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. $\smile\hspace{-0.9em}\raisebox{0.3em}{..}$

1.1.3 Modelling problems with antiparallel edges

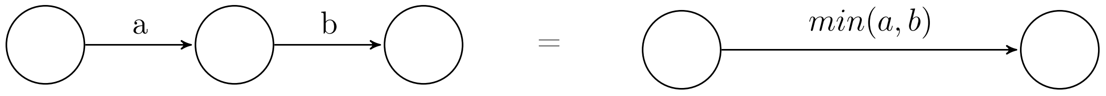
Antiparallel edges is 2 edges going to/from (v,u) - so 2 edges between 2 vertices but with opposite directions. To come around that, we transform our network into an equivalent one containing no antiparallel edges (adding an extra vertex for that, so we can split one of the edges). The resulting network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

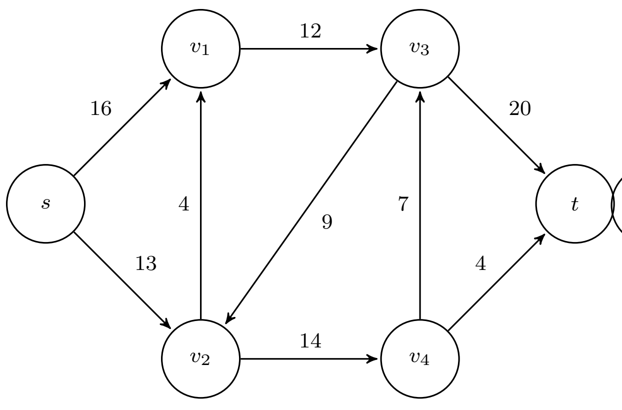


Figure 1: Flow Network G

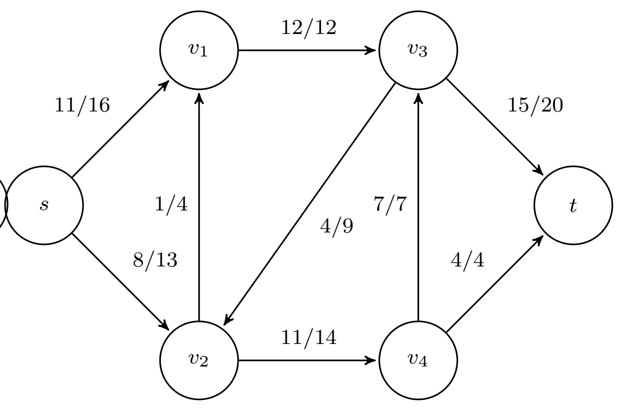


Figure 2: Flow f in G

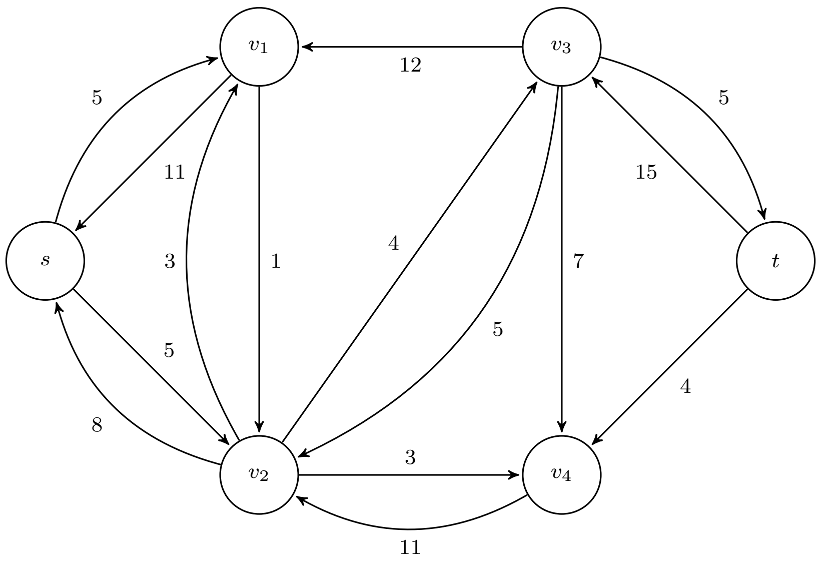


Figure 3: Residual Network G_f

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

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

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 $f'(v, u) \leq c_f(v, u) = f(u, v)$, and hence

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

In addition we have,

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

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!

