Better Text Formatting For the Mobile Web With Javascript

Felix Angerborn

LIU-IDA/LITH-EX-A--14/047--SE
Linköping 2015
Better Text Formatting For the Mobile Web With Javascript

Felix Angerborn

LIU-ID/ID-EX-A--14/047--SE
Linköping 2015

Supervisor: Anders Fröberg
IDA, Linköpings universitet
Magnus Gasslander
Opera Software ASA

Examiner: Erik Berglund
IDA, Linköpings universitet
Summary

This thesis investigates the possibility of using a better algorithm for text formatting on the web, especially for mobile devices. For this purpose, the Knuth & Plass algorithm for breaking paragraphs into lines has been implemented in Javascript.

Formatting today is done well in magazines and books, but on the internet it is very rarely good because a better formatting algorithm is not currently supported by browsers. The formatters used there is generally simple greedy algorithms that stuff words on a line until the line is full, then moving on to the next line. Such algorithms leaves no guarantee on how good the result is.

A better kind of formatter is the Knuth & Plass algorithm which, simply explained, minimizes the sum of the square of difference between the wanted line length and the actual line length. With this, and a special threshold value that restricts lines to be too short or too long, the resulting formatting from this algorithm is usually very good.

Tests of the implementation shows that the algorithm takes about 55 ms to format 200 words on a Samsung Galaxy S4. Javascript profiling tools showed that almost all of this time was spent on measuring the words of the text. Optimization of the measurements shortened this time to 5 ms for 200 words, a 10x improvement! Although the improvement methods are not always useable. Other potentially performance-improving changes were applied but had little to no effect on the algorithm.

Comparing the implemented algorithm to others available in browsers shows clearly that the resulting formatting is better. With the improved speed of measuring this algorithm is fast enough to be used even on most mobile devices, coupled with the better formatting a smarter algorithm it is definitely something that can or even should be used.
Abstract

As people read more and longer texts on the web, the simple formatting options that exists in todays browsers creates worse results than necessary. On behalf of Opera Software in Linköping, a better algorithm has been implemented in Javascript with the purpose of delivering a visually better experience for the reader. The implementation is first and foremost for mobile devices and therefore a large part of the thesis has been the evaluation and optimization of performance.

Keywords: Javascript, Line breaking, Formatting, Formatter, Optimization
# Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coffeescript</td>
<td>A programming language that compiles in to Javascript</td>
</tr>
<tr>
<td>Complexity</td>
<td>A way to measure how much work an algorithm needs to do, usually measured in amounts of input to the algorithm.</td>
</tr>
<tr>
<td>Caching</td>
<td>Storing earlier results so that they can be retrieved quickly when needed.</td>
</tr>
<tr>
<td>Dynamic programming</td>
<td>A method in programming</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>Formatter</td>
<td>A program that formats text</td>
</tr>
<tr>
<td>Javascript</td>
<td>A programming language mostly used for the web</td>
</tr>
<tr>
<td>Knuth &amp; Plass algorithm</td>
<td>The algorithm implemented and evaluated in this thesis. Developed by Donald Knuth &amp; Michael Plass during the 1970’s</td>
</tr>
<tr>
<td>Mobile device</td>
<td>Smaller device such as a mobile phone or a tablet, laptops are generally not included.</td>
</tr>
</tbody>
</table>
# Table of contents

Summary ............................................................................................................. 4
Abstract .............................................................................................................. 5
Glossary .............................................................................................................. 6

Chapter 1  Introduction ..................................................................................... 9
  Background ..................................................................................................... 9
  Previous work ............................................................................................... 10
  *Breaking paragraphs into lines* ................................................................. 10
  SMAWK ......................................................................................................... 10
  Others ........................................................................................................... 10

Purpose ............................................................................................................ 10
Problem description ......................................................................................... 11
Thesis structure .............................................................................................. 11

Chapter 2  Algorithms ..................................................................................... 12
  Choosing algorithm ...................................................................................... 12
  The Knuth & Plass algorithm ...................................................................... 13
    Definitions ................................................................................................. 13
    Problem insight ......................................................................................... 14
    The algorithm itself ................................................................................... 16
  Other algorithms .......................................................................................... 19

Chapter 3  Performance ................................................................................... 21
  Analyzing complexity .................................................................................. 21
  Conclusions .................................................................................................. 22
  Javascript ...................................................................................................... 23
  Measuring ..................................................................................................... 24
  Other improvements ..................................................................................... 31
  Mobile ........................................................................................................... 33

Chapter 4  Results ............................................................................................ 35

Chapter 5  Conclusions ................................................................................. 38

Chapter 6  Discussion ...................................................................................... 39

References ....................................................................................................... 40

Copyright .......................................................................................................... 42
Chapter 1  Introduction

This introductory chapter serves the purpose of introducing the reader to this thesis, what formatting is and why it is an important part to any literature, printed or not.

The subchapters is:

- **Background** – Covers what the formatting problem is, when it emerges and how this thesis will work with it.
- **Purpose** – The purpose of this thesis.
- **Previous work** – A lot of work have been done on formatting and here only the most important parts are mentioned.
- **Problem description** – Describes further the problem of line breaking and why it can be such a hard problem to solve.
- **Thesis structure** – Overlay of the structure of the thesis.

**Background**

Breaking a paragraph into lines is a problem that emerges in almost any text, when it is done sufficiently well it usually goes unnoticed. On the other hand, when it is badly done it is easily noticed and may be a distraction to the user. In newspapers and books this is in most cases not a problem and it is done well, on the web easier and faster solutions are used and these may very well be considered bad algorithms when the resulting formatting is in focus.

As the web is very different from printed literature it is natural that the used formatting may differ. But when people nowadays read more and longer texts such as e-books and newspapers, a better formatting option would be a very welcome addition.

Bad formatting is an easy problem to solve, but good formatting is a much harder problem that requires more thought. Formatting has been an area of research in computer sciences since long and there exists several algorithms for it. The most well known of which is the Knuth & Plass algorithm that is also used in this thesis. It is also one of the only formatting algorithms that is easily accessible from a paper and not patented.
Previous work

Breaking paragraphs into lines
The paper on which most of the algorithmic work in this thesis is based on is written by Donald Knuth and Michael Plass during the 1970s. The algorithm they invented is very much still relevant in the area and is one of the most used as it is the formatter that is used in TeX. It is a very fast algorithm and uses what Knuth likes to call discrete dynamic programming, which is what makes it that fast.

A paper was written solely on the purpose to describe their algorithm, that paper is called Breaking paragraphs into lines.

