Artur Jeż: Autoreferat

Imię i nazwisko: Artur Jeż

Dyplomy i stopnie naukowe

- Magister informatyki: 2006
  Instytut Informatyki, Uniwersytet Wrocławski.
- Magister matematyki: 2006
  Instytut Matematyczny, Uniwersytet Wrocławski.

  III miejsce w konkursie PTM na pracę magisterską.

- Doktor nauk matematycznych w zakresie informatyki: 2010
  Instytut Informatyki, Uniwersytet Wrocławski.
  Gramatyki koniunkcyjne i układy równań nad zbiorami liczb naturalnych
  promotorzy: prof. Krzysztof Loryś, dr hab. Alexander Okhotin.

  Nagroda prezesa Rady Ministrów za rozprawę doktorską.

Informacja o dotychczasowym zatrudnieniu.

- od X 2010 adiunkt, Instytut Informatyki, Wydział Matematyki i Informatyki, Uniwersytet Wrocławski
  (X 2012–XI 2014 na urlopie naukowym)
- X 2012–XI 2014 postdok, Max Planck Institute für Informatik, Saarbrücken, Niemcy
  (od II 2013: stypendium Humboldta)

Wskazanie osiagnięcia naukowego. Osiągnięciem naukowym *Rekompresja: nowe podejście do równań słów, unifikacji i skompresowanych danych* jest cykl jednotematycznych prac:

[H1] A. Jeż, „Compressed Membership for NFA (DFA) with Compressed Labels is in NP (P)", *Symposium on Theoretical Aspects of Computer Science (STACS)*, (Paryż, Francja, 2012), LIPICs 14, 136–147,
pełna wersja: „The Complexity of Compressed Membership Problems for Finite Automata" *Theory of Computing Systems*, 55:4 (2014), 685–718, wydanie specjalne po STACS 2012.

[H2] A. Jeż, „Faster fully compressed pattern matching by recompression", *International Colloquium on Automata, Languages and Programming (ICALP (A))*, (Warwick, Wielka Brytania, 2012), LNCS 7391, 533–544,
pełna wersja: przyjęta do *ACM Transactions on Algorithms*.

[H3] A. Jeż „Recompression: a simple and powerful technique for word equations", *Symposium on Theoretical Aspects of Computer Science (STACS)*, (Kilonia, Niemcy, 2013), LIPIcs 20, 233–244.

[H4] A. Jeż, „Recompression: Word Equations and Beyond", *Developments in Language Theory (DLT)*, (Paryż, Francja, 2013), LNCS 7907, 12–26, referat zaproszony — praca przeglądowa.

[H5] A. Jeż, „Approximation of grammar-based compression via recompression", *Annual Symposium on Combinatorial Pattern Matching (CPM)*, (Bad Herrenalb, Niemcy, 2013), LNCS 7922, 165–176.

[H6] A. Jeż, „One-variable word equations in linear time", *International Colloquium on Automata, Languages and Programming (ICALP (B))*, (Ryga, Łotwa, 2013), LNCS 7966,

324–335,
pełna wersja: *Algorithmica* (zaakceptowana i dostępna online).

[H7] A. Jeż, M. Lohrey, „Approximation of smallest linear tree grammar”, *Symposium on Theoretical Aspects of Computer Science (STACS)*, (Lyon, Francja, 2014), LIPIcs 25, 445–457.

[H8] V. Diekert, A. Jeż, W. Plandowski, „Finding All Solutions of Equations in Free Groups and Monoids with Involution”, *International Computer Science Symposium in Russia (CSR)*, (Moskwa, Rosja, 2014), LNCS 8476, 1–15.

[H9] A. Jeż, „A *really* simple approximation of smallest grammar”, *Annual Symposium on Combinatorial Pattern Matching (CPM)*, (Moskwa, Rosja, 2014), LNCS 8486, 182–191.

[H10] A. Jeż, „Context unification is in PSPACE”, *International Colloquium on Automata, Languages and Programming (ICALP (B))*, (Kopenhaga, Dania, 2014), LNCS 8573, 244–255.

Ze względu na rozmiar, omówienie wyników prac zawarte jest w części 1.

Poza pracami składającymi się na wymienione wyżej osiągnięcie naukowe, prowadziłem również inne badania naukowe, omówienie wyników tychże prac zawarte jest w części 2.

Ze względu na rozmiar prac, wyniki podane są w pewnym uproszczeniu: subtelności założeń i wyników zostały pominięte a całość autoreferatu ma za zadanie oddanie ogólnej idei wyników, metod i dowodów w przypadku osiągnięcia naukowego oraz ogólnej idei wyników w przypadku pozostałych prac. Precyzyjne i jednoznaczne sformułowania można znaleźć w pracach stanowiących osiągnięcie naukowe.

Pełne wersje prac dostępne są na stronie domowej habilitanta `www.ii.uni.wroc.pl/~aje`.

Część 1. Omówienie osiągnięcia naukowego: Rekompresja

Przedstawione prace opierają się na zaproponowanej przeze mnie technice *rekompresji*. W skrócie, opiera się ona na dokonywaniu prostych operacji kompresji (zastępowanie podsłów ab przez nową literę c oraz zastępowanie podsłów a^k przez literę a_k), które dokonywane są na słowach. Słowa te są być może zadane w niejawnej postaci, np. słów reprezentowanych przez rozwiązania równania na słowach, przez gramatykę, poprzez kompresję blokową itp. Metoda ta okazała się zaskakująco skuteczna i ogólna, przegląd tych zastosowań stanowi cel tej części.

1. Stan przed

W tym rozdziale znajduje się krótki opis ważnych wyników i trendów w dziedzinach, których dotyczą prace stanowiące osiągnięcie naukowe.

1.1. **Równania w słowach.** Problem równań w słowach, tj. rozwiązania równania w algebrze słów, był po raz pierwszy rozważany przez A. Markowa w latach 50. XX wieku. W problemie tym dostajemy równanie postaci

$$u = v$$

gdzie u oraz v są ciągami liter (z ustalonego alfabetu) oraz zmiennych a *rozwiązanie* to podstawienie, które zamienia tę formalną równość w prawdziwą równość dwóch ciągów liter (nad ustalonym wcześniej alfabetem). Stosunkowo łatwo można zredukować ten problem do 10. problemu Hilberta, tj. pytania o rozwiązania układu równań diofantycznych. Według powszechnego wówczas przekonania, 10. problem Hilberta jest nierozstrzygalny i A. Markow chciał udowodnić to poprzez pokazanie nierozstrzygalności problemu słów.

Faktycznie, 10. problem Hilberta jest nierozstrzygalny, tym niemniej okazało się, iż problem równań słów jest *rozstrzygalny*, co zostało po raz pierwszy pokazane przez G. Makanina [79]. Dowód terminacji algorytmu jest bardzo skomplikowany (oraz daje stosunkowo słabe oszacowanie na złożoność obliczeniową), dlatego też przez lata uzyskiwano kolejne uproszczenia tegoż algorytmu [51, 113, 60, 44]. Uproszczenie takie ma potencjalnie wiele zastosowań: wydaje się naturalne, iż prostszy algorytm można będzie uogólnić (np. do przypadku równań w grupach), niż algorytm skomplikowany, co więcej, prostszy algorytm powinien być skuteczniejszy w stosowaniu i należy oczekiwać, że miałby też mniejszą złożoność obliczeniową.

1.1.1. *Uszczegółowienia.* Dosyć łatwo pokazać, iż równania w słowach są NP-trudne (i jak dotychczas nie udało się uzyskać żadnego lepszego ograniczenia dolnego). W naturalny sposób stymulowało to badania szukające ograniczonych podklas równań w słowach, dla których możliwe jest podanie wydajnych (tj. wielomianowych) algorytmów [11]. Jedną z intensywnie badanych podklas są równania z ograniczoną ilością zmiennych: wiadomo, że zarówno równania z jedną [25, 62] jak i z dwoma [11, 50, 24] zmiennymi można rozwiązać w czasie wielomianowym. Kwestia złożoności obliczeniowej równań z trzema zmiennymi pozostaje otwarta, jak dotychczas nie wiadomo nawet, czy są one w NP [106].

1.1.2. *Uogólnienia.* Od momentu opublikowania przez G. Makanina jego rozwiązania prowadzone prace nad rozszerzeniem tegoż algorytmu do innych struktur. Trzy kierunki wydają się bardzo naturalne:

- dodanie do równań dodatkowych więzów;
- równania w grupie wolnej;
- równania termów.

Więzy. Z punktu widzenia zastosowań, atrakcyjne jest rozważanie równań w których dodatkowo pozwalamy na użycie więzów, tj. żądamy, aby rozwiązanie dla X spełniało jakieś dodatkowe własności. Po raz pierwszy dokonano takiego rozszerzenia w przypadku więzów regularnych [113], jednocześnie dla wielu innych typu ograniczeń, np. warunków wiążących długości rozwiązań, wciąż nie wiadomo, czy tak sformułowany problem jest rozstrzygalny (staje się on nierozstrzygalny, jeśli pozwolimy na zliczanie wystąpień poszczególnych liter w podstawieniach [6]).
Grupa wolna. Z algebraicznego punktu widzenia, problem równań słów to rozwiązywanie równań w półgrupie wolnej. Naturalna jest próba rozszerzenia tego algorytmu dla równań w grupie wolnej i następnie być może do większej klasy grup (zauważmy jednocześnie, że istnieją grupy i półgrupy, dla których problem równań w słowach jest nierozstrzygalny). Pierwszy algorytm dla przypadku równań w grupach wolnych został podany przez G. Makanina [80, 81], warto dodać, iż jego algorytm nie jest pierwotnie-rekurencyjny [61]. Ponadto A. Razborov pokazał, iż algorytm ten mógł być użyty do podania skończonego opisu wszystkich rozwiązań równania [103] (bardziej przystępny opis algorytmu dostępny jest w [56]). Warto zaznaczyć, iż ten opis był pierwszym krokiem w stronę udowodnienia hipotezy Tarskiego (mówiącej o rozstrzygalności teorii grup wolnych) [57].
Termy. Na słowa można również patrzeć jak na bardzo proste termy: litery to po prostu symbole funkcyjne o arności 1. Wtedy równania w słowach są równaniami na (bardzo prostych) termach. Powszechnie wiadomo, że spełnialność równań termów można rozstrzygnąć w wielomianowym czasie, jeśli zmienne reprezentują tylko pełne termy [104]; tak postawiony problem nie uogólnia jednak równań w słowach. Naturalnym uogólnieniem unifikacji termów oraz równań w słowach jest *unifikacja drugiego rzędu*, w którym pozwalamy, aby zmienne

reprezentowały funkcje przyjmujące argumenty będące zamkniętymi termami. Niestety, dość łatwo pokazać, iż problem ten jest nierozstrzygalny, nawet w pewnych ograniczonych przypadkach [42, 28, 65, 67]. *Unifikacja kontekstów* [17, 18, 108] jest naturalnym problemem „po środku": pozwalamy, aby zmienne reprezentowały funkcje, wymagamy jednak, aby używały one swoich argumentów *dokładnie raz*. Łatwo pokazać, że problem ten wciąż uogólnia równania w słowach, z drugiej strony, dowody nierozstrzygalności unifikacji drugiego rzędu nie przekładają się bezpośrednio na unifikację kontekstów. Ponadto, problem ten ma naturalne powiązania z innymi znanymi problemami (equality up to constraints [85], liniowa unifikacja drugiego rzędu [68, 65], one-step term rewriting [86], bounded second order unification [110], ...). Niestety, od dwóch dekad pytanie, czy unifikacja kontekstów jest rozstrzygalna, pozostało otwarte. Mimo intensywnych badań, wiedza na temat rozstrzygalności tego problemu jest bardzo skąpa, znane są jedynie rozwiązania w bardzo ograniczonych przypadkach [18, 109, 66, 65, 112, 111, 69, 32].

1.2. **Operacje na skompresowanych danych.** [1] W ostatnich latach obserwujemy istotny wzrost ilości produkowanych danych. Dane te są oczywiście przesyłane, przetwarzane i składowane; ze względu na ogromne rozmiary i związane z tym koszty, kompresja stała się operacją rutynową. Rodzi to niestety pewne problemy: przed każdym użyciem potrzebna jest dekompresja, konieczność ta często niweczy zyski uzyskane wskutek kompresji.

Aby zapobiec takim stratom, prowadzone są intensywne badania nad algorytmami, które operują *bezpośrednio na skompresowanych danych* [33, 34, 35, 36, 39, 40, 41, 46, 97]. Z teoretycznego punktu widzenia oznacza to, iż chcemy, aby czas działania algorytmu zależny był od rozmiaru skompresowanych danych, nie zaś od rozmiaru danych zdekompresowanych.

Standardy kompresji różnią się między sobą zarówno w ogólnych założeniach, jak i w szczegółach implementacyjnych, dlatego też algorytmy działające na skompresowanych danych istotnie różnią się w zależności od rozważanego formatu kompresji. Jednym z takich formatów (mniej praktycznym, lecz ważnym z punktu widzenia teorii informatyki) są tzw. *Straight Line Programs* (*SLP*); SLP jest po prostu gramatyką bezkontekstową, która generuje dokładnie jedno słowo. Choć taka definicja kompresji nie wydaje się na pierwszy rzut oka dobrze umotywowana, zyskała sobie dużą popularność: z jednej strony postać skompresowana (tj. gramatyka generująca dane słowo) jest stosunkowo łatwa w dalszej obróbce, a z drugiej strony algorytmy kompresji blokowej (np. LZW, LZ77, LZ78) okazały się być blisko związane z SLP: istnieją proste algorytmy tłumaczące jedną formę kompresji na drugą [105, 12]. Co więcej, ze względu na wspomnianą wcześniej prostotę i naturalną definicję indukcyjną, algorytmy działające na danych skompresowanych przy użyciu kompresji blokowych zwykle zaczynają się od przetłumaczenia takiej postaci na SLP [33, 34].

Znane są liczne wydajne algorytmy działające na danych dostarczonych w postaci SLP, z punktu widzenia tego referatu warto już teraz wspomnieć, że dla dwóch SLP $\mathcal{A}$ oraz $\mathcal{B}$ można w wielomianowym czasie sprawdzić, czy reprezentują ono to samo słowo [97]. Co więcej, możliwe jest nawet zwrócenie (skompresowanej) listy wszystkich wystąpień słowa zdefiniowanego przez $\mathcal{B}$ w słowie zdefiniowanym przez $\mathcal{A}$ [53, 39, 40, 83, 46, 72]. Ten problem jest jednym z najważniejszych w dziedzinie, dlatego też nawet drobne ulepszenia w algorytmach go rozwiązujących są bardzo pożądane.

1.3. **Kompresja i równania w słowach.** Przez ponad 20 lat od opublikowania algorytmu przez G. Makanina postęp w kwestii algorytmów dla problemu równań słów był bardzo mały:

[1] Związek tego zagadnienia z równaniami w słowach zostanie wyjaśniony w następnym rozdziale.

oryginalny algorytm był poprawiany w różnych miejscach, dzięki czemu udało się uzyskać lepsze ograniczenia jego złożoności obliczeniowej, jednak główna idea (oraz ogólny poziom skomplikowania dowodu) pozostał podobny.

Pierwszy przełom dokonany został przez W. Plandowskiego oraz W. Ryttera [101], którzy po raz pierwszy użyli kompresji do rozwiązania równań w słowach. Pokazali oni, iż najkrótsze rozwiązanie (o rozmiarze N) równania w słowach (o rozmiarze n) ma reprezentację w postaci SLP o rozmiarze $\mathsf{poly}(n, \log N)$[2]; używając algorytmu do sprawdzania równości dwóch SLP [97] w prosty sposób daje to (niedeterministyczny) algorytm działający w czasie $\mathsf{poly}(n, \log N)$. Praca ta nie dawała jednak żadnego ograniczenia na N, jedyne znane (poczwórnie wykładnicze od n) pochodziło z algorytmu Makanina, co pozwalało ono na uzyskanie złożoności 3NEXPTIME. Niedługo potem ograniczenie zostało polepszone do potrójnie wykładniczego [44], co natychmiast dawało algorytm z klasy 2NEXPTIME, jednocześnie ta sama praca ulepszała algorytm Makanina, tak iż działał on w EXPSPACE.

W. Plandowski podał następnie lepsze (podwójnie wykładnicze) oszacowanie na wielkość najkrótszego rozwiązania [98] i tym samym uzyskał algorytm w klasie NEXPTIME, w szczególności był to najszybszy ówcześnie znany algorytm. Dowód ten był oparty na nowatorskich faktoryzacjach słów. Dzięki lepszemu zrozumieniu wzajemnej zależności pomiędzy faktoryzacjami a algorytmem udało mu się następnie poprawić złożoność algorytmu do PSPACE [99].

Należy zaznaczyć, iż rozwiązanie zaproponowane przez W. Plandowskiego jest istotnie prostsze, niż podane przez G. Makanina. W szczególności, pozwoliło ono na duże łatwiejsze rozszerzenia: V. Diekert, C. Gutiérrez i Ch. Hagenah [23] pokazali, że algorytm W. Plandowskiego można rozszerzyć do przypadku, w którym zezwalamy na więzy regularne w równaniach (tj. zadajemy, iż słowo podstawiane pod zmienną X jest z języka regularnego, którego definicja przy użyciu automatu skończonego jest częścią wejścia) oraz inwersję; tak rozszerzony algorytm dalej działa w pamięci wielomianowej. To rozszerzenie pozwala z kolei na rozwiązywanie równań w grupach wolnych [23] (warto zauważyć, że w ogólności nie wiadomo, czy problem równań słów w grupie wolnej jest trudniejszy czy może prostszy, niż problem równań słów w półgrupie wolnej).

Z drugiej strony, W. Plandowski pokazał, iż podany przez niego algorytm może być użyty do generowania skończonej reprezentacji wszystkich rozwiązań równania w słowach [100], co pozwala na rozstrzyganie wielu problemów decyzyjnych dotyczących zbioru rozwiązań równania (skończoność, ograniczoność, ograniczenie na wykładnik periodyczności itp.). Nie wiadomo, czy można uogólnić ten algorytm tak, by generował wszystkie rozwiązania równania z więzami regularnymi w półgrupie z inwersją (lub w grupie wolnej).[3]

Pojawienie się nowego algorytmu i wykazania związków kompresji z problemem równań w słowach ożywiło nadzieję na rozwiązanie problemu unifikacji kontekstów. Początki okazały się być dość obiecujące: używając „drzewowych" odpowiedników SLP udało się ustalić złożoność obliczeniową kilku problemów związanych z unifikacją kontekstów [32, 66, 19]. Niestety, podejście to nie udało się w pełni uogólnić — nie znaleziono drzewowych odpowiedników użytych przez W. Plandowskiego faktoryzacji słów.

---

[2] $\mathsf{poly}(n, \log N)$ oznacza klasę funkcji wielomianowych w zmiennych n i N.

[3] Wersja konferencyjna pracy W. Plandowskiego [100] zawierała stwierdzenie, iż jest to możliwe, niestety, praca zawierała błąd i nie wiadomo, czy ten algorytm można uogólnić tak, aby stosował się również do półgrup z inwersją; więzy regularne nie są tu problemem.

Warto nadmienić, że zaproponowane przez W. Ryttera i W. Plandowskiego podejście, w którym kompresujemy rozwiązanie problemu przy użyciu SLP (lub też w przypadku niedeterministycznym — zgadujemy skompresowaną reprezentację) a następnie dokonujemy operacji na tymże SLP używając znanych algorytmów działających w czasie wielomianowym, okazało się niezwykle owocne w wielu dziedzinach informatyki. Dostępna praca przeglądowa M. Lohrey'a opisuje wiele tego typu zastosowań [76].

2. Rekompresja

Obecnie przedstawię główne idee metody rekompresji. Zostanie ona zaprezentowana na przykładzie rozwiązania problemu równań w słowach. Zaznaczyć należy, że nie jest to problem, dla którego ta technika została stworzona [H1], ani też nie jest to najbardziej doniosłe (zdaniem habilitanta) rozwiązanie uzyskane dzięki tej technice [H10]. Tym niemniej, jest to problem na którym najprościej można wyłożyć podstawowe idee i własności tej techniki.

Technika ta jest ukoronowaniem wielu podejść i metod, które pojawiły się niezależnie w różnych dziedzinach informatyki:

- Pomysł zastępowania krótkich podsłów przez nowe litery i iterowanie tej procedury został użyty przez K. Mehlhorna [82] w strukturze danych do sprawdzania równości dynamicznie zmieniających się napisów; metoda ta została ulepszona w późniejszej pracy S. Alstrupa i in. [2].
- M. Lohrey i Ch. Mathissen [78] w pracy o rozpoznawaniu skompresowanego tekstu przez skompresowany automat również stosowali pomysł zastępowania podsłów i modyfikacji instancji, tak aby ta modyfikacja była możliwa do przeprowadzenia. Zamiany takie jednak nie były iterowane i nowo wprowadzone litery nie były zastępowane.
- Metodę bardzo zbliżoną do rekompresji, zamieniającą maksymalne bloki liter oraz pary, zastosował H. Sakamoto w algorytmie aproksymacyjnym dla problemu konstrukcji najmniejszej gramatyki dla słowa [107]. Jego metoda była oparta na algorytmie RePair [63] — praktycznym kompresorze gramatykowym. Ponieważ jednak w tym wypadku tekst był zadany jawnie, analiza jest istotnie prostsza, w szczególności nie modyfikuje gramatyki na podstawie zaaplikowanych kompresji par i bloków.

2.1. Rekompresja i równania w słowach.

Rozpocznę od ścisłego sformułowania problemu równań w słowach: rozważamy skończony alfabet Σ oraz zbiór zmiennych $\mathcal{X}$; w czasie działania algorytmu alfabet Σ będzie zwiększał się o nowe litery, lecz zawsze pozostanie skończony. Równanie w słowach jest postaci '$u = v$', gdzie $u, v \in (\Sigma \cup \mathcal{X})^*$ a jego rozwiązanie to homomorfizm $S : \Sigma \cup \mathcal{X} \mapsto \Sigma^*$, który jest stały na Σ, tj. $S(a) = a$, oraz spełnia to równanie, to znaczy słowa $S(u)$ oraz $S(v)$ są równe. Przez n oznaczamy rozmiar tego równania, tzn. $|u| + |v|$. W ogólności, algorytm stosuje się z jedynie kosmetycznymi zmianami również do przypadku układu równań na słowach, dla prostoty przekazu nie będę się jednak tym przypadkiem zajmować.

