Flexible Font-Size Specification in Web Documents

Richard Lin†, Kim Marriott† and Peter Stuckey‡

†School of Computer Science & Software Engineering
Monash University
Clayton, Victoria 3168, AUSTRALIA
{rlin,marriott}csse.monash.edu.au

‡Department of Computer Science
University of Melbourne
Parkville, Victoria 3057, AUSTRALIA
pjscs.mu.oz.au

Keywords: document layout, constraint solving, combinatorial optimization.

Abstract. Web document designers typically have access to many dozens of different fonts, carefully choosing the type face and font size to achieve a certain feel, while most browsers provide only a few browser-dependant fonts. A key challenge in document delivery over the Internet is therefore to choose fonts on the viewing machine which most closely reflect the desires of the designer yet meet the requirements of the viewer. Here we present a very general model for font-size selection, in which the designer and viewer may specify both *required* and *desired* constraints between the font-sizes. It is the browser's responsibility to find which locally available fonts best satisfy these possibly conflicting desires. We show that, in general, this *font-size selection problem* is NP-hard. In practice, however, the constraints occurring in such problems usually contain no more than two variables and are acyclic. We give a polynomial time algorithm for this case. An empirical comparison shows that this polynomial time algorithm is considerably faster than the standard general purpose backtracking approach with arc consistency.

1 Introduction

One of the main technical challenges of document delivery over the Internet results from the tension between the desire of the document designer to specify the exact appearance of the document, the document viewer's desires and needs, and the capabilities of the viewing device and browser. This tension is particularly apparent when choosing fonts. Designers typically have access to many dozens of different fonts, carefully choosing the type face and font size to achieve a certain feel, while most browsers provide only a few browser-dependant fonts and the viewer's poor eyesight or low display device resolution may dictate that large font sizes be used. The key problem, therefore, is how to choose fonts that are available on the viewing machine and which reflect the desires of the designer, yet also satisfy the requirements of the viewer.

Recently, as part of the HTML 4.0 standard, Cascading Style Sheets (CSS 1.0) [Lie and Bos, 1997] allow the document designer to give an ordered list of font preferences with which to display each element of the document and to specify attributes of the font relative to the attributes of the parent font. However, as we discuss in the next section, CSS is still not powerful enough, in particular for specification of desired font-sizes.

Here we present a very general model for font-size selection, in which the designer and viewer may specify *required* constraints between the font sizes, such as all fonts must be 9pt or larger, and *desired* constraints, such as this font should be as close as possible to 13pt. It is the browser's responsibility to find fonts which are available to the browser and which best satisfy the possibly conflicting desires of the designer and viewer.

Unfortunately, as we show, the *font-size selection problem* is NP-hard and so in general, expensive general purpose techniques developed for solving combinatorial optimization problems over finite domains are required to solve it. However, in practice the constraints occurring in such problems usually contain no more than two variables and are acyclic. We give a polynomial time algorithm for this case. Finally, we provide an empirical comparison between our polynomial time algorithm and a general purpose backtracking approach which employs arc consistency to prune the search space.

There is, of course, a huge literature devoted to solving combinatorial optimization problems over finite domains, see for instance the bibliography in [Marriott and Stuckey, 1998]. However, to our knowledge the specific problem of font-size selection is unstudied. The idea of using required and desired constraints with a weight to capture the designer, viewer and browser's requirements and desires is based on our earlier work on Web document layout in which linear arithmetic constraints are used to capture these conflicting desires for placement of figures and table layout in Web documents [Borning et al., 1997a]. Of course, such systems of desired and required constraints are simply an instance of constraint hierarchies [Borning et al., 1992]. However, algorithms for solving constraint hierarchies have not considered finite domain constraints. The earliest algorithms, such as DeltaBlue [Freeman-Benson and Maloney, 1989], are based on local propagation and are therefore restricted to acyclic equality constraints and so cannot handle inequalities or finite domains. More recent algorithms, such as Cassowary [Borning et al., 1997b], cater for arbitrary linear arithmetic constraints, but cannot handle finite domains.