SMAWK
Since Knuth and Plass wrote their paper, other discoveries has been made in text formatting. One of which is the SMAWK algorithm, named after the lastnames of its inventors. It is originally an algorithm for finding the minimum value of each row in a totally monotone matrix, but can supposedly be used to find the best formatting for a text, in linear time.

Others
Many other have implemented algorithms for formatting, whether it be the Knuth and Plass algorithm or something of an own concoction, formatting is applied anywhere were text resides. It is not a unique problem after all and can be solved in many ways, though not any as effective as the Knuth and Plass or the SMAWK algorithms, yet.

Purpose

The work for this thesis is a proposition from Opera Linköping, which has found a need for improved text formatting in some parts of their iOS/Android browser applications. The parts of these apps that needs better formatting displays static text and images inside HTML pages.

The purpose of this thesis is to:

- Implement a formatting algorithm that yields a better result than the ones available in browser, the algorithm is to be based on the Knuth and Plass algorithm.
- Analyze the performance of the implementation to determine if it is possible to use the algorithm on mobile devices.

As the implementation needs to work in apps on both iOS and Android, it will be made in Javascript that easily can be run in both applications.
Problem description

Consider a normal text that needs to be broken into lines, a text consisting of words with spaces between them. Any position in the text where the current line may be ended and a new begun is called a potential breakpoint. Potential breakpoints are in most of cases all of the spaces and the possible hyphenation locations in the text, in this thesis we shall not consider possible hyphenations because good hyphenation is a problem entirely of its own.

It is also important to know how the text is supposed to be formatted, with only the text there is no formatting that can be done. This can be decided beforehand or be dependent on the width of the element supposed to contain the text. In either case, this width is to be given to the algorithm with the text so that it can fit the text appropriately. Usually it is a list of widths, they may all be the same or they can be of different sizes as for example to fit the text around an image or an aside.

Given a way to compare and rate the different solutions it is possible to find the optimal choice of breakpoints. How to compare lines and rate solutions is part of the algorithm, and how this is done in this thesis is described in chapter 2.

The purpose of the algorithm is to find the best choice of breakpoints so as to fit the text in to the wanted line lengths as good as possible. It is easy to get fooled by the simplicity of the problem, but the number of different formatings is equal to $2^k$ where $k$ is the number of possible breakpoints in the text. This is an extremely high number for any text of substantial length and evaluating all of the possible formattings is to slow even for the fastest computer.

Thesis structure

To give the reader the best experience from this document, it is written to reflect the workflow of this thesis. This is the structure.

- Chapter 2 mentions different algorithms and how to chose between them. It also goes in to more detail on the specific algorithm evaluated in this thesis.
- Chapter 3 analyzes the performance of the algorithm and what parts of the implementation that have been optimized and how this have been done.
- Chapter 4 shows the results of the implementation and compares it to the simpler available algorithms in browsers.
- Chapter 5 draws conclusions on whether the implementation is useable or not, when to use smarter algorithms and how to make them useful on the web.
- Chapter 6 discusses further improvements of the implementation and its uses.
Chapter 2  Algorithms

Chapter 2 will cover the algorithms of the thesis. The contents of this chapter in summary

- **Choosing algorithm** – How the proper algorithm for this thesis work was chosen. This was already decided before the work started, but the reasons as to why it was chosen is mentioned here anyway. The algorithm is great for static text and descriptions of it is easy to access in publications by Knuth.
- **The Knuth & Plass algorithm** – A subchapter only about the implemented algorithm. Describes how to get a grasp of the subject by thinking of it as a graph, why it is possible to solve with a shortest path algorithm and why that is not a preferred method. Finally a brief explanation about the structure of the algorithm and how it actually works.
- **Other algorithms** – Mentions a few other algorithms, a greedy one and an improved greedy. Analyzes the weak and strong points of these and compares to the Knuth & Plass algorithm. There are many differences but they are both useable in different scenarios.

After reading this chapter the reader should have a grasp on one way to think around the formatting problem along with a basic understanding of how different algorithms works to solve it.

**Choosing algorithm**

As the number of possible formattings grow exponentially an algorithm has to either be efficient enough to find the best formatting without evaluating them all or not care whether the solution is optimal or not. In browsers today, the latter version is applied, they are using a kind of algorithm that finds a solution very fast but does in no way guarantee the solution to be any good.

The chosen algorithm for this thesis, the Knuth & Plass algorithm, yields a very good result in most cases and is superior to simpler algorithms because it can look forward and take decisions that leads to a good position later. It minimizes raggedness in the margins of unjustified text and keeps the space width close to the font standard in justified text. Even though the optimal solution is usually much better, it is very seldomly a perfect fit and still results in a slightly uneven margin and altered space widths.

The algorithms all have their strong and weak points, including but not limited to performance and the resulting formatting. The best algorithm for a certain scenario depends on a number of things and later in this chapter important differences will be mentioned.

In this thesis, it was a requirement to base the work on the Knuth & Plass algorithm. Even without the requirement this is the algorithm that would be chosen because of its availability and the excellent papers on it.
The Knuth & Plass algorithm

The Knuth & Plass algorithm is one of the only openly available and one of the most known algorithms available for text formatting. As the name gives away, it was developed by Donald Knuth and Michael Plass. Knuth is one of the big figures in typography and developed a typesetting system called TeX that uses a much larger and more advanced version of the algorithm mentioned here.

The algorithm is of the type dynamic programming. This means that it makes use of the optimal substructure of the formatting problem. What this means for the algorithm is that many solutions can be discarded before they are looked further in to, which in turns makes the algorithm much more efficient.

Definitions

The algorithm assumes that the text to be formatted is represented as a list of text objects. This list consists of three different types of objects that all represents a part of the text, these are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glue</td>
<td>Whitespace, used mainly between words but can also be used as paragraph indentation. Each glue holds three values. Width that is how wide this space is without modification. Stretchability and shrinkability which is how much this space may be shrunken or stretched.</td>
</tr>
<tr>
<td>Box</td>
<td>Usually the words and other parts of visible text, holds their own width and the text they represent.</td>
</tr>
<tr>
<td>Penalty</td>
<td>These represent especially good or bad places to break for a new line. The higher the penalty is, the worse it is to make a break here. Usually these are used to force a break at a location by making it infinitely good, or the exact opposite to make a location extremely unfit for a breakpoint. Forced breaks are always used as the last object in the list to end the paragraph.</td>
</tr>
</tbody>
</table>

The structure of the list for a somewhat normal text is to start with a box or a glue depending on if the paragraph shall have the first word indented or not. Then follows alternating boxes and glues until the list ends with a forced break.

