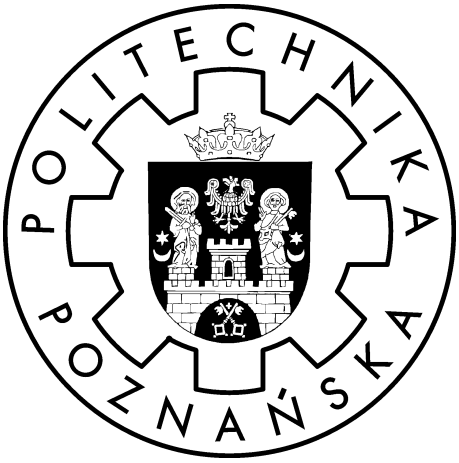
Politechnika Poznańska
Wydział Elektryczny
Instytut Automatyki, Robotyki i Inżynierii Informatycznej



Praca dyplomowa magisterska

BADAWCZY SYSTEM ROZPOZNAWANIA WYKORZYSTUJĄCY OBRAZ TĘCZÓWKI OKA

Jakub Lamprecht

Promotor
dr inż. Tomasz Piaścik

Poznań, 2019

Tutaj przychodzi karta pracy dyplomowej;
oryginał wstawiamy do wersji dla archiwum PP, w pozostałych kopiach wstawiamy ksero.

Spis treści

Rozdział 1

Wprowadzenie

Ogolny opis pracy - wstep do tematyki + motywacja wyboru tematu.

1.1 Cel i zakres pracy

Celem tej pracy było zaprojektowanie i stworzenie aplikacji pozwalającej na przeprowadzenie procesu rozpoznawania tęczówki oka oraz przegląd literatury i implementacja algorytmów normalizacji, enkodowania oraz dopasowania tęczówki. Jednym z wymagań dla tworzonego programu, było aby działał on w dwóch trybach:

- Krokowym - tryb umożliwiający użytkownikowi stopniowe przejście przez proces rozpoznawania z możliwością wyboru metod oraz parametrów dla poszczególnych kroków, a także podglądu wyników dla każdego z nich.
- Wsadowym - tryb umożliwiający automatyczne przeprowadzenie procesu rozpoznawania opisanego przez plik konfiguracyjny wygenerowany w trybie krokowym dla większego zbioru obrazów.

Ważnym aspektem dla tworzonej aplikacji była również użyteczność i przyjazność interfejsu użytkownika, który miał zachęcać do korzystania z aplikacji.

1.2 Struktura pracy

Opis poszczególnych rozdziałów - zrób jak już wymyślisz rozdziały.

Rozdział 2

Podstawy teoretyczne

2.1 Podstawowe pojęcia

Niech Σ oznacza skończony zbiór o rozmiarze $|\Sigma| \geq 1$, zwany dalej alfabetem [23]. Elementami tego zbioru są symbole (w przypadku tekstu zwykle utożsamiane z pojedynczymi literami lub znakami). Alfabet określa się jako indeksowalny, jeżeli istnieje permutacja indeksów taka, że $\forall_{i=0..|\Sigma|-1} : a_0 < a_1 < \cdots < a_{|\Sigma|-1}$ (symbole mogą być uporządkowane).

Niech x oznacza ciąg symboli (pojęcia symbol, znak i litera będą stosowane zamiennie) o długości n. Element ciągu x występujący na pozycji i oznacza się jako $x[i]$, $i \in \langle 0, n-1 \rangle$. Przez $x[a..b]$, $0 \leq a \leq b \leq n-1$ rozumie się podciąg x rozpoczynający się na pozycji a, a kończący na pozycji b [7]. Warto zauważyć, że $x = x[0..n-1]$.

Prefiks to taki podciąg x, którego pierwszy element jest jednocześnie pierwszym elementem ciągu x, tj. $x[0..b]$, $0 \leq b \leq n-1$. *Sufiks* oznacza taki podciąg x, którego ostatni element jest ostatnim znakiem ciągu x, tj. $x[a..n-1], 0 \leq a \leq n-1$. Przez S_i^x (lub S_i w przypadku braku niejednoznaczności) rozumie się sufiks $x[i..n-1]$. Ciąg x jest jednocześnie swoim najdłuższym prefiksem jak i sufiksem [7].

Konkatenacją dwóch ciągów x i y o długości n i m nazwiemy ciąg z, dla którego $z = x \circ y = x[0]x[1]..x[n-1]y[0]y[1]..y[m-1]$.

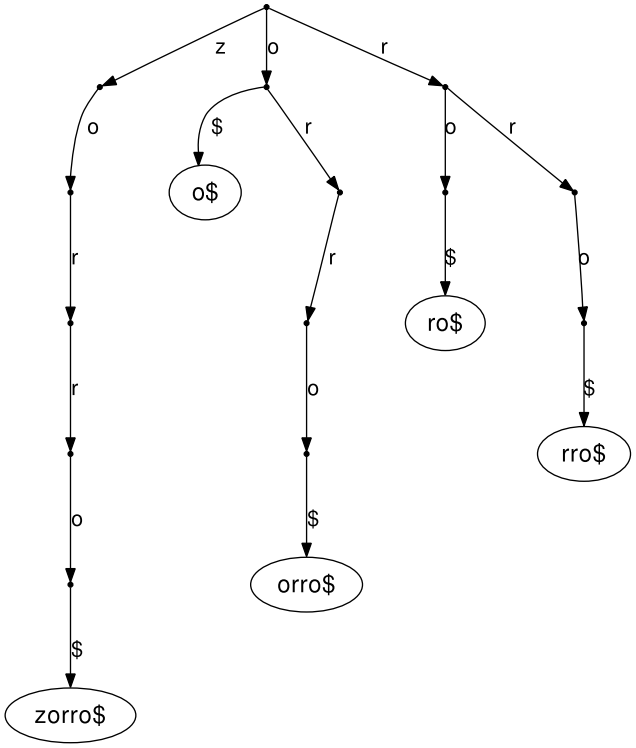
2.2 Struktury danych

Drzewo trie (ang. *trie*) [13] jest typem drzewa przeszukiwań, w którym węzłom nie odpowiadają klucze, lecz ich fragmenty. W dalszej części pracy rozpatrywane będą drzewa, których kluczami są ciągi symboli z pewnego alfabetu.[1] W tego typu drzewach krawędzie etykietowane są symbolami tego alfabetu, a „wartość" klucza danego węzła wynika z jego pozycji i jest konkatenacją etykiet krawędzi leżących na ścieżce prowadzącej od korzenia drzewa do tego węzła. Wszystkie podwęzły danego węzła mają wspólny prefiks równy wartości klucza ich rodzica. Nieco inny wariant takiego drzewa, nazwany *drzewem skompresowanym* (ang. *radix tree, Patricia trie*) [13], polega na tym, że węzły posiadające tylko jednego potomka są z nimi łączone, a symbole z usuniętych krawędzi są konkatenowane (zob. rys. 2.1).

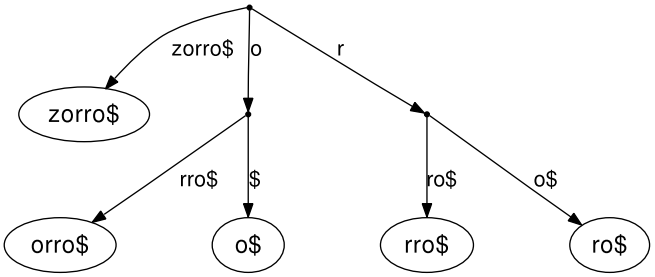
Drzewo sufiksów (ang. *suffix tree*) [7] dla ciągu symboli x z alfabetu Σ jest skompresowanym drzewem zbudowanym na zbiorze wszystkich sufiksów x o następujących cechach:

- krawędzie są etykietowane niepustymi ciągami symboli,
- każdy sufiks jest reprezentowany w drzewie jako ścieżka od korzenia do liścia,

[1] W nomenklaturze polskiej zarówno *tree*, jak i *trie* nazywane są drzewami [12], str. 528. O ile nie będzie powiedziane inaczej, przez drzewo będziemy rozumieli drzewo *trie*.



Rysunek 2.1: Drzewo *trie* dla zbioru sufiksów wyrazu „zorro” z dodanym unikalnym symbolem końcowym \$. Dla przejrzystości pominięto łuk od korzenia do liścia o wartości \$.



Rysunek 2.2: Drzewo sufiksów wyrazu „zorro\$”. Dla przejrzystości pominięto łuk od korzenia do liścia o wartości \$.

- wszystkie węzły wewnętrzne drzewa posiadają co najmniej dwóch potomków.

Zwyczajowo na końcu ciągu x umieszczany jest znak specjalny \$, leksykograficznie mniejszy od wszystkich symboli z alfabetu Σ. Dzięki temu zabiegowi żaden sufiks ciągu x nie będzie prefiksem innego sufiksu, co zapewnia zachowanie wszystkich wyżej wymienionych własności (w przeciwnym przypadku sufiksy mogłyby kończyć się w węzłach wewnętrznych drzewa). Rysunek 2.2 prezentuje drzewo sufiksów dla sekwencji „zorro\$”. Trzy „klasyczne” algorytmy tworzenia drzew sufiksów omówione zostały w publikacjach [27], [19] i [26]. Ich porównanie można znaleźć w pracy [6].

W literaturze powszechne jest etykietowanie sufiksów pozycją ich pierwszego symbolu w słowie x – przez sufiks o etykiecie i rozumie się podciąg $x[i..n-1]$. *Tablica sufiksów* (ang. *suffix array*) [23] słowa x jest tablicą etykiet sufiksów uporządkowaną rosnąco według porządku leksykograficznego sufiksów. Tablicę sufiksów słowa x oznacza się SA_x lub SA jeżeli pominięcie identyfikatora słowa nie wprowadza niejednoznaczności. Formalnie, $SA[j] = i \iff x[i..n-1]$ jest j-tym sufiksem słowa x według porządku leksykograficznego. Warto zauważyć, że SA jest zawsze permutacją liczb $0..n-1$. Rysunki 2.3 oraz 2.4

j	$SA[j] = i$	$x[i..n-1]$
0	5	$
1	4	o$
2	1	orro$
3	3	ro$
4	2	rro$
5	0	zorro$

Rysunek 2.3: Tablica sufiksów dla sufiksów ciągu „zorro$”.

	0	1	2	3	4	5	
$x = [$	z	o	r	r	o	$	]
$SA = [$	5	4	1	3	2	0	]
$ISA = [$	5	2	4	3	1	0	]
$lcp = [$	-	0	1	0	1	0	]

Rysunek 2.4: Tablica sufiksów *SA*, odwrotna tablica sufiksów *ISA* oraz tablica najdłuższych prefiksów *lcp* dla sekwencji „zorro$”.

prezentują tablicę sufiksów sekwencji symboli „zorro$”.

Dla tablicy sufiksów można zbudować *tablicę najdłuższych wspólnych podciągów* (ang. *longest common prefix array*) [23] o długości n, której elementy $lcp[i], i = 1..n-1$ oznaczają długość najdłuższego wspólnego prefiksu sufiksów $SA[i]$ i $SA[i-1]$. Przykładowa tablica najdłuższych wspólnych prefiksów zaprezentowana jest na rysunku 2.4. Tablicę *lcp* można obliczyć w czasie liniowym znając *SA* zgodnie z metodami omówionymi w artykułach [11] i [16].

Strukturą komplementarną do tablicy sufiksów *SA* jest *odwrotna tablica sufiksów* (ang. *inverse suffix array*) [23] oznaczana jako *ISA*. Jest ona permutacją liczb $0..n-1$ spełniającą zależność: $ISA[i] = j \iff SA[j] = i$. Przykład odwrotnej tablicy sufiksów znajduje się na rysunku 2.4. Wartość k-tego wpisu w tablicy *SA* oznacza identyfikator sufiksu na k-tej pozycji w porządku leksykograficznym; k-ty wpis w tablicy *ISA* oznacza pozycję sufiksu k w tablicy *SA* (oraz w porządku leksykograficznym).

Przez *h-grupę* (ang. *h-group*) [23] rozumie się podzbiór sufiksów słowa x o wspólnym prefiksie długości $h > 0$. Podział sufiksów na *h-grupy* otrzymuje się poprzez ich częściowe uporządkowanie ze względu na wartość h pierwszych symboli sufiksu. Proces ten nosi nazwę *h-sortowania* (ang. *h-sort*), w jego wyniku sufiksy uporządkowane są według *h-porządku* (ang. *h-order*). Sufiksy należące do jednej *h-grupy* są sobie równe pod względem *h-porządku*. Algorytmy *h-sortowania* są zazwyczaj stabilne, czyli zachowują wcześniejszy porządek sufiksów. Każdej *h-grupie* można przypisać pewien identyfikator (ang. *h-rank*). W zależności od potrzeb algorytmu, wartości identyfikatorów dobierane są na jeden z trzech sposobów:

1. grupa identyfikowana jest pozycją pierwszego jej elementu w przybliżonej tablicy sufiksów – głowa (ang. *head*) grupy,

2. grupa identyfikowana jest pozycją ostatniego elementu w przybliżonej tablicy sufiksów – ogon (ang. *tail*) grupy,

3. każdej z *h-grup* (nawet jednoelementowym) przypisywane są rosnące identyfikatory zgodnie z kolejnością ich pojawienia się w SA_h.

Wyniki *h-sortowania* zachowuje się w *przybliżonej tablicy sufiksów* (ang. *approximate suffix array*) [23] oznaczanej SA_h. Możliwe jest również wyznaczenie *przybliżonej odwrotnej tablicy sufiksów* (ang. *ap-*

	0	1	2	3	4	5	6	7	8	9	10	11	
x = [	a	b	e	a	c	a	d	a	b	e	a	$	]
SA_1 = [	11	(0	3	5	7	10)	(1	8)	4	6	(3	9)	]
ISA_1 = [	1	6	10	1	8	1	9	1	6	10	1	0	]
lub [	5	7	11	5	8	5	9	5	7	11	5	0	]
lub [	1	3	5	1	3	1	4	1	2	5	1	0	]

Rysunek 2.5: Przykłady przybliżonej tablicy sufiksów oraz przybliżonej odwrotnej tablicy sufiksów dla $h = 1$. *H-grupy* wyróżnione zostały nawiasami. Na rysunku przedstawione są 3 wersje tablicy ISA_1 różniące się metodami przypisywania identyfikatorów grupom (wypisane w kolejności przedstawienia w tekście). Źródło: [23].

proximate inverse suffix array) [23] ISA_h. SA_h jest permutacją liczb $0..n-1$, natomiast ISA_h może zawierać wartości powtarzające się. Wynika to z tego, że każdemu sufiksowi należącemu do danej *h-grupy* przypisywany jest jej identyfikator. Przykład przybliżonej tablicy sufiksów znajduje się na rysunku 2.5.

W niektórych publikacjach przyjęto inną konwencję nazewniczą, *h-grupa* nazywana jest tam *h-kubełkiem* lub po prostu *kubełkiem* (ang. *bucket*). Tablica ISA_h nazywana jest *tablicą wskaźników na kubełki* (ang. *bucket pointer array*) [24].

Rozdział 3

Wybrane metody tworzenia tablic sufiksów

W poniższym rozdziale omówione są wybrane metody tworzenia tablic sufiksów. Algorytmy wybrane zostały na podstawie wyników testów wydajnościowych zaprezentowanych w pracach [23] i [24] oraz publikowanych na stronach [F] i [E]. Algorytmy uzyskujące dobre wyniki na dużych zbiorach danych – szybkie *w praktyce* pochodzą z rodziny algorytmów wykorzystujących *polepszanie wszerz*. W poniższym zestawieniu opisane zostały również algorytmy reprezentujące inne grupy metod tworzenia tablic sufiksów: *polepszanie wgłąb* i *sortowanie zredukowanych ciągów znaków*.

3.1 Algorytm *skew*

Algorytm *skew* został opracowany przez Petera Sandersa i Juhę Kärkkäinena [9]. Dostępna jest również implementacja tego algorytmu ich autorstwa [G]. Opisywana metoda pochodzi z rodziny algorytmów wykorzystujących *sortowanie zredukowanych ciągów znaków*. Algorytm wykonuje 3 główne kroki:

1. Tworzenie tablicy sufiksów zbudowanej z sufiksów S_i o indeksach $i \bmod 3 \neq 0$. Problem jest redukowany do tworzenia tablicy sufiksów ciągu długości 2/3 rozmiaru ciągu wejściowego, a następnie rozwiązywany rekurencyjne.

2. Tworzenie tablicy sufiksów zbudowanej z sufiksów pominiętych w pierwszym kroku z wykorzystaniem tablic sufiksów z kroku 1.

3. Połączenie zbudowanych tablic w jedną.

Pierwszy (i najbardziej czasochłonny) krok algorytmu polega na posortowaniu sufiksów S_i dla $i \bmod 3 \neq 0$, czyli utworzeniu tablicy sufiksów SA^{12}. Jeżeli w wyniku *h-sortowania* dla $h = 3$ wszystkie grupy są jednoelementowe, to ten etap algorytmu się kończy. W przeciwnym przypadku każdemu sufiksowi S_i nadawany jest identyfikator $x_i^{'} \in [1, 2n/3]$ będący identyfikatorem jego *3-grupy* powstałej po *3-sortowaniu*. Następnie tworzony jest ciąg $x^{12} = [x_i^{'} : i \bmod 3 = 1] \circ [x_i^{'} : i \bmod 3 = 2]$, którego tablica sufiksów $SA_{x^{12}}$ obliczana jest rekurencyjnie. Tablica SA^{12} wypełniana jest zgodnie z poniższym wzorem (n_1 oznacza liczbę sufiksów o etykietach i takich, że $i \bmod 3 = 1$):

$$SA^{12}[i] = \begin{cases} 1 + 3k & \textit{jeżeli } k = SA_{x^{12}}[i] < n_1, \\ 2 + 3(k - n_1) & \textit{w przeciwnym wypadku.} \end{cases}$$

Drugi krok algorytmu polega na utworzeniu tablicy sufiksów SA^0 złożonej z sufiksów S_i, gdzie $i \bmod 3 = 0$. Tablica ta powstaje w wyniku sortowania par $(x[i], S_{i+1})$. Ponieważ porządek sufiksów

	0	1	2	3	4	5	6	7	8	9	10	11	
x = [	b	a	d	d	a	d	d	a	c	c	a	$	]
$S_i, i \bmod 3 = 0$ = [	0			3			6			9			]
$S_i, i \bmod 3 \neq 0$ = [		1	2		4	5		7	8		10	11	]
$x[i..i+2], i \bmod 3 \neq 0$ = [		add	dda		add	dda		acc	cca		a$-	$–	]

3-sortowanie sufiksów o etykietach $i \bmod 3 \neq 0$ (wartości są indeksami po posortowaniu).

x' = [		3	5		3	5		2	4		1	0	]

Tworzenie ciągu $x^{12} = [x'_i : i \bmod 3 = 1] \circ [x'_i : i \bmod 3 = 2]$.

x^{12} = [	3	3	2	1	5	5	4	0	]

Obliczanie $SA_{x^{12}}$ i SA^{12}.

$SA_{x^{12}}$ = [	7	3	2	1	0	6	5	4	]
$ISA_{x^{12}}$ = [	4	3	2	1	7	6	5	0	]
SA^{12} = [	11	10	7	4	1	8	5	2	]

Sufiksy S_i, $i \bmod 3 = 1$ oraz S_j, $j = i - 1$ zgodnie z kolejnością w SA^{12}.

[	10	7	4	1	]
[	9	6	3	0	]
$x[j]$ = [	c	d	d	b	]

Stabilne *1-sortowanie* sufiksów S_j.

SA^0 = [	0	9	6	3	]

Scalanie tablic SA^0 i SA^{12}.

SA = [	11	10	7	4	1	0	9	8	6	3	5	2	]

Rysunek 3.1: Przykład tworzenia tablicy sufiksów według algorytmu *skew*. Źródło: opracowanie własne.

