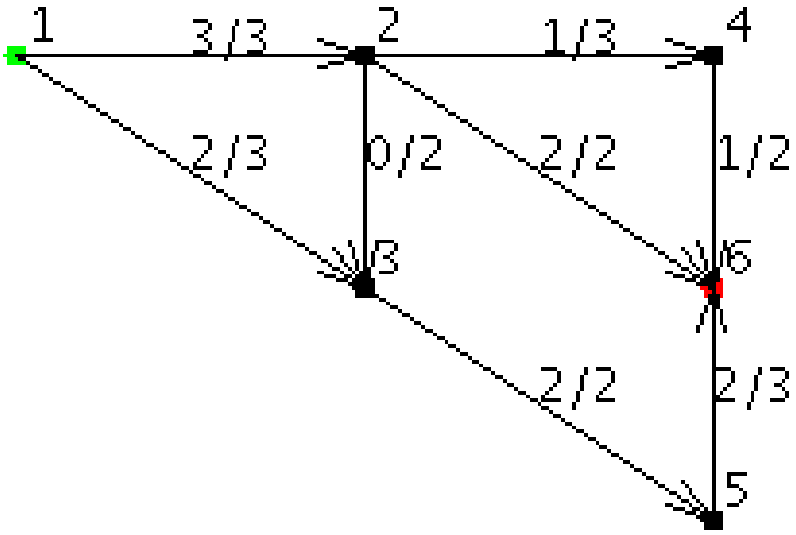
Tabel 6.1: Resultaten voor graaf in figuur 6.1

.

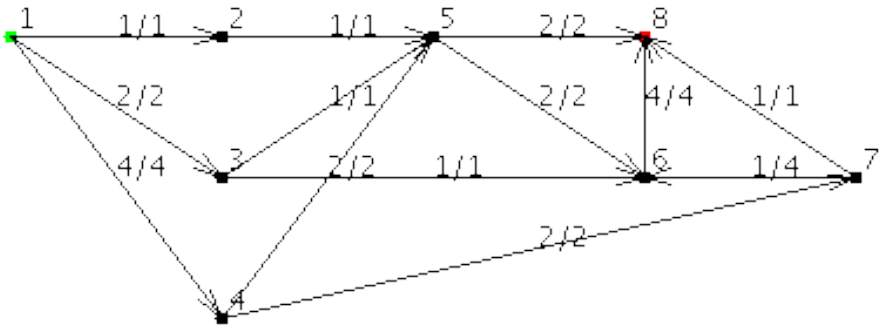
| | |
|---|---|
| Maximale flow | 7 |
| Knopen bezocht | 43 |
| Kanten bezocht | 123 |
| Iteraties | 6 |
| Benodigde tijd | 361.438 ns |

Tabel 6.2: Resultaten voor graaf in figuur 6.2

.




Figuur 6.1: De eerste graaf




Figuur 6.2: De tweede graaf

# Hoofdstuk 7

# Conclusie

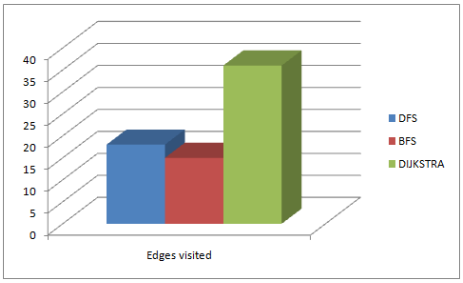
Aan de hand van de analyses uit de vorige hoofdstukken is te zien dat de BFS methode het beste werkt bij de geanalyseerde grafen.

| Graaf 1 | DFS | BFS | DIJKSTRA |
|---|---|---|---|
| Bezochte knopen | 17 | 11 | 18 |
| Bezochte kanten | 18 | 15 | 36 |
| Iteraties | 5 | 4 | 4 |
| Tijd | 70.261 ns | 57.061 ns | 151.140 ns. |

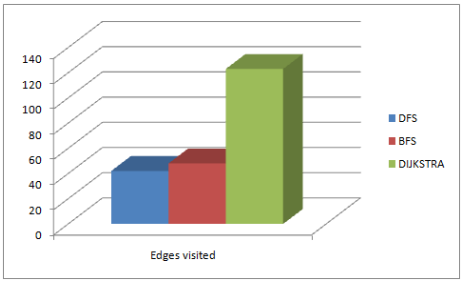
Tabel 7.1: Resultaten voor graaf 1

| Graaf 2 | DFS | BFS | DIJKSTRA |
|---|---|---|---|
| Bezochte knopen | 31 | 29 | 43 |
| Bezochte kanten | 42 | 48 | 123 |
| Iteraties | 7 | 7 | 6 |
| Tijd | 124.450 ns. | 149.954 ns. | 361.438 ns. |