As is already known, the algorithm also needs a list of the lengths of the lines. In this implementation it is chosen to give this as a simple list of integers to the algorithm, if varying line widths are not needed then it of course suffices with a simple integer. In this implementation there is a need to handle varying line widths and therefore the algorithm always expects a list.

The algorithm can also take a threshold value \( p \) that represents an upper limit on the adjustment ratio of any line. This means that it restricts how bad of a fit for a line is acceptable.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demerits</td>
<td>Badness value, used by the algorithm to determine which formatting is the best. The formatting which has the least demerits is considered to be the best. Mostly it is the difference between actual and wanted line lengths that contributes to the demerit value.</td>
</tr>
<tr>
<td>Active Node</td>
<td>A text object that has been chosen to be used as a breakpoint recently. All chosen nodes are to be active until the distance between the currently examined node and it is so long that it can not be considered as a line.</td>
</tr>
<tr>
<td>Adjustment Ratio</td>
<td>The adjustment ratio is created as a measurement of how good or bad a line fits in with its expected line length. We shall see that this is very useful to keep track of the fitting and it also easily lets us compare different sets of breakpoints. Let $J$ be the expected line width, $L$ the actual line width, $Y$ is the total sum of stretchability for this line and $Z$ is the total sum of shrinkability. Then we define the adjustment ratio $r$ as follows:</td>
</tr>
<tr>
<td></td>
<td>If $L = J$ (This line fits perfectly) then $r = 0$</td>
</tr>
<tr>
<td></td>
<td>If $L &lt; J$ (Shorter) then $r = (J - L)/Y$, $Y$ must be $&gt; 0$ otherwise $r$ is undefined in this case.</td>
</tr>
<tr>
<td></td>
<td>If $L &gt; J$ (Longer) then $r = (L - J)/Z$, $Z$ must be $&gt; 0$ otherwise $r$ is undefined in this case.</td>
</tr>
<tr>
<td></td>
<td>With this definition an absolute value on $r$ larger than 1 would mean that there is not enough shrink- or stretchability on this line to adjust this line from the actual width to the expected width without making the whitespaces either narrower or wider than acceptable. We will later make use of a simplified version of the adjustment ratio to compare results from our implemented algorithm with other typesetting algorithms.</td>
</tr>
</tbody>
</table>

**Problem insight**

A way of gaining understanding of the problem of line breaking is to imagine the text as a graph where nodes represent possible breakpoints and edges represent a line. An edge between two nodes $a$ and $b$ means that there is a line starting at and including $a$ that spans up to but not including $b$. An edge can naturally only exist between $a$ and $b$ if $a$ appears before $b$ in the text, otherwise the edge could hold no meaning to it. With a graph like this the problem can be formulated as finding the best path from the first to the last breakpoint. The type of graph created for this problem is a **Directed Acyclic Graph**, or DAG.
In the image above is a simple example of how such a graph could look like for a very short sentence. The numbers represents at which position each node is in the original text, and the edges describes that the text is broken in to a total of three lines.

Analyzing a graph of breakpoints is is possible to find an approximate number of eligible ways to break up that text into lines. Let $k$ be the amount of breakpoints in the text that can be chosen, then the total number of possible paths would be $2^k$, each breakpoint can either be included or excluded so each breakpoint presents a choice between two options. To show how fast this grows, a text of 10 legal breakpoints would have $2^{10}$ or 1024 different choices and a text of 100 legal breakpoints would have $2^{100} \approx 1.27 \times 10^{30}$ choices. Of course most of these choices are ridiculous and by introducing the adjustment ratio it becomes easy to separate many of these bad choices from the good ones.

Forced breaks can be ignored during this approximation and they add no extra paths since they always have to be included.

To find the actual best path there needs to be some weight on the edges, and the most important part is that the lines are as close to the wanted length as possible. So it makes sense to weight the edges according to this. Edges are therefore weighted by the square of the absolute value of the adjustment ratio that the row would have, meaning that the algorithm punishes rows hard for being longer or shorter than expected.

**text: aaa bb cc ddddd**

**line length: 6 characters**

```
  aaa bb 0 aaa 3
  cc   4 bb cc 1
  ddddd 1 ddddd 1
```

*Figure 2: Example of squared punishment*
In the example a text is to be formatted to fit on lines of six characters width, to the left is the result of a simple greedy algorithm appending each word on a line changing lines when a word can not fit. On the right is the result from trying to minimize the squared difference between actual and wanted length. The numbers shows the difference for each line.

In this example it is clear that the resulting formatting from minimizing squares is better, although it does not always show this clearly it is generally the case. An example of the actual algorithm compared in a similar fashion on a longer text to the chrome native formatting is shown under chapter 4.

The best set of breakpoints is the one where the sum of the paths edges from start to finish is the smallest while no adjustment ratio exceeds the threshold. At first sight this appears to resemble the well-known shortest path problem in a directed acyclic graph. Although, we also have to take in account that there might be different line widths for different line numbers and because of this the weight function on the edges will also have to depend on the current line number. Still, this may very well be solved by a shortest path algorithm if it is possible to modify it to handle adjustment ratio thresholds and the dynamic edges of the graph.

**The algorithm itself**

Even though the problem is solvable by using some kind of shortest path algorithm, a naive implementation would be very inefficient because of the many solutions to evaluate. In most cases the implementation would be rendered almost useless because of this. Clearly a faster solution is required and in this part of the thesis, Knuths algorithm will be quickly introduced and explained.

In fact, Knuths algorithm is a modified shortest path algorithm that takes advantage of the specific problems structure to utilize discrete dynamic programming. What is meant by discrete dynamic programming in the case of Knuths algorithm is to use the optimal substructure of the problem to prune the tree of possible solutions, resulting in the need to search much fewer branches.

Consider two different sets of breakpoints; these are incomplete but end at the exact same breakpoint for now. This means that some part of the text is already formatted in two different ways a and b and they have the exact same part of the text left to typeset, lets call it the rest. If the rest is to be typeset optimally in both a and b then it would mean that it is typeset exactly the same and resulting in the same sum of demerits for a and b. Given this information, the one of a and b with the lower amount of demerits before formatting the rest will still have the lowest amount of demerits.

The explanation of the algorithm in this part is quite brief but should result in the reader understanding the important parts of it. To start, simplified pseudo code of the algorithm:
Create an active node representing the starting point
for each breakpoint b:
  initialize the feasible breaks at b to the empty set
  for each active node a:
    calculate the adjustment ratio r
    if r < -1 or b is a forced break:
      remove a from the list of active nodes (deactivating it)
    if r >= -1 and r < p
      record a feasible break from a to b
    if there exists any feasible breaks at b:
      find the best of the feasible breaks and append as active nodes
  choose the active node with the least total demerits, this is the best formatting

Figure 3: Simplified pseudo code for the algorithm.