S_{i+1} jest zawarty w SA^{12}, do znalezienia SA^0 wystarczy posortować stabilnie elementy $SA^{12}[j]$ reprezentujące sufiksy $SA_{i+1}, i \bmod 3 = 0$ według wartości $x[i]$. Można tego dokonać w czasie liniowym jednym krokiem sortowania kubełkowego.

Trzeci krok algorytmu polega na połączeniu tablic SA^{12} i SA^0 w jedną tablicę sufiksów. Porównywanie sufiksów S_j, gdzie $j \bmod 3 = 0$ i $S_i, i \bmod 3 \neq 0$ odbywa się na jeden z dwóch sposobów:

- Jeżeli $i \bmod 3 = 1$, to porządek sufiksów S_i i S_j można ustalić porównując pary $(x[i], S_{i+1})$ oraz $(x[j], S_{j+1})$. Ponieważ $i + 1 \bmod 3 = 2$ i $j + 1 \bmod 3 = 1$ porządek sufiksów S_{i+1} i S_{j+1} można ustalić na podstawie ich pozycji w $SA_{x^{12}}$. Ta pozycja może być ustalana w czasie stałym, jeżeli obliczona zostanie tablica $ISA_{x^{12}}$.
- Analogicznie, jeżeli $i \bmod 3 = 2$, to porównywane są trójki $(x[i], x[i+1], S_{i+2})$ i $(x[j], x[j+1], S_{j+2})$, przy czym sufiksy S_{i+2} i S_{j+2} są zastępowane odpowiednimi wpisami z tablicy $ISA_{x^{12}}$.

3.2 Algorytm *qsufsort*

Algorytm *qsufsort* został przedstawiony w pracy [14] autorstwa Jespera Larrsona i Kunihiko Sadakane. Autorzy pracę udostępniają również implementację w języku `C` [L]. Opisywana metoda należy do rodziny algorytmów wykonujących polepszanie wgłąb, opiera się na algorytmie *prefix-doubling*.

Sortowanie sufiksów wykonywane jest algorytmem *ternary split quicksort* [1]. Algorytm *qsufsort* wykorzystuje również twierdzenie opublikowane w pracy [10]: dla obliczonych SA_h i ISA_h posortowanie sufiksów S_i według par $(ISA_h[i], ISA_h[i+h])$, $i + h \leq n$ tworzy *2h-porządek* sufiksów, czyli porządek według prefiksów długości $2h$.[1] Wynika to z tego, że do ustalenia porządku sufiksów o wspólnym pre-

[1] Sufiksy $S_i, i > n - h$ są zawsze w pełni uporządkowane.

	0	1	2	3	4	5	6	7	8	9	10	11	
x = [	a	b	e	a	c	a	d	a	b	e	a	$	]
SA_1 = [	11	(0	3	5	7	10)	(1	8)	4	6	(2	9)	]
ISA_1 = [	5	7	11	5	8	5	9	5	7	11	5	0	]
L = [	-1	5					2		-2		2		]
SA_2 = [	11	10	(0	7)	3	5	(1	8)	4	6	(2	9)	]
ISA_2 = [	3	7	11	4	8	5	9	3	7	11	2	0	]
L = [	-2		2		-2		2		-2		2		]
SA_4 = [	11	10	(0	7)	3	5	8	1	4	6	9	2	]
ISA_4 = [	3	7	11	4	8	5	9	3	6	10	1	0	]
L = [	-2		2		-8								]
SA_8 = [	11	10	7	0	3	5	8	1	4	6	9	2	]
ISA_8 = [	3	7	11	4	8	5	9	2	6	10	1	0	]
L = [	-12												]

Rysunek 3.2: Przebieg algorytmu *qsufsort* dla słowa „abeacadabea". Źródło: opracowanie własne na podstawie [23].

fiksie długości h wykorzystujemy porządek ich *h-następców*, którzy są również posortowani według h pierwszych znaków, co daje w konsekwencji porządek sufiksów według prefiksów długości $2h$.

Na początku działania algorytm wykonuje *1-sort*, w wyniku którego powstają SA_1 i ISA_1. Numery grup w odwróconej tablicy sufiksów przypisywane są poprzez wybór ogona (pozycji ostatniego sufiksu w tablicy SA_h) danej grupy. Algorytm *qsufsort* utrzymuje również tablicę L długości n używaną w celu określania rozmiarów grup. Wartość $L[j] = d$ oznacza, że grupa rozpoczynająca się na pozycji j ma d elementów. Wartości ujemne w tablicy L oznaczają ciągi jednoelementowych grup (na przykład -2 oznacza dwie jednoelementowe grupy).

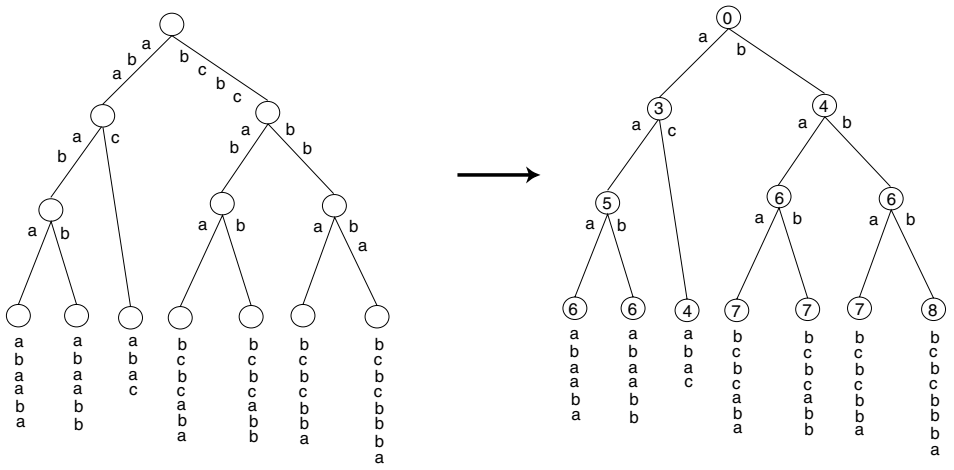
Przeglądanie tablicy L od lewej do prawej umożliwia omijanie grup jednoelementowych w procesie polepszania kubełków. Każda *h-grupa* sortowana jest osobno, jej sufiksy S_i porównywane są na podstawie wartości $ISA_h[i+h]$. Po posortowaniu wszystkich grup otrzymujemy *2h-porządek*. Algorytm kończy swoje działanie, gdy wszystkie h-grupy są jednoelementowe, czyli gdy $L[0] = -n$. W przeciwnym wypadku h jest podwajane, a proces polepszania kubełków wykonywany po raz kolejny.

Algorytm *qsufsort* da się optymalizować pod kątem redukcji zużycia pamięci. Możliwe jest kompletne wyeliminowanie tablicy L. Do wyznaczania *h-grup* zawierających więcej niż 1 element można wykorzystać tablicę *ISA*, rozmiar grupy wyznacza się wtedy na podstawie jej pierwszego elementu. Jeżeli j oznacza pozycję pierwszego elementu grupy w SA, a $i = SA_h[j]$, to jej rozmiar wynosi $ISA_h[i] - j + 1$. Złożoność pamięciowa algorytmu może również zostać zredukowana poprzez nadpisanie ciągu wejściowego i wykorzystanie tego obszaru pamięci do przechowywania *ISA* po obliczeniu SA_1.

3.3 Algorytm *deep shallow*

Algorytm *deep shallow* jest rozwinięciem algorytmu *copy* opracowanym przez Paolo Ferraginę i Giovanniego Manziniego. Opublikowany został w pracy [17]. Kod algorytmu w języku `C` dostępny jest pod adresem [J], jego autorem jest Giovanni Manzini.

Pierwszym krokiem algorytmu jest uporządkowanie sufiksów według dwóch pierwszych znaków (*2-sortowanie*). Każdy z utworzonych w ten sposób kubełków sortowany jest algorytmem *multikey quicksort* [2], który jest przerywany po osiągnięciu poziomu rekurencji równemu L, czyli jeżeli wewnątrz kubełka istnieje grupa sufiksów o wspólnym prefiksie długości L. Podejście to nosi nazwę sortowania płytkiego (ang. *shallow sorting*).



Rysunek 3.3: Drzewo skompresowane oraz odpowiadające mu drzewo *blind trie* zbudowane na zbiorze sekwencji *abaaba, abaabb, abac, bcbcaba, bcbcabb, bcbcbba, bcbcbbba*. Źródło: [17].

Sortowanie znajdowanych sufiksów o wspólnym prefiksie długości L nazwane zostało przez autorów algorytmu *deep shallow* sortowaniem głębokim (ang. *deep sort*). Przebieg działania głębokiego sortowania zależy od wielkości zbioru sortowanych sufiksów. Jeżeli liczność zbioru nie przekracza zadanej wartości B, to sufiksy są sortowane algorytmem *blind sort*. W przeciwnym przypadku do posortowania sufiksów wykorzystane zostanie zmodyfikowany algorytm *ternary split quicksort* [1]. Autorzy sugerują użycie wartości $B = n \times 0.0005$ jako progu wielkości zbioru.

Algorytm *blind sort* opiera swoje działanie na strukturze danych nazywanej *blind trie* [4], która jest typem skompresowanego drzewa którego węzły wewnętrzne przechowują liczby określające długość wspólnego prefiksu węzłów potomnych (jeżeli węzeł zawiera liczbę k, to jego węzły potomne różnią się na pozycji $k+1$). Przykładowe drzewo *blind trie* znajduje się na rysunku 3.3.

Algorytm *blind sort* tworzy drzewo *blind trie*, a następnie przegląda je od lewej do prawej uzyskując w ten sposób porządek leksykograficzny sekwencji podanych na wejściu (w drzewie *blind trie* węzły potomne danego węzła są uporządkowane). Poważną wadą metody *blind sort* jest jej złożoność pamięciowa, sięgająca nawet $36m$, gdzie m oznacza liczbę ciągów do posortowania.

Do sortowania zbiorów większych niż B autorzy algorytmu *deep shallow* użyli zmodyfikowanego algorytmu *ternary split quicksort*, opisanego w pracy [1]. Dokonali oni następujących zmian:

1. Jeżeli na dowolnym etapie rekursji zbiór sufiksów jest mniejszy niż B, to do jego sortowania użyty zostaje algorytm *blind sort*.

2. Podczas fazy podziału sufiksów obliczane są L_S i L_L, czyli długość najdłuższego wspólnego prefiksu elementu osiowego (ang. *pivot*) i zbioru elementów mniejszych (L_S) oraz większych (L_L) od elementu osiowego. Dzięki temu podczas sortowania sufiksów w tych zbiorach można pomijać prefiksy długości L_S lub L_L.

Paolo Ferragina i Giovanni Manzini w pracy [17] wymieniają trzy zalety dwuetapowego podejścia do problemu sortowania sufiksów zastosowanego w algorytmie *deep shallow*:

1. Szybkie wykrywanie grup sufiksów o długim wspólnym prefiksie.

2. Rozmiar stosu użytego podczas rekurencji w fazie sortowania płytkiego jest ograniczony parametrem L i nie zależy od rozmiaru wejścia.

3. Jeżeli długość najdłuższego wspólnego prefiksu sufiksów jednego kubełka nie przekracza L, to uporządkowywane są one efektywnym algorytmem sortowania ciągu znaków (*multikey quicksort*).

3.4 Algorytm *two-stage*

Hideo Itoh i Hozumi Tanaka zaproponowali w pracy [8] algorytm tworzenia tablic sufiksów o nazwie *two-stage*. Algorytm ten dzieli sufiksy na dwie kategorie: sufiksy typu A to wszystkie sufiksy S_i spełniające nierówność $x[i] > x[i+1]$, sufiksy typu B to sufiksy spełniające nierówność $x[i] \leq x[i+1]$.

Algorytm rozpoczyna swoje działanie od obliczenia SA_1. Początek i koniec każdej grupy zapamiętywany jest w tablicach *head* i *tail*. Długość tych tablic odpowiada wielkości słownika symboli. Sufiksy w każdej *1-grupie* są uporządkowywane w taki sposób, żeby sufiksy typu A były przed sufiksami typu B. Wynika to z tego, że jeżeli sufiksy S_i typu A i S_j typu B mają wspólny prefiks długości jednego symbolu, to sufiks S_i poprzedza sufiks S_j w porządku leksykograficznym. Indeksy pierwszych sufiksów typu B każdej grupy zapamiętywane są w tablicy *part* długości σ. Kolejnym krokiem algorytmu jest sortowanie sufiksów typu B wewnątrz każdej grupy. Autorzy pracy [8] sugerują użycie do tego celu algorytmu [2], nie jest to jednak konieczne i można do tego celu użyć innego algorytmu sortowania ciągów znaków.

Następnie ustalany jest porządek sufiksów typu A. Tablica SA przeglądana jest od lewej do prawej, znajdując wartości $i = SA[j], j = 0, 1, \ldots, n-1$. Jeżeli S_{i-1} jest jeszcze nieuporządkowanym sufiksem typu A, to umieszczany jest na początku grupy do której należy. Odpowiednia wartość w tablicy *head* jest potem inkrementowana. Po jednokrotnym przejrzeniu SA sufiksy są już w pełni posortowane, co kończy algorytm (przykład na rysunku 3.4).

	0	1	2	3	4	5	6	7	8	9	10	11	
x = [	b	a	d	d	a	d	d	a	c	c	a	$	]
type = [	A	B	B	A	B	B	A	B	B	A	A	B	]
						1-sort							
SA = [	11	(–	1	4	7)	(–)	(–	8)	(–	–	2	5)	]
type = [	B	A	B	B	B	A	A	B	A	A	B	B	]
						sortowanie sufiksów typu B							
SA = [	11	(–	7	4	1)	(–)	(–	8)	(–	–	5	2)	]
						wstawianie sufiksów typu A							
SA = [	11	10	7	4	1	–	–	8	–	–	5	2	]
SA = [	11	10	7	4	1	–	9	8	–	–	5	2	]
SA = [	11	10	7	4	1	–	9	8	6	–	5	2	]
SA = [	11	10	7	4	1	–	9	8	6	3	5	2	]
SA = [	11	10	7	4	1	0	9	8	6	3	5	2	]

Rysunek 3.4: Przebieg algorytmu *two-stage* dla słowa „baddaddacca”. Nieuporządkowane sufiksy typu A oznaczane są znakiem „–”. Źródło: opracowanie własne na podstawie [23].

3.5 Algorytm *improved two-stage*

Algorytm *improved two-stage* opisany został w pracy [15], której autorami są Simon Puglisi i Michael Maniscalco. Kwestia autorstwa tego algorytmu pozostaje niejasna, metoda ta posiada aż trzy niezależne implementacje powstałe przed publikacją artykułu:

- *archon* [I] autorstwa Dimy Małyszewa,

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
x = [	e	d	a	b	d	c	c	d	e	e	d	a	b	$	]
type = [	A	A	B	B^*	A	B	B	B^*	A	A	A	B^*	A		]
						Obliczanie wielkości *2-grup*									
SA_2 = [	–	(–	–)	–	–	–	–	(–	–)	–	–	(–	–)	–	]
						Sortowanie sufiksów typu B^*									
SA = [	–	11	–	–	3	–	–	–	–	–	7	–	–	–	]
						Wstawianie sufiksów typu B									
SA = [	–	11	–	–	3	–	6	–	–	–	7	–	–	–	]
SA = [	–	11	–	–	3	5	6	–	–	–	7	–	–	–	]
SA = [	–	11	2	–	3	5	6	–	–	–	7	–	–	–	]
						Wstawianie sufiksów typu A									
SA = [	13	11	2	–	3	5	6	–	–	–	7	–	–	–	]
SA = [	13	11	2	12	3	5	6	–	–	–	7	–	–	–	]
SA = [	13	11	2	12	3	5	6	10	–	–	7	–	–	–	]
SA = [	13	11	2	12	3	5	6	10	1	–	7	–	–	–	]
SA = [	13	11	2	12	3	5	6	10	1	4	7	–	–	–	]
SA = [	13	11	2	12	3	5	6	10	1	4	7	9	–	–	]
SA = [	13	11	2	12	3	5	6	10	1	4	7	9	0	–	]
SA = [	13	11	2	12	3	5	6	10	1	4	7	9	0	8	]

Rysunek 3.5: Przebieg algorytmu *divsufsort* dla słowa „edabdccdeedab”. Źródło: opracowanie własne na podstawie [20].

- *divsufsort* [K] której autorem jest Yuta Mori,
- *msufsort* [F] autorstwa Michaela Maniscalco.

Wymienione implementacje algorytmu *improved two-stage* napisane zostały w języku `C++`.

Na początku sufiksy są dzielone na dwie kategorie: sufiksy typu A to sufiksy S_i spełniające nierówność $S_i > S_{i+1}$, sufiksy typu B to sufiksy spełniające nierówność $S_i \leq S_{i+1}$. Sufiks o identyfikatorze $n-1$ należy do obu grup. Następnie, sufiksy S_i typu B których następny sufiks S_{i+1} jest typu A oznaczane są jako sufiksy typu B^*.

Kolejnym krokiem algorytmu jest znalezienie granic grup które powstałyby po *2-sortowaniu*. Następnie tworzony jest *2-porządek* sufiksów typu B^*. *2-sortowanie* wszystkich sufiksów nie jest konieczne w tym momencie, a jego pominięcie zmniejsza czas działania algorytmu. Sufiksy typu B^* należące do jednej grupy są sortowane zgodnie z metodami przedstawionymi w pracach [2, 17] i umieszczane na początkowych pozycjach wewnątrz kubełka w tablicy SA.

Porządek sufiksów typu B ustalany jest poprzez przeglądanie tablicy SA od prawej do lewej. Dla każdego sufiksu $SA[i]$ odczytanego podczas przeglądania tablicy sufiks $SA[i]-1$ jest wstawiany na ostatnią pustą pozycję wewnątrz swojej grupy jeżeli jest sufiksem typu B.

Wstawianie sufiksów typu A wykonywane jest w sposób identyczny jak w algorytmie *two-stage* (przykład przedstawiono na rysunku 3.5).

3.6 Algorytm *bpr*

Klaus-Bernd Schürmann w pracy [24] zaproponował algorytm *bpr* (*bucket pointer refinement*, polepszanie wskaźników na kubełki). Algorytm ten jest poprawioną wersją metody przedstawionej w pracy [25] napisanej przez Klausa-Bernda Schürmanna i Jensa Stoye. Kod algorytmu *bpr* w języku `C++` dostępny jest na stronie [M].

	0	1	2	3	4	5	6	7	8	
x = [	D	E	B	D	E	B	D	E	A	]
	Podział na kubełki, $q = 2$									
	A	BD			DE		EA	EB		
SA = [	8	**(2**	**5)**	(0	3	6)	7	(1	4)	]
ISA = [	5	8	2	5	**8**	2	5	**6**	0	]
	Po posortowaniu kubełka BD									
SA = [	8	5	2	**(0**	**3**	**6)**	7	(1	4)	]
ISA = [	5	8	**2**	5	8	**1**	5	6	**0**	]
	Po posortowaniu kubełka DE									
SA = [	8	5	2	6	3	0	7	**(1**	**4)**	]
ISA = [	5	8	2	**4**	8	1	**3**	6	0	]
	Po posortowaniu kubełka EB, koniec algorytmu									
SA = [	8	5	2	6	3	0	7	4	1	]
ISA = [	5	8	2	4	7	1	3	6	0	]

Rysunek 3.6: Przebieg algorytmu *bpr* dla słowa „DEBDEBDEA”. Elementy wyróżnione w tablicy SA przeznaczone do uporządkowania w danym kroku, elementy wyróżnione w tablicy ISA to klucze wykorzystywane do sortowania kubełka. Źródło: opracowanie własne na podstawie [24].