Ustalmy dowolne rozwiązanie S równania $u = v$, bez zmniejszenia ogólności możemy założyć, że jest to *rozwiązanie najkrótsze*, tzn. minimalizujące $|S(u)|$; niech N oznacza *długość rozwiązania*, czyli $|S(u)|$. Z wcześniejszych prac W. Plandowskiego i W. Ryttera [101] wiemy, że $S(u)$ (a także $S(X)$ dla każdej zmiennej) można przedstawić przy użyciu SLP (o rozmiarze $\mathsf{poly}(n, \log N)$), w istocie podobny wniosek można też wysnuć z późniejszych prac [98, 99, 100]. Niezależnie od postaci rozwiązania i SLP, wiadomo, że jedna z produkcji w tym SLP jest postaci $c \to ab$, gdzie c jest nieterminalem SLP a $a, b \in \Sigma$ są literami. Spróbujmy „odwrócić" tę

produkcję, tj. zastąpić w $S(u)$ wszystkie pary liter ab przez c. Stosunkowo łatwo sformalizować tę operację dla słów, nie jest jasne, co należy zrobić w przypadku równań, przyjrzyjmy się więc dokładnie prostszemu fragmentowi.

Algorytm 1 KompPar(ab, w) Kompresja pary ab

1: niech $c \in \Sigma$ będzie nie używaną literą
2: zastąp wszystkie wystąpienia ab w w przez c

Rozważmy zadane jawnie słowo w. Wykonanie na nim „kompresji pary" ab jest łatwe (zastępujemy wszystkie pary ab przez c), o ile $a \neq b$: zastępowanie par aa jest kłopotliwe, gdyż takie pary mogą na siebie „nachodzić". Rozwiązanie jest proste: zastępujemy *maksymalne bloki* litery a: blok a^ℓ jest *maksymalny*, gdy na lewo i prawo od niego nie ma litery a (w szczególności może tam nie być żadnej litery).

Bardziej formalnie, operacje te zadane są następująco:

- *kompresja pary ab* Dla danego słowa w zamień wszystkie występowania ab w w przez nie używaną literę c.
- *kompresja bloków a* Dla danego słowa w zamień wszystkie występowania maksymalnych bloków a^ℓ dla $\ell > 1$ w w przez nie używane litery a_ℓ.

W kompresji par zawsze zakładamy, że litery a oraz b są różne.

Zauważmy, że odwrócenie każdej z wymienionych operacji tworzy produkcję dla SLP: zastąpienie ab przez c odpowiada produkcji $c \to ab$, podobnie zastąpienie a^ℓ przez a_ℓ odpowiada produkcji $a_\ell \to a^\ell$.

Algorytm 2 KompBlok(a, w) Kompresja bloków litery a

1: **dla** $\ell > 1$ **wykonaj**
2: niech $a_\ell \in \Sigma$ będzie nie używaną literą
3: zastąp wszystkie maksymalne bloki a^ℓ w w przez a_ℓ

Kompresja słowa w następuje poprzez iterowanie kompresji par i kompresji bloków, przy czym symbole zastępujące pary i bloki traktowane są jak zwykłe litery. Jest wiele możliwości implementacji takiej iteracji (które zależą od kolejności, sposobu traktowania „nowych" liter, itp.), w zasadzie każdy „rozsądny" wariant jest dobry, poniżej przedstawię jeden z nich. Wypisuje on wszystkie występujące na początku pary i bloki, następnie kolejno wykonuje kompresję według stworzonych wcześniej list. Po wykonaniu kompresji dla wszystkich elementów, powtarza iterację.

Algorytm 3 Kompresja(w) Kompresuje zadane słowo w do SLP

1: **dopóki** $|w| > 1$ **wykonaj**
2: $L \leftarrow$ lista liter w w
3: $P \leftarrow$ lista par różnych liter w w
4: **dla** $a \in L$ **wykonaj**
5: KompBlok(a, w)
6: **dla** $ab \in P$ **wykonaj**
7: KompPar(ab, w)

Przez *fazę* algorytmu Kompresja rozumiemy jedną iterację głównej pętli. Łatwo pokazać, że jedna faza zmniejsza długość w o stały czynnik.

Lemat 1. *Niech a, b będą dwiema kolejnymi literami słowa w na początku fazy. Na koniec fazy przynajmniej jedna z tych liter uległa kompresji. W szczególności, jeśli w' jest słowem na końcu fazy, to $|w'| \leq \frac{2|w|+2}{3}$; co więcej,* Kompresja$(w)$ *ma $\mathcal{O}(\log |w|)$ faz.*

Dowód opiera się na prostej analizie przypadków i nie nastręcza żadnych trudności.

Znamy już algorytm, który sprawnie kompresuje słowa. Łato zauważyć, że jeśli równolegle skompresujemy nim dwa słowa, powiedzmy w_1 i w_2, uzyskując słowa w_1' i w_2', to $w_1 = w_2$ wtedy i tylko wtedy, gdy $w_1' = w_2'$. Uzasadnia to zastosowanie Kompresja równocześnie do obu stron równania słów, pozostaje pokazać, jak to zrobić.

Ustalmy rozwiązanie S oraz parę ab (dla $a \neq b$) i zastanówmy się, w jaki sposób konkretne wystąpienie ab znalazło się w $S(u)$.

Definicja 1. *Dla równania $u = v$, rozwiązania S oraz pary ab mówimy, że wystąpienie ab w $S(u)$ (lub $S(v)$) jest*

- wystąpieniem jawnym*, jeśli składa się tylko i wyłącznie z liter występujących w u (lub v);*
- wystąpieniem niejawnym*, jeśli składa się tylko i wyłącznie z liter pochodzących z $S(X)$ dla ustalonego wystąpienia jakiejś zmiennej X;*
- wystąpieniem krzyżującym *w pozostałych przypadkach.*

Para ab jest krzyżująca *(dla rozwiązania S) jeśli ma choć jedno wystąpienie krzyżujące; jest* nie krzyżująca *(dla S) w przeciwnym przypadku.*

Analogicznie definiujemy wystąpienia jawne, niejawne *i* krzyżujące *dla bloków litery a; a jest* krzyżujące*, jeśli jakiś jej blok ma krzyżujące wystąpienie. (Równoważnie: aa jest krzyżujące).*

Przykład 1. Równanie $aaXbbabababa = XaabbYabX$ ma jedyne rozwiązanie $S(X) = a$, $S(Y) = abab$. Para ba jest krzyżująca (bo Y jest poprzedzone literą b oraz za Y jest litera a), para ab nie jest krzyżująca. Litera b nie jest krzyżująca, litera a jest krzyżująca (bo X jest poprzedzone literą a po lewej stronie oraz po X jest litera a po prawej stronie równania).

Algorytm 4 KompPar$(ab, \text{`}u = v\text{'})$ Kompresja pary ab w równaniu $u = v$

1: niech $c \in \Sigma$ będzie nie używaną literą
2: zastąp wszystkie wystąpienia ab w $u = v$ przez c

Algorytm 5 KompBlok$(a, \text{`}u = v\text{'})$ Kompresja bloków litery a w równaniu '$u = v$'

1: **dla** $\ell > 1$ **wykonaj**
2: niech $a_\ell \in \Sigma$ będzie nie używaną literą
3: zastąp wszystkie maksymalne bloki a^ℓ w '$u = v$' przez a_ℓ

Ustalmy parę ab oraz rozwiązanie S równania $u = v$. Jeśli ab jest nie krzyżująca, wykonanie KompPar$(ab, S(u))$ jest proste: należy zastąpić każdą parę wystąpień jawnych (co robimy bezpośrednio na równaniu) oraz każdą parę wystąpień niejawnych, co w pewnym sensie dzieje

się „samo”, gdyż rozwiązanie nie jest nigdzie zapisane. Ze względu na podobieństwo do kompresji pary w zadanym jawnie słowie (KompPar) będziemy mówili o KompPar$(ab, `u = v')$, które dokonuje kompresji par w równaniu. Podany argument pokazuje, że wykonanie takiej kompresji zachowuje spełnialność równania. To samo dotyczy kompresji bloków, nazywanej dla uproszczenia KompBlok$(a, `u = v')$. Łatwo też pokazać, że jeśli otrzymane w ten sposób równanie ma rozwiązanie, to również oryginalne równanie ma rozwiązanie (otrzymujemy je przez wymianę liter c na ab, argument dla kompresji bloków jest identyczny).

Lemat 2. *Niech równania $u = v$ ma rozwiązanie S, takie że ab jest nie krzyżujące dla tego rozwiązania. Wtedy równanie $u' = v'$ otrzymane przez zastosowanie* KompPar$(ab, `u = v')$ *jest spełnialne.*

Jeśli otrzymane równanie $u' = v'$ jest spełnialne, to również równanie $u = v$ jest spełnialne. To samo odnosi się do KompBlok$(a, `u = v')$.

Niestety, ta obserwacja nie pozwala jeszcze zasymulować algorytmu Kompresja(w) bezpośrednio na równaniu: w ogólności nie możemy bowiem zapewnić, że para ab (litera a) jest nie krzyżująca oraz nie wiemy, jak wyglądają pary, które mają tylko wystąpienia niejawne. Okazuje się, że drugi problem w zasadzie nie występuje: jeśli ograniczymy się do rozważania rozwiązań najkrótszych, każde para mająca wystąpienie niejawne ma też wystąpienie krzyżujące lub jawne, podobna uwaga odnosi się do bloków.

Lemat 3 ([101])**.** *Niech S będzie najkrótszym rozwiązaniem równania $`u = v'$. Wtedy:*

- *Jeśli ab jest podsłowem $S(u)$, gdzie $a \neq b$, to a, b mają wystąpienia jawne w równaniu zaś ab ma też wystąpienie jawne lub krzyżujące*
- *Jeśli a^k jest maksymalnym blokiem w $S(u)$ to a ma wystąpienie jawne w równaniu oraz a^k ma wystąpienie jawne lub krzyżujące w $`u = v'$.*

Dowód jest prosty: gdyby para (blok) miała tylko wystąpienia niejawne, to można je usunąć z rozwiązania, otrzymując krótsze rozwiązanie, co przeczy temu, że rozważaliśmy rozwiązanie najkrótsze.

Wracając do kwestii par krzyżujących: jeśli ustalimy parę ab (literę a), to łatwo uczynić ją nie krzyżującą: z Definicji 1 dość łatwo wysnuć wniosek, że para ab jest krzyżująca, jeśli dla pewnych zmiennych X oraz Y (niekoniecznie różnych) spełniony jest jeden z następujących warunków (zakładamy, że rozwiązanie nie przydziela żadnej zmiennej słowa pustego — jest to uzasadnione, gdyż w takim przypadku możemy po prostu usunąć taką zmienną z równania):

(CP1) aX występuję w równaniu oraz $S(X)$ rozpoczyna się od b;
(CP2) Yb występuję w równaniu oraz $S(Y)$ kończy się na a;
(CP3) YX występuję w równaniu, oraz $S(X)$ rozpoczyna się od b, zaś $S(Y)$ kończy się na a.

W każdym z tych przypadków postępowanie jest naturalne: w pierwszym ‘wypychamy’ z X literę b na lewo, w drugim a z Y na prawo, w trzecim dokonujemy obu tych operacji. Okazuje się, że rozwiązanie może być jeszcze bardziej systematyczne: w ogóle nie patrzymy na wystąpienia, a jedynie na rozwiązanie $S(X)$ dla każdej zmiennej.

- jeśli $S(X)$ zaczyna się od b, to zastępujemy wystąpienia X przez bX (zmieniając rozwiązanie $S(X) = bw$ na $S'(X) = w$), jeśli w nowym rozwiązaniu $S(X) = \epsilon$, to znaczy jest puste, to usuwamy X z równania;
- jeśli $S(X)$ kończy się na a, to postępujemy symetrycznie.

Algorytm ten nazywamy Wypchnij.

Algorytm 6 Wypchnij$(a, b, `u = v')$

```
1: dla X: zmienna wykonaj
2:     jeśli pierwsza litera S(X) to b to                          ▷ Zgadnij
3:         zastąp każde X w 'u = v' przez bX
                          ▷ Zmieniamy rozwiązanie S(X) = bw na S(X) = w
4:         jeśli S(X) = ε to                                       ▷ Zgadnij
5:             usuń X z u i v
6:     ...                    ▷ Wykonaj symetryczną operację dla ostatniej litery i a
```

Łatwo zauważyć, że przy odpowiednim wykonaniu uzyskane równanie ma rozwiązanie, dla którego ab nie jest krzyżujące: dla przykładu, jeśli aX występuje w równaniu i $S(X)$ zaczyna się od b, to zgadujemy ten fakt i otrzymujemy abX; dowód wymaga dokładnego zdefiniowania lematu oraz sprawdzenia kilku przypadków, ale nie nastręcza trudności.

Lemat 4. *Jeśli równanie $u = v$ ma rozwiązanie to po odpowiednim wykonaniu* Wypchnij$(a, b, `u = v')$ *(dla właściwych wyborów niedeterministycznych) otrzymane równanie ma odpowiadające rozwiązanie, dla którego ab nie jest parą krzyżującą.*

Jeśli otrzymane równanie ma rozwiązanie, to również oryginalne równanie było spełnialne.

Tym samym wiemy już, jak postępować z kompresją pary krzyżującej ab: należy najpierw sprawić, że nie jest ona krzyżująca (Wypchnij) a potem skompresować tak, jak parę nie krzyżującą (KompPar).

Podobnie chcielibyśmy postąpić w przypadku kompresji bloków. Dla bloków nie krzyżujących analogiczny algorytm KompBlok$(a, `u = v')$ jest zdefiniowany naturalnie. Pozostaje powiedzieć, jak sprawić, że a nie jest literą krzyżującą. Niestety, jeśli aX występuje w równaniu i $S(X)$ zaczyna się od a to po zastąpieniu X przez aX wciąż może zdarzyć się, że $S(X)$ dalej zaczyna się od a. Nie pozostaje nam nic innego, jak iterowanie tej procedury tak długo, aż pierwszą literą nie będzie a (co uwzględnia możliwość, że usuniemy całą zmienną X). Zauważmy, że zamiast robić to litera po literze, możemy dokonać tej operacji w jednym kroku: wystarczy usunąć ze zmiennej X a-prefiks i a-sufiks słowa $S(X)$ (jeśli $w = a^\ell w' a^r$, gdzie w' nie zaczyna i nie kończy się na a, a-prefiks w to a^ℓ, a a-sufiks to a^r; jeśli $w = a^\ell$ to a-sufiks jest pusty). Algorytm ten nazwiemy WypPrefSuf

Algorytm 7 WypPrefSuf$(a, `u = v')$ Wypychanie prefiksów i sufiksów

```
1: dla X: zmienna wykonaj
2:     zgadnij długość ℓ, r a-prefiksu i sufiksu S(X)          ▷ S(X) = a^ℓ w a^r
                                                  ▷ Jeśli S(X) = a^ℓ to r = 0
3:     zastąp wystąpienia X w u i v przez a^ℓ X a^r     ▷ ℓ, r są zapisane jako ciągi binarne
4:                                  ▷ zamień rozwiązanie S(X) = a^ℓ w b^r na S(X) = w
5:     jeśli S(X) = ε to                                           ▷ Zgadnij
6:         usuń X z u and v
```

Tak samo jak w przypadku Wypchnij można pokazać, że po odpowiednim wykonaniu WypPrefSuf otrzymane równanie ma (odpowiadające) rozwiązanie w którym a nie jest krzyżujące. Niestety, pojawia się problem: musimy zapisać długości ℓ oraz r dla prefiksów i sufiksów. Możemy je zapisać po prostu jako liczby binarne, wtedy będą używały $\mathcal{O}(\log \ell + \log r)$ bitów,

ale w ogólności są one dowolnie duże. Na szczęście, jesteśmy w stanie pokazać, że w *niektórych* rozwiązaniach te wartości są wykładnicze (czyli ich zapis: wielomianowy). Wniosek ten można z łatwością wysnuć z ograniczenia na wykładnik okresowości [60], jest on dokładniej omówiony w rozdziale 2.1.2. W chwili obecnej wystarczy nam wiedzieć, że

Lemat 5 ([60])**.** *W rozwiązaniu najmniejszym równania $u = v$ każdy a-prefiks i a-sufiks ma długość najwyżej wykładniczą (od $|u| + |v|$).*

Tym samym w algorytmie WypPrefSuf wystarczy że ograniczymy się do a-prefiksów oraz a-sufiksów długości wykładniczej.

Lemat 6. *Niech $u = v$ ma najkrótsze rozwiązanie S, po odpowiednim wykonaniu* WypPrefSuf$(a, `u = v')$ *(dla właściwych wyborów niedeterministycznych) otrzymane równanie $u' = v'$ ma odpowiadające rozwiązanie S', dla którego a nie jest literą krzyżującą, co więcej, otrzymane a-sufiksy i prefiksy mają najwyżej wykładniczą długość oraz $S(u) = S(u')$.*

Jeśli otrzymane równanie $u' = v'$ ma rozwiązanie, to również oryginalne równanie było spełnialne.

Po zastosowaniu Wypchnij możemy skompresować a-bloki używając algorytmu KompBlok$(a, `u = v')$, zauważmy, że po jego wykonaniu długie bloki liter a zostaną zastąpione przez pojedyncze litery.

Jesteśmy już gotowi, aby zasymulować Kompresja bezpośrednio na równaniu: zgodnie z algorytmem, wykonujemy najpierw kompresję bloków, najpierw dla liter nie krzyżujących, potem dla krzyżujących. Następnie kompresujemy pary, również najpierw dla par nie krzyżujących, a potem dla krzyżujących.

Algorytm 8 SpełRówSłów Rozstrzyganie spełnialności równania w słowach

dopóki $|u| > 1$ lub $|u| > 1$ **wykonaj** ▷ Równanie nie jest trywialne
  $L \leftarrow$ lista liter w u, v ▷ Obecne w równaniu
  $P \leftarrow$ lista par różnych liter w u, v ▷ Obecne w równaniu lub krzyżujace
  $L_1 \leftarrow$ lista liter nie krzyżujących ▷ Zgadnij
  dla $a \in L_1$ **wykonaj**
    KompBlok$(a, `u = v')$ ▷ Kompresujemy litery nie krzyżujące
  dla $a \in L \setminus L_1$ **wykonaj**
    WypPrefSuf$(a, `u = v')$ ▷ a przestaje być literą nie krzyżującą
    KompBlok$(a, `u = v')$ ▷ Kompresujemy
  $P_1 \leftarrow$ lista par nie krzyżujących z P ▷ Zgadnij
  dla $ab \in P_1$ **wykonaj**
    KompPar(ab, w) ▷ Kompresujemy pary nie krzyżujące
  dla $ab \in P \setminus P_1$ **wykonaj**
    Wypchnij$(a, b, `u = v')$ ▷ ab przestaje być parą krzyżującą
    KompPar(ab, w) ▷ Kompresujemy pary nie krzyżujące
Rozwiąż problem naiwnie ▷ Gdy strony równania mają długość 1, problem jest prosty

Poprawność algorytmu wynika z poprzednich rozważań na temat poprawności KompBlok, WypPrefSuf, KompPar i Wypchnij. Jeśli ograniczymy się do rozwiązań najkrótszych, to dodatkowo symulujemy Kompresja na najkrótszym rozwiązaniu. Tym samym algorytm ten będzie

miał jedynie $\log N$ iteracji głównej pętli (*faz*), gdyż w każdej z faz długość najkrótszego rozwiązania spada o stały czynnik, przez analogię do Lematu 1.

Lemat 7. *Algorytm* SpełRówSłów *ma* $\mathcal{O}(\log N)$ *faz, gdzie* N *to długość najkrótszego rozwiązania wejściowego rozwiązania.*

Jak na razie osiągnąłem jedynie słabszą wersję wyniku W. Plandowskiego oraz W. Rytter a [101], gdyż nie wiemy nawet, ile trwa jedna faza — jej czas działania zależy oczywiście od wielkości równania, a ta nie jest jak na razie znana. Spróbujmy więc ograniczyć rozmiar równania. W oczywisty sposób rośnie ono w czasie algorytmu: każda wypchnięcie liter i prefiksów zwiększa jego długość; na szczęście jesteśmy w stanie stwierdzić, o ile: Wypchnij dla każdej pary krzyżującej wprowadza $2n$ liter (po jednej na każdą „stronę" zmiennej), tak samo WypPrefSuf (zauważmy, że formalnie WypPrefSufwprowadza długie ciągi, ale są one zaraz potem zastępowane przez pojedyncze litery, dlatego możemy myśleć o tym, że wprowadzamy jedynie $2n$ liter). Jednocześnie charakteryzacja (CP1)–(CP3) pozwala łatwo stwierdzić, że jest najwyżej $2n$ par krzyżujących i najwyżej $2n$ liter krzyżujących (bo każdą parę krzyżującą i literę krzyżująca możemy związać z wystąpieniem zmiennej i jej jedną „stroną").

Lemat 8. *W jednej fazie* SpełRówSłów *wprowadza* $\mathcal{O}(n^2)$ *liter do równania.*

Jednocześnie, rekompresja kompresuje każdy fragment słowa w instancji: argument w Lemacie 1 nie zakłada nigdzie, że mówimy o całym słowie, możemy go też stosować do podsłów w równaniu. Tym samym, jeśli równanie po jednej fazie to $u' = v'$, otrzymujemy oszacowanie na długość równania

$$|u' = v'| \leq \frac{2}{3}|u = v| + cn^2$$

dla pewnej stałej c. Ta zależność pozwala na pokazanie, że wielkość równania pozostaje kwadratowa. Co do dodatkowego zużycia pamięci, pozostaje jedynie ocenić, ile miejsca jest potrzebne do zapisania długości bloków. Jako że ograniczamy się do rozwiązań najkrótszych, są one wykładnicze, czyli potrzebują wielomianowej pamięci (dokładna analiza pokazuje, że starczy $\mathcal{O}(n^2)$ bitów). Co daje, że algorytm SpełRówSłów działa w pamięci kwadratowej i tym samym w PSPACE.

Lemat 9. SpełRówSłów *działa w pamięci* $\mathcal{O}(n^2)$.

Powyższą analizę da się istotnie poprawić.