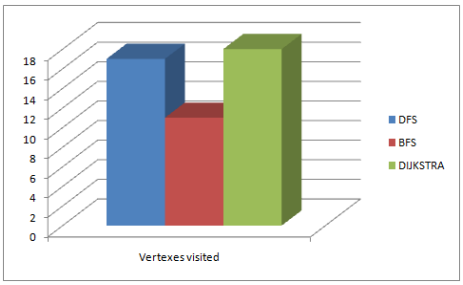
Tabel 7.2: Resultaten voor graaf 2



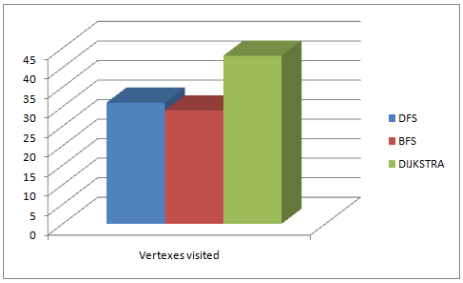

Figuur 7.1: Bezochte kanten van graaf 1



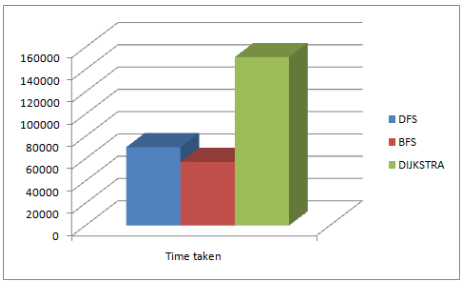

Figuur 7.2: Bezochte kanten van graaf 2



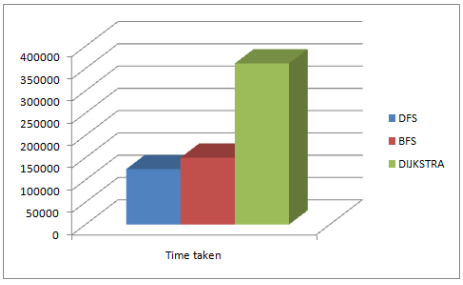

Figuur 7.3: Bezochte knopen van graaf 1




Figuur 7.4: Bezochte knopen van graaf 2




Figuur 7.5: Tijd voor graaf 1




Figuur 7.6: Tijd voor graaf 2

# Lijst van figuren

# Lijst van tabellen

# Lijst van algoritmen

# Bijlage A

# Source Code

# Bijlage B

# AugmentedPath

Listing B.1: AugmentedPath Source Code

```
public abstract class AugmentedPath {

        public enum Method {DFS, BFS, DIJKSTRA}

        protected static Profiler profiler = new Profiler();

        public static final AugmentedPath NULL = new NullPath();

        public abstract List<Edge> getAugmentedPath(Graph g, Vertex s, Vertex t

        protected static List<Edge> findAugmentedPath(HashMap<Vertex, Edge> par
                List<Edge> augmentedPath = new LinkedList<Edge>();
                Vertex iterator = t;
                Edge parent = parents.get(iterator);
                boolean sourceReached = false;
                while(parent != null){
                        augmentedPath.add(parent);
                        iterator = g.opposite(iterator, parent);
                        parent = parents.get(iterator);
                        if(iterator.equals(s)){
                                sourceReached = true;
                        }
                }
                if(sourceReached){
                        return augmentedPath;
                }
                return Collections.emptyList();
        }

        public static Profiler getProfiler() {
                return profiler;
        }
```

```
        private static final class NullPath extends AugmentedPath {
                @Override
                public List<Edge> getAugmentedPath(Graph g, Vertex s, Vertex t)
                        return Collections.emptyList();
                }
        }

}
```

