Zastosowania teorii automatów do przetwarzania dokumentów XML

Autoreferat rozprawy doktorskiej

Paweł Parys

Maj 2011

1 Dokumenty XML

Przedstawiana rozprawa dotyczy szybkich algorytmów służących do przetwarzania dokumentów XML. Format XML jest ostatnio bardzo popularnym sposobem zapisu danych. Dokument XML to odpowiednio sformatowany tekst. Poniżej znajduje się przykładowy dokument.

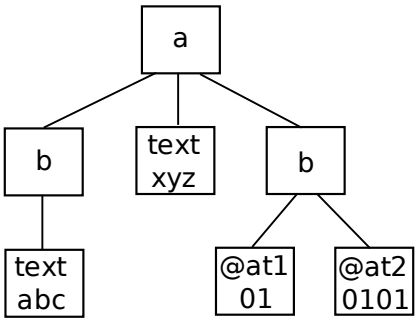
⟨a⟩
  ⟨b⟩abc⟨/b⟩
  xyz
  ⟨b at1 = "01" at2 = "0101"⟩⟨/b⟩
⟨/a⟩

W dokumencie takim możemy wyróżnić *znaczniki*. Są to fragmenty otoczone nawiasami trójkątnymi. Znaczniki zaczynające od znaków ⟨/, na przykład ⟨/a⟩, nazywamy znacznikami *zamykającymi*. Pozostałe znaczniki to znaczniki *otwierające*. Pierwsze słowo pojawiające się wewnątrz nawiasów trójkątnych to etykieta znacznika (na przykład etykietą ⟨b at1 = "01" at2 = "0101"⟩ jest b). Znaczniki otwierające i zamykające występują zawsze w parach: znacznik otwierający otwiera *element*, a znacznik zamykający o tej samej etykiecie go zamyka. Pary te muszą tworzyć poprawne nawiasowanie, czyli na przykład napis ⟨a⟩⟨b⟩⟨/a⟩⟨/b⟩ nie jest dozwolony. Zatem dokument XML można rozumieć jako drzewo, nazywane *drzewem dokumentu*, którego wierzchołkami są poszczególne elementy. Przykładowo dla powyższego dokumentu mamy korzeń o etykiecie a, mający dwóch synów o etykiecie b.

W dokumencie XML mamy również tekst pisany poza znacznikami. Dla każdego maksymalnego fragmentu tekstu możemy w drzewie umieścić odpowiadający mu wierzchołek. Wierzchołki takie oznaczymy specjalną etykietą text, nie występującą nigdzie indziej. Natomiast sam fragment tekstu nazwiemy *wartością* (lub *daną*) występującą w danym wierzchołku.

Trzecim rodzajem wierzchołków będą atrybuty. Przykładowo, występujący w powyższym dokumencie znacznik ⟨b at1 = "01" at2 = "0101"⟩ ma dwa atrybuty, mają one etykiety at1 i at2 oraz wartości odpowiednio 01 i 0101. Atrybuty mogą występować wyłącznie wewnątrz znaczników otwierających. Wierzchołki odpowiadające atrybutom również umieszczamy w drzewie dokumentu. Aby odróżnić atrybuty od elementów, ich etykiety w drzewie poprzedzimy znakiem @.

Drzewo odpowiadające naszemu przykładowemu dokumentowi znajduje się na poniższym rysunku.



Rysunek 1. Drzewo reprezentujące przykładowy dokument XML

2 Zapytania XPath

Ważnym zagadnieniem jest odnajdywanie w dokumentach XML danych, które spełniają zadane kryteria. Często korzysta się przy tym z języka XPath, służącego do zapisywania zapytań. Problem algorytmiczny rozważany w rozprawie jest następujący: dane jest zapytanie XPath oraz dokument XML, należy znaleźć wszystkie wierzchołki drzewa dokumentu spełniające to zapytanie.

W XPath mamy dwa rodzaje zapytań: zapytania unarne (nazywane też predykatami lub filtrami) oraz zapytania binarne (nazywane też wyrażeniami ścieżkowymi). Podstawowy rodzaj zapytań to zapytania unarne; wyrażenia ścieżkowe mają raczej charakter pomocniczy, są używane wewnątrz zapytań unarnych. Zapytanie unarne dla ustalonego drzewa dokumentu wyznacza zbiór jego wierzchołków. Innymi słowy, przy ustalonym drzewie i jego wierzchołku możemy pytać, czy zapytanie jest prawdziwe w tym wierzchołku, czy nie. Typowe zapytanie unarne to `a`; sprawdza ono, czy etykieta wierzchołka to `a` (równoważnie: zwraca wszystkie wierzchołki o etykiecie `a`). Z kolei wyrażenie ścieżkowe dla ustalonego drzewa dokumentu wyznacza zbiór par wierzchołków (x, y). Innymi słowy, przy ustalonym drzewie i wierzchołku x drzewa możemy pytać, jakie wierzchołki y są z niego osiągalne za pomocą tego wyrażenia ścieżkowego. Intuicyjnie, wyrażenie ścieżkowe opisuje ścieżkę między wierzchołkiem x a wierzchołkiem y (niekoniecznie najkrótszą). Typowe wyrażenie ścieżkowe to `parent/child`; wyznacza ono takie pary (x, y), że z x można dojść do y idąc najpierw do ojca, a następnie do dziecka (zatem x jest bratem y lub $x = y$). W zapytaniach unarnych możemy zagnieżdżać wyrażenia ścieżkowe i odwrotnie. Przykładem zapytania unarnego może być `child[b]`. Wyznacza ono te wierzchołki, których syn ma etykietę `b`. Używana przez nas definicja języka XPath znajduje się w Rozdziale 1.2.2 przedstawianej rozprawy.

Zauważmy, że wszystkie przykładowe zapytania zaprezentowane do tej pory nie odwołują się w ogóle do danych, sprawdzają jedynie strukturę drzewa i etykiety wierzchołków. Fragment języka XPath, który zawiera tylko takie zapytania, nazywany jest CoreXPath.

Zwykle zakłada się, że etykiety znaczników oraz nazwy atrybutów (w przeciwieństwie do danych tekstowych czy wartości atrybutów) pochodzą z ustalonego, niewielkiego skończonego zbioru. Zresztą nawet bez tego założenia możemy utożsamić ze sobą wszystkie nazwy, które nie są użyte w zapytaniu, gdyż z punktu widzenia zapytania są one nierozróżnialne. Możemy zatem myśleć o drzewie dokumentu jako o drzewie etykietowanym elementami pewnego skończonego zbioru.