Wynik 1 ([H3])**.** *Algorytmu oparty na rekompresji (niedeterministycznie) rozstrzyga problem spełnialności równania w słowach w pamięci* $\mathcal{O}(n \log n)$ *(liczonej w bitach). Co więcej, przechowywane równanie ma liniową długość.*

2.1.1. *Wielkość rozwiązania.* Jedną z kluczowych kwestii dotyczących rozwiązań równań w słowach jest wielkość najkrótszego rozwiązania. Pierwsze oszacowania pochodziły od G. Makanina, liczne ulepszenia jego argumentu dały ostatecznie potrójnie wykładnicze oszacowanie [44], obecnie najlepsze znane oszacowanie pochodzi od W. Plandowskiego i jest podwójnie wykładnicze [98]; oszacowanie to opiera się na analizie specjalnie zaprojektowanych faktoryzacji słów. Z drugiej strony, wiadomo jedynie, że najkrótsze rozwiązania mogą być wykładnicze, dość powszechne jest przekonanie, iż prawdziwe jest również wykładnicze oszacowanie górne, co dawałoby dowód NP-zupełności problemu spełniania równań w słowach [101].

Algorytm oparty na rekompresji umożliwia również uzyskanie podwójnie wykładniczego ograniczenia górnego na wielkość najkrótszego rozwiązania równania w słowach. Taki wniosek

jest naturalny: wiemy, iż algorytm działa w czasie $\mathsf{poly}(n, \log N)$, z drugiej strony używa $\mathsf{poly}(n)$ pamięci, czyli działa w czasie $\mathsf{exp}(n)$[4]. Zrównując te dwie wartości uzyskujemy, iż $\log N$ jest wykładnicze od n, czyli N jest podwójnie wykładnicze od n.

Niestety, oszacowania na pamięć oraz czas działania są ograniczeniami górnymi i tym samym nie można ich bezpośrednio porównać, taka operacja możliwa jest jedynie, gdy jedno oszacowanie jest oszacowaniem górnym, a drugie dolnym. Na szczęście, oszacowanie na czas działania jest dość dokładne w tym sensie, że możliwe jest również podanie oszacowania dolnego w zależności od $\log N$ oraz n, jest ono postaci $\log N/\mathsf{poly}(n)$: w jednej fazie jesteśmy w stanie „skleić" jedynie wykładniczo wiele (od n) liter w jedną literę (przy kompresji bloków), tym samym długość rozwiązania najkrótszego maleję najwyżej $\mathsf{exp}(n)$ razy w fazie i tym samym ilość faz to przynajmniej $\log_{\mathsf{exp}(n)}(N) = \log N/\mathsf{poly}(n)$. Oszacowanie $\log N/\mathsf{poly}(n)$ jest oszacowaniem dolnym i tym samym można je porównać z oszacowaniem górnym $\mathsf{exp}(n)$, uzyskując wykładnicze oszacowanie na $\log N$ (i tym samym podwójnie wykładnicze oszacowanie na N).

Wynik 2 ([H3])**.** *Używając algorytmu opartego na rekompresji można uzyskać podwójnie wykładnicze oszacowanie na długość najkrótszego rozwiązania równania w słowach.*

2.1.2. *Ograniczenie na wykładnik okresowości.* Wykładnik okresowości rozwiązania S równania $u = v$ w słowach to maksymalne k, takie że istnieje niepuste słowo $w^k \neq \epsilon$ będące podsłowem $S(u)$ oraz nie ma niepustego słowa w' takiego że $(w')^{k+1}$ jest podsłowem $S(u)$. Jednym z kluczowych etapów pracy G. Makanina było oszacowanie wykładnika okresowości najkrótszego rozwiązanie równania w słowach. Oszacowanie to zostało poprawione przez A. Kościelskiego i L. Pacholskiego [60]: jest on najwyżej wykładniczy (od n), oszacowanie to jest ścisłe. Jest ono jednym z częściej używanych wyników w teorii równań słów, w szczególności większość późniejszych prac dotyczących równań w słowach używała tego oszacowanie.

Algorytm oparty na rekompresji używa jedynie bardzo ograniczonej wersji tego oszacowania: niech Σ-wykładnik okresowości rozwiązania S to maksymalne k, takie że a^k jest podsłowem $S(u)$ dla pewnej litery a, natomiast b^{k+1} nie jest dla żadnej litery b. Łatwo zauważyć, że przy kompresji bloków korzystamy z ograniczenia na Σ-wykładnik okresowości najkrótszego rozwiązania (bo potrzebujemy oszacowania na ilość pamięci potrzebnej do zapisu długości wypchniętych bloków). Dowód tak uproszczonej wersji oszacowania na wykładnik okresowości jest istotnie prostszy, niż jego pełnej wersji. Co więcej, samego algorytmu opartego na rekompresji można użyć do udowodnienia oszacowania na wykładnik okresowości.

Wynik 3 ([H3])**.** *Używając algorytmu opartego na rekompresji można sprowadzić problem ograniczenia górnego na wykładnik okresowości do problemu ograniczenia górnego na Σ-wykładnika okresowości.*

Załóżmy, że rozwiązanie ma wykładnik okresowości k, w szczególności istnieje podsłowo w^k w $S(u)$. Co się stanie, jeśli dokonamy kompresji par lub kompresji bloków na tym rozwiązaniu? Jeśli w jest literą, to zostanie ono zastąpione przez jedną literę i tym samym stracimy informację o k, jednak z oszacowania na Σ-wykładnik okresowości wiemy, iż k jest najwyżej wykładnicze. Jeśli w nie jest literą, to w idealnym przypadku każda kopia w w w^k zostanie skompresowana osobno (i tak samo) i tym samym nowe rozwiązanie $S'(u')$ będzie zawierało słowo $(w')^k$ czyli wykładnik okresowości się zachowa (łatwo pokazać, że nie może on wzrosnąć wskutek kompresji par lub bloków). Niestety, kompresje mogą pojawić się pomiędzy kopiami

[4]Przez $\mathsf{exp}(n)$ określamy klasę funkcji wykładniczych od n.

w, wtedy musimy „przesunąć" nasze słowo tak, by zgadzało się z tymi kompresjami: zawsze można wybrać z w^k podsłowo w_1^{k-1}, takie że każda kopia w_1 w w_1^{k-1} jest kompresowana osobno. Tym samym wykładnik okresowości po kompresji wynosi przynajmniej $k-1$, tzn. spada o najwyżej 1 wskutek kompresji par lub bloków. Jako że algorytm wykonuje jedynie wykładniczo wiele kompresji, uzyskujemy żądane wykładnicze ograniczenie na wykładnik okresowości.

2.1.3. *Stała liczba zmiennych.* Używający przez rekompresję algorytm przechowuje równanie o liniowej długości, ale zużycie pamięci wynosi $\mathcal{O}(n \log n)$ (liczone w bitach), gdyż w ogólności każdy symbol równania może być inny. Tym niemniej, zużycie pamięci jest stosunkowo małe i naturalna jest pytanie, czy może zostać zmniejszone do $\mathcal{O}(n)$, tj. chcielibyśmy pokazać, że spełnialność równań w słowach można (niedeterministycznie) rozstrzygać w pamięci liniowej, czyli że są one językiem kontekstowym. Niestety, nie udało się pokazać takiego ograniczenia w pełnej ogólności, jednak udało się to w pewnym ograniczonym przypadku, gdy liczba zmiennych użytych w równaniu jest stała.

Wynik 4 ([H3])**.** *Oparty na rekompresji algorytm rozwiązywania równań w słowach używa $\mathcal{O}(mk^{ck})$ pamięci (liczonej w bitach), gdzie k jest ilością zmiennych, m ilością bitów potrzebnych do zapisania wejściowego równania, a c pewną stałą. Jeśli k jest stałe, to algorytm działa w pamięci liniowej, tj. problem jest kontekstowy.*

Jak zauważyliśmy, problemem jest sposób zapisu liter w równaniu. Ulepszenia oryginalnego algorytmu opierają się na używaniu dwóch sposobów zapisu:

- świeża litera zastępująca ab (lub a^ℓ) jest kodowana przy użyciu ciągu ab (odpowiednio: a^ℓ).
- pokazujemy, że różne fragmenty XwY w równaniu zachowują się tak samo o ile żadna ze zmiennych X, Y nie została usunięta z równania. Tym samym litery wypchnięte z X możemy etykietować $(XwY)\#i$, gdzie $\#i$ oznacza, że jest to i-ta litera wypchnięta przez X i jest zapisane binarnie.

Odpowiednie użycie tych metod pozwala ograniczyć pamięć do $\mathcal{O}(kn)$ (gdzie k to ilość zmiennych), dopóki ilość zmiennych nie zmienia się. Jako że mamy jedynie k różnych zmiennych, analizę tę stosujemy k razy, uzyskując żądane ograniczenie.

2.1.4. *Generowanie wszystkich rozwiązań.* Jak na razie zajmowaliśmy się jedynie sprawdzaniem spełnialności równania w słowach. W ogólności jedno równanie może mieć wiele rozwiązań i pożądany jest opis ich wszystkich. Po raz pierwszy taki opis podał W. Plandowski [100], pokażemy, że rekompresja może być użyta do uzyskania podobnego wyniku. Łatwo uwierzyć, że operacje kompresji par oraz kompresji bloków „zachowują" rozwiązania: jeśli S jest rozwiązaniem równania $u = v$ to możemy przeprowadzić kompresję par (bloków) na słowie $S(u)$ i zasymulować ją (używając odpowiednich wyborów niedeterministycznych) na równaniu $u = v$, uzyskując równanie $u' = v'$ z „odpowiadającym" rozwiązaniem $S'(u')$. Tym samym możemy utworzyć graf: jego wierzchołki będą etykietowane równaniami, a krawędzie między równaniem $u = v$ a równaniem $u' = v'$ będzie opisywać, w jaki sposób można uzyskać rozwiązanie równania $u = v$ z rozwiązań równania $u' = v'$ (zauważmy też, że wierzchołek $u = v$ może mieć wiele krawędzi do różnych innych wierzchołków). Opis tego przekształcenia jest naturalny: zastąpienie litery c przez parę ab, zastąpienie a_ℓ przez a^ℓ lub też doczepienie do $S(X)$ litery z jednej lub drugiej strony. Pozostaje jedynie zadbać, by wszystkie wierzchołki i krawędzie miały wielomianowy opis.

Niestety, pojawia się problem długości bloków: łatwo zauważyć, że równanie $aXXXX = XaYY$ ma rozwiązania postaci $S(X) = a^{\ell_X}$, $S(Y) = a^{\ell_Y}$, gdzie dodatkowo $4\ell_X + 1 = \ell_X + 1 + 2\ell_Y$. Równanie to ma nieskończenie wiele rozwiązań i zastąpienie X przez a^{ℓ_X} może zużyć dowolnie wiele pamięci i przekształcić to równania w nieskończenie wiele innych równań. Z drugiej strony, w kolejnym kroku nasz algorytm zastępuje bloki długości $4\ell_X + 1 = \ell_X + 2\ell_Y + 1$ nową literą i dokładna długość tego bloku nie jest istotna, ważna jest jedynie równość. W ogólności, kompresję bloków można poprawić tak, aby używała parametrów liczbowych zamiast konkretnych długości prefiksów i sufiksów. Pierwsze modyfikacja dotyczy algorytmu Wypchnij: wypycha on prefiks a^{ℓ_X} ze zmiennej X, jednak ℓ_X jest parametrem liczbowym, a nie konkretną liczbą. Następnie (niedeterministycznie) identyfikujemy bloki o tej samej długości i sprawdzamy, czy takie bloki faktycznie mogą być równe: zgadnięte równości odpowiadają układowi liniowych równań diofantycznych. Co więcej, każde możliwe rozwiązanie odpowiada jakiemuś rozwiązaniu tego układu. Tym samym, dzięki tej modyfikacji unikamy konieczności rozważania dowolnie długich równań i jesteśmy w stanie zapewnić, że równanie będzie miało zawsze wielomianową długość (zauważmy też, że ta modyfikacja w istocie uwalnia nas też od konieczności stosowania oszacowania na Σ-wykładnik okresowości). Niestety, efektem ubocznym jest to, że krawędzie w grafie będą etykietowane układami równań diofantycznych i każde rozwiązanie takiego równania odpowiada jednemu przekształceniu rozwiązań równań w słowach.

Wynik 5 ([H3])**.** *Używając algorytmu opartego na rekompresji można utworzyć (w pamięci wielomianowej) skończony graf reprezentujący wszystkie rozwiązania równania w słowach. Każdy wierzchołek i krawędź mają wielomianowe opisy, cały graf ma zaś najwyżej wykładniczo wiele wierzchołków.*

2.1.5. *Równania z jedną zmienną.* Jak już wspomniałem, jedna z badanych podklas równań w słowach były równania z jedną zmienną. Dosyć łatwo pokazać, iż można je rozstrzygnąć w czasie $\mathcal{O}(n^3)$, poprawienie czasu działania do kwadratowego wymaga jedynie drobnych obserwacji [22]. Pierwszy nietrywialny algorytm dla tego problemu działał w czasie $\mathcal{O}(n \log n)$ [87], natomiast K. Dąbrowski i W. Plandowski podali algorytm o czasie działania $\mathcal{O}(n + \#_X \log n)$ [25], gdzie $\#_X$ jest ilością wystąpień zmiennej X w oryginalnym równaniu.

Jeśli przyjrzymy się algorytmowi opartemu na rekompresji, dość łatwo pokazać, iż determinizuje się on w przypadku równań z jedną zmienną: algorytm ten dokonuje następujących wyborów niedeterministycznych:

- jaka jest pierwsza (ostatnia) litera $S(X)$
- jak jest długość prefiksu a^ℓ (sufiksu a^r) w $S(X)$.

Łatwo pokazać, że jeśli równanie ma tylko jedną zmienną, to obie te informacje są jednoznacznie określone przez równanie.

Lemat 10. *Bez straty ogólności równanie w słowach z jedną zmienną jest postaci*

$$A_0 X A_1 \dots A_{k-1} X A_k = X B_1 \dots B_{\ell-1} X B_\ell \ , \tag{1}$$

gdzie A_0 jest niepustym słowem i dokładnie jedno ze słów A_k, B_ℓ jest puste.

Niech pierwsza litera A_0 to a. Każde rozwiązanie $S(X) \notin a^+$ ma a-prefiks taki sam jak a-prefiks A_0; analogiczny fakt zachodzi też dla a-sufiksów.

Lemat 10 pozwala na prosty algorytm oparty na rekompresji: w każdej fazie identyfikujemy potencjalne rozwiązanie z a^+, gdzie a jest pierwszą literą A_0, i weryfikujemy czy jest to istotnie

rozwiązanie. Następnie wykonujemy rekompresję: pozostałe rozwiązania mają taki sam a-prefiks.

Naturalne implementacja tego algorytmu ma ten sam czas działania, co algorytm R. Dąbrowskiego i W. Plandowskiego, tj. $\mathcal{O}(n+\#_X \log n)$. Możliwe jest poprawienie czasu działania do liniowego, wymaga to jednak wielu nietrywialnych poprawek do algorytmu i stosowania wydajnych struktur danych (tablice sufiksowe ze strukturą do liczenia lcp, tj. najdłuższego wspólnego prefiksu). W szczególności:

- Algorytm przechowuje układ równań i czasami rozbija jedno równanie na mniejsze, co pozwala na zmniejszenie rozmiaru. Co więcej, zapamiętujemy, które słowa w równaniu są równe i przechowujemy tylko jedną kopię takich słów.
- Pokazujemy, że pewną klasę rozwiązań algorytm znajduje w $\mathcal{O}(1)$ faz, następnie w wielu miejscach udowadniamy, że problematyczne przypadki są w istocie z tej klasy.
- Poprawiamy procedurę testowania rozwiązania (niektóre potencjalne rozwiązania są odrzucane ze względu na swoje własności strukturalne) oraz używamy dużo dokładniejszej analizy kosztu testowania: dla każdego słowa w równaniu z osobna liczymy, czy brało udział w teście, tym samym czasem jesteśmy w stanie ustalić, że testowanie trwało dużo krócej, niż czas potrzebny do przeczytania całego równania.

Wynik 6 ([H6])**.** *Używając algorytmu opartego na rekompresji w liniowym czasie możemy podać wszystkie rozwiązania równania w słowach z jedną zmienną.*

2.1.6. *Równania z więzami regularnymi oraz inwersją; równania w grupie.* Jak już zostało powiedziane, naturalne i ważne jest rozszerzenie równań w słowach o więzy regularne oraz inwersję, w szczególności pozwala ono rozwiązać problem spełnialności równań w grupie wolnej [23] (redukcja między tymi problemami jest czysto syntaktyczna i nie zależy w żaden sposób od algorytmu rozwiązującego równania w słowach).

Dodanie więzów regularnych do opartego na rekompresji algorytmu SpełRówSłów nie nastręcza większych trudności: wszystkie takie więzy można zakodować przy użyciu jednego automatu niedeterministycznego (warunki dla zmiennych różnią się jedynie zbiorem stanów akceptujących). Dla każdej litery c przechowujemy *funkcja przejścia*; tj. funkcję $f_c : Q \mapsto 2^Q$ mówiącą, iż automat po przeczytaniu tej litery przechodzi ze stanu q do jednego ze stanów $f_c(q)$. Funkcję tę naturalnie rozszerzamy na słowa: dalej określa stany, które możemy otrzymać po przeczytaniu słowa w (startując ze stanu q); formalnie $f_{wa} = (f_w \circ f_a)(q) = \{p \mid \exists q' \in f_w(q) \text{ i } p \in f_a(q')\}$ dla litery a. Jeśli wprowadzamy nowa literę c (która zastępuje słowo w), to w naturalny sposób rozszerzamy funkcję przejścia: $f_c \leftarrow f_w$. Więzy regularne dla zmiennych można wyrazić w kategoriach tej funkcji: stwierdzenie, że $S(X)$ ma byś zaakceptowane przez automat oznacza, że $f_{S(X)}(q_0)$ jest jednym ze stanów akceptujących. Wystarczy więc zgadnąć wartość $f_{S(X)}$, która spełnia ten warunek; możemy więc mówić o wartości funkcji f_X dla zmiennej X. Wypchnięcie litery ze zmiennej sprawia, że musimy poprawić funkcję przejścia, tj. jeśli zastępujemy X przez aX to chcemy, aby $f_X = f_a \circ f_{X'}$, analogicznie definiujemy f_X gdy wypychamy litery na prawo.

Więcej problemów nastręcza *inwersja*: intuicyjnie odpowiada ona odwrotności w grupie, natomiast na poziomie półgrupy wymagamy, że $\overline{\overline{a}} = a$ dla każdej litery a oraz $\overline{a_1 a_2 \dots a_m} = \overline{a_m} \dots \overline{a_2}\,\overline{a_1}$. Taka definicja ma konsekwencje dla kompresji: gdy kompresujemy parę ab do c, to powinniśmy jednocześnie zastąpić słowo $\overline{ab} = \overline{b}\,\overline{a}$ przez literę $\overline{c}$. Na pierwszy rzut oka jest to proste, problemy pojawiają się, gdy te dwie pary nie są rozłączne, tzn. gdy $\overline{a} = a$ (lub $\overline{b} = b$), których to możliwości nie można w ogólności wykluczyć. W takim wypadku w ciągu $ba\overline{b}$ przy kompresji pary ba chcemy jednocześnie zastąpić ba oraz $a\overline{b}$, co nie jest możliwe.

Aby temu zaradzić, zastępujemy jednocześnie maksymalne spójne fragmenty, które dają się pokryć parami ab lub $\overline{b}\overline{a}$, w tym wypadku jest to cała trójka $ba\overline{b}$. W najgorszym wypadku (gdy $a = \overline{a}$ i $b = \overline{b}$) musimy zastępować ciągi postaci $(ab)^n$, co łączy w sobie cechy kompresji par i bloków.

W tym momencie rozszerzenie rekompresji do półgrupy z inwersją nie nastręcza już trudności.

Wynik 7 ([H8])**.** *Oparty na rekompresji algorytm może w pamięci wielomianowej wygenerować opis wszystkich rozwiązań równania słów w półgrupie wolnej z inwersją i więzami regularnymi jak również równania z więzami regularnymi w grupie wolnej.*

2.2. **Rekompresja i skompresowane dane.** Metoda rekompresji jest inspirowana pochodzącymi z algorytmiki technikami [82, 2]. W tym rozdziale pokażemy, że jest ona w stanie spłacić ten dług: wiele czysto algorytmicznych problemów dla skompresowanych danych da się w ten sposób rozwiązać. Co zaskakujące, biorąc pod uwagę ogólność metody, podane rozwiązania są nie gorsze, a czasami wręcz lepsze, niż obecnie istniejące.

2.2.1. *Straight Line Programs i rekompresja.* Formalnie, *Straight Line Programme* (*SLP*) to gramatyka, której nieterminale są liniowo uporządkowane (bez zmniejszenia ogólności: X_1, X_2, ..., X_m), i dodatkowo każdy nieterminal ma dokładnie jedną produkcję oraz jeśli X_j występuje w produkcji dla X_i to $j < i$. Będziemy używali liter $\mathcal{A}$, $\mathcal{B}$, itp. na oznaczenie SLP. Tym samym każdy nieterminal X_i generuje dokładnie jedno słowo, które nazywamy $\mathrm{val}(X_i)$, natomiast całe SLP $\mathcal{A}$ definiuje słowo $\mathrm{val}(\mathcal{A}) = \mathrm{val}(X_m)$.

W naturalny sposób SLP można traktować jako układ równań słów (ze zmiennymi $X_1, \ldots, X_m$): produkcję $X_i \to \alpha_i$, traktujemy jako równość $X_i = \alpha_i$; zauważmy, że równość ta ma sens, gdyż $\mathrm{val}(X) = \mathrm{val}(\alpha)$ (gdzie val jest rozszerzone do ciągów liter i nieterminali w naturalny sposób), co więcej, nie ma innych rozwiązań takiego równania. Tym samym metodę rekompresji można stosować do SLP (co prawda jak na razie stosowaliśmy ją jedynie do pojedynczego równania, lecz bez żadnego problemu uogólnia się ona też do układów równań). W szczególności, ograniczenie z Lematu 9 wciąż się stosują, co pokazuje, że w czasie rekompresji rozważane SLP ma rozmiar wielomianowy.

Do rozwiązania pozostaje kwestia efektywności: rekompresja dla równań w słowach jest wysoce niedeterministyczna (choć, jak zauważyliśmy, nie w przypadku równania z jedna zmienną), natomiast algorytmy dla SLP powinny, o ile to możliwe, być deterministyczne (w istocie, często zależy nam nawet na wydajnych algorytmach, tj. na obniżeniu stopnia wielomianu).

