Tablice sufiksów i drzewa sufiksów

dr inż. Marcin Ciura

mgc@agh.edu.pl

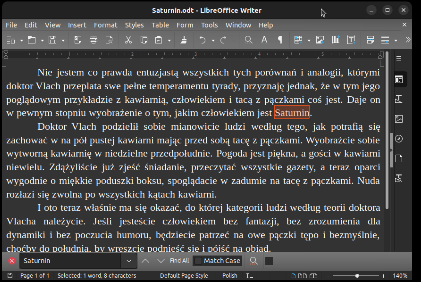
Wydział Informatyki, Akademia Górniczo-Hutnicza

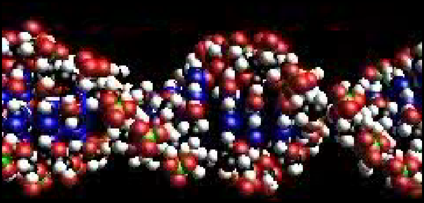
Plan wykładu

- Tablice sufiksów
- Zastosowania tablic sufiksów
- Drzewa sufiksów
- Zastosowania drzew sufiksów
- Algorytm Ukkonena budowania drzew sufiksów

Po co nam tablice sufiksów i drzewa sufiksów?

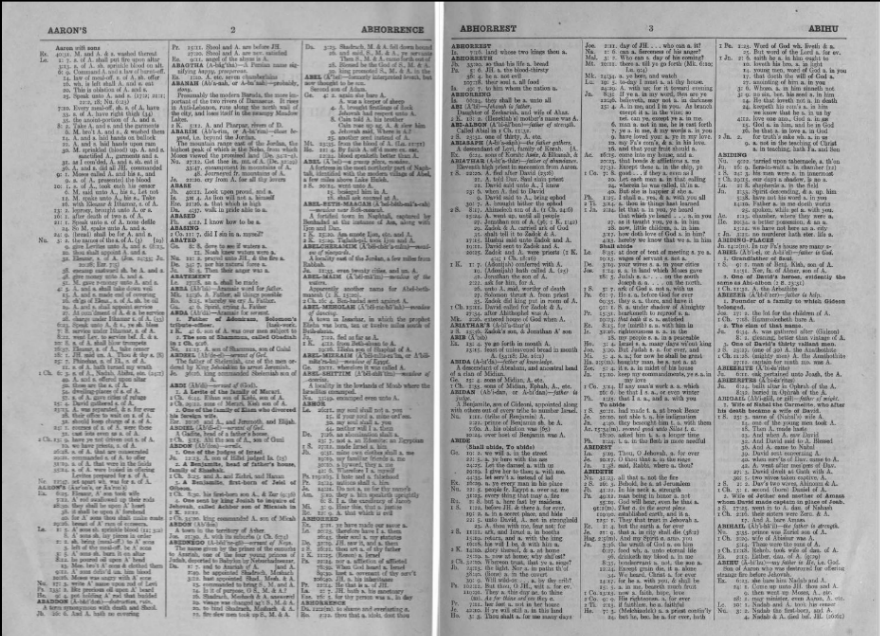
Żeby wyszukiwać wzorce w tekstach, i nie tylko po to :-)





Prehistoria

Konkordancje


AARON'S 2 ABHORRENCE

ABHORREST 3 ABIHU

Indeksy

Tablice sufiksów

Udi Manber

Izraelski informatyk. W 1990 roku wraz z Genem Myersem wprowadził pojęcie tablicy sufiksów. Był wiceprezesem w Amazonie, Google i YouTube.

Gene Myers (31.12.1953–)

Amerykański informatyk i bioinformatyk. Jest współautorem programu BLAST (basic local alignment search tool), który porównuje łańcuchy w biologicznych bazach danych. Artykuł z 1990 roku, w którym opisano BLAST, był do 2024 roku cytowany ponad 180 tysięcy razy.

Tablica sufiksów zawiera posortowane indeksy tych pozycji w danym łańcuchu *T*, od których zaczynają się sufiksy tego łańcucha. Tak sortuję elementy tablicy sufiksów:

Jeśli pewien sufiks S_1 poprzedza inny sufiks S_2 w porządku leksykograficznym, to indeks tej pozycji łańcucha *T*, od której zaczyna się sufiks S_1, znajduje się w tablicy sufiksów łańcucha *T* przed indeksem tej pozycji łańcucha *T*, od której zaczyna się sufiks S_2

Tablice sufiksów – przykład

Tablica sufiksów łańcucha ananas to `[0 2 4 1 3 5]`

od pozycji `0` łańcucha ananas zaczyna się jego sufiks ananas

od pozycji 2 łańcucha ananas zaczyna się jego sufiks anas

od pozycji 4 łańcucha ananas zaczyna się jego sufiks as

od pozycji 1 łańcucha ananas zaczyna się jego sufiks nanas

od pozycji 3 łańcucha ananas zaczyna się jego sufiks nas

od pozycji 5 łańcucha ananas zaczyna się jego sufiks s

Tablice sufiksów – najprostszy algorytm tworzenia

```go
type suffix struct {
	s []byte
	n int
}

func sortedSuffixes(text []byte) []suffix {
	suffixes := make([]suffix, len(text))
	for i := range suffixes {
		suffixes[i].s = text[i:]
		suffixes[i].n = i
	}
	sort.Slice(suffixes, func(i, j int) bool {
		return bytes.Compare(suffixes[i].s, suffixes[j].s) < 0
	})
	return suffixes
}
```

Tablice sufiksów – najprostszy algorytm tworzenia

```
// Index implementuje indeks łańcucha `text`. Ten indeks
// składa się z tablicy sufiksów łańcucha `text`
// i z łańcucha `text`
type Index struct {
   suffixes []int
   text     []byte
}
```

Tablice sufiksów – najprostszy algorytm tworzenia

```
// SuffixArray zwraca indeks łańcucha `text`
func SuffixArray(text []byte) Index {
    index := Index{
        make([]int, len(text)),
        make([]byte, len(text)),
    }
    index.text = text
    for i, suf := sortedSuffixes(text) {
        index.suffixes[i] = suf.n
    }
    return index
}
```

Tablice sufiksów – najprostszy algorytm tworzenia

Za pomocą funkcji `SuffixArray` mogę stworzyć tablicę sufiksów łańcucha `text` w czasie $O(|\texttt{text}|^2 \log|\texttt{text}|)$

Istnieją szybsze algorytmy tworzenia tablic sufiksów. Najszybsze z tych algorytmów działają w czasie $O(|\texttt{text}|)$

Tablice sufiksów – zajmowana pamięć

```go
// Index implementuje indeks łańcucha `text`. Ten indeks
// składa się z tablicy sufiksów łańcucha `text`
// i z łańcucha `text`
type Index struct {
    suffixes []int
    text     []byte
}
```

Wewnątrz struktury `Index`:

- każdy element tablicy `suffixes` zajmuje 1 słowo maszynowe
- każdy znak łańcucha `text` zajmuje 1 bajt