Z kolei zapytanie możemy przekształcić do skończonego automatu na drzewach. Model automatu musi być odpowiednio zaadoptowany do sytuacji, gdyż standardowo automat skończony akceptuje lub odrzuca drzewo, podczas gdy my chcemy wyznaczać zbiór wierzchołków drzewa. Jedne z możliwych rozwiązań jest takie, że rozszerzamy alfabet. Jeden z wierzchołków drzewa będzie wyróżniony. Okazuje się, iż dla danego zapytania możemy skonstruować automat, który zaakceptuje drzewo z jednym wierzchołkiem wyróżnionym wtedy i tylko wtedy, gdy wierzchołek ten spełnia nasze zapytanie. Bardzo łatwo, po prostu symulując automat, możemy w czasie liniowym względem rozmiaru drzewa sprawdzić, czy drzewo z zaznaczonym wierzchołkiem będzie zaakceptowane, a zatem czy dany wierzchołek spełnia nasze zapytanie. Okazuje się jednak, że możemy znacznie więcej: możemy w jednym przebiegu, w czasie liniowym względem rozmiaru

drzewa, wyznaczyć od razu wszystkie wierzchołki takie, że jeśli je zaznaczymy, to automat zaakceptuje. Oznacza to, iż w czasie liniowym możemy wyznaczyć wszystkie wierzchołki spełniające dane zapytanie. Trzeba jednak zastrzec, iż rozmiar automatu może być wykładniczy względem rozmiaru zapytania. Zatem dostajemy algorytm liniowy ze względu na rozmiar dokumentu, lecz wykładniczy ze względu na rozmiar zapytania. Automatowe techniki obliczania zapytań są opisane w [Nev02].

Inne podejście do ewaluowania zapytań XPath opiera się na programowaniu dynamicznym. Umożliwia ono wyznaczenie wszystkich wierzchołków drzewa spełniających dane zapytanie w czasie liniowym zarówno ze względu na rozmiar dokumentu, jak i na rozmiar zapytania [GKP05]. Idea jest bardzo prosta: dla każdego podwyrażenia wyznaczamy zbiór wierzchołków, które spełniają to podwyrażenie. Jest to możliwe wyłącznie dla podwyrażeń, które są zapytaniami unarnymi, gdyż liczba par spełniających wyrażenia ścieżkowe może być kwadratowa względem rozmiaru drzewa, więc obliczanie wszystkich takich par byłoby za wolne. Okazuje się jednak, iż wystarczy dla każdego x stwierdzić, czy istnieje y taki, że (x, y) spełnia dane wyrażenie ścieżkowe.

3 Rzeczywiste systemy korzystające z XPath

Mimo iż opisany powyżej algorytm został opracowany już kilka lat temu, nie jest on używany w powszechnie używanych programach komputerowych korzystających z XPath. Już w [GKP02, GKP05] pokazano, że występujące w praktyce programy wyliczające zapytania XPath działają bardzo nieefektywnie. Konkretnie mają one (przy pewnych zapytaniach) złożoność $O(|t|^{|\varphi|})$, gdzie $|t|$ to rozmiar dokumentu, a $|\varphi|$ rozmiar zapytania. Jako że eksperymenty przeprowadzone w tych artykułach są już dosyć stare, powtarzamy je w niniejszej rozprawie, korzystając z najnowszych wersji programów. Co do zasady wyniki naszych eksperymentów są takie same.

Jak wygląda zapytanie, które powoduje taki czas działania? Jest ono bardzo proste. Po prostu sprawdzamy, czy istnieje syn, który ma ojca, który ma syna, który ma ojca, który ma syna... Przykładowe zapytanie o rozmiarze 4 w języku XPath wyglądałoby następująco:

```
child/parent/child/parent/child/parent/child/parent
```

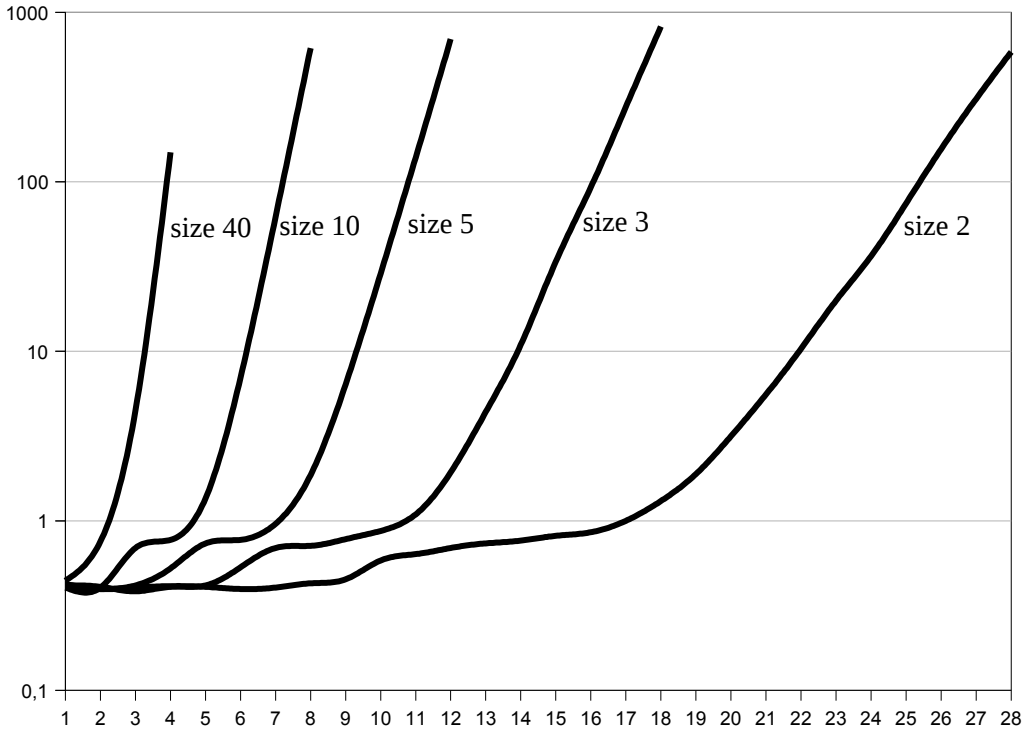
W jaki sposób wygląda naiwne wyliczanie takiego wyrażenia. Po prostu rekurencyjnie próbujemy iść taką ścieżką, próbując każdy kolejny krok wykonać na wszystkie możliwe sposoby. Jednak wszystkich możliwych ścieżek jest bardzo dużo mimo, że wszystkie one prowadzą od wierzchołka do niego samego. Na tym przykładzie widzimy także, jak za pomocą programowania dynamicznego możnaby łatwo przyspieszyć wyliczanie wyrażenia. Po prostu dla każdego sufiksu naszego wyrażenia ścieżkowego obliczamy czy z dowolnego wierzchołka x istnieje ścieżka pasująca do tego sufiksu. Łatwo wyliczyć to dla dłuższego sufiksu korzystając z wyników dla krótszego sufiksu.