Na szczęście, łatwo zauważyć, że niedeterminizm rekompresji można w przypadku SLP usunąć: jest on jedynie w:

(1) ustaleniu, czy $\mathrm{val}(X_i) = \epsilon$;
(2) ustaleniu pierwszej (ostatniej) litery $\mathrm{val}(X_i)$;
(3) ustaleniu długości a-prefiksu i a-sufiksu $\mathrm{val}(X_i)$.

Zauważmy, że na wszystkie te pytanie łatwo odpowiedzieć, jeśli znamy już odpowiedzi dla X_j dla $j < i$: niech $X_i \to \alpha_i$, najpierw usuwamy z α_i wszystkie nieterminale X_j, dla których $\mathrm{val}(X_j) = \epsilon$, następnie

(1) $\mathrm{val}(X_i) = \epsilon$ wtedy i tylko wtedy gdy $\alpha_i = \epsilon$;
(2) pierwsza litera $\mathrm{val}(X_i)$ to pierwsza litera α_i lub pierwsza litera $\mathrm{val}(X_j)$, jeśli pierwszym symbolem w α_i jest X_j;

(3) długość a-prefiksu zależy tylko od liter a w α_i oraz długości a-prefiksów nieterminali w α_i (i symetrycznie dla sufiksu).

Wszystkie te warunki da się sprawdzić w liniowym czasie, tym samym rekompresja dla SLP działa w czasie wielomianowym od wielkości SLP, czyli wielomianowym.

Lemat 11. *Rekompresja na SLP ma wielomianowy czas działania.*

2.2.2. *Równość SLP oraz wyszukiwanie skompresowanego wzorca w skompresowanym tekście.* Jednym z pierwszych (i zapewne najważniejszych) problemów dla SLP jest sprawdzanie równości, to jest dla dwóch SLP stwierdzenie, czy zadają one to samo słowo. Pierwszy wielomianowy algorytm dla tego problemu podał w 1994 roku W. Plandowski [97], dokładniej, algorytm ten działał w czasie $\mathcal{O}(n^4)$. Dalsze badania tego problemu skupiły się głównie na ogólniejszym problemie *w pełni skompresowanego wyszukiwania wzorca*: dla zadanych SLP $\mathcal{A}$ oraz $\mathcal{B}$ mamy stwierdzić, czy $\mathrm{val}(\mathcal{A})$ występuje w $\mathrm{val}(\mathcal{B})$ (jako podsłowo). Pierwsze rozwiązanie tego problemu podali w 1995 M. Karpiński i in. [53]. L. Gąsieniec i in. [40] zaprezentowali szybszy algorytm zrandomizowany. W 1997 M. Miyazaki i in. [83] skonstruowali algorytm działające w czasie $\mathcal{O}(n^4)$; szybszy algorytm działający w czasie $\mathcal{O}(n^2)$ dla pewnej podklasy SLP został podany w 2000 r. przez M. Hirao i in. [46]. Ukoronowaniem tych badań był algorytm Yu. Lifshitsa, o czasie działania $\mathcal{O}(n^3)$ [72]. Warto zaznaczyć, że wszystkie rozszerzały i usprawniały technikę zaproponowaną przez W. Plandowskiego

Rekompresja w naturalny sposób nadaje się do sprawdzania równości SLP: dla zadanych SLP dodajemy $\mathcal{A}$ oraz $\mathcal{B}$ równanie $X_{m_\mathcal{A}} = Y_{m_\mathcal{B}}$ i pytamy o jego spełnialność. Jak już zauważyliśmy, algorytm oparty na rekompresji będzie miał wielomianowy czas działania (Lemat 11). Okazuje się, iż przy odpowiedniej implementacji (z wykorzystaniem wielu nietrywialnych rozwiązań algorytmicznych) możliwa jest implementacja działająca w czasie $\mathcal{O}((|\mathcal{A}|+|\mathcal{B}|)\log N)$, gdzie $N = |\mathrm{val}(\mathcal{A})| = |\mathrm{val}(\mathcal{B})|$ (jeśli $|\mathrm{val}(\mathcal{A})| \neq |\mathrm{val}(\mathcal{B})|$ to oczywiście $\mathcal{A}$ oraz $\mathcal{B}$ nie są równe).

W celu uzyskania tak małego czasu działania potrzebne są również optymalizacje. Najistotniejsze z nich to zauważenie, że część kompresji można wykonywać równolegle:

- Wszystkie kompresje bloków, również dla różnych liter, można wykonać jednocześnie.
- Jeśli Σ_ℓ i Σ_r są rozłączne, to kompresje wszystkich par ab, takich że $a \in \Sigma_\ell$ oraz $b \in \Sigma_r$ również można wykonać równocześnie.
- Nie dokonujemy kompresji wszystkich par, kompresujemy jedynie pary z $\mathcal{O}(1)$ partycji Σ_ℓ, Σ_r, które pokrywają „dużo" liter w SLP oraz słowie definiowanym przez to SLP.

Wynik 8 ([H2])**.** *Oparty na rekompresji algorytm sprawdzający równość dwóch SLP ma czas działania $\mathcal{O}(n \log N)$, gdzie n jest sumą rozmiarów SLP zaś N ich (zdekompresowaną) długością.*

Niestety, aby zastosować tę metodę do przypadku wyszukiwania wzorca niezbędne są pewne modyfikacje: zauważmy, że jeśli dla wzorca ab oraz tekstu bab dokonamy kompresji pary ba, uzyskując ab oraz cb, to utracimy jedyne wystąpienie wzorca w tekście. Intuicyjnie, problem bierze się z wykonania kompresji częściowo na i częściowo poza wystąpieniem wzorca w tekście. Aby tego uniknąć, wykonujemy operacje w określonej kolejności, która zależy od pierwszych i ostatnich liter wzorca i tekstu. (W podanym przykładzie, wykonanie kompresji ba zachowuje wystąpienia wzorca). Podobne podejście trzeba też zastosować do kompresji bloków.

Wynik 9 ([H2])**.** *Oparty na rekompresji algorytm sprawdzający występowanie skompresowanego wzorca w skompresowanym tekście ma czas działania $\mathcal{O}(n \log N)$, gdzie n jest sumą rozmiarów SLP zaś N długością (zdekompresowanego) wzorca.*

2.2.3. *Problemy decyzyjne dla skompresowanych automatów.* Problemem, w którym po raz pierwszy zastosowano technikę rekompresji, jest problem rozpoznawania skompresowanego słowa przez skompresowany automat.

Popularność kompresji każe przemyśleć ponownie znane i dobrze zbadane problemy: co zmieni się, jeśli dopuścimy, by ich wejście zadane było w skompresowany sposób? Z punktu widzenia języków formalnych, najważniejsze pytania dotyczą akceptowania (lub rozpoznawania) słów przez rozmaite automaty i gramatyki. Tego typu problemy po raz pierwszy badali W. Plandowski i W. Rytter [102] i od tamtego czasu ustalono złożoność obliczeniową wielu tego typu problemów [7, 73, 74, 75]. Jednym z ciekawszych (i najwidoczniej trudniejszych) problemów, który pozostał otwarty przez wiele lat, jest wspomniany już problem rozpoznawania skompresowanego słowa przez skompresowany automat. W problemie tym nie tylko wejście automatu podane jest w sposób skompresowany (jako SLP), lecz również sam automat jest skompresowany, tj. jego przejścia etykietowane są nie tyle literami, co całymi słowami i słowa te są zadane w skompresowany sposób, tzn. jako SLP (akceptowanie przez taki automat zadane jest w naturalny sposób). Dosyć łatwo pokazać, że problem ten jest NP-trudny i o kilku przypadkach szczególnych pokazano, iż istotnie są w NP [102, 78]. Dość powszechnie wierzono, że problem ten w istocie jest w NP.

Zastosowanie rekompresji na SLP w instancji wymaga modyfikacji automatu, są one jednak dość proste:

- Gdy zastępujemy ab przez c to jeśli ze stanu p do q da się przejść słowem ab, dodajemy przejście przez c.
- Analogicznie postępujemy dla a^ℓ. Zauważmy, że ℓ może być wykładnicze, jednak weryfikacja, czy takie przejście jest poprawne jest możliwe w NP [102]. W tym miejscu nasz algorytm używa niedeterminizmu (przypomnijmy, że gdzieś ten niedeterminizm wystąpić „musi", gdyż problem jest NP-trudny).
- Gdy zastępujemy X przez aX to dla przejście automatu z p do q etykietowanego przez X dodajemy stan p', usuwamy to przejście przez X, następnie zadajemy przejścia $\delta(p, a, p')$, $\delta(p', X, q)$. Podobną operację wykonujemy, gdy zastępujemy X przez Xb lub X przez $a^\ell X$, w ostatnim przypadku przejście jest etykietowane blokiem a^ℓ, jednak po kompresji bloków można je usunąć, gdyż rozważane przez nas słowo ich już nie zawiera.

Łatwo pokazać, że algorytm dodaje jedynie wielomianowo wiele stanów w czasie całego swojego działania. Sprawdzenie poprawności algorytmu oraz jego wielomianowości jest rutynowe, tym samym algorytm oparty na rekompresji rozwiązuje ten problem w NP, tak jak przewidywano.

W momencie ustalenia złożoności obliczeniowej problemu rozpoznawania skompresowanego słowa przez skompresowany automat (niedeterministyczny), naturalna jest również próba określenia złożoności obliczeniowej w przypadku automatu deterministycznego, tym bardziej, że dowód NP-trudności wymaga niedeterminizmu w automacie. Przypomnijmy, że niedeterminizm potrzebny był jedynie przy sprawdzaniu, czy między stanami p i q można przejść słowem a^ℓ, gdzie ℓ jest wykładnicze. W przypadku automatów deterministycznych problem ten staje się prosty, gdyż istnieje dokładnie jedna, być może pętląca się, ścieżka, po której można przejść słowami z a^*. Co więcej, wszystkie modyfikacje automatu zachowują determinizm automatu, czyli cały algorytm jest deterministyczny.

Wynik 10 ([H2])**.** *Oparty na rekompresji algorytm rozstrzyga w* NP*, czy skompresowany automat niedeterministyczny akceptuje skompresowane słowo. Jeśli automat jest deterministyczny, ten sam algorytm jest w* P*.*

2.2.4. *Konstrukcja najmniejszego SLP dla słowa.* Jeśli chcemy rozważać SLP jako model kompresji, musimy umieć szybko i wydajnie kompresować słowa do tej postaci, tj. rozważamy problem konstrukcji najmniejszego SLP generującego dane słowo. Niestety, wiadomo, że problem decyzyjny określenia rozmiaru takiego SLP jest NP-trudny [115], co więcej, niemożliwa jest aproksymacja lepsza niż stała 8569/8568 [12]. Znanych jest kilka [12, 105, 107] algorytmów aproksymacyjnych o współczynniku aproksymacji $\mathcal{O}(\log(n/g))$, gdzie g jest rozmiarem najmniejszego SLP generującego dane słowo (istnieją też liczne wydaje algorytmy heurystyczne, nieposiadające dobrej gwarancji aproksymacji [63, 58, 84]). Znane algorytmy aproksymacyjne są dość skomplikowane technicznie oraz oparte na związkach SLP ze standardem kompresji LZ77, w szczególności ich analiza tak naprawdę pokazuje, iż generują one SLP o rozmiarze $\mathcal{O}(\ell \log(n/\ell))$, gdzie ℓ jest rozmiarem kodowania danego słowa przy użyciu standardu LZ77. Jako że $\ell \leq g$ oraz funkcja $x \log(n/x)$ jest monotoniczna, uzyskujemy również oszacowanie $\mathcal{O}(g \log(n/g))$.

Rekompresję możemy w naturalny sposób stosować do danego jawnie słowa (przypomnijmy, że algorytm Kompresja był tak właśnie zdefiniowany), jako produkt uboczny otrzymujemy SLP generujące to słowo, na pierwszy rzut oka nie jest jasne jednak, jakiej wielkości jest to SLP. Okazuje się, że i tym razem pomocna jest rekompresja: w ramach eksperymentu myślowego możemy (równolegle) przeprowadzić rekompresję również na najmniejszym SLP generującym to słowo (jest ono rozmiaru g). W ten sposób możemy powiązać rozmiar tworzonego SLP z rozmiarem najmniejszego SLP, w naturalny sposób możemy pokazać, że rozmiar stworzonego SLP to $\mathcal{O}(g \log n)$, oszacowanie to można pewnym nakładem pracy poprawić do $\mathcal{O}(g \log(n/g))$. W ten sposób dostajemy kolejny algorytm o tym samym współczynniku aproksymacji, ma on jednak kilka poważnych zalet:

- sam algorytm jest prosty;
- analiza nie opiera się na LZ77, a jedynie na modyfikowaniu SLP;
- algorytm jest „standardowy”, w tym sensie, że poza rekompresją nie wymaga żadnych dodatkowych pomysłów.

Brak związków z LZ77 daje pewną nadzieję na poprawę gwarancji aproksymacji: zauważmy, że możliwe jest, że istnieje algorytm o współczynniku aproksymacji $o(\log n)$, który wciąż jednak generuje SLP rozmiaru $\mathcal{O}(\ell \log(n/\ell))$. Ponadto, algorytm oparty na rekompresji można uogólnić do drzew (co zostaje pokazane w rozdziale 2.3.5, Wynik 14), poprzednie algorytmy nie uogólniały się do tego przypadku, gdyż standard LZ77 nie ma znanego wariantu zachowującego w jakiś sposób strukturę drzewa.

Wynik 11 ([H5])**.** *Oparty na rekompresji algorytm generowania SLP dla słowa ma współczynnik aproksymacji* $\mathcal{O}(\log(n/g))$*, gdzie g jest rozmiarem najmniejszego SLP dla zadanego na wejściu słowa.*

Uproszczone podejście. Konstruowane wcześniej algorytmy aproksymujące dla problemu najmniejszego SLP [105, 12, 107] korzystały ze standardu kompresji LZ77, w standardzie tym reprezentujemy słowo w jako $f_1 \cdot f_2 \cdots f_\ell$, gdzie każde f_i jest albo pojedynczą literą (*wolna litera*), albo fragmentem, który wystąpił wcześniej (*fragment*). Formalnie $|f_i| = 1$ lub $f_i = w[j \,..\, j + |f_i| - 1]$ dla pewnego $j \leq |f_1 \cdots f_{i-1}|$. Intuicyjnie, możemy myśleć, że te

przypadki odpowiadają literom oraz nieterminalom w produkcjach SLP. *Rozmiarem* reprezentacji $f_1 \cdot f_2 \cdots f_\ell$ jest ilość fragmentów i wolnych liter, w tym wypadku: ℓ. Wiadomo, iż najmniejsza taka reprezentacja może być skonstruowana w liniowym czasie, znanych jest wiele wydajnych algorytmów obliczających taką reprezentację [1, 13, 20, 43, 52, 88]. Co więcej, wiadomo, że najmniejsze SLP dla słowa jest nie mniejsze niż najmniejsza reprezentacja LZ77 [105] (co jest dość intuicyjne: SLP jest w istocie reprezentacją tego typu o specyficznej formie).

LZ77 jest powszechnie znane i dobrze zrozumiane, jest też wiele wydajnych implementacji znajdujących najmniejszą taką faktoryzację. Tym samym użycie LZ77 do problemu najmniejszego SLP ma niepodważalną zaletę: konstrukcja, analiza i implementacja algorytmu aproksymacyjnego może użyć wielu „czarnych skrzynek". Okazuje się, że podejście oparte na rekompresji wyposażone dodatkowo w reprezentacje LZ77 daje bardzo prosty (w opisie i analizie) algorytm.

Spróbujmy najprostszego podejścia do konstrukcji SLP: parujemy pierwszą literę z drugą, trzecią z czwartą, itp. Po parowanie zastępujemy pary nowymi literami, oczywiście wszystkie wystąpienia ustalonej pary zastępujemy tą samą literą. Następnie czynności te powtarzamy, dopóki nie uzyskamy jednoliterowego słowa. Przyjrzyjmy się, jak nasze parowanie „pasuje" do faktoryzacji LZ77. W idealnym przypadku chcielibyśmy, aby fragment f_i został sparowany tak samo, jak jego definicja $w[j \ldots j + |f_i| - 1]$. Niestety, ze względu na parzystość może się stać inaczej. Tym samym chcielibyśmy „poprawić" parowanie f_i, tak aby było sparowane tak samo, jak $w[j \ldots j + |f_i| - 1]$. Jako że pierwsza litera $w[j \ldots j + |f_i| - 1]$ może być sparowana z literą na lewo (a ostatnia z literą na prawo), być może usuwamy pierwszą i/lub ostatnią literę z fragmentu f_i, tj. zastępujemy f_i przez $af_i'b$ (lub af_i' lub $f_i'b$), gdzie f_i' jest nowym fragmentem, a a oraz b są nowymi wolnymi literami. Operacja ta jest podobna do operacji Wypchnij w przypadku SLP. Może się zdarzyć, że w ten sposób obok siebie sąsiadować będą dwie wolne, niesparowane litery — w takim przypadku parujemy je. Odpowiednia implementacja tworzy w czasie jednego przejścia słowa od lewej do prawej parowanie o następujących własnościach:

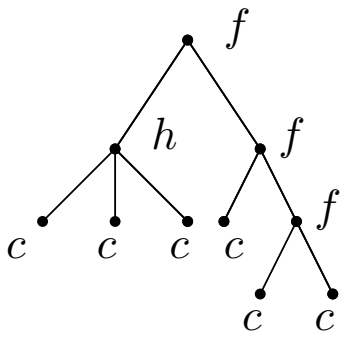
- nie ma dwóch kolejnych niesparowanych liter;
- jeśli $f_i = w[j \,..\, j + |f| - 1]$ ma definicję $w[j' \,..\, j' + |f| - 1]$ to litery w $w[j \,..\, j + |f| - 1]$ oraz $w[j' \,..\, j' + |f| - 1]$ są tak samo sparowane.

Prosta analiza wykazuje, że wielkość SLP to dokładnie ilość wolnych liter utworzonych przez algorytm. Jako że algorytm skraca słowo o stały czynnik w każdej fazie, ma $\mathcal{O}(\log n)$ faz i tym samym generuje gramatykę rozmiaru $\mathcal{O}(\ell \log n)$, dokładniejsza analiza pokazuje ograniczenie $\mathcal{O}(\ell \log(n/\ell))$. Jako że $\ell \leq g$ (gdzie g jest rozmiarem najmniejszego SLP dla słowa wejściowego) a funkcja $x \log(n/x)$ jest monotoniczna od x, uzyskujemy żądane oszacowanie na rozmiar skonstruowanego SLP.

Wynik 12 ([H9])**.** *Oparty na rekompresji i standardzie LZ77 algorytm generowania gramatyki dla słowa ma współczynnik aproksymacji $\mathcal{O}(\log(n/g)$, gdzie g jest rozmiarem najmniejszej gramatyki dla zadanego na wejściu słowa.*

2.3. **Rekompresja dla termów.** Liczne wyniki uzyskane dzięki rekompresji połączone z prostotą tej metody pozwoliły żywić nadzieję, że tego typu podejście da się rozszerzyć do termów. Jest to obiecujący kierunek badań, gdyż wiele odpowiedników problemów rozwiązanych i dobrze zrozumianych dla słów w przypadku termów pozostaje wciąż słabo zbadana.

Wygodnie jest nam myśleć o termach jako o ukorzenionych, uporządkowanych i etykietowanych drzewach, odpowiednia ilustracja znajduje się na Rysunku 1:



RYSUNEK 1. Term $f(h(c,c,c), f(c, f(c,c)))$ jako drzewo, f ma arność 2, h arność 3, natomiast c: 0.

***ukorzenionych*:** Drzewo posiada wyróżniony korzeń.
***uporządkowanych*:** Dzieci każdego wierzchołka występują w określonej kolejności.
***etykietowanych*:** Wierzchołki drzewa są etykietowane literami z alfabetu Σ (zwanego zwykle *sygnaturą*). Co więcej, każda litera $f \in \Sigma$ ma arność $\mathrm{ar}(f) \geq 0$ i wierzchołek etykietowany literą f ma dokładnie $\mathrm{ar}(f)$ dzieci.

2.3.1. *Kompresja termów.* Najbardziej naturalne podejście uogólniające rozwiązania dla słów do termów opiera się na zapisie termu jako słowa i narzucenie więzów, które wymuszają, iż faktycznie takie słowo odpowiada poprawnie zbudowanemu termowi. Niestety, takie więzy nie są regularne i próby uwzględnienie ich w rekompresji zakończyły się niepowodzeniem [96] (choć być może jest to możliwe).

Zamiast próby modyfikacji gotowej metody, wróćmy do jej bazowej idei: iterujemy lokalne operacje kompresji, które gwarantują nam zmniejszanie słowa (termu) o stały czynnik, operacje te można zasymulować bezpośrednio na równaniu. Jako że kilka problemów dla termów rozwiązano używając metod opartych na kompresji [32, 66, 19, 30, 31], możemy mieć nadzieję, że podejście takie zakończy się sukcesem.

Kompresja par i bloków z łatwością stosuje się do ciągów liter o arności 1 (takie ciągi w istocie możemy traktować jak słowa), niestety, nie ma żadnej gwarancji, że term posiada choć jedną taką literę. Intuicyjnie, oczekujemy raczej, iż term posiada głównie liście. Prowadzi to do kolejnej operacji lokalnej kompresji: *kompresji liści*. Rozważmy wierzchołek z etykietą f oraz jego dziecko (będące liściem) na pozycji i. Chcemy skompresować f wraz z tym dzieckiem, pozostawiając wszystkie inne dzieci nietknięte. Formalnie, dla danego f o arności przynajmniej 1, pozycji $1 \leq i \leq \mathrm{ar}(f)$ oraz litery c o arności 0 operacja $\mathsf{KompLiści}(f, i, c, t)$ (*kompresja liści*) zastępuje w termie t wierzchołki o etykiecie f i podtermach $t_1, \ldots, t_{i-1}, c, t_{i+1}, \ldots, t_{\mathrm{ar}(f)}$ (gdzie c oraz pozycja i są ustalone, natomiast pozostałe termy $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_{\mathrm{ar}(f)}$ — dowolne) przez term o etykiecie f' oraz podtermach $t'_1, \ldots, t'_{i-1}, t'_{i+1}, \ldots, t'_{\mathrm{ar}(f)}$, uzyskanych rekurencyjnie przez kompresję liści na termach $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_{\mathrm{ar}(f)}$; innymi słowy, najpierw zastępujemy etykietę f przez f' a następnie usuwamy i-te dziecko (ma ono etykietę c) i rekurencyjnie wykonujemy to samo w poddrzewach. Innymi słowy, taką kompresję aplikujemy do wszystkich takich wystąpień f i c jednocześnie.

