Examn notes for Advanced Algorithms and Datastructures 2014

Martin Jørgensen

June 10, 2014

Contents

Dispositions

Max-Flow

1. (Define a Flow Network)
    - (Capacity Constraint)
    - (Flow Conservation)
2. Define a Max Flow
3. (How to have multible source/sink networks)
4. (Introduce Residual Networks)
5. (Introduce Augmenting Paths)
6. Cuts - In particular the min cut max flow
7. Introduce Ford-Fulkerson / Edmonds-Karp

Fibonacci Heaps

1. Mergeable heaps

2. Structure

3. Operations

    - Make-Heap
    - Insert
    - ExtractMin
    - Union/Merge
    - DecreaseKey
    - Delete

Wrwite something about $D(n)$ since it's used for all the proofs.

NP-Completeness

1. Polynomial time vs Superpolynomial time (soft vs hard)
2. P-Class problems
3. NP-Class problems
4. Decisions vs Optimization problems
5. Reductions and verifiability/certificates

Randomized Algorithms

1. WHOOP

Hashing

1. WHOOP

Exact Exponential Algorithms

1. WHOOP

Approximation Algorithms

1. WHOOP

Computational Geometry

1. WHOOP

Linear Programming and Optimization

1. WHOOP

Notes

Max-Flow

Flow Network

A flow network $G = (V, E)$ is a directed graph where each edge $(u, v) \in E$ have a nonnegative capacity $c(u, v) \geq 0$. In addition, for any edge (u, v) there can be no antiparallel edge (v, u).

Two vertices in the network have special characteristics the source s and sink t. We assume each vertex $v \in V$ lies on some path from s to t, that is, for each vertex $v \in V$, the flow network contains a path $s \leadsto v \leadsto t$.

Flow Definition

We have a flow network $G = (V, E)$ with a source s and a sink t, the network has a capacity function $c(u, v)$. A flow is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the two following properties:

- **Capacity Constraint**:
  For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$
- **Flow Conservation**:
  For all $u \in V - \{s, t\}$ we require

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

  When $(u, v) \notin E$, there can be no flow from u to v, and $f(u, v) = 0$. We call the nonnegative quantity $f(u, v)$ the flow from vertex u to vertex v.

The value $|f|$ of a flow f is defined as

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

that is the total flow out of the source minus the flow into the source. For an example of a flow see Figure 1

Antiparallel Edges and Multiple Sources/Sinks

Since a flow network cannot contain anti-parallel edges, but we want to be able to represent them in our graph, we need a way to do so. This is done by inserting an additional node v' and let one of the edges go through this node instead, see Figure 2 for an example.

If a network have multiple sources or sinks, we can convert it to a single source/sink network by adding a supersource and supersink. An example of such conversion can be seen in Figure 3.

Flow Examples

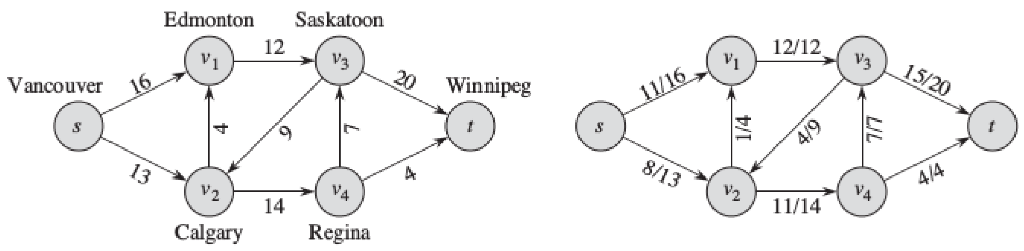

Figure 1: Example flow.

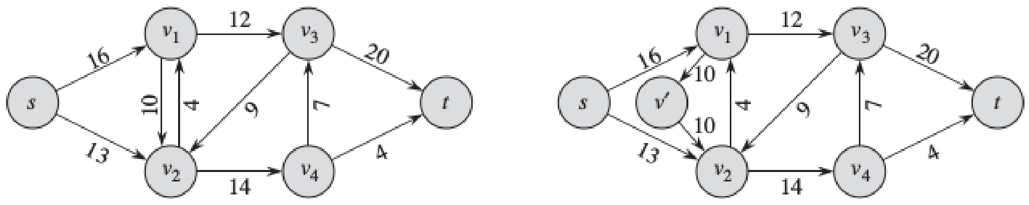

Figure 2: Conversion from antiparallel edges to proper flow.