Pierwszym krokiem algorytmu jest posortowanie sufiksów według q-pierwszych znaków, czyli *q-sortowanie*. Wartość parametru q obliczana jest na podstawie liczby różnych symboli (czyli wielkości alfabetu) w ciągu wejściowym, jej wartość maleje wraz ze wzrostem wielkości słownika sekwencji wejściowej. Następnie obliczana jest przybliżona odwrotna tablica sufiksów ISA (kubełki identyfikowane są pozycją ich ostatniego elementu w przybliżonej tablicy sufiksów), nazywana w pracy [24] *tablicą wskaźników na kubełki*.

Algorytm *bpr* działa według schematu *polepszania kubełków wgłąb* i korzysta z techniki *pull*. Sufiksy S_i i S_j należące do jednej *h-grupy* porównywane są poprzez porównanie wartości $ISA[h+i]$ i $ISA[h+j]$.

Każda ze znalezionych na początku grup sortowana jest algorytmem *ternary split quicksort* [1]. Parametr h przyjmuje początkowo wartość q. Algorytm ten wybiera jeden z sufiksów danej grupy jako element osiowy (ang. *pivot*) S_p o wartości klucza $K_p = ISA[h+p]$. Sufiksy danej grupy dzielone są na trzy podgrupy: sufiksy o wartości klucza mniejszej, równej lub większej niż K_p. Każda z tych podgrup jest następnie w ten sam sposób sortowana, przy czym dla grupy sufiksów o wartości klucza równej elementowi osiowemu wartość parametru h zwiększana jest o q, ponieważ wszystkie jej sufiksy mają wspólny prefiks długości $h+q$. Wynika to z tego, że elementy kubełka o wspólnym prefiksie długości h sortowane są na podstawie pozycji ich *h-następników*, dla których nie znamy długości wspólnego prefiksu. Wiadomo tylko tyle, że były posortowane na początku działania algorytmu według q pierwszych znaków. Podobnie jak w algorytmie *qsufsort*, równa wartość klucza $ISA[h+i]$ i $ISA[h+j]$ oznacza że sufiksy S_i i S_j mają wspólny prefiks długości $h+q$. Fundamentalna różnica między algorytmami *bpr* i *qsufsort* polega na tym, że pierwszy z nich realizuje schemat *polepszania kubełków wgłąb*, a drugi *polepszania wszerz*. Dzięki temu, algorytm *qsufsort* w kolejnej iteracji może podwajać wartość parametru h.

Istotną cechą algorytmu *bpr* jest to, że po każdym podziale grupy na podgrupy wykonywana jest aktualizacja wpisów w tablicy ISA odpowiadających jej elementom, zgodnie ze wzorem $ISA[i] = $ *pozycja ostatniego elementu podgrupy*. Na rysunku 3.6 przedstawiony został przebieg działania algorytmu *bpr* na przykładzie ciągu „DEBDEBDEA”.

Rozdział 4

Zaimplementowane algorytmy

Zgodnie z założeniami w niniejszej pracy zaimplementowane zostały rozwiązania dla następujących etapów procesu rozpoznawania tęczówki:

- przetwarzanie wstępne,
- normalizacja,
- kodowanie,
- dopasowanie.

Algorytmy segmentacji tęczówki oraz usuwania zakłóceń zostały pominięte, ponieważ były przedmiotem innej pracy dyplomowej. W dalszej części rozdziału opisane zostaną zaimplementowane algorytmy dla wyżej wymienionych etapów.

4.1 Przetwarzanie wstępne

Odpowiednie przygotowanie obrazu przed rozpoczęciem procesu rozpozanawania jest bardzo ważne. Pozwala ono na polepszenie jakości zdjęcia przez usunięcie zakłóceń czy poprawienie kontrastu, co znacznie wpływa na jakość działania całego systemu.

Różne metody segmentacji wymagają odmiennego wstępnego przygotowania obrazu. Często proces przygotowania obrazu różni się nawet dla poszczególnych etapów procesu segmentacji, tzn. innych operacji może wymagać obraz w trakcie szukania parametrów źrenicy, a jeszcze innego w trakcie szukania parametrów tęczówki. Z tego względu przetwarzanie wstępne może byc dość skomplikowane i składac się z wielu różnorodnych kroków.

Ponieważ algorytmy segmentacji często polegają na konkretnych operacjach przygotowawczych, których może być wiele, umożliwienie użytkownikowi ich parametryzacji bardzo skomplikowałoby interfejs aplikacji i znacznie pogorszyło by jego czytelność, jednocześnie dając użytkownikowi niewiele pola do manipulacji tą częścią procesu.

Ze względu na zależność procesu przygotowania obrazu od wybranego algorytmu segmentacji, implementacja przetwarzania wstępnego przeniesiona została do implementacji poszczególnych metod segmentacji, a użytkownikowi udostępnione zostały wyłącznie uniwersalne algorytmy takie jak:

- filtr Gaussa
- filtr medianowy

- normalizacja histogramu
- filter2D

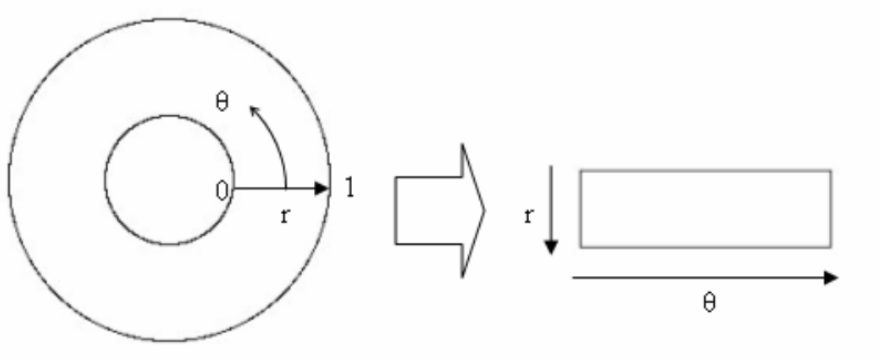
Implementacje tych alogrytmów wykorzystują podstawowe funkcje z biblioteki OpenCV takie jak: `GaussianBlur`, `medianBlur`, `equalizeHist` oraz `filter2D`.

4.2 Normalizacja

Porównywanie tęczówki może być problematyczne, ze względu na potencjalne niezgodności w wymiarach obrazu. Najczęstszą przyczyną tych niespójności jest zmiana rozmiaru tęczówki będąca wynikiem rozszerzania lub kurczenia się źrenicy oka w zależności od intensywności oświetlenia w otoczeniu. Niespójności te mogą wynikać również z innych źródeł np. różna odległość w trakcie pobierania zdjęcia, różnice w obrocie kamery, inna rotacja oka, czy nawet nachylenie głowy w trakcie pobierania zdjęcia [3].

W celu umożliwienia porównania dwóch obrazów, z których każde mogło być zrobione w odmiennych warunkach, obraz poddaje się procesowi normalizacji, który ma za zadanie uspójnić rozmiary tęczówki w systemie.
Ważną informacją dla procesu normalizacji jest także to, że środek źrenicy i tęczówki nie muszą znajdować się w tym samym punkcie, co musi być wzięte pod uwagę przy opracowywaniu rozwiązania.

W aplikacji zdecydowano się na użycie normalizacji metodą Daugmana [3], która polega na przekształceniu obrazu tęczówki z postaci pierścienia do postaci prostokąta przez zmianę układu współrzędnych, w którym prezentowane są wartości poszczególnych punktów tęczówki. Zmiana ta polega na przejściu z opisu wartości pikseli w kartezjańskim układzie współrzędnych (x, y), do opisu za pomocą pary współrzędnych biegunowych (r, θ), gdzie r jest wartością z przedziału $[0, 1]$, a θ jest kątem z przedziału $[0, 2\pi]$.



Rysunek 4.1: Model normalizacji Daugmana [18]

Wybrany rozmiar prostokąta decyduje o ilości informacji znajdujących się w znormalizowanym obrazie. Wiersze w tak uzyskanym prostokącie można interpretować jako kolejne okręgi na obrazie tęczówki. Jak widać na powyższym rysunku 4.1, wybrana wysokość prostokąta decyduje o ilości okręgów - rozdzielczość promieniowa , a szerokość prostokąta decyduje o rozdzielczości kątowej.

Przejście ze współrzędnych kartezjańskich (x, y) do współrzędnych biegunowych (r,θ) opisane zostało przez Daugmana [3] równaniami 4.1:

$$I(x(r,\theta), y(r,\theta)) \rightarrow I(r,\theta),$$

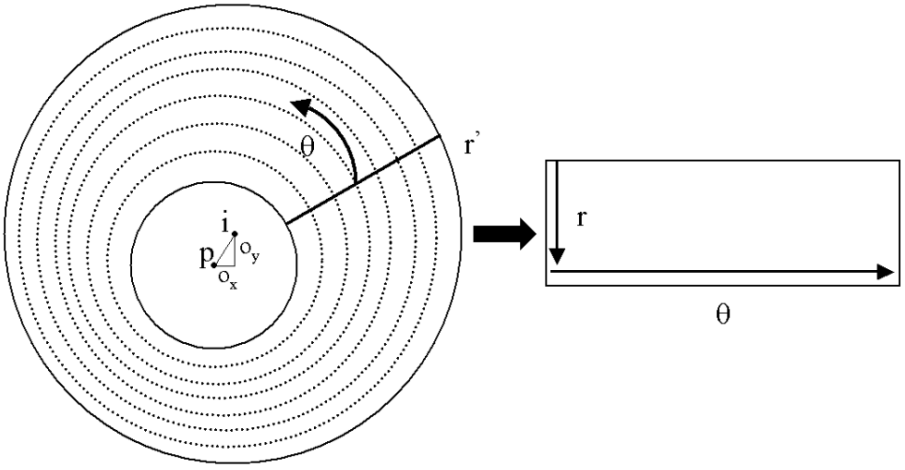
$$\begin{aligned} x(r,\theta) &= (1-r)\,x_p(\theta) + r\,x_s(\theta), \\ y(r,\theta) &= (1-r)\,y_p(\theta) + r\,y_s(\theta), \end{aligned} \tag{4.1}$$

gdzie:

x_p, y_p - współrzędne na okręgu granicznym źrenicy wzdłuż kierunku θ,

x_s, y_s - współrzędne na okręgu granicznym tęczówki wzdłuż kierunku θ.

Tak jak wcześniej wspomniano, środki źrenicy i tęczówki nie zawsze znajdują się na jednej osi. Z tego względu wyznaczając kolejne punkty znormalizowanego obrazu należy uwzględnić różną odległość między granicą źrenicy a tęczówki w zależności od rozpatrywanego kąta θ, co dobrze zobrazowane zostało na rysunku 4.2. Niezależnie od tych różnic, wzdłuż każdego kierunku θ wybierana jest stała liczba punktów w taki sposób, aby otrzymany obraz był prostokątem o stałych wymiarach.



Rysunek 4.2: Model normalizacji Daugmana z uwzględnieniem niewspółosiowości tęczówki i źrenicy [22]

Uwzględnienie tej niewspółosiowości polega na wyznaczeniu wartości odległości między granicami źrenicy i tęczówki dla każdego rozpatrywanego kąta [18] zgodnie z równaniami 4.2.

$$r' = \sqrt{\alpha}\beta \pm \sqrt{\alpha\beta^2 - \alpha - r_I^2},$$

$$\begin{aligned} \alpha &= o_x^2 + o_y^2, \\ \beta &= \cos\left(\pi - \arctan\left(\frac{o_y}{o_x}\right) - \theta\right). \end{aligned} \tag{4.2}$$

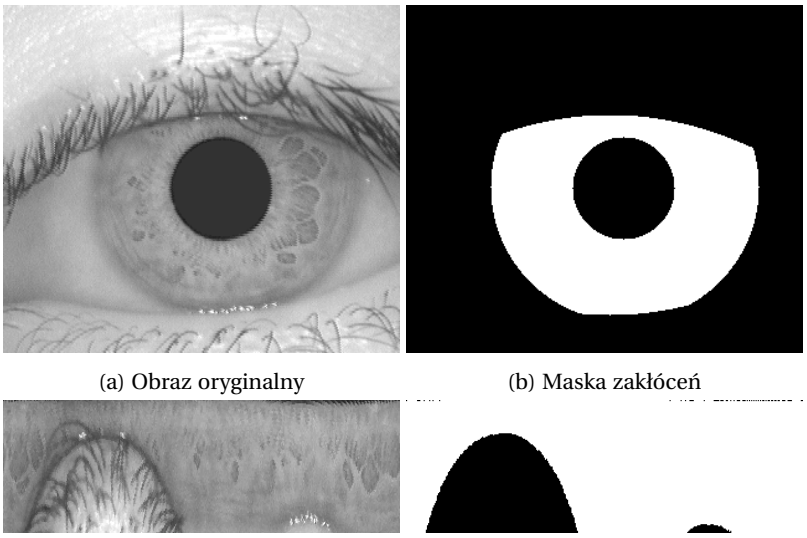
gdzie:

o_x, o_y - różnica położeń między środkiem źrenicy i tęczówki,

r' - odległość między granicą źrenicy, a granicą tęczówki pod kątem θ,

r_I - promień tęczówki

Jeżeli w procesie segmentacji wygenerowana została dodatkowo maska zawierająca informacje o zakłoceniach znajdujących się w obrębie tęczówki, takich jak powieki, czy rzęsy, wygenerowaną maskę również należy poddać procesowi normalizacji, aby umożliwić korzystanie z niej podczas porównywania tęczówek. Przykład procesu normalizacji przedstawiony został na rysunku 4.3.



(a) Obraz oryginalny (b) Maska zakłóceń

(c) Tęczówka po normalizacji (d) Maska zakłóceń po normalizacji

Rysunek 4.3: Przykład normalizacji metodą Daugmana

Takie rozwiązanie zapewnia odporność na rozszerzanie i kurczenie się źrenicy, niewspółosiowość okręgów tęczówki i źrenicy a także różnicę położenia ich między różnymi obrazami. Metoda ta nie kompensuje jednak różnic rotacji tęczówki. Kompensacja ta jest natomiast zapewniana w procesie dopasowywania obrazów, który opisany został w późniejszej części tego rozdziału.

4.3 Kodowanie

Kodowanie tęczówki można rozumieć jako opis jej cech. Dobry algorytm ekstrakcji cech sprawi, że wynik miary dopasowania dla obrazów tej samej tęczówki będzie znajdował się w innym przedziale niż podczas porównywania obrazów dwóch różnych tęczówek. Dzięki temu w końcowym etapie procesu rozpoznawania możliwe jest podjęcie decyzji o rozpoznaniu bądź nierozpoznaniu tęczówki.

Aby zapewnić dobrą jakość identyfikacji tęczówki, proces kodowania powinien wyciągać z obrazu tylko najważniejsze i najbardziej rozróżnialne jego cechy. Oppenheim i Lim [21] pokazali w swojej pracy, że najważniejsze cechy obrazu niesie ze sobą widmo fazowe. Widmo amplitudowe zawiera informacje o mniej indywidualnych cechach, a także jest zależne od czynników zewnętrznych takich jak kontrast czy oświetlenie. Z tego względu podczas kodowania tęczówki wykorzystane powinno być właśnie widmo fazowe obrazu. W celu jego uzyskania należy przenieść znormalizowany obraz z dziedziny przestrzennej do dziedziny częstotliwości.

Daugman [3] w swojej pracy zaproponował użycie w tym celu filtrów Gabora, dzięki którym można uzyskać połączoną reprezentację obrazu w przestrzeni oraz częstotliwości. Filtry te powstają w wyniku modulacji sinusoidy oraz cosinusoidy za pomocą funkcji Gaussowskiej. Częstotliwość środkowa filtru wyznaczana jest przez częstotliwość sinusoidy, natomiast przepustowość filtru określana jest przez szerokość wykorzystanej funkcji Gaussowskiej.

Jedną z wad filtrów Gabora jest występowanie niezerowej składowej stałej dla przepustowości większej niż jedna oktawa [5]. Zerową składową stałą dla każdej przepustowości można natomiast otrzy-

mać przez zastosowanie filtru, którego charakterystyka częstotliwościowa amplitudowa ma rozkład Gaussowski nie w skali liniowej, a w skali logarytmicznej. Charakterystyka częstotliwościowa amplitudowa takiego filtru jest opisana równaniem 4.3:

$$G(f) = \exp\left(\frac{-\left(\log\left(f/f_0\right)\right)^2}{2\left(\log\left(\sigma/f_0\right)\right)^2}\right), \tag{4.3}$$

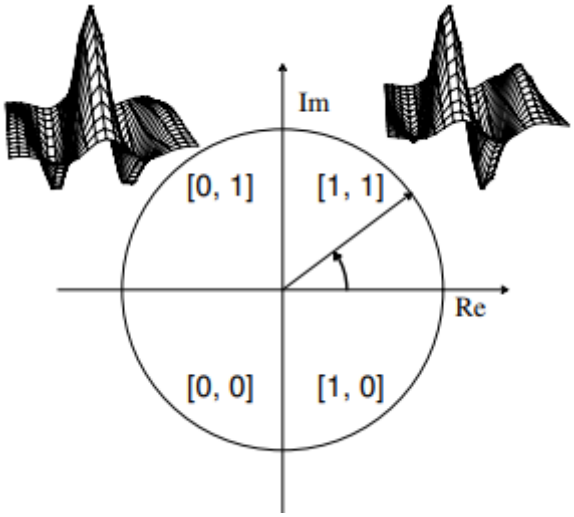
gdzie:

f_0 - częstotliwość środkowa filtra,

σ - przepustowość filtra.

W celu zakodowania cech charakterystycznych znormalizowanego obrazu tęczówki, dla każdego wiersza obrazu obliczany jest jego splot z falkami Log Gabora. Jeżeli moduł wyniku splotu w danym punkcie jest bardzo blisko zera, wówczas informacja fazowa jest nieznacząca, a punkt ten oznaczany jest w masce zakłóceń jako bit nieznaczący.