# Bijlage C

# AugmentedPathBFS

Listing C.1: AugmentedPathBFS Source Code

```
ublic class AugmentedPathBFS extends AugmentedPath {

        public List<Edge> getAugmentedPath(Graph g, Vertex s, Vertex t) {
                List<Edge> path = new LinkedList<Edge>();
                Queue<Vertex> vertexQueue = new LinkedList<Vertex>();

                //Map <x,y>. Vertex y discovered by following x
                Map<Vertex, Edge> discoveryMap = new HashMap<Vertex, Edge>();

                s.status = VertexStatus.EXPLORED;
                vertexQueue.add(s);

                Vertex w;
                List<Edge> unionEdge;

                while (!vertexQueue.isEmpty()) {
                        w = vertexQueue.poll();
                        profiler.visitedVertex();
                        unionEdge = w.getAllEdges();
                        for (Edge e : unionEdge) {
                                if (e.status == EdgeStatus.UNEXPLORED
                                                && (w.getResidualCapacity(e) >
                                        profiler.visitedEdge();
                                        Vertex next = g.opposite(w, e);
                                        if (next.status == VertexStatus.UNEXPLO
                                                vertexQueue.add(next);
                                                discoveryMap.put(next, e);
                                                next.status = VertexStatus.EXPL
                                                e.status = EdgeStatus.DISCOVERY

                                                if (e.start.equals(w)) {
                                                        e.forward = true;
                                                } else {
```

```
                                        e.forward = false;
                                    }

                                    if (next == t) {
                                        //sink found. Trace bac
                                        boolean traceDone = fal
                                        Edge currentEdge;
                                        Vertex previousVertex =
                                        while(!traceDone) {
                                            currentEdge = d
                                            path.add(curren

                                            previousVertex

                                            if(previousVert
                                                traceDo
                                        }

                                        return path;
                                    }
                                } else {
                                    e.status = EdgeStatus.BACK;
                                }
                            }
                        }

                        unionEdge = null;
                    }

                    return path;
                }

            }
```

# Bijlage D

# AugmentedPathDFS

Listing D.1: AugmentedPathDFS Source Code

```
public class AugmentedPathDFS extends AugmentedPath {

        private boolean stop = false;

        public List<Edge> getAugmentedPath(Graph g, Vertex s, Vertex t){
                HashMap<Vertex, Edge> parents = getPathDFS(g, s, t, new HashMap
                return findAugmentedPath(parents, g, s, t);
        }

        public HashMap<Vertex, Edge> getPathDFS(Graph g, Vertex s, Vertex t, Ha
                s.status = VertexStatus.EXPLORED;
                profiler.visitedVertex();
                stop = false;
                for (Edge e : s.getAllEdges()) {
                        if (e.status == EdgeStatus.UNEXPLORED
                                        && s.getResidualCapacity(e) > 0) {
                                profiler.visitedEdge();
                                Vertex w = g.opposite(s, e);
                                if (w.status == VertexStatus.UNEXPLORED) {
                                        e.status = EdgeStatus.DISCOVERY;
                                        //profiler.visitedVertex();
                                        parents.put(w, e);
                                        if (e.start.equals(s)) {
                                                e.forward = true;
                                        } else {
                                                e.forward = false;
                                        }
                                        if(w.equals(t)){
                                                stop = true;
                                                return parents;
                                        }
                                        getPathDFS(g, w, t, parents);
                                } else {
```

```
                                        e.status = EdgeStatus.BACK;
                                }
                        }
                        if(stop){
                                return parents;
                        }
                }
                return parents;
        }
}
```

# Bijlage E

# AugmentedPathDijkstra

Listing E.1: AugmentedPathDijkstra Source Code

```
public class AugmentedPathDijkstra extends AugmentedPath {

        public List<Edge> getAugmentedPath(Graph g, Vertex s, Vertex t){
                HashMap<Vertex, Edge> parents = new HashMap<Vertex, Edge>();
                PriorityQueue<Vertex> Q = new PriorityQueue<Vertex>();
                for(Vertex v : g.vertexList){
                        if(v.equals(s)){
                                v.maxFlow = Integer.MAX_VALUE;
                        } else {
                                v.maxFlow = 0;
                        }
                        Q.add(v);
                        parents.put(v, null);
                }
                while(!Q.isEmpty()){
                        Vertex u = Q.remove();
                        profiler.visitedVertex();
                        if(u.equals(t)){
                                return findAugmentedPath(parents, g, s, t);
                        }
                        for(Edge e : u.getAllEdges()){
                                profiler.visitedEdge();
                                Vertex z = g.opposite(u, e);
                                int r = Math.min(u.getResidualCapacity(e), u.ma
                                if(r > z.maxFlow && !e.equals(parents.get(z))){
                                        z.maxFlow = r;
                                        parents.put(z, e);
                                        if (e.start.equals(u)) {
                                                e.forward = true;
                                        } else {
                                                e.forward = false;
                                        }
                                        if(Q.remove(z)){
```

```
                                                    Q.add(z);
                                                }
                                            }
                                        }
                                    }
                                    return Collections.emptyList();
                                }

                            }
```

