Advanced Algorithms and Datastructures

Exam Notes

Author:
Jenny-Margrethe Vej – 250986 – rwj935

Block 4, April-June, 2014

Contents

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 $f : V \times V \to \mathbb{R}$ 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[3]**:** For all $u \in V - \{s,t\}$, we require

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

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

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

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 [**?**, p. 710]. A firm wants to transport goods from one place to anther, using a third part driver.

Guessing this particular section is not thaaaaat important for the exam. $\smile$

1.1.3 Modelling problems with antiparallel edges

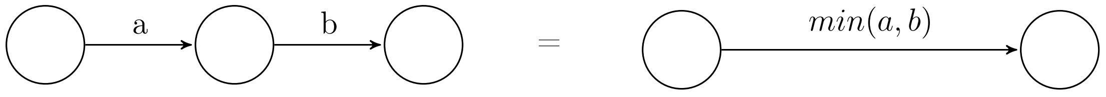
Antiparallel edges is 2 edges going to/from (v,u) - so 2 edges between 2 vertices but with opposite directions. To come around that, we transform our network into an equivalent one containing no antiparallel edges (adding an extra vertex for that, so we can split one of the edges). The resulting network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

[1]We call the nonnegative quantity $f(u,v)$ the flow from vertex u to vertex v

[2]Capacity Function c

[3]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

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

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

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

Example:

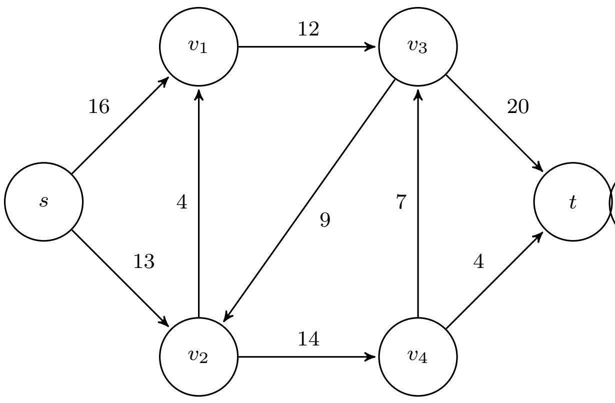


Figure 1: Flow Network G

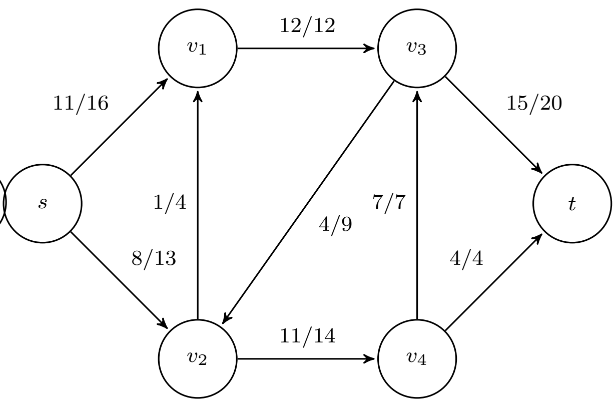


Figure 2: Flow f in G

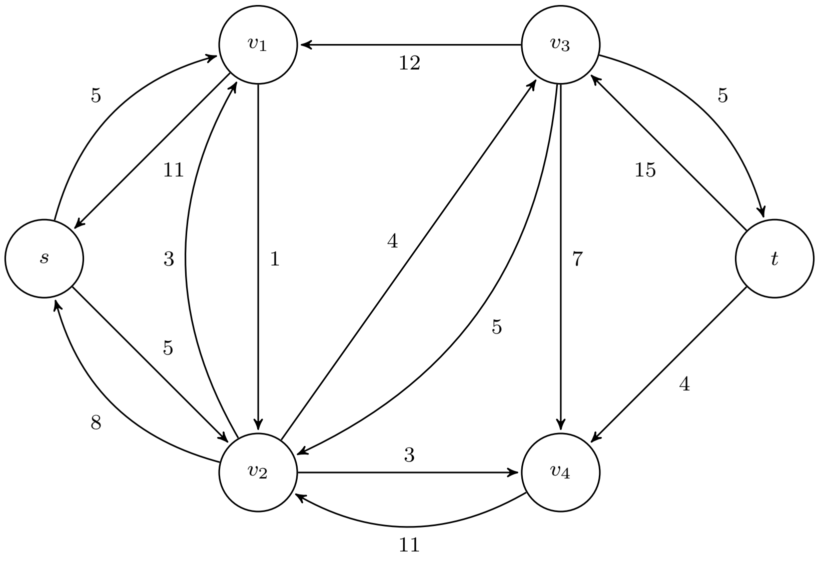


Figure 3: Residual Network G_f

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

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

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 constrains - we observe that if $(u, v) \in E$ then $c_f(v, u) = f(u, v)$. Therefore, we have $f'(v, u) \le c_f(v, u) = f(u, v)$, and hence

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

In addition we have,

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

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

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

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

A augmenting path is a simpe path, p from s to t in G_f.
The capacity of an augmenting path is the one of a critical edge on p. We can write this as $c_f(p) = min\{c_f(u,v) : (u,v) \in p\}$. The flow in G_f from path p can be defined as

$$f_p(u,v) = \begin{cases} c_p(p) & if (u,v) \in p \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

It is now obvious that for any augmenting path p we have $f \uparrow f_p$ as a flow in G with $|f \uparrow f_p| = |f| + |f_p| > |f|$

1.2.3 Cuts of Flow Networks

A cut (S,T), of a flow network $G = (V,E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. We define both the flow across a cut and the capacity of a cut:

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \tag{7}$$

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v) \tag{8}$$

A minimum cut is a cut with minimal capacity of all cuts.
For any cut (S,T) we have $f(S,T) = |f|$. The prove for this is:

First we look at the flow definition:

$$\begin{aligned} |f| &= \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) \\ &= \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{u \in S - \{s\}} \left(\sum_{v \in V} f(u,v) - \sum_{f(v,u)} \right) \end{aligned}$$

Besides that, see page 722 - the main ideas is:

1. Expand the right-hand summation
2. Regroup into summations over $v \in V$ with edges going into v and another group with edged going out of v
3. Split all $v \in V$ into $v \in S$ and $v \in T$. Uses $S \cup T = V$
4. Cancel out therms and we get what we want

This means that $|f|$ (the value of f) is bounded above by the capacity of all cuts. Especially the flow is bounded above by the capacity of a minimum cut. This leeds us to the Max-flow min-cut theorem. Max-flow min-cut *duality* (Pawel er fan af dette ord) is that the following are equivalent:

1. f is a maximum flow in G
2. The residual network G_f contains no augmenting paths
3. $|f| = c(S,T)$ for some cut (S,T) of G

Proof
(1) $\Rightarrow$ (2): If we assume f has a max flow, and that there is an augmenting path p in G_f. Let f_p denote the max flow along p - then $|f \uparrow f_p| = |f| + |f_p| > |f|$ (hence, $f \uparrow f_p$ would be a flow in G wit a bigger value $\rightarrow$ contradiction)

(2) $\Rightarrow$ (3): We define $S = \{v \in V$: there exists a path from s to v in $G_f\}$ and $T = V \setminus S(S,T) \rightarrow$ is a cut. Now we would like to show $f(S,T) = c(S,T)$.

Consider edge $(u, v) \in E$ with $u \in S$ and $v \in T \rightarrow f(u, v) = c(u, v)$
Consider edge $(v, u) \in E$ with $u \in S$ and $v \in T \rightarrow f(v, u) = 0$

Then

$$\begin{aligned} f(S,T) &= \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u) \\ &= \sum_{u\in S}\sum_{v\in T} c(u,v) - \sum_{u\in S}\sum_{v\in T} 0 \\ &= c(S,T) \end{aligned}$$

(3) $\Rightarrow$ (1): We let f be a flow in G, $|f'| \leq c(S,T) = |f| \Rightarrow f$ is a max flow. (In other words - we cannot have $|f| > c(S,T)$)

1.2.4 The basic Ford-Fulkerson Algorithm

I have decided this page in the book can not be that important due to all the stuff above (The Ford-Fulkerson Algorithm is basically just an expansion on the Ford-Fulkerson Method from above)- the stuff above must be the important part (including the analysis below) for Ford-Fulkerson. But read it and remember it for the overall understanding of the topic.