2.0.1 Mergeable Heaps

A mergeable heap is any data structure that supports the following five operations in which each has a key:

- MAKE-HEAP() creates and returns a new heap containing no elements
- INSERT(H, x) inserts an element x, whose *key* has already been filled in, into heap H
- MINIMUM(H) returns a pointer to the element in heap H whose key is minimum
- EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element
- UNION(H_1, H_2) creates and returns a new heap that contains all the elements of heaps H_1 and H_2. Heaps H_1 and H_2 are "destroyed" by this operation

Besides the mergeable-heap operations above, Fibonacci heaps also support the following two operations:

- DECREASE-KEY(H, x, k) assigns to element x within heap H the new key value k, which we assume to be no greater than its current value
- DELETE(H, x) deletes element x from heap H

Procedure	Binary heap (worst case)	Fibonacci heap (amortised)
MAKE-HEAP	$\Phi(1)$	$\Phi(1)$
INSERT	$\Phi(\lg(n))$	$\Phi(1)$
MINIMUM	$\Phi(1)$	$\Phi(1)$
EXTRACT-MIN	$\Phi(\lg(n))$	$O(\lg(n))$
UNION	$\Phi(n)$	$\Phi(1)$
DECREASE-KEY	$\Phi(\lg(n))$	$\Phi(1)$
DELETE	$\Phi(\lg(n))$	$O(\lg(n))$

Table 1: Running times for operations on two implementations of mergeable heaps. The number of items in the heap(s) at the time of an operation is denoted by n

2.0.2 Fibonacci heaps in theory and practice

Fibonacci heaps are theoretically desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed. This can be the case for some algorithms in graph problems, that call DECREASE-KEY once per edge. Fast algorithms for computing minimum spanning trees and finding single-source shortest paths make essential use of Fibonacci heaps.

From a practical point of view, however, the constant factors and complexity of Fibonacci heaps make them less desirable than ordinary binary (or k-ary) heaps for most applications. As such, Fibonacci heaps are pre-dominantly of theoretical interest.

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.

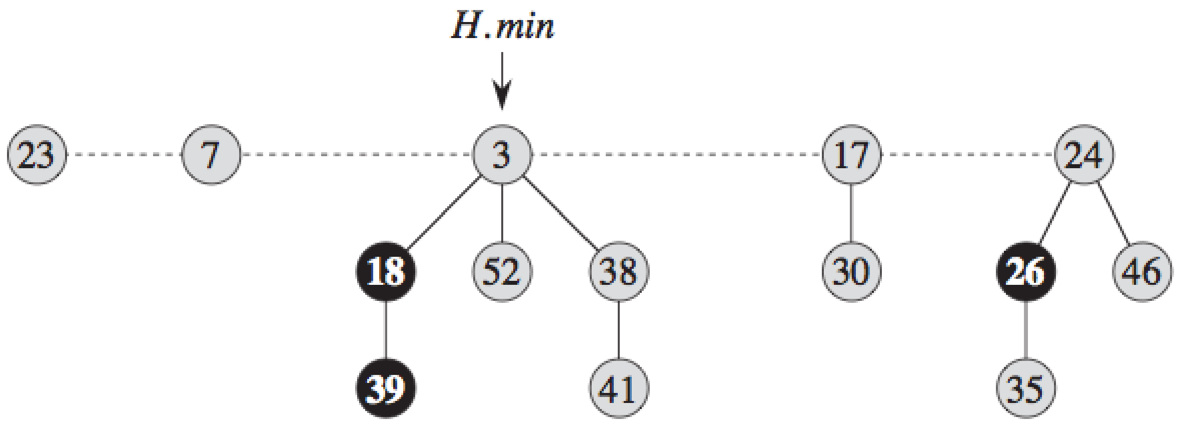


Figure 4: Fibonacci heap consisting of five min-heap-ordered trees and 14 nodes

We store the number of children in the cild list of node x in $x.degree$. The boolean-valued attribute $x.mark$ indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node becomes unmarked whenever it is made the child of another node. Root nodes are unmarked.

We access a Fibonacci heap H by a pointer $H.min$ to the root of a tree containing the minimum key; we call this node the *minimum node* of the Fibonacci heap.

The roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular, doubly linked list called the *root list* of the Fibonacci heap. Finally, $H.n$ refers to the number of nodes currently in the heap.

2.1.1 Potential Function

In the following, the potential function of fibonacci heap is defined by

$$\Phi(H) = t(H) + 2m(H) \tag{9}$$

where $t(H)$ is the number of trees in the root list and $m(H)$ is the number of marked nodes. The potential function captures the state of a system at a given point, and we can then calculate an amortised[6] time as the actual time plus a constant times the change in state 'energy'. This provides a valid upper bound on running time.

2.1.2 Maximum Degree

Meeh, nothing important, I think...

2.2 Mergeable-heap Operations

The mergeable-heap operations on Fibonacci heaps delay work as long as possible. We shall see that, no matter what the root list looks like before a EXTRACT-MIN operation, afterward each node in the root list has a degree that is unique within the root list, which leads to a root list of size at most $D(n) + 1$.