# Bijlage F

# Controller

Listing F.1: Controller Source Code

```
public class Controller {

        private View view;
        private MaxFlowFordFulkerson mfff;

        public Controller(){
                view = new View();
                view.addNextFlowListener(nextFlowListener());
                view.addFindMaxFlowListener(findMaxFlowListener());
                view.addResetListener(resetListener());
                view.addLoadListener(loadFileListener());
        }

        private void loadGraphFile(String fileName){
                File file = new File(fileName);
                if(!file.exists()){
                        JOptionPane.showMessageDialog(new JFrame(),
                                        "File not found!",
                                        "Warning!",
                                        JOptionPane.WARNING_MESSAGE);
                        return;
                }
                Graph graph = GraphBuilder.buildGraph(file);
                mfff = new MaxFlowFordFulkerson(graph, graph.startPoint, graph.
                mfff.setGraphListener(updateViewListener());
                mfff.completeReset();
                view.loadGraph(graph);
        }

        private ActionListener loadFileListener(){
                return new ActionListener() {
                        @Override
                        public void actionPerformed(ActionEvent e) {
```

```
                        loadGraphFile(view.getFileName());
                }
        };
}

private ActionListener findMaxFlowListener(){
        return new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                        if(mfff == null){
                                JOptionPane.showMessageDialog(new JFram
                                        "No graph loaded yet!",
                                        "Warning!",
                                        JOptionPane.WARNING_MESSAGE
                                return;
                        }
                        mfff.findMaxFlow(view.getSelectedMethod());
                }
        };
}

private ActionListener nextFlowListener(){
        return new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                        if(mfff == null){
                                JOptionPane.showMessageDialog(new JFram
                                        "No graph loaded yet!",
                                        "Warning!",
                                        JOptionPane.WARNING_MESSAGE
                                return;
                        }
                        mfff.nextFlow(view.getSelectedMethod());
                }
        };
}

private ActionListener resetListener(){
        return new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                        if(mfff == null){
                                JOptionPane.showMessageDialog(new JFram
                                        "No graph loaded yet!",
                                        "Warning!",
                                        JOptionPane.WARNING_MESSAGE
                                return;
                        }
                        mfff.completeReset();
                }
```

```
                };
        }

        private ActionListener updateViewListener(){
                return new ActionListener() {
                        @Override
                        public void actionPerformed(ActionEvent e) {
                                view.updateGraph();
                                view.setLabelText(e.getActionCommand());
                        }
                };
        }

        public static void main(String args[]){
                new Controller();
        }
}
```

# Bijlage G

# Controllers

Listing G.1: Controllers Source Code

```
public class Controllers extends JPanel {

        private static final long serialVersionUID = 2396396168572312934L;
        private JButton nextFlowButton = new JButton("Next flow");
        private JButton findMaxFlowButton = new JButton("Max flow");
        private JComboBox methodBox;
        private JButton resetButton = new JButton("Reset");
        private JTextArea label = new JTextArea("");
        private JButton loadButton = new JButton("Load");
        private JTextField inputFileField = new JTextField("input1.txt");

        private Box topBox = Box.createHorizontalBox();
        private Box middleBox = Box.createHorizontalBox();
        private Box bottomBox = Box.createHorizontalBox();

        public Controllers(){
                setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));

                Dimension d = new Dimension(400, 27);
                topBox.setMaximumSize(d);
                middleBox.setMaximumSize(d);

                Dimension d2 = new Dimension(400, 200);
                bottomBox.setMaximumSize(d2);
                add(topBox);
                add(middleBox);
                add(bottomBox);

                topBox.add(nextFlowButton);
                methodBox = new JComboBox();
                methodBox.addItem(Method.DFS);
                methodBox.addItem(Method.BFS);
                methodBox.addItem(Method.DIJKSTRA);
```

```
                topBox.add(findMaxFlowButton);
                topBox.add(methodBox);
                topBox.add(resetButton);

                middleBox.add(loadButton);

                middleBox.add(inputFileField);

                label.setEditable(false);
                bottomBox.add(label);

        }

        public void addFindMaxFlowListener(ActionListener al){
                findMaxFlowButton.addActionListener(al);
        }

        public void addResetListener(ActionListener al){
                resetButton.addActionListener(al);
        }

        public String getFileName(){
                return inputFileField.getText();
        }

        public void addLoadListener(ActionListener al){
                loadButton.addActionListener(al);
        }

        public void addNextFlowButtonListener(ActionListener al){
                nextFlowButton.addActionListener(al);
        }

        public Method getSelectedMethod(){
                return (Method) methodBox.getSelectedItem();
        }

        public void setLabelText(String text){
                label.setText(text);
        }
}
```

