ふりがな付き HTML 文書に対する文字列探索

齋藤 匡(ただし)[1,a)]　岩崎 英哉[2,b)]

概要：日本語文書に見られるふりがなは，機械的な文字列探索を難しくする要因にもなる．HTML においても，ふりがなは以前からよく用いられているが，そのページ内検索はどの最新ブラウザでも満足に行えないままである．たとえば Firefox では，ふりがなは検索対象から除外される．本研究では，この問題を解決する文字列探索の方法を考案した上で JavaScript ライブラリとして実装し，主要な複数ブラウザでの動作と有効性の評価を行った．

キーワード：ふりがな，ルビ，HTML，文字列探索

1. はじめに

日本語の学習と利用において，ふりがなは欠かせない存在である．難読漢字の読みを示す注釈として，あるいは文芸での一手法などとして，目にしない日はないほどに身近な存在である．その歴史が日本語での漢字採用まで遡れるほど，ふりがなは長く利用されてきた [17]．

しかし現代の情報化社会において，ふりがなを電子的に不自由なく活用できているとは言い難い．ソフトウェア開発の多国籍化とコードベース共通化の結果，日本語のふりがなのように世界的な需要が少ない機能は，アプリケーションに実装されないのが現実である．

Web ブラウザでのページ内検索はその典型例と言える．ブラウザでふりがなの表示が可能になって 20 年以上経過するが，どの主要ブラウザであっても，ふりがなを含むテキストに対して自然な文字列探索を行えない状態が続いている．たとえば，Firefox 107 でふりがなの文字列探索を試みると，図 1 のようにマッチは **0 件**になる．

ここでの「自然な文字列探索」とは，単にふりがなへのマッチを指すのではない．本論文においては，探索の対象がふりがなか，その下側の文字か，あるいはふりがなのない通常の本文か，といった区別なくマッチできる文字列探索を指す．たとえば「わがはいは」を探すなら，「吾輩(わがはい)は」のふりがな部分と後続文字でも，ふりがなのない本文「わがはいは」そのものでも，どちらにもマッチするべきである．

本論文ではこの問題を解決するため，ふりがな付き HTML 文書における自然な文字列探索を実現する手法を提案する．ふりがな付きの文字列テキ

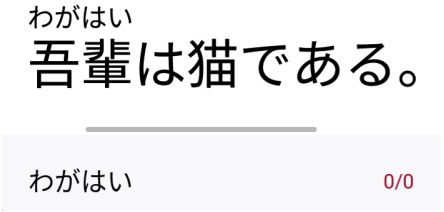


図 1　Firefox でのふりがな付き文書のページ内検索例

[1] 電気通信大学 大学院情報理工学研究科
[2] 明治大学 理工学部
[a)] 20tsaitou@ipl.cs.uec.ac.jp
[b)] hideya.iwasaki@acm.org

ストは，複数に分岐して並列に存在し，また合流するような文字の並びであると捉え，それに沿ったデータ構造と探索法を用いる．これにより，従来不可能であった探索が可能になることを示す．

我々は有効性実証のため，ブラウザで動作する JavaScript ライブラリ Ruby Finder として提案手法を実装し，評価を行った．その結果，複数の主要ブラウザすべてにおいて実用的な速度で動作することが示された．

本研究で示した文字列探索は 20 年間以上欠けていた機能を実現したもので，日本語利用者ならば誰でもメリットを享受できる．またそれに留まらず，中国・シンガポール・台湾など [9]，国際的にも活用され得る機能である．これを移植性の高い形で実現し，複数の主要ブラウザでの動作を確認した点には社会的な価値があると考える．

以降，HTML 文書内の被検索対象文字列を**テキスト**，ユーザが文書内で探す検索対象文字列を**パターン**と呼ぶ．

本論文の構成は以下の通りである．2 節では，背景となるふりがな自体の概要と HTML における表現法，現行主要ブラウザのページ内検索での問題点を示した上，その望ましい挙動を提示する．3 節では関連研究に触れ，従来の文字列探索をはじめとする既存手法の限界とふりがなを扱うソフトウェアの現状について述べる．4 節では提案の中核となる「分岐するテキスト」という考えの必要性と共に，データ構造と探索方法を示す．5 節で JavaScript ライブラリとしての実装を示し，6 節でその評価を行う．最後に 7 節で本論文のまとめと今後の課題を述べる．

2. ふりがなと HTML 文書

ふりがなは古くから，漢字の読みを説明するのみに留まらず，意味の多重化 [17] 実現の手段として用いられてきた．現代においてもその需要は衰えていないが，ふりがなの電子的な活用には多くの困難が残っている．

2.1 ふりがなの概要

ふりがなは文書中の文字に対して付ける注釈であり，伝統的には**ルビ**とも呼ばれる．本論文での表記はすべて「ふりがな」としたが，その内容は仮名（ひらがな・カタカナ）に限定されない．

典型的な利用例は，漢字の読みを説明することである．図 2 のように漢字に小さくひらがな・カタカナを添えることで，その読みを明確にする．また，ふりがなの有用性は難読漢字に留まらない．日本の義務教育で経験するように，日本語学習者が漢字学習時の補助としてふりがなを活用する場面は多い．

ここでの漢字のように，説明される側の文字を**親文字**と呼ぶ．親文字に対してどうふりがなを配置するか，どの親文字にどの程度ふりがなを振るか，それぞれ複数のスタイルがある．これらを含む，日本語処理の包括的な資料 [14] が存在する．

2.2 HTML と <ruby> タグ

HTML 中のふりがなは <ruby> と関連タグによって表現される．1999 年，Internet Explorer 5 によって独自に実装されたのを皮切りに，2001 年の XHTML を利用した規格制定 [3] を経て，すでに 20 年以上の歴史を持つ．