2 The Font-Size Selection Problem

As indicated previously, one of the main technical challenges of document delivery over the Internet results from the tension between the desire of the document designer to specify the exact appearance of the document, the document viewer's

```
<BODY STYLE=
  "font-family:san-serif;
   font-size:13pt">
<P STYLE="font-size:50%">
Some text in a paragraph:
<BLOCKQUOTE STYLE=
  "font-size:200%">
And now some text in a
quotation.
</BLOCKQUOTE>
</P>
</BODY>
```

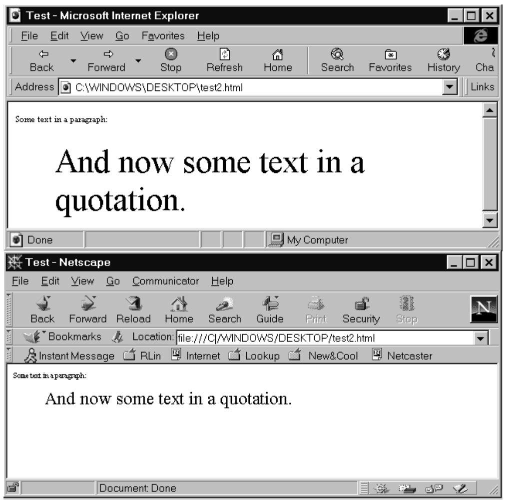

Fig. 1. Example CSS specification and its appearance

desires and needs and the capabilities of the viewing device and browser. This tension is particularly apparent when choosing fonts.

This problem is being partially addressed by the W3 Consortium's push for standards for font transmission over the Internet. When these become finalised, the browser will be able to automatically download the designer's desired fonts along with the document. However, font transmission does not entirely solve the problem for at least two reasons. The first is that it does not take into account the needs and desires of the viewer, such as for large fonts, and the capabilities of the viewing device, such as poor resolution. The second is that downloading fonts is time consuming and expensive over lines with low bandwidth. For this reason the browser's default behaviour will almost certainly be to first download the document and display it with local fonts, then download the designer's fonts and redisplay the document using the correct fonts. Thus, for the foreseeable future there will be a need for the browser to choose fonts which best meet the (possibly conflicting) demands of the designer and the viewer.

In accord with its underlying philosophy of only specifying the logical structure of a document, early versions of HTML provided little support for the designer to specify fonts. More recently, as part of the HTML 4.0 standard, Cascading Style Sheets (CSS 1.0) [Lie and Bos, 1997] allow the document designer to give an ordered list of font preferences with which to display each element of the document. Font preferences may either be associated directly with the element or given in a style sheet. Figure 1 gives a simple example of the capabilities of CSS and also shows the appearance using Netscape and Microsoft Internet Explorer.

Cascading style sheets go a long way toward solving the problem of flexible font specification. However they are still not powerful enough, in particular with regard to font size. CSSs allow the design to either give an *absolute* size for the font, such as `13pt` or `xx-small`, or a *relative* size, such as `50%` or `larger`, which is in term's of the parent element's font size. For instance, in Figure 1 the font size of the body of the HTML document is 13pt, while the paragraph font size is $\frac{50}{100} \times 13$, say 6pt, and the block quotation font size is $\frac{200}{100} \times 6$, say 12pt. Restrictions are that:

- It is not possible to refer directly to sizes other than that of the parent. For instance, in Figure 1 we may have wanted to specify that the block quotation font size is the same as the body size. Indeed we have attempted to do this, but because of truncation errors we have obtained two different font sizes.
- The specification of CSS does not precisely detail how fonts are chosen when no font is ideal. For instance if 13pt is not available will the browser choose 12pt or 14pt? This means that the appearance may be browser dependant even when exactly the same fonts are present. We note that this is the behavior shown in Figure 1 when two different browsers running on the same computer and with access to the same fonts display the same document with very different font-sizes despite the seemingly precise CSS font-size specification in the document. Ideally, the designer should be able to specify what is to be done when the desired size is not present.
- It is not possible to directly combine preferences from the viewer, browser and designer. In the case that there are multiple styles relating to the same element then rather complex conflict resolution rules will ensure that only one of those styles is actually considered. Thus, if a sight-impaired viewer required that all fonts in Figure 1 be 9pt or larger, then either this requirement or the original constraints of the designer will be ignored.

