Implementacja wydajnych struktur danych do praktycznych operacji na słowach

(Implementation of efficient data structures for practical string operations)

Michał Górniak

Instytut Informatyki Uniwersytetu Wrocławskiego

11 września 2020

Opis problemu

Chcemy utrzymywać i modyfikować zbiór słów $\mathcal{S}$, słowa w tym słowniku będą mogły być bardzo długie, więc przy dodawaniu nowych słów, będziemy dostawać od programu etykietę – liczbę naturalną dzięki której będziemy mogli łatwo odwoływać się do wybranych słów. Oznaczmy przez $\mathcal{W}(l)$ słowo reprezentowane przez etykietę l.

Opis problemu

Chcemy utrzymywać i modyfikować zbiór słów $\mathcal{S}$, słowa w tym słowniku będą mogły być bardzo długie, więc przy dodawaniu nowych słów, będziemy dostawać od programu etykietę – liczbę naturalną dzięki której będziemy mogli łatwo odwoływać się do wybranych słów. Oznaczmy przez $\mathcal{W}(l)$ słowo reprezentowane przez etykietę l.

W każdym momencie możemy modyfikować słownik lub zadawać zapytania:

- $\texttt{make\_string}(w)$ – $\mathcal{S} := \mathcal{S} \cup \{w\}$,
- $\texttt{concat}(l_1, l_2)$ – $\mathcal{S} := \mathcal{S} \cup \{\mathcal{W}(l_1)\mathcal{W}(l_2)\}$,
- $\texttt{split}(l, p)$ – $\mathcal{S} := \mathcal{S} \cup \{\mathcal{W}(l)_{\ldots p}, \mathcal{W}(l)_{(p+1)\ldots}\}$,

Opis problemu

Chcemy utrzymywać i modyfikować zbiór słów $\mathcal{S}$, słowa w tym słowniku będą mogły być bardzo długie, więc przy dodawaniu nowych słów, będziemy dostawać od programu etykietę – liczbę naturalną dzięki której będziemy mogli łatwo odwoływać się do wybranych słów. Oznaczmy przez $\mathcal{W}(l)$ słowo reprezentowane przez etykietę l.

W każdym momencie możemy modyfikować słownik lub zadawać zapytania:

- `make_string`(w) – $\mathcal{S} := \mathcal{S} \cup \{w\}$,
- `concat`(l_1, l_2) – $\mathcal{S} := \mathcal{S} \cup \{\mathcal{W}(l_1)\mathcal{W}(l_2)\}$,
- `split`(l, p) – $\mathcal{S} := \mathcal{S} \cup \{\mathcal{W}(l)_{\ldots p}, \mathcal{W}(l)_{(p+1)\ldots}\}$,
- `equals`(l_1, l_2) – zwróć $\mathcal{W}(l_1) = \mathcal{W}(l_2)$,
- `smaller`(l_1, l_2) – zwróć $\mathcal{W}(l_1) <_{\text{lex}} \mathcal{W}(l_2)$,
- `lcp`(l_1, l_2) – zwróć najdłuższy wspólny prefiks $\mathcal{W}(l_1)$ i $\mathcal{W}(l_2)$.

Tabela złożoności (modyfikacje słownika)

Niech n oznacza długość słowa, które jest podane jako pierwszy argument funkcji (lub długość słowa pod etykietą z pierwszego argumentu). W przypadku funkcji o dwóch argumentach będących słowami lub etykietami, długość drugiego słowa oznaczmy przez m.

Tabela złożoności (modyfikacje słownika)

Niech n oznacza długość słowa, które jest podane jako pierwszy argument funkcji (lub długość słowa pod etykietą z pierwszego argumentu). W przypadku funkcji o dwóch argumentach będących słowami lub etykietami, długość drugiego słowa oznaczmy przez m.