Zatem tablica sufiksów łańcucha `text` zajmuje `(sizeof(int)+1)*len(text)` bajtów

Zastosowania tablic sufiksów

Zastosowania tablic sufiksów – 1

Tak wyszukuję wszystkie wystąpienia danego wzorca P w danym tekście T:

- Buduję tablicę sufiksów `suffixes` łańcucha T
- Znajduję indeks `i` pierwszego takiego elementu tablicy `suffixes`, który nie poprzedza wzorca T w porządku leksykograficznym
- Znajduję indeks `j` pierwszego takiego elementu tablicy `suffixes`, którego prefiks o długości $|P|$ występuje w porządku leksykograficznym za wzorcem P
- Wycinek `suffixes[i:j]` zawiera indeksy tych pozycji, od których zaczynają się wystąpienia wzorca P w tekście T

Zastosowania tablic sufiksów – 1

```go
// Suffix zwraca ten sufiks łańcucha `x.text`, który
// zaczyna się od `i`-tej pozycji tego łańcucha
func (x Index) Suffix(i int) []byte {
	return x.text[i:]
}
```

Zastosowania tablic sufiksów – 1

```go
// LookupAll zwraca wycinek, którego elementami są wszystkie
// te pozycje, na których łańcuch `s` występuje w łańcuchu `x.text`
func (x Index) LookupAll(s []byte) []int {
    i := sort.Search(len(x.suffixes), func(i int) bool {
        return bytes.Compare(x.Suffix(x.suffixes[i]), s) >= 0
    })
    j := i + sort.Search(len(x.suffixes) - i, func(j int) bool {
        return !bytes.HasPrefix(x.Suffix(x.suffixes[i+j]), s)
    })
    return x.suffixes[i:j]
}
```

Zastosowania tablic sufiksów – 1

Tak wyszukuję wszystkie wystąpienia wzorca `issi` w tekście `mississippi`:

```
10: i
 7: ippi
>4: issippi
>1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

Wzorzec `issi` występuje w tekście `mississippi` na pozycjach 4 i 1

Zastosowania tablic sufiksów – 1

Korzystając z tablicy sufiksów, mogę znaleźć wszystkie wystąpienia danego wzorca P w danym tekście T w czasie $O(\log|T|)$ dzięki temu, że w funkcji `sort.Search` zastosowano wyszukiwanie binarne, które działa w czasie $O(\log|T|)$

Zastosowania tablic sufiksów – 2

```go
// lenOfCommonPrefix zwraca długość najdłuższego
// wspólnego prefiksu łańcuchów `s` i `t`
func lenOfCommonPrefix(s, t []byte) int {
	k := 0
	for ; k < min(len(s), len(t)); k++ {
		if s[k] != t[k] {
			return k
		}
	}
	return k
}
```

Zastosowania tablic sufiksów – 2

```
// LongestRepeatingSubstring znajduje najdłuższy taki łańcuch,
// który występuje w łańcuchu `text` co~najmniej `k` razy
func LongestRepeatingSubstring(text []byte, k int) []byte {
    index := SuffixArray(text)
    substr := []byte{}
    for i := 0; i + k <= len(a); i++ {
        m := lenOfCommonPrefix(
            index.Suffix(i), index.Suffix(i+k-1))
        if m > len(substr) {
            substr = index.Suffix(i)[:m]
        }
    }
    return substr
}
```

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście `mississippi`:

```
10: i
 7: ippi
 4: issippi
 1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście `mississippi`:

```
10: i
 7: ippi
 4: issippi
 1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

`i`

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście `mississippi`:

```
10: i
 7: ippi
 4: issippi
 1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

`i`

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście `mississippi`:

```
10: i
 7: ippi
 4: issippi
 1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

`issi`

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście `mississippi`:

```
10: i
 7: ippi
 4: issippi
 1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

`issi`

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście `mississippi`:

```
10: i
 7: ippi
 4: issippi
 1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

`issi`

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście `mississippi`:

```
10: i
 7: ippi
 4: issippi
 1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

`issi`

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście `mississippi`:

```
10: i
 7: ippi
 4: issippi
 1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

`issi`

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście `mississippi`:

```
10: i
 7: ippi
 4: issippi
 1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

`issi`

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście `mississippi`:

```
10: i
 7: ippi
 4: issippi
 1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

`issi`

Zastosowania tablic sufiksów – 2

Tak wyszukuję najdłuższy taki łańcuch, który powtarza się 2 razy w tekście `mississippi`:

```
10: i
 7: ippi
 4: issippi
 1: ississippi
 0: mississippi
 9: pi
 8: ppi
 6: sippi
 3: sissippi
 5: ssippi
 2: ssissippi
```

`issi`

Zastosowania tablic sufiksów – 2

Korzystając z tablic sufiksów mogę znaleźć najdłuższy taki łańcuch, który występuje co najmniej k razy w łańcuchu T, w czasie $O(|T|^2)$

Drzewa sufiksów

Peter Weiner

Amerykański informatyk i przedsiębiorca. Był promotorem doktoratu Briana Kernighana. Założył Interactive Systems Corporation, pierwszą komercyjną firmę zajmującą się UNIX-em. Wprowadził pojęcie drzewa sufiksów. Jest autorem algorytmu tworzenia drzew sufiksów, który działa w czasie liniowym względem długości łańcucha. Donald Knuth nazwał ten algorytm „algorytmem roku 1973”.

Drzewa trie

Drzewo trie – takie drzewo wyszukiwania, którego krawędzie są opisane przez fragmenty kluczy wyszukiwania

Nazwa trie pochodzi od wyrazu retrieval

Nieskompresowane drzewo trie – takie drzewo trie, którego krawędzie są opisane przez pojedyncze znaki

Skompresowane drzewo trie – takie drzewo trie, w którym każdy taki węzeł, z którego wychodzi dokładnie 1 krawędź, został połączony ze swoim rodzicem w 1 węzeł. Krawędzie skompresowanego drzewa trie są opisane przez niepuste podłańcuchy kluczy wyszukiwania

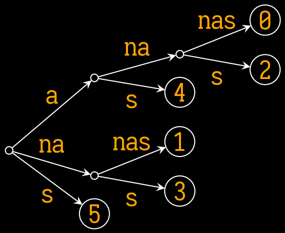
Drzewa sufiksów

Drzewo sufiksów łańcucha T to takie skompresowane drzewo trie, w którym:

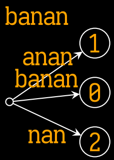
- Każda krawędź między węzłami jest opisana przez pewien niepusty podłańcuch łańcucha T
- Z każdego węzła wewnętrznego wychodzą 2 krawędzie lub więcej krawędzi
- Z żadnego węzła nie wychodzą takie 2 krawędzie, których opis zaczynałby się tym samym znakiem
- Jest $|T|$ liści. Te liście mają numery od 0 do $T - 1$
- Gdy połączę etykiety każdej krawędzi na ścieżce od korzenia drzewa do i-tego liścia tego drzewa, otrzymam sufiks $T[i:]$

Drzewa sufiksów – przykład

ananas



Drzewa sufiksów



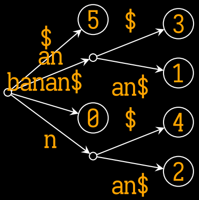
Jeśli pewien sufiks łańcucha T jest prefiksem innego sufiksu łańcucha T, czyli jeśli pewien sufiks łańcucha T jest równy innemu podłańcuchowi łańcucha T, to drzewo sufiksów łańcucha T ma mniej liści niż powinno mieć

Na przykład w drzewie sufiksów łańcucha banan brakuje liści 3 i 4 na końcach krawędzi an i n

W takim przypadku budujemy drzewa sufiksów takich łańcuchów, na których końcu jest dodany wartownik. Ten wartownik jest zwykle oznaczany znakiem dolara $

Drzewa sufiksów – przykład

banan$



Drzewo sufiksów – implementacja

Jak przechowywać węzły drzewa sufiksów? Jest kilka możliwości:

- Tablica wskaźników do węzłów potomnych, która ma tyle elementów, ile alfabet ma znaków
- Lista wskaźników do węzłów potomnych, posortowana według znaków
- Drzewo, na przykład drzewo binarne. Etykietami krawędzi tego drzewa są znaki, a węzły tego drzewa zawierają wskaźniki do węzłów potomnych
- Tablica haszująca znaki na wskaźniki do węzłów potomnych

Drzewo sufiksów – zajmowana pamięć

Przykładowa implementacja krawędzi drzewa sufiksów i węzła drzewa sufiksów:

```
type edge struct {
   firstChar  byte
   targetNode int
}
type node struct {
   start, end int
   suffixLink int
   outEdges   []edge
}
```

Drzewo sufiksów – zajmowana pamięć

- Krawędź drzewa sufiksów:
    - 2 słowa maszynowe – pierwszy znak etykiety i węzeł docelowy
- Węzeł drzewa sufiksów:
    - 2 słowa maszynowe – indeksy początku i końca podłańcucha, który opisuje krawędź
    - 1 słowo maszynowe – łącze do sufiksu, o którym będzie mowa później
    - 3 słowa maszynowe na początek, długość i pojemność wycinka `outEdges`
- Łańcuch `text`:
    - 1 bajt na każdy znak

Drzewo sufiksów – zajmowana pamięć

Drzewo sufiksów łańcucha `text` ma:

- `len(text)` liści
- co najwyżej `len(text)` węzłów wewnętrznych
- o 1 krawędź mniej niż ma węzłów

Zatem podana implementacja drzewa sufiksów łańcucha `text` zajmuje:

- co najwyżej `(2*len(text))*6` słów maszynowych na węzły
- co najwyżej `(2*len(text))*2` słowa maszynowe na krawędzie
- `len(text)` bajtów na łańcuch `text`

Zatem drzewo sufiksów zajmuje co najwyżej `(16*sizeof(int)+1)*len(text)` bajtów

Drzewa sufiksów – wyzwanie 1

Udowodnię, że drzewo sufiksów łańcucha T:

- ma tyle samo liści, ile łańcuch T ma znaków
- ma mniej wierzchołków wewnętrznych niż łańcuch T ma znaków,

czyli że drzewo sufiksów łańcucha T ma rozmiar $O(|T|)$

Drzewa sufiksów – wyzwanie 2

Spośród łańcuchów, które mają 5 znaków, podam łańcuch, którego drzewo sufiksów ma najwięcej krawędzi

Zastosowania drzew sufiksów

Zastosowania drzew sufiksów – 1

Tak sprawdzam, czy wzorzec P występuje w tekście T:

- Buduję drzewo sufiksów łańcucha T
- Szukam w drzewie sufiksów łańcucha T takiej ścieżki od korzenia, że połączone etykiety krawędzi, które składają się na tę ścieżkę, tworzą łańcuch P
- Jeśli w drzewie sufiksów istnieje taka ścieżka, to znaczy, że wzorzec P występuje w tekście T
- Jeśli w drzewie sufiksów nie ma takiej ścieżki, to znaczy, że wzorzec P nie występuje w tekście T

Zastosowania drzew sufiksów – 2

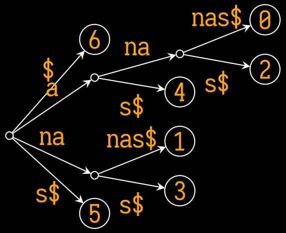
Znam tekst T. Ten tekst się nie zmienia. Chcę znaleźć wszystkie wystąpienia każdego z wielu wzorców $P_1, P_2, \ldots P_n$ w tym tekście

Tworzę drzewo sufiksów łańcucha T

Przechodzę ścieżką od korzenia drzewa wzdłuż tych krawędzi, których etykiety łączą się we wzorzec P_i. W ten sposób dochodzę do pewnego węzła N. Potem przeszukuję węzły potomne węzła N, na przykład w głąb. Każdy liść drzewa, który znajduję, zawiera indeks pozycji, od której zaczyna się pewne wystąpienie wzorca P_i w tekście

W ten sposób mogę znaleźć w tekście T wszystkie wystąpienia dowolnego wzorca P_i w czasie $O(|P_i| +$ liczba wystąpień wzorca P_i w tekście $T)$. Ten czas nie zależy od długości łańcucha T

Zastosowania drzew sufiksów – 2



Wystąpienia wzorca na zaczynają się od pozycji 1 i 3

Wystąpienia wzorca an zaczynają się od pozycji 0 i 2

Wystąpienia wzorca a zaczynają się od pozycji 0, 2 i 4

Wystąpień wzorca x nie ma

Zastosowania drzew sufiksów – 3

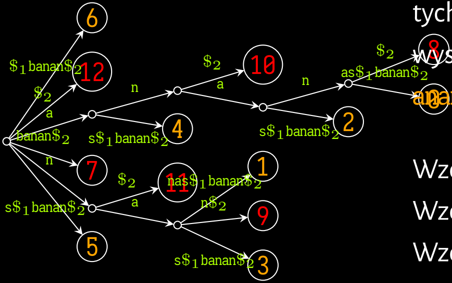
Jak znaleźć wszystkie te łańcuchy ze zbioru $\{T_1, T_2, \ldots T_n\}$, które zawierają dany wzorzec P?

Tworzę drzewo sufiksów łańcucha $T_1\$_1 T_2\$_2 \ldots T_n\$_n$

Przechodzę ścieżką od korzenia drzewa sufiksów łańcucha $T_1\$_1 T_2\$_2 \ldots T_n\$_n$ wzdłuż tych krawędzi, których etykiety łączą się we wzorzec P. W ten sposób dochodzę do pewnego węzła N

Potem przeszukuję węzły potomne węzła N, na przykład w głąb. Każdy liść drzewa, który znajduję, zawiera indeks pozycji, od której zaczyna się pewne wystąpienie wzorca P w tekście

Jak znaleźć wszystkie te łańcuchy ze zbioru $\{T_1, T_2, \ldots T_n\}$, które zawierają dany wzorzec P?



Wyznaczam te łańcuchy, które zawierają dany wzorzec, na podstawie indeksów tych pozycji, od których zaczynają się wystąpienia tego wzorca w tekście

ananas$\$_1$banan$\$_2$

Wzorzec an występuje w łańcuchach 1 i 2

Wzorzec na występuje w łańcuchach 1 i 2

Wzorzec as występuje w łańcuchu 1

Wzorzec ban występuje w łańcuchu 2

Wzorzec x nie występuje w żadnym z tych łańcuchów

Zastosowania drzew sufiksów – 3

Jak znaleźć wszystkie łańcuchy ze zbioru $\{T_1, T_2, \ldots T_n\}$, które zawierają dany wzorzec P?

Przykład zastosowania: Baza danych DNA mitochondrialnego żołnierzy USA