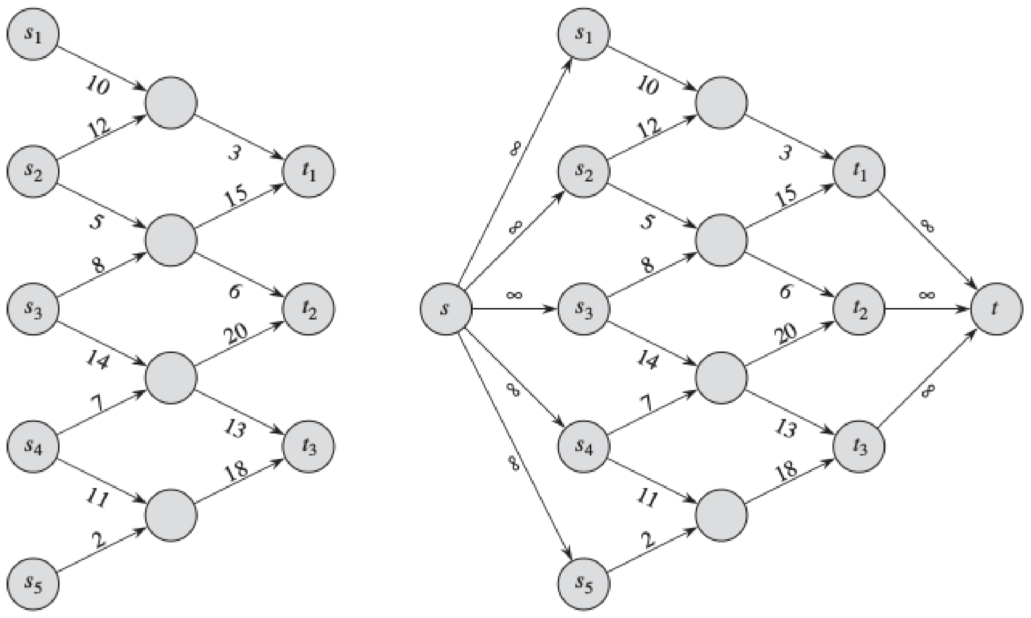

Figure 3: Example of a graph with multiple sources and sink, combined using a supersource and supersink.

Residual Networks

Given a flow network G and a flow f the residual network G_f consists of edges and capacities that represent how we can change the flow on edges of G. Suppose 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 then define the residual capacity $c_f(u, v)$ like this:

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

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

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

An example of a residual network can be seen in Figure 4.

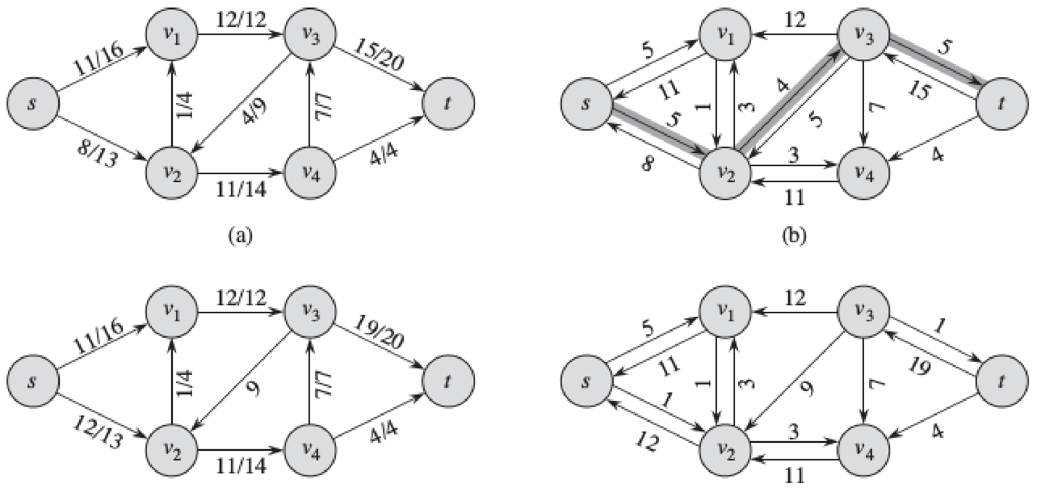

Figure 4: An example of a flow being augmented and showing the residual graph.