Wynik splotu obrazu z filtrem jest następnie poddawany kwantyzacji [3] do czterech wartości odpowiadających czterem ćwiartkom płaszczyzny zespolonej przez sprawdzenie znaku części rzeczywistej i urojonej uzyskanego wyniku. Proces kwantyzacji przedstawiony jest na rysunku 4.4. W wyniku kwantyzacji znormalizowany obraz tęczówki przekształcany jest do wzoru tęczówki w postaci ciągu bitów. Przykładowy proces kodowania przedstawiony został na rysunku 4.5



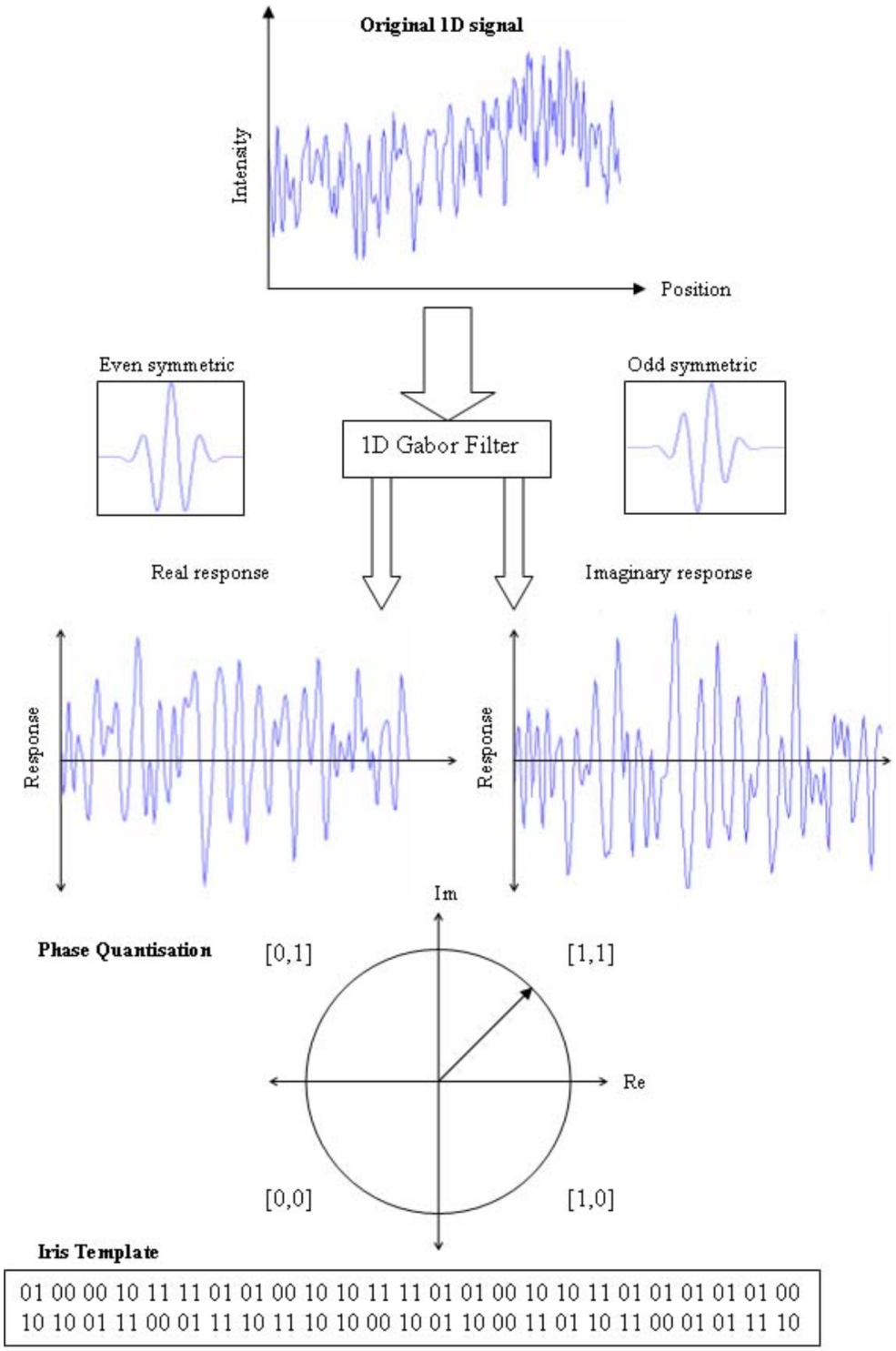
Rysunek 4.4: Ilustracja procesu kwantyzacji

W celu uzyskania większej ilości informacji o tęczówce możliwe jest zastosowanie kilku filtrów Gabora o różnych parametrach. Wówczas obraz poddawany jest splotowi z każdym z takich filtrów, w związku z czym powielana jest liczba bitów kodujących każdy punkt tęczówki.

Z każdym punktem na obrazie tęczówki związana jest pojedyńcza wartość w masce zakłóceń. W wyniku procesu kodowania każdy element obrazu jest reprezentowany przez przynajmniej dwie wartości. W związku z tym należy zaktualizować maskę zakłóceń, aby miała ona ten sam rozmiar co otrzymany wzór tęczówki. W tym celu każdy bit w masce zakłóceń jest powielany tyle razy, ile wartości reprezentuje pojedyńczy punkt we wzorze tęczówki. Przykładowo, jeżeli w procesie kodowania wykorzystany został jeden filtr, transformacja przykładowej maski wyglądałaby następująco:

$$0|1|1|0|0|1 \rightarrow 00|11|11|00|00|11$$

Dorzuc przyklad?



Rysunek 4.5: Ilustracja procesu kodowania

4.4 Dopasowanie

W procesie kodowania obraz tęczówki przekształcony zostaje do wzoru tęczówki w postaci ciągu bitowego, dzięki czemu porównanie dwóch obrazów tęczówek sprowadza się do porównania dwóch ciągów bitowych. Stopień różnicy między dwoma wzorami tęczówki, można określić obliczając odległość Hamminga. Mówi ona o liczbie miejsc, w których dwa słowa bitowe przyjmują różne wartości.

Ustalając pewien próg wartości odległości Hamminga można podjąć decyzję, czy dwa dane ciągi reprezentują tę samą tęczówkę.

Wyznaczenie progu dla odległości Hamminga zdefiniowanej w ten sposób zależałoby od długości porównywanych ciągów bitowych. Aby uniezależnić wartość progu od tej długości, miarę można zdefiniować jako sumę niezgodnych bitów podzieloną przez całkowitą liczbę bitów 4.4:

$$HD = \frac{1}{N} \sum_{j=1}^{N} X_j \oplus Y_j, \tag{4.4}$$

gdzie:

X_j, Y_j reprezentują wartości bitów na miejscu j w ciągach odpowiednio X oraz Y,

N - długość ciągów X i Y,

$\oplus$ - operator alternatywy rozłącznej.

W trakcie procesu kodowania oprócz wzoru tęczówki tworzony jest także wzór dla maski zakłóceń. Zawiera on informacje o tym, które bity we wzorze tęczówki reprezentują tęczówkę i powinny być uwzględnione, a które bity reprezentują zakłócenia takie jak powieki, czy rzęsy i nie powinny być porównywane w procesie dopasowywania.
Jako że porównywane są dwa obrazy, generowane są również dwie maski zakłóceń i należy uwzględnić każdą z nich. Ponieważ elementy znaczące w masce zakłóceń oznaczone zostały kolorem białym, co odpowiada wartości 1 w postaci bitowej. Ostateczną maskę dla procesu dopasowania można zdefiniować jako iloczyn logiczny tych dwóch masek - w ten sposób stworzony zostanie jeden ciąg bitowy reprezentujący bity znaczące za pomocą wartości 1 oraz bity nieznaczące za pomocą wartości 0. Równanie 4.5 przedstawia definicję odległości Hamminga z uwzględnieniem obu masek zakłóceń:

$$HD = \frac{1}{\sum_{j=1}^{N} \left(Xm_j \cap Ym_j\right)} \sum_{k=1}^{N} (X_k \oplus Y_k) \cap (Xm_k \cap Ym_k), \tag{4.5}$$

gdzie:

X, Y - wzory tęczówki,

Xm, Ym - wzory masek zakłóceń dla wzorów tęczówek X, Y,

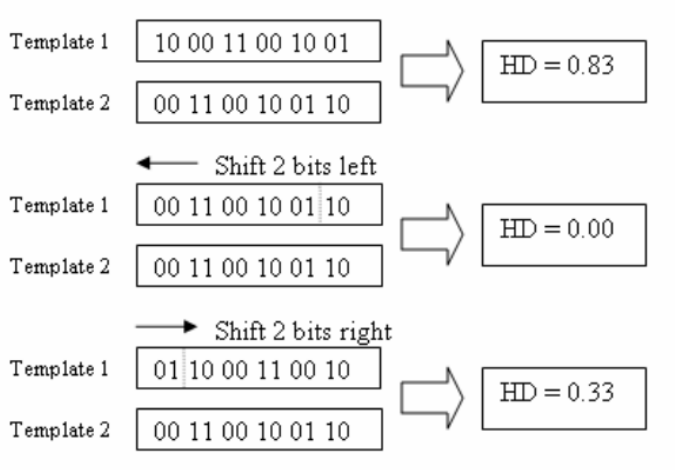
N - długość wzorów tęczówek i masek zakłóceń,

$\oplus$ - operator alternatywy rozłącznej,

$\cap$ - operator iloczynu logicznego.

Tak jak wcześniej wspomniano, poprzednie procesy w żaden sposób nie uwzględniały możliwości różnic w rotacji tęczówki na pobranych obrazach. W celu kompensacji tych niespójności podczas procesu dopasowania jeden z wzorów tęczówki poddawany jest przesunięciom bitowym, które odpowiadają rotacji tęczówki o kąt zależny od rozdzielczości kątowej wybranej podczas procesu normalizacji. Maska zakłóceń odpowiadająca temu wzorowi również poddawana jest tym przesunięciom. Jedno przesunięcie w procesie dopasowania odpowiada dwóm przesunięciom bitowym - jednemu w lewo i drugiemu w prawo, z których oba wykonywane są względem oryginalnego wzoru tęczówki. Proces ten zobrazowany został na rysunku 4.6

W zależności od tego ile bitów koduje pojedyńczy punkt siatkówki, tyle bitów zmienia swoje miejsca w trakcie pojedyńczego przesunięcia. Liczba przesuniętych bitów zależy od ilości filtrów Gabora użytych w procesie kodowania, ponieważ każdy z takich filtrów wygeneruje dwa bity reprezentujące pojedyńczy punkt siatkówki.



Rysunek 4.6: Schemat przedstawiający pojedyńcze przesunięcie wzoru tęczówki. Przykład ten przedstawia wzór do wygenerowania którego wykorzystany został jeden filtr Gabora [18].

Odległosć Hamminga obliczana jest dla każdego z tych przesunięć. Decyzja o tym, czy dwa wzory reprezentują tę samą tęczówkę podejmowana jest na podstawie najmniejszej uzyskanej wartości, która reprezentuje najlepsze dopasowanie dwóch wzorów. Liczba przesunięć potrzebna do kompensacji niespójności rotacji zależy od tego z jaką dokładnością kątową pobierane są zdjęcia tęczówki.

Rozdział 5

Testy wydajnościowe

5.1 Wstęp

W poniższym rozdziale przedstawione zostaną wyniki testów wydajnościowych implementacji algorytmów opisanych w poprzedniej części pracy (poza algorytmem *two-stage*, który został rozwinięty i poprawiony przez *improved two-stage*). Implementacja w języku `Java` została oparta o oryginalny kod, udostępniony przez autorów. Podstawą implementacji metody *improved two-stage* był kod algorytmu *divsufsort* napisany przez Yutę Moriego [K].

Algorytmy tworzenia tablic sufiksów zaimplementowane zostały w ten sposób, żeby na wejściu przyjmowały sekwencje liczb całkowitych typu `int` zajmujących w języku `Java` 4 bajty. Spowodowało to zwiększenie złożoności pamięciowej algorytmów (wartości podane w tabeli **??** zakładały wejście w postaci sekwencji jednobajtowych elementów). Powodem zwiększenia rozmiaru pojedynczego elementu wejściowego było umożliwienie tworzenia tablic sufiksów dla sekwencji symboli z alfabetów o dużym rozmiarze. Nie jest to jednak możliwe w praktyce w przypadku wszystkich algorytmów; rzeczywista złożoność pamięciowa implementacji wybranych metod oraz ich zależność od rozmiaru alfabetu wejściowego opisane zostały w tabeli 5.1. Testowane implementacje algorytmów *bpr*, *deep-shallow* i *qsufsort* dla zaoszczędzenia pamięci nadpisywały ciąg wejściowy. Na potrzeby testów wydajnościowych zaimplementowano również naiwny algorytm tworzenia tablic sufiksów opierający się o zwykły algorytm sortujący *quicksort*. W dalszej części rozdziału metoda naiwna nazywana będzie *naive sort* lub NS.

Wykonane testy algorytmów można podzielić na dwie główne kategorie: testy na wejściu generowanym losowo, oraz testy na wejściu wczytywanym z plików. Testy drugiego typu wykonywane były na kilku różnych maszynach wirtualnych:

- maszyna Java HotSpot 64-Bit Server VM 11.0-b16 firmy Sun Microsystems, oznaczana w dalszej części pracy jako `sun`,
- maszyna IBM J9 VM 2.4 firmy IBM, oznaczana w dalszej części pracy jako `ibm`,
- BEA JRockit(R) R27.5.0-110_o-99226-1.6.0_03 firmy BEA (aktualnie przejęta przez Oracle), oznaczana w dalszej części pracy jako `jrockit`,
- Apache Harmony DRLVM 11.2.0 tworzona przez Apache Software Foundation, oznaczana w dalszej części pracy jako `harmony`.

Testy przeprowadzone zostały na kilku komputerach testowych o następujących parametrach:

- dwurdzeniowy procesor Athlon 5200 o prędkości 2.6 GHz, 64-bitowy system operacyjny OpenSuSE, jądro w wersji 2.6.22.19-0.2-default,

Nazwa	Złożoność pamięciowa
skew	$16n$
bpr	$4\|\Sigma\|^3 + 12n$
deep-shallow	$4\|\Sigma\|^2 + (x+8)n$
divsufsort	$4\|\Sigma\|^2 + 8n$
qsufsort	$8n$
naive sort	$8n$

Tablica 5.1: Zaimplementowane algorytmy tworzenia tablic sufiksów. Parametr n oznacza długość wejścia, parametr $|\Sigma|$ oznacza wielkość alfabetu sekwencji wejściowej. Algorytm *deep-shallow* wymaga alfabetu o rozmiarze nie większym niż 256, zużycie pamięci przez ten algorytm zależy od wielkości budowanego drzewa *blind trie*, które maksymalnie może zawierać n elementów. Rozmiar jednego elementu drzewa wynosi $x = 48$ bajtów na 64-bitowej maszynie wirtualnej firmy Sun. Algorytmy *bpr* i *divsufsort* nie mają sztywnych ograniczeń na rozmiar alfabetu, ale ze względu na wydajność nie powinny być używane na dużych alfabetach.

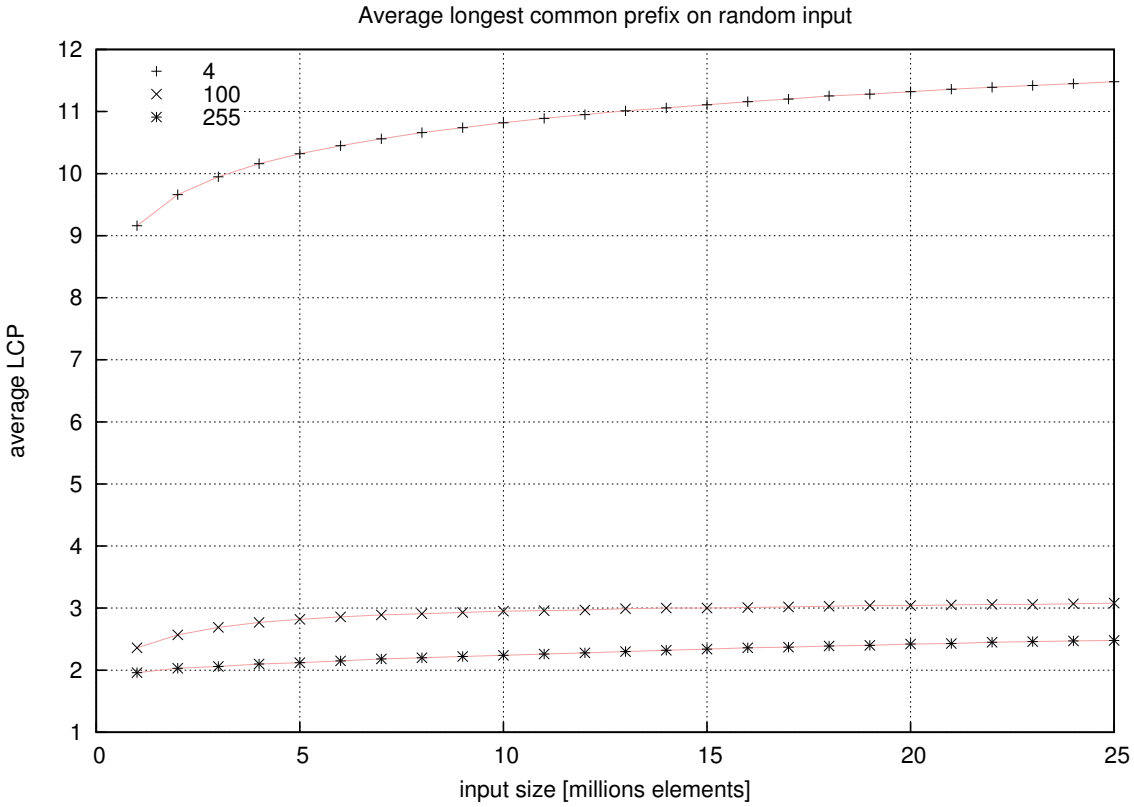
- czterordzeniowy procesor Intel Xeon x3230 o prędkości 2.66 GHz, 64-bitowy system operacyjny OpenSuSE, wersja jądra 2.6.22.19-0.2-default,
- jednordzeniowy Intel Pentium 4 o prędkości 3 GHz, 32-bitowy system operacyjny Windows XP.

Wszystkie obliczenia przeprowadzone zostały z parametrami maszyn wirtualnych odpowiadającym opcjom „`-Xmx2g -server`" – zwiększają one maksymalny rozmiar sterty do 2Gb, a maszyna wirtualna pracuje w trybie „serwerowym" (agresywna optymalizacja JIT).

Czas działania algorytmów mierzony był przez klasę uruchamiającą testy, poprzez liczenie różnicy wartości zwracanych przez metodę `System.currentTimeMillis()` przed i po obliczeniach (tzw. *wall time*). Do mierzenia zużycia pamięci wykorzystano specjalnie w tym celu napisany aspekt [B]. Mierzone było zużycie pamięci wewnątrz wirtualnej maszyny na początku działania algorytmu, pod jego koniec oraz po zakończeniu wybranych metod wewnątrz klas. Mierzenie zużycia pamięci w języku `Java` jest trudne, bowiem maszyna wirtualna nie pozwala na dokładne oszacowanie wszystkich dokonanych alokacji pamięci. Do celów testowych posłużono się programowaniem aspektowym i „wpleciono" (ang. *code weaving*) dynamiczne instrukcje mierzące różnicę w aktualnej ilości zaalokowanej pamięci względem startu algorytmu. Nie jest to oszacowanie idealne, ale pozwala na zgrubne stwierdzenie ile pamięci zużywa dany algorytm.

Test na jednej instancji wejścia (pliku lub wygenerowanej sekwencji o ustalonych parametrach) powtarzany był 10 razy, a jego wynikiem jest średnia z zebranych pomiarów. Testy których wyniki zbierano poprzedzano kilkoma uruchomieniami algorytmu. Celem tego zabiegu było wyeliminowanie błędów pomiarowych wynikających z wolniejszego działania kodu, który nie został skompilowany do kodu natywnego przez maszynę wirtualną. Testy na wejściu generowanym losowo były poprzedzone 10 rundami „rozgrzewki", natomiast przed testem na wejściu wczytywanym z pliku algorytm uruchamiany był 5 razy.

W dalszej części rozdziału prezentowane są wyniki pochodzące z pomiarów na komputerze z procesorem Xeon x3230. Wyniki z pozostałych maszyn są niemal identyczne pod względem porównywania i rankingu algorytmów, dlatego też w wielu miejscach je pominięto (wszystkie wyniki są na dołączonej do pracy płycie CD). Samo porównanie efektywności wykonania programów w różnych maszynach wirtualnych znajduje się w rozdziale 5.3.2.



Rysunek 5.1: Średnie *lcp* w zależności od długości wejścia dla alfabetów o wielkości 4, 100 i 255 symboli.