The active nodes are essentially chosen breakpoints containing information about the last node before it. Several active nodes can reference the same earlier node as its parent, this creates a tree that is traversable from child and upwards to the parent of the node and to its parent and so on. Traversing the tree this way backwards from the last breakpoint is how the algorithm reconstructs the breakpoint set of the best found solution.

The tree of nodes needs a root and that is why the first part of the algorithm is to create that root, an active node that has no parent of its own. Every node is created as active, which in the implementation means that it belongs to a special list where all active nodes are stored. A node is set to be inactive when a breakpoint is examined where the distance between the breakpoint and the active node is so far that the adjustment ratio would exceed the threshold. This means that the former active node is removed from the list of active nodes.

After the initial root for the active nodes is created every possible breakpoint is examined in order of appearance in the text. Every examined breakpoint has a list of the so-called feasible breaks at this breakpoint, a feasible break is a possible connection between this breakpoint and an active node. For every breakpoint all active nodes will be matched against it to see if it is possible to break the line at this breakpoint, given that the last breakpoint was the active node. If the adjustment ratio is between -1 and p this breakpoint and the belonging active node is decided to be a feasible break. A feasible break contains information such as the total demerits and the current line number. After all active nodes has been looked at for a breakpoint there is either zero or more feasible breaks in the list, the one of these with the lowest amount of demerits is chosen to become an active node and is added to the list of active nodes, the other feasible breaks are discarded.

Because of the optimal substructure of the problem the feasible breaks that do not have the lowest amount of demerits can be discarded. The reason behind this discarding is that explained earlier in this chapter. A feasible break with higher demerits is not possible to generate a better formatting later, because they have the exact same text left to format.

When handling varying line widths, one feasible break is chosen for each of the line numbers that are currently in the active nodes list. This is because it is possible for a feasible break with higher demerits to end up with a better formatting given that they do not share the same current line number.
By repeating this step for each and every breakpoint the algorithm will eventually reach the last breakpoint, a forced break. It is important that the last break is forced because otherwise only solutions with an adjustment ratio between -1 and the threshold $p$ would be acceptable. Usually it is fine if the last line of a paragraph does not cover its full length, and in this case the algorithm does not bother to justify it. An exception like this is very common in formatting texts and does not interfere with the reading.

When the last breakpoint is reached, a forced break, the active node with the least demerits is chosen to be the best possible solution. Some implementations chose to punish last lines that are very short in which case some demerits would be added depending on the length of the last line. In this implementation it is not punishable for a solution to have very short lines, instead it is considered more important that the earlier done formatting is as good as possible.
Other algorithms

As have been mentioned earlier, it is definitely possible to solve the problem of line breaking with other algorithms than the Knuth & Plass. The easiest example of such an algorithm is a greedy algorithm.

Pseudo code for such an algorithm can look something like this:

```
Greedy:
for each word w
  if w makes the current line longer than acceptable
    break and create a new line
    append w to a new line
  else
    append w to the current line
```

*Figure 4: Pseudo code of a greedy formatting algorithm*

This algorithm is obviously not as complex as the Knuth & Plass algorithm, it will find a solution, but very rarely the optimal one. To analyze the complexity of this algorithm the important part to take note of is that this algorithm only will traverse the list of words once, make a decision between two cases and then be done with the formatting. Because of its simplicity this algorithm is very fast at a linear complexity and has no overhead. This is known as a first-fit algorithm.

Although it has its upsides being a very fast and trivial to implement algorithm the resulting formatting is usually bad. It is possible to improve the algorithm by some easy to make additions. A modified greedy algorithm can look something like this:

```
Greedy shuffle:
for each word w
  if w makes the current line longer than acceptable
    break and create a new line
    if the last line was not considered filled enough
      take words (or partial words by hyphenation) from the previous line to fill out
      append w to the new line
    else
      append w to the current line
```

*Figure 5: Pseudo code of a greedy formatting algorithm with shuffle*

This version of the greedy algorithm is called a shuffle because of its property to shuffle a word forward to the next line.

One might notice that this algorithm still is not capable of returning the optimal solution and it is true that this is also a first-fit algorithm. The strength in a solution like this is only part performance, it is quicker than Knuth-Plass which is good but it also results in an often much better formatting than the simple greedy solution.

The most important part of why algorithms like the two just described might be preferred over a more advanced algorithm such as Knuth & Plass is when the formatter is used in an interactive
environment such as a text field in a browser or a writing program like word. It is imperative to the user experience that the insertion point stays steady during writing. As the Knuth & Plass algorithm takes in consideration the whole paragraph while formatting adding and/or changing a word might rearrange the whole paragraph resulting in the insertion point moving around. A formatter working like that would be unusable in such a case and therefore it would definitely be better to have a simpler algorithm such as the greedy or the shuffle when the text is to be edited.

Depending on the intended use for the formatter the algorithm that should be used can definitely differ. As the use of the algorithm in this thesis work is to format longer texts and news articles that should not be changed it is more important that the result of the formatting is better. And of course performance is always very important since a paragraph cannot be displayed until it has been fully formatted, displaying a delay for the user. Usually performance issues caused by this is not a huge problem because paragraphs are in most cases quite short and texts can be split up in paragraphs before formatting, making it possible to format one paragraph at a time.
Chapter 3  Performance

This chapter analyzes the performance of the implemented algorithm, optimizes the implementation and reasons about how good it is. The subchapters in summary

- **Analyzing complexity** – Analyzes the complexity of the Knuth & Plass algorithm, it is found to be almost linear in many cases and quadratic in the amount of words in the worst case. Not much is to be done about optimizing the algorithm and it is already good as is.
- **Javascript** – Tests the performance of the implementation, finds it already fast but that measuring words is a huge bottleneck. After optimizations the algorithm runs about ten times faster on a Samsung Galaxy S4, the optimizations and their deficits are also covered. Many recommended tries trying to optimize with small code changes showed to yield no improvement.
- **Mobile** – Reflects on what is special about using the implementation on mobile devices. Main differences are that they are slower and have much smaller screens, reducing the effectiveness of the algorithm but definitely not leaving it worthless.

After the work in this chapter, the algorithm is good-to-go on most devices and should be fast enough that no human could notice when the formatter is running.