Zauważmy, że wszystkie kompresje liści można przeprowadzić równolegle: dla każdego wierzchołka możemy jednoznacznie określić, czy i z jakim innym wierzchołkiem powinien być

Algorytm 9 KompLiści(f, i, c, t): kompresja liści

```
dla v: wierzchołek w t wykonaj
    jeśli v ma etykietę f oraz jego i-te dziecko ma etykietę c to
        zastąp etykietę v przez f'
        usuń i-te dziecko v
```

skompresowany (każdy liść ma dokładnie jednego ojca, a wierzchołek będący ojcem nie jest liściem).

Połączenie tej operacji oraz zdefiniowanych wcześniej kompresji bloków oraz par pozwala na podanie algorytmu kompresującego termy, jest on odpowiednikiem algorytmu Kompresja dla drzew. Co więcej, algorytm ten zwraca gramatykę generującą to drzewo, odpowiednia definicja gramatyki drzewowej i uzasadnienie tego faktu znajdują się w dalszej części autoreferatu (rozdział 2.3.5).

Algorytm 10 KompresjaDrzew(t) Kompresuje zadany term t

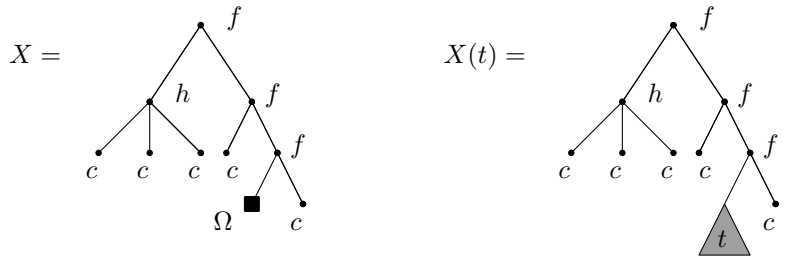
```
dopóki |t| > 1 wykonaj
    L ← lista liter unarnych w t                    ▷ Kompresja jak dla słów
    P ← lista par różnych liter unarych w t
    dla a ∈ L wykonaj
        KompBlok(a, t)
    dla ab ∈ P wykonaj
        KompPar(ab, t)
    L_0 ← lista liter o arności 0 w t               ▷ Reguła tylko dla termów
    L_{≥1} ← lista liter o arności przynajmniej 1 w t
    przeprowadź równolegle KompLiści(f, i, c, t) dla wszystkich c ∈ L_0, f ∈ L_{≥1}, i ≤ ar(f)
```

Tak jak w przypadku słów, *faza* to jedna iteracja głównej pętli algorytmu KompresjaDrzew; w ciągu jednej fazy drzewo zmniejsza się o stały czynnik.

Lemat 12. *Niech t będzie termem a t' termem uzyskanym z niego po jednej fazie* KompresjaDrzew. *Wtedy* $|t'| \leq \frac{2|t|+3}{3}$.

Dowód jest rozszerzeniem dowodu Lematu 1: jeśli term t nie zawiera żadnego symbolu o arności większej niż 1, to jest w istocie słowem i zachowanie KompresjaDrzew jest analogiczne do zachowania Kompresja (z dokładnością do jedynego liścia w t) i z Lematu 1 otrzymamy $|t'| \leq \frac{2|t|+3}{3}$ (różnica w stosunku do $\frac{2|t|+2}{3}$ bierze się z tego, że w drzewie jest dokładnie jeden liść, który być może nie zostanie skompresowany). Jeśli t nie zawiera żadnego symbolu o arności 1, to ponad połowa jego wierzchołków to liście i są one usunięte w czasie KompresjaDrzew, tym samym $|t'| < \frac{|t|}{2}$. Przypadek ogólny jest mieszanką tych dwóch analiza i daje żądany współczynnik.

2.3.2. *Unifikacja kontekstów.* Przypomnijmy, że problem unifikacji kontekstów jest rozszerzeniem równań słów do przypadku termów. Zastanówmy, jakiego rodzaju równania na termach chcielibyśmy rozważać. W oczywisty sposób zakładamy, że rozpatrujemy równania przy ustalonej sygnaturze (która jest zwykle częścią wejścia), w równaniach dopuszczamy występowanie liter oraz zmiennych. Jeśli chcemy, aby zmienne reprezentowały tylko pełne termy,



RYSUNEK 2. Kontekst i ten sam kontekst po zaaplikowaniu do argumentu.

to problem spełnialności takich równań można rozwiązać wielomianowo [104], i tym samym prawdopodobnie nie uogólnia on równań w słowach (które są NP-trudne). Fakt ten łatwo też zaobserwować, gdy przyjrzymy się dokładniej równaniom w słowach: słowa reprezentowane przez zmienne mogą być konkatenowane z obu stron, tj. reprezentują one termy z brakującym elementem.

Tym samym nasze uogólnienie powinno używać zmiennych *z argumentami*, tj. zmienne (drugiego rzędu) przyjmują parametr będący zamkniętym termem i mogą tego parametru użyć, być może wielokrotnie. Taka definicja prowadzi do *unifikacji drugiego rzędu*, o której wiadomo, że jest nierozstrzygalna nawet w bardzo ograniczonych przypadkach [42, 28, 65, 67].

Szukając podklasy pomiędzy równaniami w słowach a unifikacją drugiego rzędu nakładamy kolejne ograniczenie: argument termu ma być użyty *dokładnie raz*. Zauważmy, że problem ten wciąż uogólnia równania w słowach: jednokrotne użycie argumentu naturalnie odpowiada konkatenacji.

Formalnie, w problemie unifikacji kontekstów [17, 18, 108], rozważamy równanie $u = v$ w którym używamy zmiennych (reprezentujących pełne termy), które będziemy oznaczać małymi literami x, y, oraz zmiennych kontekstowych (reprezentujących termy z „dziurą" na argument, zwyczajowo nazywamy je *kontekstami*), które będziemy oznaczać wielkimi literami X, Y. Syntaktycznie, u i v są termami, które używają liter z sygnatury Σ (będącej częścią wejścia), zmiennych i zmiennych kontekstowych, przy czym zmienne mają arność 0, natomiast zmienne kontekstowe arność 1. *Podstawienie* S przypisuje każdej zmienne zamknięty term nad Σ, natomiast każdej zmiennej kontekstowej *kontekst*, tj. term nad $\Sigma \cup \{\Omega\}$, przy czym specjalny symbol Ω ma arność 0 i musi być użyty dokładnie raz (intuicyjnie odpowiada on miejscu, w które podstawiany jest argument). S rozszerzamy do u, v w naturalny sposób, przy czym dla zmiennej kontekstowej X term $S(X(t))$ jest otrzymany jako zastąpienie w $S(X)$ symbolu Ω przez $S(t)$. Rozwiązanie to podstawienie, dla którego $S(u) = S(v)$.

Przykład 2. Rozważmy sygnaturę $\{f, c, c'\}$, gdzie f ma arność 2 a c, c' arność 0 i równanie $X(c) = Y(c')$, gdzie X oraz Y są zmiennymi kontekstowymi. Równanie to ma rozwiązanie $S(X) = f(\Omega, c'), S(Y) = f(c, \Omega)$, wtedy $S(X(c)) = f(c, c') = S(Y(c'))$.

Dla uproszczenia zapisu, będziemy zapisywać litery unarne oraz zmienne kontekstowe tak jak w przypadku słów, tzn. piszemy $aXbc$ zamiast $a(X(b(c)))$, gdzie a, b są literami unarnymi, c stałą a X zmienną kontekstową.

Zauważmy, że zgodnie z terminologią „równań słów", problem „unifikacja kontekstów" powinien nazywać się „równania kontekstów", ale dużo popularniejsza jest nazwa „unifikacja kontekstów" (jednocześnie istnieje nazwa „unifikacja słów", jest ona jednak rzadziej stosowana).

2.3.3. *Własności.* Można pokazać, że odpowiednie uogólnienie Lematu 5 (który mówi o wykładniczym ograniczeniu na wykładnik okresowości) zachodzi również dla najmniejszych rozwiązań unifikacji kontekstów [111]. Niestety, Przykład 2 pokazuje, że Lemat 3 *nie zachodzi* dla unifikacji kontekstów: łatwo sprawdzić, że podane w nim rozwiązanie jest najmniejsze (w sensie, iż nie ma rozwiązania mającego mniej wierzchołków), jednocześnie litera f nie występuje w równaniu. W ogólności problem wynika z faktu, że rozwiązanie może używać liter, które są w sygnaturze, ale nie występują w równaniu. Na szczęście, jak pokazuje lemat, tej sytuacji można zaradzić: take litery są jedynie dwie.

Lemat 13 (porównaj Lemat 3)**.** *Niech $u = v$ będzie równaniem kontekstów nad sygnaturą Σ. Takie równanie jest spełnialna wtedy i tylko wtedy, gdy jest spełnialna jako równanie nad sygnaturą Σ', zawierającą wszystkie litery występujące w u, v, dowolną stała (może to być stała z u, v) i dowolny symbol binarny (może to być litera z u, v).*

Lemat 13 jest używany niejawnie: możemy zagwarantować, że sygnatura zawsze składa się z liter obecnych w równaniu i najwyżej dwóch innych liter. Tym samym nie musimy jej całej pamiętać i w razie potrzeby odtwarzamy ją z równania.

2.3.4. *Algorytm dla unifikacji kontekstów.* Strategia rozwiązania problemu unifikacji kontekstów jest taka sama, jak analogiczna strategia dla równań w słowach: chcemy zastosować kompresję KompresjaDrzew bezpośrednio do $S(u)$ i $S(v)$ i dokonać drobnych modyfikacji na równaniu kontekstów, tak aby było to możliwe.

Definicje wystąpień niejawnych, jawnych, krzyżujących dla par i bloków są analogiczne, jak w przypadku równań w słowach. Musimy rozszerzyć ją jeszcze na wystąpienia liści oraz ich ojców. Gdy liść c ma ojca etykietowanego literą f, mówimy, że są one wtedy parą dziecko-ojciec. W naturalny sposób mówimy o wystąpieniach jawnych, niejawnych i krzyżujących dla pary dziecko-ojciec. Jeśli para ma choć jedno wystąpienie krzyżujące (dla rozwiązania S), to jest krzyżująca.

W przypadku równań w słowach wystąpienia jawne były w pewnym sensie nieistotne (Lemat 3): w rozwiązaniu najmniejszym każde wystąpienie niejawne ma odpowiadające wystąpienie jawne lub krzyżujące. Podobna własność zachodzi również w przypadku równań kontekstów.

Lemat 14 (porównaj Lemat 3)**.** *Niech S będzie najmniejszym rozwiązaniem równania kontekstów '$u = v$'. Wtedy:*

- *Jeśli ab jest parą liter unarnych i podsłowem $S(u)$, gdzie $a \neq b$, to a, b mają wystąpienia jawne w równaniu zaś ab ma też wystąpienie jawne lub krzyżujące*
- *Jeśli a^k jest maksymalnym blokiem w $S(u)$ to a ma wystąpienie jawne w równaniu oraz a^k ma wystąpienie jawne lub krzyżujące w '$u = v$'.*
- *Jeśli (f, c) jest wystąpieniem pary ojciec-dziecko, to istnieje też wystąpienie jawne lub krzyżujące pary (f, a).*

Zauważmy, że w przypadku pary ojciec-dziecko (f, c) nie możemy zagwarantować, że choć jedna z tych liter wystepuje w równaniu.

Jeśli para ab (litera a, para ojciec-dziecko (f, c)) nie ma wystąpień krzyżujących, to algorytmy KompPar, KompBlok, KompLiści stosują się bezpośrednio do równania. Problemem są wystąpienia krzyżujące.

W przypadku wystąpień krzyżujących dla par i bloków wiemy jak postępować, jedyna zmiana to kwestia „orientacji" zmiennych: dla zmiennych słowowych litera „pierwsza" i „ostatnia" były odpowiednio literami najbardziej z lewej i z prawej, w przypadku termów są to odpowiednio korzeń i litera nad symbolem Ω (zauważmy, że zmienne reprezentujące pełne termy nie mają „ostatniej" litery). Tym samym algorytmy Wypchnij oraz KompBlok w naturalny sposób przenoszą się do przypadku termów.

Pozostaje pytanie, jak sprawić, by para ojciec-dziecko (f, c) stała się nie krzyżująca. W tym celu przedstawiamy charakteryzacją analogiczną do (CP1)–(CP3): łatwo pokazać, że (f, c) jest krzyżującą parą dziecko-ojciec (dla rozwiązania S) wtedy i tylko wtedy gdy zachodzi jeden z następujących warunków:

(CFC1) $f(\ldots, y, \ldots)$ występuję w równaniu oraz $S(y) = c$;
(CFC2) $X(c)$ występuję w równaniu oraz ostatnią literą (tj. nad specjalnym symbolem Ω) $S(X)$ jest f;
(CFC3) $X(y)$ występuję w równaniu, oraz ostatnią literą $S(X)$ jest f i $S(y) = c$.

W przypadku (CFC1) wystarczy zamienić y przez stałą c, w (CFC2) chcielibyśmy wypchnąć literę f w dół. Niestety, może mieć ona też inne dzieci niż c i nie wiemy, jakie termy są zakorzenione w tych dzieciach. Rozwiązaniem jest wprowadzenie nowej zmiennej na każde dziecko f: wprowadzamy zmienne $x_1, x_2 \ldots, x_{i-1}, x_{i+1}, \ldots, x_k$, gdzie $k = \mathrm{ar}(f)$ i zastępujemy X przez $X(f(x_1, x_2 \ldots, x_{i-1}, \cdot, x_{i+1}, \ldots, x_k))$. Zauważmy, że w ten sposób *wprowadzamy nowe zmienne*, co nie miało miejsc w przypadku równań na słowach. W trzecim przypadku (CFC3) wykonujemy obie opisane operacje.

Algorytm Wypchnij' wykonuje powyższe operacje. Warto zwrócić uwagę na dwa szczegóły: po pierwsze, możemy wykonać Wypchnij' dla wszystkich par jednocześnie; po drugie, w przeciwieństwie do Wypchnij oraz WypPrefSuf istotne jest, że wypychanie liter w dół wykonywane jest tylko wtedy, gdy jest naprawdę potrzebne, tzn. gdy któryś z warunków (CFC1)–(CFC3) jest spełniony.

Algorytm 11 Wypchnij'$(f, c, `u = v')$

```
1: dla x: zmienna wykonaj
2:     jeśli S(x) = c to                                          ▷ Zgadnij
3:         zastąp każde x w u = v przez c
4: dla X: zmienna kontekstowa wykonaj
5:     jeśli f: ostatnia litera S(X) i X(c) występuje w równaniu to
6:         niech Ω będzie i-tym dzieckiem f w S(X)                ▷ Zgadnij
7:         zastąp X w u = v przez Xf(x_1, x_2, ..., x_{i-1}, ·, x_{i+1}, ..., x_k)
8:         jeśli S(X) jest puste to                               ▷ Zgadnij
9:             usuń X z równania
```

Używając tych procedur jesteśmy już w stanie rozszerzyć algorytm KompresjaDrzew do równań kontekstów: najpierw algorytm wypisuje litery unarne występujące w równaniu. Dokonuje kompresji bloków: najpierw nie krzyżujących i potem krzyżujących. Następnie kompresuje pary (również najpierw nie krzyżujące). W ostatnim kroku wypisuje litery o arności 0 oraz pozostałe, równolegle odkrzyżowuje pary ojciec-syn i na koniec kompresuje równolegle liście.

Poprawność algorytmu wynika z wcześniejszej dyskusji na temat algorytmów KompPar, Wypchnij, KompBlok, WypPrefSuf, Wypchnij' i KompLiści. Pozostaje oszacować zużycie pamięci.

Algorytm 12 SpełUnifKont$(u = v)$ Rozstrzyganie spełnialności unifikacji kontekstów

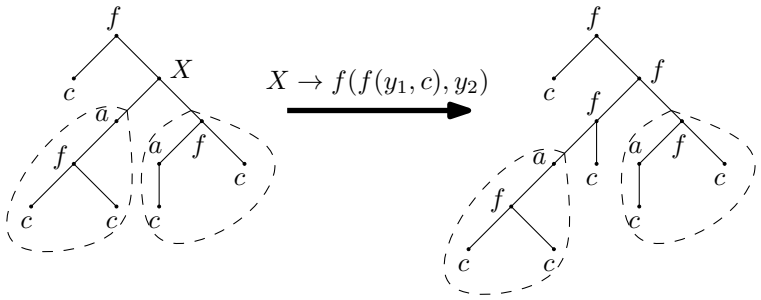
dopóki $|u| > 1$ lub $|v| > 1$ **wykonaj** ▷ Równanie nie jest trywialne
  $L \leftarrow$ lista liter unarnych w u, v
  $P \leftarrow$ lista par różnych liter unarych w u, v
  $L_1 \leftarrow$ lista unarnych liter nie krzyżujących ▷ Zgadnij
  dla $a \in L_1$ **wykonaj**
    KompBlok$(a, `u = v')$ ▷ Kompresujemy litery nie krzyżujace
  dla $a \in L \setminus L_1$ **wykonaj**
    WypPrefSuf$(a, `u = v')$ ▷ a przestaje być literą krzyżującą
    KompBlok$(a, `u = v')$ ▷ Kompresujemy
  $P_1 \leftarrow$ lista par nie krzyżujących z P ▷ Zgadnij
  dla $ab \in P_1$ **wykonaj**
    KompPar$(ab, `u = v')$ ▷ Kompresujemy pary nie krzyżujące
  dla $ab \in P \setminus P_1$ **wykonaj**
    Wypchnij$(a, b, `u = v')$ ▷ ab przestaje być parą krzyżującą
    KompPar(ab, w) ▷ Kompresujemy pary nie krzyżujące
  $L_0 \leftarrow$ lista liter o arności 0 w t
  jeśli $L_0 = \emptyset$ **to**
    dodaje jedną świeżą literę o arności 0 ▷ Lemat 13
  $L_{\geq 1} \leftarrow$ lista liter o arności przynajmniej 1 w t
  jeśli $L_{\geq 1}$ nie ma litery o arności 2 **to**
    dodaje jedną świeżą literę o arności 2 ▷ Lemat 13
  dla $f \in L_{\geq 1}, c \in L_0$ **wykonaj**
    Wypchnij'$(f, c, `u = v')$
  przeprowadź równolegle KompLiści(f, i, c, t) dla wszystkich $c \in L_0$, $f \in L_{\geq 1}$, $i \leq \mathrm{ar}(f)$
Rozwiąż problem naiwnie ▷ Gdy strony równania mają długość 1, problem jest prosty

Pojawił się nowy problem: algorytm SpełUnifKont wprowadza nowe zmienne, co przeczy podstawowej idei analizy SpełRówSłów. Na szczęście, nowych zmiennych nigdy nie ma zbyt dużo.

Lemat 15. *W każdym momencie algorytmu* SpełUnifKont *równanie zawiera najwyżej* n *zmiennych kontekstowych oraz* kn *zmiennych.*

Zauważmy, że to oszacowanie nie zależy od wyborów niedeterministycznych. Ograniczenie na ilość zmiennych kontekstowych jest proste: nigdy nie wprowadzamy nowych. Kluczowe jest oszacowanie ilości wprowadzonych zmiennych. Nowe zmienne wprowadzamy tylko w jednym miejscu: w procedurze Wypchnij' gdy zastępujemy X przez $Xf(x_1, x_2, \ldots, x_{i-1}, \cdot, x_{i+1}, \ldots, x_k)$, co następuje tylko wtedy, gdy $X(c)$ występuje w równaniu. Stowarzyszamy te zmienne ze zmienną kontekstową X. Gdy ponownie X wprowadza zmienne, znów $X(c')$ występuje w równaniu. Ale wcześniej to samo wystąpienie X miało zmienne $x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k$ w swoim poddrzewie, co oznacza, że wszystkie te zmienne musiały zostać usunięte. Czyli każdą zmienna kontekstowa ma w dowolnym momencie najwyżej $k-1$ stowarzyszonych „nowych" zmiennych, co daje obiecane oszacowanie.

W momencie ustalenia oszacowania górnego na ilość zmiennych reszta rozumowania dotyczącego rozmiaru przechowywanego równania jest taka sama jak w przypadku równań na



RYSUNEK 3. Drzewo $f(c, X(a(f(c,c)), f(a(c), c)))$ przed i po zaaplikowaniu reguły $X(y_1, y_2) \to f(f(y_1, c), y_2)$.

słowach. Ilość par krzyżujących wynosi $\mathcal{O}(kn)$ i każde usunięcie krzyżowania wprowadza $\mathcal{O}(k)$ liter. Natomiast kompresja zapewnia spadek wielkości o stały czynnik w jednej fazie. W sumie daje to wielomianowe ograniczenie na wielkość równania:

Lemat 16. *W jednej fazie do równania kontekstów wprowadza się najwyżej $\mathcal{O}(k^2n^2)$ liter. Równanie przechowywane przez algorytm* SpełUnifKont *ma rozmiar $\mathcal{O}(k^2n^2)$.*

Wynik 13 ([H10])**.** *Oparty na rekompresji algorytm rozstrzyga w* PSPACE *problemu unifikacji kontekstów.*

2.3.5. *Najmniejsza gramatyka dla termu.* Algorytm KompresjaDrzew konstruuje dla danego na wejściu termu gramatykę generującą ten term (odpowiednia definicja gramatyki drzewowej znajduje się poniżej). Na pierwszy rzut oka nie widać, jak ograniczyć rozmiar tej gramatyki, jednak można użyć podejścia analogicznego do przypadku słów, omówionego w rozdziale 2.2.4, co jest opisane poniżej.