5.2 Testy wydajnościowe na losowo generowanym wejściu

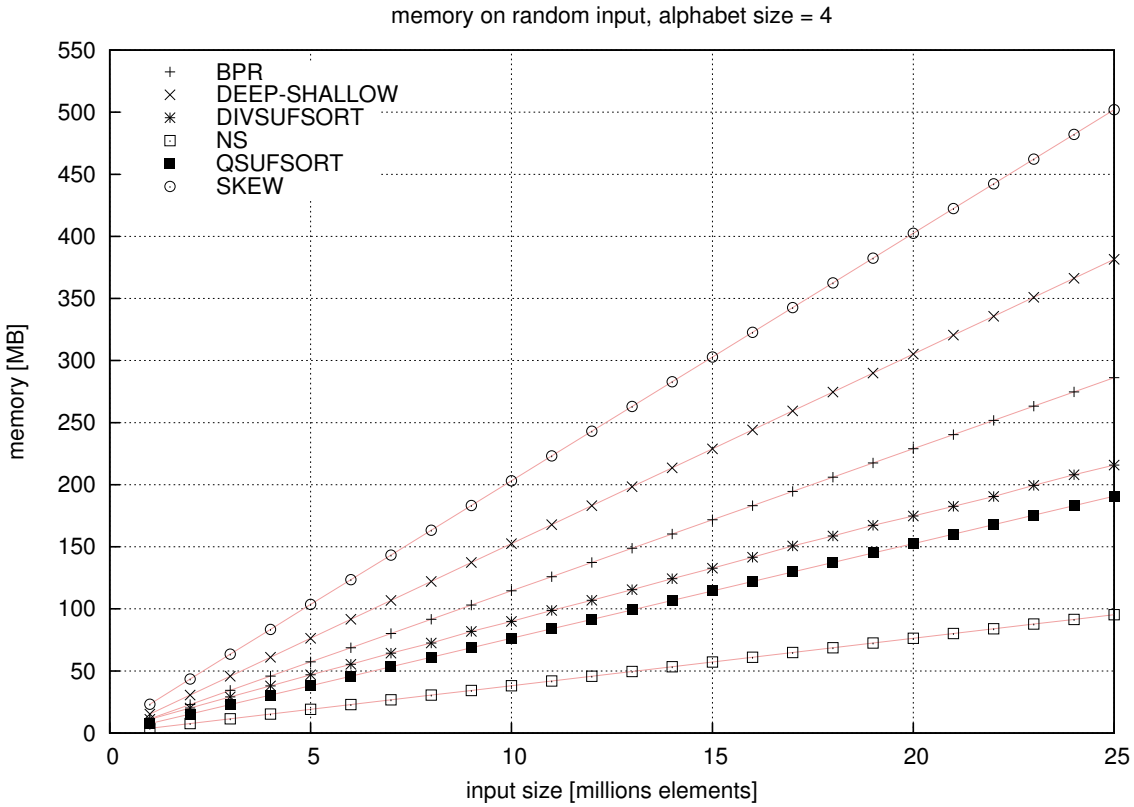
Celem testów na losowo generowanym wejściu jest zbadanie rzeczywistej zależności algorytmów od długości wejścia, wielkości alfabetu oraz średniego *lcp* ciągu wejściowego. Testy przeprowadzono tylko na jednej maszynie wirtualnej (`sun`). Ziarno generatora liczb losowych otrzymywało identyczną wartość przed testem każdego algorytmu by zapewnić powtarzalność wyników.

5.2.1 Wejście o zmiennej długości i stałej wielkości alfabetu

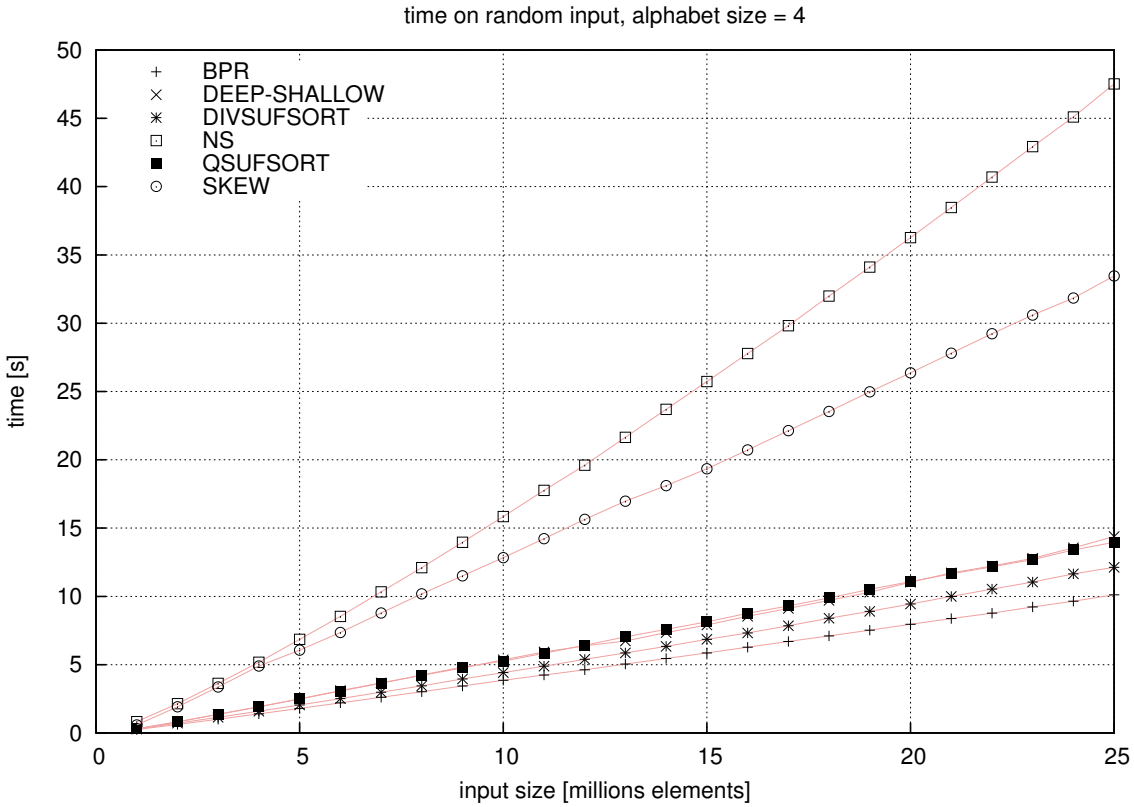
Testy algorytmów na losowym wejściu o zmiennej długości i stałej wielkości alfabetu powtórzono trzy razy. Każdy test wykonany był dla wejścia generowanego z alfabetu wielkości 4,100 i 255 elementów. Rysunek 5.1 przedstawia średnią wartość *lcp* generowanych ciągów wejściowych. Z rysunku wynika, że wartość średniego *lcp* jest mniej więcej odwrotnie proporcjonalna do wielkości alfabetu.

Rysunki 5.3 i 5.2 przedstawiają czas działania algorytmów oraz zużycie pamięci dla alfabetu wielkości 4. Algorytmy *qsufsort* i *naive sort* uzyskały najlepszy wynik złożoności pamięciowej. Zgodnie z przewidywaniami, algorytmy *skew* i *deep shallow* wypadł najgorzej w tej kwestii. Najszybszym algorytmem okazał się być algorytm *bpr*. Wyraźnie wolniejsze od pozostałych algorytmów są algorytmy *skew* i *naive sort*.

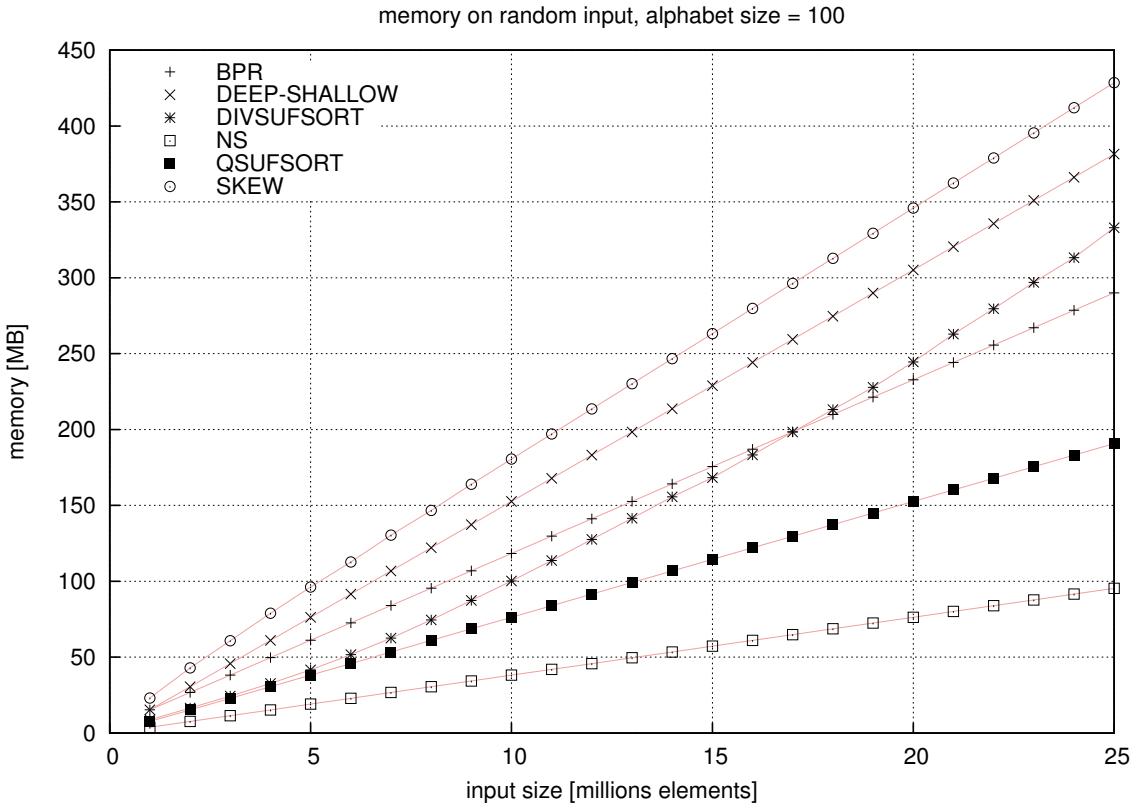
Wyniki testów wejścia generowanego z alfabetu wielkości 100 przedstawione są na rysunku 5.5 i 5.4. Rezultaty testu czasu działania algorytmu są niemal identyczne jak poprzedniego. Jedyną różnicą jest to, że algorytm *bpr* nie uzyskał najlepszego wyniku lecz znalazł się wśród kilku algorytmów uzyskujących bardo dobry wynik, zajął również więcej pamięci.



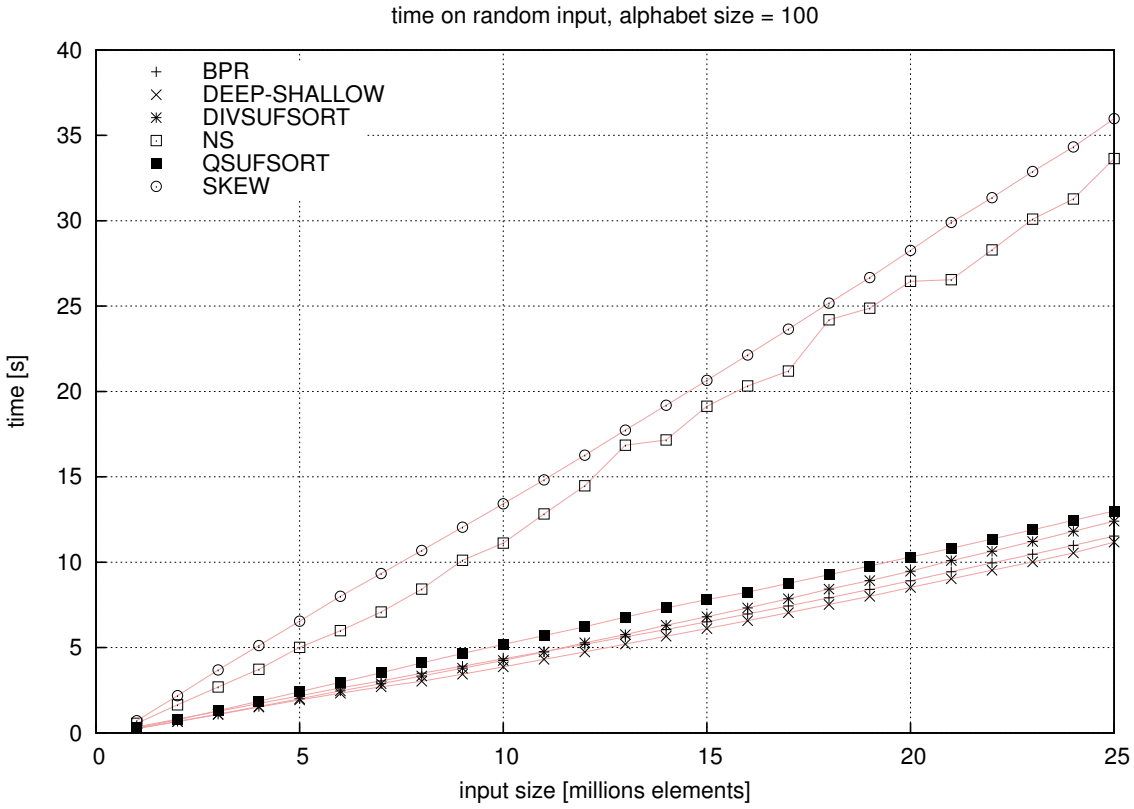
Rysunek 5.2: Zużycie pamięci w zależności od długości wejścia generowanego z alfabetu wielkości 4.



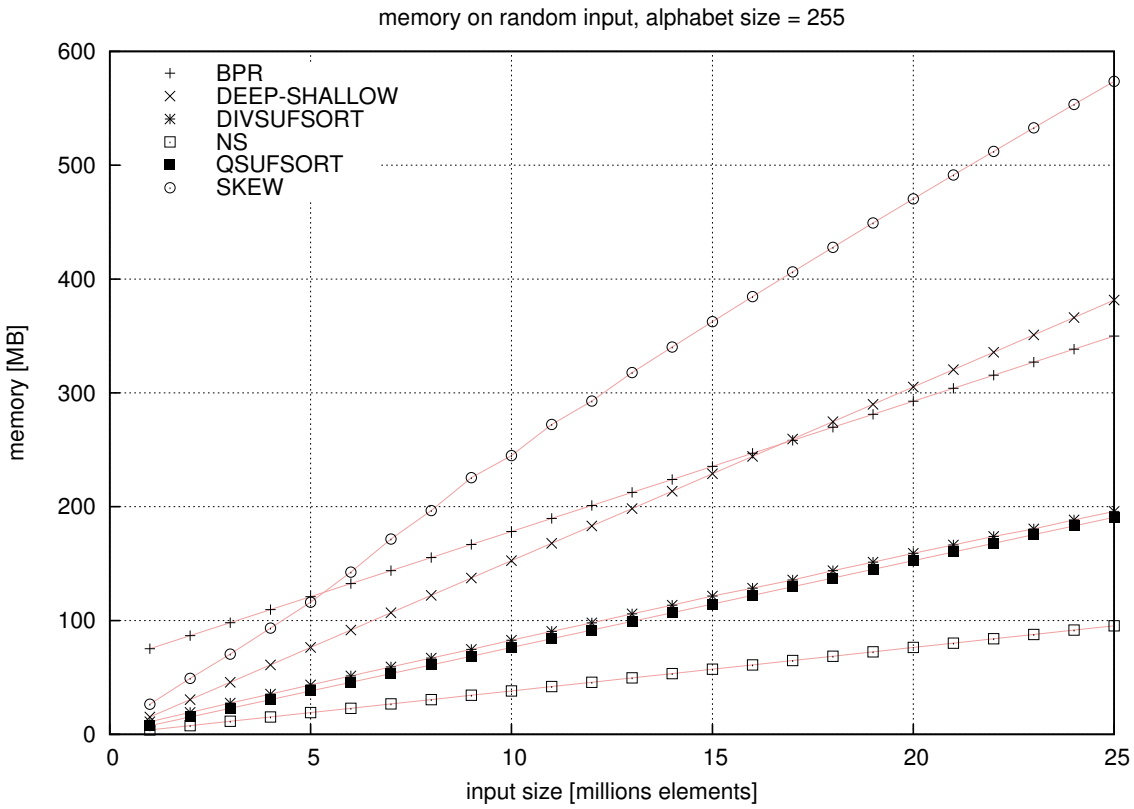
Rysunek 5.3: Czas działania algorytmów w zależności od długości wejścia generowanego z alfabetu wielkości 4.



Rysunek 5.4: Zużycie pamięci w zależności od długości wejścia generowanego z alfabetu wielkości 100.



Rysunek 5.5: Czas działania algorytmów w zależności od długości wejścia generowanego z alfabetu wielkości 100.



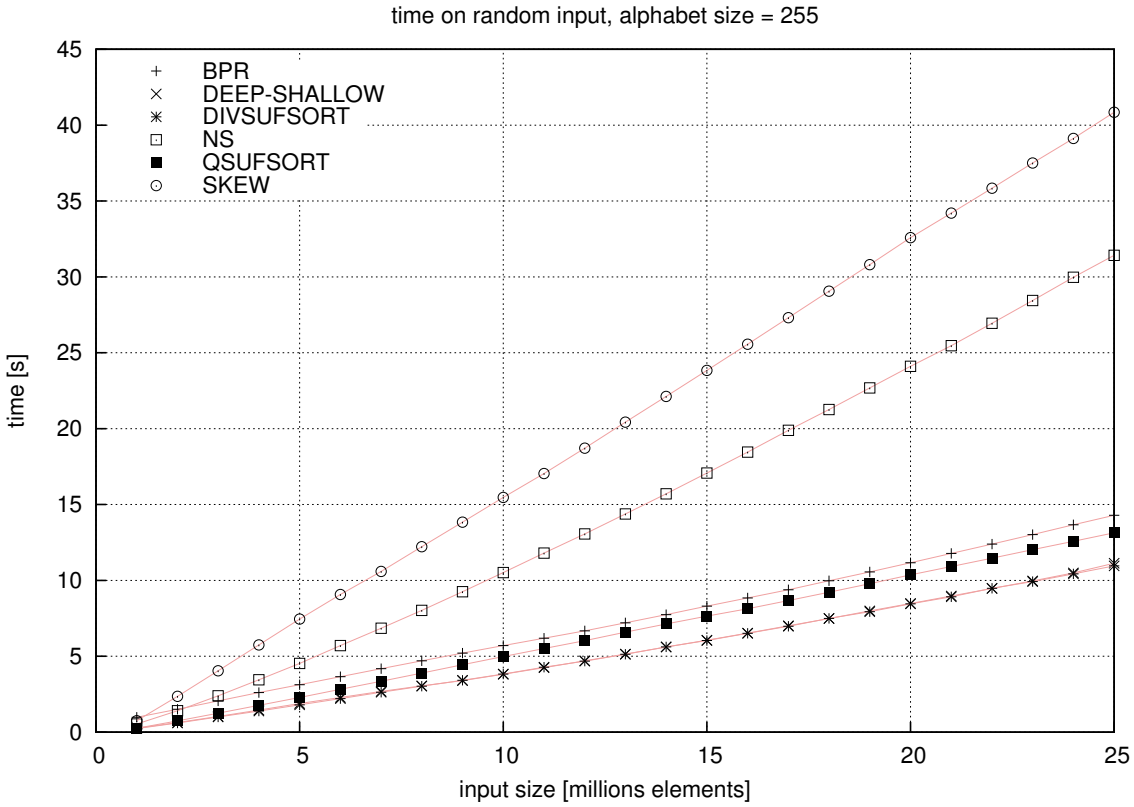
Rysunek 5.6: Zużycie pamięci w zależności od długości wejścia generowanego z alfabetu wielkości 255.

Rysunki 5.7 i 5.6 prezentują wyniki testów wejścia generowanego z alfabetu wielkości 255. Wynik tego testu są również podobne do poprzedników. Algorytm *bpr* uzyskał jeszcze gorszy wynik niż w poprzednich testach, zajął więcej pamięci i działał wolniej od większości algorytmów.