本論文の執筆現在，これら <ruby> 等のタグは主要なブラウザ Firefox[*1]，Chrome[*2]，Safari[*3] のすべてで利用可能である．これらを含め，<ruby> をサポートするブラウザの世界シェアは約 98.6% にのぼる [2]．

現在の HTML [8] では，図 2 を「<ruby>吾輩<rp>（</rp><rt>わがはい</rt><rp>）</rp></ruby>

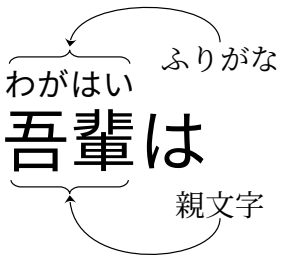


図 2　ふりがなの構造

[*1] https://www.mozilla.org/firefox/
[*2] https://www.google.com/chrome/
[*3] https://www.apple.com/safari/

表 1　主要ブラウザでの <ruby> ページ内検索時のマッチ成否

パターン	パターンの説明	Firefox	Chrome	Safari
吾輩	親文字のみ	○	○	○
わがはい	ふりがなのみ	×	○	○
吾輩は	親文字＋後続文字	○	×	×
わがはいは	ふりがな＋後続文字	×	×	○
吾輩わがはい	親文字＋ふりがな・誤マッチ	×	×	○

は」のように表記する．まず <ruby> の直下に親文字を，<rt> の中にそのふりがなにあたる文字を書く．

<rt> を囲む 2 つの <rp> は，<ruby> 未対応ブラウザへの互換性の配慮である．<rp> がない場合，<ruby> 非対応ブラウザの多くはタグだけを読み飛ばし，その中身の文字列は本文として扱ってしまう．このため，もし <rp>...</rp> がなければ，画面には一列に「吾輩わがはいは」と表示され，混乱や意味の変化が生じる．しかし <rp> によって「吾輩（わがはい）は」と表示され，文の理解が容易になる．

これ以外にも，<ruby> 中の親文字それぞれに対応するふりがなを指定する方法もある．さらに 2001 年の規格 [3] には，親文字の上下に複数のふりがなを振るなど，より多様なニーズに応えるための機能が定められていた．しかし現在，W3C から WHATWG への標準化作業移管の中で当該の機能が抜け落ちた状態であり，目下議論が続いている [7].

2.3　<ruby> とページ内検索の問題

<ruby> を使ったふりがな付き文書でページ内検索をするには，現状大きな困難を伴う．これは，どのブラウザのどのバージョンを使うか，といったブラウザの利用状況に関わらず，あらゆる場合に直面する問題である．

3 つの主要ブラウザでの <ruby> とページ内検索の挙動を表 1 に示す．「吾輩は猫である」の公開済みのふりがな付き HTML 文書 [11] を対象として，その冒頭にある <ruby> 部分「吾輩（わがはい）は」へのページ内検索を複数のパターンで試した．それぞれに対するマッチの成否を○×で表し，問題のある挙動を下線で強調した．ブラウザはいずれも執筆時点での最新版を用い，各バージョンは Firefox 107.0.1, Chrome 108.0.5359.71, Safari 16.1 (17614.2.9.1.13) であり，利用した OS は Firefox/Chrome が Debian 11.5, Safari が macOS 12.6.1 である．

まず，Firefox の挙動は 1 節で述べたとおりである．ふりがなはテキストとみなされず，親文字と <ruby> 外の文字列のみがテキストとして連結される．親文字と後続文字列は連続するため，「吾輩は」というパターンはマッチする．

次に，Chrome は親文字・ふりがなの双方をテキストとみなすため，<ruby> 中に存在する文字列「吾輩」と「わがはい」をそのままパターンとすればいずれもマッチする．ただ，いずれのテキストも <ruby> 前後の文字列とは分断された別テキストとして扱われる．そのため，後続する文字「は」を追加したパターン「吾輩は」や「わがはいは」では，いずれも図 3 のようにマッチしない．

最後に，Safari はふりがなをテキストに含むが，それは親文字の次に連なるとみなされる．つまりテキストは，2.2 節で議論した <ruby> 未対応ブラウザの表示と同様の文字列となるが，<rp> の内容が無視される点のみが異なる．結果，ふりがなを含むパターン「わがはいは」がマッチする一方で，図 4 のように本来はマッチしないはずの「吾輩わがはいは」というパターンにまでマッチしてしま

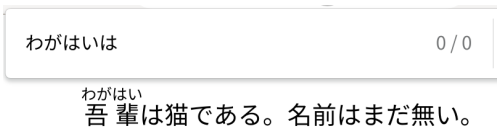


図 3　Chrome でのページ内検索例

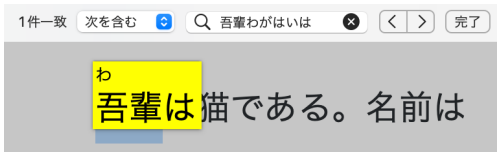


図 4　Safari でのページ内検索例

う．また，親文字と後続文字列による「吾輩は」というパターンにはマッチしない．

3 つのブラウザいずれにも問題はあるが，Firefox と Safari を比較した時，<ruby> 中の文字列がその前後と「どう連続するか」という解釈が異なると理解できる．しかし Chrome の挙動はいずれとも異なり，一文書中の文字列の連続性が無条件に失われ，あたかも別の文書であるかのような分断が発生する．