Pobieram DNA mitochondrialne od żyjących żołnierzy. Sekwencjonuję mały fragment DNA każdego żołnierza. Wybieram ten fragment tak, żeby był różny u różnych żołnierzy. Zapisuję ten fragment w bazie danych jako identyfikator żołnierza

Tak identyfikuję poległych żołnierzy:

Pobieram DNA mitochondrialne z ciał poległych żołnierzy. Sekwencjonuję ten sam fragment DNA. Porównuję ten fragment z identyfikatorami żołnierzy. Może się zdarzyć, że z ciała nie da się pobrać pełnego fragmentu DNA. Dlatego szukam takich identyfikatorów, które zawierają pobrany fragment

Zastosowania drzew sufiksów – 4

Jak znaleźć najdłuższy wspólny podłańcuch 2 łańcuchów T_1 i T_2?

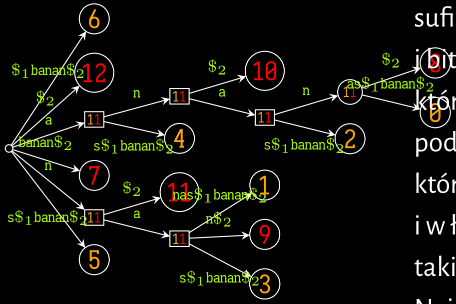
Buduję drzewo sufiksów łańcucha $T_1\$_1T_2\$_2$. Dodaję do każdego z węzłów wewnętrznych tego drzewa 2 bity

W każdym węźle wewnętrznym N tego drzewa ustawiam odpowiednie bity:

- bit 0, jeśli dowolny liść tego poddrzewa, którego korzeniem jest węzeł N, odpowiada pewnemu sufiksowi łańcucha $T_1\$_1T_2\$_2$
- bit 1, jeśli dowolny liść tego poddrzewa, którego korzeniem jest węzeł N, odpowiada pewnemu sufiksowi łańcucha $T_2\$_2$
- bit 0 i bit 1, jeśli dowolny liść tego poddrzewa, którego korzeniem jest węzeł N, odpowiada pewnemu sufiksowi łańcucha $T_1\$_1T_2\$_2$ i dowolny liść tego poddrzewa, którego korzeniem jest węzeł N, odpowiada pewnemu sufiksowi łańcucha $T_2\$_2$

Obliczam 2 bity dodane do każdego węzła wewnętrznego jako sumę logiczną (OR) 2 bitów dodanych do dzieci tego węzła, przeszukując drzewo sufiksów w głąb w kolejności wstecznej (post-order), czyli odwiedzając najpierw dzieci każdego węzła wewnętrznego, a potem ten węzeł

Jak znaleźć najdłuższy wspólny podłańcuch łańcuchów ananas i banan?



Przeszukuję od korzenia te węzły drzewa sufiksów, w których jest ustawiony i bit 0, i bit 1. Etykiety tych krawędzi, wzdłuż których przechodzę, łączą się w takie podłańcuchy łańcucha ananas$\$_1$banan$\$_2$, które występują i w łańcuchu ananas, i w łańcuchu banan. Szukam najdłuższego takiego wspólnego podłańcucha

Najdłuższy wspólny podłańcuch łańcuchów ananas i banan to łańcuch anan

Jak wykrywać zanieczyszczenia w DNA?

Przykład: DNA rzekomo wyekstrahowane z kości dinozaura okazało się bardziej podobne do ludzkiego DNA niż do DNA ptaków i krokodyli

Dany jest łańcuch DNA T_1, połączone łańcuchy DNA możliwych zanieczyszczeń T_2 i liczba całkowita k. Chcę znaleźć wszystkie takie podłańcuchy łańcucha T_2, które występują w łańcuchu T_1 i mają długość większą niż k

Jak wykrywać zanieczyszczenia w DNA?

Buduję drzewo sufiksów łańcucha $T_1\$_1T_2\$_2$. Dodaję do każdego z węzłów wewnętrznych tego drzewa 2 bity

W każdym węźle wewnętrznym N tego drzewa ustawiam odpowiednie bity na 1:

- bit 0, jeśli dowolny liść tego poddrzewa, którego korzeniem jest węzeł N, odpowiada pewnemu sufiksowi łańcucha $T_1\$_1T_2\$_2$
- bit 1, jeśli dowolny liść tego poddrzewa, którego korzeniem jest węzeł N, odpowiada pewnemu sufiksowi łańcucha $T_2\$_2$
- bit 0 i bit 1, jeśli dowolny liść tego poddrzewa, którego korzeniem jest węzeł N, odpowiada pewnemu sufiksowi łańcucha $T_1\$_1T_2\$_2$ i dowolny liść tego poddrzewa, którego korzeniem jest węzeł N, odpowiada pewnemu sufiksowi łańcucha $T_2\$_2$

Zastosowania drzew sufiksów – 5

Wyznaczam 2 bity dodane do każdego węzła wewnętrznego jako sumę logiczną (OR) 2 bitów dodanych do dzieci tego węzła, przeszukując drzewo sufiksów w głąb w kolejności wstecznej (post-order), czyli odwiedzając najpierw dzieci każdego węzła wewnętrznego, a potem ten węzeł

Przeszukuję od korzenia te węzły drzewa sufiksów, w których jest ustawiony i bit 0, i bit 1. Etykiety tych krawędzi, wzdłuż których przechodzę, łączą się w takie podłańcuchy łańcucha $T_1\$_1T_2\$_2$, które występują i w łańcuchu T_1, i w łańcuchu T_2. Szukam wszystkich takich podłańcuchów, których długość jest większa niż dana stała k

Zastosowania drzew sufiksów – 6

Ile razy dany wzorzec P występuje w danym tekście T?

Buduję drzewo sufiksów łańcucha T

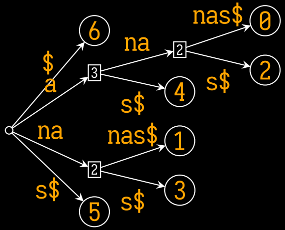
Dodaję licznik do każdego węzła tego drzewa. W tym liczniku zapamiętuję, ile liści ma poddrzewo, którego korzeniem jest ten węzeł. Wyznaczam wartość licznika dodanego do każdego węzła jako sumę wartości liczników dodanych do dzieci tego węzła, przeszukując drzewo sufiksów w głąb w kolejności wstecznej (post-order), czyli odwiedzając najpierw dzieci każdego węzła wewnętrznego, a potem ten węzeł

