Politechnika Łódzka
Instytut Informatyki Stosowanej

Techniki kompresji i równoległości bitowej w wybranych problemach wyszukiwania wzorców w tekście

Rozprawa doktorska

STRESZCZENIE

Robert Susik

Promotor: dr hab. Szymon Grabowski, prof. PŁ

27 lutego 2018

1 Wprowadzenie

1.1 Wstęp

Niniejsza praca doktorska skupia się na zagadnieniach ściśle związanych z działem algorytmiki określanego mianem stringologii (ang. *stringology*). Termin ten odnosi się do zagadnień i problemów związanych z przetwarzaniem ciągów tekstowych w różnej formie. Dział ten jest szczególnie istotny ze względu na swoje praktyczne zastosowanie w otaczającej nas rzeczywistości. Tekst jest podstawowym nośnikiem informacji cyfrowej, który tak mocno spopularyzował się, że obecnie trudno byłoby sobie wyobrazić istnienie cyfrowego świata pozbawionego danych w postaci ciągów tekstowych. A ponieważ tak wiele danych jest dostępnych w postaci cyfrowej (np. w sieci Internet), naturalnym problemem, z którym przychodzi spotkać się każdemu z nas jest szukanie (wyszukiwanie) informacji. W niniejszej rozprawie omawiane są głównie algorytmy skupiające się na rozwiązaniu tego problemu, które można podzielić na dwie kategorie:

- Algorytmy online – wyszukują wzorzec w zbiorze danych bez konieczności odczytania całego zbioru przed rozpoczęciem szukania informacji (co pozwala na odnajdywanie wzorców w strumieniu danych).

- Algorytmy offline – przed rozpoczęciem wyszukiwania informacji wymagają wcześniej przetworzenia zbioru danych. Algorytmy te często budują stratną lub bezstratną reprezentację zbioru danych umożliwiającą szybsze odszukanie wzorca w zbiorze. Technika ta jest nazywana indeksowaniem.

Problem szukania ciągów tekstowych można podzielić ze względu na kryteria, jak na przykład liczba szukanych wzorców (jeden, czy wiele), dokładność (czy szukany jest dokładnie taki sam wzorzec, czy przybliżony), czy specyfika tekstu, w którym przeprowadzane jest wyszukiwanie (np. kod `DNA`). Tekst w którym szukany jest wzorzec może być zapisany w formie skompresowanej (format pozwalający na ograniczenie zajętości miejsca w pamięci) lub nieskompresowanej. W pierwszym przypadku, w przeciwieństwie do drugiego, konieczna jest dekompresja (przywrócenie oryginalnej formy danych – przeciwieństwo kompresji) i dopiero uruchomienie algorytmu wyszukującego tekst lub

zastosowanie odpowiedniego algorytmu szukającego tekst (często wspólnie z odpowiednim formatem kompresji tekstu) bezpośrednio w tekście skompresowanym, co pozwala zaoszczędzić czas na dekompresji tekstu.

Problemy, dla których zaproponowano rozwiązania to wyszukiwanie:

- dokładne wielu wzorców,
- w tekście skompresowanym,
- przybliżone,
- wzorców obróconych,
- z wykorzystaniem struktury indeksującej.

Rozwiązania wyżej wymienionych problemów łączy zastosowanie techniki równoległości bitowej. Ostatnia pozycja natomiast wyróżnia się zastosowaniem lekkiej struktury indeksującej tekst, która oprócz prostoty implementacji pozwala na szybkie budowanie indeksu (szybsze niż w przypadku tradycyjnych algorytmów offline) oraz szybkie wyszukiwanie informacji (znacznie szybsze niż w przypadku algorytmów online).

Wszystkie powyżej wymienione rozwiązania skupiają się na udowodnieniu założonych tez, które sformułowano następująco:

1. Techniki równoległości bitowej umożliwiają przyspieszenie wyszukiwania wielu wzorców, szukania przybliżonego, wyszukiwania w tekście skompresowanym oraz indeksowania.
2. Przyspieszone wyszukiwanie wzorców oraz budowa indeksu o korzystnych relacjach użytkowych jest możliwa dzięki zastosowaniu odpowiednich algorytmów kompresji tekstu.

Każdy z zaproponowanych algorytmów został wnikliwie przetestowany, a wyniki porównane z konkurencyjnymi rozwiązaniami, aby potwierdzić słuszność zastosowanego podejścia. Zdecydowana większość z umieszczonych w rozprawie wyników została opublikowana, a kody źródłowe udostępnione publicznie. Wymienione powyżej rozwiązania zostały opublikowane w następujących artykułach:

- **R. Susik**, Sz. Grabowski, and S. Deorowicz. Fast and Simple Circular Pattern Matching. In *Man-Machine Interactions 3*, pages 537–544. Springer, 2014
- **R. Susik**, Sz. Grabowski, and K. Fredriksson. Multiple Pattern Matching Revisited. In *Proceedings of the Prague Stringology Conference*, pages 59–70, 2014
- **R. Susik** and Sz. Grabowski. Engineering the Counting Filter for String Matching Algorithms. In *Algorithms, Networking and Sensing for Data Processing, Mobile Computing and Applications*, pages 125–140. Łódź University of Technology Press, 2016
- Sz. Grabowski, **R. Susik**, and M. Raniszewski. A Bloom Filter based Semi-Index on q-grams. *Software: Practice and Experience*, 47(6):799–811, 2017
- **R. Susik**. Applying a q-gram based Multiple String Matching Algorithm for Approximate Matching. *Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska*, 7(3):47–50, 2017

1.2 Wyszukiwanie jednego wzorca

Problem wyszukiwania jednego wzorca można zdefiniować następująco: dla danego ciągu tekstowego $T[0 \dots n-1]$ oraz wzorca $P[0 \dots m-1]$ (gdzie $n \geqslant m$), składających się z elementów nazywanych dalej znakami lub symbolami, należącymi do wspólnego alfabetu $\Sigma \in \{0, 1, \dots, \sigma - 1\}$ o rozmiarze σ, znaleźć wszystkie i $(0 \leqslant i < n)$, takie że $P[0 \dots m-1] = T[i \dots i+m-1]$. Yao [Yao79] udowodnił, że złożoność czasowa rozwiązania tego problemu posiada asymptotyczną granicę dolną, która w przypadku średnim wynosi $\Omega(n \log_\sigma(m)/m)$.