Nawiasem mówiąc, analogiczne polecenie wydane w linii poleceń Linuxa również powoduje sprawdzanie wszystkich możliwości i bardzo długi czas działania. Chodzi o polecenie

```
ls */../*/../*/../*/..
```

Tutaj jednak jest to w jakimś stopniu usprawiedliwione, gdyż semantyka jest taka, iż za gwiazdki należy powstawiać nazwy katalogów na wszystkie możliwe sposoby, a następnie przekazać wszystkie powstałe w ten sposób napisy jako parametry do polecenia `ls`. Natomiast w przypadku języka XPath przeglądanie wszystkich możliwości zupełnie nie ma uzasadnienia, gdyż interesuje nas tylko, czy możliwość przejścia danym sposobem istnieje, czy nie istnieje.

Przeprowadziliśmy testy trzech bibliotek. Były to: Xalan w wersji 2.7.1, Saxon w wersji 9.3.0.4 (HE) oraz libxml2 w wersji 2.6.31. Są to jedne z najbardziej popularnych obecnie systemów. Wyniki testowania znajdują się w Rozdziale 1.3 przedstawianej rozprawy. Poniżej przedstawiamy jeden z wykresów.



Rysunek 2. Czas działania programu Xalan podczas testów (w sekundach), w zależności od rozmiaru zapytania (oś X) oraz rozmiaru dokumentu (poszczególne linie reprezentują różne rozmiary dokumentu)

4 Zapytania odwołujące się do danych

Jak można się spodziewać, język XPath pozwala także na pisanie zapytań odwołujących się do danych (czyli fragmentów tekstu lub wartości atrybutów). Ważne jest przy tym, że wartości te można porównywać między sobą. Przykładowe zapytanie odwołujące się do danych to

$$\texttt{child}[\texttt{@at1}] = \texttt{descendant/child}[\texttt{@at2}].$$

Wybiera ono takie wierzchołki x, które mają dziecko y o etykiecie `@at1` oraz mają potomka mającego dziecko z o etykiecie `@at2`, przy czym wartość danej w y i w z muszą być takie same. Jeśli przypomnimy sobie, że dzieci o etykietach zaczynających się od @ oznaczają atrybuty, możemy to powiedzieć inaczej. Zapytanie to wybiera takie wierzchołki x, które mają potomka y takiego, że wartość atrybutu `@at1` w x oraz atrybutu `@at2` w y są takie same.

W przypadku takich zapytań można także użyć programowanie dynamicznego. Dostaje się w ten sposób algorytmy wielomianowe (ale już nie liniowe) ze względu na rozmiar dokumentu t, jak i ze względu na rozmiar zapytania φ. Najlepszy znany algorytm, działający dla dowolnych zapytań z XPath 1.0, ma złożoność $O(|\varphi|^2|t|^4)$ [GKP03] (patrz także [GKP05]). Dla rozważanego w przedstawianej rozprawie fragmentu FOXPath algorytm używający programowania dynamicznego ma złożoność $O(|\varphi||t|^2)$.

Przedstawianą rozprawę, jak również artykuły na których jest ona oparta: [BP08], [Par09], [BP10] oraz [BP], można uznać za uogólnienie podejścia automatowego do zapytań, które odnoszą się do wartości atrybutów i tekstu. Zgodnie z terminologią stosowaną w [BK09], rozważany przez nas fragment języka XPath nazywa się FOXPath (aczkolwiek my nie uwzględniamy identyfikatorów wierzchołków). Pierwszy algorytm dla tego fragmentu mający złożoność liniową ze względu na rozmiar dokumentu podaliśmy w [BP08]. Stała występująca przy czasie liniowym w tym algorytmie była jednak wykładnicza ze względu na rozmiar zapytania. Z drugiej strony, algorytm

ten obsługuje także rozszerzenie języka XPath, w którym mogą występować dowolne wyrażenie regularne jako wyrażenia ścieżkowe. Będziemy używać nazwy *Regular XPath* na to rozszerzenie języka XPath, w przeciwieństwie do fragmentu *FOXPath*, w którym wyrażenia ścieżkowe nie mogą używać gwiazdki Kleene'a, co jest zgodne ze specyfikacją języka XPath [CD99]. Algorytm w [BP08] używa metod algebraicznych, takich jak monoidy skończone oraz jednorodne drzewa rozkładu Simona. W rozprawie przedstawiamy nieco inny algorytm, mający tą samą złożoność, lecz używający automatów deterministycznych zamiast monoidów.

Następnie, w [Par09], podaliśmy algorytm działający w czasie liniowym ze względu na rozmiar dokumentu i wielomianowym ze względu na rozmiar zapytania. Ten algorytm korzysta z tego, że wyrażenia ścieżkowe są z FOXPath (w którym nie można zapisać dowolnych wyrażeń regularnych). Zatem ten algorytm nie działa dla zapytań z Regular XPath, tylko z FOXPath.

Rozprawa opisuje także trzeci, niepublikowany wcześniej algorytm, który jest prostszą wersją algorytmów z [BP08] and [Par09]. Ma on złożoność $O(|t| \log |t|)$ ze względu na rozmiar dokumentu t, wielomianową złożoność ze względu na rozmiar zapytania i działa dla całego Regular XPath. Prawdopodobnie spośród czterech przedstawianych algorytmów właśnie ten może być najbardziej użyteczny w praktyce. Jest łatwiejszy do zrozumienia i zaimplementowania, co przypuszczalnie rekompensuje dodatkowy czynnik $\log |t|$.

Następnie, rozprawa opisuje algorytm zaprezentowany w [BP10], który działa dla zapytań z Regular XPath w czasie liniowym ze względu na rozmiar dokumentu i wielomianowym ze względu na rozmiar zapytania i wysokość dokumentu. Warto przy tym zwrócić uwagę, że typowy dokument XML, nawet bardzo duży, ma bardzo małą wysokość.

Te cztery algorytmy stanowią główną zawartość przedstawianej rozprawy. Podsumowuje to poniższe twierdzenie.

Twierdzenie 1
Niech t będzie dokumentem XML, a φ zapytaniem z Regular XPath. Zbiór wierzchołków t spełniających φ może być obliczony w czasie

- $O(|\varphi|^3|t| \log |t|)$, *lub*
- $O(2^{O(|\varphi|)}|t|)$, *lub*
- *liniowym w $|t|$, wielomianowym w $|\varphi|$ oraz w wysokości t, lub*
- *gdy φ jest z FOXPath—w czasie $O(|\varphi|^3|t|)$.*

Powyższe twierdzenie mówi o obliczaniu zapytań unarnych, wybierających wierzchołki. Jak już wspomnieliśmy, w XPath istnieją także wyrażenia ścieżkowe, wybierające pary wierzchołków. Co do zasady, wyrażenia ścieżkowe nie mogą być obliczane w czasie liniowym ze względu na rozmiar dokumentu, gdyż czasami kwadratowo wiele par może spełniać takie zapytanie. Możliwe jest jednak znajdowanie wyniku takiego zapytania w czasie liniowym ze względu na sumę: liczba zwracanych par plus rozmiar dokumentu. Co więcej, prezentujemy algorytmy działające na zasadzie tzw. stałego opóźnienia: pierwsza para spełniająca α znajdowana jest w czasie liniowym ze względu na rozmiar dokumentu, a każda kolejna para w czasie stałym. Zatem, gdy ktoś chce znaleźć tylko jedną, dowolną parę (lub co najwyżej liniową liczbę par), czas działania jest liniowy ze względu na rozmiar dokumentu.