2.2.1 Creating a New Fibonacci Heap

To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object H, where $H.n = 0$ and $H.min = NIL$; there are no trees in H. Because $t(H) = 0$ and $m(H) = 0$, the potential of the empty Fibonacci heap is $\Phi(H) = 0$. The amortised cost of MAKE-FIB-HEAP is thus equal to its $O(1)$ actual cost.

2.2.2 Inserting a Node

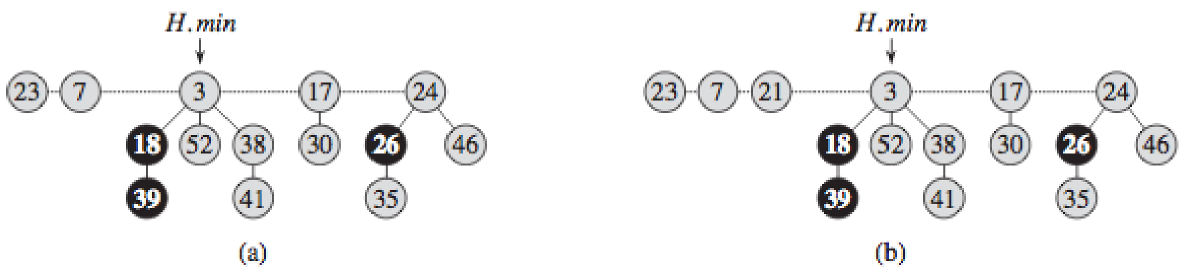


Figure 5: Inserting a node into a Fibonacci heap. **(a)** A Fibonacci heap H. **(b)** Fibonacci heap H after inserting the node with key 21. The node becomes its own min-heap-ordered tree and is then added to the root list, becoming the left sibling of the root.

[6]Amortiseret = Udover n operationer, så er det gennemsnittet

To determine the amortised cost of FIB-HEAP-INSERT, let H be the input Fibonacci heap and H' be the resulting Fibonacci heap. Then $t(H') = t(H)+1$ and $m(H') = m(H)$, and the increase in potential is $((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1$.

Since the actual cost is $O(1)$, the amortised cost is $O(1) + 1 = O(1)$.

2.2.3 Finding the Minimum Node

The minimum node of a Fibonacci heap H is given by the pointer $H.min$, so we can find the minimum node in $O(1)$ actual time. Because the potential of H does not change, the amortised cost of this operation is equal to its $O(1)$ actual cost.

2.2.4 Uniting Two Fibonacci Heaps

We simply concatenate the root lists of H_1 and H_2 and then determines the new minimum node. Afterward, the objects representing H_1 and H_2 will never be used again.

The change in potential is
$\Phi(H) - (\Phi(H_1) + \Phi(H_2)) = (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) = 0$ because $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. The amortised cost of FIB-HEAP-UNION is therefore equal to its $O(1)$ actual cost.

2.2.5 Extracting the Minimum Node

The process of extracting the minimum node is the most complicated of the operations presented in this section. It is also where the delayed work of consolidating trees in the root list finally occurs.

FIB-HEAP-EXTRACT-MIN works by first making a root out of each of the minimum node's children and removing the minimum node from the root list. It then consolidates the root list by linking roots of equal degree until at most one root remains of each degree.

Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct degree value:

1. Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$
2. ***Link*** y to x: remove y from the root list, and make y a child of x by calling the FIB-HEAP-LINK procedure. This procedure increments the attribute $x.degree$ and clears the mark on y.

So what we do is, removing $H.min$ and add its children to the root list. Then we start the consolidation with array A, starting at the new $H.min$ going to right. When we have to nodes with same degree (height) in a row, we merge them. In the end we have a bunch of nodes at the root list with different degrees, and we mark the new $H.min$.

There are several things we must estimate the cost of: extracting the minimum node, the main for-loop through the root list and within that the while-loop that links together trees.

The amortised cost of extracting the minimum node of an n-node Fibonacci heap is $O(D(n))$. Let H denote the Fibonacci heap prior to the ExtractMin operation. The actual cost of extracting the minimum node contributes $O(D(n))$ work from processing at most $D(n)$ children of the minimum node.

Upon calling consolidate, the size of the root list is at most $D(n) + t(H) - 1$: the original $t(H)$ root-list nodes, minus the extracted root node, plus the children of the extracted node, at most $D(n)$. Every iteration of the inner while-loop, we know that one root is linked to another, and so the total number of iterations of the while loop over all iterations of the for loop is at most the number of roots in the root list. Hence, the total amount of work performed in the for loop is at most proportional to $D(n) + t(H)$. The total actual work in extracting the minimum node is therefore $O(D(n) + t(H))$. The potential before extracting the minimum node is $t(H) + 2m(H)$. The potential afterwards is at most $(D(n)+1)+2m(H)$, since at most $D(n)+1$ roots remain and no nodes get marked during this operation. The amortised cost is thus at most:

$$\begin{aligned}
&O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\
&= O(D(n)) + O(t(H)) - O(t(H)) \\
&= O(D(n))
\end{aligned}$$

if we scale up the units of the potential to dominate the constant hidden in $t(H)$. We will see later that $D(n) = O(\lg n)$, giving an amortised running time of $O(\lg n)$ for extracting the minimum node.