1.2.5 Analysis of Ford-Fulkerson

The running time depends on how we find the augmenting path p. If we assume an appropriate data structure where we can represent the directed graph, the time to find an appropriate path can be linear in the number of edges, if using breadth-first[4] or depth-first[5] search.

This gives os $O(E)$ work per iteration of the while loop, and at most the same number of iterations as the value of the maximum flow (since we increase by at least one unit per iteration). The total running time is therefore $O(E|f^*|)$ where $|f^*|$ is the maximum flow.

1.2.6 The Edmond-Karp Algorithm

We can improve the bound on Ford-Fulkerson by finding the augmenting path p with breadth-first search. That is, we choose the augmenting path as a shortest path from s

[4]In graph theory, breadth-first search (BFS) is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbour the currently visited node. The BFS begins at a root node and inspects all the neighbouring nodes. Then for each of those neighbour nodes in turn, it inspects their neighbour nodes which were unvisited, and so on

[5]Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

to t in the residual network, where each edge has unit distance (weight). Doing this to the Ford-Fulkerson Method is called the Edmonds-Karp algorithm.

This gives us $O(VE^2)$ as running time.

First step on proving this is to see that the minimum path length (amount of edges) in G_f increases monotonically. Let $\delta_f(s, v)$ be the minimum path length from s to v in G_f. Then we look at the smallest $\delta_{f'}(s, v)$ that changed when augmenting f with f'. Let u be the vertex before v in the path from s to v. We must have $\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$. We also have $\delta_f(s, u) = \delta_{f'}(s, u)$. Therefore $(u, v) \neq E_f$. We must therefore have had $\delta_f(s, v) = \delta_f(s, u) - 1$ which leads to contradiction.

We can use this to show that an edge can at most be critical $\frac{|V|}{2}$ times.

Simply look at $\delta_f(s, v) = \delta_f(s, u) + 1$ when (u, v) is critical. In order for the edge to return to the residual network we must have (v, u) be critical. This can only happen when $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$. Since $\delta_{f'}(s, v) + 1 \geq \delta_f(s, v) + 1 = \delta_f(s, u) + 2$ the result follows.

Each augmenting path has at least one critical edge and only $O(VE)$ times can there be a critical edge. This gives the running time.

1.3 Maximum bipartite matching

This topic sucks! Jeg har basically bare kopieret noter fra Søren D. Men fatter sgu ikke så meget af det. Derfor er det heller ikke med i min disposition. Håber på, jeg aldrig bliver spurgt ind til det.

1.3.1 The Maximum-bipartite-mathing problem

A matching is a subset $M \subseteq E$ such that for all vertices $v \in V$ at most one edge in M is incident to v. A maximum matching is a matching M that has $|M| \geq |M'|$ for any other matching M'.

1.3.2 Finding a maximum bipartite matching

We can create a graph G' with nodes $V' = V \cup \{s, t\}$ and edges

$$E' = \{(s, u) : u \in L\} \cup \{(u, v) : (u, v) \in E\} \cup \{(u, t) : u \in R\}$$

All capacities $c(u, v) = 1$. It is clear that $|E'| = O(|E|)$ because we assume each node $v \in L \cup R$ has one edge incident in E.

We wanna show that a flow f in G' corresponds to a matching M in G. First we say that a flow is integer-valued if $f(u, v)$ is integer for all $(u, v) \in V \cdot V$. The claim is that if M is a matching in G it corresponds to a flow f in G' with $|f| = |M|$ and the other way around.

To proof this, we say for a matching M create a flow $f(u, v) = f(s, u) = f(u, t) = 1$ for all $(u, v) \in M$. It is easy to see that the constraints are satisfied and that $|f| = |M|$. For a flow f we create a matching

$$M = \{(u, v) : u \in L, v \in R, \text{and} f(u, v) > 0$$

Because f is integer-valued this is okay. To see that this is a matching use that $(s, u) = 1$ and flow conservation means the sum over $(u, v) = 1$ (or 0). We can also use this to show that $|M| = |f|$.