Twierdzenie 2
Niech t będzie dokumentem XML, a α wyrażeniem ścieżkowym z Regular XPath. Wszystkie pary wierzchołków t spełniające α mogą być obliczone, jedna po drugiej, w czasie

- *pierwsza para: $O(|\alpha|^3|t| \log |t|)$, każda następna para: $O(|\alpha|^3 \log |t|)$, lub*
- *pierwsza para: $O(2^{O(|\alpha|)}|t|)$, każda następna para: $O(2^{O(|\alpha|)})$, lub*
- *pierwsza para: liniowym w $|t|$, wielomianowym w $|\alpha|$ oraz w wysokości t, każda następna para: wielomianowym w $|\alpha|$ oraz w wysokości t, lub*
- *gdy φ jest z FOXPath—pierwsza para: $O(|\alpha|^3|t|)$, każda następna para: $O(|\alpha|^3)$.*

5 Ważne podproblemy pojawiające się przy ewaluacji zapytań XPath

Podczas opracowywania algorytmu ewaluującego zapytania XPath kluczowe okazało się rozwiązanie dwóch podproblemów, opisanych poniżej. Redukcja problemu obliczania zapytań XPath (Twierdzenie 1) do tych problemów jest techniczne lecz nie bardzo trudne i jest opisane w Rozdziale 5 rozprawy. W obu podproblemach na wejściu dany jest pewien zbiór skończony Q (w domyśle: zbiór stanów pewnego automatu skończonego). Oznaczmy zbiór relacji binarnych nad Q przez R_Q (tzn. elementy R_Q to podzbiory $Q \times Q$). Dane jest także drzewo binarne t, którego krawędzie etykietowane są elementami R_Q. Dla danych dwóch wierzchołków x, y drzewa t takich, że x jest przodkiem y, przez $val_t(x, y)$ oznaczmy złożenie wszystkich relacji będących etykietami krawędzi na ścieżce z x do y (w kolejności, w której występują na tej ścieżce). Relację tą będziemy nazywać wartością ścieżki z x do y.

Pierwszy podproblem będziemy nazywać problemem szybkiej ewaluacji ścieżek. Początkowo możemy wykonać pewne (dowolne) obliczenia w czasie liniowym ze względu na rozmiar t, lub ewentualnie w czasie $|t| \log |t|$ (zakładając, że rozmiar Q jest stały). Następnie mamy szybko odpowiadać na zapytania następującej postaci. Dostajemy dwa wierzchołki x i y w t takie, że x jest przodkiem y, i mamy obliczyć $val_t(x, y)$. Odpowiedź powinna zostać udzielona w czasie stałym—niezależnym od rozmiaru drzewa t czy od długości ścieżki z x do y, lub najwyżej w czasie $O(\log |t|)$.

Zobaczmy przykładowe zastosowanie takiego problemu. Załóżmy, że dany jest automat skończony $\mathcal{A}$ na słowach, o stanach Q, oraz drzewo t z etykietami na krawędziach. Początkowo możemy wykonać pewne (dowolne) obliczenia w czasie liniowym ze względu na rozmiar t. Następnie mamy szybko odpowiadać na zapytania następującej postaci. Dostajemy dwa wierzchołki x i y w drzewie i mamy szybko odpowiedzieć, czy ścieżka z x do y (tzn. słowo powstałe przez odczytanie etykiet na tej ścieżce) jest akceptowana przez $\mathcal{A}$. Aby to rozwiązać, możemy zamienić każdą etykietę a w drzewie przez relację przejścia stanów automatu $\mathcal{A}$ przy wczytywaniu tej litery. Wówczas, gdy x jest przodkiem y, $val_t(x, y)$ to relacja przejścia automatu przy odczytywaniu etykiet na ścieżce z x do y. Na jej podstawie możemy natychmiast stwierdzić, czy $\mathcal{A}$ akceptuje taką ścieżkę. Natomiast $val_t(x, y)$ możemy obliczyć korzystając z problemu szybkiej ewaluacji ścieżek. Jeśli x miałby być potomkiem y, możemy robić tak samo, lecz w drzewie musimy zapisać odwrotności powyższych relacji; wówczas ich składanie odpowiada ruchowi automatu od potomka do przodka. W ogólności, musimy znaleźć najbliższego wspólnego przodka z wierzchołków x i y (okazuje się, że można to zrobić w czasie stałym, po odpowiednim przygotowaniu drzewa), w jednym drzewie odczytać relację przejścia na ścieżce z x do z, w drugim relację przejścia na ścieżce z z do y, a następnie złożyć.

Drugi ważny podproblem będziemy nazywać problemem dystrybucji tzw. nawiasów. Podobnie jak poprzednio mamy drzewo t etykietowane relacjami nad Q. *Nawiasem* będziemy nazywać czwórkę (x, y, Q_x, Q_y), gdzie x jest przodkiem y w drzewie t, natomiast Q_x i Q_y są podzbiorami Q. Dla zbioru $P \subseteq Q$ oraz relacji r nad Q będziemy oznaczać

$$P \circ r = \{q \,|\, (p, q) \in r,\, p \in P\} \qquad \text{oraz} \qquad r \circ P = \{q \,|\, (q, p) \in r,\, p \in P\}.$$

Powiemy, że nawias *wybiera* parę (p, q) w wierzchołku z, jeśli z jest między x i y (być może jest równy jednemu z nich) oraz $p \in Q_x \circ val_t(x, z)$ i $q \in val_t(z, y) \circ Q_y$. W problemie dystrybucji nawiasów dostajemy zbiór nawiasów S; naszym zadaniem jest wyznaczyć zbiór par wybieranych w każdym wierzchołku. Problem ten jest podobny do pewnego rodzaju zapytań XPath. Gdybyśmy każdy nawias przetwarzali osobno, musielibyśmy dodać jakieś pary we wszystkich wierzchołkach między x i y, więc czas działania musiałby być co najmniej $O(|t| \cdot |S|)$ (co zresztą łatwo uzyskać). My jednak chcemy rozwiązać ten problem w czasie $O(|t| + |S|)$ (lub co najwyżej $O(|t| \log |t| + |S|)$).

Dla uproszczenia w poniższym tekście będziemy zakładać, że w drzewie t każdy wierzchołek ma co najwyżej jednego syna. Takie drzewo jest w istocie słowem, oznaczmy je $w = a_1 a_2 \ldots a_n$; wierzchołki odpowiadają pozycjom między literami tego słowa.