2.4　望ましい挙動

現状の主要ブラウザの挙動には，いずれも大きな問題がある．しかし現在，ふりがな付きテキストに対する検索として望ましい挙動が合意されているとも言い難い．

望ましい挙動を考える前提として，我々は日本語での「親文字」「ふりがな」二要素間に以下の関係が成り立つと考える．

(1) 両者で一つの塊をなす．
(2) 両者に前後関係はなく並列に存在する．
(3) 両者とも塊前後の文字列との連続性を持つ．

第一に，親文字とふりがなは両者で一つの塊である．全体を一つの <ruby> タグの塊として表現する既存仕様から見ても，これは自然な発想である．

第二に，親文字とふりがなに前後関係はない．仮に「吾輩（わがはい）」という表記が「わがはい（吾輩）」へと入れ替わった場合，表記の揺れとは解釈できても，意味が崩れたとまでは言えない．また「超電磁砲（レールガン）」のような自明でないふりがなは，複数の言葉が並列して一つの概念を表現する例である．

第三に，親文字とふりがなは共に，塊の前後との連続性を持つ．例えば発音上，「吾輩は」「わがはいは」は共に正しく，親文字「吾輩」とふりがな「わがはい」は共に後続の文字「は」へと繋がる．また，別の例「とんと見当（けんとう）がつかぬ」においても，ふりがな付きの塊がそれぞれ前後の「とんと」「がつかぬ」へと繋がり，「とんと見当がつかぬ」「とんとけんとうがつかぬ」と読めるため，同様の考えが成り立つ．

以上から我々は，ふりがな付きテキストの検索における親文字とふりがなは**区別せず検索できる**のが望ましい挙動であると考える．ページ内検索のユーザは，入力するパターンが二要素どちらであるか，あるいはそれが <ruby> の塊に含まれるか・含まれないか・塊から前後に出るか，などを意識せずに検索できる必要がある．

この望ましい挙動を行うページ内検索の実現には，テキスト中の任意の箇所で任意の数の塊が存在する文字列探索を行う必要がある．またこの塊は，親文字とふりがなという 2 つの文字列が「重なった」存在であるとも考えることができる．

3.　関連研究

文字列探索のアルゴリズムは古くから研究されている．正しく動かすだけならば，一文字ずつ力任せに照合する方法でも十分だが，より効率的な探索が行える工夫をしたアルゴリズムが数多く存在し，その代表例として KMP 法 [5] がある．しかし，これらはいずれもテキストが一列に並んでいることを前提としているため，今回のように「2 つの文字列どちらか」という分岐・合流があるテキストにそのまま当てはめることはできない．

正規表現によるマッチではパターンを柔軟に指定できるが[*4]，これも文字列探索と同様，テキスト側に柔軟性を持たせる手法ではない．

全文検索の分野では，膨大な数の文書から効率的に情報を見つけ出す手法が長年研究されてきた．しかしその多くは確率的アルゴリズムであり，再現率と適合率・精度とのトレードオフがあることはよく知られている [6]．これに対して本研究の対象は，ふりがなを含む一つの文書テキストに対する文字列探索である．検索対象が制限される反面，

[*4] https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html

得られる結果が安定している決定的アルゴリズムである上，多くの全文検索で前提とする大規模な事前計算も必要ない．

ふりがなの電子的な扱いを論じた研究は，国内を中心に複数存在する．しかし，文字列探索の問題として正面から捉えたものは見当たらない．ふりがなを取り上げた全文検索エンジンの一つ [13] では，ふりがなを他の本文から独立して検索するために，2.3 節で述べた Chrome と同じ不連続性の問題がある．また，日本語文書の電子処理としてふりがなの「除去」を前提とする研究 [12], [15] が多く見つかる．これらの存在は，現状のソフトウェアにおけるふりがな付き文字列の検索性の低さ・問題の大きさと，それを解決する研究の必要性を示唆している．

HTML の <ruby> に焦点を絞った研究の中にも，文字列探索の問題に触れているものがある．Firefox に <ruby> と縦書き機能を実装した研究 [9] では，親文字を示す <rb> タグの再導入によって検索性が向上する可能性を指摘している．しかしこれは <ruby> 中の文字列に限定した議論であり，その前後との不連続性や，親文字とふりがなの関係性から起こる検索性の低下には触れていない．また W3C・WHATWG による複数の関連文書においても，<ruby> による検索性の低下に触れた論考は見当たらない [1], [3], [8], [14].

2020 年，コロナウイルス禍のもとであらゆる教育がインターネットへの移行を強いられた際に，障害児教育で顕在化したふりがなと <ruby> タグの問題に対処した研究がある [16]．本研究が着目する文字列探索とは別の観点だが，ふりがなに対する電子的処理の支援が不十分であるという共通の問題を扱っている．

4. 分岐のあるテキストの文字列探索

2.4 節における望ましい挙動を実現するには，文字列という基本的なデータ構造から再考する必要がある．本研究では，ふりがな付き文字列テキストを**分岐・合流する文字列**であると捉え，データ構造と探索アルゴリズムを提案する．

4.1 データ構造

2.4 節に挙げた 3 つの関係性を実現するため，まずテキストがひと繋がりの文字列であるという前提を捨てる．関係性を素直に表現するなら，「**は猫(ねこ)である。**」に対して図 5 のようなグラフによるデータ構造が考えられる．文字列として，最初の「は」・親文字の「猫」・ふりがなの「ねこ」・後続の「である。」の 4 つが存在し，最初の文字は親文字とふりがなの両者へ繋がり，また両者は共通の後続文字へと繋がる．この時，親文字とふりがなは対等に存在するため，ふりがなが複数に増えた場合でも問題は起こらない．

本論文では図 5 のような構造の全体を**分岐文字列**と呼ぶ．これを文書全体に適用すると，図 6 のように文字列の分岐と合流が繰り返される構造になる．分岐と合流は文書の終わりまで，任意の位置・回数で交互に発生する．また，文書の最初や最後にもふりがなを振ることは可能なため，分岐文字列が分岐した状態で始まること・終わることも許容される．