Analyzing complexity

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Amount of words per line</td>
</tr>
<tr>
<td>n</td>
<td>Amount of words</td>
</tr>
<tr>
<td>bp</td>
<td>Amount of possible breakpoints</td>
</tr>
</tbody>
</table>

From the descriptions of the algorithms in the last chapter it should be quite easy to deduce that the Knuth-Plass algorithm is slower than the greedy versions. Finding a complexity for the greedy algorithms is easy as they only loops through each word once which gives the complexity $O(n)$. The observant also notices that the forward shuffle actually might go back and move a previous word a line forward, this would make the actual complexity $O(2*n)$ which is considered the same as $O(n)$.

In the case of the Knuth-Plass algorithm it is harder to determine the complexity because of the algorithm in itself being much more complex. However, one might notice that for each breakpoint the algorithm searches through all of the active nodes. Since the algorithm loops over the breakpoints only once, this would give a complexity equal to $O(bp*a)$. In the absolute worst case the list of active nodes will be the amount of breakpoints already examined and this would give a complexity of $O(bp^2)$, but in the general case the amount of active nodes are the amount of words that fit on a line. So the complexity of the algorithm depends on the line lengths, or rather, the amount of words that fit on a line. With this in mind the algorithm shall be defined to have a
complexity equal to $O(a^*b^p)$. As in most cases $b^p \approx n$ it would be easier to compare if we say that the complexity is $O(n^*a)$.

**Conclusions**

In terms of performance the Knuth & Plass algorithm is not too bad compared to the other algorithms mentioned. With the computational power that exists in computers and mobile devices today, an algorithm with time complexity of $O(a^*n)$ should be viable for formatting most texts fast enough that the user does not notice it. In cases where there are an absurd amount of text that has to be formatted all at the same time and cannot be split into shorter texts before formatting, it might be better to use a faster algorithm such as the linear ones. This should be a very rare occurrence.

The algorithm in itself is already efficient and there is not much improvement to be done without removing parts of the functionality. For example, one improvement that can be made is to not handle lines of varying widths which would make it both faster and slightly easier to implement. However, the difference in performance is not very noticeable and the extra effort required is small enough that there is very little to gain from it, as it is also very much needed in this implementation it was never an option to leave it out for extra performance.

The time complexity of an algorithm is something that always comes to mind when it comes to efficiency and is in most cases a determining factor in whether an algorithm is usable or not. The complexity that we've arrived at could almost be considered linear in many cases where the line lengths are quite short such as they usually are on mobile devices.

We shall see in the next chapter how fast this implementation really is and if it is viable for use on mobile devices.
Javascript

To measure the performance of the formatting program on mobile devices a simple html page was created that runs the algorithm a given amount of times and displays the results in an easy to understand way. As the first implementation of the non-optimized algorithm was finished it was tried on a Samsung Galaxy S4 (SG4). This is the result.

Words: 200
Total time: 55.7239820001123
Prepare time: 53.5339450009017
Prepare % of total: 96.06984834641266

Figure 6: Non-optimized algorithm on Samsung galaxy S4. Blue represents the time taken for the preparations.

As can be seen in the image the algorithm was run on a text 200 words long, this can be considered to be a quite long paragraph as paragraphs are generally in the length of 5-6 sentences. The times shown are mean values of the runs and shown in milliseconds. The above chart shows that the total time to run the algorithm on 200 words was 55 ms and out of these the preparation method that prepares the text to be handled by the actual algorithm took 96% of the times.

As it seems the algorithm is quite efficient and even though the preparation steps takes a long time, the formatter could be used on texts in the range of a few hundred words without problems, assuming the mobile executing had similar or better hardware than the SG4. Though, on longer texts and slower devices it might still be a visible delay to a human which could be disturbing and/or annoying to the user.

To improve the performance the algorithm is run on a desktop computer to take advantage of the great Javascript profiling tools that are available.
Output from running the not yet optimized algorithm in chromes profiler yielded this:

<table>
<thead>
<tr>
<th>Function</th>
<th>Self (%)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>get clientWidth</td>
<td>62.06%</td>
<td>62.06%</td>
</tr>
<tr>
<td>set innerHTML</td>
<td>19.86%</td>
<td>19.86</td>
</tr>
<tr>
<td>idle</td>
<td>10.77%</td>
<td>10.77</td>
</tr>
<tr>
<td>findbreaks</td>
<td>2.92%</td>
<td>3.06%</td>
</tr>
<tr>
<td>javabreak.breaklines</td>
<td>1.76%</td>
<td>88.34%</td>
</tr>
<tr>
<td>javabreak.prepare</td>
<td>1.32%</td>
<td>83.31%</td>
</tr>
<tr>
<td>garbage collector</td>
<td>0.40%</td>
<td>0.40%</td>
</tr>
<tr>
<td>program</td>
<td>0.31%</td>
<td>0.31%</td>
</tr>
</tbody>
</table>

Figure 7: Data from chromes profiler running the algorithm

Figure 8: Same data shown in ms

This is the results from running the algorithm in chromes profiler with the first implementation of the algorithm. The test is the same as mentioned before, 200 words 1000 times. The total column shows how much percent of time is consumed by calls to the function in the function column in total, including calls to other functions in it whereas self excludes subcalls to other functions. Generally it is most rewarding to look in the self column to find out where the code spends most of its time.

In this particular case the results shows that the top 2 time consuming functions together eats a whopping 82% of the algorithms run time, it would indeed be a good choice to start here to increase the efficiency of our algorithm.

The most time consuming is the call to clientWidth which is a property of an element in the page that corresponds to the inner width of the element in pixels. The second most time consuming function is set innerHTML which is used to set the inner html contents of an element of the page. Both of these are only used in the method that is used to measure the actual width of a word after it has been rendered in the browser. As it seems, the algorithm is not the slow part of the current iteration but rather the way that we find the width of a string.

**Measuring**

The first implementation of a measuring method used for the algorithm measures a text by placing it inside a div element using set innerHTML and then measures it with the .clientWidth property on the div element.
var measure_method = function(text) {
    measureDiv.innerHTML = text;
    var w = measureDiv.clientWidth;
    return w;
};

A very concise method, one would think that this should be quite fast. However, setting the innerHTML and getting the measurement from clientWidth are both very slow procedures because they communicate with the DOM. Communication with the DOM from Javascript is very time consuming and as of now this communication is done several times for each word in the text to be formatted.

There are different ways to set the inner contents of an element such as appendChild and removeChild, although these also comes with the same problems that innerHTML does and communicates with the DOM and therefore are not an improvement. Unfortunately there exists very few ways in which one can measure the length of a string in the browser and therefore there are not many possible improvements to be done.