# Bijlage H

# Edge

Listing H.1: Edge Source Code

```
public class Edge {

        public final int capacity;
        public int flow;
        public Vertex start;
        public Vertex end;
        public Color color = Color.black;

        //To be used when on the search for an augmenting path
        //Not to be used when creating a graph
        public boolean forward = true;

        public enum EdgeStatus {UNEXPLORED, BACK, DISCOVERY};
        public EdgeStatus status = EdgeStatus.UNEXPLORED;

        public Edge(int capacity) {
                this.capacity = capacity;
        }
}
```

# Bijlage I

# Graph

Listing I.1: Graph Source Code

```
public class Graph {

        public List<Edge> edgeList;
        public List<Vertex> vertexList;

        public Vertex startPoint;
        public Vertex endPoint;

        public void setStartPoint(Vertex v) {
                edgeList = new LinkedList<Edge>();
                vertexList = new LinkedList<Vertex>();
                this.startPoint = v;
                insert(v);
        }

        public void setEndPoint(Vertex v) {
                this.endPoint = v;
                insert(v);
        }

        public Vertex opposite(Vertex v, Edge e){
                if(e.start.equals(v)){
                        return e.end;
                }
                return e.start;
        }

        //insert(Vertex)
        public void insert(Vertex v) {
                this.vertexList.add(v);
        }

        //insert(Edge e, Vertex start, Vertex eind)
```

```
public void insert(Edge e, Vertex start, Vertex end) {
        edgeList.add(e);
        e.start = start;
        e.end = end;
        start.addOutgoing(e);
        end.addIncoming(e);
}

//areAdjacent(Vertex a, Vertex b)
public boolean areAdjacent(Vertex a, Vertex b) {
        for(Edge e : a.outgoingEdges) {
                if(e.end == b)
                        return true;
        }

        return false;
}

public Edge getConnectingEdge(Vertex a, Vertex b) {
        for(Edge e : a.outgoingEdges) {
                if(e.end == b)
                        return e;
        }

        return null;
}

/**
 * Finds a Vertex with the specified id within the vertexList.
 *
 * Access time O(n). Where n is the number of vertexes
 *
 * @param id
 * @return The vertex, or null.
 */
public Vertex findVertex(int id) {
        for(Vertex v : vertexList)
                if(v.id == id)
                        return v;

        return null;
}

public void printGraph(){
        for(Vertex v : this.vertexList){
                System.out.println("Vertex " + v.id);
                for(Edge e : v.outgoingEdges){
                        System.out.println("ForwardEdge to " + e.end.id
                }
                for(Edge e : v.incomingEdges){
```

```
                    System.out.println("BackwardEdge to " + e.start
                }
            }
        }
```

# Bijlage J

# GraphBuiler

Listing J.1: GraphBuiler Source Code

```
public class GraphBuilder {

        public static Graph buildGraph(File f) {
                BufferedReader reader = null;
                Graph g = new Graph();

                try {
                        reader = new BufferedReader(new FileReader(f));

                        String line = reader.readLine();

                        String[] split = line.split("␣");
                        int numVertexes = Integer.parseInt(split[0]);
                        int numEdges = Integer.parseInt(split[1]);

                        line = reader.readLine();
                        split = line.split("␣");

                        int start = Integer.parseInt(split[0]);
                        int end = Integer.parseInt(split[1]);


                        Vertex a = new Vertex(start);
                        Vertex b = new Vertex(end);

                        g.setStartPoint(a);
                        g.setEndPoint(b);

                        for(int m = 0; m < numEdges; ++m) {
                                line = reader.readLine();

                                if(line == null) {
                                        throw new IOException("Unexpected␣end␣o
```

```
                        }

                        split = line.split("␣");

                        if(split.length != 3) {
                                throw new IOException("Malformed␣input:
                        }

                        int s = Integer.parseInt(split[0]); //Start ver
                        int e = Integer.parseInt(split[1]); //End verte
                        int c = Integer.parseInt(split[2]); //Capacity

                        a = g.findVertex(s);
                        b = g.findVertex(e);

                        if(a == null) {
                                a = new Vertex(s);
                                g.insert(a);
                        }

                        if(b == null) {
                                b = new Vertex(e);
                                g.insert(b);
                        }

                        g.insert(new Edge(c), a, b);
                }

                if(g.vertexList.size() != numVertexes)
                        throw new IOException(String.format("Not␣all␣ve


                return g;
        } catch(FileNotFoundException fnfe) {
                System.err.println("File␣not␣found:␣" + f);
        } catch(IOException ioe) {
                ioe.printStackTrace();
        } finally {
                try {
                        reader.close();
                }catch(Exception e){}
        }

        return null; //Failure, return null! BAM
    }
}
```