2.3 Decreasing a Key and Deleting a Node

2.3.1 Decreasing a Key

To decrease a key a few things happens. We start with an initial Fibonacci heap. It is important to notice the marked nodes. A node is now decreased and it is moved to the root list, and its parent is marked. IF the parent was marked already, the marked parent is cut from its own parent and moved to the root list unmarked. When we reach an unmarked node or a root key, the algorithm stops, and we update $H.min$ if necessary.

The FIB-HEAP-DECREASE-KEY procedure takes $O(1)$ time plus the time to perform the cascading cuts. Suppose that a given invocation of FIB-HEAP-DECREASE-KEY results in c calls of CASCADING-CUT - each of these calls takes $O(1)$ time exclusive recursive calls. Thus, the actual cost is $O(c)$.

Next we compute the change in potential. We let H denote the Fibonacci heap just prior to the FIB-HEAP-DECREASE-KEY operation. The call to CUT creates a new tree rooted at node x and clears x's mark bit. Each call of CASCADING-CUT, except for the last one, cuts a marked node and clears the mark bit. The Fibonacci heap now contains $t(H)+c$ trees and at most $m(H)-c+2$ marked nodes. The change in potential is therefore at most

$((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H))=4-c.$

Thus, the amortised cost of FIB-HEAP-DECREASE-KEY is at most $O(c)+4-c=O(1)$, since we can scale up the units of potential to dominate the constant hidden in $O(c)$. You can now see why we defined the potential function to include a term that is twice the number of marked nodes. When a marked node y is cut by a cascading cut, its mark bit is cleared, which reduces the potential by 2. One unit of potential pays for the cut and the clearing of the mark bit, and the other unit compensates for the unit increase in potential due to node y becoming a root.

2.3.2 Deleting a Node

Deleting a node works by first setting its key to minus infinity, and then extracting the minimum. We already know that decreasing a key takes $O(1)$ amortised time, and extracting minimum takes $O(D(n))$ amortised time. As we will see that $D(n)=O(\lg n)$, deleting a node runs in amortised time of $O(\lg n)$.

2.4 Bounding the Maximum Degree

To prove that the amortised time of FIB-HEAP-EXTRACT-MIN and FIB-HEAP-DELTE is $O(\lg n)$, we must show that the upper bound $D(n)$ on the degree of any node of an n-node Fibonacci heap is $O(\lg n)$. In particular, we shall show that $D(n)\leq\lfloor\log_\phi n\rfloor$, where ϕ is the golden ration, defined in the book as $\phi=\frac{(1+\sqrt{5})}{2}=1.61803...$

The key to the analysis is as follows. For each node x within a Fibonacci heap, define size(x) to be the number of nodes, including x itself, in the subtree rooted at x. We shall show that size(x) is exponential in $x.degree$.

INDSÆT NOGET HER, SOM DU KAN LÆSE OP PÅ, NÅR DU ØVER. DER VIL NÆPPE VÆRE TID TIL AT SNAKKE ET 4 SIDERS OPLÆG IGENNEM MED 5 LEMMAER/COROLLARER

2.5 Disposition

3 NP-Completeness

Who wouldn't love to be examinated in this? GO NUTS! YOU CAN DEFINITELY DO IT!!

3.0.1 NP-Completeness and the classes P and NP

Informal description of the three classes - more formal later:

P: This class consist of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.

NP: The class NP consists of those problems that are "verifiable" in polynomial time, meaning that if we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem.

Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate.

NPC: A problem is in the NPC class if it is in NP and as "hard" as any problem in NP. If any NP-Complete problem can be solved in polynomial time, then every problem in NP has a polynomial-time algorithm.

3.0.2 Overview of showing problems to be NP-Complete

We rely in three key concepts in showing a problem to be NP-Complete:

1. Decision problems vs. optimisation problems
2. Reductions
3. A first NP-Complete problem

3.1 Polynomial Time

We begin our study of NP-completeness by formalising our notion of polynomial- time solvable problems. We generally regard these problems as tractable, but for philosophical, not mathematical, reasons. We can offer three supporting arguments.

1. Very few practical problems require time on the order of a high-degree polynomial, for example $\Theta(n^{100})$
2. Second, for many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another

3. Third, the class of polynomial-time solvable problems has nice closure proper- ties, since polynomials are closed under addition, multiplication, and composition

3.1.1 Abstract Problems

To understand the class of polynomial-time solvable problems, we must first have a formal notion of what a "problem" is. We define an abstract problem Q to be a binary relation on a set I of problem instances and a set S of problem solutions. This formulation of an abstract problem is more general than we need for our purposes, since the theory of NP-completeness restricts attention to decision problems: those having a yes/no solution. In this case, we can view an abstract decision problem as a function that maps the instance set I to the solution set $\{0, 1\}$.