1.3 Wyszukiwanie wielu wzorców

Problem wyszukiwania wielu wzorców można zdefiniować następująco: dla danego ciągu tekstowego T o długości n oraz zbioru wzorców $P \in P_0, P_1, \dots, P_{r-1}$, każdy jednakowej bądź różnej długości $M \in \{M_0, M_1, \dots, M_{r-1}\}$ (gdzie $n \geqslant \max_{x=0}^{r-1} M_x$) posiadających wspólny alfabet $\Sigma \in \{0, 1, \dots, \sigma - 1\}$ o rozmiarze σ, znaleźć i $(0 \leqslant i < n)$, gdzie

$P_x[0 \ldots M_x{-}1] = T[i \ldots i{+}M_x{-}1]$, dla $0 \leqslant x \leqslant r{-}1$. Naiwnym rozwiązaniem opisanego problemu jest r-krotne użycie jednego z algorytmów wyszukiwania jedynego wzorca, osobno dla każdego z r wzorców. Taki narzut czasowy nie zawsze jest dopuszczalny. Navarro i Fredriksson [NF04] udowodnili, że można określić asymptotyczną granicę dolną dla algorytmów rozwiązujących ten problem analogicznie do problemu szukania jednego wzorca i granica ta wynosi $\Omega(n \log_\sigma(rm)/m)$ w przypadku średnim.

1.4 Wyszukiwanie w tekście skompresowanym

W obliczu rosnących ilości danych tekstowych opracowano wiele metod ich kompresji pozwalających zredukować ilość zajmowanego miejsca oraz ograniczyć transfer, a jednocześnie skrócić czas niezbędny do ich przesłania. Stopień kompresji *cr* jest wyrażany jako stosunek tekstu nieskompresowanego do tekstu skompresowanego *cr = rozmiar tekstu nieskompresowanego/rozmiar tekstu skompresowanego*. W niniejszej pracy wykorzystano kompresję tekstu w postaci słów kodowych. Słowo kodowe to ustalony ciąg bitów (lub bajtów) reprezentujący pewien symbol bądź słowo tekstowe. Szczególnym sposobem zapisu słów kodowych są kody bajtowe. Istnieje wiele algorytmów stosujących taką formę zapisu, a w pracy tej wykorzystano algorytm kodowania ETDC. Dodatkowo nieco inną formą kompresji jest filtr Blooma, który zastosowano w opracowanym rozwiązaniu semiindeksu – BFSI. Jest to w pewnym sensie stratna forma reprezentacji tekstu umożliwiająca szybsze wyszukiwanie.

1.5 Wyszukiwanie przybliżone

Problem przybliżonego wyszukiwania wzorców (ang. *approximate pattern matching*) można zdefiniować następująco: dany jest tekst $T[0 \ldots n{-}1]$ oraz wzorzec $P[0 \ldots m{-}1]$, gdzie $m < n$. Znaki tekstu T i wzorca P należą do wspólnego alfabetu. Należy znaleźć wszystkie i takie, że $\mathrm{ed}(T[i \ldots i{+}m{-}1], P[0 \ldots m{-}1]) \leqslant k$, gdzie $\mathrm{ed}(T_1, T_2)$, to funkcja zwracająca odległość edycyjną (ang. *edit distance*), czyli liczbę operacji jaką należy wykonać, aby przekształcić $T_1 \rightarrow T_2$. Wśród odległości edycyjnych wymienionych w rozprawie są:

- Odległość Hamminga – jest miarą mówiącą na ilu miejscach dwa ciągi tekstowe T_1 i T_2 o takiej samej długości n różnią się na tych samych pozycjach. Dla zbioru pozycji $A \subseteq B$, gdzie $B \in \{0, \ldots, n-1\}$ oraz $\forall x \in A : T_1[x] \neq T_2[x]$, odległość Hamminga wynosi $\mathrm{D}_H(T_1, T_2) = |A|$, co znaczy, że należy wykonać $|A|$ operacji zamiany znaku, aby przekształcić T_1 w T_2.

- Odległość Levenshteina – jest miarą określającą najmniejszą liczbę operacji pozwalających na przekształcenie ciągu $T_1[0 \ldots n-1]$ w ciąg $T_2[0 \ldots m-1]$. Operacje jakie można wykonać to: wstawienie znaku, usunięcie znaku, zamiana znaku.

2 Wyszukiwanie dokładne wielu wzorców

Wyszukiwanie wielu wzorców jest istotnym problemem w dziedzinie informatyki ze względu na mnogość zastosowań, nawet w problemach z pozoru niekoniecznie powiązanych i jest to jeden głównych problemów, na których skupia się niniejsza praca. Kody źródłowe odnośnych algorytmów udostępniono pod adresem `https://github.com/rsusik/mag` oraz `https://github.com/rsusik/magetdc`.

Jednym z pierwszych rozwiązań opracowanych w tej tematyce jest Multipattern Shift-Or, który został zaprezentowany przez Susika i in. [SGD14] w artykule poświęconym tematyce wyszukiwania wzorców obróconych. Zaletą tego rozwiązania jest elastyczność pozwalająca na łatwe zastosowanie klas znaków bez większej ingerencji w kod źródłowy.

Technikę klas znaków zastosowano do reprezentacji wielu wzorców w postaci pojedynczego wzorca P', jednak wymusza to wykonanie weryfikacji w celu potwierdzenia, czy na odnalezionej pozycji jest wzorzec znajdujący się w zbiorze wzorców wyszukiwanych. Podejście to zaadoptowano z algorytmami: Shift-Or (SO), Fast Shift-Or (FSO), Fase Average Optimal Shift-Or (FAOSO), a rozwiązania te nazwano odpowiednio Multipattern Shift-Or (MSO), Multipattern Fast Shift-Or (MFSO) oraz Multipattern Fast Average Optimal Shift-Or (MFAOSO).

Najszybszym i jednocześnie najbardziej złożonym rozwiązaniem jest MFAOSO. Algorytm ten przyjmuje szereg parametrów, dzięki czemu można go zoptymalizować

pod konkretny zestaw wzorców czy alfabet. Złożoność czasowa algorytmu FAOSO w przypadku średnim wynosi $O(n \log_\sigma m/m)$ ([FG09, Podrozdz. 7.2]), co daje czas $O(n \log_\sigma (rm)/m)$ dla MFAOSO (dla $m \leqslant w$).

Multi AOSO on q-Grams (MAG) jest rozwinięciem wcześniej opisanych algorytmów (MFSO oraz MFAOSO) wzbogaconym o szereg usprawnień, takich jak technika q-gramów, mapowanie alfabetu (kilka wariantów) oraz możliwość szukania w tekście skompresowanym. Technika q-gramów jest znana w stringologii i znalazła wcześniej zastosowanie w problemach wyszukiwania dokładnego, przybliżonego (pojedynczego i wielu wzorców), a także w technikach filtracji.