Zastosowania drzew sufiksów – 6

Ile razy dany wzorzec *P* występuje w danym tekście *T*?

Przechodzę ścieżką od korzenia tego drzewa sufiksów wzdłuż tych krawędzi, których etykiety składają się we wzorzec *P*. W ten sposób dochodzę do pewnego węzła *N*. Odczytuję z licznika węzła *N* liczbę liści w poddrzewie węzła *N*. Gdy nie ma takiego węzła, to znaczy, że wzorzec *P* występuje 0 razy w tekście *T*

Ile razy dany wzorzec występuje w danym tekście?



Zastosowania drzew sufiksów – 7

Jak znaleźć najdłuższy podłańcuch wspólny dla wielu różnych łańcuchów T_1, T_2, ..., T_n, ale niekoniecznie wspólny dla wszystkich tych łańcuchów?

Przykład: DNA wielu różnych gatunków. Mutacje tych fragmentów DNA, które kodują krytyczne funkcje, i mutacje innych fragmentów DNA zdarzają się z jednakowym prawdopodobieństwem. Jednak mutacje tych fragmentów DNA, które kodują krytyczne funkcje, częściej powodują przedwczesną śmierć organizmu, przez co ten organizm nie ma potomków. Zatem te fragmenty DNA, które kodują krytyczne funkcje, mniej się zmieniają z pokolenia na pokolenie.

Gdy znajdę najdłuższe wspólne podłańcuchy DNA wielu gatunków, dowiem się, które fragmenty DNA kodują krytyczne funkcje

Jak znaleźć najdłuższy podłańcuch wspólny dla wielu różnych łańcuchów T_1, T_2, ..., T_n, ale niekoniecznie wspólny dla wszystkich tych łańcuchów?

Buduję drzewo sufiksów łańcucha $T_1\$_1T_2\$_2 \ldots T_n\$_n$. Przypisuję do każdego z węzłów wewnętrznych tego drzewa tablicę n bitów. k-ty bit tej tablicy w węźle wewnętrznym N jest ustawiony na 1, gdy dowolny liść tego poddrzewa, którego korzeniem jest węzeł N, odpowiada pewnemu sufiksowi łańcucha $T_1\$_1 \ldots T_k\$_k \ldots T_n\$_n$, który zaczyna się w łańcuchu T_k

Obliczam tablicę bitów każdego węzła wewnętrznego jako sumę logiczną (OR) tablic bitów dodanych do dzieci tego węzła, przeszukując drzewo sufiksów w głąb w kolejności wstecznej (post-order), czyli odwiedzając najpierw dzieci każdego węzła wewnętrznego, a potem ten węzeł

W tablicy bitów węzła N tyle bitów jest ustawionych na 1, ile różnych numerów łańcuchów pojawia się w liściach tego poddrzewa, którego korzeniem jest węzeł N

Przy ustalonej liczbie ℓ przeszukuję od korzenia takie węzły drzewa sufiksów, że w tablicach bitów tych węzłów co najmniej ℓ bitów jest ustawionych. Etykiety tych krawędzi, wzdłuż których przechodzę, łączą się w podłańcuchy łańcucha $T_1\$_1T_2\$_2\ldots T_n\$_n$. Szukam najdłuższego takiego podłańcucha

Algorytm Ukkonena

Tę część wykładu opracowałem na podstawie odpowiedzi Johannesa Gollera ze Stack Overflow: `https://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english`

Esko Ukkonen (26.1.1950–)

Fiński informatyk. Emerytowany profesor Uniwersytetu Helsińskiego. W 1995 roku opublikował jeden z prostszych algorytmów budowania drzew sufiksów. Ten algorytm działa w czasie liniowym względem długości łańcucha. W 2000 roku Esko Ukkonen otrzymał najwyższe odznaczenie Finlandii, Order Białej Róży Finlandii.

Drzewo sufiksów to skompresowane drzewo trie. Etykiety jego krawędzi to pary liczb całkowitych [od,do]. Liczby od i do to indeksy początku podłańcucha i końca podłańcucha. Chociaż etykieta każdej krawędzi może mieć dowolną długość, zajmuje tylko $O(1)$ komórek pamięci

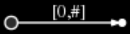
Algorytm Ukkonena wykonuje kroki, przeglądając znaki łańcucha T od początku do końca tego łańcucha. Na każdy znak ciągu przypada jeden krok. Chociaż każdy krok może obejmować więcej niż jedną pojedynczą operację, wykażę, że całkowita liczba operacji wynosi $O(|T|)$, i że każda z tych operacji zajmuje czas $O(1)$

Algorytm Ukkonena

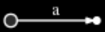
abcabxabcd

Krok 1: Zaczynam od lewej. Najpierw dodaję do drzewa sufiksów tylko pojedynczy znak a, tworząc krawędź od korzenia drzewa, czyli od węzła root, do liścia i oznaczając ją [0,#]. Oznacza to, że krawędź reprezentuje podłańcuch, który zaczyna się na pozycji 0 i kończący na bieżącym końcu. Symbolem # oznaczam bieżący koniec, który znajduje się na pozycji 1 (zaraz za a)

Zatem początkowe drzewo sufiksów wygląda tak:



czyli

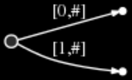


abcabxabcd