Augmenting Flows

Augmenting paths are simply flows that can be added to other flows in order to increase the flow value through the network. Augmenting flows are described using the $\uparrow$ operator like so:

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

an example of an applied augmenting path can bee seen in Figure 4.

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 be induced by f, and let f' be a flow in G_f. Then the function $f \uparrow f'$ is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

Augmenting Paths

Given a network $G = (V, E)$ and a flow f, an augmenting path is a simple path from s to t in the residual network G_f. The shaded path in Figure 4(b) is an augmenting path. We can increase the flow on each edge in the augmenting path p by an amount equal to the residual capacity of p given by

$$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is on } p\}$$

That is, the smallest amount of spare capacity on any edge in p.

Lemma 26.2 let $G = (V, E)$ be a flownetwork, let f be a flow in G, and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbb{R}$ by

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ is on } p, \\ 0 & \text{otherwise.} \end{cases}$$

Then, f_p is flow in G_f with value $|f_p| = c_f(p) > 0$.

Corollary 26.3 let $G = (V, E)$ be a flownetwork, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defined as in Lemma 26.2, and suppose that we augment f by f_p. Then the function $f \uparrow f'$ is a flow in G with value $|f \uparrow f'| = |f| + |f_p| > |f|$.

Cuts in 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 $t \in T$. If f isa a flow, then the net flow $f(S, T)$ across the cut (S, T) is defined to be

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

The capacity of the cut (S, T) is

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

A **minimum cut** of a network is a cut whose capacity is minimum over all cuts of the network. Note that for capacity we count only edges going from S to T while for flow we count edges going both directions. Lemma 26.4 shows that for a given flow f, the net flow across any cut is the same and it equals $|f|$, the value of the flow.

Lemma 26.4 Let f be a flow in a flow network $G = (V, E)$ with source s and sink t, and let (S, T) be any cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$

Corollary 26.5 The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Max-flow min-cut theorem:

Theorem 26.6 If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a mximum 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) $\implies$ (2): If we assume f is a maximum flow but there still is an augmenting path p in the residual graph G_f, then by Corollary 26.3, the flow found by $f \uparrow f_p$ is a flow with value strictly greater than |f|, contradicting the assumption that f is a max flow.

(2) $\implies$ (3): Suppose G_f has no augmenting paths, that is, that G_f contains no paths from s to t. We define

$$S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$$

and $T = V - S$. The partition (S, T) is a cut: we have $s \in S$ trivially, and $t \notin S$ because there is no path from s to t in G_f.

We now consider a pair of vertices $u \in S$ and $v \in T$. If $(u, v) \in E$, we must have $f(u, v) = c(u, v)$ otherwise $(u, v) \in E_f$ wich would place v in S.

If $(v, u) \in E$, we must have $f(v, u) = 0$, because otherwise $c_f(u, v) = f(v, u)$ would be positive and we would have $(u, v) \in E_f$, whouch would place v in S. If neither (u, v) or (v, u) is in E, then $f(u, v) = f(v, u) = 0$. We thus have

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

By Lemma 26.4, we then have $|f| = f(S,T) = c(S,T)$.

(3) $\implies$ (1): By corollary 26.5, $|f| \le c(S,T)$ for all cuts (S,T). The condition $|f| = c(S,T)$ thus implies that f is a maximum flow. □

Ford-Fulkerson

The general algorithm:

```
FORD-FULKERSON-METHOD($G, s, t$)
1  initialize flow $f$ to 0
2  **while** there exists an augmenting path $p$ in the residual network $G_f$
3      augment flow $f$ along $p$
4  **return** $f$
```

Implementation:

```
FORD-FULKERSON($G, s, t$)
1  **for** each edge $(u,v) \in G.E$
2      $(u,v).f = 0$
3  **while** there exists a path $p$ from $s$ to $t$ in the residual network $G_f$
4      $c_f(p) = \min\{c_f(u,v) : (u,v)$ is in $p\}$
5      **for** each edge $(u,v)$ in $p$
6          **if** $(u,v) \in E$
7              $(u,v).f = (u,v).f + c_f(p)$
8          **else** $(v,u).f = (v,u).f - c_f(p)$
```

Assume we can pick the path p in linear time, the loop header is running in $O(E)$. If f^* denote a maximum flow, then the while loop is executed at most $|f^*|$ times, since each augmentation must increase the flow value with at least one. The for loop inside the loop can be done in $O(E)$ since the longest p can be no longer than $|E|$. Giving a running time of $O(|f^*|\dot{(}E+E)) = O(E|f^*|)$.