These restrictions ensure that the font size of each element can be calculated using a single expression directly from the font size of the parent element.

Ideally, both the designer and the viewer should be able to specify both *required* constraints between the font sizes, such as all fonts must be 9pt or larger, and *desired* constraints, such as this font should be as close as possible to 13pt. We would also like to be able to combine the conflicting desired constraints and find the solution which globally minimizes the conflict, taking into account the strength of the associated desire. In order to do this we associate with each desired constraint c an *error function* ϵ over the variables in c which returns the error associated with a particular assignment to the variables. The error is required to be a non-negative real number and to be 0 if the constraint is satisfied. For instance the error associated with an equation $s = t$ might be $|s - t|$ and for an inequality $s \leq t$ which is more important it might be $1000|s - t|$ when $s > t$ and 0 otherwise.

Given a collection of required and desired constraints over some font variables we would like the browser to find the assignment of local font sizes to each font variable which satisfies all of the required constraints and which minimizes the error associated with each desired constraint. This is formalized by:

Definition. Font size selection problem: The problem statement consists of

n font variables, $F_1, ..., F_n$ such that

- Each variable F_i has a *domain*, $dom(F_i)$ which is the set of sizes that this font variable can take.
- There is a conjunction of arithmetic *required constraints*, C, over the variables $F_1, ..., F_n$.
- There is a conjunction of arithmetic *desired constraints*, $D_1, ..., D_m$, over the variables $F_1, ..., F_n$ with associated *error functions*, $\epsilon_1, ..., \epsilon_m$.

An assignment of sizes to the font variables, θ maps font variables to sizes, i.e. $\theta \equiv \{F_1 \mapsto d_1, \ldots, F_n \mapsto d_n\}$. A size assignment θ *satisfies* the problem if $\theta(F_i) \in dom(F_i)$ for each i and θ satisfies the required constraints C. The FONT SIZE MINIMIZATION problem is to find an assignment of sizes to the font variables, θ, which satisfies the problem and which minimizes $\sum_{i=1}^{m} \theta(\epsilon_i)$.

Note that the desired and required constraints are obtained by combining the relationships dictated by the designer and viewer, while the domains are dictated by the choice of fonts available to the browser.

Example 2.1 To return to Figure 1, we have three variables FB, FP and FQ which are respectively the font size of the body, paragraph and quotation. Imagine that the available font sizes for san-serif are 5pt, 9pt, 10pt and 14pt. Then we have that $dom(FB) = dom(FP) = dom(FQ) = \{5, 9, 10, 14\}$. Our sight-impaired viewer requires that all fonts are at least 7pt in size and the designer requires that font-size of the body and quotation is the same so we have the required constraints

$$\begin{array}{l} C_1 : FB \geq 7 \wedge C_2 : FP \geq 7 \quad \wedge \\ C_3 : FQ \geq 7 \wedge C_4 : FB = FQ. \end{array}$$

Our designer would like the body font-size to be 13pt and, if this is not possible would prefer it to be bigger than 12pt, and wants the paragraph font size to be half that of the body. Thus the desired constraints are

$$D_1 : FB = 13 \wedge D_2 : FB \geq 12 \wedge D_3 : 2FP = FB$$

and the associated error functions are

$$\begin{array}{ll} \epsilon_1(FB) & = |FB - 13| \\ \epsilon_2(FB) & = \begin{cases} |12 - FB| & \text{if } 12 \geq FB \\ 0 & \text{otherwise} \end{cases} \\ \epsilon_3(FP, FB) & = |2FP - FB| \end{array}$$

The optimal answer to this problem is $\{FB \mapsto 14, FP \mapsto 9, FQ \mapsto 14\}$.