Termin q-gram oznacza ciąg tekstowy o długości q. Wyróżnia się dwa typy q-gramów: nakładające się (ang. *overlapping*) oraz nienakładające się (ang. *non-overlapping*). Nakładające się to takie, gdzie dla danego ciągu $S[0 \ldots n-1]$ o długości n można utworzyć $n-q+1$ q-gramów, czyli $\{S[0 \ldots q-1], S[1 \ldots q], \ldots, S[n-q \ldots n-1]\}$. Nienakładające się to takie, gdzie można utworzyć $\lfloor n/q \rfloor$ q-gramów, czyli $\{S[0 \ldots q-1], S[q \ldots 2q-1], \ldots, S[(\lfloor n/q \rfloor - 1)q \ldots \lfloor n/q \rfloor q - 1]\}$.

W algorytmie tym zastosowano numeryczną reprezentację q-grama (superznak), której wartość wynosi $\sum_{i=0}^{q-1} S[i]\sigma^{q-i-1}$. Zastosowanie q-gramów w rozwiązaniu tym przyczynia się do rozszerzenia alfabetu z σ do superalfabetu o rozmiarze σ^q, co dobrze komponuje się z zastosowaniem klas znaków, gdzie nakładające się wzorce mogą powodować wzrost liczby weryfikacji. Rozszerzenie rozmiaru alfabetu znacząco niweluje ten efekt, dzięki czemu wyszukiwanie większej liczby wzorców jest możliwe. Rozmiar utworzonego alfabetu rośnie bardzo szybko wraz ze wzrostem σ lub q, dlatego też konieczne okazało się zastosowanie techniki mapowania (kwantyzacji) alfabetu. Tak skwantyzowany alfabet jest wykorzystywany w procesie wyszukiwania wzorców, natomiast weryfikacja odnalezionych pozycji odbywa się na oryginalnym alfabecie.

W algorytmie tym rozważono szereg metod kwantyzacji alfabetu:

- Mapowanie oparte na histogramie (HAM),
- Mapowanie oparte na histogramie q-gramów (HAMq),
- Mapowanie alfabetu połączone z generacją q-gramów (CAMq), w tym:
  - CAMq(dna) – zoptymalizowany dla alfabetu `DNA`,

– CAMq(opt) – zredukowana liczba operacji.

Rozwiązanie to zostało zaimplementowane w języku C++ i gruntownie przetestowane na trzech zbiorach danych: `English`, `DNA` oraz `Proteins` o rozmiarze 200 MiB pobranych z repozytorium Pizza & Chili.

Zaproponowane rozwiązanie porównano z następującymi algorytmami:

- BNDM on q-grams (BG) [STK06],
- Shift-Or on q-grams (SOG) [STK06],
- BMH on q-grams (HG) [STK06],
- Rabin-Karp, algorytm opierający się na zastosowaniu techniki funkcji mieszkających oraz wyszukiwania binarnego (RK) [STK06],
- Multibom and Multibsom, warianty algorytmu Set Backward Oracle Matching [AR99],
- Succinct Backward DAWG Matching (SBDM) [Fre09].

Wyniki testów w przypadku alfabetu `DNA` okazały się korzystne dla algorytmu SBDM, który osiągnął lepsze rezultaty dla krótkich wzorców ($m \in \{8, 16\}$), jednak dla dłuższych wzorców ($m \in \{32, 64\}$) rozwiązanie MAG okazało się szybsze. Dla zbioru `English` SOG osiągnął najlepszy wynik dla najmniejszego rozmiaru wzorca ($m = 8$), następnie dla $m = 16$ rozwiązania MAG wraz z SBDM okazały się najszybsze, natomiast dla długich wzorców ($m \in \{32, 64\}$) najlepsze wyniki osiągnął algorytm MAG. W przypadku zbioru `Proteins` najlepszym rozwiązaniem niemal dla każdej długości wzorca okazał się `MAG`, osiągając nieznacznie słabsze wyniki od SOG oraz BG dla krótkich wzorców ($m = 8$). Najbardziej porównywalnym pod względem szybkości działania algorytmem do MAG okazał się SBDM, który osiągnął lepsze wyniki dla krótkich wzorców, natomiast dla $m = 8$ na zbiorach `English` oraz `Proteins` najlepszy okazał się algorytm SOG.

2.1 Wyszukiwanie dokładne wielu wzorców w tekście skompresowanym

Rozwiązanie MAG przetestowano także dla problematyki wyszukiwania w tekście skompresowanym. Kompresja tekstu została wykonana za pomocą techniki ETDC. Zastosowanie kompresji pozwala na zmniejszenie czasu oraz zasobów (np. pamięci RAM) koniecznych do przeprowadzania procesu wyszukiwania, ponieważ rozmiar danych, w których przeprowadzany jest proces szukania jest pomniejszony. Podobnie jak w przypadku testów na tekście nieskompresowanym, przetestowano różne metody mapowania alfabetu. Testy przeprowadzono na zbiorze `English`, którego rozmiar po kompresji wyniósł pomiędzy 33% a 35% oryginalnego rozmiaru tekstu (włączając tablicę pomocniczą wykorzystywaną przez algorytm).

Udowodniono, że zastosowanie kodów bajtowych nie tylko pozwala zaoszczędzić miejsce w pamięci, lecz także przyspieszyć szybkość wyszukiwania (w przypadku długich wzorców) w porównaniu do tekstu nieskompresowanego. Rozwiązanie to pozwala na zniwelowanie wpływu rozmiaru słowa maszynowego na wydajność algorytmu, ponieważ kody bajtowe są zazwyczaj znacznie krótsze od oryginalnych ciągów tekstowych (słów wraz z separatorami), a ograniczenie długości wzorca w tym przypadku odnosi się nie do pojedynczych znaków, lecz do długości kodu bajtowego.

MAG okazał się udanym połączeniem znanych technik, które zaowocowało obiecującymi wynikami, pokonując konkurencyjne algorytmy w większości rozważanych przypadków. Szczególnie udane okazało się zastosowanie metod mapowania alfabetu, które umożliwiło wykorzystanie większych q-gramów. Opracowano, zaimplementowano i przetestowano szereg metod kwantyzacji alfabetu, opartą na histogramie (HAM), która wyróżnia się elastycznością, umożliwiając wybór dowolnej wartości σ', połączoną z generacją q-gramów, gdzie wyróżniono kilka wariantów, CAMq podstawowy wariant, CAMq(opt) (zoptymalizowany przez zastąpienie sumowania operacjami bitowego przesunięcia i mnożenia) oraz CAMq(dna) (zoptymalizowany dla alfabetu DNA). Każdy z wariantów CAMq charakteryzuje się tym, że nie ma konieczności wykonywania wstępnego przetwarzania (np. budowania histogramu), ani też nie ma potrzeby sięgania do tablicy mapowania alfabetu, jednak obarczony jest mniejszą elastycznością (ograniczo-

na jest liczba możliwych wartości σ') oraz zależnością od wielkości słowa maszynowego (co ogranicza wielkość q-grama i wartość parametru ℓ). Wykazano, że zaproponowane rozwiązanie jest czasowo subliniowe w przypadku średnim.