分岐文字列の中で分岐・合流の基点となる個々の文字列を，テキストである分岐文字列全体との対比のために**部分テキスト**と呼ぶ．図 5 や図 6 では，四角で囲まれた文字列それぞれが部分テキストである．

次節で述べるとおり，分岐文字列の構造は概念上の理解に留まらず，実行時のメモリにおいても同様に表現する必要がある．

4.2 既存アルゴリズム適用の困難

前節の分岐文字列構造に対して既存の文字列探索アルゴリズムを適用できるなら，2.4 節の議論通りの文字列探索，つまり親文字とふりがなを区別しない探索が可能になる．しかし文字列探索の手法は文字通り，テキストが従来の文字列であるこ

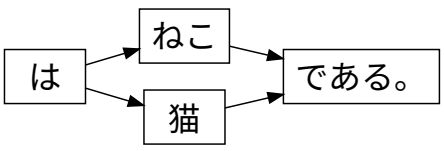


図 5　テキストの分岐構造の例 (1)

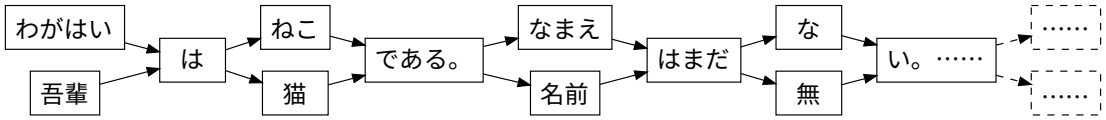


図 6　テキストの分岐構造の例 (2)

と，つまり分岐を持たない連なりであることを前提とする．これを分岐文字列に適用するのは困難である．

この前提を満たすための愚直な方法として，分岐を分解して作られる複数のテキストを順に文字列探索することが考えられる．図 5 の例であれば，二つの文字列「は猫である。」「はねこである。」を探索時に生成し，それぞれについて通常の文字列探索を行えば良い．

しかしこの考えが現実的でないことは，図 6 を見れば明らかである．本文冒頭の抜粋であるこの図に限っても <ruby> による分岐が 4 つ存在するため，素朴に行えば，計 $2^4 = 16$ 個のテキストに対する文字列探索が必要となる．以後も <ruby> が現れるたびにテキストは倍に増え続け，必要な探索数は指数関数的に増加するため，既存アルゴリズムの単純な適用は現実的でない．

4.3　力任せ法

分岐文字列には既存の文字列探索を適用できないため，独自の探索方法を考案する必要がある．まず最初に，通常の文字列探索での力任せ法を基礎とした方法を示す．力任せ法とは，文字通り力任せにテキスト・パターンのすべての位置から文字を順に照合する方法である．時間効率は悪いが，実装が単純になる利点がある．

力任せ法を分岐文字列に適用するには，テキスト中の文字列である部分テキストすべてを順に辿った上で，探索開始位置として部分テキスト中のすべての文字位置を列挙する必要がある．通常の文字列で文字位置を示す場合，整数の添字一つで十分である．しかし分岐文字列は複数の部分テキストを含むため，テキストの文字位置を定めるには「どの部分テキストか」と「部分テキスト中のどの文字位置か」の 2 つの情報が必要になる．

本節での方法は全体として，上記 2 つの情報を順に辿る単純なものである．ただし，パターンと部分テキストを照合している最中，部分テキスト側が途切れる場合が問題となる．通常の文字列探索の場合，テキスト末尾に到達すれば「マッチ失敗」として処理を終了できる．しかし分岐文字列には，その後続となる別の部分テキストが存在し得るため，直ちに処理を終わらせることはできない．よって，部分テキストが先に途切れた場合には「部分マッチ成功」とした上で，残りのパターンから次の部分テキストとの照合を再開する．また，ここでの「次の部分テキスト」は，分岐によって複数存在し得ることにも注意を要する．次の部分テキストを探索する際，内部ではグラフの幅優先探索を利用している．

図 7 は図 6 から「は猫(ねこ)」をテキストとして抜粋し，パターンと共に文字位置の情報を加えたものである．3 つの部分テキストは，それぞれ s_0，s_1，s_2 の識別子を持つ文字の配列と捉える．この分岐文字列テキストからパターン p「は猫」を探索する場合を想定する．

まず，s_0 の最初の文字から p の照合を開始する．それぞれの最初の文字を示す添字 0 から順に $s_0[i] == p[i]$ で照合するが，$i = 0$ の文字は共に「は」なので，最初の照合は成功する．

成功すると，次は添字を増やして $i = 1$ 同士の照合を行う．しかし s_0 には 1 文字しか存在しないためにテキスト側の文字列が先に途切れ，照合は行えない．この時，次の部分テキストがあるため

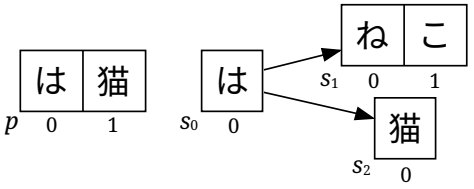


図 7　位置情報を付加したテキストとパターンの例

に「部分マッチ成功」として，その続きにある $p[1]$「猫」から照合を継続する．

ここで s_0 の「次の部分テキスト」は s_1，s_2 の 2 つ存在するため，$p[1]$ から継続する照合は 2 回行う．

1 回目は s_1「ねこ」との照合を行うが，初めの $s_1[0] == p[1]$ が成立せず失敗する．ここでテキスト全体の終端に（分岐したまま）達したので「マッチ失敗」となるが，あくまで継続した照合の 1 回目部分のみの失敗である．

2 回目は s_2「猫」との照合を行うが，これは 1 回目の成否には関わらない．まず $s_2[0] == p[1]$ が成功し，それぞれの添字を増やそうとするが，p は 2 文字しか存在しないためパターンの末尾に達する．よって「マッチ成功」として，2 回目部分の探索も終わる．