We also need to show that when $c(u, v)$ is integral for all (u, v) the maximum flow found by Ford-Fulkerson will be integer-valued. This is done easily by induction over the iteration. Base case is trivial cause $|f| = 0$.

We can now proof by contradiction that a maximum flow corresponds to a maximum matching.

1.4 Disposition

The following disposition is what you can manage in 15 minutes.

- Flow Network G
- Flow $|f|$
  - Capacity Constraints
  - Flow Conservation
- Ford-Fulkerson Method
  - Residual Network & Augmenting Paths
  - Max-Flow Min-Cut Theorem (with prove)
  - Running time (perhaps mention Edmonds-Karp - at least have an understanding of the prove, but you have no time to go through the prove)

2 Fibonacci Heaps

Okaaaaaay! This is awesome! Kick some serious ass!

2.1 Structure of Fibonacci Heaps

A Fibonacci Heap is a collection of rooted trees that are min-heap ordered. That is, each tree obeys the min-heap property: the key of a node is greater than or equal to the key of its parent. Each node contains a pointer to its parent and a pointer to any one of its children. The children are linked together in a circular, doubly linked list. In a circular, doubly linked list we can insert a node into any location or remove a node from anywhere in the list in $O(1)$ time. We can also concatenate two such lists in $O(1)$ time.

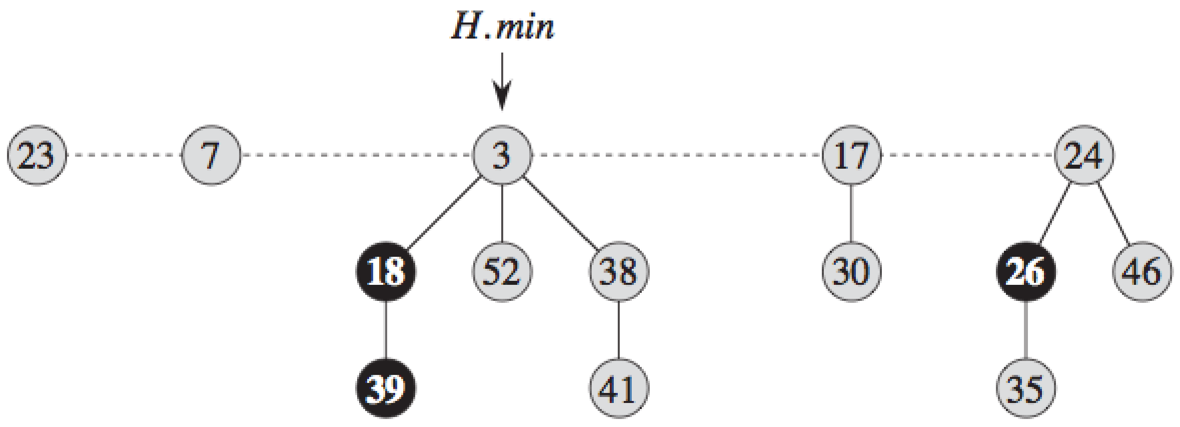


Figure 4: Fibonacci heap consisting of five min-heap-ordered trees and 14 nodes

2.1.1 Potential Function

2.1.2 Maximum Degree

2.2 Mergeable-heap Operations

2.2.1 Creating a New Fibonacci Heap

2.2.2 Inserting a Node

2.2.3 Finding the Minimum Node

2.2.4 Uniting Two Fibonacci Heaps

2.2.5 Extracting the Minimum Node

2.3 Decreasing a Key and Deleting a Node

2.3.1 Decreasing a Key

2.3.2 Deleting a Node

2.4 Bounding the Maximum Degree

2.5 Disposition

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 $U = \{0, 1, ..., m - 1\}$ where m is not too large, and no two elements have the same key.

Representing it by a direct-address table, or array $T[0...m - 1]$, 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

```
DIRECT-ADDRESS-SEARCH($T, k$)
return $T[k]$

DIRECT-ADDRESS-INSERT$T, x$
return $T[key[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE$T, x$
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 stores 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-DELETE$T, x$
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 α 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 $\mathbb{N} = \{0, 1, 2, ...\}$ 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

[?]