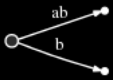
Krok 2: Teraz przechodzę do pozycji 2 (zaraz za b). Celem każdego kroku jest dodać wszystkie sufiksy aż do bieżącej pozycji. Aby osiągnąć ten cel:

- rozszerzam istniejącą krawędź a do ab
- wstawiam jedną nową krawędź dla b

Teraz drzewo sufiksów wygląda tak:



czyli



Algorytm Ukkonena

:

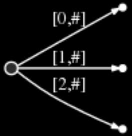
- Reprezentacja krawędzi ab jest taka sama jak w początkowym drzewie: [0,#]. Jej znaczenie samo się zmieniło, bo zmieniłem bieżącą pozycję # z 1 na 2
- Każda krawędź zajmuje $O(1)$ miejsca, bo składa się tylko z dwóch wskaźników do tekstu, niezależnie od tego, jak długiemu podłańcuchowi odpowiada

Algorytm Ukkonena

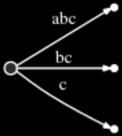
abcabxabcd

Krok 3: Znowu zwiększam pozycję o 1 i aktualizuję drzewo sufiksów, dodając znak c na końcu każdej istniejącej krawędzi i wstawiając jedną nową krawędź dla nowego sufiksu c

Teraz drzewo sufiksów wygląda tak:



czyli

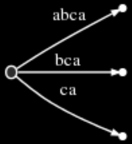


Zauważam, że:

- Po każdym kroku drzewo to trie, które buduję, jest poprawnym drzewem sufiksów prefiksu łańcucha T aż do bieżącej pozycji
- Wykonałem tyle kroków, ile przeczytałem znaków
- W każdym kroku wykonałem $O(1)$ operacji, bo zwiększanie zmiennej # o 1 zajmuje czas $O(1)$ i jedną nową krawędź dla ostatniego znaku można wstawić w czasie $O(1)$
- Zatem drzewo sufiksów takiego łańcucha `text`, w którym znaki się nie powtarzają, można zbudować w czasie $O(|\texttt{text}|)$

abcabxabcd

Krok 4: Przesuwam # na pozycję 4. W ten sposób niejawnie przesuwam końce wszystkich krawędzi aż do tej pozycji. Teraz drzewo sufiksów wygląda tak:



Algorytm Ukkonena

Teraz należy wstawić do korzenia a, ostatni sufiks bieżącego kroku

Zanim to zrobię, wprowadzam oprócz zmiennej # jeszcze 2 zmienne:

- Aktywne położenie: trójka (`active_node`, `active_edge`, `active_length`)
- `backlog`: liczba całkowita, która wskazuje, ile nowych sufiksów należy dodać do drzewa

Algorytm Ukkonena

- Kiedy wstawiałem do drzewa sufiksów znaki `abc`, aktywnym położeniem było zawsze (`root`, `'$'`, `0`), czyli aktywnym węzłem był korzeń drzewa, aktywną krawędzią był znak wartownika, a aktywna długość była równa 0. Zatem każda nowa krawędź, którą dodawałem do drzewa sufiksów, była dodawana do korzenia drzewa
- Na początku każdego kroku zmienna `backlog` była ustawiana na 1. Oznaczało to, że na końcu każdego kroku należało wstawić do drzewa sufiksów tylko 1 sufiks

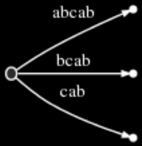
Teraz znak a znowu występuje w łańcuchu. Dlatego aktywne położenie i zmienna `backlog` będą się zmieniać. Kiedy wstawiam do korzenia drzewa bieżący ostatni znak a, zauważam, że z korzenia już wychodzi krawędź, która zaczyna się od znaku a: krawędź abca. W takim przypadku:

- Nie dodaję nowej krawędzi [4,#] do korzenia drzewa. Zamiast tego zauważam, że sufiks a już jest dodany do drzewa. Nie przeszkadza mi to, że ten sufiks kończy się wewnątrz dłuższej krawędzi.
- Ustawiam aktywne położenie na (`root`, `'a'`, `1`). Oznacza to, że aktywne położenie znajduje się za pozycją 1 tej krawędzi, która wychodzi z węzła `root` i ma etykietę a. Zauważam, że pierwszy znak krawędzi jednoznacznie wyznacza tę krawędź, bo z danego węzła może wychodzić tylko jedna krawędź, która zaczyna się od danego znaku
- Zwiększam wartość zmiennej `backlog` o 1. Teraz ta zmienna ma wartość 2

Zauważam, że gdy ostatni sufiks, który należy wstawić do drzewa sufiksów, już występuje w drzewie sufiksów, nie zmieniam drzewa sufiksów. W takim przypadku zmieniam tylko bieżące położenie i wartość zmiennej `backlog`. Mogę je zmienić w czasie $O(1)$

abcabxabcd

Krok 5: Zmieniam bieżącą pozycję # na 5. Teraz drzewo sufiksów wygląda tak:



Zmienna `backlog` ma wartość 2, więc należy wstawić do drzewa sufiksów 2 ostatnie sufiksy tego podłańcucha, który kończy się na bieżącej pozycji

- Sufiks a z poprzedniego kroku nie został wstawiony do drzewa sufiksów. W bieżącym kroku zwiększyłem bieżącą pozycję o 1, więc ten sufiks urósł z a do ab
- Należy też wstawić do drzewa sufiksów nowy ostatni sufiks, czyli b

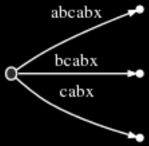
Chcę wstawiać te 2 sufiksy do drzewa sufiksów w kolejności od najdłuższego do najkrótszego sufiksu