3 Redukcja wybranych problemów tekstowych do problemu wyszukiwania dokładnego wielu wzorców

W rozdziale tym przedstawiono zastosowania algorytmu wyszukującego wiele wzorców do innych problemów w dziedzinie stringologii: wyszukiwania wzorców obróconych oraz wyszukiwania przybliżonego z dokładnością do k błędów (odległości Hamminga oraz Levenshteina). Czynność taka nazywana jest redukcją problemu (sprowadzeniem pewnego problemu do innego problemu).

Wyniki zaprezentowane w tym rozdziale zostały opublikowane w [SGD14, Sus17]. Kody źródłowe implementacji zostały udostępnione pod adresami `https://github.com/rsusik/fscpm`, `https://github.com/rsusik/magc` oraz `https://github.com/rsusik/maga`.

3.1 Wyszukiwanie wzorców obróconych

Wyszukiwanie wzorców obróconych (ang. *Circular Pattern Matching*) jest problemem, w którym należy odnaleźć wszystkie rotacje wzorca P w tekście T. Rotacją wzorca nazywamy każdy z m ciągów tekstowych, który rozpoczyna się na pozycji j i kończy na pozycji $j-1$, co można opisać następująco: $P_j^r = P[j] \dots P[m-1]P[0] \dots P[j-1]$. Problem wyszukiwania wzorców obróconych można zdefiniować w następujący sposób: dany jest tekst $T[0 \dots n-1]$ oraz wzorzec $P[0 \dots m-1]$, gdzie $m \leqslant n$, znaleźć wszystkie i takie, że $T[i \dots i+m-1] \in \{P[0 \dots m-1], P[j+1 \dots m-1]P[0 \dots j]\}$, gdzie $0 \leqslant j \leqslant m-2$ oraz $0 \leqslant i \leqslant n-m$. Naiwne rozwiązanie polega na uruchomieniu dowolnego algorytmu wyszukiwania dokładnego dla każdej rotacji. Liczba rotacji wzorca (włącznie ze wzorcem) wynosi m (długość wzorca), co oznacza, że szukanie wzorca obróconego zajęłoby m razy więcej czasu niż szukanie pojedynczego wzorca. Wyniki zaprezentowane w tym podrozdziale zostały opublikowane w 2013 roku przez Susika i in. [SGD14].

Zaproponowano rozwiązanie opierające się na prostej obserwacji [SGD14, rozdz. 3] mówiącej (w skrócie), że aby znaleźć wszystkie m rotacji wzorca, wystarczy odszukać jeden z dwóch podciągów $P_1 = P[0 \dots m/2 - 1]$, $P_2 = P[m/2 \dots m - 1]$ o długości $m/2$ (dla wzorców o parzystej liczbie znaków), ponieważ w każdej z rotacji występuje przynajmniej jeden z dwóch podciągów wzorca, a następnie zweryfikować odnalezioną pozycję.

Rozwiązanie to wykorzystuje algorytm wyszukiwania wielu wzorców, co oznacza, że oba podciągi tekstowe szukane są jednocześnie. Dzięki takiemu podejściu, tekst jest przetwarzany jednokrotnie. W celu przeprowadzenia testów eksperymentalnych zaadoptowano szereg algorytmów wyszukiwania dokładnego pojedynczego oraz wielu wzorców:

- Shift-Or (SO),
- Fast Shift-Or (FSO),
- Fast Average-Optimal Shift-Or (FAOSO),
- Multipattern Shift-Or (MSO),
- Multipattern Fast Shift-Or (MFSO),
- Multipattern Fast Average-Optimal Shift-Or (MFAOSO),
- Counting Filter (CF).

Pierwsze trzy algorytmy (SO, FSO, FAOSO) wymagają uruchomienia wyszukiwania dwukrotnie (osobno dla każdego z dwóch podciągów), natomiast w przypadku kolejnych trzech algorytmów (MSO, MFSO, MFAOSO) wyszukiwanie zostało przeprowadzone dla obu podciągów jednocześnie. Odmiennym przypadkiem jest ostatni algorytm (CF), gdzie wyszukiwanie opiera się na statystycznej metodzie, więc szukany jest właściwy wzorzec. Wszystkie opracowane algorytmy wymagają weryfikacji odnalezionych pozycji.

Przyprowadzone testy wykazały, że algorytm Counting Filter (CF) jest najwolniejszym ze wszystkich algorytmów, a jego efektywność rośnie wraz ze wzrostem rozmiaru wzorca, aczkolwiek nawet dla najdłuższych wzorców pozostaje najwolniejszym rozwiązaniem. Najlepszy czas osiągnął MFAOSO dla wzorców o długości $m = 64$, co przekłada się na szybkość 7–8 GB/s (dla zbiorów `English` i `Proteins`). Algorytmy SO, MSO, FSO, MFSO zachowują stabilne czasy na wszystkich zbiorach danych z wyjąt-

kiem najkrótszych wzorców ($m = 8$), gdzie najgorsze czasy występują dla zbioru `DNA` (najmniejszy rozmiar alfabetu). MFAOSO okazał się najlepszym rozwiązaniem w większości przypadków, szczególnie dla najdłuższych wzorców ($m = 64$). Zaproponowane podejście, pomimo swojej prostoty, okazało się bardzo dobrym rozwiązaniem problemu wyszukiwania wzorców obróconych. Zaproponowane algorytmy osiągają szybkość kilku gigabajtów na sekundę, co stanowi dobry wynik na tle pozostałych rozwiązań.

Konkurencyjne rozwiązanie CSBNDM (Circular Simplified Backward Nondeterministic Dawg Matching) zaprezentowane przez Chena i in. [CHL14], które zostało opublikowane w tym samym roku co [SGD14], opiera się na algorytmie Simplified Backward Nondeterministic Dawg Matching (SBNDM) [Nav01, PT03].