# Bijlage K

# GraphView

Listing K.1: GraphView Source Code

```
public class GraphView extends JPanel {

        private static final long serialVersionUID = -8380122751232428458L;
        private Graph graph;
        private HashMap<Vertex, Point> vertexPoints = new HashMap<Vertex, Point
        private HashMap<Integer, Integer> xymap = new HashMap<Integer, Integer>
        private List<Vertex> drawn = new LinkedList<Vertex>();

        private int scale = 3;

        public GraphView(){

        }

        public void setGraph(Graph g){
                this.graph = g;
                updateGraph();
        }

        public void paint(Graphics g){
                Graphics2D g2 = (Graphics2D)g;
                g2.clearRect(0, 0, this.getWidth(), this.getHeight());
                if(graph == null){
                        return;
                }
                vertexPoints.clear();
                drawn.clear();
                xymap.clear();
                draw(graph.startPoint, new Point(10, 20), g2);
        }

        private void draw(Vertex v, Point p, Graphics2D g){
                if(!vertexPoints.containsKey(v)){
```

```
                vertexPoints.put(v, p);
            }
            p = vertexPoints.get(v);
            drawVertex(v, p, g);
            drawn.add(v);

            if(!xymap.containsKey(p.x)){
                xymap.put(p.x, p.y);
            }
            int y = xymap.get(p.x);

            for(Edge e : v.getAllEdges()){
                Vertex opposite = graph.opposite(v, e);
                Point p1 = new Point();
                Point p2 = new Point();
                if(vertexPoints.containsKey(opposite)){
                    if(e.start.equals(v)){
                        p1 = p;
                        p2 = vertexPoints.get(opposite);
                    } else {
                        p1 = vertexPoints.get(opposite);
                        p2 = p;
                    }
                } else {
                    p2 = new Point(p.x + 30, y);
                    drawVertex(opposite, p2, g);
                    vertexPoints.put(opposite, p2);
                    if(e.start.equals(v)){
                        p1 = p;
                    } else {
                        p1 = p2;
                        p2 = p;
                    }
                    y = y + 20;
                }
                drawEdge(e, p1, p2, g);
            }

            xymap.put(p.x, y);

            for(Edge e : v.getAllEdges()){
                Vertex opposite = graph.opposite(v, e);
                if(!drawn.contains(opposite)){
                    Point p2 = vertexPoints.get(opposite);
                    draw(opposite, p2, g);
                }
            }
        }

        private void drawVertex(Vertex v, Point p, Graphics2D g){
```

```
            if(v.equals(graph.startPoint)){
                    g.setColor(Color.GREEN);
            } else if(v.equals(graph.endPoint)) {
                    g.setColor(Color.red);
            }
            p = new Point(p.x * scale, p.y * scale);
            g.fillOval(p.x - (1 * scale), p.y - (1 * scale), 2 * scale, 2 *
            g.setColor(Color.black);
            g.drawString(v.id + "", p.x + scale, p.y - scale);
            updateSize(p, g);
    }

    private void drawEdge(Edge e, Point p1, Point p2, Graphics2D g){
            p1 = new Point(p1.x * scale, p1.y * scale);
            p2 = new Point(p2.x * scale, p2.y * scale);

            g.setColor(e.color);
            g.drawLine(p1.x, p1.y, p2.x, p2.y);

            double arrowLength = 4 * scale;
            double arrowAngle = 15;
            double angle;
            if(p1.y == p2.y){
                    if(p1.x < p2.x){
                            angle = 0;
                    } else {
                            angle = 180;
                    }
            } else if(p1.x == p2.x){
                    if(p1.y < p2.y){
                            angle = 90;
                    } else {
                            angle = 270;
                    }
            } else {
                    double xDistance = Math.abs(p2.x-p1.x);
                    double yDistance = Math.abs(p2.y-p1.y);
                    double xyDistance = Math.sqrt(xDistance*xDistance + yD
                    angle = Math.toDegrees(Math.acos(xDistance/xyDistance))
                    if(p2.y < p1.y){
                            angle = 360 - angle;
                    }
                    if(p2.x < p1.x){
                            angle = 180 - angle;
                    }
            }
            double y1 = Math.sin(Math.toRadians((angle + arrowAngle))) * ar
            double x1 = Math.cos(Math.toRadians((angle + arrowAngle))) * ar

            double y2 = Math.sin(Math.toRadians((-angle + arrowAngle))) * a
```

```
                double x2 = Math.cos(Math.toRadians((-angle + arrowAngle))) * a

                g.drawLine(p2.x, p2.y, (int)(p2.x - x1), (int)(p2.y - y1));
                g.drawLine(p2.x, p2.y, (int)(p2.x - x2), (int)(p2.y + y2));

                g.drawString(e.flow + "/" + e.capacity, (p1.x + p2.x) / 2, (p1.
                g.setColor(Color.black);
        }

        private void updateSize(Point p, Graphics g){
                Dimension d = new Dimension(getPreferredSize().width, getPrefer
                if(p.x > d.width){
                        d.width = p.x + 30;
                }
                if(p.y > d.height){
                        d.height = p.y + 30;
                }
                setPreferredSize(d);
                setMaximumSize(d);
                setMinimumSize(d);
                setSize(d);
        }

        public void updateGraph(){
                this.repaint();
        }
}
```