Rozwiązanie	`make_string`	`concat`	`split`
Naiwne	deterministyczna $\mathcal{O}(n)$	deterministyczna $\mathcal{O}(n+m)$	deterministyczna $\mathcal{O}(n)$
Drzewce	oczekiwana $\mathcal{O}(n)$	oczekiwana $\mathcal{O}(\log(n+m))$	oczekiwana $\mathcal{O}(\log n)$
Parsingi	oczekiwana $\mathcal{O}(n)$	oczekiwana $\mathcal{O}(\log(n+m))$	oczekiwana $\mathcal{O}(\log n)$
Gawrychowski et al.	$\mathcal{O}(n)$ w.h.p.	$\mathcal{O}(\log(n+m))$ w.h.p.	$\mathcal{O}(\log n)$ w.h.p.
Mehlhorn et al.	deterministyczna $\mathcal{O}(n)$	deterministyczna $\mathcal{O}(\log^2(n+m)\log^\star(n+m))$	deterministyczna $\mathcal{O}(\log^2 n \log^\star n)$
Alstrup et al.	$\mathcal{O}(n)$ w.h.p.	$\mathcal{O}(\log(n+m)\log^\star(n+m))$ w.h.p.	$\mathcal{O}(\log n \log^\star n)$ w.h.p.

Tabela złożoności (zapytania do struktury)

Rozwiązanie	`equals`	`smaller`	`lcp`
Naiwne	deterministyczna $\mathcal{O}(\min(n, m))$	deterministyczna $\mathcal{O}(\min(n, m))$	deterministyczna $\mathcal{O}(\min(n, m))$
Drzewce	Monte Carlo $\mathcal{O}(1)$	oczekiwana Monte Carlo $\mathcal{O}(\log^2(n+m))$	oczekiwana Monte Carlo $\mathcal{O}(\log^2(n+m))$
Parsingi	deterministyczna $\mathcal{O}(1)$	oczekiwana $\mathcal{O}(\log(n+m))$	oczekiwana $\mathcal{O}(\log(n+m))$
Parsingi (wolne `lcp`)	deterministyczna $\mathcal{O}(1)$	oczekiwana $\mathcal{O}(\log^2(n+m))$	oczekiwana $\mathcal{O}(\log^2(n+m))$
Gawrychowski et al.	deterministyczna $\mathcal{O}(1)$	deterministyczna $\mathcal{O}(1)$	deterministyczna $\mathcal{O}(1)$
Mehlhorn et al.	deterministyczna $\mathcal{O}(1)$	—	—
Alstrup et al.	deterministyczna $\mathcal{O}(1)$	deterministyczna $\mathcal{O}(1)$	deterministyczna $\mathcal{O}(1)$

O implementacji

Struktura kodu

Kod został podzielony na foldery, dzięki czemu można łatwo nawigować po projekcie. Implementacja każdej klasy znajduje się w osobnym pliku. Aby wiernie odwzorować problem z zadania, zaimplementowane struktury danych dziedziczą z abstrakcyjnej klasy `solution.h`.

O implementacji

Struktura kodu

Kod został podzielony na foldery, dzięki czemu można łatwo nawigować po projekcie. Implementacja każdej klasy znajduje się w osobnym pliku. Aby wiernie odwzorować problem z zadania, zaimplementowane struktury danych dziedziczą z abstrakcyjnej klasy `solution.h`.

Implementacja `solution.h`

```
1  class solution {
2      public:
3          virtual ~solution() = 0;
4          virtual int make_string(std::vector<int> &word) = 0;
5          virtual int concat(int label1, int label2) = 0;
6          virtual std::pair<int, int> split(int label, int position) = 0;
7          virtual bool equals(int label1, int label2) = 0;
8          virtual bool smaller(int label1, int label2) = 0;
9          virtual int longest_common_prefix(int label1, int label2) = 0;
10 };
```

Drzewce

Pomysł

Każde drzewo zbalansowane reprezentuje jedno słowo ze słownika. Używamy drzew zbalansowanych, które można łączyć i dzielić w czasie logarytmicznym. W każdym wierzchołku trzymamy hasz całego poddrzewa. Równość dwóch słów sprawdzamy porównując hasze, a `lcp` i porównanie leksykograficzne poprzez wyszukiwanie binarne po wyniku wraz w każdorazowym rozcinaniem drzewa.

Drzewce

Pomysł

Każde drzewo zbalansowane reprezentuje jedno słowo ze słownika. Używamy drzew zbalansowanych, które można łączyć i dzielić w czasie logarytmicznym. W każdym wierzchołku trzymamy hasz całego poddrzewa. Równość dwóch słów sprawdzamy porównując hasze, a `lcp` i porównanie leksykograficzne poprzez wyszukiwanie binarne po wyniku wraz w każdorazowym rozcinaniem drzewa.

