

# ADVANCED ALGORITHMS AND DATASTRUCTURES

## EXAM NOTES

*Author:*

Jenny-Margrethe VEJ – 250986 – rwj935

Block 4, April-June, 2014

# Contents

<b>1</b>	<b>Maximum Flow</b>	<b>2</b>
1.1	Flow Networks . . . . .	2
1.1.1	Flow Networks and Flow . . . . .	2
1.1.2	An example of a flow . . . . .	2
1.1.3	Modelling problems with antiparallel edges . . . . .	2
1.1.4	Networks with multiple sources and sinks . . . . .	3
1.2	The Ford-Fulkerson Method . . . . .	3
1.2.1	Residual Network . . . . .	3
1.2.2	Augmenting paths . . . . .	6
1.2.3	Cuts of Flow Networks . . . . .	6
1.2.4	The basic Ford-Fulkerson Algorithm . . . . .	6
1.2.5	Analysis of Ford-Fulkerson . . . . .	6
1.2.6	The Edmond-Karp Algorithm . . . . .	6
1.3	Maximum bipartite matching . . . . .	6
1.3.1	The Maximum-bipartite-matching problem . . . . .	6
1.3.2	Finding a maximum bipartite matching . . . . .	6
<b>2</b>	<b>Fibonacci Heaps</b>	<b>7</b>
<b>3</b>	<b>NP-Completeness</b>	<b>8</b>
<b>4</b>	<b>Randomised Algorithms</b>	<b>9</b>
<b>5</b>	<b>Hashing</b>	<b>10</b>
5.1	Direct-address tables . . . . .	10
5.2	Hash tables . . . . .	10
5.3	Hash functions . . . . .	11
5.3.1	The division method . . . . .	12
5.3.2	The multiplication method . . . . .	12
5.3.3	Universal hashing . . . . .	12
5.4	Open addressing . . . . .	12
5.5	Perfect hashing . . . . .	12
<b>6</b>	<b>Exact Exponential Algorithms</b>	<b>13</b>
<b>7</b>	<b>Approximation Algorithms</b>	<b>14</b>
<b>8</b>	<b>Computational Geometry</b>	<b>15</b>
<b>9</b>	<b>Linear Programming and Optimisation</b>	<b>16</b>

# 1 Maximum Flow

Wuuuh, Max Flow!! You Rock! Kick some ass!

## 1.1 Flow Networks

### 1.1.1 Flow Networks and Flow

Let  $G = (V, E)$  be a flow network with a capacity function  $c$ . Let  $s$  be the source of the network, and let  $t$  be the sink. A flow in  $G$  is a real-valued function  $f : V \times V \rightarrow \mathbb{R}$  that satisfies the following two properties:

**Capacity Constraints:** For all  $u, v \in V$ , we require  $0 \leq f(u, v)^1 \leq c(u, v)^2$

**Flow Conservation**<sup>3</sup>: For all  $u \in V - \{s, t\}$ , we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) \quad (1)$$

When  $(u, v) \notin E$ , there can be no flow from  $u$  to  $v$ , and  $f(u, v) = 0$ . The value  $|f|$  of a flow  $f$  is defined as

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s), \quad (2)$$

Typically, a flow network will not have any edges into the source, and the flow into the source, given by the summation  $\sum_{v \in V} f(v, s)$ , will be 0.

### 1.1.2 An example of a flow

For example the transporting example from the book [CLRS [2009], p. 710]. A firm wants to transport goods from one place to another, using a third part driver.