Zacznijmy od zdefiniowania gramatyki drzewowej, jest to naturalne uogólnienie gramatyki bezkontekstowej dla słów. Symbole nieterminalne w takiej gramatyce mają *arność*, natomiast reguły są postaci $X \to t$, gdzie t jest termem zbudowanym z liter, nieterminali (nieterminal o arności k ma k dzieci) oraz *parametrów* $y_1, y_2, \ldots, y_{\mathrm{ar}(X)}$ (które są symbolami o arności 0 spoza sygnatury), przy czym każdy parametr jest użyty dokładnie raz. Wyprowadzenie wygląda następująco: nieterminal $X(t_1, \ldots, t_k)$ (gdzie k jest arnością X) jest zastąpiony przez $t[t_1, \ldots, t_k]$, co oznacza term t w którym parametry $y_1, \ldots, y_k$ są zastąpione przez $t_1, \ldots, t_k$; odpowiednia ilustracja znajduje się na Rysunku 3. Symbol startowy ma arność 0.

Aby podkreślić zależność nieterminala X od jego parametru, czasami piszemy $X(y_1, \ldots, y_k)$ zamiast X oraz $X(y_1, \ldots, y_k) \to t$, gdzie k jest arnością X.

Jesteśmy zainteresowani jedynie gramatykami generującymi dokładnie jedno słowo, tj. *drzewowymi SLP*. Używamy określenia $\mathrm{val}(X)$ na oznaczenie drzewa generowanego przez nieterminal X, zaznaczmy, że każdy z parametrów $y_1, \ldots, y_{\mathrm{ar}(X)}$ etykietuje dokładnie jeden z liści $\mathrm{val}(X)$.

Łatwo sprawdzić, że wszystkie kompresje wykonywane przez KompresjaDrzew odpowiadają produkcjom tak rozumianej gramatyki: kompresja pary ab w c odpowiada produkcji $c(y) \to a(b(y))$, kompresja bloków odpowiada produkcjom $a_\ell(y) \to \underbrace{a(a \cdots (a}_{\ell\text{razy}}(y)) \cdots)$ natomiast kompresja c jako i-tego dziecka f produkcji $f'(y_1, \ldots y_{i-1}, y_{i+1}, \ldots, y_{k-1}) \to f(y_1, \ldots y_{i-1}, c, y_{i+1}, \ldots, y_{k-1})$.

Chcielibyśmy potraktować drzewowe SLP jako układ równań kontekstów, niestety, drzewowe SLP pozwalają na użycie nieograniczonej ilości parametrów, zaś w unifikacji kontekstów

jest dokładnie jeden parametr. Z drugiej strony, jeśli drzewowe SLP ma nieterminale o arności najwyżej 1 (nazywamy je w takim przypadku *monadycznym*) to istotnie możemy patrzeć na nie jako na układ równań kontekstów: regułę $X \to t$ zastępujemy równaniem $X(y) = t$, gdzie y jest jedynym parametrem w t lub równaniem $X = t$, gdy t nie ma parametru (w pierwszym przypadku X jest zmienną kontekstową, w drugim zwykła zmienną). Tak jak w przypadku równań na słowach i SLP, układ takich równań ma jedyne rozwiązanie S, w którym $S(X) = \mathrm{val}(X)$.

Na szczęście wiadomo, że każde drzewowe SLP można przekształcić w monadyczne drzewowe SLP kosztem niewielkiego zwiększenia rozmiaru.

Lemat 17 ([77])**.** *Dla każdego drzewowego SLP można w wielomianowym czasie skonstruować monadyczne drzewowe SLP generujące to samo drzewo. Skonstruowanego drzewowe SLP jest rozmiaru $\mathcal{O}(rg)$, gdzie g jest rozmiarem oryginalnego drzewowego SLP, zaś r maksymalną arnością liter w sygnaturze.*

Tym samym, postępujemy jak w przypadku konstrukcji SLP dla słowa: uruchamiamy algorytm KompresjaDrzew na wejściowym słowie t. Analiza opiera się na eksperymencie myślowym: rozpatrujemy najmniejsze monadyczne drzewowe SPL generujące t, traktujemy je jako układ równań kontekstów. Równolegle z każdą kompresją KompresjaDrzew dokonujemy analogicznych kompresji w algorytmie SpełUnifKont na układzie równań kontekstów. Ilość tworzonych nowych liter i rozmiar ich produkcji można oszacować na podstawie rozmiaru instancji trzymanej przez algorytm SpełUnifKont. Dokładna analiza pozwala oszacować rozmiar otrzymanego drzewowego SLP na $\mathcal{O}(gr \log(n/g))$, gdzie g jest rozmiarem najmniejszego drzewowego SLP generującego dane drzewo.

Wynik 14 ([H7])**.** *Oparty na rekompresji algorytm generowania gramatyki dla drzewa ma współczynnik aproksymacji $\mathcal{O}(r \log(n/g))$, gdzie g jest rozmiarem najmniejszego drzewowego SLP dla zadanego na wejściu drzewa a r maksymalną arnością alfabetu.*[5]

Jest to pierwszy nietrywialny współczynnik aproksymacji dla tego problemu.

Część 2. Omówienie innych wyników naukowych

Poza pracami stanowiącymi dzieło naukowe będące przedmiotem habilitacji, moje badania doprowadziły do publikacji następujących prac naukowych, przy czym

- prace [D1]–[D7] stanowiły moją rozprawę doktorską,
- prace [E1]–[E7] opublikowane zostały w czasie studiów doktoranckich jednak nie były częścią rozprawy doktorskiej,
- prace [I1]–[I7] zostały napisane po ukończeniu studiów doktoranckich, nie są jednak częścią jednotematycznego cyklu prac stanowiącego osiągnięcie naukowe.

Prezentacja wyników dokonana zostanie za względu na tematykę badań; taka klasyfikacja nie pokrywa się całkowicie z poprzednim podziałem.

[D1] A. Jeż, „Conjunctive grammars generate non-regular languages over unary alphabet”, *Developments in Language Theory (DLT)*, (Turku, Finlandia, 2007), LNCS 4588, 242–253,
pełna wersja: *International Journal of Foundations of Computer Science*, 19:3 (2008), 597–615, wydanie specjalne po DLT 2007.

[5]Wersja konferencyjna pracy zawiera jedynie słabszy wynik: aproksymację $\mathcal{O}(r^2 \log(n/g)$, ale zgłoszona do czasopisma wersja zawiera poprawione oszacowanie.

[D2] A. Jeż, A. Okhotin, „Conjunctive grammars over a unary alphabet: undecidability and unbounded growth", *International Computer Science Symposium in Russia (CSR)*, (Jekaterynburg, Rosja, 2007), LNCS 4649, 168–181,
pełna wersja: *Theory of Computing Systems*, 46:1 (2010), 27–58, wydanie specjalne po CSR 2007.

Nagroda za *najlepszą pracę w potoku teoretycznym*.

[D3] A. Jeż, A. Okhotin, „Complexity of solutions of equations over sets of natural numbers", *Symposium on Theoretical Aspects of Computer Science (STACS)*, (Bordeaux, Francja, 2008), LIPIcs 1, 373–384,
pełna wersja: *Theory of Computing Systems*, 48:2 (2011), 319–342.

[D4] A. Jeż, A. Okhotin, „On the computational completeness of equations over sets of natural numbers", *International Colloquium on Automata, Languages and Programming (ICALP (B))*, (Reykjavik, Islandia, 2008), LNCS 5126, 63–74,
pełna wersja: „Computational completeness of equations over sets of natural numbers", *Information and Computation*, 237 (2014), 56–94.

[D5] A. Jeż, A. Okhotin, „Equations over sets of natural numbers with addition only", *Symposium on Theoretical Aspects of Computer Science (STACS)*, (Fryburg, Niemcy, 2009), LIPIcs 3, 577–588.

[D6] A. Jeż, A. Okhotin, „One-nonterminal conjunctive grammars over a unary alphabet", *International Computer Science Symposium in Russia (CSR)*, (Nowosybirsk, Rosja, 2009), LNCS 5675, 191–202,
pełna wersja: *Theory of Computing Systems*, 49:2 (2011), 319–342, wydanie specjalne po CSR 2009.

[D7] A. Jeż, A. Okhotin, „Univariate Equations Over Sets of Natural Numbers" *Fundamenta Informaticae*, 104 (2010), 329–348.

[E1] M. Grech, A. Jeż, A. Kisielewicz, „Graphical complexity of products of permutation groups", *Discrete Mathematics*, 308:7 (2008), 1142–1152.

[E2] M. Bieńkowski, M. Chrobak, C. Dürr, M. Hurand, A. Jeż, Ł. Jeż, G. Stachowiak, „Collecting Weighted Items from a Dynamic Queue", *Symposium on Discrete Algorithms (SODA)*, (Nowy Jork, USA, 2009), SIAM SODA, 1126–1135,
pełna wersja: *Algorithmica*, 65:1 (2013), 60–94.

[E3] A. Jeż, J. Łopuszański, „On the Two-Dimensional Cow Search Problem", *Information Processing Letters*, 109:11 (2009), 543–547.

[E4] P. Gawrychowski, A. Jeż, „Hyper-minimisation made efficient", *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, (Nowy Smokowiec, Słowacja, 2009), LNCS 5734, 356–368.

Nagroda za *najlepszą pracę studencką*.

[E5] A. Jeż, A. Okhotin, „On equations over sets of integers", *Symposium on Theoretical Aspects of Computer Science (STACS)*, (Nancy, Francja, 2010), LIPIcs 5, 477–488,
pełna wersja: „Representing Hyper-arithmetical Sets by Equations over Sets of Integers" *Theory of Computing Systems*, 51:2 (2012), 196–228, wydanie specjalne po STACS 2010.

[E6] P. Gawrychowski, A. Jeż, Ł. Jeż, „Validating the Knuth-Morris-Pratt failure function, fast and online", *International Computer Science Symposium in Russia (CSR)*, (Kazań, Rosja, 2010), LNCS 6072, 132–143,

pełna wersja: *Theory of Computing Systems*, 54:2 (2014), 337–372, wydanie specjalne po CSR, 2010.

[E7] A. Jeż, A. Okhotin, „Least and greatest solutions of equations over sets of integers", *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, (Brno, Czechy, 2010), LNCS 6281, 441–452.

[I1] P. Gawrychowski, A. Jeż, A. Maletti, „On Minimising Automata with Errors", *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, (Warszawa, 2011), LNCS 6907, 327–338.

[I2] A. Jeż, A. Maletti „Computing all ℓ-cover automata fast", *„International Conference on Implementation and Application of Automata (CIAA)*, (Blois, Francja, 2011), LNCS 6807, 203–214.

[I3] A. Jeż, T. Jurdziński, „Length-reducing Automata (almost) without Auxiliary Symbols", *Journal of Automata, Languages and Combinatorics*, 16:(2–4) (2011), 141–164.

[I4] A. Jeż, A. Maletti, „Hyper-Minimization for Deterministic Tree Automata", *„International Conference on Implementation and Application of Automata (CIAA)*, (Porto, Portugalia, 2012), LNCS 7381, 217–228,
pełna wersja: *International Journal of Foundations of Computer Science*, 24:6 (2013), 815–830, wydanie specjalne po CIAA 2012.

[I5] A. Jeż, A. Okhotin, „On the number of nonterminal symbols in unambiguous conjunctive grammars", *International Workshop on Descriptional Complexity of Formal Systems (DCFS)*, (Braga, Portugalia, 2012), LNCS 7386, 183–195.

[I6] M. Bieńkowski, M. Chrobak, C. Dürr, M. Hurand, A. Jeż, Ł. Jeż, G. Stachowiak, „A ϕ-competitive algorithm for collecting items with increasing weights from a dynamic queue", *Theoretical Computer Science*, 475 (2013), 92–102.

[I7] A. Jeż, A. Okhotin, „Unambiguous conjunctive grammars over a one-letter alphabet", *Developments in Language Theory (DLT)*, (Paryż, Francja, 2013), LNCS 7907, 277–288.

3. Równania nad zbiorami liczb, gramatyki koniunkcyjne

3.1. **Układy równań języków.** Dwoma głównymi formalizmami w teorii języków formalnych są gramatyki formalne oraz automaty. Trzecim, zyskującym rosnące zainteresowanie, są *układy równań języków formalnych*. Są one postaci

$$\begin{cases} \varphi_1(X_1,\ldots,X_n) &= \psi_1(X_1,\ldots,X_n) \\ & \vdots \\ \varphi_m(X_1,\ldots,X_n) &= \psi_m(X_1,\ldots,X_n) \end{cases},$$

gdzie każde φ_i, ψ_i składa się ze zmiennych (reprezentujących języki), stałych (będącymi językami z pewnej klasy) oraz operacji na językach, (zwykle: konkatenacji, sumy, jak również przecięcia, dopełnienie i być może innych). Ważną podklasą układów równań języków są układy w postaci *rozwiązanej*, w których każda zmienna X_i ma dokładnie jedno równanie definiujące ją, tzn. są postaci

$$\begin{cases} X_1 &= \varphi_1(X_1,\ldots,X_n) \\ & \vdots \\ X_n &= \varphi_n(X_1,\ldots,X_n) \end{cases}.$$

O ile operacje użyte w φ_i są monotoniczne, równania w postaci rozwiązanej zawsze mają najmniejsze i największe rozwiązanie, co wynika z twierdzenia Tarskiego o punkcie stałym [116].

Badania nad układami równań języków rozpoczęły się stosunkowo wcześnie, jednak przełomowym momentem było zauważenie związku między układami równań w postaci rozwiązanej a gramatykami, dokonane przez S. Ginsburga i G.Rice'a [38]; Dokładniej, pokazali oni, że gramatyki bezkontekstowe odpowiadają najmniejszym rozwiązaniom układów równań języków w postaci rozwiązanej, w której dopuszczamy użycie operacji konkatenacji oraz sumy zbiorów (oraz skończonych stałych).

Z punktu widzenia zastosowań i siły wyrazu, główną wadą gramatyk bezkontekstowych jest brak domkniętości tej klasy na przecięcie. Problemowi temu próbowano wielokrotnie zaradzić i zdefiniowano wiele rodzajów rozszerzeń gramatyk bezkontekstowych, z różnych względów żadne z tych rozszerzeń nie zostało uznane za kanoniczne. Niedawno A. Okhotin zaproponował inne podejście do problemu: w *gramatykach koniunkcyjnych* [89] pozwalamy na dowolne użycie przecięcia w ciele reguły, zaś semantykę zadajemy przy użyciu układów równań języków formalnych. Takie podejście okazało się pod wieloma względami skuteczne: wiele znanych algorytmów dla gramatyk bezokonteskotwych rozszerza się do gramatyk koniunkcyjnych [90, 95, 93].

3.2. **Języki formalne a zbiory liczb naturalnych.** Gdy zajmujemy się alfabetem jednoliterowym, na język L możemy w istocie patrzeć jak na zbiór liczb $\{n \mid a^n \in L\}$. W ten sposób formalizmy teorii języków formalnych mogą służyć do definiowania zbiorów liczb. W szczególności, układy równań języków zamieniają się w układy równań nad zbiorami liczb, zaś operacja konkatenacji języków odpowiada operacji sumy Minkowsky'ego:

$$A + B = \{a + b \mid a \in A, b \in B\} \ .$$

Badania nad tego typu reprezentacjami zbiorów liczb są obecne w informatyce od wpływowej pracy L. Stockmeyera i A. Meyera [114].

3.2.1. *Układy w postaci rozwiązanej.* Gdy użyjemy gramatyk bezkontekstowych do takiego opisu liczb, dość łatwo pokazać, że otrzymujemy zbiory będące skończoną sumą ciągów arytmetycznych, co odpowiada językom regularnym. Znane są konstrukcje systemów w postaci rozwiązanej, której używają też dopełnienia, i mają rozwiązania spoza tej klasy [64]. Pozostawało pytaniem otwartym [92], co dzieje się w przypadku gramatyk koniunkcyjnych. Dosyć niespodziewanie okazało się, iż również w tym przypadku istnieją rozwiązania niebędące skończoną sumą ciągów arytmetycznych. Metoda użyta w konstrukcji kontrprzykładu opiera się na manipulacji zapisem pozycyjnym liczb. W efekcie jesteśmy w stanie konstruować zbiory liczb których zapis k-arny ma określone własności. Dla uproszczenia zapisu, dla alfabetu cyfr $\Sigma_k = \{0, 1, \ldots, k-1\}$ oraz języka $L \subseteq \Sigma_k^*$ niech $(L)_k$ oznacza zbiór liczb, których k-pozycyjna notacja jest w języku L.

Wynik 15 ([D1])**.** *Układy równań w postaci rozwiązanej używające operacji* $\{+, \cap, \cup\}$ *są w stanie wyrazić każdy zbiór postaci* $(L)_k$*, gdzie* $L \subseteq \{0 \ldots, k-1\}^*$ *jest językiem regularnym.*

Wynik ten pozwolił na obalenie wcześniejszej hipotezy, że wszystkie takie zbiory są w istocie skończonymi sumami ciągów arytmetycznych [92]. Poza tym poszerzył „arsenał stałych", których możemy użyć jako czarnych skrzynek w innych konstrukcjach. Przykładowo, pozwolił na ustalenie, że złożoność obliczeniowa sprawdzania przynależności liczby do rozwiązania takich układów równań to DEXPTIME: przynależność do tej klasy jest prosta i była znana, jednak pokazanie trudności było nietrywialne.

Wynik 16 ([D3])**.** *Problem przynależności liczby do najmniejszego rozwiązania układy równań w postaci rozwiązanej używające operacji* $\{+, \cap, \cup\}$ *jest* DEXPTIME*-zupełny.*

Niestety, klasa wyrażalnych zbiorów pozostała wciąż stosunkowo mała. Użyte techniki dało się jednak uogólnić, tak iż zamiast klasy języków regularnych byliśmy w stanie opisać zbiory, których notacja pozycyjna jest rozpoznawana przez automat kratowy (trellis automata). Klasa ta nie jest szeroko znana a definicja automatów dość techniczna, dla nas istotne jest jedynie, że jest ona zamknięta na operacje boolowskie oraz zawiera liniowe gramatyki bezkontekstowe, co w szczególności pozwala stwierdzić, ze zawiera język zapisów przebiegów maszyny Turinga.

Wynik 17 ([D2])**.** *Układy równań w postaci rozwiązanej używające operacji* $\{+, \cap, \cup\}$ *są w stanie wyrazić każdy zbiór postaci* $(L)_k$*, gdzie* $L \subseteq \{0 \ldots, k-1\}^*$ *jest językiem rozpoznawanym przez automat kratowy. W szczególności problem niepustości rozwiązania takiego układu równań jest nierozstrzygalny.*

Kolejnym rozważanym kierunkiem badań jest sprawdzenie, co stanie się, gdy ograniczymy dozwoloną postać równań: czy będą miały taką samą siłe wyrazu, czy może staną się prostsze? Naturalnym obostrzeniem jest ograniczenie ilości zmiennych. Okazuje się, że możliwe jest *zakodowanie* dowolnego układu w postaci rozwiązanej w układzie z tylko jedną zmienną. Kodowanie jest naturalne: gdy chcemy zakodować k zmiennych w jednej, wartości zmiennej X_i kodowane są na i-tej *ścieżce modulo* k rozwiązanie dla X, tj. $X_i = \{n \mid kn + i \in X\}$.

Wynik 18 ([D6])**.** *Układy równań w postaci rozwiązanej używające operacji* $\{+, \cap, \cup\}$ *używające jednej zmiennej są w stanie kodować dowolny układ tej postaci używający dowolnej ilości zmiennych.*

Kolejna interesujące podklasa ma swoje źródło w językach formalnych. Ponieważ gramatyki są używane do definiowania składni języków programowania, jest pożądanym, aby słowo miało dokładnie jedno wyprowadzenie. Taką klasę gramatyk nazywamy *jednoznacznymi*. Odpowiednią definicję można łatwo przenieść też na układy równań języków formalnych, a z nich: na układy równań zbiorów liczb.

Wszystkie podane dotychczas konstrukcje nie są jednoznaczne. Jednak ich odpowiednia modyfikacja pozwala uzyskać kodowanie zbiorów liczb, których zapis pozycyjny jest rozpoznawany przez automaty kratowe.

Wynik 19 ([D7])**.** *Jednoznaczne układy równań w postaci rozwiązanej używające operacji* $\{+, \cap, \cup\}$ *są w stanie wyrazić kodowanie dowolnego zbioru postaci* $(L)_k$*, dla dowolnego* $L \subseteq \{0 \ldots, k-1\}^*$ *rozpoznawanego przez automat kratowy.*

Możliwe też jest połączenie wyników dotyczących układów jednoznacznych oraz wyników dotyczących układów z ograniczoną ilością zmiennych.

Wynik 20 ([D5])**.** *Jednoznaczne układy równań nad zbiorami liczb z jedna zmienną definiują jedynie zbiory będące skończonymi sumami ciągów arytmetycznych, z dwoma zmiennymi również zbiory spoza tej klasy lecz wciąż definiują istotnie mniej zbiorów, niż układy z trzema zmiennymi.*

3.2.2. *Układy w postaci ogólnej.* W przypadku układów równań języków w postaci ogólnej, kilka lat temu udało się ustalić ich pełną charakteryzację:

Lemat 18 ([94, 91])**.** *Każde rozwiązanie jedyne (najmniejsze, największe) układu równań języków używających ciągłych i monotonicznych operacji jest rekurencyjne (rekurencyjnie-przeliczalne, ko-rekurencyjnie przeliczalne).*

Każdy zbiór rekurencyjne (rekurencyjnie-przeliczalne, ko-rekurencyjnie przeliczalne) nad alfabetem dwuliterowym jest rozwiązaniem jedynym (najmniejszym, największym) układu równań języków używającego operacji $\{\cdot, \cup\}$ jak również układu używającego operacji $\{\cdot, \cap\}$.

Ograniczenie górne w oczywisty sposób również stosuje się do przypadku równań nad zbiorami liczb (bo odpowiadają one układom nad alfabetem unarnym), jednak konstrukcja użyta w ograniczeniu dolnym w ogólności się nie przenosi. Chcieliśmy ustalić dokładną moc wyrazu układów równań nad zbiorami liczb.

Każdy układ w postaci rozwiązanej jest też w szczególności układem równań w postaci ogólnej, tak więc zbiory postaci $(L)_k$ dla L rozpoznawanego przez automat kratowy są rozwiązaniami jedynymi takich układów. W szczególności, wyrażalne w ten sposób są zapisy historii obliczeń maszyn Turinga. Przy użyciu odpowiedniej konstrukcji jesteśmy w stanie z zapisów historii obliczeń wydobyć same słowa, które są akceptowane (w przypadku Lematu 18 robiliśmy to przy użyciu konkatenacji). Zauważmy, że już w pierwszym kroku (adaptacja wyników z przypadku równań w postaci rozwiązanej) użyliśmy operacji $\{+, \cap, \cup\}$. Jednak dzięki dostępności układów równań w postaci ogólnej jesteśmy w stanie symulować jedną operację boolowską przez druga i tym samym nasza konstrukcja używa tylko jednej operacji.