3.1.2 Encodings

In order for a computer program to solve an abstract problem, we must represent problem instances in a way that the program understands. An encoding of a set S of abstract objects is a mapping e from S to the set of binary strings.

Thus, a computer algorithm that "solves" some abstract decision problem actually takes an encoding of a problem instance as input. We call a problem whose instance set is the set of binary strings a concrete problem. We say that an algorithm solves a concrete problem in time $O(T(n))$ if, when it is provided a problem instance i of length $n = |i|$, the algorithm can produce the solution in $O(T(n))$ time. A concrete problem is polynomial-time solvable, therefore, if there exists an algorithm to solve it in time $O(T(n))$ for some constant k.

We can now formally define the complexity class P as the set of concrete decision problems that are polynomial-time solvable.

We would like to extend the definition of polynomial-time solvability from concrete problems to abstract problems by using encodings as the bridge, but we would like the definition to be independent of any particular encoding. That is, the efficiency of solving a problem should not depend on how the problem is encoded. Unfortunately, it depends quite heavily on the encoding if the algorithm runs in either polynomial or superpolynomial time.

We cannot really talk about solving an abstract problem without first specifying an encoding. Nevertheless, in practice, if we rule out "expensive" encodings such as unary ones, the actual encoding of a problem makes little difference to whether the problem can be solved in polynomial time.

Lemma 34.1 Let Q be an abstract decision problem on an instance set I, and let e_1 and e_2 be polynomially related encodings on I. Then, $e_1(Q) \in P$ if and only if $e_2(Q) \in P$.[7]

Thus, whether an abstract problem has its instances encoded in binary or base 3 does not affect its "complexity", that is, whether it is polynomial-time solvable or not; but if instances are encoded in unary, its complexity may change. We shall assume that the encoding of an integer is polynomially related to its binary representation, and that the encoding of a finite set is polynomially related to its encoding as a list of its elements, enclosed in braces and separated by commas. To denote the standard encoding of an object, we shall enclose the object in angle braces. Thus, $\langle G \rangle$ denotes the standard encoding of a graph G.

3.1.3 A Formal-Language Framework

By focusing on decision problems, we can take advantage of the machinery of formal-language theory. Let us review some definitions from that theory. An alphabet Σ is a finite set of symbols. A language L over Σ is any set of strings made up of symbols from Σ.

We denote the empty string by ϵ, the empty language by $\emptyset$, and the language of all strings over Σ by Σ^*. Every language L over Σ is a subset of Σ^*.

The next part in the book is talking about the operations we can do on the languages. I have decided not to make any notes to that.

3.2 Polynomial-time Verification

3.2.1 Hamiltonian cycles

The problem of finding a hamiltonian cycle in an undirected graph has been studied for over a hundred years. Formally, a hamiltonian cycle of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V. The Hamiltonian-cycle problem is NP-Complete, which we will prove later.

3.2.2 Verification Algorithms

We define a verification algorithm as being a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a certificate. A two-argument algorithm A verifies an input string x if there exists a certificate y such that $A(x, y) = 1$. The language verified by a verification algorithm A is
$L = \{x \in \{0,1\}^* : \text{ there exists} y \in \{0,1\}^* \text{ such that} A(x, y) = 1\}$.

[7] I have decided not to focus on all the small proves, but on the proves I think could be relevant for my disposition. The prove for the Lemma 34.1 is on page 1057 in the book

3.2.3 The complexity Class NP

The complexity class NP is the class of languages that can be verified by a polynomial-time algorithm. More precisely, a language L belongs to NP if and only if there exist a two-input polynomial-time algorithm A and a constant c such that

$$L = \{x \in \{0,1\}^* : \text{ there exists a certificate} y \text{ with} |y| = O(|x|^c) \text{ such that} A(x,y) = 1$$

We say that algorithm A verifies language L in polynomial time.

3.3 NP-Completeness and Reductability

Perhaps the most compelling reason why theoretical computer scientists believe that $P \neq NP$ comes from the existence of the class of "NP-complete" problems. This class has the intriguing property that if any NP-complete problem can be solved in polynomial time, then every problem in NP has a polynomial-time solution, that is, $P = NP$.