Guessing this particular section is not thaaaaat important for the exam. ☺

### 1.1.3 Modelling problems with antiparallel edges

Antiparallel edges is 2 edges going to/from  $(v, u)$  - so 2 edges between 2 vertices but with opposite directions. To come around that, we transform our network into an equivalent one containing no antiparallel edges (adding an extra vertex for that, so we can split one of the edges). The resulting network is equivalent to the original one, due to the fact,

---

<sup>1</sup>We call the nonnegative quantity  $f(u, v)$  the flow from vertex  $u$  to vertex  $v$

<sup>2</sup>Capacity Function  $c$

<sup>3</sup>Vertices are conduit junctions, and other than the source and sink, material flows through the vertices without collecting in them. In other words, the rate at which material enters a vertex must equal the rate at which it leaves the vertex. This property is called Flow Conservation

that you do not add or subtract anything from the capacity. It is the same:



#### 1.1.4 Networks with multiple sources and sinks

A maximum flow problem may have several sources and sinks, rather than just one of each. To fix that, we just add a supersource and a supersink with infinity capacity from  $s$  to each of the multiple sources (and to  $t$ ).

## 1.2 The Ford-Fulkerson Method

---

```

1: FORD-FULKERSON-METHOD( $G, s, t$ )
2: initialise flow  $f$  to 0
3: while there exists an augmenting path  $p$  in the residual network  $G_f$  do
4:   augment flow  $f$  along  $p$ 
5: end while
6: return  $f$ 

```

---

### 1.2.1 Residual Network

Suppose that we have a flow network  $G = (V, E)$  with source  $s$  and sink  $t$ . Let  $f$  be a flow in  $G$ , and consider a pair of vertices  $u, v \in V$ . We define the residual capacity  $c_f(u, v)$  by

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E, \\ f(v, u) & \text{if } (v, u) \in E, \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

As an example of equation (3), if  $c(u, v) = 16$  and  $f(u, v) = 11$ , then we can increase  $f(u, v)$  by up to  $c_f(u, v) = 5$  units before we exceed the capacity constraint on edge  $(u, v)$ . We also wish to allow an algorithm to return up to 11 units of flow from  $v$  to  $u$ , and hence  $c_f(v, u) = 11$ .

Given a flow network  $G = (V, E)$  and a flow  $f$ , the residual network of  $G$  induced by  $f$  is  $G_f = (V, E_f)$ , where

$$E_f = \{(u, v) \in V \cdot V : c_f(u, v) > 0\} \quad (4)$$

Example:

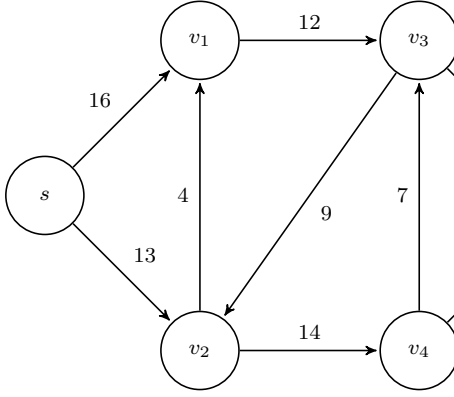


Figure 1: Flow Network  $G$

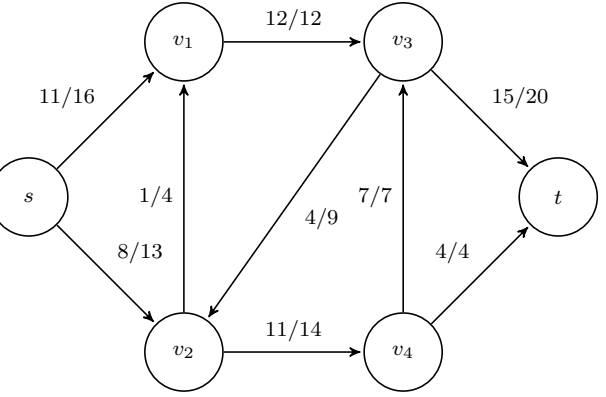


Figure 2: Flow  $f$  in  $G$

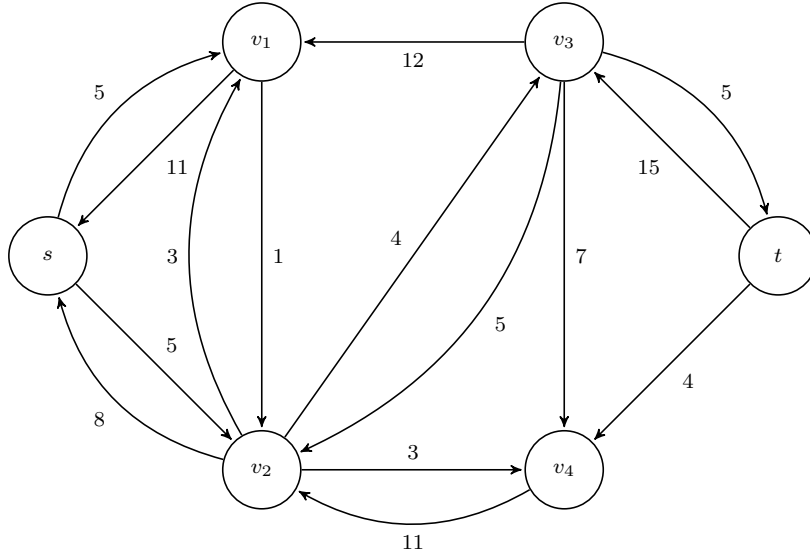


Figure 3: Residual Network  $G_f$

The residual network  $G_f$  is similar to a flow network with capacities given by  $c_f$ . It does not satisfy the definition of a flow network because it may contain both an edge  $(u, v)$  and its reversal  $(v, u)$ . Other than this difference, a residual network has the same properties as a flow network, and we can define a flow in the residual network as one that satisfies the definition of a flow, but with respect to capacities  $c_f$  in the network  $G_f$ .

If  $f$  is a flow in  $G$  and  $f'$  is a flow in the corresponding residual network  $G_f$ , we define  $f \uparrow f'$ , the **augmentation** of flow  $f$  by  $f'$ , to be the function from  $V \cdot V$  to  $\mathbb{R}$ , defined by

$$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & \text{if } (u, v) \in E, \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

**Lemma 26.1**

Let  $G = (V, E)$  be a flow network with source  $s$  and sink  $t$ , and let  $f$  be a flow in  $G$ . Let  $G_f$  be the residual network of  $G$  induced by  $f$ , and let  $f'$  be a flow in  $G_f$ . Then the function  $f \uparrow f'$  defined in the function (5) is a flow in  $G$  with value  $|f \uparrow f'| = |f| + |f'|$ .

**Prove for Lemma 26.1**

First we check that  $f \uparrow f'$  obeys capacity constraints for each edge in  $E$  and flow conservation for edges in  $V - \{s, t\}$ . Let us start with capacity constraints - we observe that if  $(u, v) \in E$  then  $c_f(v, u) = f(u, v)$ . Therefore, we have  $f'(v, u) \leq c_f(v, u) = f(u, v)$ , and hence

$$\begin{aligned} (f \uparrow f')(u, v) &= f(u, v) + f'(u, v) - f'(v, u) && \text{(by equation (5))} \\ &\geq f(u, v) + f'(u, v) - f(u, v) && \text{(because } f'(v, u) \leq f(u, v) \text{)} \\ &= f'(u, v) \\ &\geq 0 \end{aligned}$$

In addition we have,

$$\begin{aligned} (f \uparrow f')(u, v) &= f(u, v) + f'(u, v) - f'(v, u) && \text{(by equation (5))} \\ &\leq f(u, v) + f'(u, v) && \text{(because flows are nonnegative)} \\ &\leq f(u, v) + c_f(u, v) && \text{(capacity constraint)} \\ &= f(u, v) + c(u, v) - f(u, v) && \text{(definition of } c_f \text{)} \\ &= c(u, v) \end{aligned}$$

For flow conservation, because both  $f$  and  $f'$  obey flow conservation, we have that for all  $u \in V - \{s, t\}$ ,

$$\begin{aligned} \sum_{v \in V} (f \uparrow f')(u, v) &= \sum_{v \in V} (f(u, v) + f'(u, v) - f'(v, u)) \\ &= \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) - \sum_{v \in V} f'(v, u) \\ &= \sum_{v \in V} f(v, u) + \sum_{v \in V} f'(v, u) - \sum_{v \in V} f'(u, v) \\ &= \sum_{v \in V} (f(v, u) + f'(v, u) - f'(u, v)) \\ &= \sum_{v \in V} (f \uparrow f')(v, u) \end{aligned}$$

where the third line follows from the second by flow conservation.