6 Rozwiązanie liniowo-logarytmiczne

Przyjrzyjmy się najpierw problemowi szybkiej ewaluacji ścieżek (w przypadku gdy drzewo jest słowem). Dosyć łatwo możemy osiągnąć czas działania etapu przygotowującego $O(|Q|^3 \cdot n)$ oraz zapytania $O(|Q|^3 \log n)$. Ustalmy zbiór Q oraz słowo (drzewo) wejściowe $w = a_1 a_2 \dots a_n$. Zauważmy, że jeśli znane są wartości dla dwóch podsłów u i v, to w czasie $O(|Q|^3)$ można obliczyć wartość dla ich konkatenacji uv: jest to po prostu złożenie wartości tych podsłów (tzn. $val_t(x,z) = val_t(x,y) \circ val_t(y,z)$). Będziemy postępować zgodnie ze strategią „dziel i zwyciężaj", w następujący sposób. Dla każdej potęgi dwójki k oraz dla każdego i (z odpowiedniego zakresu) obliczamy wartość dla podsłowa $a_{ik} a_{ik+1} \dots a_{(i+1)k-1}$. Dla podsłów składających się z pojedynczych liter wartość to po prostu relacja zapisana w jedynej literze. Dla dłuższych słów liczymy ją na podstawie wyników dla słów o połowę krótszych, zatem całość zajmuje czas $O(|Q|^3 n)$. (Na marginesie dodajmy, że gdy t jest dowolnym drzewem binarnym, konstrukcja analogicznej struktury danych zajmuje nieco większy czas $O(|Q|^3 |t| \log |t|)$).

Rozważmy teraz etap odpowiedzi na zapytanie o wartość podsłowa: rozważmy dwa indeksy i, j. Podsłowo $a_i a_{i+1} \dots a_j$ możemy podzielić na $O(\log n)$ podsłów będących w naszej strukturze, dla których wartość jest obliczona i zapamiętana. Wystarczy więc złożyć te zapamiętane wartości, aby uzyskać wartość całego podsłowa. Zajmuje to czas $O(|Q|^3 \log n)$.

Przejdźmy teraz do problemu dystrybucji nawiasów. Ważne tutaj będzie, że każdy nawias można podzielić na dwa krótsze. Konkretnie, dla dowolnego nawiasu (x, y, Q_x, Q_y) możemy równoważnie rozpatrywać dwa nawiasy

$$(x, z, Q_x, val_t(z,y) \circ Q_y) \qquad \text{oraz} \qquad (z, y, Q_x \circ val_t(x,z), Q_y)$$

dla dowolnego wierzchołka z między x a y. Chodzi o to, iż dowolna para (p, q) będzie wybrana przez oryginalny nawias w pewnym wierzchołku z, wtedy i tylko wtedy, gdy będzie wybrana przez jeden z nowych nawiasów. Zgodnie z tą zasadą, postępując podobnie jak powyżej, dowolny nawias można podzielić na maksymalnie $2 \cdot \log n$ nawiasów odpowiadających podsłowom dla których wcześniej stablicowaliśmy wartości (czyli podsłowom $a_{ik} a_{ik+1} \dots a_{(i+1)k-1}$ w których k jest potęgą dwójki). Każdy nawias z S można w ten sposób rozbić w czasie $O(|Q|^2 \log n)$.

Ponadto każdy nawias można rozbić na $|Q_x| \cdot |Q_y|$ nawiasów między tymi samymi wierzchołkami, w których zbiory są jednoelementowe. Konkretnie, bierzemy nawiasy $(x, y, \{p\}, \{q\})$ dla wszystkich $p \in Q_x$, $q \in Q_y$. Postępujemy w ten sposób dla wszystkich posiadanych obecnie nawiasów. Następnie, rozpoczynając od najdłuższych nawiasów, rozdzielamy każdy nawias na dwa o połowę krótsze, a te z kolei na nawiasy, w których zbiory są jednoelementowe (czyli w sumie z jednego nawiasu dostajemy nie więcej niż $2|Q|$ nawiasów). Wszystkie nawiasy, które dostajemy w krokach pośrednich, odpowiadają podsłowom $a_{ik} a_{ik+1} \dots a_{(i+1)k-1}$, w których k jest potęgą dwójki, oraz mają jednoelementowe zbiory. Takich podsłów jest tylko $2n$, zatem nawiasów przetworzymy w sumie co najwyżej $2|Q|^2 n$, gdyż dla każdego podsłowa jest co najwyżej $|Q|^2$ różnych nawiasów mających jednoelementowe zbiory. Ważne jest, że każdy nawias przetwarzamy tylko raz (tzn. eliminujemy nawiasy, które otrzymaliśmy wielokrotnie). Jeden nawias możemy przetworzyć w czasie $O(|Q|)$, zatem czas po czasie $O(|Q|^3 n)$ dostaniemy tylko nawiasy długości 1. Z takich nawiasów możemy łatwo odzyskać wybierane pary, gdyż mogą być one wybrane tylko w jednym bądź drugim końcu nawiasu. Podsumowując, czas działania tego algorytmu to $O(|Q|^3(|t| + |S| \log |t|))$.

Uogólnienie powyższego rozwiązania tych dwóch problemów na drzewa daje dowód pierwszego wariantu Twierdzenia 1.

7 Rozwiązanie w czasie liniowym, drzewa rozkładu

Wróćmy do problemu szybkiej ewaluacji ścieżek, w przypadku gdy drzewo jest słowem. Poprzednio ewaluowaliśmy wartość ścieżki (podsłowa) w czasie logarytmicznym. Okazuje się jednak, iż istnieje struktura danych, przy której wartość podsłowa można uzyskać w czasie niezależnym od n. Struktura ta bazuje na jednorodnych drzewach rozkładu. *Drzewo rozkładu* to drzewo, którego wierzchołki odpowiadają podsłowom, dzieci wierzchołka wyznaczają podział jego podsłowa

na krótsze podsłowa, a w liściach mamy podsłowa jednoliterowe. Rozwiązanie przedstawione powyżej de facto używa takiego drzewa rozkładu. Każdy wierzchołek tego drzewa ma dwóch synów, a całe drzewo ma głębokość $O(\log n)$.