3.3.1 Reductability

Lemma 34.3
If $L_1, L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \leq_p L_2,$ then $L_2 \in P$ implies $L_1 \in P$

3.3.2 NP-completeness

Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial-time factor. That is, if $L_1 \leq_p L_2$, then L_1 is not more than a polynomial factor harder than L_2, which is why the "less than or equal to" notation for reduction is mnemonic. We can now define the set of NP-complete languages, which are the hardest problems in NP. A language $L \subseteq \{0,1\}^*$ is NP-complete if

1. $L \in$ NP, and
2. $L' \leq_p L$ for every $L' \in$ NP

If a language L satisfies property 2, but not necessarily property 1, we say that L is NP-hard. We also define NPC to be the class of NP-complete languages.

As the following theorem shows, NP-completeness is at the crux of deciding whether P is in fact equal to NP.

Theorem 34.4
If any NP-complete problem is polynomial-time solvable, then P = NP. Equivalently, if

any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.

3.3.3 Circuit Satisfiability

We now focus on demonstrating the existence of an NP-complete problem: the circuit-satisfiability problem. Boolean combinational circuits are built from boolean combinational elements that are interconnected by wires. A boolean combinational element is any circuit element that has a constant number of boolean inputs and outputs and that performs a well-defined function.

The boolean combinational elements that we use in the circuit-satisfiability problem compute simple boolean functions, and they are known as logic gates. The three basic logic gates that we use in the circuit-satisfiability problem is the NOT gate ($\neg$), the AND gate ($\wedge$) and the OR gate ($\vee$).

A boolean combinational circuit consists of one or more boolean combinational elements interconnected by wires. A wire can connect the output of one element to the input of another, thereby providing the output value of the first element as an input value of the second.

Boolean combinational circuits contain no cycles. In other words, suppose we create a directed graph $G = (V, E)$ with one vertex for each combinational element and with k directed edges for each wire whose fan-out is k; the graph contains a directed edge (u, v) if a wire connects the output of element u to an input of element v. Then G must be acyclic.

The circuit-satisfiability problem is, "Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?" In order to pose this question formally, however, we must agree on a standard encoding for circuits.

Lemma 34.5 The circuit-satisfiability problem belongs to the class NP
Lemma 34.6 The circuit-satisfiability problem is NP-hard
Theorem 34.7 The circuit-satisfiability problem is NP-complete

The proves for the last lemmas and theorem is in the book - I do not think I have to prove it for the exam. At least I hope not.

3.4 NP-completeness proofs

In this section, we shall show how to prove that languages are NP-Complete without directly reducing every language in NP to the given language. The following lemma is the basis of our method for showing that a language is NP-Complete.

Lemma 34.8 If L is a language such that $L' \leq_p L$ for some $L' \in NPC$, then L is NP-hard. If, in addition, $L \in$ NP, then $L \in$ NPC.

By reducing a known NP-Complete language L' to L, we implicitly reduce every language in NP to L. Thus Lemma 34.8 gives us a method for proving that a language L is NP-Complete:

1. Prove $L \in$ NP
2. Select a known NP-Complete language L'
3. Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L
4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$
5. Prove that the algorithm computing f runs in polynomial time

(Steps 2–5 show that L is NP-hard.) This methodology of reducing from a single known NP-Complete language is far simpler than the more complicated process of showing directly how to reduce from every language in NP.

3.4.1 Formula satisfiability

Meeeh - skim it again and that is it.

3.5 NP-Complete Problems

This section is examples of NP-Complete problems. I have chosen to make notes to only 2 (maybe only one, if I give up before time).

3.5.1 The Clique Problem

To long (but read what the problem is)

3.5.2 The Vertex Cover Problem

A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). That is, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E. The size of a vertex cover is the number of vertices in it.

Theorem 34.12 The Vertex-Cover Problem is NP-Complete

Prove:
- *First* we prove that VERTEX-COVER $\in$ NP. We have a graph $G = (V, E)$ and an integer k. The certificate we choose is the vertex cover $V' \subseteq V$ itself. The verification algorithm affirms that $IV'I = k$ and then it checks, for each edge $(u, v) \in E$, that $u \in V'$ or $v \in V'$. We can easily verify the certificate in polynomial time.

- *Next* we prove that the Vertex-Cover Problem is NP-hard by showing that CLIQUE $\leq_p$ VERTEX-COVER. (THIS IS NOT DONE RIGHT NOW - CAN NOT HANDLE CLIQUE 30 MINUTES TO MINDNIGHT ON A FRIDAY)

3.5.3 The Hamiltonian-Cycle Problem

To long (but read what the problem is)

3.5.4 The Traveling-Salesman Problem

In the traveling-salesman problem, which is closely related to the Hamiltonian-Cycle Problem, a salesman must visit n cities. We model this problem with a complete graph with n vertices, and the salesman wants to make a tour, or Hamiltonian Cycle, visiting each city exactly once, and finishing in the city he starts from.

$$\begin{aligned} TSP = \{\langle G, c, k\rangle : \; & G = (V, E) \text{ is a complete graph} \\ & c \text{ is a function from } V \text{ x } V \rightarrow \mathbb{Z} \\ & k \in \mathbb{Z} \\ & G \text{ has a traveling-salesman tour with cost at most } k\} \end{aligned}$$

Theorem 34.14 The Traveling-Salesman Problem is NP-Complete

Prove:
- *First* we show that TSP belongs to NP. Given an instance of the problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is at most k. This process can certainly be done in polynomial time.