Edmonds-Karp

Is a Ford-Fulkerson implementation that uses Shortest-Path to find the path p in line 3 of the Ford-Fulkerson algorithm. Each edge is given unit-weight and the algorithm will then pick the shortest path each time. It then has a running-time of $O(VE^2)$.

Lemma 26.7 If Edmonds-Karp is run on a flow network $G = (V, E)$ with source s and sink t, then for all vertices $v \in V - \{s, t\}$, the shortest-path distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.

Proof-ish Intuitively, if we chose a path p from $s \rightarrow v$ that is a shortest path, and then assume there is a path p' from $s \rightarrow v$ which is shorter, we contradict our initial statement that p is a shortest path. This proves that the path length do not decrease. p' might have the same length as p or might be longer. $\square$

Theorem 26.8 If Edmonds-Karp is run on a flow network $G = (V, E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE)$.

Write down and understand the proof...

Fibonacci Heaps

Write something about mergable heaps

Fibonacci heaps are a datastructure that supports a set of operations qualifying it as a “meargeable heap”, meaning it supports the following operations.

- `Make-Heap()` Creates and returns a new heap with no elements.
- `Insert(H,x)` Inserts element x whose key have 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 both heaps. Both heaps are destroyed by the operation.

Apart from the meargeable heap operations above, Fibonacci Heaps also support the following two operations.

- `Decrease-Key(H,x,k)` Assigns to element x in heap H the new key value k, which cannot be greater than it’s current value.
- `Delete(H,x)` Deletes element x from H.

Fibonacci Heaps are by default min-heaps, but could just as well be max heaps, then we would just replace the Minimum, Extract-Min and Decrease-Key operations with Maximum, Extract-Max and Increase-Key instead.

Fibonacci Heaps have a benefit in the fact that many operations are run in constat amortized time. So if these operations are used frequently, the Fibonacci Heap is a well suited structure.

Structure

A Fibonaccci Heap is a collectoin of rooted trees that are “min-heap ordered”. That is, each tree obeys the minimum-heap property: The key of a node is greater than or equal to the key of its parent. A node x contains a pointer $x.p$ to its parent and a pointer $x.child$ to any one of its children. This list is called the

Procedure	Binary heap(worst case)	Fibonacci heap (amortized)
Make-heap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(lgn)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(1)$
Extract-Min	$\Theta(lgn)$	$O(lgn)$
Union	$\Theta(n)$	$\Theta(1)$
Decrease-Key	$\Theta(lgn)$	$\Theta(1)$
Delete	$\Theta(lgn)$	$O(lgn)$

Table 1: Amortized running times of normal binary heaps and Fibonacci Heaps.

child list. Each node also contains 2 pointers $x.left$ and $x.right$, these points to a nodes siblings or to the node itself if it has n osiblings. This forms a circular doubly-linked list called the child list. The nodes may appear in the child list in any order.

Nodes have 2 additional properties, $x.degree$ which is how many children a node have, and a boolean value $x.mark$. $x.mark$ indicates if x has lost a child since x was made the child of another node. Nodes initially have $x.mark$ = False.