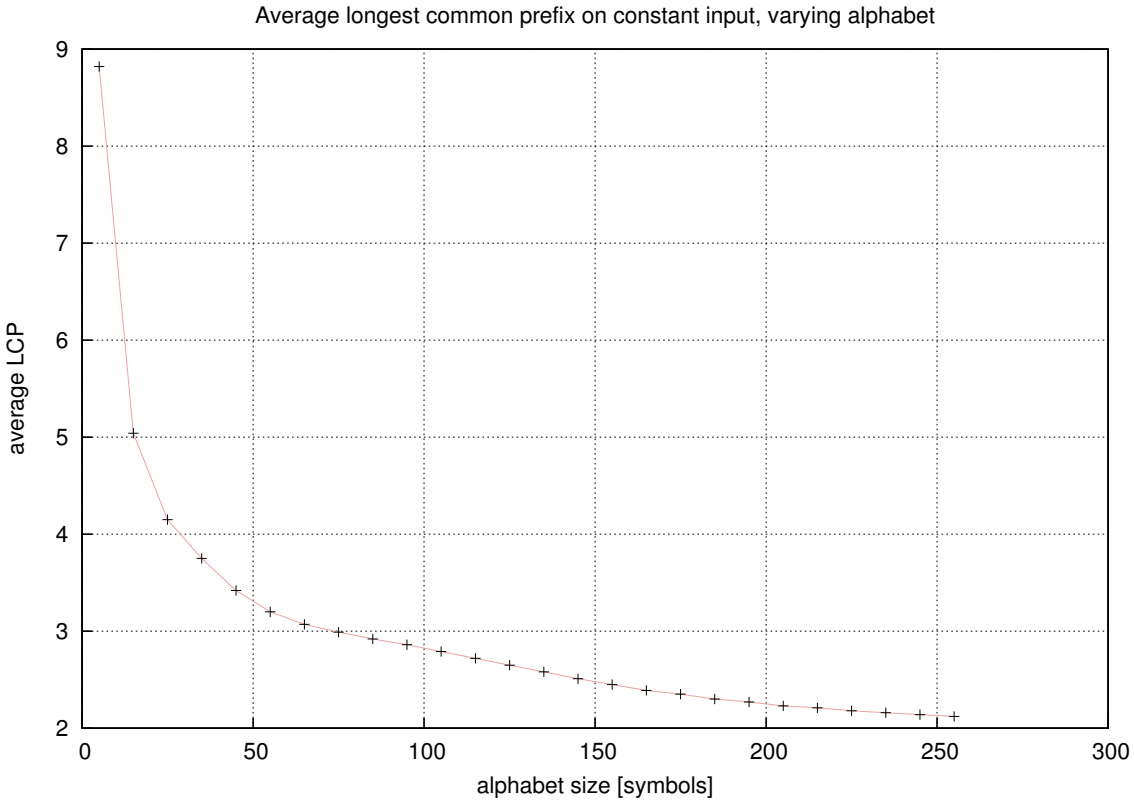
5.2.2 Wejście o stałej długości i zmiennej wielkości alfabetu

Wejście generowane na potrzeby testu miało długość 5 000 000 elementów. Wykres 5.8 przedstawia średnie *lcp* wygenerowanego wejścia w zależności od wielkości alfabetu. Wykres 5.9 przedstawia czasy działania algorytmów na generowanym wejściu.

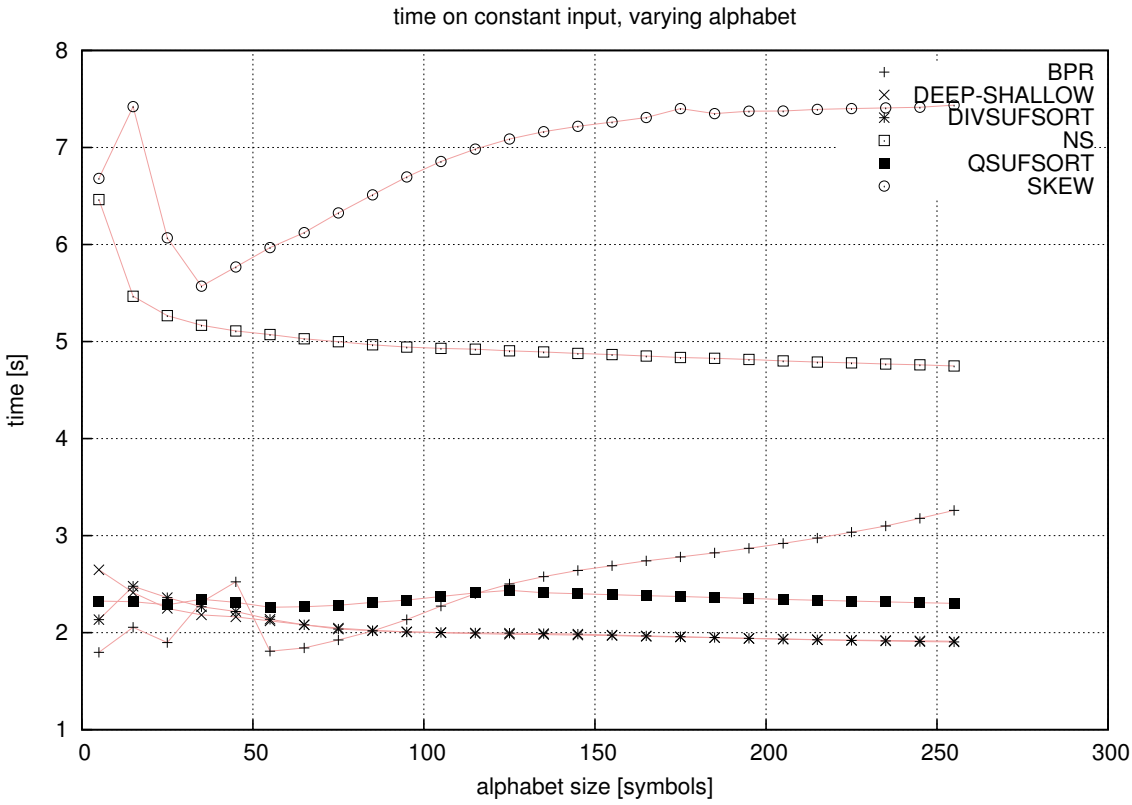
Wyniki testu pokazują które algorytmy zależą od średniego *lcp* i wielkości alfabetu sekwencji wejściowej. Algorytm *bpr* jest tego najlepszym przykładem – im większa jest jedna z tych wartości, tym gorzej sobie radzi. Algorytm *skew* uzyskuje lepsze wyniki dla sekwencji o większej wartości średniego *lcp*. Pozostałe algorytmy nie wykazują większej zależności pomiędzy czasem ich działania a wielkością alfabetu i średnim *lcp*.



Rysunek 5.7: Czas działania algorytmów w zależności od długości wejścia generowanego z alfabetu wielkości 255.



Rysunek 5.8: Średnie *lcp* generowanego wejścia w zależności od wielkości alfabetu.



Rysunek 5.9: Czas działania algorytmów w zależności od wielkości alfabetu.

5.3 Testy wydajnościowe na wejściu wczytywanym z plików

Zaimplementowane algorytmy przetestowane zostały na dwóch zestawach plików (korpusach) specjalnie przygotowanych do testowania algorytmów tworzenia tablic sufiksów. Pierwszy z nich nosi nazwę `Gauntlet`, powstał z inicjatywy Michaela Maniscalco. Ideą stojącą za stworzeniem tego korpusu było zebranie plików o nietypowej strukturze w celu testowania algorytmów na szczególnych przypadkach wejścia. Wkład w powstanie tego zbioru plików wnieśli Yuta Mori, Simon Puglisi oraz Graham Houston. Korpus dostępny jest do pobrania pod adresem [C].

Drugi korpus testowy opracowany został przez Giovanniego Manziniego. W jego skład wchodzą rzeczywiste pliki o dużym rozmiarze pochodzące z różnych źródeł. Zestaw plików dostępny jest do pobrania pod adresem [D]. Tabela 5.2 prezentuje charakterystykę plików obu korpusów.

Plik	Rozmiar	Średnie *lcp*
`abac`	200 000	99 997
`abba`	10 500 600	2 773 939
`book1x20`	15 375 420	6 938 159
`fib_s14930352`	14 930 352	3 940 597
`fss10`	12 078 908	2 454 179
`fss9`	2 851 443	579 353
`houston`	3 840 000	52 083
`paper5x80`	981 924	239 421
`test1`	2 097 152	1 048 064
`test2`	2 097 152	1 048 064
`test3`	2 097 152	984 064

Plik	Rozmiar	Średnie *lcp*
`chr22.dna`	34 553 758	1 979
`etext99`	105 277 340	1 108
`gcc-3.0.tar`	86 630 400	8 603
`howto`	39 422 105	267
`jdk13c`	69 728 899	678
`linux-2.4.5.tar`	116 254 720	479
`rctail96`	114 711 151	282
`rfc`	116 421 901	93
`sprot34.dat`	109 617 186	89
`w3c2`	104 201 579	42 299

Tablica 5.2: Charakterystyka plików wchodzących w skład korpusu `Gauntlet` (po lewej) i korpusu Giovanniego Manziniego (po prawej). Rozmiar i średnie *lcp* podano w bajtach.

5.3.1 Szczegółowe wyniki na maszynie wirtualnej firmy Sun

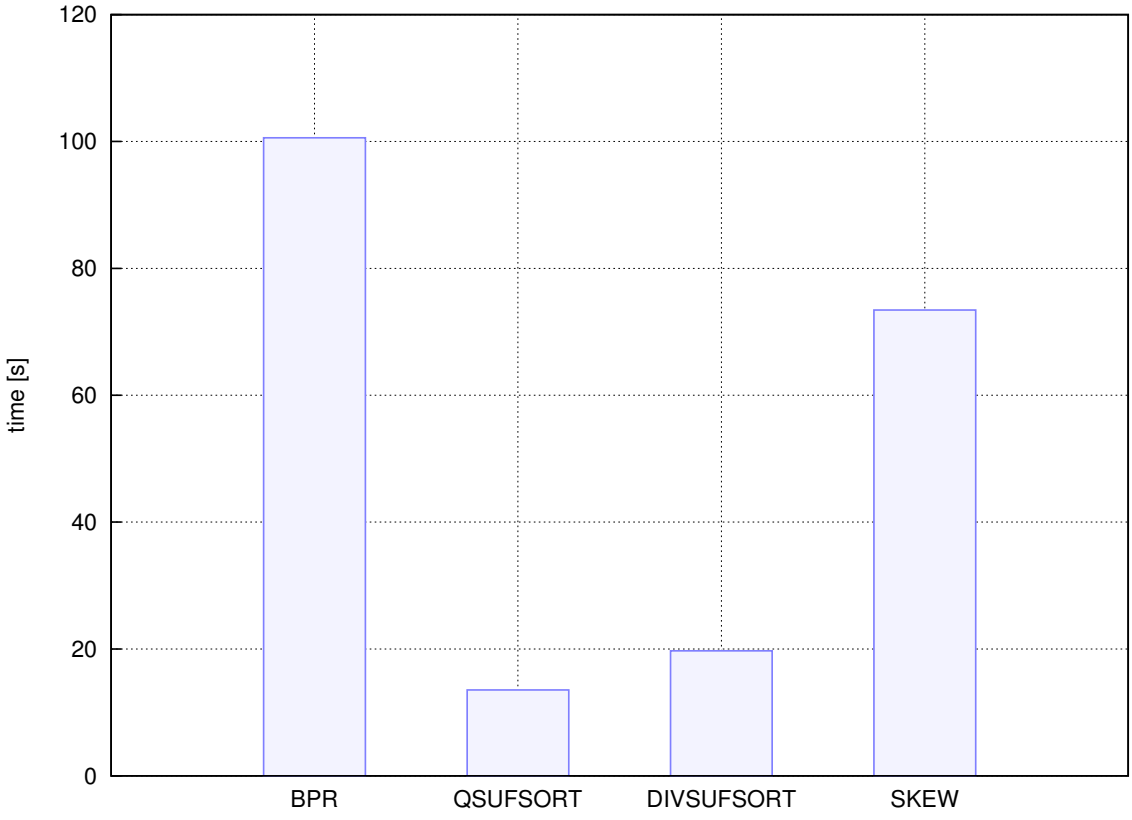
Ze względu na wysokie podobieństwo wyników, w poniższym rozdziale prezentowane są szczegółowe rezultaty tylko z jednej maszyny wirtualnej (`sun`). Podobnie jak w przypadku testów na różnych komputerach, wyniki testów na różnych maszynach wirtualnych są identyczne w sensie rankingu algorytmów.

Algorytm *deep-shallow* został pominięty w poniższych testach ze względu na nieakceptowalnie długi czas działania. Testy z jego udziałem wykazały wielokrotnie gorszy czas działania algorytmu od najgorszego z pozostałych. Również ze względu na czas działania algorytmu pominięto metodę naiwną. Algorytmy *skew* i *bpr* podczas przetwarzania niektórych plików kończyły się wyjątkiem braku pamięci. Takie przypadki oznaczone zostały w zestawieniach znakiem –, błąd działania algorytmu na jednym z plików danego korpusu powodował pominięcie tej metody w podsumowaniu testów na całym zbiorze plików.

Wyniki testów na korpusie `The Gauntlet` przedstawione zostały w tabeli 5.3. Rysunek 5.10 obrazuje porównanie sumy czasów działania algorytmów na wszystkich plikach korpusu (poza tymi algorytmami, które błędnie zakończyły działanie). Analogiczne wyniki testów na korpusie Giovanniego Manziniego przedstawione są w tabeli 5.4 oraz rysunku 5.11. Najlepszym algorytmem okazał się być *qsufsort*, który uzyskał najlepszy wynik na znakomitej większości plików. Drugie miejsce przypadło algorytmowi *divsufsort*. Pozostałe algorytmy uzyskiwały znacznie gorsze wyniki lub nie kończyły obliczeń.

	bpr	*divsufsort*	*qsufsort*	*skew*
`abac`	1.07	0.01	**0.00**	0.03
`abba`	3.80	2.49	**2.34**	12.09
`book1x20`	**3.36**	4.11	3.44	23.23
`fib_s14930352`	9.68	6.08	**3.27**	15.88
`fss10`	5.18	4.84	**2.65**	13.01
`fss9`	1.01	0.74	**0.57**	2.14
`houston`	2.29	**0.18**	0.43	1.20
`paper5x80`	0.14	0.12	**0.06**	0.47
`test1`	2.37	0.40	**0.21**	2.17
`test2`	0.81	0.30	**0.21**	2.18
`test3`	70.87	0.45	**0.36**	1.04
Total	100.58	19.70	**13.55**	73.44

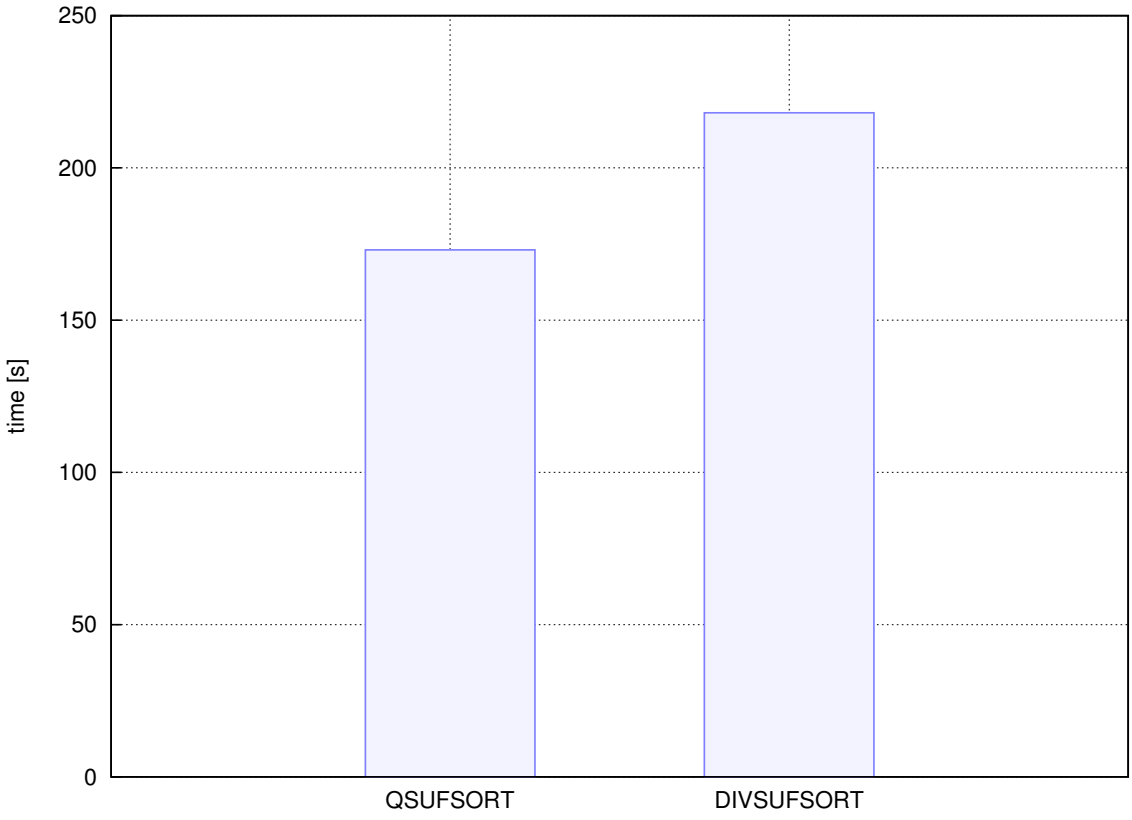
Tablica 5.3: Czas działania algorytmów na plikach z korpusu `Gauntlet`.



Rysunek 5.10: Sumaryczny czas działania algorytmów na plikach z korpusu `Gauntlet`.

	bpr	*divsufsort*	*qsufsort*	*skew*
`chr22.dna`	**6.85**	8.56	7.70	64.78
`etext99`	—	30.66	**25.56**	—
`gcc-3.0.tar`	20.73	18.62	**14.22**	—
`howto`	8.07	9.03	**7.45**	83.77
`jdk13c`	15.63	15.44	**11.58**	—
`linux-2.4.5.tar`	—	23.94	**19.98**	—
`rctail96`	—	29.96	**23.89**	—
`rfc`	—	26.96	**23.78**	—
`sprot34.dat`	—	31.66	**22.24**	—
`w3c2`	—	23.25	**16.71**	—
Total		218.09	**173.10**	

Tablica 5.4: Czas działania algorytmów na plikach z korpusu Giovanniego Manziniego.



Rysunek 5.11: Sumaryczny czas działania algorytmów na plikach z korpusu Giovanniego Manziniego.

5.3.2 Porównanie czasów działania algorytmów na różnych maszynach wirtualnych

Poniżej przedstawione są wyniki podsumowań czasów działania algorytmów na korpusach testowych. Jak już wspomniano wcześniej, ranking algorytmów jest identyczny dla każdej maszyny wirtualnej. Celem prezentowania zestawień jest znalezienie takiej maszyny wirtualnej, na której algorytmy działają najszybciej. Tabele 5.6 i 5.5 zawierają czasy działań algorytmów na plikach z korpusów Giovanniego Manziniego i `The Gauntlet`. Ze względu na bardzo niskie wartości odchylenia standardowego nie umieszczono go tabelach. Wizualne porównanie sum tych wyników znajduje się na rysunkach 5.13 i 5.12.