- *Second* we show that TSP is NP-Hard by showing that HAM-CYCLE $\leq_p$ TSP. Let $G = (V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$.

The instance of TSP is then $\langle G', c, 0\rangle$, which we can easily create in polynomial time.

- *Last* We show that graph G has a hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'. Thus, h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1, the cost of tour h' is exactly 0 and each edge on the tour must have cost 0. Therefore, h' contains only edges in E. We conclude that h' is a hamiltonian cycle in graph G.

3.5.5 The Subset-Sum Problem

To long (but read what the problem is)

3.6 Disposition

4 Randomised Algorithms

OMG!!! This you can do! Even in danglish

4.1 Randomised Quicksort

Vi vil sortere sættet S på n numre. Hvis vi kan finde et y af S, sådan at halvdelen er mindre end y, så kan vi dele sættet i 2 bortset fra y, sådan at S_1 indeholder alle elementer, der er $< y$ og S_2 resten. Så sorterer vi rekursivt S_1 og S_2 og outputter elementerne af S_1 sorteret stigende, efterfuldt af y og så elementerne af S_2, også i stigende rækkefølge. Hvis vi kunne finde y i cn trin for en eller anden konstant cm ville vi kunne partionere $S \setminus \{y\}$ til S_1 og S_2 i $n-1$ tilsvarende trin ved at sammenligne hvert element af S med y - og dermed ville det totale antal trin være givet af følgende rekurrens:

$$T(n) \leq 2T(\frac{n}{2}) + (c+1)n \tag{10}$$

hvor $T(k)$ repræsenterer tiden det tager for denne metode at sortere k input. Ved direkte substitution kan vi verificere, at $T(n) \leq c'n \lg n$ for en konstant c'.

Der er en vis udfordring i, at dele S_1 og S_2 i to lige store dele. Her prøver vi at gøre det med tilfældighed, og ender med algoritmen: RandQS = Randomised Quicksort[8].

Køretiden for RandQS kan analyseres til følgende:
Køretiden for randomiseret Quicksort er analyseret ud fra antallet af sammenligninger. Vi vil gerne finde the forventede antal af sammenligninger.

For $1 \leq i \leq n$, lad $S_{(i)}$ angive elementet af rang i (det $i$te mindste element) i sæt S. Således er $S_{(1)}$ det mindste element og $S_{(n)}$ det største. Definér den tilfældige variabel X_{ij} til at antage værdien 1 hvis $S_{(i)}$ og $S_{(j)}$ er sammenlignet, og værdien 0 ellers.

Det totale antal sammenligninger er

$$\sum_{i=1}^{n} \sum_{j>i} X_{ij}$$

Det forventede antal sammenligninger er (by Linearity of Expectation)

$$\mathbb{E}[\sum_{i=1}^{n} \sum_{j>i} X_{ij}] = \sum_{i=1}^{n} \sum_{j>i} \mathbb{E}[X_{ij}]$$

[8]RandQS er et eksempel på en randomiseret algoritme - altså en algoritme, der foretager et tilfældigt valg på et tidspunkt i sin eksekvering - her i step 1

Så lader vi p_{ij} angive sansynligheden for at $S_{(i)}$ og $S_{(j)}$ bliver sammenlignet i en udførelse. X_{ij} kan antage værdierne 0 og 1, så:

$$\mathbb{E}[X_{ij}] = p_{ij} \times 1 + (1 - p_{ij}) \times 0 = p_{ij}$$

For at bestemme p_{ij}, kigger vi på udførelsen af RandQS som et binært søgetræ T, hvor hver node er et særskilt element af S. Roden i træet er pivoten y, som vi valgte indledningsvist. Venstre subtræ indeholder elementerne i S_1 og højre subtræ S_2. Strukturen bestemmes rekursivt ved sekventielle kørsler af RandQS. Man sammenligner roden af træet y med elementerne i henholdsvis S_1 og S_2, men elementerne i S_1 og S_2 sammenlignes ikke med hinanden.