O drzewcach

Implementacja drzewców nie wymaga implementacji rotacji drzewa. Drzewo musi zachowywać porządek kopcowy patrząc na priorytety, a porządek BST patrząc na klucze. W ten sposób zawsze powstaje **dokładnie** jeden drzewiec.

Drzewce

Pomysł

Każde drzewo zbalansowane reprezentuje jedno słowo ze słownika. Używamy drzew zbalansowanych, które można łączyć i dzielić w czasie logarytmicznym. W każdym wierzchołku trzymamy hasz całego poddrzewa. Równość dwóch słów sprawdzamy porównując hasze, a `lcp` i porównanie leksykograficzne poprzez wyszukiwanie binarne po wyniku wraz w każdorazowym rozcinaniem drzewa.

O drzewcach

Implementacja drzewców nie wymaga implementacji rotacji drzewa. Drzewo musi zachowywać porządek kopcowy patrząc na priorytety, a porządek BST patrząc na klucze. W ten sposób zawsze powstaje **dokładnie** jeden drzewiec.

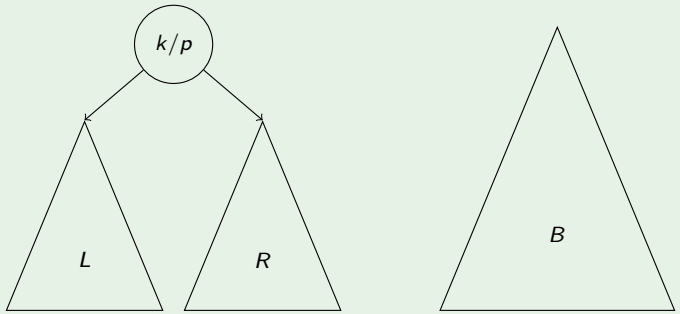
Przykład

W kolejnym slajdzie opiszemy działanie funkcji `merge`.

Operacja `merge` (teoria)

Przykład

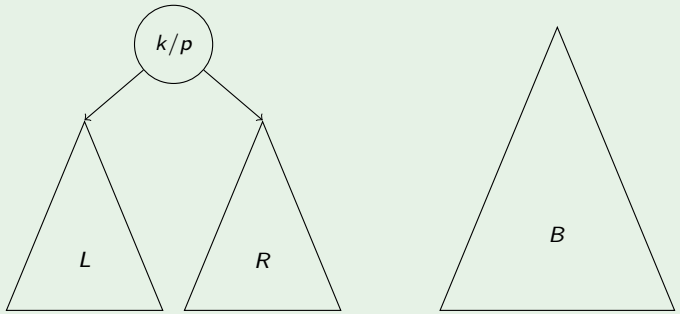
Załóżmy, że chcemy złączyć drzewce A oraz B i bez straty ogólności A ma większy priorytet w korzeniu niż B.



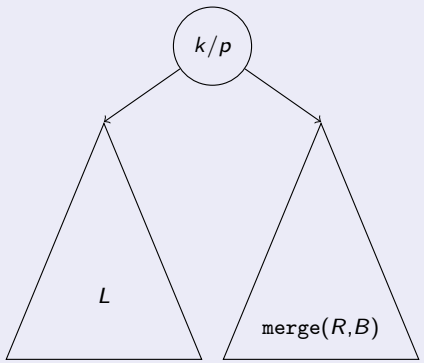
Operacja `merge` (teoria)

Przykład

Załóżmy, że chcemy złączyć drzewce A oraz B i bez straty ogólności A ma większy priorytet w korzeniu niż B.



Wystarczy zatem wywołać rekurencyjnie $\texttt{merge}(R, B)$ i podpiąć odpowiednio wskaźniki.