「部分マッチ成功」から継続した 2 回の照合を経て，$s_0[0]$ から開始する探索がようやく終わり，「$s_0[0]$ と $s_2[0]$ に 1 件マッチ」という結果を記録する．以上について，テキストにおける開始位置を一文字ずつずらし，同様の手順による探索を行う．

以上をまとめた擬似コードを図 8 に示す．本論文では，擬似コードの文法や利用可能な機能は JavaScript に準じている．関数 `bf()` は分岐文字列テキスト `text` からパターン `pat` を探索し, すべてのマッチ結果を配列として返す．また `matchFrom()` は，指定されたテキストの開始位置から幅優先探索を利用してパターンのマッチを試す関数であるが，定義を省略している．

まず，マッチは複数回起こり得るので，2 行目で配列 `matched` を準備する．これが最後の 10 行目で返されるマッチ結果となる．

```
1  function bf(text, pat) {
2    const matched = [];
3    for (const sid of text.sids) {
4      const str = text.getString(sid);
5      for (let i = 0; i < str.length; i++) {
6        const m = matchFrom(text, sid, i, pat);
7        if (m) matched.push(...m);
8      }
9    }
10   return matched;
11 }
```

図 8　力任せ法に基づく文字列探索

3 行目の `for...of` 文では，`text` 中の部分テキスト識別子のリスト `text.sids` に対して繰り返し，その要素を `sid` に入れる．この識別子とは，図 7 での s_0，s_1，s_2 のことである．4 行目でそれを使い，テキストから部分テキスト `str` を得る．

5 行目，`str` の全文字位置を整数 `i` として繰り返す．こうして 6 行目の時点で定まる `sid` と `i` が，本節冒頭に記した「2 つの情報」である．これらをテキストの探索開始位置として，`matchFrom()` によるマッチを試みる．これ以後は先程の例で詳しく説明したために省略するが，マッチ成立時に `matchFrom()` が返すのは配列である．これは，同じ位置から始めた探索でも複数の分岐経路でマッチする可能性があることと，後述するハイライトのためにすべての分岐経路の情報が必要になることが理由である．マッチがあれば，その内容を配列 `matched` に追加する．

4.4　力任せ法の改良

素朴な力任せ法は実装しやすいが，自力で一文字ずつ照合する処理の効率は高くない．本節では，多くのプログラミング言語で標準的に提供される既存文字列探索の機能を活用した改良を図る．

既存の文字列探索処理を分岐文字列にそのまま適用できないのは，マッチ結果が複数の部分テキストにまたがるためである．しかしパターンの全長が 1 であれば，複数部分テキストにまたがるマッチは起こりえない．また 2 文字以上のパターンにも当然，先頭の 1 文字目は存在する．

これらから，ある部分テキストの探索を開始する際，パターンの先頭文字 $p[0]$ に限って探索を行う改良を考えた．一つの部分テキストに限れば，その内容は通常の文字列である．そのため，言語処理系が提供する高速な標準機能を利用でき，探索開始位置としてすべての文字位置を指定する必要がなくなる．また，標準機能によって先頭の文字位置が判明した以降は従来の処理を踏襲できるため，コードの変更も少ない．

```
1  function bf2(text, pat) {
2    const matched = [];
3    for (const sid of text.sids) {
4      const str = text.getString(sid);
5      for (let i, j = 0;
             (i = str.indexOf(pat[0], j)) >= 0;
             j = i + 1) {
6        const m = matchFrom(text, sid, i, pat);
7        if (m) matched.push(...m);
8      }
9    }
10   return matched;
11 }
```

図 9　indexOf() による改良版

改良した擬似コードを図 9 に示す．改良前の図 8 との差は 5 行目のみである．図 8 では str.length 回, 愚直に 1 ずつ i を変化させるのに対して, 図 9 では JavaScript 標準に含まれる文字列探索メソッド String.prototype.indexOf()[*5]を活用している．

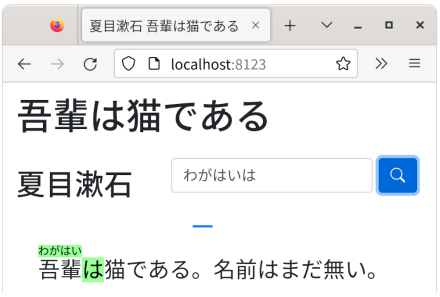
str.indexOf(pat[0], j) は，パターンの先頭文字が str の文字位置 j 以降に存在する位置を非負整数で返し，存在しない場合は -1 を返す．これによって，matchFrom() によるグラフ探索をすべての文字位置から行わず，1 文字目のマッチが分かっている位置のみに探索開始を限定することができる．

また一般に，あるパターンがテキストに含まれる回数は多くなく，ほとんどの文字位置での探索は 1 文字目から失敗する．そのため, 大半は indexOf() が直ちに -1 を返し，グラフ探索の回数が大幅に減ることが期待できる．

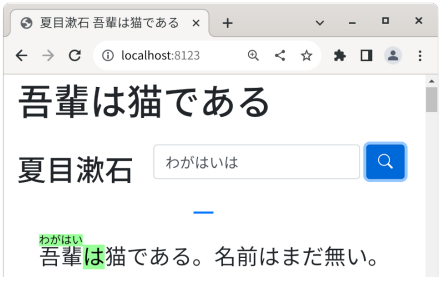
5. 実装

提案の有効性を評価・実証するため，我々は 4 節で論じた各機能をライブラリ **Ruby Finder** として実装した．Ruby Finder は，DOM オブジェクトからの分岐文字列テキストの構築とその文字列探索を実現する．ブラウザで動作するユーザレベルの JavaScript ライブラリであり，外部ライブラリも不要なため，任意の Web ページへの組み込みが容易である．前述の主要 3 ブラウザすべてで