For at udregne p_{ij} laver vi to observationer:

1. Der er en sammenligning mellem $S_{(i)}$ og $S_{(j)}$, hvis og kun hvis $S_{(i)}$ eller $S_{(j)}$ fremkommer i permutationen π tidligere end noget element $S_{(l)}$ sådan at $i < l < j$. Intuitively, they are only compared if there is some pivot node l that puts i in S_1 and j in S_2. To see this, let $S_{(k)}$ be the earliest in π from among all elements of rank between i and j. If $k \notin \{i, j\}$ then $S_{(i)}$ will belong to the left sub-tree of $S_{(k)}$ while $S_{(j)}$ will belong to the right sub-tree of $S_{(k)}$, implying that there is no comparison between $S_{(i)}$ and $S_{(j)}$. Conversely, if $k \in \{i, j\}$ then it must be that there is a parent-child relationship between $S_{(i)}$ and $S_{(j)}$ implying that the two are compared.

2. Any of the elements $S_{(i)}, S_{(i+1)}, \ldots, S_{(j)}$ is equally likely to be chosen as the pivot and hence appear first in π. Thus, the probability that the first element is either $S_{(i)}$ or $S_{(j)}$ is exactly $2/(j - i + 1)$.

This establishes that $p_{ij} = 2/(j - i + 1)$. The expected number of comparisons is thus:

$$\begin{aligned}
\sum_{i=1}^{n} \sum_{j>i} p_{ij} &= \sum_{i=1}^{n} \sum_{j>i} \frac{2}{j - i + 1} \\
&\leq \sum_{i=1}^{n} \sum_{k=1}^{n-i+1} \frac{2}{k} \\
&\leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k}
\end{aligned}$$

It follows that the expected number of comparisons is bounded by $2nH_n$, where H_n is the *nth* Harmonic number, defined by $H_n = \sum_{k=1}^{n} 1/k$. We know that $H_n \sim ln\ n + \Theta(1)$, making the expected running time of RandQS $O(n\ lg\ n)$

4.2 Las Vegas & Monte-Carlo

An algorithm that *always* gives the correct solution, but whose running time is variable or perhaps even unbounded, is a **Las Vegas** algorithm. In contrast, an algorithm that

may sometimes produce an incorrect solution (although we can often bound the probability of this), but we know will always terminate, is a **Monte-Carlo** algorithm. One useful property of a Monte-Carlo algorithm is that we can reduce the risk of failure to be arbitrarily small by running the experiment repeatedly with independent random choices each time.
For decision problems (yes or no problems), there are two kinds of Monte-Carlo algorithms: those with one-sided error and those with two-sided error. A Monte-Carlo algorithm is said to have a two-sided error if there is a non-zero probability that it errs when it outputs either `YES` or `NO`. It is said to have one-sided error if the probability that it errs is zero for at least one of the possible outputs (`YES` / `NO`).
By definition, a Las Vegas algorithm is a Monte-Carlo algorithm with error probability 0. We say that a Las Vegas algorithm is an *efficient* Las Vegas algorithm if on any input its expected running time is bounded by a polynomial function of the input size. Similarly, we say that a Monte-Carlo algorithm is an *efficient* Monte-Carlo algorithm if on any input its worst-case running time is bounded by a polynomial function of the input size.

4.2.1 Markov & Chebyshev's inequalities

Markov inequality Let Y be a random variable assuming only non-negative values. Then for all $t \in \mathbb{R}^+$,

$$\Pr[Y \geq t] \leq \frac{\mathrm{E[Y]}}{t}.$$

Equivalently,

$$\Pr[Y \geq k\mathrm{E}[Y]] \leq \frac{1}{k}.$$

Proof Define a function $f(y)$:

$$f(y) = \begin{cases} 1 & \text{iff } y \geq t \\ 0 & \text{otherwise.} \end{cases}$$

Then $\Pr[Y \geq t] = \mathrm{E}[f(y)]$. Since $f(y) \leq y/t$ for all y,

$$\mathrm{E}[f(Y)] \leq \mathrm{E}\left[\frac{Y}{t}\right] = \frac{\mathrm{E}[Y]}{t},$$

and the theorem follows.

Chebyshevs inequality Let X be a random variable with expectation μ_X and a standard deviation of σ_X. Then for any $t \in \mathbb{R}^+$,

$$\Pr[|X - \mu_X| \geq t\sigma_X] \leq \frac{1}{t^2}$$

Proof Note that

$$\Pr[|X - \mu_X| \geq t\sigma_X] = \Pr[(X - \mu_X)^2 \geq t^2\sigma_X^2]$$

The random variable $Y = (X - \mu_X)^2$ has expectation σ_X^2, and applying the Markov inequality to Y bounds this probability from above by $1/t^2$.

4.3 Disposition

Forslag - bygget på Martin G's noter - måske er Søren D's bedre...

1. Randomised Quicksort
2. Las Vegas, Monte Carlo
3. 1-sided vs. 2-sided
4. Bounded Running Times
5. Markov & Chebyshev's Inequalities
6. Randomised Selection

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 $\Phi(|K|)$.

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: $\Phi(n)$ + 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 $\Phi(1 + \alpha)$, 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 $\Phi(1+\alpha)$, 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

[?]