FONT SIZE MINIMIZATION is an example of a optimization problem over a finite domain [Marriott and Stuckey, 1998]. Many such problems are well-known to be NP-hard. Indeed it is not too difficult to prove that this problem is also NP-hard:

Theorem 2.1 FONT SIZE MINIMIZATION is NP-hard.

Proof: The proof is by reduction of VERTEX COVER [Garey and Johnson, 1979] to font selection. We only require binary required constraints and unary desired constraints. The VERTEX COVER problem is: given a graph G and number k is there a set C of k vertices such that all edges in G intersect at least one node in C. Suppose G is made up of nodes $1, \ldots, n$ and edges $E = \{(a_1, b_1), \ldots, (a_l, b_l)\}$. The font selection problem involves variables $F_1, \ldots, F_n$ each with domain $\{0, 1\}$ (indicating whether the vertex is in the set C). The required constraints are $F_{a_i} + F_{b_i} \geq 1$ for $1 \leq i \leq l$. Any solution of the required constraints gives a vertex cover for the original problem by selecting $C \equiv \{i \mid F_i = 1\}$. The desired constraints are $D_i \equiv F_i = 0, 1 \leq i \leq n$ with error function $\epsilon_i(F_i) = F_i$. A solution of FONT SIZE MINIMIZATION such that $\sum_{i=1}^{n} \epsilon_i = \sum_{i=1}^{n} F_i \leq k$ gives a solution of VERTEX COVER.

Despite the intractability of most finite domain optimization problems, in part because of their commercial importance, considerable effort in the operations research, artificial intelligence and constraint programming communities has been spent on finding general purpose algorithms which are efficient for practical problems. One of the most popular approaches is to combine *backtracking search* with some form of *consistency* technique [Marriott and Stuckey, 1998]. Because of backtracking, this method is guaranteed to find an optimal solution while efficiency is achieved by using the consistency method to prune the search space.

Example 2.2 The backtracking with arc-consistency approach to Example 2.1 starts with initial domain $\{5, 9, 10, 14\}$ for each variable and applies consistency checking for each required constraint. The inconsistent value 5 is removed from the domains of each of the variables since it does not satisfy the required constraint that the fonts are larger than 7. Then a variable is selected, say FB, and each value in its domain tried in turn. After setting $FB = 9$, arc consistency reduces the domain of FQ to $\{9\}$ in order to satisfy $FB = FQ$. Next another variable is selected, say FP, and each value in its domain tried, first $FP = 9$. Finally the last variable FQ is selected and its unique value is selected. This gives $\{FB \mapsto 9, FP \mapsto 9, FQ \mapsto 9\}$ which has total error $4 + 3 + 9 = 16$. Backtracking goes back to the last choice made $FP = 9$ and tries the next choice $FP = 10$. The solution with minimal total error is kept. Figure 2 show the search tree examined, together with the optimization value of all the solutions found.

Clearly in the worst case, backtracking with arc-consistency has exponential behaviour. This is not surprising given Theorem 2.1 from which it follows that any algorithm for solving the font-selection problem almost certainly has worst-case exponential behaviour.

3 Restricting the Font-Size Selection Problem

At this point it is worth re-examining our motivating example for font size selection problem (Example 2.1). Close observation reveals that the required