# Bijlage L

# MaxFlowFordFulkerson

Listing L.1: MaxFlowFordFulkerson Source Code

```
public class MaxFlowFordFulkerson {

        private Graph g;
        private Vertex s, t;

        private ActionListener listener;

        //MaxFlowFordFulkerson algorithm described on page 390
        public MaxFlowFordFulkerson(Graph g, Vertex s, Vertex t){
                this.g = g;
                this.s = s;
                this.t = t;
                for(Edge e : g.edgeList){
                        e.flow = 0;
                }
        }

        public void findMaxFlow(Method m){
                while(!AugmentedPath.getProfiler().getMaxFlowFound()){
                        nextFlow(m);
                }
        }

        public void nextFlow(Method m){
                //Reset all directions in graph
                resetGraphStatus();
                AugmentedPath.getProfiler().incIterations();
                AugmentedPath.getProfiler().startStopwatch();
                List<Edge> augmentedPath = getAugmentedPath(m);
                if(augmentedPath.size() > 0){
                        int resCap = getResidualCapacity(augmentedPath);
                        pushResCap(augmentedPath, resCap);
                } else {
```

```
                AugmentedPath.getProfiler().setMaxFlowFound(true);
        }
        AugmentedPath.getProfiler().pauseStopwatch();
        graphChanged(); //Update View
}

public void completeReset(){
        for(Edge e : g.edgeList){
                e.status = EdgeStatus.UNEXPLORED;
                e.color = Color.black;
                e.flow = 0;
        }
        for(Vertex v : g.vertexList){
                v.status = VertexStatus.UNEXPLORED;
                v.maxFlow = 0;
        }
        AugmentedPath.getProfiler().reset();
        graphChanged(); //Update view
}

public void resetGraphStatus(){
        for(Edge e : g.edgeList){
                e.status = EdgeStatus.UNEXPLORED;
                e.color = Color.black;
        }
        for(Vertex v : g.vertexList){
                v.status = VertexStatus.UNEXPLORED;
        }
}

private List<Edge> getAugmentedPath(Method m){
        AugmentedPath ap;
        switch (m) {
        case DFS:
                ap = new AugmentedPathDFS();
                break;
        case BFS:
                ap = new AugmentedPathBFS();
                break;
        case DIJKSTRA:
                ap = new AugmentedPathDijkstra();
                break;
        default:
                ap = AugmentedPath.NULL;
                break;
        }
        return ap.getAugmentedPath(g, s, t);
}

//Computes the residual capacity of the augmenting path
```

```
        private int getResidualCapacity(List<Edge> augmentingPath){
                int maxFlow = Integer.MAX_VALUE;
                for(Edge e : augmentingPath){
                        int resCap;
                        if(e.forward){
                                resCap = e.capacity - e.flow;
                        } else {
                                resCap = e.flow;
                        }
                        if(resCap < maxFlow){
                                maxFlow = resCap;
                        }
                }
                return maxFlow;
        }

        //Pushes the residual capacity along the augmenting path
        private void pushResCap(List<Edge> augmentedPath, int resCap){
                for(Edge e : augmentedPath){
                        e.color = Color.blue;
                        if(e.forward){
                                e.flow = e.flow + resCap;
                        } else {
                                e.flow = e.flow - resCap;
                        }
                }
        }

        private void graphChanged(){
                AugmentedPath.getProfiler().update(g);
                ActionEvent ae = new ActionEvent(g.startPoint, g.startPoint.id,
                listener.actionPerformed(ae);
        }

        public void setGraphListener(ActionListener listener){
                this.listener = listener;
        }

}
```