Algorytmy te przetestowano i otrzymane wyniki porównano z algorytmem MFAOSO. Dodatkowo dostosowano algorytm MAG [SGF14] do wyszukiwania wzorców obróconych (w oparciu o taką samą technikę jak rozwiązanie MFAOSO) i nazwano MAGC (MAG for Circular Pattern Matching).

Dla zbioru `DNA` najlepsze wyniki osiągnął algorytm MAGC, aczkolwiek dla krótkich wzorców $m = \{8, 16\}$ lepszy okazał się CSBNDM. Algorytm MFAOSO dominował nad rozwiązaniem CSBNDM dla długich wzorców $m = \{32, 48, 64\}$ jednak MAGC okazał się lepszy od MFAOSO w całej testowanej dziedzinie.

W przypadku alfabetu `Proteins` najlepszy okazał się algorytm CSBNDM, natomiast dla zbioru `English` rozwiązanie MFAOSO osiągnęło lepsze wyniki od wariantów CSBNDM w całej badanej dziedzinie.

Zaprezentowane rozwiązanie mimo swojej prostoty jest w stanie konkurować z innymi rozwiązaniami problemu wyszukiwania wzorców obróconych (ang. *Circular Pattern Matching*). Podejście to okazało się szybsze od konkurencyjnego rozwiązania dla dowolnego rozmiaru wzorca dla alfabetu `English`. Osiągnięto także lepsze wyniki dla długich i średniej długości wzorców ($m > 16$) dla alfabetu `DNA`. W przypadku zbioru `Proteins` konkurencyjne rozwiązanie okazało się szybsze, jednak nie jest wykluczone, że inna kombinacja wariantów i parametrów rozwiązania MAG pozwoliłaby poprawić rezultaty.

3.2 Wyszukiwanie przybliżone

Rozwiązanie zaproponowane przez Susika [Sus17] redukuje problem przybliżonego szukania wzorców do problemu szukania dokładnego wielu wzorców (ang. *partitioning into exact search*). Przedstawione rozwiązanie MAG for Approximate pattern matching (MAGA) jest algorytmem online opierającym się na obserwacji:

Lemat. *Jeżeli wzorzec tekstowy* $P[0 \dots m-1]$ *można podzielić na* $k+1$ *podwzorców* $\mathcal{P} \in \{P_0[0 \dots m/(k+1)-1], P_1[m/(k+1) \dots 2m/(k+1)-1], \dots, P_k[m-mk/(k+1) \dots m-1]\}$ *(dla parzystych wzorców), to wzorzec można odnaleźć z dokładnością do* k *błędów, szukając dokładnie wszystkie podwzorce w tekście* $T[0 \dots n-1]$ *i weryfikując odnalezione pozycje.*

W rozwiązaniu tym wymagana jest weryfikacja wszystkich odnalezionych pozycji, do czego zastosowano algorytm Counting Filter dla odległości Levenshteina.

Wyniki niniejszego rozwiązania zostały opublikowane w 2017 roku przez Susika [Sus17]. Wszystkie warianty rozwiązania konkurencyjnego [FN04] nazwano AOSMASM i wybrano najlepszy wynik.

Testy eksperymentalne dowiodły, że zaproponowane podejście jest udanym rozwiązaniem problemu szukania wzorców przybliżonych. Algorytm MAGA osiągnął lepsze wyniki od AOSMASM w przypadku wyszukiwania wzorców dla małej liczby dopuszczalnych błędów (k), a dodatkowo powiększenie liczby wzorców (r) pozwalało osiągnąć nawet 6-krotne przyspieszenie. Biorąc po uwagę fakt, że MAGA radzi sobie lepiej z szukaniem dużej liczby wzorców (np. 10 tys.) część wyników (jak np. wyniki dla różnych długości wzorców) mogłaby okazać się korzystniejsza dla tego algorytmu. Algorytm MAGA okazał się znacznie szybszy od AOSMASM w przypadku szukania dużej liczby ($r > 1000$) długich wzorców ($m > 32$) przy małej liczbie dopuszczalnych błędów ($k \leqslant 3$) oraz dużego rozmiaru alfabetu ($\sigma > 4$). Rozwiązanie okazało się udanym zastosowaniem algorytmu szukania wielu wzorców do problematyki wyszukiwania przybliżonego.

4 Filtr zliczający

W rozdziale tym opisano alternatywne rozwiązanie filtra zliczającego (ang. *Counting Filter*) wyróżniające się znacznie wyższą szybkością filtracji od implementacji dostępnej w literaturze, szczególnie dla długich wzorców. Poprzez filtr zliczający (CF) rozumiana jest implementacja tego rozwiązania zaproponowana przez Navarro w [Nav97].

Zaproponowany algorytm oparto na technice q-gramów, redukcji alfabetu oraz instrukcjach SSE. Opisano kilka wariantów opracowanego algorytmu, a następnie porównano ich szybkość z rozwiązaniem dostępnym w literaturze. Sprawdzono także użyteczność opracowanego algorytmu w problematyce wyszukiwania translokacji i inwersji w łańcuchu DNA, a wyniki porównano z konkurencyjnym algorytmem.

Wyniki niniejszego rozwiązania zostały opublikowane w 2016 roku przez Susika i Grabowskiego w [SG16]. Kody źródłowe implementacji zostały udostępnione pod adresem `https://github.com/rsusik/cf2`.

Filtr zliczający (CF) jest algorytmem [GL89, JTU96, Nav97], który znajduje zastosowanie w wyszukiwaniu wzorców przybliżonych. W przeciwieństwie do tradycyjnych algorytmów, technika filtracji zamiast szukać miejsc, w których wzorzec może się znajdować, skupia się na odrzucaniu (filtracji) danych wejściowych, które nie spełniają zadanych kryteriów (czyli tych fragmentów tekstowych, w których wzorzec na pewno nie występuje). Filtr zliczający nie jest kompletnym algorytmem wyszukiwania wzorców, a jego celem jest odszukanie miejsca, w którym wzorzec może występować (ale nie musi). W związku z tym konieczna jest weryfikacja pozycji, które zostały odnalezione przez algorytm.

W implementacji nowych wariantów wykorzystano instrukcje Streaming SIMD (Single Instruction, Multiple Data) Extensions (SSE), które są obecne w większości nowoczesnych procesorów. Instrukcje te pozwalają na zrównoleglenie obliczeń poprzez wykonywanie instrukcji na wektorach danych przechowywanych w rejestrach procesora, których liczba uzależniona jest od ich wielkości. Pozwala to na wykonanie operacji jednocześnie dla wielu zmiennych. Oprócz zastosowania instrukcji SSE, uproszczono algorytm, zastosowano technikę przeskoków, umożliwiającą pominięcie tych fragmentów

tekstu, w których na pewno nie występuje wzorzec, technikę q-gramów oraz mapowania alfabetu.