W celu stworzenia algorytmu o czasie działania niezależnym od n, chcemy mieć do dyspozycji drzewo rozkładu o stałej (niezależnej od n) głębokości. Aby to było możliwe, musimy dozwolić na istnienie w nim wierzchołków mających dowolnie wiele dzieci. Niezbędny jest jednak warunek jednorodności. Rozważmy pewien wierzchołek (drzewa rozkładu) x oraz jego dzieci $x_1, x_2, \ldots, x_k$. Odpowiadają one pewnym podsłowom $w_1, w_2, \ldots, w_k$ naszego słowa. Powiemy, że wierzchołek x jest *jednorodny*, jeśli dla każdych indeksów $i \leq j$ wartość podsłowa $w_i w_{i+1} \ldots w_j$ jest taka sama. Powiemy, że drzewo rozkładu jest *jednorodne*, jeśli każdy jego wierzchołek mający przynajmniej trzech synów jest jednorodny. Zauważmy, że nie wymagamy nic w przypadku wierzchołków mających dwóch dzieci, zatem binarne drzewo rozkładu zawsze jest jednorodne (przeciwnie: gdybyśmy wymagali jednorodności od wszystkich wierzchołków, to wartość każdego podsłowa musiałaby być taka sama, więc większość słów nie miałaby żadnego drzewa rozkładu).

Załóżmy, że mamy (dla pewnego słowa $w = a_1 a_2 \ldots a_n$) jednorodne drzewo rozkładu pewnej głębokości h. Zauważmy, że ma ono co najwyżej $2n$ wierzchołków. Zatem w czasie liniowym (konkretnie $O(|Q|^3 n)$) możemy w każdym wierzchołku obliczyć wartość podsłowa odpowiadającego temu wierzchołkowi, idąc po prostu od liści do korzenia. Co ważniejsze, możemy następnie w czasie $O(|Q|^3 h)$ obliczyć wartość dowolnego podsłowa $a_i a_{i+1} \ldots a_j$. Jest tak dlatego, że każde podsłowo możemy podzielić na co najwyżej $2h$ podsłów, z których każde odpowiada pewnemu ciągowi braci z drzewa (ale nie każde odpowiada pojedynczemu wierzchołkowi drzewa—to byłoby niemożliwe). Natomiast dla każdego ciągu braci, zgodnie z definicją jednorodności, wartość jest taka sama jak dla jednego z nich, czyli jest zapamiętana w drzewie.

Również problem dystrybucji nawiasów można rozwiązać korzystając z jednorodnego drzewa rozkładu o głębokości h w czasie $O(|Q|^3(|t|+h\cdot|S|))$. Pierwszy krok jest taki sam jak w rozwiązaniu liniowo-logarytmicznym: każdy nawias zamieniamy na co najwyżej $2h$ nawiasów, z których każdy odpowiada pewnemu ciągowi braci z drzewa rozkładu (ściślej: podsłowo między początkiem a końcem nawiasu odpowiada pewnemu ciągowi braci z drzewa rozkładu). Każdy nawias z S można w ten sposób rozbić w czasie $O(|Q|^2 \cdot h)$. Następnie każdy z otrzymanych nawiasów zastępujemy przez nawiasy, w których zbiory są jednoelementowe.

Teraz pojawia się nowa trudność, której nie było w rozwiązaniu liniowo-logarytmicznym: musimy wyeliminować także nawiasy odpowiadające ciągom braci drzewa rozkładu (a nie tylko pojedynczym wierzchołkom drzewa rozkładu). Otóż dzięki jednorodności okazuje się, że jeśli mamy dwa nawiasy (x_1, y_1, Q_x, Q_y) oraz (x_2, y_2, Q_x, Q_y) (mające te samych zbiory) odpowiadające ciągom synów tego samego wierzchołka drzewa rozkładu, oraz x_2 i y_2 są pomiędzy x_1 i y_1, to drugi, krótszy, nawias można usunąć (dodając jeszcze pewne nawiasy odpowiadające pojedynczym wierzchołkom drzewa rozkładu, blisko początku i końca drugiego nawiasu); (prawie) wszystkie pary wybierane przez krótszy nawias są także wybierane przez dłuższy nawias. Szczegóły tutaj pomijamy; obserwacja ta jest opisana przez Lemat 4.8 w rozprawie. Ważne jest jednak, że dzięki tej obserwacji, dla każdej pary zbiorów (jednoelementowych) Q_x, Q_y, oraz każdej pozycji y w słowie, wystarczy trzymać jeden, najdłuższy nawias o takich zbiorach i kończący się na tej pozycji. Powoduje to, że liczba nawiasów, które będziemy mieli od tej pory, jest ograniczona przez liczbę wierzchołków drzewa rozkładu, razy $|Q|^2$. Zatem możemy postępować jak poprzednio: zaczynając od najdłuższych nawiasów, każdy nawias zastępujemy przez najwyżej $2|Q|$ nawiasów o połowę krótszych, mających jednoelementowe zbiory (obojętnie czy nasz nawias odpowiada pojedynczemu wierzchołkowi drzewa rozkładu, czy ciągowi braci tego drzewa). W każdym momencie eliminujemy zbędne nawiasy, zgodnie z powyższą regułą, dzięki czemu przetworzymy co najwyżej $O(|Q|^2|t|)$ nawiasów. Zatem czas działania tego algorytmu jest taki jak zadeklarowaliśmy: $O(|Q|^3(|t| + h \cdot |S|))$.

Pozostaje pytanie, czy dla każdego słowa i automatu istnieje jednorodne drzewo rozkładu o stałej głębokości (tj. niezależnej od długości słowa). Okazuje się, że tak; mówi o tym twierdzenie udowodnione przez I. Simona [Sim90]. T. Colcombet [Col07b] podał algorytm obliczający to drzewo w czasie liniowym ze względu na n. Należy jednak zwrócić uwagę, iż zarówno wysokość takiego drzewa, jak i czas potrzebny do jego obliczenia, jest wykładniczy ze względu na $|Q|$. (Konkretnie, cytowane twierdzenia mówią o ogólniejszej sytuacji, gdy zamiast zbioru relacji R_Q

mamy dowolny monoid. Wysokość drzewa jest liniowa ze względu na rozmiar monoidu, lecz niestety liczba relacji binarnych nad Q jest wykładnicza.)

Powyższe rozważania można uogólnić do sytuacji, gdy mamy do czynienia z dowolnym drzewem zamiast ze słowem. Nie jest to jednak natychmiastowe. Jednorodne drzewa rozkładu należy konstruować wzdłuż każdej ścieżki drzewa od korzenia do liści. Aby jednak rozwiązanie miało złożoność liniową względem rozmiaru drzewa (a nie względem sumy długości wszystkich ścieżek od korzenia do liści), część drzew rozkładu nad wspólną częścią dwóch ścieżek powinna być taka sama (w odpowiednio rozumianym sensie). Przy oryginalnej definicji jednorodnego drzewa rozkładu tak jednak nie musi być: można przykład słów różniących się tylko ostatnią literą, dla których jednorodne drzewa rozkładu muszą różnić się już przy początku słowa. Trudność ta nie występuje, jeśli zastosujemy osłabioną wersję definicji jednorodnego drzewa rozkładu, zaproponowaną przez T. Colcombeta [Col07a] jego drzewa rozkładu można generować w sposób deterministyczny: część drzewa dla prefiksu słowa zależy tylko od tego prefiksu. Rozwiązanie takie zostało użyte w pracy [BP08] i daje drugą część Twierdzenia 1. W przedstawianej rozprawie drugą część Twierdzenia 1 dowodzimy nieco innymi metodami, korzystającymi z automatów deterministycznych.