Wynik 21 ([D4])**.** *Klasa rozwiązań jedynych (najmniejszych, największych) układów równań zbiorów liczb naturalnych używających operacji $\{+, \cup\}$ lub $\{+, \cap\}$ to dokładnie klasa zbiorów rekurencyjnych (rekurencyjnie-przeliczalnych, ko-rekurencyjnie przeliczalnych).*

Pojawia się pytanie, czy uzyskany wynik da się wzmocnić, np. poprzez nałożenie dalszych ograniczeń na użyte operacje. Ponieważ używamy jedynie dwóch operacji, w pewnym sensie graniczny przypadek to jedna operacja, w naszym wypadku $\{+\}$ (układy używające jedynie operacji $\cap$ lub $\cup$ są trywialne). Układy używające tylko jednej operacji i skończonych stałych są trywialne, lecz są w stanie symulować ogólne układy, o ile dopuścimy użycie stałych będących nieskończonymi ciągami arytmetycznymi.

Wynik 22 ([D5])**.** *Klasa rozwiązań jedynych (najmniejszych, największych) układów równań zbiorów liczb naturalnych używających operacji $\{+\}$ oraz stałych będących nieskończonymi ciągami arytmetycznymi koduje klasę zbiorów rekurencyjnych (rekurencyjnie-przeliczalnych, ko-rekurencyjnie przeliczalnych).*

Podobny wynik uzyskamy, jeśli ograniczymy ilość dopuszczalnych zmiennych do jednej.

Wynik 23 ([D7])**.** *Klasa rozwiązań jedynych (najmniejszych, największych) układów równań zbiorów liczb naturalnych używających operacji $\{+, \cup\}$ lub $\{+, \cap\}$ i dokładnie jednej zmiennej koduje klasę zbiorów rekurencyjnych (rekurencyjnie-przeliczalnych, ko-rekurencyjnie przeliczalnych).*

3.3. **Operacje nieciągłe: odejmowanie.** Choć oryginalna motywacja badania układów równań nad liczbami naturalnymi pochodzi z języków formalnych, rozszerzyliśmy nasze badania również w naturalnych kierunkach, które takiej motywacji już nie posiadają. Konkretnie, rozważamy analogiczne układy, lecz tym razem nad zbiorami liczb całkowitych. Okazuje się, że w pewnym sensie odpowiada to rozważaniu układów w liczbach naturalnych z dodatkową operacją „$-$" odejmowania kompleksowego, zdefiniowaną analogicznie jak suma Minkowsky'ego. Operacja sumy kompleksowej nie jest ciągła w przypadku zbiorów liczb całkowitych[6] (analogicznie: operacja odejmowania kompleksowego nie jest ciągła w przypadku

[6] Podkreślmy, że nie chodzi o ciągłość odejmowania liczb, lecz o ciągłość odejmowania *zbiorów* liczb.

zbiorów liczb naturalnych), tym samym znane ograniczenia górne nie przenoszą się na ten przypadek.

Trywialne ograniczenie dla rozwiązań jedynych to klasa zbiorów hiper-arytmetycznych, czyli przecięcie zbiorów z klasy Π^1_1 i Σ^1_1 z hierarchii analitycznej: zdanie „wektor zbiorów spełnia podane równanie" można wyrazić w języku arytmetyki rozszerzonym o predykat $x \in X$, nazwijmy je Eq. Wtedy największe i najmniejsze rozwiązanie spełniają formuły

$$\varphi(x) = \exists X_1 \ldots \exists X_n : \ Eq(X_1, \ldots, X_n) \wedge x \in X_1 \text{ oraz}$$
$$\varphi'(x) = \forall X_1 \ldots \forall X_n, \ Eq(X_1, \ldots, X_n) \to x \in X_1 \ .$$

Okazuje się, że to trywialne ograniczenie górne jest osiągalne.

Wynik 24 ([E5])**.** *Klasa zbiorów reprezentowanych jako jedyne rozwiązanie układów równań nad liczbami naturalnymi i operacjami* $\{+, -, \cup\}$ *to dokładnie klasa zbiorów hiper-arytmetycznych.*

Taki sam fakt zachodzi dla zbiorów liczb całkowitych oraz operacji $\{+, \cup\}$*.*

W przypadku rozwiązań najmniejszych dla układów w postaci rozwiązanej, wciąż pozostajemy w klasie zbiorów rekurencyjnie przeliczalnych, gdyż wprowadzone nieciągłe operacje wciąż są monotoniczne. Z drugiej strony, rozwiązanie największe pokrywa się z trywialnym ograniczeniem Σ^1_1.

Wynik 25 ([E7])**.** *Klasa zbiorów reprezentowanych jako największe (najmniejsze) rozwiązanie układów równań w postaci rozwiązanej nad liczbami naturalnymi i operacjami* $\{+, -, \cup, \cap\}$ *to dokładnie klasa zbiorów* Σ^1_1 *w hierarchii zbiorów analitycznych (odpowiednio: rekurencyjnie-przeliczalnych).*

Taki sam fakt zachodzi dla zbiorów liczb całkowitych oraz operacji $\{+, \cup, \cap\}$*.*

4. Minimalizacja z błędami dla automatów skończonych

Ze względu na swoją prostotę i łatwość w stosowaniu, automaty skończone zajmują istotne miejsce w informatyce, zarówno teoretycznej jak i stosowanej. W tym rozdziale zajmować się będziemy jedynie automatami deterministycznymi, dlatego też *automat* będzie oznaczało deterministyczny automat skończony (DFA). Dla przypomnienia, automat jest zadany przez skończony alfabet Σ, zbiór stanów Q, stan początkowy $q_0 \in Q$, podzbiór stanów akceptujących $F \subseteq Q$ oraz funkcję przejścia $\delta : Q \times \Sigma \mapsto Q$. Automat akceptuje słowa, które przeprowadzają ze stanu początkowego do jednego ze stanów akceptujących.

W oczywisty sposób, gdy dwa automaty rozpoznają ten sam język, uznajemy je za *równoważne* i zwykle jesteśmy zainteresowani konstrukcją *minimalnego* automatu równoważnego wejściowemu. Tak zdefiniowany problem minimalizacji jest jednym z najważniejszych zadań w teorii automatów. Najszybszy asymptotycznie algorytm, o czasie działania $\mathcal{O}(|\Sigma| n \log n)$, jest autorstwa J. Hopcrofta [47]. Pytanie, czy możliwy jest szybszy algorytm dla tego problemy, pozostaje od lat otwarte. W niektórych przypadkach wiadomo jednak, że minimalizacja może zostać przeprowadzona szybciej: np. gdy automat jest acykliczny, minimalizację można przeprowadzić w liniowym czasie.

W niektórych zastosowaniach zakłada się, że funkcja przejścia automatu jest częściowa, tzn. nie wszystkie przejścia są zdefiniowane; jest to istotne o tyle, że wiele „prawdziwych" automatów istotnie ma dużo niezdefiniowanych przejść. Istnieją warianty algorytmu Hopcrofta biorące pod uwagę taką możliwość, ich czas działania zależy od $|\delta|$ (rozmiaru funkcji przejścia), a nie od $n|\Sigma|$, tj. wynosi $\mathcal{O}(|\delta| \log n)$.

Automaty minimalne wciąż mogą być bardzo duże i ich dalsze zmniejszenie jest pożądane. W wielu zastosowaniach możemy zadowolić się pewnym przybliżeniem oryginalnego automatu: dopuszcza się odstępstwa na skończeniu wielu słowach, na konkretnej liczbie słów, na słowach o określonej długości itp. Poniżej przedstawię uzyskane przeze mnie wyniki dla tego typu problemów.

4.1. **Hiper-minimalizacja i k-minimalizacja.** W problemie hiper-minimalizacji zajmujemy się problemem „granicznego" zachowania automatu, tj. jesteśmy gotowi dopuścić skończoną ilość błędów: dwa automaty są *hiper-równoważne*, gdy ich języki różnią się na jedynie skończenie wielu słowach [4]; formalnie, dwa automaty M i N są hiper-równoważne, gdy $|L(N)\triangle L(M)| < \infty$, gdzie „$\triangle$" oznacza różnicę symetryczną zbiorów. W problemie *hiper-minimalizacji* dla danego na wejściu automatu N chcemy skonstruować najmniejszy hiper-równoważny mu automat. Poprzednio znany był dla tego problemu algorytm o kwadratowym czasie działania [4].

Rozszerzyliśmy to pojęcie, tak aby dozwolone były błędy na słowach o długości nie przekraczającej k: dwa automaty są *k-równoważne*, gdy ich języki różnią się na słowach o długości mniejszej niż k, tj. $L(N)\triangle L(M) \subseteq \Sigma^{<k}$. Łatwo pokazać, że dwa automaty (o n i m stanach) są hiper-równoważne, jeśli są $\max(n,m)$-równoważne. Tym samym problem hiper-minimalizacji sprowadza się do problemu k-minimalizacji.

Podaliśmy algorytm k-minimalizacyjny o czasie działania $\mathcal{O}(|\delta|\log^2 n)$, gdzie $|\delta|$ jest rozmiarem funkcji przejścia.

Wynik 26 ([E4])**.** *k-minimalizacja może zostać przeprowadzona w czasie $\mathcal{O}(|\delta|\log^2 n)$, gdzie $|\delta|$ jest rozmiarem funkcji przejścia, a n ilością stanów automatu wejściowego.*

Algorytm ten jest oparty na iteracyjnym scalaniu stanów „k-odpowiadających" (tzn. takich, które w pewnym sensie rozpoznają k-równoważne języki). Należy zaznaczyć, że odpowiednia definicja k-odpowiedniości jest trudna a dowód poprawności algorytmu jest skomplikowany i nieoczywisty. Naiwna analiza daje kwadratowy czas działania (tak jak w poprzednim algorytmie hiper-minimalizacyjnym [4]), jednak odpowiednie struktury danych i analiza pozwalają na ograniczenie czasu działania do $\mathcal{O}(|\delta|\log^2 n)$.

Dalsze prace nad tym problemem prowadziliśmy w dwóch kierunkach:

- rozszerzenie definicji k-równoważności: dopuszczenie tylko ograniczonej liczby błędów między automatami itp.;
- uproszczenie algorytmu i zmniejszenie zawiłości dowodu.

Pierwszy cel okazał się trudny do osiągnięcia: badane rozszerzenia okazały się być NP-trudne.

Wynik 27 ([I1])**.** *Hiper-minimalizacja z ograniczoną liczbą błędów jest* NP*-trudna: dla danego na wejściu automatu M oraz liczb m, s pytanie, czy istnieje automat N o najwyżej s stanach oraz taki że $|L(M)\triangle L(N)| \leq m$ jest* NP*-trudne.*

k-minimalizacja z ograniczoną liczbą błędów jest NP*-trudna dla danego na wejściu automatu M oraz liczb k, m pytanie, czy istnieje automat k-minimalny N taki że $|L(M)\triangle L(N)| \leq m$ jest* NP*-trudne.*

Tym niemniej, udało się uzyskać spory postęp w pierwszym problemie: zdefiniowana wcześniej k-odpowiedniość stanów została przeformułowana w terminach odległości między stanami (odległość to długość najdłuższego słowa, na którym stany się różnią). Nowy algorytm k-minimalizujący działa w dwóch fazach: w pierwszej oblicza odległości między stanami, w

drugiej wykorzystuje drzewo odległości do wyznaczenia stanów, które należy skleić w automacie k-minimalnym; drugą fazę można łatwo wykonać przy przeglądaniu drzewa odległości od liści do korzenia, w szczególności można to zrobić w liniowym czasie. Zauważmy, że pierwsza faza jest niezależna od k, dzięki temu w czasie $\mathcal{O}(|\delta| \log^2 n)$ umiemy wyznaczyć rozmiary automatów k-minimalnych dla wszystkich możliwych k jak również iteracyjnie tworzyć automaty k-minimalne dla kolejnych wartości $k = 0$ (automat równoważny), 1, 2, ..., n (automat hiper-równoważny).

Jak wiadomo, minimalizacja automatu acyklicznego jest możliwa w liniowym czasie, tj. szybciej niż w ogólnym przypadku. Uzyskaliśmy podobny wynik również w przypadku k-minimalizacji: k-minimalizacja dla automatów acyklicznych jest możliwa w czasie $\mathcal{O}(|\delta| \log n)$.

Wynik 28 ([I1])**.** *W czasie $\mathcal{O}(|\delta| \log^2 n)$ można podać wielkości k-minimalnych automatów dla wszystkich możliwych wartości k, gdzie $|\delta|$ jest rozmiarem funkcji przejścia a n ilością stanów automatu. Algorytm generujący te liczby w k-tej fazie pamięta automat k-minimalny. W przypadku automatów acyklicznych, algorytm działa w czasie $\mathcal{O}(|\delta| \log n)$.*

4.2. **Hiper-minimalizacja dla termów.** Wariant automatów skończonych jest też oczywiście zdefiniowany dla drzew[7] i wiele spośród definicji i algorytmów znanych dla automatów na słowach przenosi się do przypadku drzewowego. W szczególności, znane są uogólnienia algorytmu Hopcrofta do tego przypadku [37], działające w czasie $\mathcal{O}(|\delta| \log n)$, gdzie $|\delta|$ jest rozmiarem funkcji przejścia, a n ilością stanów automatu.

Pojęcie hiper-równoważności w naturalny sposób uogólnia się do przypadku automatów na drzewach. Co więcej, możliwa jest liniowa redukcja, która sprowadza problem konstrukcji hiper-minimalnego automatu drzewowego do konstrukcji hiper-minimalnego automatu dla słów.

Wynik 29 ([I4])**.** *Problem hiper-minimalizacji automatów drzewowych można zredukować do problemu hiper-minimalizacji autmatów w czasie $\mathcal{O}(|\delta|)$, gdzie $|\delta|$ jest rozmiarem funkcji przejścia. W szczególności, cały problem można rozwiązać w czasie $\mathcal{O}(|\delta| \log^2 n)$, gdzie n jest ilością stanów automatu wejściowego.*

4.3. **Automaty pokrywające.** W wielu zastosowaniach język L automatu skończonego jest skończony. Tym samym, sprawdzanie przynależności słowa w do języka można wykonać w następujący sposób:

(1) sprawdzić, czy w jest dostatecznie krótkie,
(2) sprawdzić, czy w należy do języka.

Łatwo zauważyć, że automat użyty w drugim kroku może rozpoznawać inny język, o ile tylko zgadza się z automatem wejściowym na dostatecznie krótkich słowach.

Ta obserwacja doprowadziła do definicji *automatu k-pokrywającego*: dla automatu N oraz liczby k automat M jest k-pokrywający, jeśli N oraz M rozpoznają te same słowa długości nie większej niż k, formalnie $L(M) \cap \Sigma^{\leq k} = L(N) \cap \Sigma^{\leq k}$ [10]. Odpowiadający problem minimalizacyjny pyta o konstrukcję minimalnego automatu pokrywającego (dla zadanego na wejściu automatu N). Zauważmy, że podejście to jest w pewien sposób dualne do problemu k-minimalizacji (należy jednak zaznaczyć, że pojawiło się wcześniej).

Najlepszy znany algorytm konstruujący minimalny automat pokrywający działa w czasie $\mathcal{O}(|\Sigma| n \log n)$ [59], uogólnia on algorytm Hopcrofta, a jego działania istotnie zależy od wartości k. Okazuje się, że podejście podobne do zastosowanego w przypadku k-minimalizacji

[7]Tak jak poprzednio, przez drzewa rozumiemy tu w istocie termy.

można wykorzystać również w tym przypadku. Konstrukcję minimalnego automatu dzielimy na dwie fazy: w pierwszej budujemy drzewo odległości między stanami, gdzie odległość to długość najmniejszego słowa, na którym te dwa stany się różnią (drzewo to jest *implicite* konstruowane w czasie algorytmu Hopcrofta), w drugiej fazie używamy drzewa do stwierdzenia, które stany w automacie powinny być scalone w celu uzyskania automatu k-pokrywającego. W szczególności, nasz algorytm może działać iteracyjnie dla kolejnych wartości k, generując coraz mniejsze automaty (czas działania to wciąż $\mathcal{O}(|\Sigma|n \log n)$. Algorytm ten jest prostszy i ogólniejszy, niż poprzednio znany.

Wynik 30 ([I2])**.** *W czasie $\mathcal{O}(|\Sigma|n \log n)$ można podać wielkości minimalnych automatów k-pokrywających dla wszystkich wartości k. Algorytm generujący te liczby w k-tej fazie pamięta automat k-pokrywający.*

5. Inne

Wymienione wcześniej prace opisały moje główne zainteresowania i osiągnięcia naukowe. W tym rozdziale zostaną przedstawione tematy, które nie lokują się w centrum moich zainteresowań, tym niemniej udało mi się uzyskać w nich pewne wyniki.

5.1. **Algorytmy online.** Klasyczna analiza algorytmów (pod kątem ich czasu działania), nie jest przydatna, gdy główna trudność algorytmiczna wynika z nieznajmości przyszłości. Analiza konkurencyjna algorytmów online jest nowoczesną odpowiedzią na tego typu problemy: algorytm uzyskuje informację o instancji element po elemencie i musi za każdym razem udzielić odpowiedzi, co więcej, odpowiedzi te muszą być „spójne" z poprzednio udzielonymi. Aby ocenić jakość algorytmu, porównujemy go z najlepszym możliwym rozwiązaniem *dla danej instancji*. Współczynnik konkurencyjności to maksimum tych liczb (po wszystkich możliwych instancjach).

5.1.1. *Szukanie prostej na płaszczyźnie.* Jednym z pierwszych problemów (a zwykle pierwszym), który poznaje się w czasie wykładu z algorytmów online [8], jest tzw. problem szukania krowy: w tym problemie jesteśmy ustawieni w punkcie 0 osi liczbowej i wiemy, iż w pewnym miejscu na osi znajduje się cel, który pragniemy odnaleźć [5]. Koszt strategii to odległość, którą musimy przejść, zanim odnajdziemy cel. Istnieje wiele wariantów tego problemu, większość z nich została dokładnie zbadana już w oryginalnej pracy lub wkrótce po.

Jednym ze słabiej zbadanych był wariant poszukiwania na płaszczyźnie prostej, która jest równoległa do którejś z osi układu współrzędnych. Pokazaliśmy strategię, która ma współczynnik konkurencyjności $12,54\ldots$, poprawiając poprzedni wynik $9\sqrt{2} = 13,02\ldots$ [8]. Najistotniejszą cechą naszej strategii jest jednak to, że nie jest ona *samopodobna*, pomimo tego, iż wcześniej podawano heurystyczne argumenty, które miały pokazywać, że najlepsza strategia jest w istocie samopodobna [8].

Wynik 31 ([E3])**.** *Problem poszukiwania na płaszczyźnie linii równoległej do którejś z osi układu współrzędnych ma współczynnik konkurencyjności nie gorszy niż $12,54\ldots$.*

5.1.2. *Szeregowanie zadań o nieznanych czasach pojawienia i wygasania.* Jednym z klasycznych problemów online jest szeregowania zadań o nieznanych czasach pojawienia. Każde z zadań ma *czas przybycia*, *czas wygasania* oraz *wartość*, wszystkie te informacje są ujawnione w czasie przybycia a wartość zostaje wypłacona, jeśli faktycznie zostanie ukończone przed terminem wygaśnięcia. W literaturze można znaleźć wiele algorytmów dla tego problemu, istotnie używają one informacji o terminach wygasania [3, 14, 15, 27, 45, 54, 55, 70, 71].

Zajęliśmy się ogólniejszym problem, w którym dokładne terminy wygasania również nie są znane, znana jest jedynie kolejność wygasania. Model taki ma praktyczną motywację: w przypadku awarii sieci lub pojawienia się transmisji o większym priorytecie może się zdarzyć, że łącze nie będzie dostępne. Udało nam się uzyskać kilka ciekawych wyników dotyczących tego problemu — niektóre spośród algorytmów dla zadań o znanych terminach wygasania da się zaadaptować także do tego, bardziej ogólnego modelu. Ponadto w paru bardziej ograniczonych wersjach tego problemu podaliśmy nowe algorytmy.

Wynik 32 ([E2], [I6])**.** *Istnieje algorytm* $1,897$*-konkurencyjny dla problemu szeregowania pakietów o nieznanych czasach przybycia i wygasania. Problem ten ma ograniczenie dolne* $1,632$*. W przypadku kolejek FIFO ulepszony algorytm ma współczynnik* $1,737$*. Wariant algorytmu dla przypadku monotonicznego, tj. takiego, w którym wartość przedmiotów rośnie z czasem, ma współczynnik konkurencyjności* $1,618$*, jest to też ograniczenie dolne.*

5.2. **Grafy i grupy permutacji.** Istnieje wiele równoległych spojrzeń na grupy. W jednym z nich nie myślimy o grupach abstrakcyjnych, lecz o konkretnych grupach permutacji, tj. każdy element grupy jest permutacją pewnego obiektu kombinatorycznego. W szczególności poszukuje się klas obiektów, których grupy automorfizmów dają wszystkie grupy permutacji. W najlepszym przypadku, dla danej grupy konstrukcja odpowiedniego obiektu powinna być prosta a sam obiekt mały.

Wiadomo, że grupy symetrii grafów reprezentują jedynie podklasę wszystkich grup symetrii. W *supergrafach* krawędzie pogrupowane są w warstwy, warstwę 0 stanowią wierzchołki, natomiast krawędzie z i-tej warstwy mogą łączyć dowolne elementy z niższych warstw. Wszystkie grupy są reprezentowane jako grupy automorfizmów supergrafów. Co więcej, jeśli grupa permutacji G jest produktem (sumą prostą, splotem) grup H_1 i H_2, to znając grafy reprezentujące H_1 i H_2 możemy prosto podać graf reprezentujący G.