One way of improving the time spent for communication with the DOM is to cut down the amount of calls to innerHTML and clientWidth, but since it is imperative to know the length of each word how could one minimize these calls? An idea is to cache each string that is measured so that if the same string is to be measured again, instead of working with the DOM the answer can be retrieved from a cache nearly instantaneously.

An implementation of a measure method using a cache could look something like this

```javascript
var cache = {};
var cache_words = function(text) {
    var w = cache[text];
    if (w !== undefined) {
        return w;
    }
    measureDiv.innerHTML = text;
    w = measureDiv.clientWidth;
    cache[text] = w;
    return w;
};
```

*Figure 9: An implementation of a caching measuring method*
And this exact method yields this data from the profiler in chrome

<table>
<thead>
<tr>
<th>Set</th>
<th>%</th>
<th>Total</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>55.50%</td>
<td>55.50%</td>
<td>10</td>
<td><code>get clientWidth</code></td>
</tr>
<tr>
<td>15.42%</td>
<td>15.42%</td>
<td>3</td>
<td><code>set innerHTML</code></td>
</tr>
<tr>
<td>12.98%</td>
<td>12.98%</td>
<td>2</td>
<td>(idle)</td>
</tr>
<tr>
<td>5.60%</td>
<td>79.69%</td>
<td>5</td>
<td><code>Javabreak.prepare</code></td>
</tr>
<tr>
<td>3.81%</td>
<td>3.81%</td>
<td>2</td>
<td><code>findbreak</code></td>
</tr>
<tr>
<td>2.00%</td>
<td>85.09%</td>
<td>1</td>
<td><code>Javabreak.breaklines</code></td>
</tr>
<tr>
<td>0.55%</td>
<td>0.55%</td>
<td>2</td>
<td>(garbage collector)</td>
</tr>
<tr>
<td>0.34%</td>
<td>0.34%</td>
<td>1</td>
<td>(program)</td>
</tr>
</tbody>
</table>

*Figure 10: Data from chrome's profiler running the algorithm with a caching measure method*

In theory an implementation like this would yield a better result than that of the previous one because certain words tend to be repeated many times throughout a text. In some texts the words might not repeat very often, or not at all in the worst case. In a case such as that this solution would be on par with the previous one as the access to the cache is so quick that it is insignificant even for a longer text when added up. In the above data collected from the profiler it shows that the performance gain from caching is there, but it is not good enough for us and we need another solution.

It is possible to take the idea of caching even a step further. Usually the amount of letters in a language is quite few and they are heavily repeated during a text. Consider the English language that consists of 25 letters in turn combines in to over 170,000 words. Of course there also exists other characters in written texts such as exclamation marks and punctuations but even with the addition of these they are very few. Caching letters would still leave us with the same worst case scenario, but a text only containing each letter from an alphabet once is highly unlikely and possibly quite short making the algorithms runtime short enough in either way.

```javascript
var cache_letters = function(text) {
    var letters, i, total_width, letter, w;
    letters = text.split('');
    total_width = 0;
    for(i = 0; i < letters.length; i++){
        letter = letters[i];
        w = cache_words(letter);
        total_width += w;
    }
    return total_width;
};
```

*Figure 11: A measuring method caching letters*
As is shown in the profiler, caching letters greatly reduces the amount of time spent in the two previously most time consuming calls of the algorithm. Since this caching almost always guarantees an improvement of the same order as is shown in the profiler, it is both much more reliable and faster than to cache each word.

However, there are not only upsides to measuring and caching the letters separately. Many fonts today are what is called proportional fonts which consists of glyphs that may vary in width, while its counterpart monospace fonts only may contain glyphs of the same width. Generally proportional fonts are easier to read and looks better, which is why most prefer to use them. Many proportional fonts utilizes Kerning, which is a process that adjusts the spacing between characters. Without going to far in to detail about kerning, it is usually used to bring characters closer together and give them some shared area.

On the green lines kerning is activated with the CSS-property `text-render: optimizeLegibility` and on the red lines kerning is not activated. As can be seen the difference is in many cases either not existent or the difference is small enough to not make a difference. The first two strings are exactly the same length with or without kerning activated. The difference on the last line is six pixels which amounts to one pixel per letter, although to be fair the A-V relationship is one of the most adjusted in most fonts. This still shows that kerning can make a real difference when measuring a word as a whole as opposed to adding the individual characters widths together.

When kerning is deactivated the letter-caching measurer is very fast in comparison to the other two and gives the same results. With kerning active, it might give a wrong answer causing the algorithm to take decisions on the formatting from faulty information. Usually the resulting formatting is good
enough and not to far off from the optimal solution, but there is no way to guarantee how small the error might be. Therefore it would not be recommended to use letter-caching with kerning active, otherwise it is a great improvement performance wise and yields the exact same results as the other methods.

There exists few alternatives for measuring a word except for the method already used, the only other known possibility during this thesis is to use a canvas element. The canvas element comes with a method `measureText` that returns the dimensions of a given text. This method requires that the font used in the canvas 2d context is specifically set to be the same as in the element where the text will be rendered after formatting.

A simple implementation of a measurer utilizing the canvas element looks like this:

```javascript
var ccanvas;
var canvas_measure = function(text) {
    if (!ccanvas) { // create canvas if it doesn't exist
        var canvas = document.createElement('canvas');
        canvas.getContext('2d').font = "12px Verdana";
        var docFragment = document.createElementDocumentFragment();
        docFragment.appendChild(canvas);
        ccanvas = canvas;
    }
    return ccanvas.getContext('2d').measureText(text).width;
}
```

*Figure 14: Measure method using the canvas element*

Running the same tests as before, but using the canvas method yields the following result from the chrome profiler.

<table>
<thead>
<tr>
<th>Self %</th>
<th>Total %</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.66%</td>
<td>30.69%</td>
<td>(anonymous function)</td>
</tr>
<tr>
<td>20.29%</td>
<td>59.69%</td>
<td>▲findbreaks</td>
</tr>
<tr>
<td>17.66%</td>
<td>92.66%</td>
<td>▲Javabreak.breaklines</td>
</tr>
<tr>
<td>8.67%</td>
<td>8.67%</td>
<td>▲measureText</td>
</tr>
<tr>
<td>4.69%</td>
<td>4.69%</td>
<td>(idle)</td>
</tr>
<tr>
<td>2.14%</td>
<td>2.14%</td>
<td>▲getContext</td>
</tr>
<tr>
<td>2.14%</td>
<td>2.14%</td>
<td>(garbage collector)</td>
</tr>
<tr>
<td>2.10%</td>
<td>13.33%</td>
<td>▲Javabreak.prepare</td>
</tr>
<tr>
<td>0.74%</td>
<td>0.74%</td>
<td>▲Javabreak.activatednode</td>
</tr>
<tr>
<td>0.61%</td>
<td>0.61%</td>
<td>▲(anonymous function)</td>
</tr>
<tr>
<td>0.43%</td>
<td>0.43%</td>
<td>▲getText</td>
</tr>
</tbody>
</table>

*Figure 15: Profile from chrome running the algorithm with a canvas measure method*
As can be seen in the profiling this canvas method is a lot better than even the letter caching method, seen from a performance kind of view. Only 9% time is used on the measureText method which is a huge improvement from all other methods mentioned before. Using a canvas we can work around the DOM communication which makes this method very fast.