Operacja merge (implementacja)

```
1  balanced_trees::treap *balanced_trees::merge(balanced_trees::treap *t1,
2                                               balanced_trees::treap *t2) {
3      if (t1 == nullptr) {
4          return t2;
5      }
6      if (t2 == nullptr) {
7          return t1;
8      }
9      if (t1->priority > t2->priority) {
10         auto return_treap = new treap(*t1);
11         return_treap->right_subtree = merge(t1->right_subtree, t2);
12         update_values(return_treap);
13         return return_treap;
14     } else {
15         auto return_treap = new treap(*t2);
16         return_treap->left_subtree = merge(t1, t2->left_subtree);
17         update_values(return_treap);
18         return return_treap;
19     }
20 }
```

Drzewa symboli (*parse trees*)

Gramatyka bezkontekstowa

Tworzymy gramatykę bezkontekstową, ale bez pojedynczego symbolu startowego. Utrzymujemy natomiast, że każdy symbol generuje **dokładnie jedno słowo**. Każdy symbol ma przypisany poziom, na którym występuje. Symbole powstają podczas tworzenia *drzewa symbolu* nad nowym słowem.

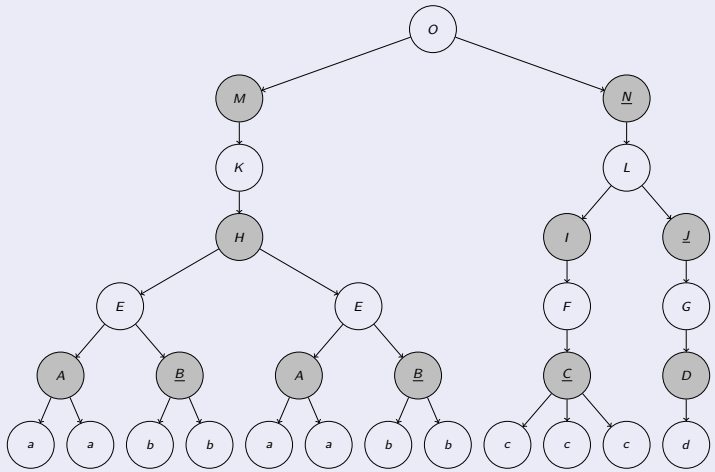
Drzewa symboli (*parse trees*)

Gramatyka bezkontekstowa

Tworzymy gramatykę bezkontekstową, ale bez pojedynczego symbolu startowego. Utrzymujemy natomiast, że każdy symbol generuje **dokładnie jedno słowo**. Każdy symbol ma przypisany poziom, na którym występuje. Symbole powstają podczas tworzenia *drzewa symbolu* nad nowym słowem.

Tworzenie drzewa symbolu

Polega na naprzemiennym wykonywaniu kompresji typu RLE (*run-length encoding*) oraz SHRINK, która zależy od losowych bitów symboli.



Twierdzenie o oczekiwanym czasie budowy drzewa symbolu

Twierdzenie

Oczekiwany czas budowy drzewa nad słowem długości n, wynosi $\mathcal{O}(n)$.

Twierdzenie o oczekiwanym czasie budowy drzewa symbolu

Twierdzenie

Oczekiwany czas budowy drzewa nad słowem długości n, wynosi $\mathcal{O}(n)$.

Fakt z pracy *Optimal Dynamic Strings*

Po zastosowaniu kompresji typu RLE, a następnie kompresji typu SHRINK na dowolnym słowie w, oczekiwana długość słowa w' powstałego w wyniku kompresji jest nie większa niż $\frac{3}{4}|w| + \frac{1}{4}$.

Twierdzenie o oczekiwanym czasie budowy drzewa symbolu

Twierdzenie

Oczekiwany czas budowy drzewa nad słowem długości n, wynosi $\mathcal{O}(n)$.

Fakt z pracy *Optimal Dynamic Strings*

Po zastosowaniu kompresji typu RLE, a następnie kompresji typu SHRINK na dowolnym słowie w, oczekiwana długość słowa w' powstałego w wyniku kompresji jest nie większa niż $\frac{3}{4}|w| + \frac{1}{4}$.

Szkic dowodu

- indukcja względem długości słowa,
- wykorzystanie powyższego faktu wraz z nierównością Markowa, aby oszacować prawdopodobieństwo, że długość słowa po dwóch kolejnych fazach kompresji skróci się przez stały czynnik.

Dekompozycja context-insensitive

Motywacja

Nie możemy w prosty sposób (podobnie jak w drzewach zbalansowanych) łączyć dwóch drzew symboli. Okazuje się jednak, że możemy zawsze znaleźć pewną *warstwę* drzewa symbolu, której RLE jest długości logarytmicznej i jest ona typu context-insensitive.