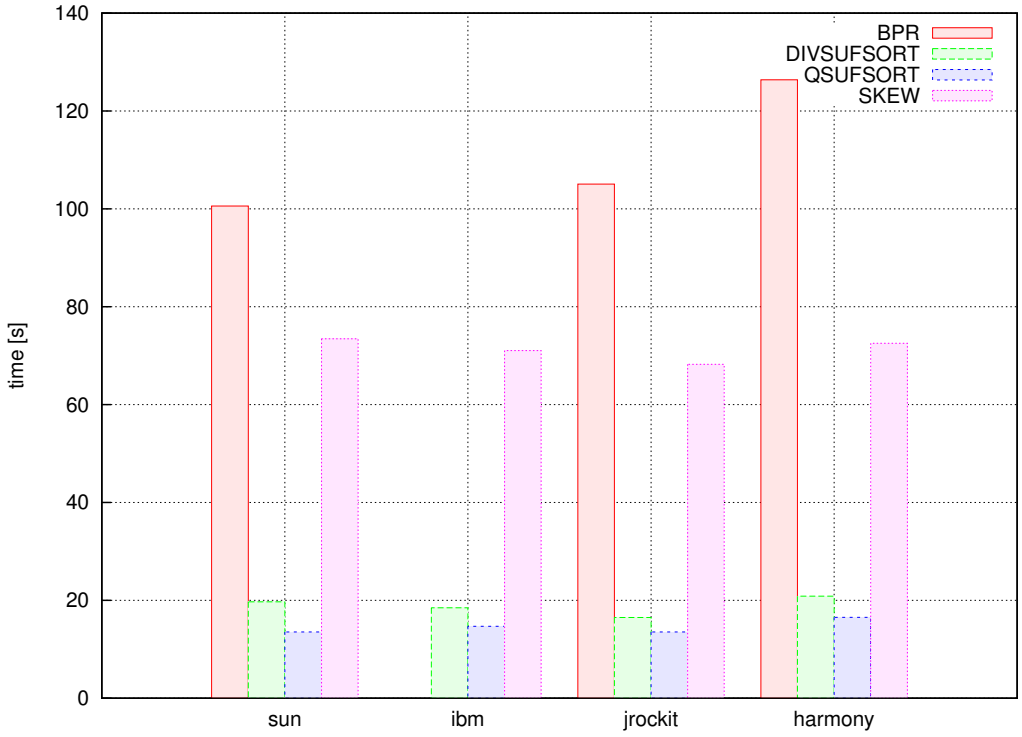
Testowane maszyny wirtualne wypadły bardzo podobnie. Na maszynie `harmony` algorytmy działały wolniej, wyniki pozostałych trzech maszyn są do siebie zbliżone. Spośród tych wyników wyróżnia się jednak wynik testu algorytmu *divsufsort* na korpusie Manziniego przeprowadzonego na maszynie wirtualnej `jrockit` – dwukrotnie gorszy niż na pozostałych maszynach. Wyniki w tabeli A.4 pokazują wolniejsze działania algorytmu na każdym z plików wejściowych. Analiza zapisów przebiegu działania algorytmu wykazała bardzo duże odchylenie standardowe czasu działania algorytmu, co sugeruje wpływ maszyny wirtualnej (*JIT*, *garbage collector*) lub jakiś inny proces obliczeniowy działające w tle i zaburzający pomiar czasu.

	bpr	*divsufsort*	*qsufsort*	*skew*
`sun`	100.58	19.70	13.55	73.44
`ibm`	—	18.49	14.71	71.02
`jrockit`	105.05	16.49	13.55	68.24
`harmony`	126.39	20.85	16.51	72.53

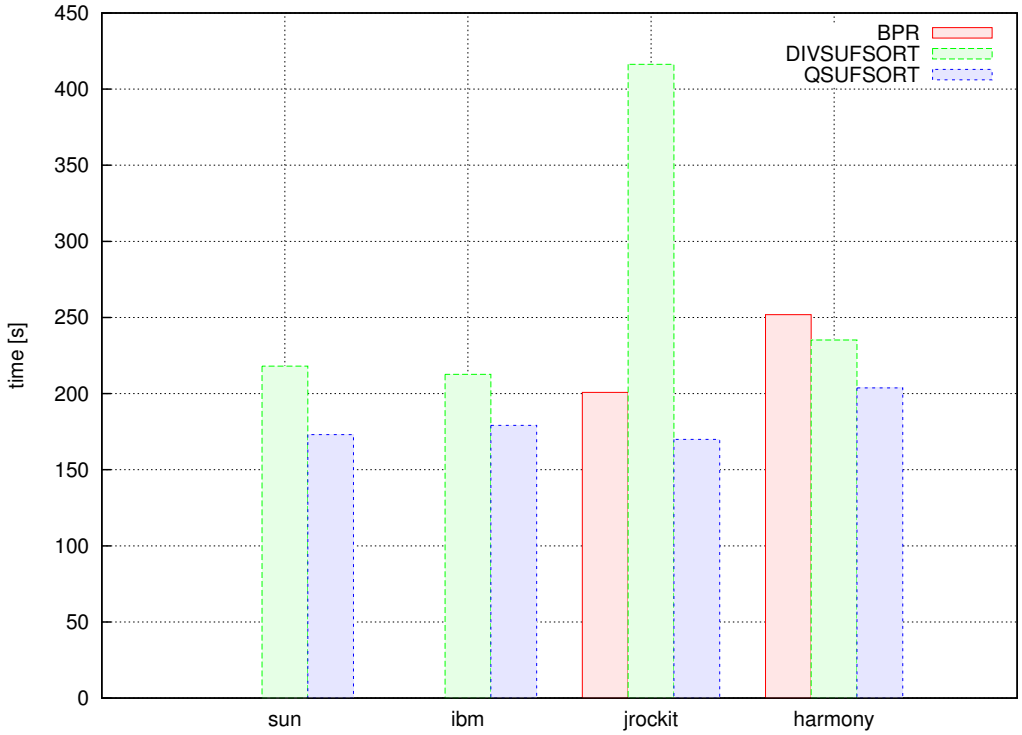
Tablica 5.5: Czasy działania algorytmów na korpusie `Gauntlet` na różnych maszynach wirtualnych. Podane wartości wyrażone są w sekundach.

	bpr	*divsufsort*	*qsufsort*
`sun`	—	218.09	173.10
`ibm`	—	212.62	179.12
`jrockit`	200.84	416.30	170.00
`harmony`	251.82	235.16	203.76

Tablica 5.6: Czasy działania algorytmów na korpusie Giovanniego Manziniego na różnych maszynach wirtualnych. Podane wartości wyrażone są w sekundach.



Rysunek 5.12: Porównanie czasu działania algorytmów na korpusie `Gauntlet` na różnych maszynach wirtualnych.



Rysunek 5.13: Porównanie czasu działania algorytmów na korpusie Giovanniego Manziniego na różnych maszynach wirtualnych.

5.4 Analiza wyników

Najlepszym spośród zaimplementowanych algorytmów okazał się algorytm *qsufsort*. Cechuje się on odpornością na specyficzne typy danych wejściowych, nie zależy w dużym stopniu ani od rozmiaru alfabetu sekwencji wejściowej ani od wartości średniego *lcp*. Dodatkowym atutem jest brak użycia dodatkowej pamięci ponad tą wymaganą do przechowania wejścia i wyjścia algorytmu (oraz stosu zużytego podczas rekurencyjnego sortowania). Na drugim miejscu znalazł się algorytm *divsufsort* ustępujący zwycięzcy zarówno pod względem czasu działania, jak i zużycia pamięci. Pozostałe algorytmy wypadły dużo gorzej niż wymieniona dwójka.

Co ciekawe, publikowane w internecie wyniki testów wydajnościowych implementacji algorytmów w języku `C++` [F, E] dają nieco inny obraz rankingu tych algorytmów. Najlepsze wyniki uzyskują tam implementacje algorytmu *improved two-stage*, czyli *archon*, *divsufsort* i *msufsort*. Algorytm *qsufsort* uzyskuje dobre, choć wyraźnie gorsze wyniki. Wytłumaczenie przyczyn tego faktu leży zapewne w różnicach między językami `C++` a `Java` (oraz działaniem kodu natywnego i kodu uruchamianego na maszynie wirtualnej). Algorytm *qsufsort* jest prostym algorytmem, zarówno koncepcyjnie jak i implementacyjnie. Przepisanie go z `C++` do `Javy` nie było trudne. Algorytm *divsufsort* jest dużo bardziej złożony (jego implementacja w języku `Java` jest o ponad 2000 linii kodu dłuższa od *qsufsort*). Z powodu wykorzystywania wskaźników oraz instrukcji `#define` w oryginalnej implementacji, kod w języku `Java` różni się od oryginału w wielu miejscach. Wydaje się, że dominujące dla szybkości działania różnice to:

- allokacja złożonych struktur danych (i tablic lokalnych) następuję w języku `Java` na stercie, a nie na stosie; oprócz samego narzutu allokacji dochodzi tu również koszt procesu oczyszczania pamięci (*garbage collector*);
- wskaźniki z języków niskopoziomowych muszą być modelowane jako indeksy nad tablicami typów prostych, co w języku `Java` wiąże się z dodatkowym narzutem sprawdzenia czy indeks nie przekracza rozmiaru tablicy;
- rozmiar kodu natywnego generowanego dynamicznie w języku `Java` prawdopodobnie przekraczał wielokrotnie rozmiar kodu skompilowanego w języku `C`, powodując gorsze użytkowanie pamięci podręcznej procesora.

Algorytm *skew* jako jedyny spośród testowanych algorytmów posiadał liniową teoretyczną złożoność obliczeniową. W testach wydajnościowych wypadł jednak bardzo słabo, zarówno w testach implementacji w języku `Java`, jak i w języku `C++`. Przyczyną takiego zachowania algorytmu jest jego bardzo duża złożoność pamięciowa zwiększana dodatkowo przez rekurencyjne wywoływanie algorytmu. Ponadto, rekurencja wiąże się z dodatkowymi kosztami związanymi z obsługą stosu. Należy również pamiętać, że podane złożoności pozostałych algorytmów są wartościami pesymistycznymi, które są rzadko osiągane w praktyce.

Rozdział 6

Podsumowanie i kierunki dalszego rozwoju

Nadrzędnym celem niniejszej pracy było napisanie (w formie biblioteki) implementacji wybranych algorytmów tworzenia tablic sufiksów w celu analizy ich efektywności i zachowania w języku wysokiego poziomu, jakim jest język `Java`. Zadanie to obejmowało zapoznanie się z dostępną literaturą poświęconą tematyce tablic sufiksów, wybór najciekawszych algorytmów oraz ich opisanie. Dodatkowo, w wyniku pracy powinna była powstać klasyfikacja algorytmów tworzenia tablic sufiksów.

Największym wyzwaniem podczas tworzenia pracy było znalezienie najlepszych spośród algorytmów tworzenia tablic sufiksów. Bardzo pomocne były w tym opracowania zawierające zestawienia algorytmów [23, 24] oraz publikowane w internecie wyniki testów wydajnościowych [F, E]. Pewnym problemem podczas implementacji algorytmów było tłumaczenie do języka `Java` takich konstrukcji języka `C++`, jak instrukcje `define`, wskaźniki oraz inne polecenia preprocesora.

Podsumowując, należy stwierdzić, że wszystkie zakładane cele zostały zrealizowane. Przegląd literatury, opisy algorytmów oraz ich klasyfikacja znalazły się w tekście tej pracy, podobnie jak wyniki testów wydajnościowych powstałej implementacji. Według naszej wiedzy niniejsza praca jest pierwszą publikacją w języku polskim poświęconą w całości tematyce algorytmów tworzenia tablic sufiksów. Zaimplementowana biblioteka programowa jest zaś na dzień dzisiejszy jedyną taką pozycją dedykowaną algorytmom tworzenia tablic sufiksów napisaną w języku `Java`. Kod źródłowy biblioteki jest udostępniony na licencji BSD, można go pobrać ze strony projektu: `http://www.jsuffixarrays.org`.

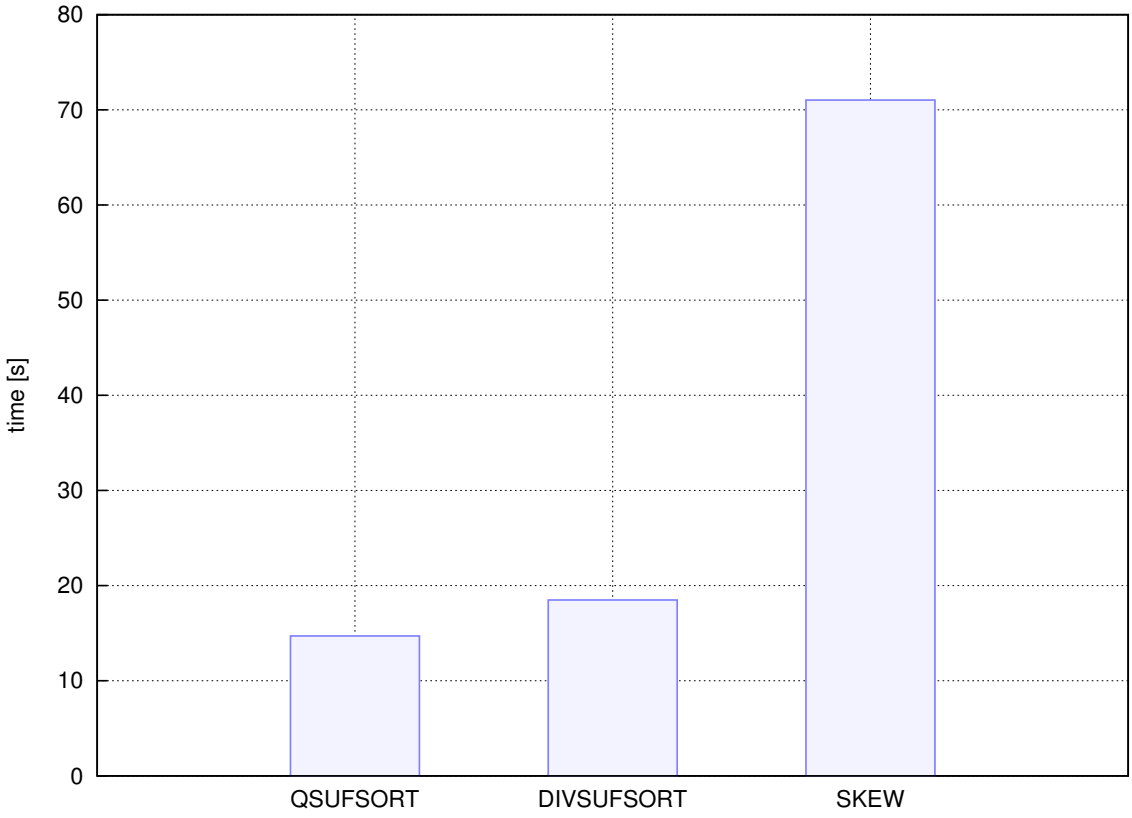
Przewidywany dalszy rozwój biblioteki zakłada opracowanie algorytmu dobierającego najlepszą metodę tworzenia tablic sufiksów na podstawie charakterystyki danych wejściowych (rozmiaru alfabetu i rozkładu symboli w danych wejściowych). Interesującym kierunkiem rozwoju biblioteki jest również stworzenie własnej implementacji algorytmu *improved two-stage*, dostosowanej do specyfiki języka `Java`.

Dodatek A

Wyniki dla pozostałych maszyn wirtualnych

	bpr	*divsufsort*	*qsufsort*	*skew*
`abac`	—	0.01	**0.01**	0.03
`abba`	4.13	2.76	**2.51**	11.71
`book1x20`	**3.48**	3.76	3.67	23.45
`fib_s14930352`	10.47	5.48	**3.59**	14.46
`fss10`	5.73	4.40	**2.94**	12.38
`fss9`	1.06	0.68	**0.66**	2.11
`houston`	—	**0.20**	0.47	1.23
`paper5x80`	0.18	0.12	**0.07**	0.43
`test1`	2.39	0.37	**0.21**	2.16
`test2`	0.86	0.32	**0.21**	2.10
`test3`	71.43	0.38	**0.38**	0.95
Total		18.49	**14.71**	71.02

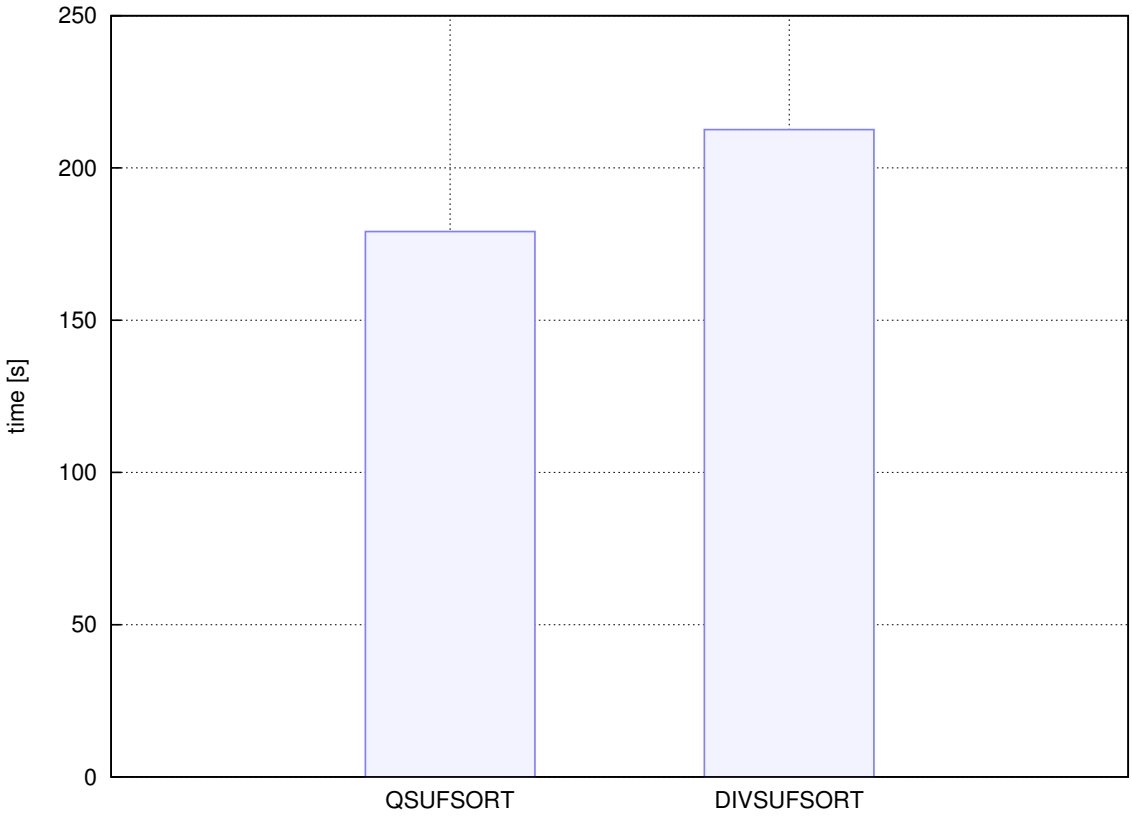
Tablica A.1: Czas działania algorytmów na plikach z korpusu `The Gauntlet` dla maszyny wirtualnej `ibm`.



Rysunek A.1: Sumaryczny czas działania algorytmów na plikach z korpusu `The Gauntlet` dla maszyny wirtualnej `ibm`.

	bpr	*divsufsort*	*qsufsort*	*skew*
`chr22.dna`	**7.31**	8.54	8.15	60.73
`etext99`	28.21	29.20	**26.35**	—
`gcc-3.0.tar`	—	17.25	**14.81**	—
`howto`	8.44	10.33	**7.94**	79.66
`jdk13c`	16.44	14.74	**12.36**	—
`linux-2.4.5.tar`	25.60	24.09	**20.30**	—
`rctail96`	30.55	28.08	**24.60**	—
`rfc`	28.93	25.92	**24.43**	—
`sprot34.dat`	28.55	28.94	**22.89**	—
`w3c2`	24.87	25.55	**17.29**	—
Total		212.62	**179.12**	

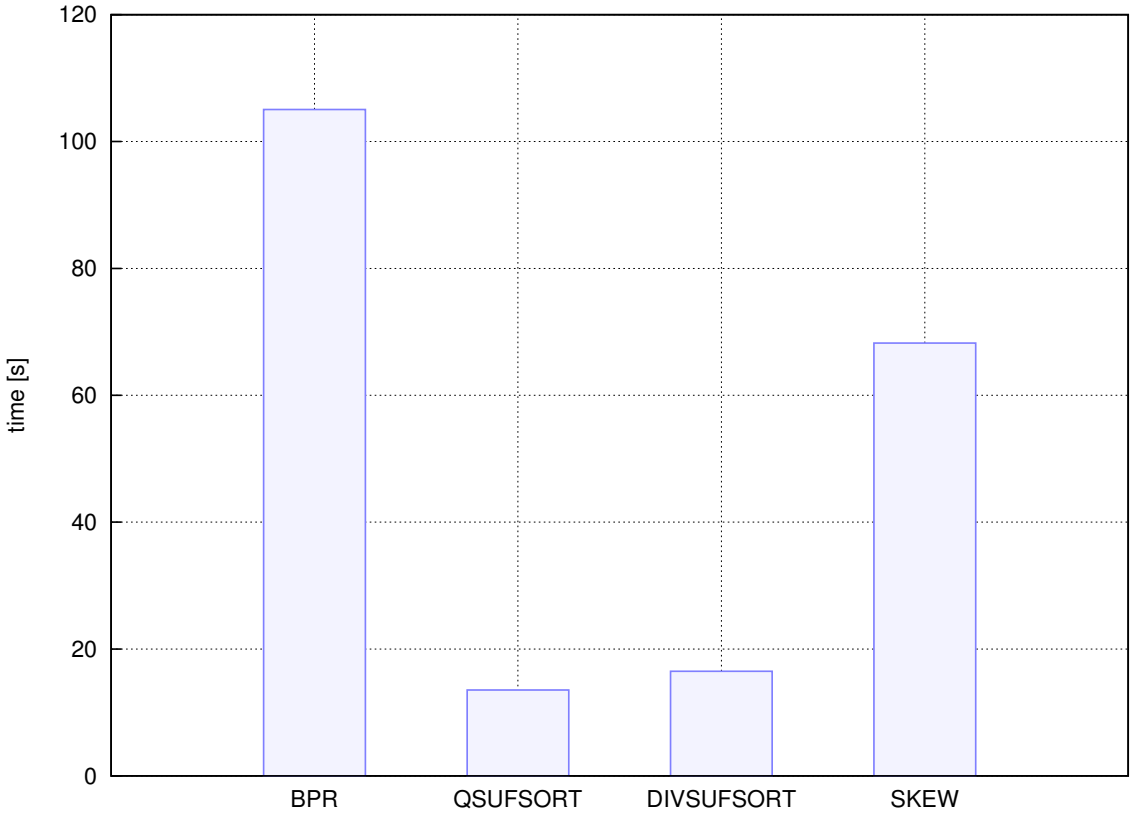
Tablica A.2: Czas działania algorytmów na plikach z korpusu Giovanniego Manziniego dla maszyny wirtualnej `ibm`.