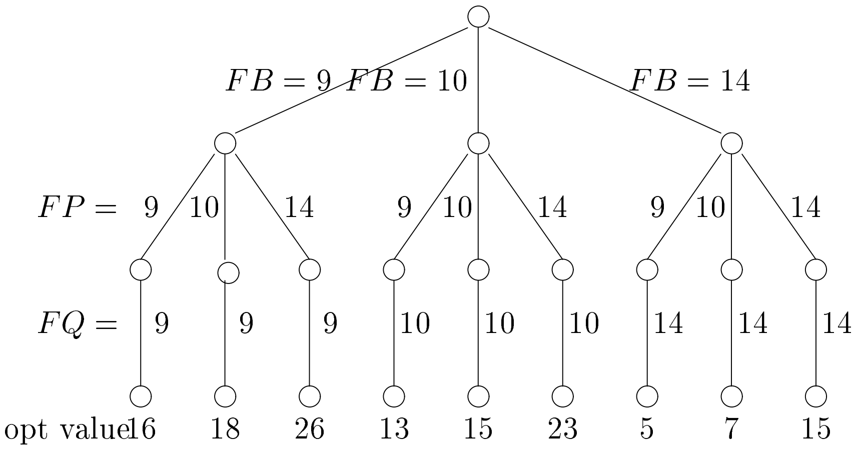

Fig. 2. Backtracking arc-consistency for font selection

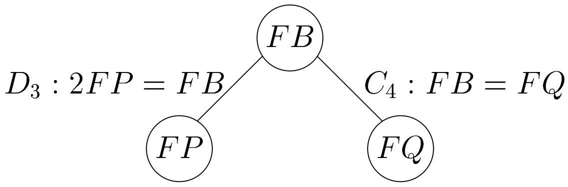

Fig. 3. Constraint graph for problem of Example 2.1

and desired constraints in the problem have two characteristics which mean that expensive general purpose constraint solving algorithms are not required.

The first characteristic is that the constraints are either *unary* or *binary*. That is to say, at most two variables occur in each required and desired constraint. This is one reason why we chose arc consistency in the previous section since it is the strongest consistency method for binary constraints.

The second characteristic is that if the system of constraints is viewed as a *constraint graph*, with variables as nodes and an edge between two nodes if there is a required or desired constraint involving those two variables, then this graph is acyclic. See for example Figure 3 which shows the constraint graph for Example 2.1. Note that multiple constraints between the same variables collapse into a single edge in the constraint graph.

Definition. A *binary acyclic font-size selection problem*, or *BAFSS problem* for short, is a font size selection problem whose required and desired constraints are unary or binary and whose corresponding constraint graph is acyclic.

In practice, virtually all font-size selection problems are BAFSS problems. For instance, the font-size constraints arising from CSS 1.0 always give BAFSS problems since constraints only occur between two font-sizes if the corresponding document elements are parent and child.

The restriction to acyclic constraints makes the problem much easier. In fact a straightforward dynamic programming approach can be used. The essential idea is to view the constraint graph as a tree, arbitrarily choosing the root, and then visit the nodes in the tree bottom-up. When a node in the tree, corresponding to variable v say, is visited the algorithm computes the best partial assignment to the variables whose nodes are lower in the tree for each value in the domain of v.

A precise algorithm is given in Figure 4. Assumptions behind the algorithm are that the font variables are numbered from 1 upwards, the set of desired and required constraints contains only binary constraints and the corresponding constraint graph is acyclic and is a spanning tree of the variables. If the last assumption is not true then we can simply add "*true*" constraints to arbitrarily connect the disconnected variables.

If v and v' are variables we let C_v and $C_{v,v'}$ be the conjunction of unary and binary required constraints involving v and v and v', respectively. Similarly we let ϵ_v and $\epsilon_{v,v'}$ be the error function obtained by summing the error functions of the unary and binary desired constraints involving v and v and v', respectively. The set of variables $connected(v)$ are those other variables in a binary desired or required constraint involving v.

The algorithm uses dynamic programming to construct a global table $best$ such that $best(v,d) = \langle f, p \rangle$ indicating that for variable v with domain value d the best assignment to the variables "below v" is p and this gives the value f. The algorithm starts from variable 1, arbitrarily choosing this to be the root of the constraint graph and then performs a recursive depth-first traversal of the constraint graph. When a node, for variable v say, is visited the algorithm calls itself recursively to construct the table $best$ for each of the node's children. Once this is done it uses this information to determine the entry in $best$ for v and for each value in the domain of v. Note that an ∞ in the entry for $best(v,d)$ indicates that the required constraints are unsatisfiable if v takes value d.