Aktywne położenie wskazuje, gdzie kończy się podłańcuch a i należy wstawić sufiks b: za pierwszym znakiem a na krawędzi abcab. Okazuje się, że znak b już jest na krawędzi abcab za pierwszym znakiem a. Zatem nie zmieniam drzewa sufiksów, tylko:

- Zmieniam aktywne położenie na (`root`, `'a'`, `2`). Oznacza to, że aktywne położenie znajduje się na tej samej krawędzi abcab wychodzącej z tego samego węzła `root`, o 1 znak dalej, czyli za pierwszym znakiem b
- Zwiększam wartość zmiennej `backlog` do 3

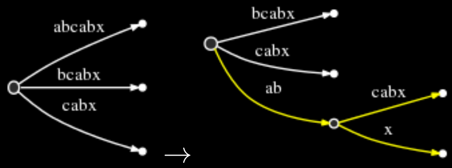
abcabxabcd

Krok 6: Zwiększam wartość zmiennej # o 1. Teraz drzewo sufiksów wygląda tak:



`abcabxabcd`

Zmienna `backlog` ma wartość 3. Zatem należy wstawić do drzewa sufiksów sufiksy `abx`, `bx` i `x`. Aktywne położenie wskazuje, gdzie kończy się podłańcuch `ab` i należy wstawić sufiks `x`: za pierwszym znakiem `b` krawędzi `abcabx` wychodzącej z węzła `root`. Znaku `x` nie ma w tym miejscu krawędzi `abcabx`. Zatem dzielę krawędź `abcabx` i wstawiam do drzewa sufiksów nowy węzeł wewnętrzny:



Algorytm Ukkonena

Każda krawędź zawiera indeksy początku i końca podłańcucha, zatem mogę podzielić krawędź i wstawić do drzewa sufiksów nowy węzeł wewnętrzny w czasie $O(1)$

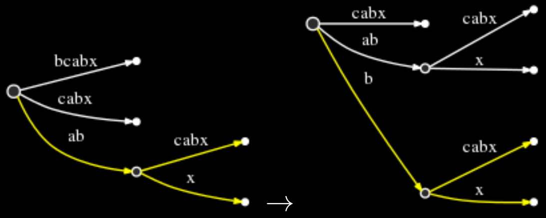
Poradziłem sobie z sufiksem abx. Zmniejszam wartość zmiennej `backlog` do 2. Chcę wstawić do drzewa sufiksów następny zaległy sufiks, czyli bx. Najpierw jednak należy zaktualizować aktywne położenie. Korzystam z tej reguły:

Reguła 1: Jeśli `active_node == root` i `active_length > 0`, to:

- `active_node` nadal będzie mieć wartość `root`
- Przesuwam `active_edge` w prawo do pierwszego znaku następnego zaległego sufiksu
- Zmniejszam `active_length` o 1

abcabxabcd

Zatem nowe aktywne położenie (root , 'b' ,1) wskazuje, że następny sufiks należy wstawić na krawędzi bcabx, za pierwszym znakiem, czyli za znakiem b. Mogę znaleźć to miejsce w czasie $O(1)$ i sprawdzić, czy znak x już tam jest, czy nie. Gdyby znak x tam był, zakończyłbym krok 6. Ale znaku x tam nie ma, więc wstawiam go, dzieląc krawędź bcabx:



W czasie $O(1)$ podzieliłem krawędź i zmieniłem:

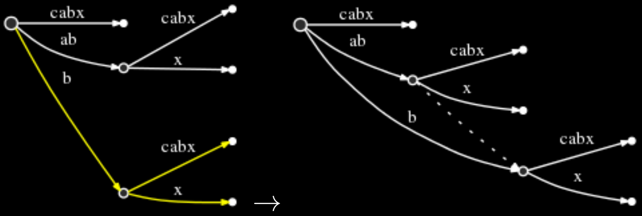
- wartość zmiennej `backlog` na 1
- aktywne położenie na (`root`, `'x'`, `0`) zgodnie z Regułą 1

Należy zrobić jeszcze jedną rzecz. Mówi o tym ta reguła:

Reguła 2: Kiedy podzieliłem jakąś krawędź i dodałem do drzewa sufiksów nowy węzeł, i nie jest to pierwszy węzeł, który dodałem do drzewa w bieżącym kroku, łączę ten węzeł, który dodałem jako przedostatni, z tym węzłem, który dodałem jako ostatni, specjalnym łączem do sufiksu. Później będzie mowa o tym, do czego się przydają te łącza

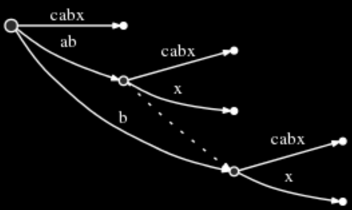
abcabxabcd

Kropkowana linia oznacza łącze do sufiksu:

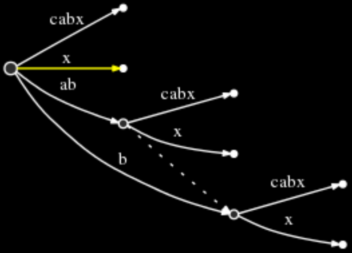


abcabxabcd

Nadal chcę wstawić do drzewa sufiksów ostatni sufiks bieżącego kroku, czyli sufiks x. Wartość zmiennej active_length spadła do 0, więc wstawiam ten sufiks do korzenia drzewa. Z korzenia drzewa nie wychodzi żadna krawędź, która zaczynałaby się od znaku x, więc wstawiam do korzenia drzewa nową krawędź:



→



Algorytm Ukkonena

Krok 7: Zmieniam wartość zmiennej # na 7. W ten sposób dodaję znak `a` na końcu wszystkich tych krawędzi, które prowadzą do liści drzewa sufiksów. Potem próbuję wstawić nowy ostatni znak, czyli `a`, w `active_node`, czyli w węźle `root`. Z węzła `root` już wychodzi krawędź, która zaczyna się od znaku `a`, więc:

- Zmieniam aktywne położenie na (`root`, `'a'`, `1`)
- Kończę ten krok

Krok 8: Zmieniam wartość zmiennej # na 8. W ten sposób dodaję znak `b` na końcu wszystkich tych krawędzi, które prowadzą do liści drzewa sufiksów. Na krawędzi `ab`, która wychodzi z węzła `root`, za jej pierwszym znakiem już jest znak `b`, więc podobnie jak w poprzednich krokach zmieniam aktywne położenie na (`root`, `'a'`, `2`) i wartość zmiennej `backlog` na 2. W czasie $O(1)$ zauważam, że aktywne położenie znajduje się teraz na końcu krawędzi `ab`. Zatem zmieniam je na (`node_ab`, `'\0'`, `0`). Węzłem `node_ab` nazywam ten węzeł, do którego prowadzi z węzła `root` krawędź z etykietą `ab`

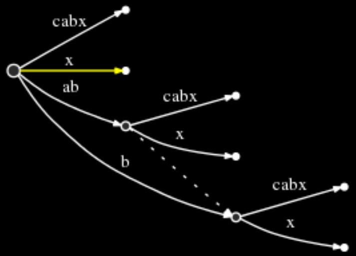
abcabxabcd