Podstawowy wariant zaproponowanej implementacji filtra zliczającego nosi nazwę CF2. Algorytm ten został następnie usprawniony poprzez zrównoleglenie pętli obliczającej sumę wartości bezwzględnych różnic, dzięki instrukcjom SSE.

Inną obserwacją, którą wykorzystano do usprawnienia algorytmu jest możliwość pominięcia kroków w których wiadomo, że na pewno nie będzie dopasowania. Takie podejście pozwala na pominięcie maksymalnie $m - k$ kroków (gdzie k to liczba dopuszczalnych błędów). Rozwiązanie to jest szczególnie przydatne w przypadku dużego alfabetu (σ), długich wzorców (m) i małej liczby błędów (k).

Innym ulepszeniem, jakie zaproponowano jest zastosowanie superalfabetu opartego na technice q-gramów. Pozwala to na redukcję liczby weryfikacji, co jest jedną z najbardziej czasochłonnych czynności wykonywanych przez algorytm, a jednocześnie mapowanie alfabetu pozwoliło na dokładne oszacowanie wymaganych zasobów (RAM), natomiast instrukcje SSE umożliwiły wykonywanie pojedynczych operacji na kilku zmiennych (16) równolegle, a metoda przeskoków (SKIP) pozwoliła na pominięcie zbędnych iteracji pętli.

Algorytmy zostały przetestowane pod kątem dwóch zastosowań: wyszukiwania fragmentów kodu DNA dla problematyki translokacji i inwersji (tylko zbiór `DNA` oraz $k = 0$) oraz wyszukiwanie wzorców z dokładnością do k błędów. W pierwszym przypadku szukane są fragmenty tekstu, w których może wystąpić dopasowanie, natomiast nie jest wykonywana weryfikacja odnalezionych pozycji. W drugim przypadku rozważane jest wyszukiwanie z dokładnością do $k \in \{1, 2, 3\}$ błędów.

Implementacja filtra zliczającego z zastosowaniem instrukcji SSE oraz techniki q-gramów doprowadziła do znacznej poprawy szybkości działania algorytmu oraz zmniejszenia liczby weryfikacji niezbędnych do wykonania w stosunku do implementacji dostępnej w literaturze. Zaproponowane rozwiązanie osiągnęło 4-krotne przyspieszenie oraz wykonało o blisko 5 rzędów wielkości mniej weryfikacji. Do przyspieszenia algorytmu przyczyniły się dwa czynniki, zastosowanie instrukcji SSE (20% przyspieszenie), co sprawiło, że zamiast wykonywać kilkukrotnie jedną operację, można wykonać jedną operację dla wektora zmiennych jednocześnie; oraz technika pomijania zbędnych iteracji

(4-krotne przyspieszenie). Dodatkowo zastosowanie mapowania alfabetu oraz techniki q-gramów pozwoliło na kontrolę zasobów niezbędnych do działania algorytmu.

5 Semiindeks BFSI: lekka struktura indeksująca tekst z wykorzystaniem filtra Blooma

W rozdziale tym opisano lekką strukturę indeksującą tekst (semiindeks), wykorzystującą filtr Blooma i technikę q-gramów. Nazwa semiindeks została wcześniej użyta przez Claude i in. [CNP+12] w odniesieniu do algorytmu o podobnej specyfice. Jest to rozwiązanie cechujące się szybszym procesem budowy indeksu niż rozwiązania offline i szybszym szukaniem tekstu niż rozwiązania online. Co więcej, blokowe podejście do organizacji danych pozwala na łatwe zrównoleglenie budowy oraz tworzenie semiindeksu przyrostowo, co często nie jest łatwe dla indeksów klasycznych.

Wyniki tego zaprezentowanego zostały opublikowane w 2017 roku przez Grabowskiego i in. [GSR17]. Kody źródłowe implementacji zostały udostępnione pod adresem `https://github.com/rsusik/bfsi`.

Filtr Blooma [Blo70] jest strukturą danych znajdującą wiele zastosowań w dziedzinie informatyki i często stanowi ważny komponent takich rozwiązań. Struktura ta jest w pewnym sensie stratną reprezentacją zbioru danych pozwalającą na wykluczenie obecności elementu w danym zbiorze, aczkolwiek nie pozwala na potwierdzenie, czy element należy do zbioru.

W algorytmie BFSI zaadoptowano ideę minimizerów [RHH+04]. Jest to szczególny przypadek szerzej znanej i stosowanej (np. do wykrywania podobnych zbiorów, duplikatów, itd.) techniki MinHash [Bro97]. Minimizer można zdefiniować następująco: dla tekstu $T[0 \dots n-1]$, gdzie $0 \leqslant i < n$, ramki tekstu $T[i \dots i+w-1]$, znaleźć taki podciąg o długości p, który jest najmniejszy leksykograficznie spośród zbioru podciągów rozpoczynających się wewnątrz ramki $S = \{s_0, s_1, \dots, s_{w-1}\}$, gdzie $s_j = T[i+j \dots i+j+p-1]$, $|S| = w$. Ciąg ten nazywany jest minimizerem $\mathcal{M} = min_{lex}(S)$. Ideę tą zaadoptowano w celu odczytu q-gramów z tekstu, w ten sposób, że pierwszy symbol minimizera jednocześnie stanowi początkowy symbol q-grama (ciągu znaków o długości q).

BFSI (Bloom Filter based Semi-Index) jest algorytmem zaprezentowanym przez Grabowskiego i in. [GSR17], który zaliczyć można do lekkiej struktury indeksującej (ang. *semi-index*) znajdującej się na pograniczu algorytmów online i offline. Idea polega na wykorzystaniu techniki filtrowania (filtr Blooma), q-gramów i minimizerów w celu odrzucenia bloków tekstu, w których na pewno nie występuje szukany wzorzec, a następnie na przeszukaniu tych bloków tekstowych, w których wzorzec może występować przy pomocy algorytmu online (FAOSO). Zaprezentowane rozwiązanie, a przede wszystkim struktura danych (układ bitów), ogranicza znacząco błędne odczyty danych z pamięci podręcznej (ang. *cache miss*), co w wielu algorytmach znacząco ogranicza szybkość działania.

Kody źródłowe konkurencyjnego algorytmu zaproponowanego przez Claude i in. [CNP+12] otrzymano od autorów.

Istnieje możliwość osiągnięcia znacznie lepszych wyników poprzez dobranie korzystniejszych wartości tych parametrów pod konkretny zbiór danych, jednak aby ograniczyć liczbę zmiennych, parametrom tym przypisano stałe wartości.