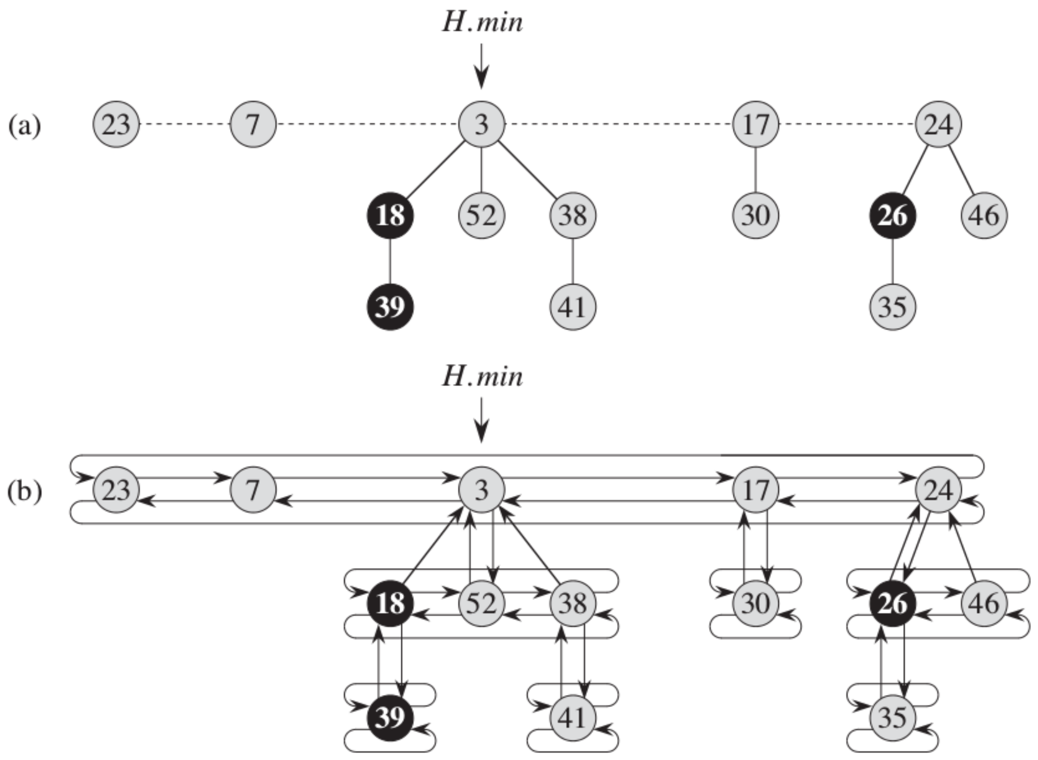

Figure 5: An example of a Fibonacci Heap with(b) and without(a) the child-list.

We access a fibonacci Heap by the pointer $H.min$ which points to the root of a tree containing the minimum key, this node is called the minimum node. If there are several nodes with the smallest key, any of those will work. When the heap is empty, $H.min$ = NIL. The roots of all the tree in a Fibonacci Heap ar linked togeter in a circular doubly-linked list called the "root list". $H.min$ points to a node in the root list. The heap also have one other property, $H.n$ which is the number of nodes currently in H.

Operations

Make-Heap Simply creates a pointer $H.min$ = NIL; since there is no trees in H at this point. This operation can be performed in $O(1)$ time.

Insert Insertion is done on constant time as well.

```
FIB-HEAP-INSERT$(H, x)$
$x.degree = 1$
$x.p =$ NIL
$x.child =$ NIL
$x.mark =$ FALSE
if $H.min ==$ FALSE
    Create a root list for $H$ containing just $x$.
    $H.min = x$
else insert $x$ into $H$'s root list
    if $x.key < H.min.key$
        $H.min = x$
$H.n = H.n + 1$
```

Minimum Just follow the $H.min$ pointer and you're home safe.

Extract-Min This is the most complicated and expensive of the meargeable heap procedures, since it will be doing the work of actually consolidating the trees in the list. Most other procedures put of this work so that it can be done when using Extract-Min. This operation can be done in $O(\lg n)$ time.

```
FIB-HEAP-EXTRACT-MIN$(H)$
$z.min = H.min$
if $z \neq$ NIL
    for each child $x$ of $z$
        add $x$ to the root list of $H$
        $x.p =$ NIL
    remove $z$ from the root list of $H$
    if $z == z.right$
        $H.min =$ NIL
    else $H.min = z.right$
        CONSOLIDATE$(H)$
    $H.n = H.n - 1$
return $z$
```

Notice that $D(H.n)$ here will calculate the upper bound on the degree.

```
CONSOLIDATE($H$)
1  let $A[0..D(H.n)]$ be a new array
2  for $i = 0$ to $D(H.n)$
3      $A[i] = $ NIL
4  for each node $w$ in the root list of $H$
5      $x = w$
6      $d = x.degree$
7      while $A[d] \neq$ NIL
8          $y = A[d]$ // another node with the same degree as $x$
9          if $x.key > y.key$
10             exchange $x$ with $y$
11         FIB-HEAP-LINK($H, y, x$)
12         $A[d] = $ NIL
13         $d = d + 1$
14     $A[d] = x$
15 $H.min = $ NIL
16 for $i = 0$ to $D(H.n)$
17     if $A[i] \neq$ NIL
18         if $H.min ==$ NIL
19             create a root list for $H$ containing just $A[i]$
20             $H.min = A[i]$
21         else insert $A[i]$ into $H$'s root list
22                 if $A[i].key < H.min.key$
23                     $H.min = A[i]$
```

```
FIB-HEAP-LINK($H, y, x$)
1  remove $y$ from the root list of $H$
2  make $y$ a child of $x$, incrementing $x.degree$
3  $y.mark = $ FALSE
```