However, it does not come entirely without problems, because it is using a canvas instead of putting the string in an actual DOM element it is restricted to use only formatting options for the text that is supported by the canvas element. This means for example that the CSS property letter-spacing is not useable.

Because this version now runs so fast it is difficult to profile by only running the algorithm a 1000 times so the test now runs it 10000 times instead on the same text as all tests have been running. Comparing the efficiency of the program now to when the first simple measure method was used it runs almost 6 times as fast.
Running the formatter again on the same SG4 used before generates this image.

Words: 200
Total time: 5.882204000081401
Prepare time: 3.23985400001402
Prepare % of total: 55.078912597543116

*Figure 17: Optimized algorithm also on Samsung Galaxy S4. Blue again representing the time for the preparations.*

It is easy to spot that measuring is much less efficient on mobile devices. Even so, the formatter runs 10 times as fast on mobile devices with the canvas method so there it is an even better upgrade. 5-6 ms is extremely fast and very useable, even on slower devices and longer texts.
Other improvements

After the great performance gains given from the measuring methods, improvements from other sources yield very little extra performance. Here shall be mentioned a few methods that are recommended for efficient Javascript programs, that have not given a measurable performance improvement but can still be considered important to try for anyone looking for performance enhancement.

Looking at the profile from the canvas method at the top of the list an anonymous function is shown to take 38% of the time now, probably a next great piece to optimize. Looking at the method mentioned it looks like this.

```javascript
if (feasible.length > 0) {
    var min = feasible.reduce(
        function(a, b) {
            return a.totaldemerits < b.totaldemerits ? a : b;
        }
    );
}
```

Figure 18: Unoptimized version of the anonymous function

The function that apparently consumes almost 40% of the total time is a simple function used in a reduce call, not very efficient. This is just a simple way to find the node in the list feasible that has the least totaldemerits, surely this could be done simpler.

A simpler method could look like this. Note that the code is made a named function even though unneccessary only to be easily spotted in the profiler.

```javascript
if (feasible.length > 0) {
    var get_best = function() {
        var mini = feasible[0];
        for (var i = 1; i < feasible.length; i++) {
            if (mini.totaldemerits > feasible[i].totaldemerits) {
                mini = feasible[i];
            }
        }
        return mini;
    };
}
```

Figure 19: Testing a potential optimization of the function.

Though, the profile from running with this shows us this result.

<table>
<thead>
<tr>
<th>Self %</th>
<th>Total %</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>41.70</td>
<td>41.70</td>
<td>get_best</td>
</tr>
<tr>
<td>18.00</td>
<td>92.84</td>
<td>Javabreak.breaklines</td>
</tr>
</tbody>
</table>

Figure 20: Results of the optimization.

The profile shows us that there is no gain from this, possibly only a slowdown (although I would argue that the difference is so small that it is likely the same). The reason that this little part of the code takes so much time is because of the many times it is being run and that can not easily be changed.
A widely recommended improvement is to remove and replace `forEach` with other loops such as `while` or `for`. Fortunately the implementation was using `forEach` in several locations so this was easily tested. Substituting `forEach` with `for` and running the profiler showed no improvements. This is probably due to the algorithm running few iterations and a lot of logic being run in every iteration of the loop.

Another time-consumer, that has been known to take its time before, is dereferencing of properties on a variable in Javascript. This means that using something in the like of `object.x.y.z` is slow, there is not very many nested properties in the formatter, however since the removal of the `forEach`’s a lot of `array.length` is being used in the `for` loops. Moving these out of the loop gives no performance gains, most browser catches uses like this and optimizes it automatically so there is no use in doing it manually except for perhaps very special cases.

A simple comparison of these loops and the property dereferencing was created on jsperf.com, a simple Javascript benchmark site.

The preparation code looked like this.

```javascript
<script>
    Benchmark.prototype.setup = function() {
        var a = new Array(100000),
            sum = 0;
        for (var i = 0; i < 100000; i++)
            a[i] = i;

        var Add = function(n) {
            sum += n;
        };
    }
</script>
```

*Figure 21: preparation code for jsperf tests*
And the actual tests and results.

<table>
<thead>
<tr>
<th>Test</th>
<th>Code</th>
<th>Ops/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>forEach</strong></td>
<td><code>a.forEach(function(x){})</code></td>
<td>164 ±2.14%</td>
</tr>
<tr>
<td><strong>for</strong></td>
<td>`for (var i = 0; i &lt; a.length; i++)</td>
<td>1,041 ±2.56%</td>
</tr>
<tr>
<td><strong>for - property</strong></td>
<td>`for (var i = 0; i &lt; a.length; i++)</td>
<td>1,007 ±2.79%</td>
</tr>
<tr>
<td></td>
<td>`for (var i = 0; i &lt; a.length; i++)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Add(a[i]);</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>94% slower</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 22: test code and results*

The test can be found here as of May 2014:

http://jsperf.com/for-foreach-compare

There is performance to be gained from switching from forEach, much performance, but in this implementation the gain from the switch is so small that it is inmeasurable. There is no performance to be gained from removing the `length` from the inner context of the loop.

Even though the implementation has been searched for possible improvements, few have been found and small changes have not resulted in even one significant improvement to the performance. Changes like the ones mentioned in this chapter have not been worth their while, and there is possibly not much left to improve without changing the algorithm itself.

**Mobile**

Javascript performance for mobile devices is much the same as it is for desktop. Javascript made for mobile devices have higher demands on performance because of the generally lower computing power of mobile devices, which is why performance optimization has been an important part in creating this formatter. Lower computing power usually shows as the Javascript performing less operations per second (OPS) and this results in it taking longer to run.

There can be other differences that one needs to be aware of if performance is of the issue for mobile. Because the browser is not the same that might be used for testing on the desktop it is not entirely sure that they behave the same.