8 Szybka konstrukcja drzew rozkładu

Wiadomo, że dla pewnych słów jednorodne drzewo rozkładu musi mieć wysokość przynajmniej taką jak rozmiar monoidu (czyli w naszym przypadku $2^{|Q|^2}$). Czy to jednak stoi na przeszkodzie temu, aby je szybciej obliczać? Rozmiar drzewa rozkładu nie jest fundamentalną przeszkodą, gdyż drzewo rozkładu ma zawsze co najwyżej $2n$ wierzchołków (gdzie n jest długością słowa), niezależnie od tego, jaką ma wysokość. W przedstawianej rozprawie udowodnione zostało, że w przypadku monoidu relacji binarnych nad zbiorem Q jednorodne drzewo rozkładu można obliczyć w czasie $O(|Q|^3 n)$; zatem zależność od $|Q|$ jest wielomianowa, a nie wykładnicza. Konstrukcja opiera się na wcześniejszym dowodzie (algorytmie) z pracy T. Colcombeta. Analizujemy jednak dokładnie jakie operacje na elementach monoidu muszą być wykonywane i pokazujemy, że można je wykonywać w czasie wielomianowym od $|Q|$.

Pozostaje pytanie, czy jeśli mamy jednorodne drzewo rozkładu wysokości wykładniczej od $|Q|$, to czy możemy ewaluować wartości podsłów w czasie wielomianowym od $|Q|$. Okazuje się, że tak. Wystarczy do drzewa rozkładu dodać tak zwane wskaźniki przyspieszające. Dla dowolnego wierzchołka x w drzewie rozkładu i dowolnego k będącego potęgą dwójki, pamiętamy wskaźnik do wierzchołka y (przodka x), który leży o k poziomów wyżej w drzewie rozkładu niż x. Dodatkowo, jeśli wierzchołkowi x odpowiada podsłowo w_x, a wierzchołkowi y podsłowo w_y, wraz z tym wskaźnikiem pamiętać będziemy wartość prefiksu w_y, który jest przed w_x, oraz wartość sufiksu w_y, który jest po w_x. Zauważmy, że wskaźników tych jest tyle co wierzchołków drzewa rozkładu (czyli $\leq 2n$) razy logarytm z wysokości drzewa (który jest rzędu $|Q|^2$). Zatem możemy w czasie $O(|Q|^5 n)$ znaleźć wszystkie te wskaźniki i wartości odpowiadających im podsłów, składając po prostu dłuższe wskaźniki z dwóch o połowę krótszych.

Gdy już mamy powyższą strukturę wskaźnikową, to dowolne podsłowo $a_i a_{i+1} \dots a_j$ możemy podzielić na co najwyżej $O(\log h) = O(|Q|^2)$ (gdzie h to wysokość drzewa) podsłów odpowiadających bądź to ciągowi braci, bądź wskaźnikom z naszej struktury. Po prostu używamy naszych wskaźników przyspieszających, aby szybko dotrzeć do najniższego wspólnego przodka wierzchołków odpowiadających pierwszej i ostatniej literze a_i i a_j. Następnie, jak poprzednio, składamy zapamiętane wartości dla tych podsłów. Okazuje się, że również problem dystrybucji nawiasów można rozwiązać w czasie $O(|Q|^5 n)$, korzystając z właśnie opisanej struktury wskaźników przyspieszających.

Niestety szybki algorytm konstrukcji jednorodnych drzew rozkładu działa tylko dla słów—nie da się go uogólnić do przypadku dowolnych drzew. Dlatego algorytm wyliczania zapytań XPath, o którym mowa w trzeciej części Twierdzenia 1, ma złożoność zależną od wysokości drzewa. Po prostu kodujemy dowolne drzewo w słowie, zapisując w etykietach strukturę drzewa. Kodowanie takie może być odczytane za pomocą Regular XPath w następującym sensie: dla dowolnego zapytania φ można wyznaczyć zapytanie φ_{enc} takie, że zbiór wierzchołków wybieranych przez

φ w oryginalnym drzewie t można odtworzyć ze zbioru wierzchołków wybieranych przez φ_{enc} w kodowaniu drzewa t w słowie. Rozmiar nowego zapytania zależy jednak (liniowo) od głębokości drzewa.

9 Algorytm dla zapytań z fragmentu FOXPath

Przejdźmy teraz do omówienia czwartej części Twierdzenia 1. W części tej mamy na wejściu zapytanie z fragmentu XPath, czyli wyrażenia ścieżkowe znajdujące się w nim nie używają gwiazdki Kleene'a. Powoduje to, iż etykietowanie drzewa, które mamy na wejściu do problemów szybkiej ewaluacji ścieżek oraz dystrybucji ścieżek, nie jest dowolne, lecz jest pewnej szczególnej postaci. Dla uproszczenia przedstawimy ograniczenie na etykietowanie drzewa tylko w przypadku gdy każdy wierzchołek drzewa ma co najwyżej jednego syna (czyli drzewo jest słowem). Zakładamy, że na elementach Q istnieje porządek liniowy $\leq$ taki, że relacje będące etykietami drzewa zawierają tylko pary (p,q) dla $p \leq q$. Ponadto, każda para (q,q) występuje albo w etykietach wszystkich krawędzi, albo nie występuje w ogóle.

Można na to założenie spojrzeć z innej strony. Rozważmy wyrażenie regularne r, w którym gwiazdka Kleene'a jest nieużywana, może wystąpić jedynie A^*, gdzie A jest alfabetem wejściowym. Przetłumaczmy to wyrażenie na automat $\mathcal{A}$. Wówczas istnieje porządek liniowy $\leq$ na stanach tego automatu taki, iż przejścia ze stanu p do stanu q będą istniały tylko dla $p \leq q$. Ponadto jeśli jest przejście z q do q (dla pewnego q), to odpowiada ono podwyrażeniu A^*, więc przejście z q do q jest możliwe po każdej literze. Przypuśćmy, że nasze drzewo powstało z dowolnego drzewa, poprzez zastąpienie etykiet przez relacje przejścia automatu $\mathcal{A}$ przy wczytywaniu tych etykiet (jak w Rozdziale 5). Dostajemy w ten sposób dokładnie drzewa spełniające powyższy warunek.