Union Merging two fibonacci Heaps are done in constant:

```
FIB-HEAP-UNION($H_1, H_2$)
1  $H$ = MAKE-FIB-HEAP()
2  $H.min = H_1.min$
3  Concatenate the root list oh $H_2$ with the root list of $H$.
4  if ($H_1.min ==$ NIL) or ($H_2.min \neq$ NIL and $H_2.min.key < H_1.min.key$)
5      $H.min = H_2.min$
6  $H.n = H_1.n + H_2.n$
7  return $H$
```

DecreaseKey We can decrease the key of any node using this method, it runs in $O(1)$ time.

```
FIB-HEAP-DECREASE-KEY($H, x, k$)
1  if $k > x.key$
2      error new key is greater than current key"
3  $x.key = k$
4  $y = x.p$
5  if $y \neq$ NIL adn $x.key < y.key$
6      CUT($H, x, y$)
7      CASCADING-CUT($H, y$)
8  if $x.key < H.min.key$
9      $H.min = x$
```

```
CUT($H, x, y$)
1  remove $x$ from the child list of $y$, decrementing $y.degree$
2  add $x$ to the root list of $H$
3  $x.p$ = NIL
4  $x.mark$ = FALSE
```

```
CASCADING-CUT($H, y$)
1  $z = y.p$
2  if $z \neq$ NIL
3      if $y.mark ==$ FALSE
4          $y.mark =$ TRUE
5      else CUT($H, y, z$)
6          CASCADING-CUT($H, z$)
```

Delete Since it uses two previusly defined functions, the amortized running-time is easily calculated to $O(\lg n)$.

```
FIB-HEAP-DELETE($H, x$)
1  FIB-HEAP-DECREASE-KEY($H, x, -\infty$)
2  FIB-HEAP-EXTRACT-MIN($H$)
```

Wrwite something about $D(n)$ since it's used for all the proofs.

NP-Completeness

Example of a problem in the P class is an Euler Tour(a path in a graph that uses all edges exactly once, vertices can be visited multible times) of a graph, it can be done in $O(E)$. An NP class example that is very similar is a Hamiltonian cycle. A Hamiltonian cycle is a path that visits all vertices once.

We have three classes of problems in this subject:

P Problems that are solvable in polynomial time ($O(n^k)$ for some constant k.)

NP Problems that are verifiable on polynomial time, i.e. if we have a certificate/solution, can we check it in polynomial time. All problems in P will also be in NP.

NP-Hard A subclass of NPC problems that are “at least as hard as the hardest problems in NP”, these cannot be verified in polynomial time.

NPC Problems that are both in the set of NP problems and NP-hard problems. (NP $\cap$ NP-Hard)

Decision problems vs. optimization problems

NP-completeness does not cover optimization problems, only decision problems. We can however use the relationship between optimization and decision problems to guage if a optimization problems is in fact NP-complete.

The shortest-path problem is an opimization problem, but can converted (in polynomial time) to a decision problem if the question is posed like so: “Does a path p in the graph G exist with only k edges?”, then we iterate over k and will be able to guage the shortest bath problem as a dicision problem.

Reduction

The notion of showing that one problem us no harder or no easier than another problem applies even when both problems are decision problems. This is used in almost all NP-Completeness proof as follows: Take an instance α of problem A, that is, a specific input for the problem A, so for shortest path we may choose a graph G, and vertices u and v as well as a k. Make a polynomial time transofmration from α to instance β of problem B (which can be decided in polynomial time) with the following characteristics:

1. Transformation takes polynomial time.
2. The answers are the same. The answer for α is "yes", iff. the answer for β is "yes", same for "no".

Such an algorithm is called a reduction algorithm.

We can now solve any instance of A in polynomial time by converting α to β in polynomial time, running the polynomial time decision algorithm for B and using the answer for B as the answer for A.

Continue from p. 1052-3 something.