For instance, in the implementation made for this thesis it was evident that the measuring methods was even slower on mobile devices than they where on desktop. When it took 20-30% of the total time on desktop it generally took 50-60% of the time on a mobile device.
Fortunately the performance after improvements was good enough to be used even though the measuring was slower than expected from tests on a desktop.

Switching the focus from time to visual performance, there is also differences there. Browsers behave differently on desktop and mobile and might for example try to adapt the font size automatically if not the right meta-tags has been used. This will results in text being measured in one font size and then rendered in another, yielding the formatting useless. So to use the algorithm on a mobile device it is imperative to adapt to this.

Most mobile devices have small screens, even though they might have a very high resolution the actual screen size being used is generally smaller. This affects the algorithm in two ways, first the algorithm runs faster because of the reduced line lengths, and secondly the optimal formatting will not be as good as it is on longer line lengths. The formatting suffers from shorter line lengths because it leaves the algorithm much less room for possibilities, longer line lengths results in more stretchability and shrinkability that the algorithm may use. Still, other less smart algorithms suffers from the same problem, so the improved visual experience is still there.
Chapter 4  Results

The implementation of the algorithm is functional and the resulting formatting is according to the definition used better than simpler algorithms.

To give an easy comparison between the different algorithms an example text has been formatted, one done by the algorithm implemented in this thesis, and the other being the native Chrome formatter.

Figure 23: Knuth-Plass algorithm formatting
These two images show two different formattings of the same text. In the left column of each image is shown the difference between the actual line width and the wanted line width, which was 400, in pixels. A longer line than expected shows a negative number while a shorter shows a positive. To get these numbers from the Chrome formatting, each word was wrapped in a span. Then the Y-position for each word was looked at and if it changed from the last word it is known that there was a line break between these two words. From there the lines could easily be reconstructed and measured with the same measure method used for the other algorithm.

Last number in the left column shows the sum of the squares of the numbers in the column, this is interesting as this is typically what is being minimized. As a result of minimizing that, the worse
algorithm should in general but not always have the line that has the highest difference. In this case the implemented algorithm shows a worst line of 28 whereas the native has 42, a big difference that is clearly visible in the line. Also the total of the sum shows that the Knuth & Plass leaves with a number almost half of the native formatter.

Of course these numbers shows very clearly how the algorithm implemented here is superior, in this case. The definition of a good formatting that minimizes the squares is probably a good one since it results in the lines being very close to the wanted length, but it is very hard to define what a good formatting is. How people perceive different formattings and if they prefer one over the other would be interesting to know, but that will have to be covered in another thesis.
Chapter 5  Conclusions

The implementation in Javascript shows that this algorithm works very fast when the measuring part is optimized. If it is possible to use the canvas method for measuring even the Javascript version would be useable on most devices. A native implementation in the browser would in all likeliness be very fast and quite possible not even distinguishable from a simpler and faster algorithm from the users point of view, except for the better formatting of course.

Would it be preferred over simpler algorithms?
Yes, and no. As a web browser is used both to write text and read text there are different answers. When writing, an algorithm as the one implemented here would be unusable because the insertion point would move around and confuse the writer. One should note that this also can happen when a text changes via for example Javascript. When reading a static text however, it would definitely be superior to simpler algorithms and yield a visually more pleasing formatting, perfect for news articles or e-books etc.

How could this be used?
The best option would be to include new CSS properties for text formatting, this would make it very easy for anyone to use smarter algorithm. If it were to be a CSS property it would also be natively supported by browsers, and thus the implementation would in all probability run much faster than any Javascript implementation.

As a conclusion, the algorithm in itself is definitely viable for use, the higher complexity of it is of no trouble and formatting almost any text is fast enough that no one will notice it happening. The formatting is better than that of existing algorithms and would be a very welcome addition to the web. Optimally it should be added as a CSS property, but an implementation in Javascript is also fast enough and easy to use.
Chapter 6  Discussion

As the implementation is done and working for formatting texts, there is still improvements to be done. Some of these possible improvements are mentioned here.

- **Hyphenation**
  Hyphenation is a great improvement to any formatter, possible hyphenation points makes new breakpoints that can be chosen by the algorithm giving it more flexibility.

- **Handle HTML**
  In some use cases the text to be formatted may include HTML tags such as span or bold. This is currently not supported, but should be an easy addition to make.

- **Right-to-left text and other**
  It is not certain that all languages are read left to right, and the algorithm may need to be improved before supporting this.

- **Evaluate actual improvement**
  The results from the formatter is great, but how much do people actually take notice? It would be interesting to know if people prefers this over simpler algorithms.

- **Native implementation**
  This Javascript implementation is fast enough for most cases, but the speed is still restricted in comparison to how well a native implementation would fare. It is also much slower when none of the improved measure methods can be used, this would likely not be an issue if it was supported directly by the browser.

Even though there are improvements that can be done, the implementation is working very well and proven to yield a better formatting than the existing algorithms in browsers. Personally I find it interesting that there are no better options today on the web, it would definitely be of great use as people are reading more and more online.

It has been great fun to do this work and I have learned a lot about Javascript, how text formatting works and how very important it is.
References

The MIT Press, 2009

O’Reilly Media, 2010

Center for the Study of Language and Information, 1998

[4] Donald E. Knuth, Michael F. Plass *Breaking paragraphs into lines*  
John Wiley & Sons, 1981

http://defoe.sourceforge.net/folio/knuth-plass.html  
Last accessed May 2014

Last accessed May 2014

http://en.wikipedia.org/wiki/Kerning  
Last accessed May 2014

Last accessed May 2014

[9] dev.w3.org, *CSS Text Module*  
http://dev.w3.org/csswg/css-text  
Last accessed May 2014

[10] stackoverflow.com, *Calculating text width with Javascript*  
http://stackoverflow.com/questions/118241/calculate-text-width-with-Javascript  
Last accessed May 2014

O’Reilly Media, 2008

https://developer.chrome.com/devtools/docs/cpu-profiling  
Last accessed May 2014

Last accessed May 2014

[14] Peter Shor, Shlomo Moran, Alok Aggarwal, Robert Wilber, Maria Klawe  
*Geometric applications of a matrix-searching algorithm*  
http://dl.acm.org/citation.cfm?id=10546  
Last accessed May 2014
[15] Wonsun Ahn, Jiho Choi, Thomas Shull, Maria Garzaran, Josep Torellas
Improving Javascript performance by deconstructing the type system
http://dl.acm.org/citation.cfm?doid=2594291.2594332
Last accessed May 2014
Copyright
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Felix Angerborn]