Krok 9: Należy wstawić do drzewa sufiksów znak c. Zmieniam wartość zmiennej # na 9. W ten sposób w czasie $O(1)$ dodaję znak c na końcu wszystkich tych krawędzi, które prowadzą do liści drzewa sufiksów. Z węzła node_ab już wychodzi krawędź, której etykieta zaczyna się od znaku c. Zatem:

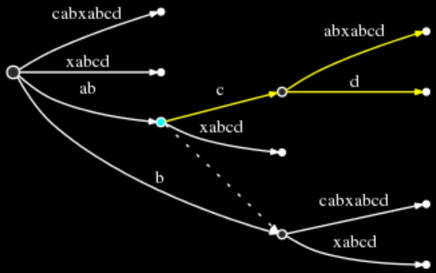
- Zmieniam aktywne położenie na (node_ab, `'c'`, 1)
- Zwiększam wartość zmiennej `backlog` o 1, do 4
- Kończę ten krok

abcabxabcd

Krok 10: Zwiększam wartość zmiennej # do 10. Zmienna backlog ma wartość 4, więc wstawiam do drzewa sufiksów sufiks abcd w aktywnym położeniu. Powoduje to podział krawędzi cabxabcd w czasie $O(1)$:



$\rightarrow$



Algorytm Ukkonena

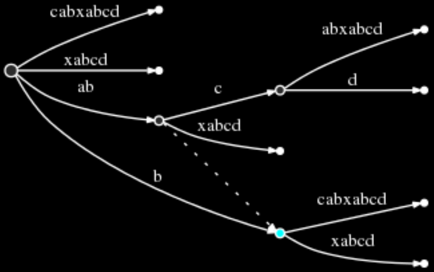
Węzeł active_node jest oznaczony kolorem. Oto ostatnia reguła:

Reguła 3: Gdy podzieliłem krawędź od węzła active_node, który nie jest węzłem root:

- Jeśli z węzła active_node wychodzi łącze do sufiksu, to ustawiam węzeł active_node na ten węzeł, na który wskazuje to łącze do sufiksu
- Jeśli z węzła active_node nie wychodzi łącze do sufiksu, to ustawiam węzeł active_node na węzeł root
- Nie zmieniam wartości zmiennych active_edge i active_length

abcabxabcd

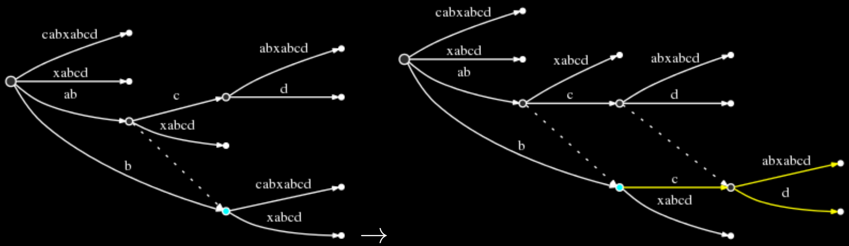
Aktywnym położeniem jest teraz (node_b, 'c', 1). Węzeł node_b jest oznaczony kolorem



Algorytm Ukkonena

abcabxabcd

Zmniejszam wartość zmiennej `backlog` do 3. Chcę dodać do drzewa sufiksów sufiks bcd. Dzięki regule 3 ustawiłem aktywne położenie na właściwy węzeł i krawędź, więc wystarczy wstawić ostatni znak tego sufiksu, czyli znak d, w bieżącym położeniu. Powoduje to kolejny podział krawędzi. Zgodnie z regułą 2 tworzę łącze do sufiksu od przedostatniego do ostatniego wstawionego węzła:

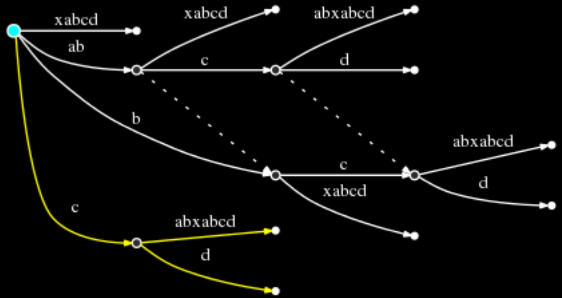


Algorytm Ukkonena

Dzięki łączom do sufiksów mogę zmieniać aktywne położenie w czasie $O(1)$

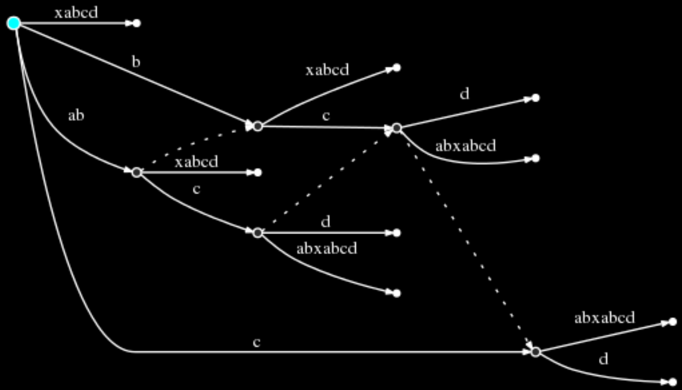
abcabxabcd

Krok 10 jeszcze się nie skończył. Zmienna backlog ma wartość 2. Trzeba skorzystać z reguły 3 i znowu zmienić aktywne położenie. Z węzła active_node nie wychodzi łącze do sufiksu, więc ustawiamy ten węzeł na węzeł root. Aktywne położenie to teraz (root, c, 1). Wstawiam znak d za pierwszym znakiem tej krawędzi wychodzącej z węzła root, która ma etykietę cabxabcd. Powoduje to kolejny podział krawędzi:



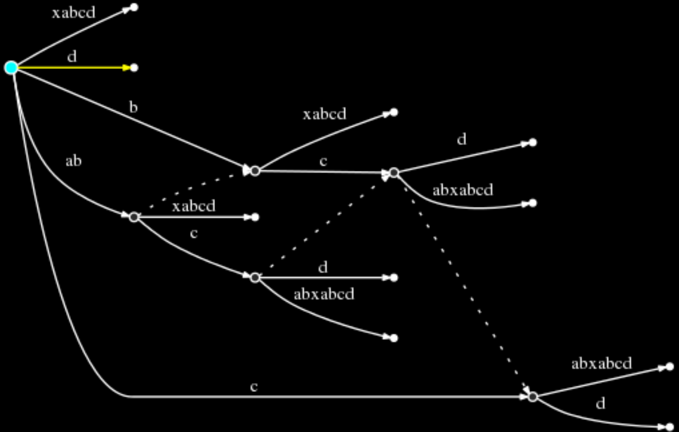
abcabxabcd

Stosuję regułę 2. Dodaję łącze do sufiksu do przedostatniego węzła utworzonego w tym kroku:



`abcabxabcd`

Teraz zmienna `backlog` ma wartość 1. Węzeł `active_node` to węzeł `root`. Korzystając z reguły 1 zmieniam aktywne położenie na (`root`, `'d'`, `0`). Ostatnie wstawienie w tym kroku to wstawienie sufiksu `d` w węźle `root`:



Gdy tekst ma n znaków, algorytm Ukkonena wykonuje n kroków. W każdym kroku albo tylko zmienia wartości zmiennych w czasie $O(1)$, albo ponadto dodaje do drzewa zaległe sufiksy, też w czasie $O(1)$. Zmienna `backlog` wskazuje, ile razy algorytm nie dodawał sufiksów do drzewa w poprzednich krokach. Ta zmienna zmniejsza się o 1 zawsze wtedy, kiedy algorytm dodaje sufiks do drzewa. Zatem algorytm Ukkonena dodaje do drzewa sufiksy $O(n)$ razy. Zatem całkowita złożoność czasowa algorytmu Ukkonena to $O(n)$.

Podsumowanie

Podsumowanie

- Tablice sufiksów
- Zastosowania tablic sufiksów
- Drzewa sufiksów
- Zastosowania drzew sufiksów
- Algorytm Ukkonena budowania drzew sufiksów

Pomysły, uwagi, pytania, sugestie

Proszę wysyłać podpisane pomysły, uwagi, pytania, sugestie na temat wykładów lub laboratoriów na adres `mgc@agh.edu.pl`

lub wpisywać anonimowe pomysły, uwagi, pytania, sugestie pod adresem `https://tiny.cc/algorytmy-tekstowe`

Do zobaczenia
na następnym wykładzie

Jego tematem będą pomysłowe algorytmy tekstowe

Źródła ilustracji

Zdeněk Jirotka, Saturnin – zrzut ekranu

https://commons.wikimedia.org/wiki/File:DNA123_rotated.png

https://classic.csunplugged.org/activities/text-compression/

Marshall Custiss Hazard, A Complete Concordance to the American Standard Version of the Holy Bible – zrzut ekranu z Archive.org

Donald E. Knuth, The Art of Computer Programming, tom 3

https://en.wikipedia.org/wiki/Udi_Manber

https://en.wikipedia.org/wiki/Eugene_Myers

https://www.linkedin.com/in/pgweiner/

https://www.cs.helsinki.fi/u/ukkonen/

https://stackoverflow.com/questions/9452701/
ukkonens-suffix-tree-algorithm-in-plain-english

Źródła zdjęć