Przeprowadzono także testy porównujące algorytm BFSI z innymi znanymi indeksami pełnotekstowymi (ang. *full-text indexes*), takimi jak:

1. Indeks FM (ang. *FM-index*) z implementacją V5 procedury *rank* [GP14],
2. Rzadka tablica sufiksowa (ang. *Sparse Suffix Array* [KU96],
3. Tablica sufiksowa wykorzystująca technikę minimizerów do próbkowania (ang. *Sampled Suffix Array on Minimizers*, SamSAMi) [GR15].

Rozwiązanie BFSI osiąga słabsze wyniki od indeksów pełnotekstowych, szczególnie dla długich wzorców ($m = 64$), jednak w przypadku krótszych wzorców ($m < 32$) wygrywa z indeksem FM na zbiorach `Sources` i `XML`. Najlepsze wyniki osiągnął algorytm SamSAMi2 równocześnie używając najwięcej pamięci.

Ponieważ jedną z zalet lekkiej struktury indeksującej jaką jest BFSI, jest szybkość jej konstrukcji oraz fakt, że proces budowy indeksu może być w prosty sposób zrównoleglony i wykonany w kilku wątkach równolegle, porównano czas budowy indeksu. Wariant BFSI w niektórych przypadkach osiąga szybkość budowy indeksu powyżej

100 MB/s, co jest lepszym wynikiem niż osiągane przez indeks FM czy tablicę sufiksową [WBO15, Bai16, LSB16].

Rozwiązanie BFSI dobrze się sprawdza w roli lekkiej struktury indeksującej (ang. *semi-index*), osiągając bardzo dobre wyniki szybkości wyszukiwania wzorców tekstowych oraz krótki czas konstrukcji indeksu jednocześnie zachowując prostotę koncepcji i implementacji. Zaprezentowana struktura opiera się na zasadzie podziału tekstu na bloki równej długości i zastosowaniu filtra Blooma w celu odnalezienia właściwego bloku, w którym może znajdować się wzorzec, dzięki czemu pewna liczba bloków nie musi być w ogóle przeszukiwana. Dodatkowo zastosowanie techniki q-gramów pozwala znacząco ograniczyć liczbę błędnych bloków zwracanych przez filtr Blooma. Testy eksperymentalne dowiodły, że rozwiązanie to pozwala na wyszukiwanie o trzy rzędy wielkości szybsze od algorytmów online, kosztem pamięci nie przekraczającej 70% rozmiaru tekstu.

6 Podsumowanie

W niniejszej pracy opracowano, zaimplementowano, gruntownie przetestowano i porównano szereg rozwiązań opartych na technice bitowej równoległości z rozwiązaniami konkurencyjnymi. Zaproponowane algorytmy rozwiązują takie problemy jak wyszukiwanie:

- wielu wzorców,
- w tekście skompresowanym,
- wzorców obróconych,
- przybliżone,
- jednego wzorca przy pomocy lekkiej struktury indeksującej.

Przeprowadzone testy eksperymentalne potwierdzają fakt, że równoległość bitowa znajduje praktyczne zastosowanie w dziedzinie stringologii. Technika ta szczególnie dobrze sprawdziła się w problematyce wyszukiwania wielu wzorców. Opracowany algorytm online pozwala na szybkie wyszukiwanie wzorców zarówno w tekście nieskompresowanym, jak i w tekście skompresowanym (kody bajtowe), osiągając lepsze wyniki od konkurencyjnych rozwiązań w większości przypadków. Zaproponowano i przetestowano

szereg alternatywnych metod mapowania alfabetu, każdy cechujący się inną charakterystyką (np. brakiem konieczności budowania histogramu). Wykazano, że zaproponowane rozwiązanie charakteryzuje się subliniową złożonością czasową w przypadku średnim. Zastosowanie kodów bajtowych pozwoliło na kompresję tekstu, a jednocześnie redukcję pamięci dla struktur potrzebnych przy wyszukiwaniu oraz przyspieszenie procesu wyszukiwania długich wzorców (w postaci słów) w stosunku do szukania w tekście nieskompresowanym.

Wykazano, że algorytm wyszukiwania wielu wzorców można z sukcesem zaadoptować dla problemu wyszukiwania wzorców obróconych, gdzie korzystając z prostej obserwacji (mówiącej, że ciąg obrócony zawsze zawiera przynajmniej jedną połowę oryginalnego ciągu tekstowego), osiągnięto bardzo dobre wyniki (7–8 GB/s na standardowym komputerze PC, w implementacji jednowątkowej) na tle istniejących rozwiązań. Algorytm ten rozszerzono i porównano także z rozwiązaniem Chena i in., które pojawiło się w późniejszym czasie, jednak tu również wyniki zaproponowanego rozwiązania okazały się bardzo konkurencyjne.

W pracy wykazano również, że algorytm MAG można zastosować do rozwiązania problemu wyszukiwania przybliżonego z dokładnością do k błędów. Podejście to pozwoliło osiągnąć nawet 6-krotne przyspieszenie (dla dużej liczby szukanych wzorców) w stosunku do konkurencyjnych rozwiązań.

Udowodniono także, że można wykorzystać technikę bitowej równoległości do opracowania nowego algorytmu filtrującego dla problematyki wyszukiwania przybliżonego. Zaprezentowano kilka wariantów, w których zastosowano: rozszerzone instrukcje procesora (SSE), technikę przeskoków i q-gramów, co znacząco poprawiło szybkość działania algorytmu. Przeprowadzone testy wykazały, iż zaproponowana implementacja osiąga szybkość wyszukiwania nawet 4-krotnie większą niż wersja standardowa tego algorytmu znana z literatury.

Technika równoległości bitowej sprawdziła się także bardzo dobrze w dziedzinie indeksowania. Lekka struktura indeksująca wykorzystująca bitową równoległość w celu redukcji liczby błędnych odczytów z pamięci podręcznej (ang. *cache misses*), okazała się bardzo dobrym rozwiązaniem osiągającym znacznie lepsze wyniki niż konkurencyjne rozwiązania. Blokowa budowa i zastosowanie filtrów Blooma – które można postrzegać

jako specyficzną formę kompresji stratnej danych – umożliwiły odrzucenie znacznej części bloków, w których z pewnością nie występuje wzorzec, a odpowiednie ułożenie bitów w przeplocie pozwoliło znacznie ograniczyć liczbę błędnych odczytów.