[*5] https://tc39.es/ecma262/2022/multipage/text-processing.html#sec-string.prototype.indexof



(a) Firefox



(b) Chrome

図 10　主要ブラウザでの動作例

の正しい動作と実用的な速度を確認している．

図 10 がその動作画面である．<ruby> によるふりがなを含む任意の Web ページで，ブラウザ組み込みのページ内検索に類似した文字列探索と，マッチ位置を識別するためのハイライトが行える．

5.1 利用例

ライブラリを利用する際のコード例を図 11 に示す．1 行目，本文の <body> 以下を表す document.body を引数にコンストラクタを呼ぶと，その木構造を解析して分岐文字列テキストが構築される．2 行目, パターン文字列を引数に find() メソッドを呼ぶと文字列探索が実行され，結果が found に返される．マッチ件数は found.count で

```
1  const finder = new RubyFinder(document.body);
2  const found = finder.find('吾輩は');
3  console.log(found.count);
4  // マッチ箇所のハイライトと削除
5  found.highlightResults();
6  found.removeHighlighting();
```

図 11　分岐文字列探索ライブラリの利用例

取得できる．

ブラウザのページ内検索の改良が動機であったため，結果の視認性を高めるハイライト機能も実装した．5 行目，マッチ結果に対して `highlightResults()` を呼ぶと，マッチした複数の DOM ノードがハイライトされ，6 行目で `removeHighlighting()` を呼ぶと元に戻る．これらハイライト機能の API は，Firefox のブラウザ拡張向け API[*6] を参考にした．

現実には，これらの機能をページ閲覧者の動きに対応して対話的に呼び出す必要がある．多くの場合，`new RubyFinder()` による分岐文字列テキストの構築は `DOMContentLoaded` イベント発生時に一度だけ行われ，`find()` の引数とするパターンは `<input>` 等のフォームから取得し，取得のタイミングと `find()` や `highlightResults()` の呼び出しは `<button>` のクリックイベントに対応する，といった利用法を想定している．

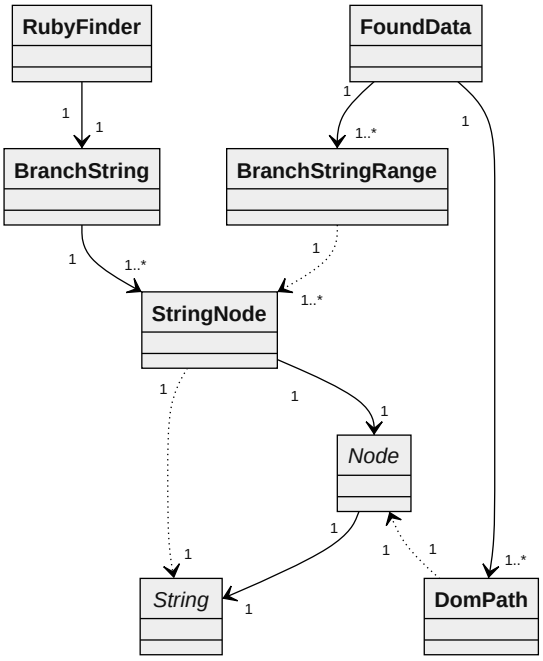


図 12　クラス構成

5.2　構成

ライブラリのクラス構成を図 12 に示す．全体は 6 つの JavaScript クラス[*7]で構成されるが，ライブラリユーザが意識する必要があるのは `RubyFinder` と `FoundData` の 2 つのみである．

`RubyFinder` がライブラリの全体を司り，DOM の木構造から分岐文字列への変換も担当する．コンストラクタに DOM ノードを渡すと，それを基点とした XPath クエリで関連ノードをフィルタする．テキストノードについてはそのまま，`<ruby>` については直下の親文字と `<rt>` から，部分テキストに対応する DOM ノードを収集する．それらを元にテキスト本体である `BranchString` オブジェクトを構築し，将来の文字列探索のために保持する．

`BranchString` は，テキストである分岐文字列を表すライブラリの中核である．部分テキスト `StringNode` を複数持ち，その中には DOM ノード `Node` と内容の文字列 `String` を持つ．`find()` メソッドによって文字列探索を行い，その結果を `FoundData` として返す．内部のデータ構造とアルゴリズムの概要は 4 節で述べたとおりだが，次節では設計上の工夫を述べる．

`FoundData` は探索結果をまとめたオブジェクトで，マッチ件数を持つほか，マッチした DOM ノード群にハイライトの追加・削除を行う．通常の文字列では，始点と終点を示す一組の整数があればハイライト対象となる範囲一つを示せる．しかし分岐文字列での範囲を一つ示すには，4.3 節末でも触れたように，マッチしたすべての部分テキストの経路を示す必要があるため，複数の DOM ノードを持たなければならない．

`BranchStringRange` はそのマッチ経路を表す．`StringNode` の配列と共に始点・終点を表す整数を持ち，それぞれが最初のノードと最後のノードにおけるマッチ開始・終了の文字位置を示す．他の中間ノードではすべての文字にマッチしているため，始点・終点以外の文字位置は必要ない．

`DomPath` はハイライトの実装で必要となるクラスで，DOM ノードを特定する整数リストを持つ．ハイライトでは DOM を書き換えるため，探索結

[*6] https://developer.mozilla.org/docs/Mozilla/Add-ons/WebExtensions/API/find

[*7] 2015 年に ECMAScript 6 で導入された構文を利用 https://262.ecma-international.org/6.0/#sec-class-definitions

果として保持するノードがルートから切り離され，無効になり得る．そのため，再度ハイライトを行う際はツリーのルートから同じ位置のノードをたどり，それをハイライトの対象とする．