Finally, we compute the value of  $f \uparrow f'$ . We remember that for any vertex  $v \in V$  we can have either  $(s, v)$  or  $(v, s)$  but never both, because we disallow antiparallel edges in  $G$ .

We define two sets:  $V_1 = \{v : (s, v) \in E\}$  and  $V_2 = \{v : (v, s) \in E\}$ . Clearly  $V_1 \cup V_2 \subseteq V$  and  $V_1 \cap V_2 = \emptyset$ . So let us compute

$$\begin{aligned} |f \uparrow f'| &= \sum_{v \in V} (f \uparrow f')(s, v) - \sum_{v \in V} (f \uparrow f')(v, s) \\ &= \sum_{v \in V_1} (f \uparrow f')(s, v) - \sum_{v \in V_2} (f \uparrow f')(v, s) \end{aligned}$$

where the second line follows because  $(f \uparrow f')(w, x)$  is 0 if  $(w, x) \notin E$ . We now apply the definition of  $f \uparrow f'$  to this, and then reorder and group terms to obtain

$$\begin{aligned} |f \uparrow f'| &= \sum_{v \in V_1} (f(s, v) - f'(v, s)) - \sum_{v \in V_2} (f(v, s) + f'(v, s)) \\ &= \sum_{v \in V_1} f(s, v) + \sum_{v \in V_1} f'(s, v) - \sum_{v \in V_1} f'(v, s) - \sum_{v \in V_2} f(v, s) - \sum_{v \in V_2} f'(v, s) - \sum_{v \in V_2} f'(s, v) \\ &= \sum_{v \in V_1} f(s, v) - \sum_{v \in V_2} f(v, s) + \sum_{v \in V_1} f'(s, v) + \sum_{v \in V_2} f'(s, v) - \sum_{v \in V_1} f'(v, s) - \sum_{v \in V_2} f'(v, s) \\ &= \sum_{v \in V_1} f(s, v) - \sum_{v \in V_2} f(v, s) + \sum_{v \in V_1 \cup V_2} f'(s, v) - \sum_{v \in V_1 \cup V_2} f'(v, s) \end{aligned}$$

In the last line we can extend all four summations to sum over  $V$ , since each additional term has value 0. Therefore we end up with

$$\begin{aligned} |f \uparrow f'| &= \sum_{v \in V} (f(s, v) - \sum_{v \in V} (f(v, s) + \sum_{v \in V} (f'(s, v) - \sum_{v \in V} (f'(v, s) \\ &= |f| + |f'| \end{aligned}$$

### 1.2.2 Augmenting paths

### 1.2.3 Cuts of Flow Networks

### 1.2.4 The basic Ford-Fulkerson Algorithm

### 1.2.5 Analysis of Ford-Fulkerson

### 1.2.6 The Edmond-Karp Algorithm

## 1.3 Maximum bipartite matching

### 1.3.1 The Maximum-bipartite-matching problem

### 1.3.2 Finding a maximum bipartite matching

## 2 Fibonacci Heaps



### 3 NP-Completeness

## 4 Randomised Algorithms

## 5 Hashing

OMG LOLZ'n SHIZZLES! You'r gonna nail this shit!

### 5.1 Direct-address tables

We have a scenario where we want to maintain a dynamic set of some sort. Each element has a key drawn from a universe  $U = \{0, 1, \dots, m - 1\}$  where  $m$  is not too large, and no two elements have the same key.