W jaki sposób możemy szybko obliczyć wartość dowolnego podsłowa słowa $w = a_1 a_2 \dots a_n$? Najpierw, dla każdego i oraz elementów $p, q \in Q$ obliczmy i zapamiętajmy pierwsze j takie, że $(p,q) \in a_i \circ a_{i+1} \circ \dots \circ a_j$ (lub informację, iż takie j nie istnieje). Można to zrobić idąc od prawej do lewej: licząc wynik dla i korzystamy z wyniku dla $i+1$. Po prostu: jeśli $(p,q) \in a_i$ to bierzemy $j = i$, w przeciwnym wypadku bierzemy najmniejsze j takie, że $(r,q) \in a_{i+1} \circ a_{i+2} \circ \dots \circ a_j$, uwzględniając wszystkie $r \in Q$ takie, że $(p,r) \in a_i$. Oznaczmy ponadto przez Q_{loop} zbiór tych elementów $q \in Q$, że para (q,q) występuje we wszystkich relacjach a_i (zgodnie z naszym założeniem o etykietowaniu każda para (q,q) albo występuje we wszystkich relacjach albo w żadnej).

Niech teraz dane będzie pewne podsłowo $a_k a_{k+1} \dots a_l$ (tzn. dostajemy tylko indeksy k i l). Przypuśćmy, że chcemy sprawdzić, czy pewna para (p,q) należy do wartości tego podsłowa (tj. do złożenia relacji $a_k, a_{k+1}, \dots, a_l$). Świadczy o tym ciąg elementów $p = q_k, q_{k+1}, q_{k+2}, \dots, q_{l+1} = q$ takich, że $(q_i, q_{i+1}) \in a_i$ dla każdego i ($k \leq i \leq l$). Kluczowa obserwacja jest taka, iż pośród $|Q|+1$ ostatnich elementów tego ciągu musi wystąpić powtórzenie, tzn. musi być $q_i = q_{i+1} \in Q_{loop}$; wynika to z naszego założenia o etykietowaniu. Niech j będzie najmniejsze takie, że $(p,q_i) \in a_k \circ a_{k+1} \circ \dots \circ a_j$. Oczywiście $j \leq i$, bo i spełnia ten warunek. Możemy założyć, że $q_j = q_{j+1} = \dots = q_i$, gdyż para (q_i, q_i) występuje w każdej relacji w naszym słowie.

Wystarczy zatem, korzystając z naszej struktury, dla każdego elementu $r \in Q_{loop}$ znaleźć najmniejsze j takie, że $(p,r) \in a_k \circ a_{k+1} \circ \dots \circ a_j$. Interesuje nas tylko sytuacja, gdy $j \leq l$. Jeśli $j \leq l - |Q|$, to wiemy, że $(p,r) \in a_k \circ a_{k+1} \circ \dots \circ a_{l-|Q|}$. Następnie ostatnie $|Q|$ relacji składamy jedna po drugiej. Postępując w ten sposób dostajemy wartość podsłowa $a_k a_{k+1} \dots a_l$ w czasie $O(|Q|^4)$.

Podobna obserwacja pozwala na szybką dystrybucję nawiasów. Otóż okazuje się, iż dowolny nawias można zamienić na $|Q|^2$ nawiasów $(x, y, \{q_x\}, \{q_y\})$, w których $q_y \in Q_{loop}$, oraz na $|Q|$ nawiasów długości 1. Załóżmy teraz, że mamy dwa nawiasy $(x, y_1, \{q_x\}, \{q_y\})$ oraz $(x, y_2, \{q_x\}, \{q_y\})$, przy czym $q_y \in Q_{loop}$. Wówczas krótszy z tych nawiasów można usunąć, gdyż każda para wybierana przez krótszy nawias jest wybierana także przez dłuższy nawias (gdyż para (q_y, q_y) jest na pewno w wartości podsłowa między y_1 a y_2). Dzięki tej obserwacji, dla każdej pary zbiorów (jednoelementowych) $\{q_x\}$, $\{q_y\}$, przy czym $q_y \in Q_{loop}$, oraz każdej pozycji x w słowie, wystarczy trzymać jeden, najdłuższy nawias o takich zbiorach i zaczynający się na tej pozycji. Powoduje to, że liczba nawiasów, które będziemy mieli od tej pory, jest ograniczona przez $2|Q|^2 n$ (gdyż liczba

nawiasów długości 1 mających zbiory jednoelementowe też jest ograniczona przez $|Q|^2 n$). Zatem możemy postępować jak poprzednio: zaczynając od najdłuższych nawiasów, każdy nawias zastępujemy przez najwyżej $2|Q|$ krótszych nawiasów mających jednoelementowe zbiory. Odcinamy przy tym jedną, pierwszą literę; powstaje więc $\leq |Q|$ nowych nawiasów długości 1 oraz $\leq |Q|$ nowych nawiasów, w których prawy zbiór jest taki jak poprzednio, więc zawiera się w Q_{loop}. W każdym momencie eliminujemy zbędne nawiasy, zgodnie z powyższą regułą, dzięki czemu przetworzymy co najwyżej $O(|Q|^2 n)$ nawiasów. Zatem czas działania tego algorytmu jest $O(|Q|^3 n)$.

Uogólnienie tych algorytmów na drzewa prowadzi do czwartej wersji Twierdzenia 1, czyli algorytmu ewaluującego zapytania z fragmentu FOXPath w czasie $O(|\varphi|^3 |t|)$.

Literatura

[BK09] M. Benedikt and C. Koch. XPath leashed. *ACM Computing Surveys*, 41:3:1–3:54, January 2009.

[BP] M. Bojańczyk and P. Parys. XPath evaluation in linear time. *Journal of the ACM*. Accepted for publication.

[BP08] M. Bojańczyk and P. Parys. XPath evaluation in linear time. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '08, pages 241–250, New York, NY, USA, 2008. ACM.

[BP10] M. Bojańczyk and P. Parys. Efficient evaluation of nondeterministic automata using factorization forests. In *Proceedings of the 37th international colloquium conference on Automata, languages and programming*, ICALP'10, pages 515–526, Berlin, Heidelberg, 2010. Springer-Verlag.

[CD99] J. Clark and S. DeRose. XML path language (XPath) version 1.0, W3C recommendation. Technical report, W3C, 1999.

[Col07a] T. Colcombet. A combinatorial theorem for trees. In *Proceedings of the 34th international colloquium conference on Automata, languages and programming*, ICALP'07, pages 901–912, Berlin, Heidelberg, 2007. Springer-Verlag.

[Col07b] T. Colcombet. On factorization forests. Technical Report hal-00125047, Irisa Rennes, 2007.

[GKP02] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 95–106. VLDB Endowment, 2002.

[GKP03] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space eficiency. In *Proceedings of the 19th International Conference on Data Engineering*, ICDE'03, pages 379–390, 2003.

[GKP05] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems*, 30:444–491, June 2005.

[Nev02] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31:39–46, September 2002.

[Par09] Paweł Parys. XPath evaluation in linear time with polynomial combined complexity. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '09, pages 55–64, New York, NY, USA, 2009. ACM.

[Sim90] I. Simon. Factorization forests of finite height. *Theoretical Computer Science*, 72(1):65–94, 1990.