5.3 BranchString と DOM 処理の分離

BranchString 設計時の工夫として，DOM 関連の処理を外部へ分離することにした．これは実装上の拡張性を残すためである．

文字列探索には部分テキストの中身 (JavaScript 標準の String) さえあれば十分だが，その結果をハイライトするためには，マッチした String が含まれる DOM ノードを知る必要がある．ただ DOM ノードから中身の String を得ることは容易なため，BranchString を素朴に実装する場合，DOM ノードのみを保持することが考えられる．

しかし String と DOM ノードを独立して扱えば，同じ実装で HTML 以外の文書を扱うことが可能となるほか，HTML の処理にも利点が生まれる．高速化の手段として Web Worker[*8]や WebAssembly [4] の利用が考えられるが，これらはいずれも DOM の利用に制限をもたらすため，関連処理の分離が前提となる．この将来を見越して，あらかじめ分離する設計とした．

分離実現のため，各部分テキストを StringNode として表現し，その中で String とハイライト用の任意のデータを組として持つ構造にした．BranchString (テキスト) は複数の StringNode (部分テキスト) から成るが，文字列探索処理では StringNode 中の String だけを利用し，結果としては StringNode を返す．これにより，String とデータとの対応関係を保持しつつ，データの中身には依存しない処理を実現した．

DOM を知る上位層である RubyFinder は，任意データ部分に DOM の Node を格納してから，下位の BranchString と StringNode を生成する．文字列探索の結果である FoundData に StringNode を持たせれば，ハイライト時にその対象の DOM ノードを認識できる．

[*8] https://html.spec.whatwg.org/multipage/workers.html

6. 評価

実装の実用性を評価するため，Ruby Finder による文字列探索の実行時間を計測した．各主要ブラウザで計測を行ったが，それらはすべて Node.js[*9] 19.2.0 上の Playwright[*10] 1.28.1 を使い，ヘッドレスブラウザとして実行した．Playwright と共に実行した各ブラウザのバージョンは以下である．

- Firefox 106.0
- Chromium 108.0.5359.29
- WebKit 16.4

ここでの Chromium[*11]と WebKit[*12]は，それぞれ商用版の Chrome と Safari の基盤となるオープンソースソフトウェアである．商用版には各社独自の機能が加えられるが，JavaScript 処理系を含め，ブラウザとしての機能は同一と言える．

評価の内容を 3 つに分け，表 2 中にまとめた．対象とした HTML 文書 2 つの詳細を表 2 (a) に示す．1 つ目の短編小説「手紙」[10] にはふりがなが含まれる上，現在転送される HTML ファイルサイズの中央値が統計上 30 KB ほどであることから[*13]，それに近い現実的なサイズのデータとして選んだ．2 つ目は従来取り上げてきた長編小説「吾輩は猫である」[11] だが，そのサイズは HTML のみで 1 MB を超え，公開済みの青空文庫[*14]全作品中でも上位に入る．日常的な例ではなく，あくまで最悪ケースに近い例外的な例として選んだ．

ここでの本文文字数は，比較対象とした通常文字列 document.body.textContent の属性値 length を指す．またそれぞれのファイルには，UTF-8 への変換，不要な外部ファイル参照の削除など，本文の意味を保った最小限の変更を施した．

各ブラウザでの文字列探索の実行時間について，短編を表 2 (b) ，長編を表 2 (c) にそれぞれ示す．実行時間は 10 回計測後に幾何平均を取り，マイクロ秒単位で四捨五入した．実行環境は Intel x64

[*9] https://nodejs.org/
[*10] https://playwright.dev/
[*11] https://www.chromium.org/Home/
[*12] https://webkit.org/
[*13] https://httparchive.org/reports/page-weight
[*14] https://github.com/aozorabunko/aozorabunko

表 2　評価結果

(a) 評価に用いた HTML ファイル

作品名	バイト数	`<ruby>` 数	本文文字数
手紙	44,312	101	12,737
吾輩は猫である	1,497,214	9,214	369,230

(b) 文字列探索の実行時間: 手紙 (µsec)

探索方式	Firefox	Chromium	WebKit
力任せ	16,042	33,738	18,598
改良力任せ	4,315	1,819	3,376
通常	912	471	480

(c) 文字列探索の実行時間: 吾輩は猫である (µsec)

探索方式	Firefox	Chromium	WebKit
力任せ	77,940	343,446	186,372
改良力任せ	8,130	10,285	12,688
通常	1,860	808	887

CPU (Core i7-3770, 2.30GHz)，メモリ 8 GB，OS は Debian 11.5 である．また，できる限り安定した計測結果を得るため，Hyper Threading の無効化や CPU Governor を `performance` にするといった事前設定を行った上，計測前には実際の計測と同じプログラムを同じ回数だけ実行した．

表 2 (b) (c) 共に，本論文で議論した力任せ・改良力任せに加え，比較対象として通常の文字列探索による実行時間も記載した．これは，`document.body.textContent` で得られる通常文字列に対して 4.4 節と同様に `indexOf()` を繰り返し使い，すべての結果を得るまでの実行時間である．

表 2 (b) は，現実的な大きさの HTML に対する計測である．単純な力任せ法を含め，すべてのケースで十分高速であることを確認できた．最も遅かった Chromium での力任せ法も約 0.034 秒で探索が完了した．これは通常の文字列探索に比べると 72 倍近く遅いが，改良力任せ法によって 18.5 倍以上高速化された．他のブラウザでも同様に，単純力任せでも十分に高速であることと，改良によって大幅な高速化が起こることが確認された．