Representing it by a direct-address table, or array  $T[0 \dots m - 1]$ , each slot or position corresponds to a key in  $U$ . If there is an element  $x$  with the key  $k$ , then  $T[k]$  contains a pointer to  $x$  - otherwise  $T[k]$  is empty represented by  $NIL$

Illustration in the book on page 254

---

```
1: DIRECT-ADDRESS-SEARCH( $T, k$ )
2: return  $T[k]$ 
3:
4: DIRECT-ADDRESS-INSERT $T, x$ 
5: return  $T[key[x]] \leftarrow x$ 
6:
7: DIRECT-ADDRESS-DELETET $T, x$ 
8: return  $T[key[x]] \leftarrow NIL$ 
```

---

Downsides of direct addressing: If the universe  $U$  is large, storing a table  $T$  of size  $|U|$  may be impractical or even impossible, given the memory available on a typical computer. Furthermore, the set  $K$  of keys actually stored may be so small relative to  $U$  that most of the space allocated for  $T$  would be wasted.

### 5.2 Hash tables

To overcome the downsides of direct addressing, we can use hash tables. When  $K$  is much smaller than  $U$ , a hash table requires much less space than a direct-address table. It can also reduce storage requirements to  $\Theta(|K|)$ .

This bound is for average-case time whereas for direct addressing it holds for the worst-case time.

With direct addressing, an element with  $k$  is stored in slot  $k$ . With hashing, this element is stored in slot  $h(k)$ ; that is, we use a hash function  $h$  to compute the slot from the key  $k$ . There is one hitch: 2 keys may hash to the same slot - called a collision. See page 256

for illustrations.

A hash function  $h$  must be deterministic in that a given input  $k$  should always produce the same output  $h(k)$ .

### Collision resolution by chaining

In chaining we place all the elements that hash to the same slot into the same linked list. See figure on page 257.

- 
- 1: DIRECT-HASH-SEARCH( $T, x$ )
  - 2: **search** for an element with key  $k$  in list  $T(T, k)$
  - 3:
  - 4: DIRECT-HASH-INSERT $T, x$
  - 5: **insert**  $x$  at the head of the list  $T[h(x.key)]$
  - 6:
  - 7: DIRECT-HASH-DELETET $T, x$
  - 8: **delete**  $x$  from the list  $T[h(x.key)]$
- 

### Analysis of hashing with chaining

Given a hash table  $T$  with  $m$  slots that stores  $n$  elements, we define the load factor  $\alpha$  for  $T$  as  $n/m$  which is the same as the average number of elements stores in a chain.

Worst case performing:  $\Theta(n)$  + the time to compute the hash function (all  $n$  keys hash to the same slot, creating a list of length  $n$ ).

Average case performing of hashing depends on how well the hash function  $h$  distributes the set of keys to be stores among the  $m$  slots, on the average.

**Theorem 11.1** In a hash table in which collisions are resolved by chaining, an unsuccessful search takes an average-case time  $\Theta(1 + \alpha)$ , under the assumption of simple uniform hashing. (OBS: Beviset herfor står på side 259!! LÆS OG FORSTÅ)

**Theorem 11.2** In a hash table in which collisions are resolved by chaining, a successful search takes average-case time  $\Theta(1 + \alpha)$ , under the assumption of simple uniform hashing. (OBS: Beviset herfor står på side 259-260!! LÆS OG FORSTÅ)

## 5.3 Hash functions

### What makes a good hash function?

A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the  $m$  slots, independently of where any other

key has hashed to. In practice, we can often employ heuristic techniques to create a hash function that performs well.

### **Interpreting keys as natural numbers**

Most hash functions assume that the universe of keys is the set  $\mathbb{N} = \{0, 1, 2, \dots\}$  of natural numbers.

#### **5.3.1 The division method**

#### **5.3.2 The multiplication method**

#### **5.3.3 Universal hashing**

### **5.4 Open addressing**

### **5.5 Perfect hashing**

## 6 Exact Exponential Algorithms

## 7 Approximation Algorithms

## 8 Computational Geometry



## 9 Linear Programming and Optimisation

[Yorgey [2012]]

## References

CLRS. *Introduction To Algorithms*. The MIT Press, USA, third edition, 2009.

Brent A. Yorgey. Monoids: Theme and variations (functional pearl). *Haskell '12*, 47: 105–116, 2012.