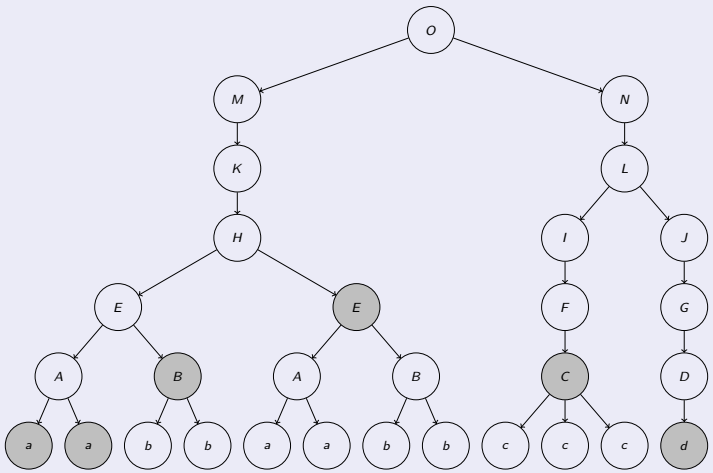
Dekompozycja context-insensitive

Motywacja

Nie możemy w prosty sposób (podobnie jak w drzewach zbalansowanych) łączyć dwóch drzew symboli. Okazuje się jednak, że możemy zawsze znaleźć pewną *warstwę* drzewa symbolu, której RLE jest długości logarytmicznej i jest ona typu context-insensitive.

Łączenie i dzielenie słów

Możemy wobec tego przy łączeniu znaleźć dekompozycje obu słów, a następnie je połączyć. Powstała lista jest dekompozycją konkatenacji początkowych dwóch słów. Następnie dobudowujemy górną część drzewa symbolu, podobnie jak przy budowaniu od liści. Dzielenie polega na wybraniu dekompozycji typu context-insensitive wybranego podsłowa i zbudowaniu nad nią drzewa symbolu.



Przykładowe uruchomienie

Przykład

Wywołanie `./run 129873 correctness 1`.

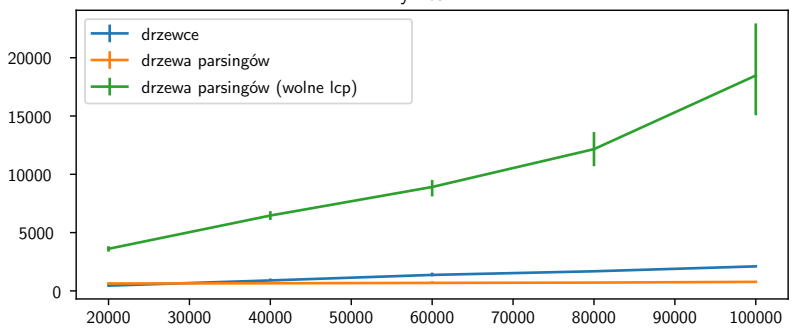
Wydruk z terminala

```
RUNNING RANDOM TESTS
------------------------------------------------------------
RUNNING BALANCED TREES SOLUTION
[**************************************************] 100%
PASSED 5147518 OUT OF 5147518 TESTS (100%)
RUNNING RANDOM TESTS
------------------------------------------------------------
RUNNING PARSE TREES LCP LOG^2 SOLUTION
[**************************************************] 100%
PASSED 5146257 OUT OF 5146257 TESTS (100%)
RUNNING RANDOM TESTS
------------------------------------------------------------
RUNNING PARSE TREES LCP LOG SOLUTION
[**************************************************] 100%
PASSED 5147611 OUT OF 5147611 TESTS (100%)
```

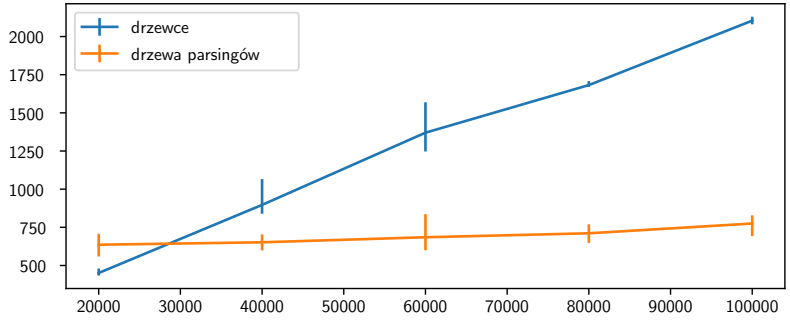
Wykresy czasów działania

Testy z tablicą sufiksową.