Rysunek A.2: Sumaryczny czas działania algorytmów na plikach z korpusu Giovanniego Manziniego dla maszyny wirtualnej `ibm`.

	bpr	*divsufsort*	*qsufsort*	*skew*
`abac`	1.04	0.01	**0.01**	0.05
`abba`	4.23	**2.19**	2.26	11.23
`book1x20`	**3.14**	3.29	3.40	22.52
`fib_s14930352`	10.05	4.94	**3.32**	14.09
`fss10`	5.36	3.88	**2.69**	12.34
`fss9`	1.05	0.63	**0.58**	1.89
`houston`	2.63	**0.24**	0.44	0.96
`paper5x80`	0.18	0.15	**0.08**	0.42
`test1`	2.26	0.41	**0.20**	1.91
`test2`	0.71	0.34	**0.20**	1.91
`test3`	74.40	0.42	**0.37**	0.91
Total	105.05	16.49	**13.55**	68.24

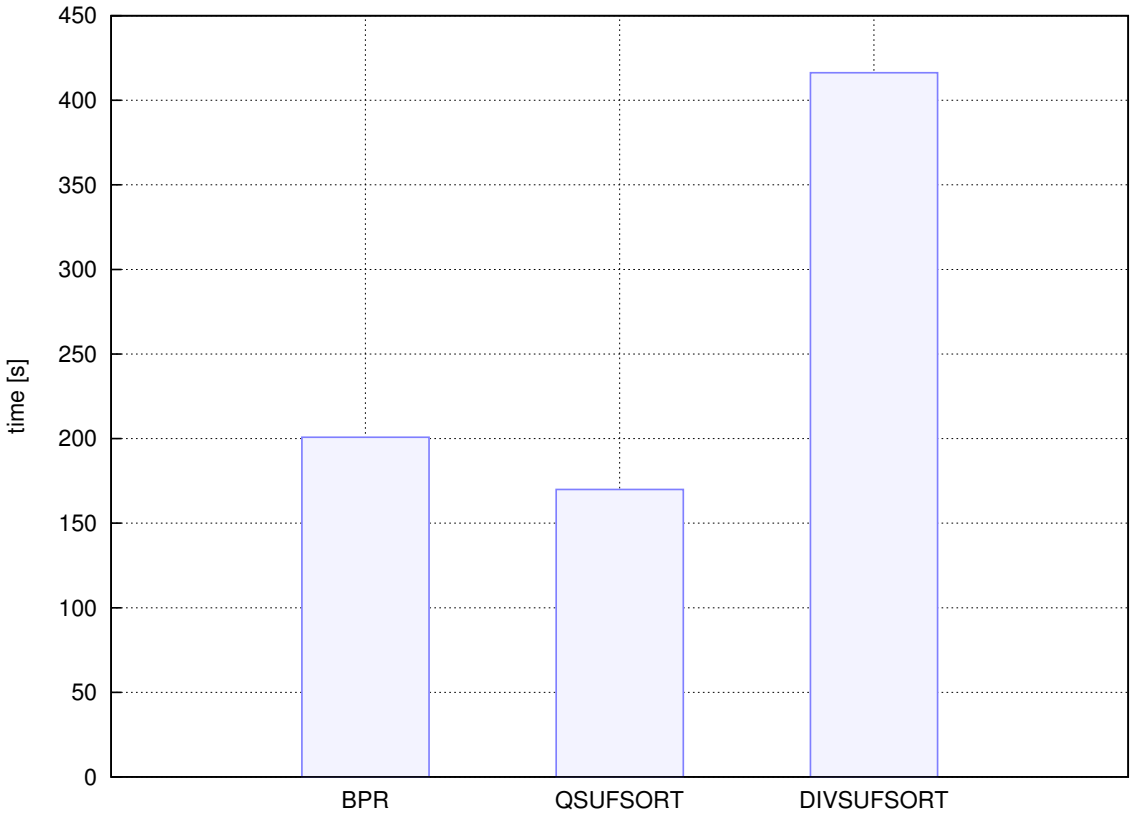
Tablica A.3: Czas działania algorytmów na plikach z korpusu `The Gauntlet` dla maszyny wirtualnej `jrockit`.



Rysunek A.3: Sumaryczny czas działania algorytmów na plikach z korpusu `The Gauntlet` dla maszyny wirtualnej `jrockit`.

	bpr	*divsufsort*	*qsufsort*	*skew*
`chr22.dna`	**6.22**	7.62	7.29	59.27
`etext99`	25.38	59.93	**25.21**	—
`gcc-3.0.tar`	19.56	37.32	**14.07**	—
`howto`	7.43	8.05	**7.20**	78.05
`jdk13c`	14.50	24.81	**11.44**	—
`linux-2.4.5.tar`	23.68	57.76	**19.53**	—
`rctail96`	28.61	56.08	**23.49**	—
`rfc`	26.42	53.69	**23.39**	—
`sprot34.dat`	26.51	65.53	**21.90**	—
`w3c2`	22.53	45.49	**16.43**	—
Total	200.84	416.30	**169.95**	

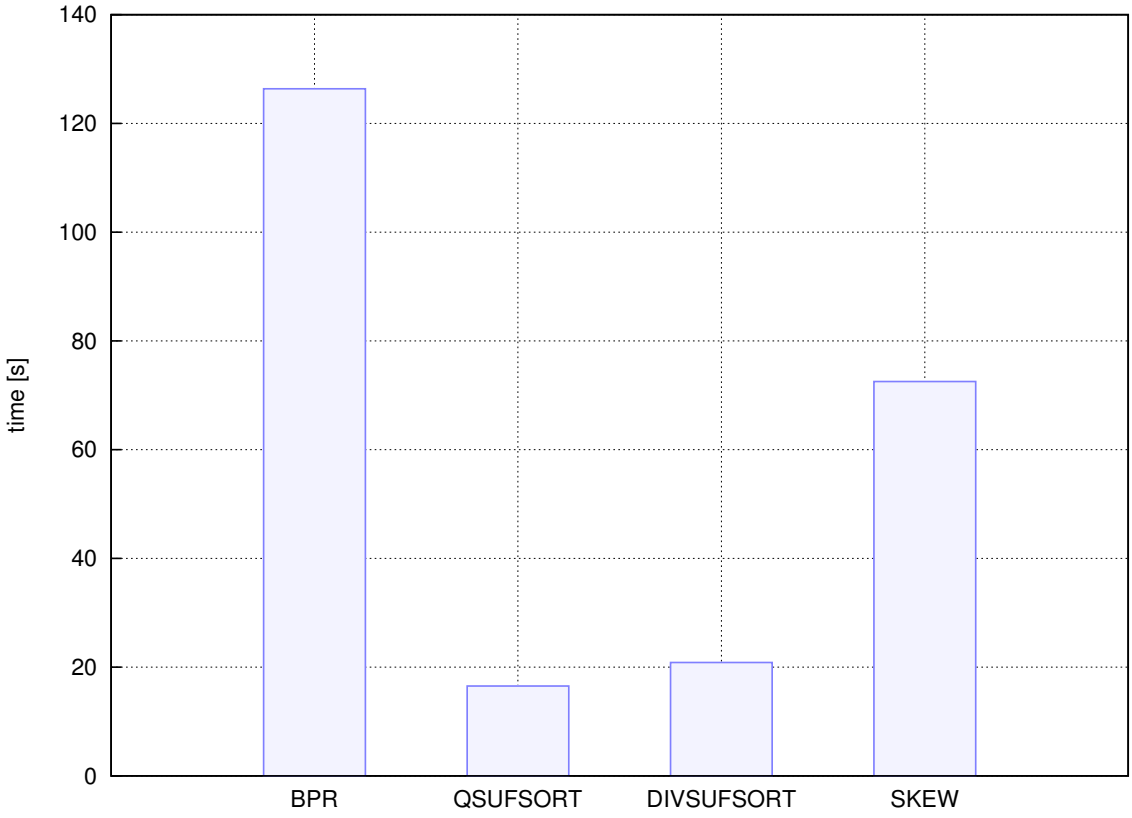
Tablica A.4: Czas działania algorytmów na plikach z korpusu Giovanniego Manziniego dla maszyny wirtualnej `jrockit`.



Rysunek A.4: Sumaryczny czas działania algorytmów na plikach z korpusu Giovanniego Manziniego dla maszyny wirtualnej `jrockit`.

	bpr	*divsufsort*	*qsufsort*	*skew*
`abac`	1.83	0.05	**0.03**	0.06
`abba`	4.41	**2.82**	2.85	11.62
`book1x20`	**3.85**	4.36	4.24	23.35
`fib_s14930352`	11.77	6.18	**4.15**	15.18
`fss10`	6.50	4.87	**3.04**	12.26
`fss9`	1.31	0.80	**0.69**	2.32
`houston`	4.05	**0.23**	0.53	1.45
`paper5x80`	0.19	0.15	**0.07**	0.53
`test1`	3.14	0.49	**0.25**	2.27
`test2`	0.96	0.36	**0.23**	2.28
`test3`	88.37	0.54	**0.42**	1.22
Total	126.39	20.85	**16.51**	72.53

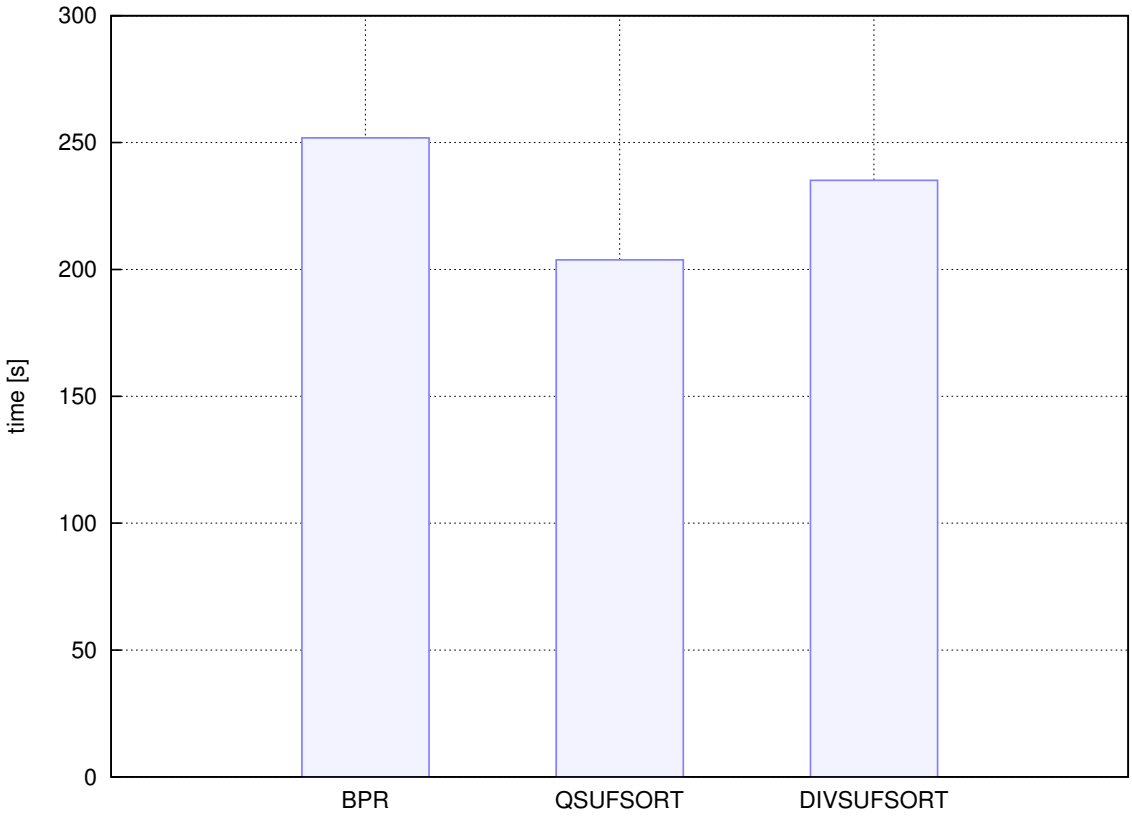
Tablica A.5: Czas działania algorytmów na plikach z korpusu `The Gauntlet` dla maszyny wirtualnej `harmony`.



Rysunek A.5: Sumaryczny czas działania algorytmów na plikach z korpusu `The Gauntlet` dla maszyny wirtualnej `harmony`.

	bpr	*divsufsort*	*qsufsort*	*skew*
`chr22.dna`	**7.93**	9.15	8.38	63.94
`etext99`	32.45	32.38	**29.48**	—
`gcc-3.0.tar`	25.45	19.74	**16.61**	—
`howto`	9.71	9.92	**8.46**	82.36
`jdk13c`	18.77	16.84	**13.59**	—
`linux-2.4.5.tar`	29.32	27.61	**24.55**	—
`rctail96`	35.18	32.71	**27.64**	—
`rfc`	32.69	29.21	**28.99**	—
`sprot34.dat`	32.81	31.97	**25.60**	—
`w3c2`	27.51	25.63	**20.47**	—
Total	251.82	235.16	**203.76**	

Tablica A.6: Czas działania algorytmów na plikach z korpusu Giovanniego Manziniego dla maszyny wirtualnej `harmony`.



Rysunek A.6: Sumaryczny czas działania algorytmów na plikach z korpusu Giovanniego Manziniego dla maszyny wirtualnej `harmony`.

Literatura

[1] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Software Practice Experience*, 23(11):1249–1265, 1993.

[2] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[3] J. Daugman. How iris recognition works. *IEEE Trans. Cir. and Sys. for Video Technol.*, 14(1):21–30, January 2004.

[4] Paolo Ferragina and Roberto Grossi. The string b-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46:236–280, 1998.

[5] David J. Field. Relations between the statistics of natural images and the response properties of cortical cells. *J. Opt. Soc. Am. A*, 4(12):2379–2394, Dec 1987.

[6] Robert Giegerich and Stefan Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[7] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, styczeń 1997.

[8] Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. In *SPIRE '99: Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, page 81, Washington, DC, USA, 1999. IEEE Computer Society.

[9] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 13th International Conference on Automata, Languages and Programming*. Springer, 2003.

[10] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 125–136, New York, NY, USA, 1972. ACM.

[11] Toru Kasai, Hiroki Aftmura, and Setsue Arikawa. Efficient substring traversal with suffix arrays, 2001.

[12] Donald E. Knuth. *Sztuka programowania (Tom 3, Sortowanie i wyszukiwanie)*. Wydawnictwa Naukowo Techniczne, Warszawa, 2002.

[13] Jesper N. Larsson. *Structures of String Matching and Data Compression*. PhD thesis, Department of Comp. Science, Lund University, 1999.

[14] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, maj 1999.

[15] Michael A. Maniscalco and Simon J. Puglisi. An efficient, versatile approach to suffix sorting. *J. Exp. Algorithmics*, 12, 2007.

[16] Giovanni Manzini. Two space saving tricks for linear time lcp array computation. *Algorithm Theory - SWAT 2004*, pages 372–383, 2004.

[17] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm (extended abstract), 2004.

[18] Libor Masek. Recognition of human iris patterns for biometric indetification. The University of Western Australia, 2003.

[19] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, kwiecień 1976.

[20] Yuta Mori. Short description of improved two-stage suffix sorting algorithm. `http://homepage3.nifty.com/wpage/software/itssort.txt`, 2005.

[21] A. V. Oppenheim and J. S. Lim. The importance of phase in signals. *Proceedings of the IEEE*, 69(5):529–541, May 1981.

[22] H. Proenca and L. A. Alexandre. Iris recognition: An analysis of the aliasing problem in the iris normalization stage. In *2006 International Conference on Computational Intelligence and Security*, volume 2, pages 1771–1774, Nov 2006.

[23] Simon J. Puglisi, William. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), 2007.

[24] Klaus-Bernd Schürmann. *Suffix Arrays in Theory and Practice*. PhD thesis, Faculty of Technology of Bielefeld University, Germany, 2007.

[25] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. *Softw. Pract. Exper.*, 37(3):309–329, 2007.

[26] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[27] Peter Weiner. Linear pattern matching algorithms. In *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

Zasoby internetowe

[A] Project Gutenberg.
http://www.gutenberg.net

[B] The AspectJ Project.
http://www.eclipse.org/aspectj/

[C] The Gauntlet (Universal Robustness Corpus).
http://www.michael-maniscalco.com/testset/gauntlet/

[D] Manzini's Large Corpus.
http://www.mfn.unipmn.it/~manzini/lightweight/corpus/

[E] Yuta Mori, Suffix Array Construction Benchmark
http://homepage3.nifty.com/wpage/benchmark/index.html

[F] Michael Maniscalco, The MSufSort Algorithm.
http://www.michael-maniscalco.com/msufsort.htm

[G] Peter Sanders, Skew algorithm.
http://www.mpi-inf.mpg.de/~sanders/programs/suffix/

[H] M. Douglas McIlroy, ssort.c
http://cm.bell-labs.com/cm/who/doug/source.html

[I] Dmitry A. Malyshev, Archon
http://kvgate.com/index.php?root/comp/arch/archon/

[J] Giovanni Manzini, A Lightweight Suffix Array and BWT Construction Algorithm
http://web.unipmn.it/~manzini/lightweight/ds.tgz

[K] Yuta Mori, libdivsufsort project homepage.
http://code.google.com/p/libdivsufsort/

[L] N.Jesper Larsson, qsufsort.c
http://www.larsson.dogma.net/qsufsort.c

[M] Klaus-Bernd Schürmann i Jens Stoye, bpr downloadpage.
http://bibiserv.techfak.uni-bielefeld.de/download/tools/bpr.html

Instytut Automatyki, Robotyki i Inżynierii Informatycznej, Wydział Elektryczny
Politechnika Poznańska

Skład przy użyciu systemu LaTeX.

BibTeX:

```
@mastersthesis{ jlamprecht-masterthesis,
    author = "Jakub Lamprecht",
    title = "{Badawczy system rozpoznawania wykorzystujący obraz tęczówki oka}",
    school = "Poznan University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2019",
}
```