Wynik 33 ([E1])**.** *Każda grupa permutacji jest reprezentowana jako grupa automorfizmów supergrafu. Jeśli grupa* G *jest produktem (sumą prostą, splotem) grup* H_1 *oraz* H_2*, to graf reprezentujący* G *można skonstruować z grup reprezentujących* H_1 *oraz* H_2*.*

5.3. **Automaty skracające.** Automaty skracające z dwoma stosami pojawiły się jako rozszerzenie automatów ze stosem. Jednocześnie automat z dwoma stosami to w istocie maszyna Turinga, konwencja ta jest szeroko stosowana i dwa stosy są zwykle identyfikowane z taśmą, tj. jeden stos ze znakami na lewo a drugi ze znakami na prawo od głowicy. Dodatkowe ograniczenie, iż wszystkie reguły muszą być skracające, pozwala na zdefiniowanie sensownej i naturalnej klasy: rosnących języków kontekstowych [9] (zawiera ona w szczególności języki bezkontekstowe).

W badaniach automatów skracających zwykle zakłada się, że mogą one używać dowolnej ilości dodatkowych symboli taśmowych (to znaczy liter nie będących elementami wejściowego alfabetu). W naszych badaniach zajęliśmy się przypadkiem, w którym ilość dodatkowych symboli jest ograniczone. Nawet gdy taśmowy jest jednoliterowy i nie może używać dodatkowych symboli, wciąż jest w stanie symulować maszynę Minsky'ego i tym samym niepustość dla takich automatów jest nierozstrzygalna. Automaty nie używające dodatkowych symboli taśmowych są w stanie rozpoznać wszystkie deterministyczne języki bezkontekstowe. Jeśli pozwolimy na użycie jednego symbolu dodatkowego, automaty takie mogą rozpoznać wszystkie języki bezkontekstowe.

Wynik 34 ([I3])**.** *Problem niepustości dla skracających automatów z dwoma stosami nad alfabetem unarnym jest nierozstrzygalny.*

Automaty z dwoma stosami nie używające dodatkowych symboli taśmowych rozpoznają wszystkie deterministyczne języki bezkontekstowe. Jeśli pozwolimy, aby używały one jednego dodatkowego symbolu, są w stanie rozpoznać każdy język bezkontekstowy.

5.4. **Weryfikacja tablicy KMP.** Jednym z nowszych trendów w algorytmach tekstowych jest *weryfikacja* rozmaitych struktur danych: dla zadanej na wejściu struktury chcemy sprawdzić, czy jest ona poprawna, tzn. odpowiada ona strukturze policzonej dla konkretnego słowa. Tego typu problemy zostały sprawdzone dla wielu typu struktur danych: dla tablicy π algorytmu MP [29, 26], parametryzowanej tablicy π [48, 49], tablicy prefiksowej [16], oraz cover array [21].

Pokazaliśmy, że taką weryfikację dla tablicy π' z algorytmu KMP można przeprowadzić w czasie liniowym, nawet jeśli wymagamy, by algorytm udzielał odpowiedzi po przeczytaniu każdej liczby z wejścia. Ponadto pokazaliśmy, iż możemy policzyć rozmiar minimalnego alfabetu, nad którym takie słowo istnieje oraz podać takie słowo, nie zwiększając złożoności czasowej.

Wynik 35 ([E6])**.** *Weryfikacja tablicy π' algorytmu KMP może być przeprowadzona online w liniowym czasie. Algorytm zwraca takie słowo nad najmniejszym możliwym alfabetem.*

Literatura

[1] Anisa Al-Hafeedh, Maxime Crochemore, Lucian Ilie, Evguenia Kopylova, William F. Smyth, German Tischler, and Munina Yusufu. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. *ACM Comput. Surv.*, 45(1):5, 2012.

[2] Stephen Alstrup, Gerth S. Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *SODA*, pages 819–828, 2000.

[3] Nir Andelman, Yishay Mansour, and An Zhu. Competitive queueing policies in QoS switches. In *Proceedings of the 14th Symposium on Discrete Algorithms (SODA)*, pages 761–770. ACM/SIAM, 2003.

[4] Andrew Badr, Viliam Geffert, and Ian Shipman. Hyper-minimizing minimized deterministic finite state automata. *RAIRO Theoret. Inform. Appl.*, 43(1):69–94, 2009.

[5] R.A. Baeza-Yates, J.C. Culberson, and G.J.E. Rawlins. Searching with uncertainty. *Research report*, 1987.

[6] Julius Richard Büchi and Steven Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. *Mathematical Logic Quarterly*, 34(4):337–342, 1988.

[7] Martin Beaudry, Pierre McKenzie, Pierre Péladeau, and Denis Thérien. Finite monoids: From word to circuit evaluation. *SIAM J. Comput.*, 26(1):138–152, 1997.

[8] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[9] Gerhard Buntrock and Friedrich Otto. Growing context-sensitive languages and Church-Rosser languages. *Inf. Comput.*, 141(1):1–36, 1998.

[10] Cezar Câmpeanu, Nicolae Santean, and Sheng Yu. Minimal cover-automata for finite languages. *Theor. Comput. Sci.*, 267(1–2):3–16, 2001.

[11] Witold Charatonik and Leszek Pacholski. Word equations with two variables. In Habib Abdulrab and Jean-Pierre Pécuchet, editors, *IWWERT*, volume 677 of *LNCS*, pages 43–56. Springer, 1991.

[12] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

[13] Gang Chen, Simon J. Puglisi, and William F. Smyth. Fast and practical algorithms for computing all the runs in a string. In Bin Ma and Kaizhong Zhang, editors, *CPM*, volume 4580 of *LNCS*, pages 307–315. Springer, 2007.

[14] Francis Y. L. Chin, Marek Chrobak, Stanley P. Y. Fung, Wojciech Jawor, Jiří Sgall, and Tomáš Tichý. Online competitive algorithms for maximizing weighted throughput of unit jobs. *Journal of Discrete Algorithms*, 4:255–276, 2006.

[15] Francis Y. L Chin and Stanley P. Y. Fung. Online scheduling for partial job values: Does timesharing or randomization help? *Algorithmica*, 37:149–164, 2003.

[16] Julien Clément, Maxime Crochemore, and Giuseppina Rindone. Reverse engineering prefix tables. In *Proceedings of 26th STACS*, pages 289–300, 2009.

[17] Hubert Comon. Completion of rewrite systems with membership constraints. Part I: Deduction rules. *J. Symb. Comput.*, 25(4):397–419, 1998.

[18] Hubert Comon. Completion of rewrite systems with membership constraints. Part II: Constraint solving. *J. Symb. Comput.*, 25(4):421–453, 1998.

[19] Carles Creus, Adria Gascon, and Guillem Godoy. One-context Unification with STG-Compressed Terms is in NP. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*, volume 15 of *LIPIcs*, pages 149–164, Dagstuhl, Germany, 2012. Schloss Dagstuhl — Leibniz Zentrum fuer Informatik.

[20] Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *DCC*, pages 482–488. IEEE Computer Society, 2008.

[21] Maxime Crochemore, Costas Iliopoulos, Solon Pissis, and German Tischler. Cover array string reconstruction. In *CPM 2010, LNCS, to appear*. Springer, 2010.

[22] Volker Diekert. Makanin's algorithm. In M. Lothaire, editor, *Algebraic Combinatorics on Words*, chapter 12, pages 342–390. Cambridge University Press, 2002.

[23] Volker Diekert, Claudio Gutiérrez, and Christian Hagenah. The existential theory of equations with rational constraints in free groups is PSPACE-complete. *Inf. Comput.*, 202(2):105–140, 2005.

[24] Robert Dąbrowski and Wojciech Plandowski. Solving two-variable word equations. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *ICALP*, volume 3142 of *LNCS*, pages 408–419. Springer, 2004.

[25] Robert Dąbrowski and Wojciech Plandowski. On word equations in one variable. *Algorithmica*, 60(4):819–828, 2011.

[26] Jean-Pierre Duval, Thierry Lecroq, and Arnaud Lefebvre. Efficient validation and construction of border arrays and validation of string matching automata. *ITA*, 43(2):281–297, 2009.

[27] Matthias Englert and Matthias Westermann. Considering suppressed packets improves buffer management in QoS switches. In *Proceedings of the 18th Symposium on Discrete Algorithms (SODA)*, pages 209–218. ACM/SIAM, 2007.

[28] William M. Farmer. Simple second-order languages for which unification is undecidable. *Theor. Comput. Sci.*, 87(1):25–41, 1991.

[29] František Franěk, Shudi Gao, Weilin Lu, P. J. Ryan, W. F. Smyth, Yu Sun, and Lu Yang. Verifying a border array in linear time. *J. Comb. Math. Comb. Comput.*, 42:223–236, 2002.

[30] Adria Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Context matching for compressed terms. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 93–102. IEEE Computer Society, 2008.

[31] Adria Gascón, Guillem Godoy, and Manfred Schmidt-Schauß. Unification and matching on compressed terms. *ACM Trans. Comput. Log.*, 12(4):26, 2011.

[32] Adria Gascón, Guillem Godoy, Manfred Schmidt-Schauß, and Ashish Tiwari. Context unification with one context variable. *J. Symb. Comput.*, 45(2):173–193, 2010.

[33] Paweł Gawrychowski. Optimal pattern matching in LZW compressed strings. In Dana Randall, editor, *SODA*, pages 362–372. SIAM, 2011.

[34] Paweł Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In Camil Demetrescu and Magnús M. Halldórsson, editors, *ESA*, volume 6942 of *LNCS*, pages 421–432. Springer, 2011.

[35] Paweł Gawrychowski. Simple and efficient LZW-compressed multiple pattern matching. In Juha Kärkkäinen and Jens Stoye, editors, *CPM*, volume 7354 of *LNCS*, pages 232–242. Springer, 2012.

[36] Paweł Gawrychowski. Tying up the loose ends in fully LZW-compressed pattern matching. In Christoph Dürr and Thomas Wilke, editors, *STACS*, volume 14 of *LIPIcs*, pages 624–635. Schloss Dagstuhl — Leibniz-Zentrum fuer Informatik, 2012.

[37] Ferenc Gécseg and Magnus Steinby. Tree languages. In Grzegorz Rozenberg and Arto Salomaa, editors, *Beyond Words*, Handbook of Formal Languages, chapter 3, pages 1–68. Springer, 1997.

[38] Seymour Ginsburg and H. Gordon Rice. Two families of languages related to ALGOL. *J. ACM*, 9(3):350–371, 1962.

[39] Leszek Gąsieniec, Marek Karpiński, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for Lempel-Ziv encoding. In Rolf G. Karlsson and Andrzej Lingas, editors, *SWAT*, volume 1097 of *LNCS*, pages 392–403. Springer, 1996.

[40] Leszek Gąsieniec, Marek Karpiński, Wojciech Plandowski, and Wojciech Rytter. Randomized efficient algorithms for compressed strings: The finger-print approach. In Daniel S. Hirschberg and Eugene W. Myers, editors, *CPM*, volume 1075 of *LNCS*, pages 39–49. Springer, 1996.

[41] Leszek Gąsieniec and Wojciech Rytter. Almost optimal fully LZW-compressed pattern matching. In *Data Compression Conference*, pages 316–325, 1999.

[42] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theor. Comput. Sci.*, 13:225–230, 1981.

[43] Keisuke Goto and Hideo Bannai. Simpler and faster Lempel Ziv factorization. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *DCC*, pages 133–142. IEEE, 2013.

[44] Claudio Gutiérrez. Satisfiability of word equations with constants is in exponential space. In *FOCS*, pages 112–119. IEEE Computer Society, 1998.

[45] Bruce Hajek. On the competitiveness of online scheduling of unit-length packets with hard deadlines in slotted time. In *Conference on Information Sciences and Systems*, pages 434–438, 2001.

[46] Masahiro Hirao, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Fully compressed pattern matching algorithm for balanced straight-line programs. In *SPIRE*, pages 132–138, 2000.

[47] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi, editor, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[48] Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Counting parameterized border arrays for a binary alphabet. In *Proc. of the 3rd LATA*, pages 422–433, 2009.

[49] Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Verifying and enumerating parameterized border arrays. *Theor. Comput. Sci.*, 412(50):6959–6981, 2011.

[50] Lucian Ilie and Wojciech Plandowski. Two-variable word equations. *ITA*, 34(6):467–501, 2000.

[51] Joxan Jaffar. Minimal and complete word unification. *J. ACM*, 37(1):47–85, 1990.

[52] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In Johannes Fischer and Peter Sanders, editors, *CPM*, volume 7922 of *LNCS*, pages 189–200. Springer, 2013.

[53] Marek Karpiński, Wojciech Rytter, and Ayumi Shinohara. Pattern-matching for strings with short descriptions. In *CPM*, pages 205–214, 1995.

[54] A. Kesselman, Zvi Lotker, Yishay Mansour, Boaz Patt-Shamir, Baruch Schieber, and Maxim Sviridenko. Buffer overflow management in QoS switches. *SIAM Journal of Computing*, 33:563–583, 2004.

[55] Alexander Kesselman, Yishay Mansour, and Rob van Stee. Improved competitive guarantees for QoS buffering. *Algorithmica*, 43:63–80, 2005.

[56] Olga Kharlampovich and Alexei Myasnikov. Irreducible affine varieties over a free group. ii: Systems in triangular quasi-quadratic form and description of residually free groups. *Journal of Algebra*, 200:517—-570, 1998.

[57] Olga Kharlampovich and Alexei Myasnikov. Elementary theory of free non-abelian groups. *Journal of Algebra*, 302:451–552, 2006.

[58] John C. Kieffer and En-Hui Yang. Sequential codes, lossless compression of individual sequences, and Kolmogorov complexity. *IEEE Transactions on Information Theory*, 42(1):29–39, 1996.

[59] Heiko Körner. A time and space efficient algorithm for minimizing cover automata for finite languages. *Int. J. Found. Comput. Sci.*, 14(6):1071–1086, 2003.

[60] Antoni Kościelski and Leszek Pacholski. Complexity of Makanin's algorithm. *J. ACM*, 43(4):670–684, 1996.

[61] Antoni Kościelski and Leszek Pacholski. Makanin's algorithm is not primitive recursive. *Theor. Comput. Sci.*, 191(1-2):145–156, 1998.

[62] Markku Laine and Wojciech Plandowski. Word equations with one unknown. *Int. J. Found. Comput. Sci.*, 22(2):345–375, 2011.

[63] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Data Compression Conference*, pages 296–305. IEEE Computer Society, 1999.

[64] Ernst L. Leiss. Unrestricted complementation in language equations over a one-letter alphabet. *Theoretical Computer Science*, 132(2):71–84, 1994.

[65] Jordi Levy. Linear second-order unification. In Harald Ganzinger, editor, *RTA*, volume 1103 of *LNCS*, pages 332–346. Springer, 1996.
[66] Jordi Levy, Manfred Schmidt-Schauß, and Mateu Villaret. On the complexity of bounded second-order unification and stratified context unification. *Logic Journal of the IGPL*, 19(6):763–789, 2011.
[67] Jordi Levy and Margus Veanes. On the undecidability of second-order unification. *Inf. Comput.*, 159(1–2):125–150, 2000.
[68] Jordi Levy and Mateu Villaret. Linear second-order unification and context unification with tree-regular constraints. In Leo Bachmair, editor, *RTA*, volume 1833 of *LNCS*, pages 156–171. Springer, 2000.
[69] Jordi Levy and Mateu Villaret. Currying second-order unification problems. In Sophie Tison, editor, *RTA*, volume 2378 of *LNCS*, pages 326–339. Springer, 2002.
[70] Fei Li, Jay Sethuraman, and Clifford Stein. An optimal online algorithm for packet scheduling with agreeable deadlines. In *Proceedings of the 16th Symposium on Discrete Algorithms (SODA)*, pages 801–802. ACM/SIAM, 2005.
[71] Fei Li, Jay Sethuraman, and Clifford Stein. Better online buffer management. In *Proceedings of the 18th Symposium on Discrete Algorithms (SODA)*, pages 199–208. ACM/SIAM, 2007.
[72] Yury Lifshits. Processing compressed texts: A tractability border. In Bin Ma and Kaizhong Zhang, editors, *CPM*, volume 4580 of *LNCS*, pages 228–240. Springer, 2007.
[73] Markus Lohrey. Word problems and membership problems on compressed words. *SIAM Journal of Computing*, 35(5):1210–1240, 2006.
[74] Markus Lohrey. Compressed membership problems for regular expressions and hierarchical automata. *International Journal of Foundations of Computer Science*, 21(5):817–841, 2010.
[75] Markus Lohrey. Leaf languages and string compression. *Inf. Comput.*, 209(6):951–965, 2011.
[76] Markus Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.
[77] Markus Lohrey, Sebastian Maneth, and Manfred Schmidt-Schauß. Parameter reduction and automata evaluation for grammar-compressed trees. *J. Comput. Syst. Sci.*, 78(5):1651–1669, 2012.
[78] Markus Lohrey and Christian Mathissen. Compressed membership in automata with compressed labels. In Alexander S. Kulikov and Nikolay K. Vereshchagin, editors, *CSR*, volume 6651 of *LNCS*, pages 275–288. Springer, 2011.
[79] Gennadiĭ Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 2(103):147–236, 1977. (in Russian).
[80] Gennadiĭ Makanin. Equations in a free group. *Izv. Akad. Nauk SSR*, Ser. Math. 46:1199–1273, 1983. English transl. in Math. USSR Izv. 21 (1983).
[81] Gennadiĭ Semyonovich Makanin. Decidability of the universal and positive theories of a free group. *Izv. Akad. Nauk SSSR*, Ser. Mat. 48:735–749, 1984. In Russian; English translation in: *Math. USSR Izvestija, 25*, 75–88, 1985.
[82] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.
[83] Masamichi Miyazaki, Ayumi Shinohara, and Masayuki Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In *CPM*, volume 1264 of *LNCS*, pages 1–11. Springer, 1997.
[84] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical strcture in sequences: A linear-time algorithm. *J. Artif. Intell. Res. (JAIR)*, 7:67–82, 1997.
[85] Joachim Niehren, Manfred Pinkal, and Peter Ruhrberg. On equality up-to constraints over finite trees, context unification, and one-step rewriting. In William McCune, editor, *CADE*, volume 1249 of *LNCS*, pages 34–48. Springer, 1997.
[86] Joachim Niehren, Manfred Pinkal, and Peter Ruhrberg. A uniform approach to underspecification and parallelism. In Philip R. Cohen and Wolfgang Wahlster, editors, *ACL*, pages 410–417. Morgan Kaufmann Publishers / ACL, 1997.
[87] S. Eyono Obono, Pavel Goralcik, and M. N. Maksimenko. Efficient solving of the word equations in one variable. In Igor Prívara, Branislav Rovan, and Peter Ruzicka, editors, *MFCS*, volume 841 of *LNCS*, pages 336–341. Springer, 1994.
[88] Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In Raffaele Giancarlo and Giovanni Manzini, editors, *CPM*, volume 6661 of *LNCS*, pages 15–26. Springer, 2011.
[89] Alexander Okhotin. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535, 2001.

[90] Alexander Okhotin. A recognition and parsing algorithm for arbitrary conjunctive grammars. *Theor. Comput. Sci.*, 302(1–3):365–399, 2003.
[91] Alexander Okhotin. Unresolved systems of language equations: Expressive power and decision problems. *Theor. Comput. Sci.*, 349(3):283–308, 2005.
[92] Alexander Okhotin. Nine open problems on conjunctive and Boolean grammars. *Bulletin of the EATCS*, 91:96–119, 2007.
[93] Alexander Okhotin. Unambiguous Boolean grammars. *Inf. Comput.*, 206(9–10):1234–1247, 2008.
[94] Alexander Okhotin. Decision problems for language equations. *J. Comput. Syst. Sci.*, 76(3–4):251–266, 2010.
[95] Alexander Okhotin. Fast parsing for Boolean grammars: A generalization of Valiant's algorithm. In Yuan Gao, Hanlin Lu, Shinnosuke Seki, and Sheng Yu, editors, *Developments in Language Theory*, volume 6224 of *Lecture Notes in Computer Science*, pages 340–351. Springer, 2010.
[96] Jan Otop. korespondencja prywatna, 2012.
[97] Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In Jan van Leeuwen, editor, *ESA*, volume 855 of *LNCS*, pages 460–470. Springer, 1994.
[98] Wojciech Plandowski. Satisfiability of word equations with constants is in NEXPTIME. In *STOC*, pages 721–725, 1999.
[99] Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004.
[100] Wojciech Plandowski. An efficient algorithm for solving word equations. In Jon M. Kleinberg, editor, *STOC*, pages 467–476. ACM, 2006.
[101] Wojciech Plandowski and Wojciech Rytter. Application of Lempel-Ziv encodings to the solution of word equations. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *LNCS*, pages 731–742. Springer, 1998.
[102] Wojciech Plandowski and Wojciech Rytter. Complexity of language recognition problems for compressed words. In Juhani Karhumäki, Hermann A. Maurer, Gheorghe Paun, and Grzegorz Rozenberg, editors, *Jewels are Forever*, pages 262–272. Springer, 1999.
[103] Alexander A. Razborov. *On Systems of Equations in Free Groups*. PhD thesis, Steklov Institute of Mathematics, 1987. In Russian.
[104] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
[105] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.
[106] Aleksi Saarela. On the complexity of Hmelevskii's theorem and satisfiability of three unknown equations. In Volker Diekert and Dirk Nowotka, editors, *Developments in Language Theory*, volume 5583 of *LNCS*, pages 443–453. Springer, 2009.
[107] Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2-4):416–430, 2005.
[108] Manfred Schmidt-Schauß. Unification of stratified second-order terms. Internal Report 12/94, Johann-Wolfgang-Goethe-Universität, 1994.
[109] Manfred Schmidt-Schauß. A decision algorithm for stratified context unification. *J. Log. Comput.*, 12(6):929–953, 2002.
[110] Manfred Schmidt-Schauß. Decidability of bounded second order unification. *Inf. Comput.*, 188(2):143–178, 2004.
[111] Manfred Schmidt-Schauß and Klaus U. Schulz. On the exponent of periodicity of minimal solutions of context equation. In *RTA*, volume 1379 of *LNCS*, pages 61–75. Springer, 1998.
[112] Manfred Schmidt-Schauß and Klaus U. Schulz. Solvability of context equations with two context variables is decidable. *J. Symb. Comput.*, 33(1):77–122, 2002.
[113] Klaus U. Schulz. Makanin's algorithm for word equations—two improvements and a generalization. In Klaus U. Schulz, editor, *IWWERT*, volume 572 of *LNCS*, pages 85–150. Springer, 1990.
[114] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong, editors, *STOC*, pages 1–9. ACM, 1973.

[115] James A. Storer and Thomas G. Szymanski. The macro model for data compression. In Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho, editors, *STOC*, pages 30–39. ACM, 1978.

[116] Alfred Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.