Wykres 4.1.



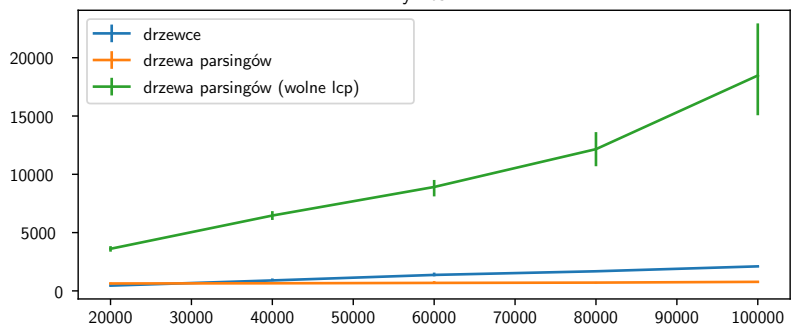
Wykres 4.2.



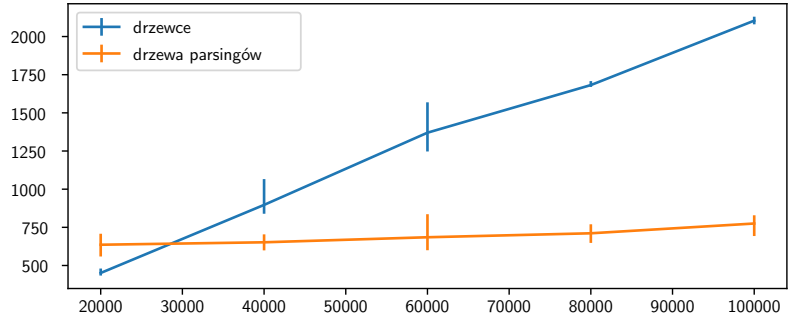
Wykresy czasów działania

Testy z tablicą sufiksową.

Wykres 4.1.

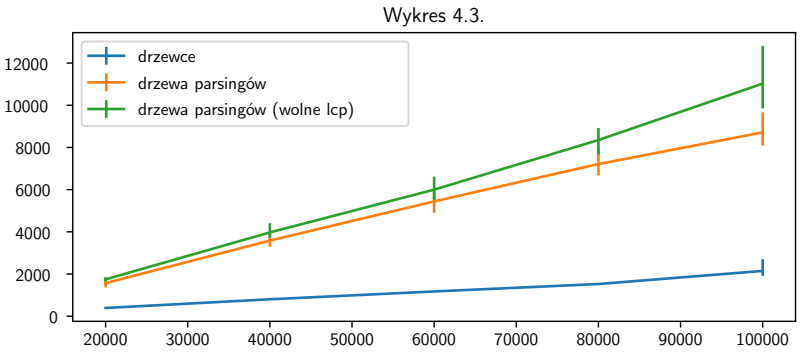


Wykres 4.2.

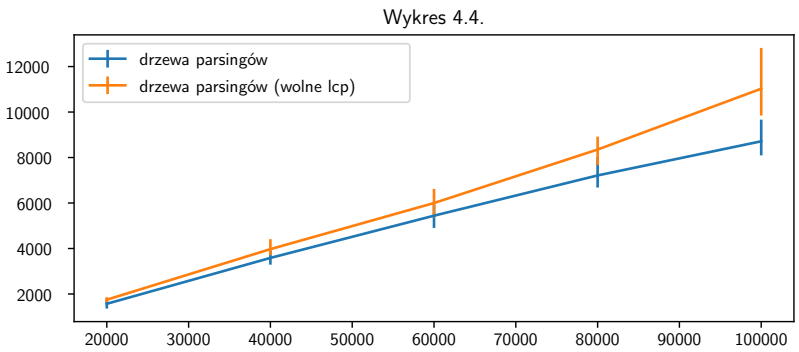


Testy z losowymi poprawnymi operacjami na słowniku.

Wykres 4.3.



Wykres 4.4.



Dziękuję za uwagę.