# Bijlage M

# Profiler

Listing M.1: Profiler Source Code

```
public class Profiler {

        private int maxFlow;
        private boolean maxFlowFound;
        private int visitedVertexes;
        private int visitedEdges;
        private long totalTime;
        private int iterations;

        private long currentTime;

        public void reset(){
                maxFlow = 0;
                maxFlowFound = false;
                visitedVertexes = 0;
                visitedEdges = 0;
                totalTime = 0;
                currentTime = 0;
                iterations = 0;
        }

        public void startStopwatch(){
                currentTime = System.nanoTime();
        }

        public void pauseStopwatch(){
                if(!maxFlowFound){
                        totalTime += System.nanoTime() - currentTime;
                }
        }

        public void update(Graph g){
                maxFlow = 0;
```

```
                for(Edge e : g.startPoint.outgoingEdges){
                        maxFlow += e.flow;
                }
                for(Edge e : g.startPoint.incomingEdges){
                        maxFlow += e.capacity - e.flow;
                }
        }

        public void visitedVertex(){
                if(!maxFlowFound){
                        visitedVertexes++;
                }
        }

        public void visitedEdge(){
                if(!maxFlowFound){
                        visitedEdges++;
                }
        }

        public void incIterations(){
                if(!maxFlowFound){
                        iterations++;
                }
        }

        public void setMaxFlowFound(boolean value){
                maxFlowFound = value;
        }

        public boolean getMaxFlowFound(){
                return maxFlowFound;
        }

        public String getStatus(){
                String message = "";
                if(maxFlowFound){
                        message += "Done!";
                }

                message += "␣\nMaxFlow:␣" + maxFlow;
                message += "␣\nVertexes␣visited:␣" + visitedVertexes;
                message += "␣\nEdge␣Visited:␣" + visitedEdges;
                message += "␣\nIterations:␣" + iterations;
                message += "␣\nTime␣taken:␣" + totalTime + "␣ns";

                return message;
        }
}
```

# Bijlage N

# Vertex

Listing N.1: Vertex Source Code

```
public class Vertex implements Comparable<Vertex> {

        public int id;
        public List<Edge> incomingEdges;
        public List<Edge> outgoingEdges;

        //Only used for Dijkstra's algorithm
        public Integer maxFlow = 0;

        //Only to be used when searching for augmented path
        public enum VertexStatus {UNEXPLORED, EXPLORED};
        public VertexStatus status = VertexStatus.UNEXPLORED;

        public Vertex(int id) {
                this.id = id;
                incomingEdges = new LinkedList<Edge>();
                outgoingEdges = new LinkedList<Edge>();
        }

        public void addIncoming(Edge e) {
                incomingEdges.add(e);
        }

        public void addOutgoing(Edge e) {
                outgoingEdges.add(e);
        }

        public int getResidualCapacity(Edge e){
                if (e.start.equals(this)) {
                        return e.capacity - e.flow;
                } else {
                        return e.flow;
                }
```

```
        }

        public List<Edge> getAllEdges(){
                List<Edge> unionEdge = new LinkedList<Edge>();
                unionEdge.addAll(this.incomingEdges);
                unionEdge.addAll(this.outgoingEdges);
                return unionEdge;
        }

        @Override
        public int compareTo(Vertex v) {
                return -this.maxFlow.compareTo(v.maxFlow);
        }
}
```

# Bijlage O

# View

Listing O.1: View Source Code

```
public class View extends JFrame {

        private static final long serialVersionUID = -5640918753333009588L;

        private GraphView graphView;
        private Controllers controllers;

        public View(){
                super("Ford Fulkerson Algorithm!");
                setLayout(new GridLayout());

                graphView = new GraphView();
                controllers = new Controllers();

                JScrollPane scrollPane = new JScrollPane(graphView);
                //scrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTA
                //scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SC
                Dimension d = new Dimension(400, 400);
                scrollPane.setPreferredSize(d);
                scrollPane.setMaximumSize(d);
                scrollPane.setMinimumSize(d);
                add(scrollPane);

                d.width = 100;
                controllers.setPreferredSize(d);
                controllers.setMaximumSize(d);
                controllers.setMinimumSize(d);
                add(controllers);

                setSize(700, 400);
                setDefaultCloseOperation(EXIT_ON_CLOSE);
                setVisible(true);
        }
```

```
        public void loadGraph(Graph g){
                graphView.setGraph(g);
        }

        public void setLabelText(String text){
                controllers.setLabelText(text);
        }

        public void addResetListener(ActionListener al){
                controllers.addResetListener(al);
        }

        public void addNextFlowListener(ActionListener al){
                controllers.addNextFlowButtonListener(al);
        }

        public void addFindMaxFlowListener(ActionListener al){
                controllers.addFindMaxFlowListener(al);
        }

        public Method getSelectedMethod(){
                return controllers.getSelectedMethod();
        }

        public void addLoadListener(ActionListener al){
                controllers.addLoadListener(al);
        }

        public String getFileName(){
                return controllers.getFileName();
        }

        public void updateGraph(){
                graphView.updateGraph();
        }
}
```