Example 3.1 Consider the BAFSS problem from Example 2.1 with ordering of the variables $[1,2,3] = [FB, FP, FQ]$. We have that $C_{FB} \equiv FB \geq 7$, $C_{FP} \equiv FP \geq 7$, $C_{FQ} \equiv FQ \geq 7$, $C_{FB,FQ} \equiv FB = FQ$, $\epsilon_{FB} \equiv \epsilon_1 + \epsilon_2$ and $\epsilon_{FB,FP} \equiv \epsilon_3$. We have $connected(FB) = \{FP, FQ\}$ and $connected(FP) = connected(FQ) = \{FB\}$. The call to $process(FB, 0)$ calls $process(FP, FB)$. We examine each value d in the domain of FP, $\{5, 9, 10, 14\}$. Since $connected(FP)$ is only the parent (FB), the loop calculates

$$\begin{array}{ll} best(FP,5) \;\;= \langle +\infty, \{FP \mapsto 5\}\rangle, & best(FP,9) \;\;= \langle 0, \{FP \mapsto 9\}\rangle, \\ best(FP,10) = \langle 0, \{FP \mapsto 10\}\rangle, & best(FP,14) = \langle 0, \{FP \mapsto 14\}\rangle. \end{array}$$

```
process(v,parent)
  for each v' ∈ connected(v) and
      v' ≠ parent do
      process(v',v)
  endfor
  for each d ∈ dom(v) do
      if d satisfies C_v then
          f := ε_v(d)
      else
          f := ∞
      endif
      p := {v ↦ d}
      for each v' ∈ connected(v) and
          v' ≠ parent do
          /* find best assignment to
          variable v' and below */
          bestsofar := ∞
          for each d' ∈ dom(v') s.t.
              ⟨d, d'⟩ satisfies C_{v,v'} do
              let ⟨f', p'⟩ = best(v', d') in
              f' := f' + ε_{v,v'}(d, d')
              if f' < bestsofar then
                  bestsofar := f'
                  bestp' := p'
              endif
          endfor
          f := f + bestsofar
          p := p ∘ bestp'
      endfor
      best(v, d) := ⟨f, p⟩
  endfor

solve()
  /* construct best table –
  0 is a non-existent parent of 1 */
  process(1,0)
  /* find best assignment */
  bestsofar := ∞
  for each d ∈ dom(1) do
      let ⟨f, p⟩ = best(1, d) in
      if f < bestsofar then
          bestsofar := f
          bestp := p
      endif
  endfor
  if bestsofar = ∞ then
      return false
  else
      return bestp
  endif
```

Fig. 4. The BAFSS Algorithm

Next $process(FB, 0)$ calls $process(FQ, FB)$. Similarly to above it calculates

$$best(FQ, 5) = \langle +\infty, \{FQ \mapsto 5\}\rangle, \qquad best(FQ, 9) = \langle 0, \{FQ \mapsto 9\}\rangle,$$
$$best(FQ, 10) = \langle 0, \{FQ \mapsto 10\}\rangle, \qquad best(FQ, 14) = \langle 0, \{FQ \mapsto 14\}\rangle.$$

The remainder of the call to $process(FB, 0)$ tries each value in the domain of FB in turn. For value $d = 9$, f is calculated as 7, and p is $\{FB \mapsto 9\}$. Now each connected variable is considered in turn. The first assignment to FP which is consistent with $FB = 9$ is $FP = 5$, which calculates $f' = +\infty$. The next assignment $FP = 9$ calculates $f' = 9$ and and sets $bestsofar$ to 9 and $bestp'$ to $\{FP \mapsto 9\}$. The remaining possible assignments $FP = 10$ or 14 calculate $f' = 11$ and 19 respectively, not improving on the current best. Hence f is updated to 16 and p becomes $\{FB \mapsto 9, FP \mapsto 9\}$. Next values for FQ are considered, there is only one consistent value (w.r.t. $C_{FB,FQ}$) that is $FQ = 9$. For this value $f' = 0$, hence f remains 16 and p becomes $\{FB \mapsto 9, FP \mapsto 9, FB \mapsto 9\}$. The calculation is similar for other values, with result:

$$\begin{aligned}
best(FB, 5) &= \langle +\infty, \{FB \mapsto 5, FP \mapsto 9, FQ \mapsto 5\}\rangle, \\
best(FB, 9) &= \langle 16, \{FB \mapsto 9, FP \mapsto 9, FQ \mapsto 9\}\rangle, \\
best(FB, 10) &= \langle 13, \{FB \mapsto 10, FP \mapsto 9, FQ \mapsto 10\}\rangle, \\
best(FB, 14) &= \langle 5, \{FB \mapsto 14, FP \mapsto 9, FQ \mapsto 14\}\rangle.
\end{aligned}$$

In $solve()$ the best solution returned is that given by $best(FB, 14)$. The advantage over the backtracking approach is clear, the evaluation of the subproblem involving FP and FQ are each performed only once instead of multiple times.

Theorem 3.1 The BAFSS Algorithm returns an optimal solution to the input BAFSS problem.

Proof: The result holds by induction on subproblems defined by the constraint graph. Define subproblem i as the BAFSS problem involving variables V_i that appear below node i in the graph and all constraints only involving variables in V_i. The induction hypothesis is that for $best(i, d) = \langle f, p\rangle$, either $f = +\infty$ or p is an optimal solution for subproblem i with domain of i set to $\{d\}$, and f is the optimal value. The base case is straightforward. The inductive step follows from the composition of optimal solutions which share no variables except i.

Complexity of the algorithm is reasonably straightforward to determine.

Theorem 3.2 The worst case time complexity of the BAFSS Algorithm is $O(D^2 \cdot (V + C))$ where V is the number of variables, C is the number of required and desired constraints and D is the maximum domain size for any variable.

Proof: Since the constraint graph is a tree, there are $V - 1$ arcs corresponding to the case $v' \in connected(v)$ and $v' \neq parent$. Hence the loop */* find best assignment to variable v′ and below */* is executed at most $D(V - 1)$ times. Each constraint (required or desired) is used (part of $C_{v,v'}$ or $\epsilon_{v,v'}$) in exactly one iteration of this loop. The inner for loop is executed at most D times. The amortised cost of all executions of the loop is thus $O(D^2 \cdot (V + C))$, where each constraint check and computation of the error value is assumed to be a constant time operation. In practice by handling unconnected parts of the constraint graph separately this can be reduced to $O(D^2 \cdot C)$.

It is interesting to consider how far we can relax the restrictions on BAFSS problems and still escape intractability. The first point to note is that acyclicity of the constraint graph is crucial. The font-size selection problems constructed in our earlier proof of NP-hardness in Theorem 2.1 only contain binary constraints but their constraint graphs may be cyclic.

The second possible relaxation would be to drop the requirement that the constraints are binary. In fact it is straightforward to modify the BAFSS Algorithm to handle the case when the number of variables in the constraints is

Problem	Var.	Cons.	SICStus-LL	SICStus-SL	BAFSS
1	3	7	0.002	0.004	0.002
2	10	27	1.640	0.090	0.057
3	20	55	10.970	0.250	0.118
4	10	40	3.370	0.220	0.073
5	20	50	12.320	0.230	0.096
6	20	71	8.270	0.260	0.125
7	10	17	22.380	0.180	0.057
8	20	35	134.440	7.940	0.125
9	10	30	116.320	0.890	0.075
10	20	29	11160.130	367.650	0.097
11	20	51	13566.150	247.230	0.130

Table 1. Empirical comparison

bounded by a fixed constant k. In this case the algorithm still remains polynomial but in D^k where D is the maximum domain size. If we allow an unbounded number of variables then the problem becomes NP-hard since finding an integer solution to a single linear arithmetic equation is NP-hard (by reduction to set partitioning), and this is a subproblem for BAFSS.