表 2 (c) は，通常扱わないほど巨大な HTML に対する計測である．改良力任せ法で大きな改善が起こる点は表 2 (b) と同様であり，最も遅かった WebKit でも約 0.013 秒で探索が完了した．一方，改良前の力任せ法では表 2 (b) 以上の劣化が起こった．Chromium では探索に約 0.34 秒掛かる上，表 2 (b) では約 72 倍だった通常との劣化比は約 425 倍に広がり，改良力任せと通常の比も同様に開いている．改良の前後共に，通常文字列探索と比べた計算量の増加が推測されるため，アルゴリズム改良の余地は大きい．

7. おわりに

本論文では，日本語文書を中心に使われるふりがなへの電子的支援が不十分な点に着目して，その自然な文字列探索を実現する手法を提案・実装した．多様な環境での動作実績と実行時性能の評価から，既に十分な実用性を備えていると考える．

今後の課題として，オープンソースソフトウェアとしての公開，ブラウザ拡張の作成，多様な HTML マークアップへの対応，KMP 法を応用したアルゴリズムの改良，そして主要ブラウザへの直接的な実装が考えられる．さらに，本研究の基盤部分は JavaScript や HTML に限定されないため，タグ付き PDF や簡易マークアップのなどの他の文書形式に対する応用を目指す．

参考文献

[1] Campbell, A., Cooper, M. and Kirkpatrick, A.: H62: Using the ruby element, Techniques for WCAG 2.1, Technical report, W3C Accessibility Guidelines Working Group (2022). `https://www.w3.org/WAI/WCAG21/Techniques/html/H62` (2022 年 11 月 28 日閲覧).

[2] Deveria, A., Schoors, L. and the GitHub community: Ruby annotation | Can I use... Support tables for HTML5, CSS3, etc. `https://caniuse.com/ruby` (2022 年 11 月 24 日閲覧).

[3] Dürst, M., Suignard, M., Sawicki, M., Ishikawa, M. and Texin, T.: Ruby Annotation, W3C Recommendation, W3C (2001). `http://www.w3.org/TR/ruby/` (2022 年 11 月 22 日閲覧).

[4] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A. and Bastien, J.: Bringing the Web up to Speed with WebAssembly, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, New York, NY, USA, Association for Computing Machinery, p. 185–200 (2017). DOI: 10.1145/3062341.3062363.

[5] Knuth, D. E., Morris, Jr., J. H. and Pratt, V. R.: Fast Pattern Matching in Strings, *SIAM Journal on Computing*, Vol. 6, No. 2, pp. 323–350 (1977).

[6] Manning, C. D., Raghavan, P. and Schütze, H.: *Introduction to Information Retrieval*, Cambridge University Press (2008).

[7] W3C and WHATWG: Agreement on HTML Ruby Markup (2022). `https://www.w3.org/2022/02/ruby-agreement` (2022 年 11 月 26 日閲覧).

[8] WHATWG (Apple, Google, Mozilla and Microsoft): HTML Living Standard, Technical report (2022). `https://html.spec.whatwg.org/` (2022 年 11 月 26 日閲覧).

[9] 塩澤元，松原俊一，Dürst, M. J.：ルビと縦書きの Web ブラウザへの実装とその背景，情報処理学会研究報告 情報基礎とアクセス技術 (IFAT), Vol. 2011-IFAT-102, No. 7, pp. 1–8 (2011).

[10] 夏目漱石：手紙，青空文庫 (入力: 柴田卓治，校正: しず). 底本: 「硝子戸の中」角川文庫，角川書店. `https://www.aozora.gr.jp/cards/000148/card798.html` (2022 年 11 月 23 日閲覧).

[11] 夏目漱石：吾輩は猫である，青空文庫 (入力: 柴田卓治，校正: 田尻幹二，高橋真也，しず，瀬戸さえ子，おのしげひこ，渡部峰子) (1905). 底本: 「夏目漱石全集 1」ちくま文庫，筑摩書房. `https://www.aozora.gr.jp/cards/000148/card789.html` (2022 年 11 月 23 日閲覧).

[12] 粟津妙華，高田雅美，城和貴：活字データの分類を用いた進化計算による近代書籍からのルビ除去，情報処理学会論文誌数理モデル化と応用 (TOM)， Vol. 8, No. 1, pp. 72–79 (2015).

[13] 山口昌也：構造化テキストに対応した全文検索システム『ひまわり』，雑誌『太陽』による確立期現代語の研究:『太陽コーパス』研究論文集 (国立国語研究所，編)，国語研究所報告 122，博文館新社，pp. 49–82 (2005).

[14] 千葉弘幸，枝本順三郎，Ishida, R.，石野恵一郎，加藤誠一，小林龍生，小林敏，McCully, N.，小野澤賢三，Sasaki, F.，下農淳司，塩澤元，薛富僑 (以前の版の編者: 阿南康宏)：日本語組版処理の要件，W3C Note, W3C (2020). `https://www.w3.org/TR/jlreq/` (2022 年 11 月 22 日閲覧)
.

[15] 久米朋子, 江見圭司：日本語学習者を対象とした日本文学作品の読解支援サイト『JL 文庫』の作成 〜『インターネット図書館青空文庫』を題材として〜，情報処理学会研究報告 コンピュータと教育 (CE), Vol. 2014-CE-123, No. 9, pp. 1–8 (2014).

[16] 濱松若葉, 星眞維美, 中川美枝子, 柴田邦臣：Web ページにおけるスクリーンリーダとルビの共存 〜障害児向け学習サイトの事例から〜，研究報告 音声言語情報処理 (SLP), Vol. 2020-SLP-133, No. 1，pp. 1–6 (2020).

[17] 小林龍生：ネットワーク社会でのルビの再評価 HTML、Unicode に即して，情報処理学会研究報告 情報メディア (IM)，Vol. 1998, No. 106 (1998-IM-034), pp. 17–22 (1998).