Opisane rozwiązania łączy ścisła zależność od wielkości słowa maszynowego. Techniki równoległości bitowej wykorzystują możliwość umieszczenia kilku zmiennych w jednym słowie maszynowym, którego rozmiar jest ograniczony (obecnie 64 bity, z możliwością rozszerzenia do 128 bitów dla instrukcji SSE oraz do 512 bitów dla AVX), co w pewnym sensie może ograniczać korzyści z nich płynące i ich zastosowanie. Wraz z postępami technologii można jednak zauważyć wyraźny trend wydłużania słowa maszynowego, dlatego też warto poświęcić uwagę technikom równoległości bitowej.

Przedstawione rozwiązania dowiodły, że techniki równoległości bitowej umożliwiają przyspieszenie wyszukiwania wielu wzorców, szukania przybliżonego, wyszukiwania w tekście skompresowanym oraz indeksowania, a zastosowanie odpowiednich algorytmów kompresji tekstu przyczyniło się do szybszego wyszukiwania wzorców i budowy indeksu o korzystnych relacjach użytkowych. Oznacza to, że postawione tezy są słuszne, a cel pracy został zrealizowany pomyślnie.

Ciekawym rozwinięciem dalszych prac może być zaadoptowanie przedstawionych rozwiązań do problematyki wyszukiwania wielu wzorców obróconych lub przybliżonego wyszukiwania wzorców obróconych. Interesujące może być także wykorzystanie innego kodowania tekstu dla algorytmu MAG. Zastosowanie instrukcji AVX mogłoby pozwolić także na dalsze zwiększenie wydajności zaprezentowanego rozwiązania filtrującego. Innym kierunkiem dalszych badań może być zaadoptowanie lekkiej struktury indeksującej (BFSI) do problematyki wyszukiwania wielu wzorców, a także ze względu na blokową budowę, wyszukiwania równoległego.

Bibliografia

[AR99] C. Allauzen and M. Raffinot. Factor Oracle of a Set of Words. Technical Report 99-11, 1999.

[Bai16] U. Baier. Linear-Time Suffix Sorting – A New Approach for Suffix Array Construction. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 54. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[Blo70] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[Bro97] A. Z. Broder. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences*, pages 21–29. IEEE, 1997.

[CHL14] K. H. Chen, G. S. Huang, and R. C. T. Lee. Bit-Parallel Algorithms for Exact Circular String Matching. *Computers & Operations Research*, 57(5):731–743, 2014.

[CNP+12] F. Claude, G. Navarro, H. Peltola, L. Salmela, and J. Tarhio. String Matching with Alphabet Sampling. *Journal of Discrete Algorithms*, 11:37–50, 2012.

[FG09] K. Fredriksson and Sz. Grabowski. Average-Optimal String Matching. *Journal of Discrete Algorithms*, 7(4):579–594, 2009.

[FN04] K. Fredriksson and G. Navarro. Average-Optimal Single and Multiple Approximate String Matching. *ACM Journal of Experimental Algorithmics*, 9:article 1.4, 2004. 45 pages.

[Fre09] K. Fredriksson. Succinct Backward-DAWG-Matching. *ACM Journal of Experimental Algorithmics*, 13:1.8–1.26, 2009.

[GL89] R. Grossi and F. Luccio. Simple and Efficient String Matching with k Mismatches. *Information Processing Letters*, 33(3):113–120, 1989.

[GP14] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.

[GR15] Sz. Grabowski and M. Raniszewski. Sampling the Suffix Array with Minimizers. In *String Processing and Information Retrieval*, volume 9309 of *Lecture Notes in Computer Science*, pages 287–298. Springer, 2015.

[GSR17] Sz. Grabowski, **R. Susik**, and M. Raniszewski. A Bloom Filter based Semi-Index on q-grams. *Software: Practice and Experience*, 47(6):799–811, 2017.

[JTU96] P. Jokinen, J. Tarhio, and E. Ukkonen. A Comparison of Approximate String Matching Algorithms. *Software: Practice and Experience*, 26(12):1439–1458, 1996.

[KU96] J. Kärkkäinen and E. Ukkonen. Sparse Suffix Trees. In *Computing and Combinatorics Conference*, volume 1090 of *Lecture Notes in Computer Science*, pages 219–230, 1996.

[LSB16] J. Labeit, J. Shun, and G. E. Blelloch. Parallel Lightweight Wavelet Tree, Suffix Array and FM-Index Construction. In *Data Compression Conference*, pages 33–42. IEEE Computer Society, 2016.

[Nav97] G. Navarro. Multiple Approximate String Matching by Counting. In *Proceedings of the 4th South American Workshop on String Processing*, pages 95–111. Carleton University Press, 1997.

[Nav01] G. Navarro. NR-grep: A Fast and Flexible Pattern-Matching Tool. *Software: Practice and Experience*, 31(13):1265–1312, 2001.

[NF04] G. Navarro and K. Fredriksson. Average Complexity of Exact and Approximate Multiple String Matching. *Theoretical Computer Science*, 321(2–3):283–290, 2004.

[PT03] H. Peltola and J. Tarhio. Alternative Algorithms for Bit-Parallel String Matching. In *String Processing and Information Retrieval*, volume 2857, pages 80–94. Springer, 2003.

[RHH+04] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke. Reducing Storage Requirements for Biological Sequence Comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[SG16] **R. Susik** and Sz. Grabowski. Engineering the Counting Filter for String Matching Algorithms. In *Algorithms, Networking and Sensing for Data Processing, Mobile Computing and Applications*, pages 125–140. Łódź University of Technology Press, 2016.

[SGD14] **R. Susik**, Sz. Grabowski, and S. Deorowicz. Fast and Simple Circular Pattern Matching. In *Man-Machine Interactions 3*, pages 537–544. Springer, 2014.

[SGF14] **R. Susik**, Sz. Grabowski, and K. Fredriksson. Multiple Pattern Matching Revisited. In *Proceedings of the Prague Stringology Conference*, pages 59–70, 2014.

[STK06] L. Salmela, J. Tarhio, and J. Kytöjoki. Multipattern String Matching with q-grams. *ACM Journal of Experimental Algorithmics*, 11, 2006.

[Sus17] **R. Susik**. Applying a q-gram based Multiple String Matching Algorithm for Approximate Matching. *Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska*, 7(3):47–50, 2017.

[WBO15] L. Wang, S. Baxter, and J. D. Owens. Fast Parallel Suffix Array on the GPU. In *Parallel and Distributed Computing*, volume 9233 of *Lecture Notes in Computer Science*, pages 573–587. Springer, 2015.

[Yao79] A. C. Yao. The Complexity of Pattern Matching for a Random String. *SIAM Journal on Computing*, 8(3):368–387, 1979.