4 Empirical Evaluation

Analysis of the worst case complexities suggests that the BAFSS Algorithm will be considerably faster than general purpose constraint solving techniques such as backtracking with arc consistency. However, the reason that backtracking with arc consistency is widely used is that it is quite fast on many practical problems. Therefore it is well-worth performing an empirical comparison of the two approaches to determine their relative efficiency in practice.

We have implemented the BAFSS Algorithm in Java. We compare it with the CLP language SICStus Prolog which provides efficient implementation of backtracking with arc consistency. We have given two implementations in SICStus Prolog, the first uses the library defined labelling predicate, while the second uses a labelling predicate especially written for this type of problem which takes into account the required constraint and uses the unary and binary equalities to guide the choice of values for variables, choosing those values with least cost first. We have tried these approaches on 11 problems, the first being Example 2.1. Table 1 details our findings. For each problem we give the number of font variables (Var) and the number of constraints (Con). We give the time in seconds to determine the solution using SICStus with the library labelling (SICStus-LL) and with the specialised labelling (SICStus-SL), then with the BAFSS Algorithm (BAFSS). All timings were done on a Sun SPARC 5 with 256M of memory and an 85MHz processor.

5 Conclusion

We have investigated the font selection problem. This problem is motivated by the need to have flexible means for the document designer to specify relative font-sizes in Web documents. We have shown that the original problem is NP-hard, but suggested very reasonable restrictions to this problem which give rise to the BAFSS problem. We give a polynomial time algorithm for this problem based on dynamic scheduling. Finally an empirical evaluation of the suggests that our algorithm works well in practice and is orders of magnitude faster than general purpose constraint solving techniques such as backtracking with arc consistency.

Currently we are integrating the BAFSS Algorithm into a prototype browser for *constraint style sheets*. Constraint style sheets like any style sheet specify the appearance of elements in a document, however unlike CSS they use constraint hierarchies to merge different style sheets together rather than cascading. They also provide full-fledged linear arithmetic constraints allowing for more flexible layout [Borning et al., 1997a]. Now with the BAFSS Algorithm they also provide more flexible handling of fonts.

We note that the current problem statement is couched in terms of font-sizes. However the discussion and results carry over to several other problems. Most obviously the font weight. However even more generally we can relax the requirement that the constraints be arithmetic and replace them by arbitrary binary constraints. Since the BAFSS Algorithm essentially uses arc consistency it will still be correct, thus the BAFSS Algorithm can be used for any time of finite domain problem in which the number of variables in the constraints is bounded and the constraint graph is acyclic. It is also possible to add use other orderings including lexicographic, to compare different solutions – cf constraint hierarchies [Borning et al., 1992].

References

Borning, Alan, Freeman-Benson, Bjorn, and Wilson, Molly (1992). Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270.

Borning, Alan, Lin, Richard, and Marriott, Kim (1997a). Constraints for the web. In *Proceedings of the Fifth ACM International Multi-Media Conference*, pages 173–182.

Borning, Alan, Marriott, Kim, Stuckey, Peter, and Xiao, Yi (1997b). Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 10th ACM Symposium on User Interface Software and Technology*, pages 87–96.

Freeman-Benson, Bjorn and Maloney, John (1989). The deltablue algorithm: An incremental constraint hierarchy solver. In *Proceedings of the Eighth Annual IEEE Phoenix Conference on Computers and Communications*, Scottsdale, Arizona. IEEE.

Garey, M. and Johnson, D. (1979). *Computers and Intractability.* W. Freeman and Company, New York.

Lie, Håkon and Bos, Bert (1997). *Cascading Style Sheets*. Addison-Wesley.
Marriott, Kim and Stuckey, Peter (1998). *Programming with Constraints: An Introduction*. The MIT Press.